The Cyc

MICROpendium

The Art Of Assembly Series

Compiled by Mike Wright
Version 19990625

Table of Contents

1. The Art Of Assembly Series 12
1.1. The Art Of Assembly — Part 1 12
1.2. The Art Of Assembly — Part 2. Starting At The Bottom 16
1.3. The Art Of Assembly — Part 3. Starting At The Top 21
1.4. The Art Of Assembly — Part 4. Memory Saving Tips 28
1.5. The Art Of Assembly — Part 5. Useful Subroutines 34
1.6. The Art Of Assembly — Part 6. The Ins And Outs 44
1.7. The Art Of Assembly — Part 7. Why A Duck? 51
1.8. The Art Of Assembly — Part 8. File Handling Tips 62
1.9. The Art Of Assembly — Part 9. More File Handling Tips 67
1.10. The Art Of Assembly — Part 10. Off The End Of The World 78
1.11. The Art Of Assembly — Part 11. Structuring Data 85
1.12. The Art Of Assembly — Part 12. Getting The Most From VDP RAM 92
1.13. The Art Of Assembly — Part 13. Randomly Speaking 95
1.14. The Art Of Assembly — Part 14. Crossing The Bridge To Option 5 100
1.15. The Art Of Assembly — Part 15. Compatibility With The Geneve 107
1.16. The Art Of Assembly — Part 16. Maximizing Speed Of Execution 113
1.17. The Art Of Assembly — Part 17. Structure Can Be Good — But 122
1.18. The Art Of Assembly — Part 18. Whoa! Slow Down! Hold On A Sec! 127
1.19. The Art Of Assembly — Part 19. Sticks And Sprites May Break My. 134
1.20. The Art Of Assembly — Part 20. The Sounds Of The TI 145
1.20.1. The Sound List Method 145
1.20.2. Direct To The Generator 147
1.20.3. More Exotic Methods 148
1.21. The Art Of Assembly — Part 21. Sound Trade Secrets 152
1.22. The Art Of Assembly — Part 22. The Business End 162
1.22.1. The Rules 162
1.22.2. Think Big 162
1.22.3. Lucky Guesses 163
1.22.4. Think Small 163
1.22.5. Ship The Product 163
1.22.6. Don't Quit Your Day Job 164
1.22.7. People Are Funny 165
1.22.8. What Do Those Tea Leaves Mean? 166
1.22.9. What To Do 166
1.23. The Art Of Assembly — Part 23. Sorting Out Sorts 167
1.23.1. Use What's There 167
1.23.2. Make Your Own 169
1.23.3. What To Do About It 169
1.23.4. The Ordered Table 170
1.23.5. The Presort Concept 170

1.24. The Art Of Assembly — Part 24. Another Sort Of Sort 177
1.24.1. Performance 177
1.24.2. The Error Reports 178
1.24.3. User Guidance 178
1.24.4. Embellishments 178
1.24.5. Program Construction 179
1.25. The Art Of Assembly — Part 25. Comparative Languages 191
1.25.1. No Clear Winner 191
1.25.2. Simple Operations 191
1.25.3. The Back-words Problem 192
1.25.4. The Registers 193
1.25.5. Loops And Reps 193
1.25.6. Initializing Data 195
1.25.7. BIOS And DOS Services 196
1.26. The Art Of Assembly — Part 26. Odds And Ends 198
1.26.1. The Strange Case. 198
1.26.2. The "Final Solution" 199
1.26.3. Other Ins And Outs 199
1.26.4. The Utilities Problem 200
1.26.5. Nobody's Perfect 201
1.27. The Art Of Assembly — Part 27. Another Potpourri 206
1.27.1. String Magic 206
1.27.2. Stashing In Memory 207
1.27.3. Finishing Touches 209
1.28. The Art Of Assembly — Part 28. Never Do This! 216
1.28.1. The Ugly Truth 216
1.28.2. Error Traps Revisited 218
1.28.3. A Fine Bowl Of Spaghetti 219
1.29. The Art Of Assembly — Part 29. Pastafazool, etc. 223
1.30. The Art Of Assembly — Part 30. Using What's There 229
1.30.1. Floating Point Numbers 229
1.30.2. Amazing Accuracy 231
1.30.3. How To Use Them 232
1.31. The Art Of Assembly — Part 31. "This Is A Football" 237
1.31.1. Just What Is Assembly, Anyway? 237
1.31.2. Some Terminology 238
1.31.3. Uses For Labels 239
1.31.4. A Little Exercise 240
1.31.5. Extended Basic Equivalent 242
1.32. The Art Of Assembly — Part 32. Some Instructions 244
1.32.1. A Moving Experience 244
1.32.2. Integer Math Instructions 245
1.32.3. The Tougher Stuff 245
1.32.4. Compare And Jump Instructions 246

1.33. The Art Of Assembly — Part 33. More Instructions 249
1.33.1. Jump Range 249
1.33.2. The Branch Instructions 251
1.33.3. Many Happy Returns 252
1.34. The Art Of Assembly — Part 34. Cramming Time! 254
1.34.1. Immediates Galore 254
1.34.2. A Shifty Business 256
1.34.3. Other Instructions 257
1.34.4. Addressing Modes 257
1.34.5. Final Exams 258
1.34.6. Parting Thoughts 258
1.35. The Art Of Assembly — Part 35. Even More's There! 260
1.35.1. Clearing The Decks 260
1.35.2. The Line Editor 261
1.35.3. Other Undocumenteds 262
1.36. The Art Of Assembly — Part 36. We Keep Learning 275
1.36.1. Do We Feel Stupid? 275
1.36.2. The Free Eye Test 276
1.36.3. Getting Even 277
1.36.4. The Challenge 279
1.37. The Art Of Assembly — Part 37. Option 5 Revisited 282
1.37.1. Making A Call 283
1.37.2. Method Two 285
1.37.3. The Exceptions 286
1.38. The Art Of Assembly — Part 38. Trying The Impossible 293
1.38.1. The Compiler Problem 293
1.38.2. What Is The Program? 293
1.38.3. Why Impossible? 294
1.38.4. Real Source Code 295
1.38.5. Variables 296
1.38.6. Modularity 296
1.38.7. An Update 296
1.38.8. Chapter 11 297
1.39. The Art Of Assembly — Part 39. More Mysteries Unraveled 301
1.39.1. Fooling XB 301
1.39.2. The ERR Report 302
1.39.3. Some Days It's Easy 302
1.39.4. And Then Other Days. 303
1.39.5. But What If. . .? 304
1.39.6. Just One More Thing. 305
1.40. The Art Of Assembly — Part 40. It Pays To Be 310
1.40.1. The Bright Idea 310
1.40.2. Step By Step 311
1.40.3. Another What If. 312

1.40.4. Serendipity Results 313
1.41. The Art Of Assembly — Part 41. It's About Time 317
1.41.1. The Metronome 317
1.41.2. Range Limits 318
1.41.3. The "EUR" Version 319
1.41.4. A Timed Input Field 319
1.41.5. Harry's Explanation 320
1.42. The Art Of Assembly — Part 42. At Long Last Bitmap 325
1.42.1. An Old Story 325
1.42.2. The Source Code 325
1.42.3. The Program 326
1.42.4. The Subroutines 326
1.43. The Art Of Assembly — Part 43. Bit-Map Explained 340
1.43.1. Last Month's Sidebar 340
1.43.2. The Sine Wave 341
1.44. The Art Of Assembly — Part 44. By Popular Request 345
1.44.1. CALLs Revisited 346
1.44.2. The Solution 347
1.45. The Art Of Assembly — Part 45. Some Surprises 352
1.45.1. The Purple Heart Bargain 352
1.45.2. Graphics Capabilities 353
1.45.3. Limitations 353
1.45.4. Time's A Wastin' 353
1.46. The Art Of Assembly — Part 46. Drawing A Straight Line 356
1.46.1. Simplicity Itself 356
1.46.2. Today's Sidebar Shows. 357
1.46.3. The Hard Part 357
1.46.4. Step-By-Step 358
1.46.5. Math Whiz Needed 359
1.47. The Art Of Assembly — Part 47. CALL FILES From XB 366
1.47.1. TI Had Reasons 366
1.47.2. The Dilemma 367
1.47.3. The Sidebar 368
1.47.4. The Tricky Part 369
1.47.5. Caution 369
1.48. The Art Of Assembly — Part 48. Another Use For CALL FILES 371
1.48.1. Our Confession 371
1.48.2. But What If. 371
1.48.3. Today's Sidebar 372
1.49. The Art Of Assembly — Part 49. Gripes And Delays 383
1.49.1. Could Be Worse. 383
1.49.2. Slowing It Down 383
1.49.3. The European Problem 384
1.49.4. Variations On The Theme 384

1.49.5. How It Works 384
1.49.6. What Have We Done? 385
1.49.7. But Then Again. 385
1.50. The Art Of Assembly — Part 50. Delays Again? 390
1.50.1. Delay With Option 390
1.50.2. Another Potential Use 390
1.50.3. What About Extended Basic? 391
1.50.4. The Source Code 392
1.51. The Art Of Assembly — Part 51. Body Of Work 398
1.51.1. The Public Domain Products 398
1.51.2. Extended Basic "Utilities" 398
1.51.3. The All-Assembly Products 400
1.51.4. The "Extras" On The Disks 401
1.51.5. A Word Of Caution 401
1.51.6. A Bud Mills Mystery! 401
1.51.7. What Will We Do Next? 401
1.52. The Art Of Assembly — Part 52. Cheap And Dirty 403
1.52.1. The Seeding Process 403
1.52.2. Okay, We'll Tell! 403
1.52.3. What Have We Done? 405
1.52.4. Other Changes 406
1.53. The Art Of Assembly — Part 53. Yes You Can! 409
1.53.1. Finding Out 409
1.53.2. Don't Thank Me! 409
1.53.3. Not Only That, But. 410
1.53.4. There Are Exceptions! 410
1.53.5. Another Look At Things 410
1.53.6. The New ACCEPT Routine 410
1.53.7. While We're At It 411
1.53.8. How Does It Work? 411
1.53.9. The Magic Part 412
1.53.10. The Cautions 412
1.54. The Art Of Assembly — Part 54. More Random Numbers 421
1.54.1. Without Replacement 421
1.54.2. The CALL LINK 421
1.54.3. The Second Routine 422
1.54.4. And Yet Another 422
1.54.5. The Outer Limits 423
1.54.6. That Fifth Parameter 423
1.54.7. The DEMO Programs 424
1.54.8. The Finer Points 424
1.55. The Art Of Assembly — Part 55. We Need Those Digits 433
1.55.1. Filling An Array 433
1.55.2. What Else Can It Do? 433

1.55.3. How Does It Work? 434
1.56. The Art Of Assembly — Part 56. Playing Cards 442
1.56.1. What's It For? 442
1.56.2. From The Top 442
1.56.3. Build And Shuffle 443
1.56.4. Displaying The Cards 443
1.56.5. Variations On The Theme 444
1.57. The Art Of Assembly — Part 57. We Interrupt This Program. 451
1.57.1. It's Another YES, BUT! 451
1.57.2. But With DSRLNK, ... 452
1.57.3. It Gets Worse 452
1.57.4. Other Things To Try 453
1.58. The Art Of Assembly — Part 58. The Limits Of Randomness 458
1.58.1. The Even Divisor Problem 458
1.58.2. It Gets Worse 458
1.58.3. Can We Fix This? 459
1.58.4. Tossing A Fair Coin 459
1.58.5. More Than One Way. 460
1.58.6. Why The Alternation? 460
1.58.7. Things They Never Taught 460
1.58.8. A Free Extra "Goodie" 461
1.59. The Art Of Assembly — Part 59. Six And A Quarter Cents 468
1.59.1. The Sidebar Program 468
1.59.2. Walking Through It 469
1.60. The Art Of Assembly — Part 60. A Moving Experience 477
1.60.1. The First Steps 477
1.60.2. The Better Way 477
1.60.3. In The Sidebar 478
1.60.4. That User Interrupt 478
1.60.5. Words Of Caution 479
1.61. The Art Of Assembly — Part 61. Full Bitmap Motion 487
1.61.1. Today's Sidebar 488
1.61.2. The Main Event 488
1.62. The Art Of Assembly — Part 62. Book Wrong! 497
1.62.1. How The Book Is WRONG 497
1.62.2. Sprite Subroutines 497
1.62.3. Using The Sidebar 500
1.63. The Art Of Assembly — Part 63. The C Connection 509
1.63.1. Crossing The Bridge 509
1.63.2. The C Side Of The Bridge 510
1.63.3. The Extra Goodies 510
1.63.4. Different Conventions 511
1.63.5. The Choices Are Yours 511
1.63.6. For Whatever They're Worth 511

1.64. The Art Of Assembly — Part 64. Thank You Notes 524
1.64.1. What Is This Script Stuff? 524
1.64.2. Art's Artful Assembler 525
1.64.3. Still More Thanks 525
1.64.4. For My Next Trick 526
1.64.5. The Public Domain Disk 527
1.64.6. The Little Details 527
1.65. The Art Of Assembly — Part 65. 3X5=1X1 538
1.65.1. A Common Denominator 538
1.65.2. The Solution 538
1.65.3. Why Do This? 538
1.65.4. Borrowed Source Code 539
1.65.5. A Rave Review 539
1.65.6. But Then Again 540
1.65.7. Today's Sidebar 540
1.66. The Art Of Assembly — Part 66. Running in Circles 545
1.66.1. The Algorithm 545
1.66.2. Our Assembly Implementation 546
1.66.3. Today's Sidebar 546
1.67. The Art Of Assembly — Part 67. Sounds Good To Me! 556
1.67.1. Continuous Background 556
1.67.2. We Interrupt This Interrupt 556
1.67.3. Two Other Things 556
1.67.4. The Bouncing Ball 556
1.67.5. Loading The Lists 557
1.67.6. Activating Sound Lists 557
1.67.7. Making Sound Lists 559
1.67.8. How Do I Get This Stuff? 559
1.67.9. Today's Sidebar 560
1.68. The Art Of Assembly — Part 68. The Ins And Outs Of Instances 565
1.68.1. The Formatter Problem 565
1.68.2. The Standalone Solution 565
1.68.3. The Drawbacks 565
1.68.4. But What About Editing? 566
1.68.5. Back To The Drawing Board 566
1.68.6. The Saving Process 566
1.68.7. Structure Of An Instance 567
1.68.8. A Small Caution 567
1.68.9. Today's Sidebar 568
1.69. The Art Of Assembly — Part 69. Click On Icon! 571
1.69.1. A Rude Awakening 571
1.69.2. The Help Line 571
1.69.3. CoCo Revisited 572
1.69.4. The Really Big Shock 573

1.69.5. Let This Be A Lesson 574
1.69.6. In The Meantime 574
1.70. The Art Of Assembly — Part 70. The Bathtub Curve 575
1.70.1. The Picture Explained 575
1.70.2. Other Applications 575
1.70.3. Colorful Filling 576
1.70.4. Nobody's Perfect 577
1.70.5. Some Cautions 577
1.70.6. The Subroutines 578
1.71. The Art Of Assembly — Part 71. Floating Points 582
1.71.1. Getting the Numbers In 582
1.71.2. The Rules Of Input 582
1.71.3. Differences From XB INPUT 583
1.71.4. Today's Sidebar 583
1.71.5. Nasty Little Details 584
1.71.6. Operation Of The Program 584
1.72. The Art Of Assembly — Part 72. Still Afloat 597
1.72.1. Starting Up 597
1.72.2. Getting The Numbers 597
1.72.3. Other Math Operations 599
1.72.4. Variations You Can Try 599
1.72.5. The "Caret" Case 600
1.73. The Art Of Assembly — Part 73. Where Is Bryn Mawr? 601
1.73.1. Assembly To The Rescue! 601
1.73.2. The Matrix Structure 602
1.73.3. The Public Domain Version 602
1.73.4. The First Eleven 603
1.73.5. The Exceptional Case 603
1.73.6. The Printing Process 604
1.73.7. The Finishing Touch 604
1.73.8. The Geography Lessons 605
1.73.9. Extra Added Attraction 605
1.74. The Art Of Assembly — Part 74. A Colorful Experiment 606
1.74.1. How Do They Work? 606
1.74.2. The Mapping Problem 606
1.74.3. Multiple Passes 607
1.74.4. Today's Sidebar shows. 608
1.74.5. The Public Domain Disk 608
1.74.6. Caveat Emptor 608
1.75. The Art Of Assembly — Part 75. Reading Disk Sectors 614
1.75.1. Headers The Answer 614
1.75.2. The Keys To The Disk 614
1.75.3. Some Assembly Required 615
1.75.4. Thanks To Travis Watford 615

1.75.5. The Catlog/Ident Process 616
1.75.6. The Sidebar Code 617
1.75.7. The Extra Little Goodies 618
1.76. Art Of Assembly — Part 76. New And Improved 626
1.76.1. Screen Clearing 626
1.76.2. Slightly Better Keyscan 626
1.76.3. Slightly Better Speed 627
1.76.4. MUCH Better Boot Tracking 627
1.76.5. Using the TRACK6 Routine 628

Unpublished articles 634
1.77. The Art Of Assembly — Part 77. Breaking New Ground 634
1.77.1. MIDI Album Without Mini-Memory 634
1.77.2. The Sudden Inspiration 634
1.77.3. Code That "Moves Itself" 634
1.77.4. Workspace Registers 635
1.77.5. A Quick Review 636
1.77.6. Fringe Benefits 636
1.77.7. Cataloging The SCSI Drive 636
1.77.8. More "Fringe Benes" 637
1.77.9. Today's Sidebar 637
1.77.10. Another Myarc Mystery 637
1.78. The Art Of Assembly — Part 78. Selective Cataloging 641
1.78.1. Ending With A Dot 641
1.78.2. Flushed With Success 642
1.78.3. Another Tough Case 642
1.78.4. While We Were There 643
1.78.5. Two More "Goodies" 643
1.79. The Art Of Assembly — Part 79. Playing It In 649
1.79.1. A Separate Program 649
1.79.2. Borrowing A Concept 649
1.79.3. Gotta Be Quick 650
1.79.4. Once It's In The Memory 651
1.79.5. Saving Your Work 651
1.79.6. Loading Saved Works 652
1.79.7. The De-Compiler 652
1.79.8. Loop Closed 652
1.80. The Art Of Assembly — Part 80. What's A NewLine? 658
1.80.1. The PC Connection Problem 658
1.80.2. Meanwhile, There's Cakewalk (TM) 658
1.80.3. Pre-AMS Ways 659
1.80.4. Once That Was Working. 659
1.80.5. The Filter Rules 660
1.80.6. The Editing Process 660

1.80.7. The Disk Is Available 660
1.81. The Art Of Assembly — Part 81. Strange Happenings 665
1.81.1. But We Digress. 665
1.81.2. And Then Came E-mail 666
1.81.3. The List Server Saga 666
1.81.4. The Other Penalty 667
1.82. The Art Of Assembly — Part 82. Click On Icon Again 668
1.83. The Art Of Assembly — Part 83. The End Of An Era 670
1.83.1. Third Party Hardware 670
1.83.2. My First Experience 670
1.83.3. The Big "G" 671
1.83.4. Other Fun Hardware 671
1.83.5. The Helpless Community 672
1.83.6. The Final Straw 673

1. The Art Of Assembly Series

This section contains the Art Of Assembly series that appeared in *MICROpendium*. Thanks to Bruce Harrison for supplying this material.

1.1. The Art Of Assembly — Part 1

By Bruce Harrison

Frustrated with Extended Basic? Tired of waiting for C? Fed up with Forth? P. O.'d at Pascal? The answer to your problems is the "Native Language" of your computer's heart, Assembly Language.

Many programmers today shun this language as being unnecessary, antiquated, and obsolete. We who do our programming in Assembly believe that it's the most valuable of all computer languages. There are three things that make Assembly worth while: (1) It maximizes the speed of execution of any operation we're trying to perform; (2) It can minimize the memory required to perform any given tasks; and, (3) Through Assembly we gain access to all the facilities and capabilities the computer has to offer. No other language can make those three things true at the same time.

From the programmer's viewpoint, there are two major drawbacks to Assembly: (1) It is very labor-intensive. A simple "Accept At" function may require two pages of source code to implement; (2) It requires a much more intimate knowledge of what really goes on in the computer. Such knowledge takes lots of study, and much trial and error plodding to acquire.

This series of articles is based upon years of experience, much of it painful, in exploring the capabilities and limitations of the TI-99/4A through Assembly programming. It is not designed as a beginner's course. For that, we recommend Ralph Molesworth's excellent book *Introduction to Assembly Language Programming on the TI-99/4A* from Steve Davis publishing.

In this first installment, we'll cover some general topics as background for the programmer who's ready to move beyond the beginner stage, but is not quite sure how to proceed. We'll cover the topics of Structure and Memory mapping. This will be very general coverage, just to give you the "feel" of thinking through your programming efforts. In later installments, we'll get into the more detailed aspects so you can become comfortable in programming with Assembly.

Structure is your servant! We say that deliberately. For many programmers the relationship becomes the wrong way around, as they slavishly "structure" far beyond any logical reason or necessity. Structure in your programming effort should help you to keep your efforts organized and focused, and in some cases will help minimize the memory required to hold your programs and data. It must not be allowed to become an end unto itself.

Perhaps a small example will help illustrate my point. In a book on PC Assembly language, the author put together a whole book of subroutines which, for the most part, could be lifted directly and used in PC programs. In some instances, however, he went overboard with structure. He gave a subroutine to place a single character on the screen. To use the subroutine, one would place the desired character's ASCII value in a register, then call the subroutine to display that character. He presented another subroutine to place a space on the screen. That subroutine simply placed ASCII 32 in the register, then called the "display a character" subroutine.

What's wrong with that process is mainly that there's twice as much "overhead" in both time and memory usage to print a space that way. The main program could put any character, including a space, in the register, then call the "display character" subroutine, rather than involve two levels of subroutine to perform the same function.

That kind of thinking is rampant in the PC Community, and is one of the reasons PC owners need *Megabytes* of memory to run commercial software packages. On the TI, with its limited memory capacity, we can't afford that kind of thinking. Again, structure is useful only so long as it serves the programmer.

I'll cite just one other example of structure gone amok, from a TI Basic program I once examined. (I won't name the program or the author.) This program used a menu selection to execute its functions. Each function was organized as a subroutine. Not one of those subroutines was called from more than one place in the main program. A simple ON-GOTO to branch directly to the desired section of the code would have done nicely, with a GOTO at the end of each function to return to the menu. In later installments of this series we'll show an efficient and effective way to perform branching from a menu-select situation, using an Assembly version of the ON-GOTO function.

So how does one sensibly apply structure without going overboard? There are two approaches which we use here at Harrison in combination. They're called Top Down and Bottom Up. From the Top Down, we recommend that some kind of overall flow chart be constructed early in the "thinking" stage of the program. For many programmers, it will help to actually draw a chart of the flow through the program's major functions. In some cases, a physical chart won't be required, but there should be at least a mental image of what the major functions are and how they should relate to one another. On occasion in my programming experience, I've ignored my own advice on this matter, and in all such cases have gone through endless agonizing revisions and re-writings of code because I omitted that first step. Once the major functions are identified, the Top Down approach proceeds to break those into smaller and smaller subdivisions of what needs to be done. From this a pattern will emerge, showing that many places in the main stream program will need the same primitive operations performed. This is where the idea of subroutines becomes a powerful tool, and it's also where the Bottom-Up idea can be useful.

In Bottom-Up programming, we start with simple functions, such as getting keystrokes from the keyboard, or placing characters on the screen, then build a program structure to optimize the use of these "primitive" tools.

Good programs need the influence of both these approaches at the same time.

Once the overall structure is broken down a couple of levels, we should have a clear view of what kinds of subroutines we'll need, and how to use them in building upward to bigger structures like menu drivers, input screens, and so on. Experienced Assembly programmers usually have a stable of existing subroutines developed as part of other efforts, so they can use those, usually with minor modifications, in the new program. In future articles, we'll present actual source code for subroutines we've found useful.

Memory is your Master! Now let's move on to the subject of Memory. There isn't much, so we must be careful how we use it. That starts with a knowledge of what we can use. There are two major blocks of memory available to the Assembly programmer. In Low Memory, from >2000 thru >3FFF, there are about 6K bytes that we can safely use, reserving the space at the beginning for the E/A utilities, and space at the end for the REF/DEF table.

In High Memory, there is lots of space, about 24K bytes from >A000 through >FFE6. In a normal E/A Option 3 program, only this 24K byte section will be open for your use as program storage. There are ways to make effective use of the low memory part as well as the high memory part, but these require techniques such as AORG, which we're not ready to cover just yet. Just to give you a hint, virtually every program we write here at Harrison involves use of AORG to give us maximum use of the available memory.

One frequently overlooked memory resource is the memory associated with the Video Display Processor, also known as VDP RAM. This can't be used directly for executable code, but can be used for a kind of "auxiliary" data storage. In most modes of VDP operation, there are about 10K bytes of VDP RAM that can be safely used to stash data.

In this series of articles, we'll show many techniques for saving memory in performing various functions.

As you already know, our good friend Barry Traver is writing a series of articles on using Assembly routines along with Extended Basic. Our series of articles is intended for the programmer who's trying to make whole programs in Assembly. We'll make every effort not to overlap Barry's efforts, but there will be instances where we may give slightly different versions of routines that he's already covered. At some point in the series, we plan to cover methods for making All-Assembly programs operate with the Extended Basic module, or perhaps we should say in spite of the XB module.

In our next article, we'll start from the bottom up with some primitive subroutines that we've used. Along with that, we'll show the techniques for minimizing use of memory and maximizing speed of execution. When the series is done, we'll offer the whole series on disk as D/V 80 files to make them easier to access.

```
LOW MEMORY

>2008 thru >24F2 = XB Utilities

>24F4
MENUCODE - ENTRY AND MENU

>2A66
PRINTCODE - PRINTING PART & CONFIGURATION


>39FE = END OF CONFIGURE PART

HIGH MEMORY

>A000
WORDCODE1 - 9898 BYTES

WORDCODE2 - 9898 BYTES

WORDCODE3 - 1132 BYTES

>F1C0 = END OF MAIN CODE

>F690 = START OF LOADER CODE

>FFE6 = END OF DSRLNK UTILITY
```

Figure 1 - Memory Mapping of Word Processor

1.2. The Art Of Assembly — Part 2. Starting At The Bottom

By Bruce Harrison

In Part one, we discussed the two approaches to program structure, Top Down and Bottom Up. In this article we'll provide some "primitive" source code sections to provide services. Please note that, in Assembly, there are about as many ways to do any given thing as there are programmers trying to do it. We'll try to provide the rationale for the way we approach things as we go along. In general, our approach is to minimize memory consumption and maximize speed of execution. Those two don't always go together, but in many cases the most memory-efficient code also executes fastest.

Bear in mind that, for the time being, we're working in the environment of an E/A Option 3 (Load and Run) program. Let's start with the matter of providing Workspace Registers. Many programs contain a source statement like:

```
WS          BSS 32
```

That's fine, but doing this uses 32 bytes of the available program memory for your registers. There is an area in low memory designated for User Workspace, at address >20BA. To use that, you can make an equate in the beginning of your source code like this:

```
WS          EQU >20BA
```

Now at your program's start point, you can simply LWPI WS, and your registers will be at >20BA, not taking up 32 bytes of program space. (Please note this should not be done when linking from Extended Basic, unless your program never returns to XB until it's finished.)

Let's quickly move on to another subject, that of a subroutine to clear the screen for you. We've used many different techniques for this, so let's explore a couple of alternatives. One can do it like this:

```
            CLS  CLR R0             Point R0 at screen origin
            LI   R2,SCRWID*24       Load R2 with total
            LI   R1,>2000           make left byte of R1 the space
LOOP        BLWP @VSBW              Write one space
            INC  R0                 Increment screen location
            DEC  R2                 Decrement counter
            JNE  LOOP               If not zero, repeat operation
            RT                      Return to calling program
```

Here you'll see one of our little tricks. Sometimes when starting a program, we don't know for sure whether we want to operate in Graphics mode or in Text mode. Thus in many places in the program we'll use the mnemonic SCRWID, then at the beginning of the program we'll put a value in for SCRWID through an equate like SCRWID EQU 32 or SCRWID EQU 40. This was really a two-barreled trick, because it also lets the assembler do some math for us. In this case, the assembler will multiply 24, the number of rows on the screen, by the number of characters per row (SCRWID) and thus will load R2 with the correct number of spaces to fill the screen. The above method will work, but won't be as fast as a method using VMBW to write whole screen lines to the screen. We can gain some speed by setting aside a block of 32 or 40 characters' space, writing a space into each of those, then writing 24 such lines to the screen. There would need to be a block of bytes reserved, like this:

```
SCRLI       BSS  SCRWID
```

There's our friend SCRWID again, this time telling the assembler how many bytes to reserve for a screen line full of characters. Now the code to clear the screen gets more complicated and takes more memory, but executes faster:

```
            CLS  LI R2,SCRWID       Sets R2 to characters in screen line
            LI   R5,>2000           Sets left byte R5 to space
            LI   R3,SCRLI           Point R3 at SCRLI
            MOV  R3,R1              Point R1 at SCRLI also
LOOP1       MOVB R5,*R3+            Move one byte and increment R3
            DEC  R2                 Decrement R2
            JNE  LOOP1              If not zero, repeat
            CLR  R0                 Point R0 to screen origin
            LI   R2,SCRWID          Set R2 again
            LI   R4,24              24 rows to clear
LOOP2       BLWP @VMBW              Write SCRWID bytes to screen
            A    R2,R0              add that many bytes to R0
            DEC  R4                 Decrement R4
            JNE  LOOP2              If not zero, repeat
            RT                      Return to calling program
```

That block of memory which we set aside as SCRLI can be used for other purposes, as you'll see when we get to some other subroutines. We can, for example, use it to stash strings.

Before we go further with subroutines, we ought to discuss how to properly "nest" them in Assembly. If you're used to programming in Basic or XB, you know that subroutines may include GOSUBs to other subroutines, and that so long as each ends with RETURN, all will be well.

In Assembly, the calling of a subroutine by BL @SUBNAM will work properly only if the subroutine does not call others. To get around this problem, we establish a "stack" to keep track of our subroutine return addresses. To do this, set up a data area somewhere (perhaps at the very end of your program) which will contain the return addresses for nested subroutines. A simple entry such as:

```
SUBSTK      BSS  24
```

This 24 bytes will suffice to hold 12 levels of nesting. The other requirement is to have a pointer to keep track of position in that stack. We simply dedicate R15 to that purpose. Somewhere in the beginning of the program, we insert LI R15,SUBSTK, so that before we call any subroutines, R15 points to the beginning of that stack.

Now in any subroutine that calls others before it returns, which we define as a High level subroutine, we place this instruction at the beginning of the subroutine:

```
            MOV  R11,*R15+
```

That puts the R11 return address in the location pointed to by R15, and makes R15 point to the next word in the stack area. At the end of one High level subroutine, we place the following code:

```
SUBRET      DECT R15                Point back to previous stack word
            MOV  *R15,R11           Move that word to R11
            RT                      Return.
```

Other high level subroutines can return by a simple B @SUBRET. Note that simple subroutines that do not in turn call others, which we'll call Low level subroutines, need only the RT at their ends to return properly. The stack area can be placed anywhere. We recommended putting it at the very end of a program so it's open-ended, as long as the program doesn't fill all of the computer's memory. Placing it elsewhere is okay so long as you're sure about how many levels of nesting are required. If you underestimate, something important could get overwritten by your stacking.

Let's say that you are writing a High level subroutine which needs to have the screen cleared before it can proceed. The subroutine would look something like this:

```
BIGSUB      MOV  R11,*R15+          Stash R11 on SUBSTK
            BL   @CLS               Clear the screen
            (rest of subroutine)
            B    @SUBRET            Go to the high level return
```

This assumes there is already another High level subroutine which ends with the code shown above at label SUBRET. By this method, subroutines may be stacked to any number of levels without losing track of the return address of any subroutine.

Now we'll move on to a few more handy subroutines, and introduce the idea of multiple entry points. Let's say you'll need an ability to move strings around in memory, and you'll also need the ability to move groups of bytes that are not organized into strings. (For our purposes, a string is merely a group of bytes where the first byte is the length, and the rest of that many bytes is the string. For example, we might have a string initialized in our data section like this:

```
CPYWRT      BYTE 14                 Length of text
            TEXT 'Copyright 1991'
```

The first byte is 14, which is the length of the string that follows.

Now let's suppose that we want to move that string to another location which we'll call TEMSTR for Temporary String. Assume that at least fifteen bytes of memory have been reserved at that place. We'll be using R9 to point to the origin of the string and R10 to point to the destination address. We can preload registers with the addresses to move from and to, like this:

```
            LI   R9,CPYWRT          Put address of CPYWRT in R9
            LI   R10,TEMSTR         Put address of TEMSTR in R10
```

Now that pointers have been set, we can proceed with a BL @MOVSTR, where the subroutine looks like this:

```
MOVSTR      MOVB *R9+,R4            Get length byte in R4
            MOVB R4,*R10+           Place that byte at R10 location
            SRL  R4,8               Right-justify length in R4
MOVBTS      MOVB *R9+,*R10+         Move one byte, inc pointers
            DEC  R4                 Decrement length count
            JNE  MOVBTS             If not zero, repeat
            RT                      Else return
```

This subroutine uses R4 as a counter for the loop at MOVBTS. We here at Harrison conventionally use R4 and R5 for loop counters or other temporary numbers. But just for a moment let's assume you have a need to move a group of bytes from one place to another but they're not organized as a string, in that there's no length byte at the beginning. Let's say you have 75 bytes to move from location XYZ to location ZXY. Here you can use the label MOVBTS as a second entry point to the subroutine. You'd do it like this:

```
            LI   R9,XYZ             Place source address in R9
            LI   R10,ZXY            Place destination in R10
            LI   R4,75              Number of bytes in R4
            BL   @MOVBTS            Call subroutine MOVBTS
```

This technique has been used many times in our programs, and we've found it very useful, in that it's more efficient in use of memory than having two separate subroutines with such similar functions.

Next, let's look at a very small subroutine which has an important lesson to teach us. Assume that you've got many places in the program that require a single-keystroke entry, such as the answer to a Y/N question. To prepare for such a subroutine, we'll put the equates STATUS EQU >837C, KEYADR EQU >8374 and KEYVAL EQU >8375. Then near the start of our program we'll insure that our key-unit is zero by writing this one line of source code CLR @KEYADR. We'll also need somewhere a byte initialized to the value >20, such as ANYKEY BYTE >20. We can then use the short subroutine like this:

```
KEYLOO      CLR  @STATUS            Clear the GPL Status byte
            BLWP @KSCAN             Use utility to scan keyboard
            CB   @ANYKEY,@STATUS    Has a key been struck?
            JNE  KEYLOO             If not, try again
            MOV  @KEYADR,R8         Else put key struck in R8
            RT                      Then Return
```

You'll notice that there's an extra instruction in there which moves the word at >8374 into R8. The left byte of that word will be zero, and the right byte will be the value of the key struck. Thus the register's value will equal the ASCII code for the keystroke. We do this on purpose because, in most cases after we return from this subroutine, we have to do a series of comparisons to the key struck. Having the key's value already in a register makes that process easier, and moving the key value into a register before exiting the subroutine uses less memory than doing it after return. Suppose we had asked a Yes/No question, and want the default answer to be No. Upon return from the above subroutine, we could have:

```
            CI   R8,89              Is answer upper case Y?
            JEQ  YES                If so, Jump
            CI   R8,121             Is answer lower case y?
            JNE  NO                 If not, answer is No
YES         (perform action for Yes)
NO          (perform action for No)
```

The activity at label YES may be a simple branching to some other part of the program, and label NO may be a simple continuation of some process, but that's not important to our point. By moving KEYADR into R8 in the subroutine, we'll save many bytes of memory if this kind of comparison needs to be done each time we've used the subroutine. The point is that the content of a subroutine should be considered very carefully. A small added function like we've shown in the above example can add up to significant savings of bytes by incorporating it into the subroutine instead of having to repeatedly perform the operation outside the subroutine.

In this article, we've just scratched the surface of the subject of subroutines. In the next article, we'll go back to the subject of structure for a bit, and discuss some of the minimum required things to get a program started and ended gracefully. In later articles of this series we'll move into more advanced subroutines, some of which will depend on things we presented here.

1.3. The Art Of Assembly — Part 3. Starting At The Top

By Bruce Harrison

In Part 2 of this series, we discussed and showed some small "primitive" subroutines and the methods for nesting them. In this article, we are going back to the "Top Down" part of writing Assembly programs. We will use for our example the Harrison Golf Score Analyzer, since its development went pretty much along the lines we're trying to encourage.

One of the first decisions you should make is how the user will interact with your program. In many games, for example, the principal means of interaction is the joystick. In a program like a Golf Score Analyzer, however, that would be a very poor interface for user input. Our preference is for simple menu interaction at the top level, so each main function of the program is readily apparent to the user, and selection of a function is just one keystroke away. In today's world of "Graphical User Interface" (GUI), where functions are represented by pictures, not words, this makes us very old-fashioned, but we do have a reason for being that way. GUIs normally require a mouse to select options, and one can't count on every customer having a mouse. Further, a mouse can't be used to input names, numbers, and other data, so with or without a mouse one still needs the keyboard. Our choice has been to require only the keyboard, and that makes "plain English" menus the natural choice for selecting functions.

Given that, we must make a decision as to what functions belong on the main menu. In the Golf Score Analyzer, we settled on eight functions for the Main Menu, and made each require only a single keystroke to select. The eight look like this:

```
1 ADD ROUNDS
2 LOAD FILE
3 DELETE DATA
4 ANALYZE DATA
5 SAVE FILE
6 ADD/EDIT COURSES
7 REVIEW COURSES
8 EXIT PROGRAM
```

It's important to always include an exit selection, so the user can easily get out of your program when he wants to. It's equally important to make it difficult or impossible to get out of the program by accident. In this program, selecting item 8 from the Main Menu is the only way to get out. We made **FCTN = (QUIT)** inactive in this program. As an aside, when users are looking at subsidiary menus, **FCTN 9 (BACK)** will get back to the previous menu, but that will not get them out of the program.

In this particular program, we had a special reason for making one and only one exit point. When the user selects item 8, we perform a check to see whether the user has modified the file currently in memory. If he's not made any changes to the file, or if he's saved it since making changes, we simply return him to either XB or E/A, depending on how he entered the program. If changes have been made, we produce a prompt asking whether he'd like to save the changed file before exiting. Any answer other than N or n is taken as Yes, and he's placed in the SAVE FILE function. We take these precautions as part of our concept of "User Friendliness".

Perhaps we could illustrate the concept of User Friendliness by an example drawn from experience. In many instances on the TI, one will encounter an error in execution of some program. Let's say we're working in XB or E/A, and try to get a nonexistent file to open for INPUT. What the TI folks will give you is a numbered "ERROR CODE", which you'll have to look up in a book. When we write our own programs, we like to provide a more definitive error indication, like "THAT FILE DOES NOT EXIST ON DRIVE x" or "THERE IS NO DISK IN DRIVE x". This way the user has a very definite idea of what's wrong. Doing this of course eats memory, since those error messages have to be stored somewhere in the computer and printed to the screen, but we think that's a worthwhile use of memory.

But we digress. Once one has decided upon a menu, the top part of the flow chart is readily apparent. There will need to be an opening section of code that sets up such things as screen mode, color scheme, and such, then displays our copyright notice. Next is a delay loop so the user can read the copyright notice, and then we clear the screen and produce the main menu. Here we had to make a decision. Since we knew there would be more than one menu, we could have each menu produced by a separate section of code, or we could provide a "Menu Driver" section of code that would produce all the required menus simply by using different data with the same code. We chose the latter, and believe that was a wise decision, because we used less memory to do it this way. Our Word Processor, which we use to prepare these articles, also has a central menu driver, but the one in the Golf Score Analyzer is better, taking lessons learned from the WP program into account.

Each menu we use has a section of data associated with it, which includes the title for the top of that menu, the selections, and a "branching" lookup table, which indicates where the program will go to when it exits that menu. The legend "SELECT BY NUMBER" goes at the bottom of each menu, so the menu driver itself places that legend on each menu it displays. In our Golf Score Analyzer, by the way, we separated the code from the data into sections of memory. That is, all the executable instructions are together in a block of memory, then all the data, including text for messages and menus, is in its own block of memory. This makes a somewhat neater arrangement for the programmer, in that separate source files contain the data, and it becomes a bit easier to keep track of what one is doing while developing the program. It also makes it easier when one comes back six months later to change something in the program.

Actually, there's no reason you can't scatter data all over the place, between sections of the executable code, but our thinking on the subject has been colored by the fact that we also program in PC Assembly language, where different memory segments are (and must be) allocated for code and data. This becomes a habit that carries over to the TI.

There we are digressing again. Just for the heck of it, let's look at some of the source code. In the Sidebar is the annotated source code associated with the menu driver for the Golf Score Analyzer. The first two executable lines are the required setup before branching to the driver. These lines set R9 to point to the data for the menu itself, and R13 to point to the lookup table for branching out of the Main Menu.

In the Driver itself (MENDRV), the first order of business is to clear the screen. The CLS subroutine is similar to the one shown in our last article, except that, since GSA was written to operate from Extended Basic, it adds an offset of >60 to the spaces it writes into SCRLI. As an ironic sidelight, we later added a loader so that GSA could be run from E/A, and in that loader we had to, among other things, re-arrange the tables in VDP so it would need the character offset.

Before delving further into the code, let's look at the structure of the data for the menu, at label MENDAT. It starts with a byte giving the length of the title for the menu. Next is the text of the title, then two bytes. The first of these is the number of items in this menu (8), and the second is the length of the first item description (13). After this is the text for the first item, followed by the strings for the rest of the items (a length byte, then text content). By organizing the data this way, we can make a loop in the menu driver that minimizes the memory used for the driver's code.

The business of getting the menu on-screen now proceeds by taking the length of the title line and manipulating that to position R0 so the title appears centered on Row 2 of the screen. The subroutine DISLI could also be called DISSTR, since what it does is take a string pointed to by R9 and display it at the screen location pointed to by R0. Another irony here is that, had we done this in E/A only, we could have used R1 as the pointer to the string, then DISLI would reduce to:

```
DISLI       MOVB *R1+,R2            Get length byte into R2
            SRL  R2,8               Right justify R2
            BLWP @VMBW              Write characters to screen
            A    R2,R1              Advance R1 beyond text
            RT                      Return
```

But we didn't do that, because we wanted GSA to be available to those who don't have the E/A module, but only the XB module. Thus we're stuck with that offset, even when the user enters the program from E/A. Live and Learn!

Another small note before we examine the rest of the source code. There is no such thing as a perfect program. As your author looks at his own Sidebar, he can see several places where it could be improved. For example, the line just before label LOP1 in the CLS subroutine could be eliminated if the line at LOP1 said MOVB @SPACE,*R6+. Our good friend Jim Peterson (TIGERCUB), calls this kind of thinking Elegant Programming, where the programmer not only wants it to work, but wants it to be fully optimized in all respects. Maybe our next program will be better, but we're not going to re-assemble GSA just for that one small possible change.

Okay, so after the title is on the screen, we have a section of code that picks up the byte just after the title text, transfers that to R8, right justifies it, then stashes it at NOITEM. As it happens, the main menu has eight items, which is the most of any menu used in the program. The next section of code does some math with R0 and R8 to position the bulk of the menu vertically centered between the top and bottom of the screen. At label MEN1, we enter a loop which prints all the selections on the menu. Each call to DISLI leaves R9 pointed at the length byte for the next item, so the loop can proceed very quickly and efficiently.

Once all the items have been displayed (after JNE MEN1) there's another of our little tricks. We want the legend to appear at row 23, column 9. To do that, we let the assembler do the math for us. The assembler multiplies 22 by the width of the screen (this would place us at row 23, column 1), then adds eight to that number. The result is an immediate value placed in R0 which puts R0 just where we wanted it. This trick can be used in many ways, but here we've used it for positioning on the screen. One takes the number of the desired row, subtracts one, then tells the assembler to multiply by SCRWID, and then to add one less than the desired column. In addition to saving us some math, this also saves some time in program execution, because the math is performed during the assembly, and all the computer has to do at running time is load that one value into R0.

Now, once the legend is on the screen, all we need do is wait for the user to press a key. KEYLOO is the subroutine that does this for us, (see Part 2 for that subroutine) and in addition places the ASCII value of the struck key in R8. Given a keystroke, the menu driver checks it against the value 15. Fifteen happens to be the ASCII value for **FCTN 9**. In any of the menus in this program, striking **FCTN 9** makes a branching to the last label in the lookup table for that menu. In the case of this main menu, that simply takes us back to KYIN for another keystroke. In other menus, that last label in the lookup table takes us back to a previous menu.

Having found some key value other than 15, the program must now make sure that the key struck in within the correct range for this menu. In this case that's 1 through 8. We move the keystroke to R5, subtract >30 so the number in R5 will be 1 through 8, not >31 through >38. Now we check for a result zero or less than zero. If either happens, the key struck was out of range, so we ignore it and jump back to label KYIN. Finally, we compare R5 to the data at NOITEM, which in this case contains 8. If it's greater than that, we again ignore the keystroke. While this menu is on-screen, hitting any key other than the numbers 1 through 8 will have no effect whatsoever.

Immediately after the operation JGT KYIN, we know that the number in R5 is a number in the range 1 through 8, so we can proceed to branch out from the menu. First we must DEC R5, so that the range is actually 0 through 7. (If **FCTN 9** had been struck, we'd jump to label ACC2 with 8 in R5.) Now, since we're going to index a table of words, not bytes, we must double the number in R5. The easiest way to double a simple integer like this is to Shift it left by one bit, and that's what we do at label ACC2. Now the number in R5 has a range of 0 through 14 (by twos), or 16 if **FCTN 9** has been pressed.

R5 now has the index value for the member of the lookup table we want. We add R13, which contains the address of the start of the lookup table. The next operation, MOV *R5,R5, takes the number at that address in the lookup table and places it in R5. Finally, we branch to the address contained in R5, and that takes us into the selected function. In effect, we have performed an ON-GOTO function based on the key struck.

In this article we looked at some overall principles for Top-Down program design, then we presented one alternative for user interaction through a Menu Selection, and showed the source code for a reasonably effective menu driver. There are many other ways to implement a menu system, and we can be sure that some of our readers will come up with better ones than ours. Our purpose in these articles is mainly to teach principles of using Assembly, so the reader can use his own creativity in this language.

In the next installment, we'll try to concentrate on ways to make code as efficient in memory use as possible, with some "wrong way" and "right way" examples.

```
* PORTIONS OF SOURCE CODE FROM GOLF SCORE ANALYZER
*
* EQUATE FOR 32 CHARACTER SCREEN
SCRWID EQU  32
*
* SETUP FOR ENTERING MENU DRIVER TO MAKE MAIN MENU
       LI   R9,MENDAT
       LI   R13,MAINBR
       B    @MENDRV
*
*
* MENU DRIVER SOURCE CODE STARTS
MENDRV
       BL   @CLS          CLEAR THE SCREEN
       LI   R0,SCRWID     SET R0 TO SCREEN WIDTH
       MOVB *R9,R1        GET LENGTH OF TITLE IN R1
       SRL  R1,8          RIGHT JUSTIFY LENGTH
       S    R1,R0         SUBTRACT LENGTH FROM SCREEN WIDTH
       SRL  R0,1          CUT THAT NUMBER IN HALF
       AI   R0,SCRWID     ADD ONE SCREEN WIDTH
* THE ABOVE SECTION SETS R0 AT A VALUE WHICH WILL AUTO-CENTER THE TITLE
* IN ROW 2 OF THE SCREEN
       BL   @DISLI        DISPLAY THAT LINE OF TEXT
* DISLI ADVANCES R9, SO IT NOW POINTS TO BYTE BEYOND END OF TITLE'S TEXT
       MOVB *R9+,R8       GET NUMBER OF ITEMS FOR MENU
       SRL  R8,8          RIGHT JUSTIFY IN R8
       MOV  R8,@NOITEM    STASH THE NUMBER OF ITEMS AS DATA
       LI   R0,8          LOAD R0 WITH MAXIMUM NUMBER OF ITEMS IN ANY MENU
       S    R8,R0         SUBTRACT THE NUMBER OF ITEMS
       AI   R0,4          ADD FOUR
       LI   R3,SCRWID     GET R3 TO EQUAL NUMBER OF CHARACTERS IN SCREEN WIDTH
       MPY  R3,R0         MULTIPLY BY WIDTH OF SCREEN
       AI   R1,8          ADD 8 FOR COLUMN POSITIONING
       MOV  R1,R0         PLACE THIS NUMBER IN R0
```

```
* THE CODE ABOVE SETS R0 TO VERTICALLY CENTER THE NUMBER OF ITEMS IN THE MENU
* FOR A MORE CONSISTENT SCREEN APPEARANCE.
MEN1   BL   @DISLI       DISPLAY A LINE OF THE MENU
       AI   R0,SCRWID*2  MOVE DOWN-SCREEN BY TWO LINES
       DEC  R8           DECREMENT COUNTER FOR NUMBER OF ITEMS
       JNE  MEN1         IF NOT ZERO, JUMP BACK TO DISPLAY NEXT ITEM
       LI   R0,22*SCRWID+8  SET R0 FOR ROW 23, COLUMN 9
       LI   R9,SELEC     POINT TO STRING FOR "SELECT BY NUMBER"
       BL   @DISLI       DISPLAY THAT LEGEND
KYIN   BL   @KEYLOO      GET A KEYSTROKE
       CI   R8,15        WAS FCTN-9 STRUCK?
       JNE  ACC1         IF NOT, JUMP AHEAD
       MOV  @NOITEM,R5   ELSE PUT NUMBER OF ITEMS IN R5
       JMP  ACC2         THEN JUMP
ACC1   MOV  R8,R5        PLACE KEYSTROKE IN R5
       S    @NUMASK,R5   SUBTRACT >30 SO R5=NUMBER
       JEQ  KYIN         IF R5 ZERO, GO GET ANOTHER KEYSTROKE, IGNORE THIS ONE
       JLT  KYIN         IF R5 < ZERO, IGNORE
       C    R5,@NOITEM   ELSE COMPARE TO NUMBER OF ITEMS
       JGT  KYIN         IF GREATER, IGNORE
* AT THIS POINT, WE KNOW A NUMBER KEY WITHIN THE CORRECT RANGE HAS BEEN STRUCK
       DEC  R5           ZERO-BASE THE VALUE IN R5
ACC2   SLA  R5,1         DOUBLE THAT NUMBER, SINCE WE'RE INDEXING BY WORDS
       A    R13,R5       ADD TO R5 THE START OF THE BRANCHING TABLE
       MOV  *R5,R5       GET THE ADDRESS OF THE SELECTED CODE SECTION INTO R5
       B    *R5          AND BRANCH TO THAT ADDRESS
* END OF MENU DRIVER SOURCE CODE
* SUBROUTINE TO CLEAR SCREEN WITH OFFSET FOR XB
CLS
       LI   R4,SCRWID    SET R4 TO WIDTH OF SCREEN
       MOV  R4,R2        MAKE R2 ALSO = WIDTH OF SCREEN
       LI   R6,SCRLI     POINT R6 AT SCREEN LINE STORAGE
       MOV  R6,R1        PLACE THAT ADDRESS IN R1 ALSO
       MOVB @SPACE,R5    PUT A SPACE WITH OFFSET INTO R5
* THE BYTE AT LABEL SPACE IS >20 + >60 FOR XB'S OFFSET
LOP1   MOVB R5,*R6+      MOVE ONE SPACE WITH OFFSET, INC R6
       DEC  R4           DECREMENT COUNTER
       JNE  LOP1         IF NOT ZERO, REPEAT
       CLR  R0           SET R0 TO SCREEN ORIGIN
       LI   R4,24        24 ROWS TO CLEAR
LOP2   BLWP @VMBW        WRITE ONE LINE OF SCRWID SPACES
       A    R2,R0        ADD SCRWID TO R0
       DEC  R4           DECREMENT ROW COUNT
       JNE  LOP2         IF NOT ZERO, REPEAT
       RT                ELSE RETURN
* SUBROUTINE TO DISPLAY ONE STRING ON THE SCREEN
DISLI  LI   R10,SCRLI    POINT AT OUR BUFFER SCRLI
       MOV  R10,R1       MAKE R1 POINT AT THAT ADDRESS ALSO
       MOVB *R9+,R4      MOVE THE LENGTH BYTE INTO R4
       SRL  R4,8         RIGHT JUSTIFY
```

```
       MOV  R4,R2        PLACE THAT NUMBER IN R2 FOR VMBW
       JEQ  DISLIX       IF THAT LENGTH WAS ZERO, GET OUT OF SUBROUTINE
DIS1   MOVB *R9+,*R10    MOVE ONE BYTE OF CONTENT, INCREMENTING R9
       AB   @OFFSET,*R10+ ADD THE >60 OFFSET, AND INCREMENT R10
       DEC  R4           DECREMENT LENGTH COUNT
       JNE  DIS1         IF NOT ZERO, REPEAT
       BLWP @VMBW        WRITE THE STRING WITH OFFSET TO SCREEN
DISLIX RT                RETURN

*  FOLLOWING LINES ARE FROM THE DATA SECTION OF SOURCE CODE
*  DATA FOR PRODUCING THE MAIN MENU
MENDAT BYTE 19           LENGTH OF TITLE
       TEXT 'GOLF SCORE ANALYZER' TITLE TEXT
       BYTE 8,13         NUMBER OF ITEMS, LENGTH OF FOLLOWING TEXT
       TEXT '1  ADD ROUNDS' TEXT LINE
       BYTE 12           LENGTH OF TEXT FOLLOWING
       TEXT '2  LOAD FILE' SECOND TEXT LINE
       BYTE 14
       TEXT '3  DELETE DATA'
       BYTE 17
       TEXT '4  ANALYZE SCORES'
       BYTE 12
       TEXT '5  SAVE FILE'
       BYTE 19
       TEXT '6  ADD/EDIT COURSES'
       BYTE 17
       TEXT '7  REVIEW COURSES'
       BYTE 15           LENGTH OF LAST TEXT LINE
       TEXT '8  EXIT PROGRAM' LAST TEXT LINE
* DATA FOR PRODUCING THE LEGEND AT BOTTOM OF ANY MENU
SELEC  BYTE 16           LENGTH OF LEGEND
       TEXT 'SELECT BY NUMBER' TEXT OF LEGEND
* LOOKUP TABLE FOR BRANCHING OUT FROM MAIN MENU
* EACH DATA ITEM AT MAINBR GIVES AN ADDRESS OF A LABEL TO WHICH CODE
* BRANCHES WHEN A SELECTION IS MADE FROM THE MAIN MENU
* THE LAST ENTRY IN THE TABLE IS WHERE THE CODE BRANCHES WHEN
* FCTN-9 WAS STRUCK.  IN THIS CASE, WE EFFECTIVELY IGNORE THAT
* KEYSTROKE BY BRANCHING TO LABEL KYIN, WHICH SIMPLY WAITS FOR ANOTHER
* KEY TO BE STRUCK
MAINBR DATA NRIN,FILGET,SELCRD,SELCRS
       DATA FILSAV,NCIN,CRSLST,BYE,KYIN
* MISCELLANEOUS DATA ITEMS
NUMASK DATA >30
NOITEM DATA 0
SCRLI  BSS  SCRWID
OFFSET BYTE >60
SPACE  BYTE >20+>60
```

1.4. The Art Of Assembly — Part 4. Memory Saving Tips

By Bruce Harrison

Back in the first installment of this series, we made the bold assertion that Memory is your Master. On the TI, that becomes apparent whenever one tries to do a really big job on this computer. Our Word Processor, which we use to prepare these articles, fills nearly all of the TI's memory capacity. On many occasions in writing and refining that program, we did "scrubdowns" on the source code, trying to find places where we could accomplish the same function with fewer bytes. That was necessary to add new features to the program. It's not unusual in a program that size (about 150 pages of source code) that one can find places to save several hundred bytes. After a couple of scrubdowns, this gets tougher.

In this article we'll pass along some of the lessons learned in that experience, and hope your Assembly programs will benefit. We'll start with one small concrete example.

Let's assume you have a variable called CURSCR, which is going to keep track of what screen in VDP RAM you're currently looking at. Since there are less than ten screens possible, you decide to make that variable a single byte:

```
CURSCR      BYTE 0
```

That's fine until you discover that for many of its uses, you need to transfer that variable into a register to perform some action, and then need to transfer it back to the variable location. Look what that requires when we want the variable value in R4:

```
            CLR  R4                 Clear the register
            MOVB @CURSCR,R4         Move the byte in
            SWPB R4                 Right justify the byte
            (do some operation)
            SWPB R4                 Move value to left byte R4
            MOVB R4,@CURSCR         Put byte back at CURSCR
```

That's okay if you only do it at one place in the program, but if it's required at many places, the one byte you saved by making the variable a byte will cost you dearly. If it were a word in memory as CURSCR DATA 0, then the above code would read:

```
            MOV  @CURSCR,R4
            (do some operation)
            MOV  R4,@CURSCR
```

This takes six fewer bytes to perform than the previously shown code, because you skip the clear operation and you also skip the two SWPB operations. Yes, you could do the same thing as the first operation by:

```
        MOVB @CURSCR,R4
        SRL  R4,8
        (do some operation)
        SWPB R4
        MOVB R4,@CURSCR
```

But that still takes four more bytes than the operation would take by the second example above.

Let's look at another small example, from the Menu Driver we showed in Part 3 of this series. After the keystroke has been accepted, we did the following:

```
ACC1    MOV  R8,R5
        S    @NUMASK,R5
```

And so on until we branch to the address contained in R5. Actually, we needn't have moved the keystroke from R8 to R5, since we really made no other use of R8 in that section of code. Therefore we could eliminate the instruction MOV R8,R5 entirely, and just substitute R8 for R5 in the rest of that section of the source code.

That particular change will only save us two bytes of memory, but it would be part of a wider "scrubdown" effort, in which many bytes might be saved over the whole program.

Just for a moment, we'll digress into the subject of Macros. The TI Assembler doesn't make any provision for them, but we don't use that Assembler. We prefer using Art Green's RAG Assembler, which does provide a capability to use Macros. A Macro is a sort of second cousin to a subroutine, but instead of being located at one place in memory and called from many other places, a Macro simply replicates a section of code wherever it's invoked. Our small subroutine MOVBTS, for example, could be implemented as a Macro. We would save some overhead that way, since the main program wouldn't need the BL @MOVBTS instruction, which in itself uses four bytes.

Nevertheless, we don't recommend using Macros on the TI, because that will become a bad habit, and larger sections of code will be replicated over and over again in your programs, eating up valuable memory space. On the PC computer, we have resorted to using some very small Macros, to perform such functions as setting segment registers. Excessive use of Macros instead of subroutines is another reason that PC programs become overly large.

You may well ask why, then, do we use (and recommend) Art Green's RAG Assembler. That's simple. The RAG Assembler provides the best error reporting scheme of any Assembler we've seen. If, for example, you have an undefined symbol in your code, it tells you on-screen at which line of which file the erroneous label occurs, and shows you that line of source code. This makes it much easier to track down and correct source code errors.

Let's get off our soapbox now and get back to some serious business. There are many ways to save bytes in programs. We often find that savings can be made simply by changing the way we perform an operation. Here's an example from one of the subroutines in Part 2 of this series. Let's look at our screen clearing subroutine:

```
CLS         LI   R2,SCRWID          Sets R2 to characters in screen line
            LI   R5,>2000           Sets left byte R5 to space
            LI   R3,SCRLI           Point R3 at SCRLI
            MOV  R3,R1              Point R1 at SCRLI also
LOOP1       MOVB R5,*R3+            Move one byte and increment R3
            DEC  R2                 Decrement R2
            JNE  LOOP1              If not zero, repeat
            CLR  R0                 Point R0 to screen origin
            LI   R2,SCRWID          Set R2 again
            LI   R4,24              24 rows to clear
LOOP2       BLWP @VMBW              Write SCRWID bytes to screen
            A    R2,R0              add that many bytes to R0
            DEC  R4                 Decrement R4
            JNE  LOOP2              If not zero, repeat
            RT                      Return to calling program
```

The part at LOOP2 will serve as an example. We could have written that as:

```
LOOP2       BLWP @VMBW              Write SCRWID bytes to screen
            AI   R0,SCRWID          add SCRWID bytes to R0
            DEC  R4                 Decrement R4
            JNE  LOOP2              If not zero, repeat
            RT                      Return to calling program
```

That would work every bit as well, but since R2 already contains SCRWID while we're executing this loop, using the instruction A R2,R0 saves us two bytes. Similarly, in the section before LOOP1, we anticipated needing R1 pointed to SCRLI, so we moved R3 to R1, rather than having to LI R1,SCRLI. That also saves two bytes.

Moving or adding values from register to register rather than from immediate values should be the practice whenever possible. Such moves not only save memory, but also execute faster.

Another practice we encourage is maximizing use of the integer math operations. In our Menu Driver, for example, we wanted to double a number in the range of 0 through 7. We could have accomplished that this way:

```
            LI   R3,2               Place 2 in R3
            MPY  R3,R5              Multiply by the value in R3
            MOV  R6,R5              Put result back into R5
```

What we actually did was simply to SLA R5,1. This saves bytes and execution time. Whenever the range of possible outcomes is 0 through 32767 or less, doubling can be done in this manner. (Negative numbers can also be doubled this way.) There will of course be instances when the MPY instruction must be used, because the result will be too large to fit in one register, but every time one can use the shortcut SLA instruction, memory and time will both be saved.

Similarly, one can divide by two with a simple SRL or SRA instruction. In general, any time one needs to multiply or divide by an integral power of two (2, 4, 8, etc.), one should look and see whether the expected range of the outcome will permit shifting the number rather than using MPY or DIV to perform the operation.

There are of course exceptions to any rule. In our music programs, we perform timing of note durations using a loop. One of our customers disassembled our code, and told us that one operation in that loop could have been performed by a simple compare operation instead of the DIV that we used. He was correct, except that we used DIV on purpose to kill time in that loop. A compare operation would take far less time to execute, but then we'd have had to find some other way to waste time in the loop, otherwise our whole scheme for timing durations would need revision.

We said in our last installment that this one would include some right and wrong examples. Here's one. Let's suppose that you have a menu on the screen, and you wish to branch out to one of six labels (FUNCT1 through FUNCT6) from that menu. Assume for the moment that the key value in question is in R8. Here's the wrong way to do that branching:

```
            AI   R8,->31           Remove number mask plus 1
            JLT  OUTRNG            If lower than 0, key is out of range
            JGT  CMP1              If greater than 0, jump ahead
            B    @FUNCT1           Else function 1 chosen
CMP1        CI   R8,1              Has #2 been chosen?
            JGT  CMP2              If greater, jump ahead
            B    @FUNCT2           Else GOTO function 2
CMP2        CI   R8,2              Has #3 been chosen?
            JGT  CMP3              If greater, jump ahead
            B    @FUNCT3           Else GOTO function 3
CMP3        CI   R8,3              Has #4 been chosen?
            JGT  CMP4              If greater, jump ahead
            B    @FUNCT4           Else GOTO function 4
CMP4        CI   R8,4              Has #5 been chosen?
            JGT  CMP5              If greater, jump ahead
            B    @FUNCT5           Else GOTO function 5
CMP5        CI   R8,5              Function 6?
            JGT  OUTRNG            If greater, key is out of range
            B    @FUNCT6           Else perform function 6
OUTRNG      (IGNORE THE KEYSTROKE)
```

Now here's the right way to do it. Start by putting a lookup table in the data section like this:

```
LUT         DATA FUNCT1,FUNCT2,FUNCT3 ... ,FUNCT6
```

Now the branching can be done like this:

```
            AI   R8,->31            Remove number mask plus 1
            JLT  OUTRNG             If result less than zero, jump
            CI   R8,5               Is number greater than 5?
            JGT  OUTRNG             If so, jump
            SLA  R8,1               Else double the number
            MOV  @LUT(R8),R5        Put selected address in R5
            B    *R5                Branch to the address in R5
OUTRNG      (IGNORE THE KEYSTROKE)
```

This takes many fewer bytes than the code shown above as the wrong way. We'll leave calculation of how many bytes fewer as an exercise for the student. On a casual first look, the twelve bytes used by the lookup table might seem wasteful, but overall we have a significant saving by "spending" those twelve bytes. This is similar to an Extended Basic situation in which a chain of IF THEN statements is replaced by an ON GOTO. That saves both bytes and execution time in XB, just as this "right" way does in Assembly.

This method will work very nicely when there's only one menu in your program. See the source code given in Part 3 for an efficient way to handle more than one menu.

One more method of memory saving we should mention, and that's what we'll call recycling. (Recycling is a fashionable term nowadays.) A small example or two should give you the idea. In our Golf Score Analyzer, we have a part of the data section of our code devoted to the copyright notice. It looks like this:

```
CPYRT       BYTE 14                 length of first line
            TEXT 'Copyright 1991'
            BYTE 17                 length of second line
            TEXT 'Harrison Software'
```

Altogether that takes up 33 bytes. It's used only once, at the very beginning of the program, then becomes wasted memory space. At later stages of the program, we needed an area of 56 bytes length to store a temporary record of a round of golf. Ordinarily one would give that a label and a BSS like:

```
TEMREC      BSS  56
```

But by placing that label just before the copyright notice, we could make it:

```
TEMREC      BSS  23
```

Thus the rest of the 56 bytes in TEMREC overwrites the copyright notice, which we don't need anymore.

In a pinch, we could also use some of the area filled by the code that places that copyright notice on the screen as data storage. We didn't do that in this instance, but we did recycle the area in memory where the Extended Basic LOAD program is loaded to store user data about courses played. Thus when our program ends, the original Extended Basic program which got our Assembly program going has been destroyed. (When exiting our program, we take steps to insure that Extended Basic will "know" that it has no program in memory.)

One final option for dealing with the shortage of memory is the use of program overlays, where a new section of program is brought in from disk and written over an existing section of code or data. We resorted to that method for some utility features in our Word Processor, but we'll save that rather complex topic for a later article in this series.

We hope these few examples will serve to inspire you in finding ways to save memory in your own programs. In our next article, we plan to include more subroutines that will be directly usable in your programs.

1.5. The Art Of Assembly — Part 5. Useful Subroutines

By Bruce Harrison

This month's article will be relatively short, but it's accompanied by a large dose of source code (see Sidebar). The source code for today is all subroutines, one of the High level variety (a subroutine that calls other subroutines) and several smaller ones.

The major purpose in this source code is to get user input from the keyboard, display it stroke by stroke on the screen, then when the **ENTER** key is pressed, to report out what's on the screen into a string at one specific location in memory. In effect, this is like the Extended Basic ACCEPT AT function for a string variable. The version shown was developed for use in our Golf Score Analyzer program. In this listing, however, we've left out the lines that deal with the character offset for Extended Basic. Thus this subroutine can be easily integrated into any E/A Option 3 type program. The label names used reflect its "Golf" origins to some extent, as the name of the big subroutine CRSIN, short for Course Name Input. In that program, this was actually used for any occasion when we wanted to accept a string of characters from the keyboard.

There is an auxiliary subroutine which we call CLRFLD (clear field) also included in the Sidebar. That is used before CRSIN, to clear the screen area into which we want user input. One can also use CRSIN without the CLRFLD, so that something already in that screen location can be edited or accepted as a default entry.

Let's say that we want to accept a 20 character string with a cleared field at Row 12, column 5 of the display screen. Here's what the main program would need to do to invoke the subroutines:

```
LI   R0,SCRWID*11+4     Set R0 to Row 12, col 5
LI   R4,20              Number of characters in R4
BL   @CLRFLD            Clear 20 characters at row 12 col 5
BL   @CRSIN             Accept the input string
```

Note that the subroutine CLRFLD restores the original value in R0 and retains the value in R4 upon exit, so the main program need not reload those two registers before calling CRSIN.

Also please note that this subroutine will not work if R0 is zero. If it's set to a value of 1, the accept will happen at Row 1, Column 2 of the screen. The adept student may modify it so it would work at the screen origin, but we've never found it necessary (or desirable) to accept a string at that screen position.

Before we get further into how this subroutine CRSIN works, we'd better deal again with that business of stacking the return address for this High level case. What's shown here assumes that your program contains other High level subroutines and that somewhere early in the program you'd pointed R15 at a stack location in memory. If this were the only high level subroutine in your program, you could simply stash R11 in R15 itself, so the opening line in CRSIN would read:

```
CRSIN       MOV  R11,R15
```

And the exit point would be:

```
CRIX        B    *R15               Branch to the address in R15
```

The other possible case is that you'd have CRSIN as the first High level subroutine in your program, in which case CRIX would be a label only, and would be followed by the short piece of code shown at label SUBRET.

The subroutine CRSIN uses three others to do its work. For normal keystroke inputs, it uses CURFRC to put the cursor on-screen, then uses KI2 to accept your keystroke into R8. When the input keystroke is one of the two "arrow" keys **FCTN S** or **FCTN D**, the special repeat-key subroutine KI2A is used. Using that subroutine allows the cursor to be moved through the input field by holding down the arrow key. There is a built-in delay in this subroutine, so the cursor will not fly to the end of the field, but move in human-speed steps. The subroutine exits immediately if you release the key. The delay imposed is modified by the subroutine, so the delay after the first cursor move is considerably less than the first move. Moving the byte at location ONE to location KI2A+2 clears the left byte of the immediate value that follows the label KI2A. When you exit by releasing the arrow key, the main subroutine resets the delay factor for a first arrow move.

This idea of having the code modify itself while you're using it is tricky, and many programmers shun its use. We considered it a worthwhile thing to do in this instance, to make the movement of the cursor more like what the TI user is accustomed to seeing.

Now let's start at the beginning of the subroutine. Some important things happen there. On entry, after stashing the return address, we clear our insert flag, so that we're sure insert mode won't be on when we didn't ask for it.

Next, we stash the starting value of R0, then move back one location and place an edge character on the screen. We then increment R0, add the length of the allowed string to it, and write another edge character. They are put there so our subroutine will easily be able to distinguish the two ends of the allowed input field. We also save this position of R0 (one beyond the last character to be accepted) for use later on. When operating in most modes, the edge character looks just like a space. This is not true when entering from E/A Option 3, in which case the edge character is a small square. You can redefine it to look like a space by:

```
            LI   R0,32*8+>800       Point at space character
            LI   R1,TEMSTR          Use our temporary string buffer
            LI   R2,8               Eight bytes to read
            BLWP @VMBR              Read eight bytes from space
            S    R2,R0              Back up to edge character
            BLWP @VMBW              Write eight bytes
```

Finally in this opening section, we subtract R4 from R0 so we're at the first character spot in the field, then stash away the value in R4 for use later.

The section of code starting at CRSI0A is the main operating loop of this subroutine. The first order of business is to grab the character present at this spot on the screen and stash that at location ALTKEY. This will become the character that alternates with the cursor while the cursor is at this position.

The very next thing is to call the little subroutine CURFRC. CURFRC is there so that every time the cursor moves to a new input location, the cursor will appear on-screen, and start a new cycle of blinking. Were this not done, the cursor could become invisible after some of your keystrokes, and we find that disconcerting. Now we call the subroutine KI2 which simply keeps blinking the cursor, alternating with whatever character was there before, until you strike a key on the keyboard.

There are some checks now performed on the value of the keystroke reported into R8 by KI2. The only one of these that's not immediately obvious is the check for the value 15. That's the ASCII code for **FCTN 9**, and behaves the same as if **ENTER** were struck. In its application within the Golf Score Analyzer, the key combination **FCTN 9** gets you back to the part of the program which called CRSIN, which then uses the fact that you exited CRSIN by **FCTN 9** to escape gracefully from whatever function you were into. If you don't need that feature, you can omit the two lines CI R8,15 and JEQ CRSDMY.

We should at this point admit that this source code has not been subjected to a thorough "scrubdown" effort. The two lines following that compare to 15 and its jump instruction may be unnecessary. We're not going to stop and make that change in the program, but will leave as an exercise for the student the determination. As it is, the subroutine does work, even if it does contain a piece of sloppy coding. Your author is human, like you.

There's another piece of inelegant code in here, concerning label CRSDMY. That stands for DUMMY! During the development of this subroutine, we got into the situation where some of our jumps to label CRSIX were out of range. We could have corrected that situation by adding labels, reversing logic, and including some B @CRSIX instructions. Instead, we wedged in that phony label CRSDMY, which simply makes a second jump to CRSIX. This is really not the soundest practice, but it's a quick, cheap, and ugly way out of a problem. We're not proud of it, but it does assemble and work correctly, so we're leaving it alone. Whenever your author starts to get too elegant with his programming, he remembers a lesson taught by his first mentor in programming the TI, a man named George R. Hendershot. The lesson was "First, get it to work!" One might add a corollary to that, such as "If it ain't broke, don't fix it!"

At label CRSC4, we see whether the insert key **FCTN 2** has been struck. If it hasn't, we move on, and if it has, we set the insert flag (INSFLG) and go back to CRSI0. Once the insert key has been struck, characters entered from the keyboard will be inserted at the current cursor position until insert is canceled by hitting the arrow keys, **FCTN 9**, or **ENTER**.

The next important keystroke the program looks for is **ENTER**. If that's been struck, we exit the subroutine. Given it's not the **ENTER** key, we check for **FCTN 1**. If that's been struck, we delete the character at the current cursor position and move all the characters right of that position in the field one spot left. Next there's one final check to see if some other key with an ASCII code less than the spacebar's 32 has been struck. If so, we ignore that keystroke.

Next there's a short section that converts lower case characters to upper case. This may be omitted if you don't need it.

At label CRSI1, we check to see whether the insert flag is set by moving that word into R1 and jumping ahead if the word was zero. If insert was in effect, we perform the steps between JEQ CRSI1A and the label CRSI1A. First, we write the character that was at the cursor position to the screen, then move our variable word ENDOC into R2 and subtract R0 from it. This makes R2 equal the number of characters between the current cursor position and the edge marker at the end of the field. Now we use TEMSTR, which will be the location for the string input when we're finished, as a temporary buffer to hold all the characters from the cursor's position to the end of the field. We then DEC R2, so that the writing back of these characters will not extend to the edge character. If R2 has become zero, that means we're at the last position in the field, so we skip ahead. Now, we increment R0 so we're writing to the next screen spot, and perform a BLWP @VMBW to write the characters back to the screen one space to the right. Finally we decrement R0 so it points to where it was when we started this section of code, and then proceed at label CRSI1A to write the struck key's character to the screen.

Had we not been in insert, we would have jumped to here and put the character on the screen. After writing one character, we increment R0 so it points at the next spot, check to see if the character we've reached is an edge character, and jump back if it is, so we don't exceed the field limit.

The rest is pretty mundane stuff, simply handling the movement of the cursor in response to the arrow keys, so we'll skip ahead to CRSIX, where this string of characters gets "reported out" to the label TEMSTR.

The first order of business is to write back the ALTKEY character to the screen, then set R0 to point at the last spot in the field. Next, we get the field length from location SAV4 into R2. We now start examining the characters in the field in reverse order, looking for a non-space character, and decrementing the count in R2 each time we find a space. This eliminates trailing spaces from the length of the reported string. Once we've found a non-space, we have the length of the string in R2, so we swap the bytes in R2, place the length byte at location TEMSTR, re-swap so R2 has the length as a word value. At this point we check to see if a null string (all spaces) is in the field and get out of here if that's so. Otherwise we set R1 to point to TEMSTR+1, and read the string's content from the screen via a BLWP @VMBR.

When we finish, TEMSTR contains one byte at the beginning to indicate length of the string, plus the string's content. From here, the main program can take the string at TEMSTR and move it to the desired memory location via the small subroutine MOVSTR, which was included in Part 2 of this series.

As the saying goes, use it in good health. This subroutine can make your life a bit easier when you are writing a program. If it does that, in addition to adding to your knowledge of Assembly programming, then it's been worth the effort.

In our next article, we'll discuss, among other topics, the business of entering and returning gracefully from programs. We'll also discuss some of the ramifications of working with Assembly programs started from Extended Basic.

```
* SUBROUTINES WHICH MAY PROVE USEFUL
* DESIGNED FOR USE IN E/A OPTION 3 PROGRAMS
* CODE BY BRUCE HARRISON - PUBLIC DOMAIN
* 22 JUNE 1991
*
* REQUIRED REFERENCES
      REF  KSCAN,VMBW,VMBR,VSBW,VSBR
*
* REQUIRED EQUATES
STATUS EQU  >837C
KEYADR EQU  >8374
KEYVAL EQU  >8375
*
* THE FOLLOWING SUBROUTINE ACCEPTS A STRING OF CHARACTERS STARTING AT LOCATION
* POINTED TO BY R0, NUMBER OF CHARACTERS TO ACCEPT MUST BE IN R4
* INPUT STRING IS PLACED AT LOCATION TEMSTR
*
CRSIN
       MOV  R11,*R15+    STACK RETURN ADDRESS
       CLR  @INSFLG      CLEAR OUR INSERT FLAG
       MOV  R0,@PGNUM    STASH R0 IN MEMORY LOCATION
       DEC  R0           DECREMENT R0
       MOVB @EDGE,R1     PLACE EDGE CHARACTER IN LEFT BYTE R1
       BLWP @VSBW        WRITE EDGE CHARACTER TO SCREEN
       INC  R0           RESET R0 TO ORIGINAL VALUE
       A    R4,R0        ADD NUMBER OF CHARACTERS TO ACCEPT
       BLWP @VSBW        WRITE AN EDGE CHARACTER TO SPOT BEYOND FIELD
       MOV  R0,@ENDOC    SAVE THIS LOCATION IN MEMORY
       S    R4,R0        RESET R0 TO ORIGINAL VALUE
       MOV  R4,@SAV4     STASH R4 IN MEMORY
CRSI0A BLWP @VSBR        READ THE CHARACTER POINTED TO BY R0
       MOVB R1,@ALTKEY   STASH THAT CHARACTER AT LOCATION ALTKEY
CRSI0  BL   @CURFRC      FORCE THE CURSOR ONTO THE SCREEN
       BL   @KI2         USE THE SCANNING SUBROUTINE WITH FLASHING CURSOR
       CI   R8,9         HAS RIGHT ARROW BEEN STRUCK?
       JEQ  CRSRT        IF SO, JUMP
       CI   R8,8         HAS LEFT ARROW BEEN STRUCK?
       JEQ  CRSBK        IF SO, JUMP
       CI   R8,10        DOWN ARROW?
       JLT  CRSC4        IF LESS, JUMP
       CI   R8,15        HAS FCTN-9 BEEN STRUCK?
```

```
       JEQ  CRSDMY       IF SO, JUMP
       CI   R8,13        HAS ENTER KEY BEEN STRUCK?
       JLT  CRSDMY       IF LESS, JUMP
CRSC4  CI   R8,4         HAS FCTN-2 (INSERT) BEEN STRUCK?
       JNE  CRSENT       IF NOT, JUMP
       INC  @INSFLG      ELSE SET INSERT FLAG
       JMP  CRSI0        THEN JUMP BACK
CRSENT CB   @KEYVAL,@ENTERV HAS ENTER BEEN STRUCK?
       JEQ  CRSDMY       IF SO, JUMP
       CI   R8,3         HAS FCTN-1 (DELETE) BEEN STRUCK?
       JEQ  CRSDEL       IF SO, JUMP
       CI   R8,32        SPACE BAR
       JLT  CRSI0        IF LESS, JUMP
* THE FOLLOWING FIVE LINES ARE NEEDED ONLY IF ONE WANTS LOWER CASE
* CHARACTERS CONVERTED TO UPPER CASE.  IF NOT, OMIT THESE FIVE LINES
       CI   R8,122       COMPARE TO LOWER CASE Z
       JGT  CRSI0        IF GREATER, JUMP
       CI   R8,97        COMPARE TO LOWER CASE A
       JLT  CRSI1        IF LOWER, JUMP
       SB   @ANYKEY,@KEYVAL ELSE SUBTRACT >20 FROM KEYSTROKE
CRSI1
       MOV  @INSFLG,R1   TEST IF INSERT FLAG ON
       JEQ  CRSI1A       IF NOT, JUMP
       MOVB @ALTKEY,R1   ELSE WRITE CURRENT CHARACTER
       BLWP @VSBW        TO CURRENT SCREEN POSITION
       MOV  @ENDOC,R2    MOVE LIMIT ADDRESS INTO R2
       S    R0,R2        SUBTRACT CURRENT R0 POSITION
       LI   R1,TEMSTR    POINT TO TEMSTR LOCATION
       BLWP @VMBR        READ CHARACTERS FROM SCREEN
       DEC  R2           DECREMENT CHARACTER COUNT
       JEQ  CRSI1A       IF R2 IS ZERO, NO INSERT - WE'RE AT LAST POSITION
       INC  R0           INCREMENT SCREEN POSITION
       BLWP @VMBW        WRITE CHARACTERS BACK
       DEC  R0           POINT BACK ONE SPOT
CRSI1A MOVB @KEYVAL,R1   MOVE THE KEY STRUCK INTO LEFT BYTE R1
       BLWP @VSBW        WRITE KEY VALUE TO SCREEN
       INC  R0           POINT AT NEXT CHARACTER POSITION
       BLWP @VSBR        READ CHARACTER THAT'S THERE
       CB   R1,@EDGE     IS THIS AN EDGE CHARACTER?
       JNE  CRSI0A       IF NOT, JUMP
       DEC  R0           ELSE BACK UP ONE CHARACTER
       JMP  CRSI0A       THEN BACK FOR ANOTHER KEY INPUT
CRSRT  MOVB @ALTKEY,R1   TAKE CURRENT SCREEN CHARACTER INTO LEFT BYTE R1
       BLWP @VSBW        WRITE CHARACTER TO SCREEN
       CLR  @INSFLG      CLEAR THE INSERT FLAG
       INC  R0           MOVE TO NEXT SPOT
       BLWP @VSBR        READ THE CHARACTER THERE
       CB   R1,@EDGE     IS THAT EDGE CHARACTER?
       JEQ  CRSRT1       IF SO, JUMP
       MOVB R1,@ALTKEY   ELSE STASH CURRENT SCREEN CHARACTER
```

```
       BL   @CURFRC      FORCE CURSOR ONTO SCREEN
       BL   @KI2A        GO SCAN KEYBOARD
       CB   @KEYVAL,@RITEV IS RIGHT ARROW STILL HELD DOWN?
       JEQ  CRSRT        IF SO, KEEP GOING RIGHT
       CB   @KEYVAL,@NOKEY HAS NO KEY BEEN STRUCK?
       JEQ  CRSRT2       IF SO, JUMP
CRSRT1 DEC  R0           BACK TO PREVIOUS SPOT
CRSRT2 MOVB @ONOFF,@KI2A+2 RESTORE DELAY CONSTANT
       MOVB @ALTKEY,R1   GET CHARACTER INTO LEFT BYTE R1
       BLWP @VSBW        WRITE TO SCREEN
       JMP  CRSI0        THEN JUMP BACK FOR ANOTHER KEY
CRSBK  MOVB @ALTKEY,R1   GET CURRENT CHARACTER IN R1
       BLWP @VSBW        WRITE TO SCREEN
       CLR  @INSFLG      CLEAR INSERT FLAG
       DEC  R0           BACK ONE SPOT
       BLWP @VSBR        READ CHARACTER FROM SCREEN
       CB   R1,@EDGE     IS THAT EDGE CHARACTER?
       JEQ  CRSBK1       IF SO, JUMP
       MOVB R1,@ALTKEY   ELSE STASH CHARACTER AT ALTKEY
       BL   @CURFRC      FORCE CURSOR ONTO SCREEN
       BL   @KI2A        GO GET KEYSTROKE
       CB   @KEYVAL,@LEFTV  IS LEFT ARROW STILL HELD DOWN?
       JEQ  CRSBK        IF SO, GO BACK AGAIN
       CB   @KEYVAL,@NOKEY HAS NO KEY BEEN STRUCK
       JEQ  CRSRT2       IF SO, JUMP
CRSBK1 INC  R0           MOVE TO NEXT SPOT
       JMP  CRSRT2       THEN JUMP
CRSDMY JMP  CRSIX        THIS IS A DUMMY JUMP TO KEEP JUMPS IN RANGE
CRSDEL MOV  R0,R7        STASH R0 IN R7
       CLR  @INSFLG      CLEAR INSERT FLAG, SINCE WE'RE DELETING
       MOV  @ENDOC,R2    END OF FIELD ADDRESS IN R2
       S    R0,R2        SUBTRACT CURRENT CHARACTER ADDRESS
       INC  R0           POINT TO NEXT CHARACTER
       DEC  R2           DECREMENT R2 COUNT
       JEQ  CRSD1        IF R2 ZERO, PRINT SPACE - WERE AT LAST POSITION
       LI   R1,TEMSTR    POINT R1 AT TEMSTR FOR TEMPORARY STORAGE
       BLWP @VMBR        READ CHARACTERS INTO LOCATION TEMSTR
       MOV  R7,R0        PUT BACK R0
       BLWP @VMBW        WRITE CHARACTERS FROM TEMSTR TO SCREEN
CRSD1  MOVB @ANYKEY,R1   PUT A SPACE IN LEFT BYTE R1
       MOV  @ENDOC,R0    GET LIMIT SPOT INTO R0
       DEC  R0           DECREMENT BY ONE
       BLWP @VSBW        WRITE A SPACE TO SPOT JUST BEFORE LIMIT
       MOV  R7,R0        GET R0 BACK AGAIN
CRSD0  B    @CRSI0A      BRANCH BACK TO BEGINNING
CRSIX  MOVB @ALTKEY,R1   WRITE CURRENT CHARACTER TO SCREEN
       BLWP @VSBW
       MOV  @ENDOC,R0    SET LIMIT POSITION IN R0
       DEC  R0           DECREMENT BY ONE
       MOV  @SAV4,R2     MOVE MAX NUMBER OF CHARACTERS INTO R2
```

```
CRSIX1 BLWP @VSBR        READ THE CHARACTER AT CURRENT R0 POSITION
       CB   R1,@ANYKEY   IS THAT A SPACE?
       JNE  CRSIXX       IF NOT, WE'VE REACHED CONTENT OF STRING
       DEC  R0           ELSE MOVE BACK ONE SPOT
       DEC  R2           DECREASE CHARACTER COUNT BY ONE
       JGT  CRSIX1       IF GREATER THAN ZERO, JUMP BACK
CRSIXX MOV  @PGNUM,R0    GET ORIGINAL R0 POSITION BACK
       SWPB R2           PUT CHARACTER COUNT IN LEFT BYTE R2
       MOVB R2,@TEMSTR   PLACE THAT AT TEMSTR
       SWPB R2           REVERSE R2 AGAIN
       JEQ  CRIX         IF R2=0, JUMP
       LI   R1,TEMSTR+1  ELSE SET R1 TO POINT TO STRING CONTENT STORAGE
CRSIX2 BLWP @VMBR        READ THE STRING FROM THE SCREEN
CRIX   B    @SUBRET      RETURN FROM THIS SUBROUTINE
*
* SUBRET IS SHOWN HERE FOR REFERENCE.  NORMALLY IT'S MADE A PART OF THE FIRST
* HIGH-LEVEL SUBROUTINE USED IN THE PROGRAM
*
SUBRET DECT R15
       MOV  *R15,R11
       RT
*
* THE FOLLOWING SUBROUTINE GETS KEYSTROKES FROM THE KEYBOARD WHILE ALTERNATING
* THE CURSOR WITH A CHARACTER STASHED AT ALTKEY
* THE LINES LIMI 2 AND LIMI 0 ALLOW THE SENSING OF FCTN-QUIT AND ALSO ALLOW
* A BEEP VIA GPLLNK TO OPERATE PROPERLY
*
KI2    CLR  @STATUS      KEY-IN WITH ALTERNATING
       BLWP @KSCAN       CHARACTER AND CURSOR
       LIMI 2            ACTIVATE INTERRUPTS
       LIMI 0            SHUT OFF INTERRUPTS
       DEC  R4           ENTER AFTER R4 SET TO >0200
       JEQ  CHNG         AND R1 TO >1E00 AND VSBW
       CB   @ANYKEY,@STATUS HAS A KEY BEEN STRUCK?
       JNE  KI2          IF NOT, RE-SCAN KEYBOARD
       MOV  @KEYADR,R8   ELSE PUT KEY'S VALUE IN R8
       RT                THEN RETURN
CHNG   CI   R1,>1E00     IS R1 SET TO CURSOR CHARACTER?
       JEQ  L1           IF SO, JUMP
       LI   R1,>1E00     ELSE SET LEFT BYTE R1 TO CURSOR
       BLWP @VSBW        WRITE CURSOR TO SCREEN
       MOVB @ONOFF,R4    PLACE TIMING IN LEFT BYTE R4
       JMP  KI2          GO BACK TO SCANNING KEYBOARD
L1     MOVB @ALTKEY,R1   PLACE ALTERNATING CHARACTER IN LEFT BYTE R1
       MOVB @ONOFF+1,R4  PLACE ALTERNATE DELAY IN LEFT BYTE R4
       BLWP @VSBW        WRITE CHARACTER TO SCREEN
       JMP  KI2          GO BACK TO SCANNING KEYBOARD
*
* THE FOLLOWING IS A SPECIAL KEY INPUT FOR REPEATING OPERATION OF
* THE RIGHT AND LEFT ARROW KEYS
```

```
* THIS SUBROUTINE INCLUDES SELF-MODIFYING CODE
*
KI2A   LI   R5,>0280     LOAD R5 WITH DELAY FACTOR
KI2B   CLR  @STATUS      CLEAR GPL STATUS
       BLWP @KSCAN       SCAN KEYBOARD
       CB   @KEYVAL,@NOKEY  HAS NO KEY BEEN STRUCK?
       JEQ  KI2C         IF SO, JUMP
       LIMI 2            SET INTERRUPTS ON
       LIMI 0            SET INTERRUPTS OFF
       DEC  R5           DECREMENT DELAY COUNTER
       JNE  KI2B         IF NOT ZERO, SCAN AGAIN
       MOVB @ONE,@KI2A+2 ELSE MODIFY DELAY COUNT
KI2C   RT                THEN RETURN
*
* THE FOLLOWING SUBROUTINE FORCES THE CURSOR CHARACTER ONTO THE SCREEN
*
CURFRC LI   R1,>1E00     PUT CURSOR CHARACTER IN LEFT BYTE R1
       LI   R4,>0100     SET DELAY FACTOR IN R4
       BLWP @VSBW        WRITE CURSOR TO SCREEN
       RT                RETURN
*
* FOLLOWING SUBROUTINE CLEARS AN INPUT FIELD
* BEGINNING AT R0 POSITION, EXTENDING NUMBER OF CHARACTERS IN R4
*
CLRFLD
       MOV  R4,R2        PLACE VALUE OF R4 IN R2
       MOV  R0,R3        SAVE R0
       MOVB @ANYKEY,R1   PUT SPACE CHARACTER IN LEFT BYTE OF R1
CLRFL1 BLWP @VSBW        WRITE ONE SPACE IN FIELD
       INC  R0           POINT TO NEXT CHARACTER SPOT
       DEC  R2           DECREMENT COUNT OF SPACES
       JNE  CLRFL1       IF NOT ZERO, REPEAT WRITING OPERATION
       MOV  R3,R0        REPLACE ORIGINAL VALUE OF R0
       RT                RETURN
*
* REQUIRED DATA SECTION
* THE FOLLOWING DATA SOURCE LINES ARE REQUIRED BY THESE SUBROUTINES
*
ONE    DATA 1
ENDOC  DATA 0
INSFLG DATA 0
PGNUM  DATA 0
SAV4   DATA 0
ONOFF  DATA >0201
EDGE   BYTE >1F
ANYKEY BYTE >20
NOKEY  BYTE >FF
ALTKEY BYTE 0
ENTERV BYTE 13
RITEV  BYTE 9
```

```
LEFTV  BYTE 8
TEMSTR BSS  41
* THE NUMBER IN THIS BSS MUST BE ONE MORE THAN THE LARGEST STRING LENGTH
* EXPECTED IN THE PROGRAM'S EXECUTION
```

1.6. The Art Of Assembly — Part 6. The Ins And Outs

By Bruce Harrison

As we promised, this part of our series will deal primarily with getting into and out of your Assembly program gracefully. We consider this an important topic, since it can make all the difference when you're writing entire programs in Assembly language. In one book that we used while trying to learn Assembly, a small program example was shown, but there was no way out of the program once it started except the On-Off switch. That shouldn't be.

TI's E/A book gives several ways of returning from programs, but we don't use any of them. Instead, there are two methods that we've used, each of which gets you back where you came into the program from. If you entered from E/A Option 3, we'll return you to that screen that says "PRESS ENTER TO CONTINUE" at the bottom. If you entered from XB, we'll send you back to XB with the * READY * and prompt on the screen.

The means of entering a program may vary all over the place, from the very simple LWPI WS to a section of code that re-arranges the locations of tables in the VDP RAM, and to even more exotic openings. All the openings have that one thing in common, setting the workspace registers to a workspace of our choosing. As we explained earlier our usual choice is to set the workspace at >20BA, which TI set aside for us to use. This can be used even when programs start from XB, so long as the program only returns to XB upon exit. Utility subroutines for use in XB programs via CALL LINK should always have a self contained workspace. We have found, for example, that the NUMASG utility will corrupt the workspace at >20BA. After returning to XB from an Assembly routine that uses NUMASG and the >20BA workspace, the XB program will break with an error.

In today's Source Code (see Sidebar) there are two separate programs, with different entry and exit methods used. Program one is of course not complete, since it needs the subroutine CRSIN and its supporting smaller subroutines given in our last article. You can combine that code with this "shell" and assemble it. When you do that combination, you'll have to delete the line of REFs and the equate for STATUS, from the subroutine's code, and the line at label TEMSTR from the subroutine's Data Section. The resulting program will serve to demonstrate the subroutine. It will also illustrate the simplest possible entry and exit for your own programs. The entry simply sets the workspace pointer, then goes about its business. The exit uses a trick passed along to us by Harry Wilhelm. We set the workspace pointer back to GPLWS, clear the status byte, then B @>006A.

That exit method will work whether you entered from E/A or Extended Basic. It may not be necessary to clear the STATUS, but the only way to find out in any particular program is to run it and see whether an error is reported when you exit. If no error is reported, then you can omit CLR @STATUS from this exit.

Our normal practice is to leave that line in, just to be on the safe side. We don't like seeing error reports on the screen, and we're too lazy to go look up their meanings in the appropriate book.

The second program uses a slightly more exotic way of entering and leaving. At the opening, it stashes away the value from R11 of whatever workspace the computer was using, then restores that to R11 of the GPL workspace before doing an RT. Early in our experiences with the TI Assembly language, we discovered that when you enter your program, the computer has essentially performed a BL operation to get into your program, so register 11 contains the return address you can use to exit. There are exceptions to this when you entered from Extended Basic, and this method from Program 2 will not always work for XB entry. The first method (B @>006A) will always work, provided only that you first load the workspace pointer with the GPL workspace (>83E0).

That brings us to a very minor point, but one that might be important in some of your programming efforts. In our music programs, we discovered that, for some reason we've not discovered, if one does NOT move R11 to someplace on entry, as in Program 2, the sending of bytes directly to the sound generator at >8400 will not work properly. We have no idea why that's so, or whether other functions might be affected, but in our music programs we use the entry method of Program 2 and the exit method of Program 1. That keeps everything working.

Both Programs are set up to be entered from E/A Option 3. In the first one, we included the code to define the edge character to look like a space, then proceeded to set up for and call our subroutine. Note that there is no screen-clearing operation here. Since this program does not auto-start, but requires you to type in the program name START at the PROGRAM NAME prompt, the screen will be cleared and set to light green for you.

After the subroutine has finished, the program takes the string just placed in TEMSTR and displays it a few lines down the screen. It then calls the subroutine again. This is done simply to give you a chance to see that the subroutine did what was intended. Pressing **ENTER** will get you out of the program and back to the E/A prompt PRESS ENTER TO CONTINUE.

The second program is intended for you to use as a small utility. We wrote this originally for our own use, because many times when we were operating with the E/A module in place, we wanted to print a source code file, but wanted a way to set the printer to skip over the perforations while printing. Before we had a Ramdisk, we kept this program on a disk with EDIT1 and the ASSM1, ASSM2 files. Now, we keep it available all the time on a Ramdisk.

This program does nothing fancy. When it loads from Option 3, it auto-starts and runs the part starting at label SKIPIT. This sets our printer to skip over some lines at the bottom of each sheet. That happens very quickly, so you may not even see the light blink on the RS-232 card. You'll also not see anything happen at the printer, since we've opened the file to the printer with the .CR option, so no line feed or carriage return will go to the printer unless we intend to send one.

The program will do its job and simply return to E/A, which will place you back at the FILE NAME prompt. If all you wanted to do was set up for skip-over, press **FCTN 9** to get out to the main E/A menu. This small program, however, has another entry point called DOUBLE. If you also want double strike printing, press **ENTER** at the FILE NAME prompt, then type in DOUBLE **ENTER** at the PROGRAM NAME prompt. This will send another three characters to your printer, putting it in double strike and sending a harmless carriage return. That carriage return is sent only so that each thing sent to the printer by this program will contain three characters. If the carriage return were not there, the 10 from the previous three character string would still be present in the VDP RAM buffer, would be sent to the printer, and would cause an unwanted line feed to occur.

The escape sequences we've put into this program will work for all models of Epson, Star Micronics, and Panasonic printers. The number of lines to skip (third byte at label PRNBYT) is ten for us, because of the way we usually have our printer's paper loaded. You may want to change that number to something less, say 5 or 6, before assembling the program. If your printer is some other make, such as an Okidata, you may need to change the escape sequences in other ways. I've run into one printer, called the Olivetti ink jet, in which sending a line feed or a carriage return, or both in either order, will always result in both a carriage return and line feed being performed. Most printers have a DIP switch setting to prevent added line feeds, but not the Olivetti.

This program incidentally introduces the new (for these articles) topic of file management. It Opens, Writes to, then Closes a file. As we've noted in the source code's annotation, there are some shortcuts we've taken here which would not generally be used in file operations. This program does, however, work nicely for its intended purpose. In a later article, we'll get deeper into file accesses, and avoid the shortcuts that were used in this program.

We promised some discussion on the ramifications of using Assembly programs that run from Extended Basic. One could nearly write a book on this topic alone. One of the big problems is this business of the character offset (>60) that one must use when operating from XB.

Strangely enough, it is possible to avoid that offset in XB. In our Word Processor, which was originally designed to run only under E/A Option 3, we avoided needing the offset by switching to the text mode and loading our character definitions starting at >800, where they are located normally when using the E/A module. To do that, we had to perform some VWTR operations, so that VDP would know where its tables were located. This operation is performed not by the Word Processor itself, but by the loader program's Assembly portion, embedded in the Extended Basic LOAD program.

Let's digress into that subject just a bit. The actual Word Processing program is stored on the disk as a series of memory-image files. There are two loaders included in the program disk, one named LOAD, which runs from Extended Basic, and one called UTIL1, which is an E/A Option 5 program file.

Both these loaders contain code to put the VDP into the required setup for the TEXT mode, place a PLEASE STAND BY message on the screen, then load in the five memory image files that comprise the actual Word Processor.

In the Assembly part of the XB LOAD program, we set up to avoid the need for offset by performing the following:

```
            LI   R4,32*8+>800       Location of space character
            MOV  R4,@>834A          Move that value to FAC location
            BLWP @GPLLNK            Use GPL Linkage
            DATA >0018              Load "Small Capitals" characters
TEXMO       LI   R0,>01F0           Setup for text mode
            BLWP @VWTR              Place VDP in text mode
            LI   R0,>074E           Setup for screen colors
            BLWP @VWTR              Write screen colors for text mode
            LI   R0,>0401           Relocate character table to >800
            BLWP @VWTR              By writing to VDP register 4
            MOVB @TEXMO+3,@>83D4    Stash the text mode byte
```

This last operation, putting the byte at TEXMO+3 at >83D4, is necessary because otherwise the computer will go back to graphics mode as soon as any keystroke is accepted.

Of course the LOAD program does a host of other operations, but these are the key ones. The next-to last two lines tell VDP to look for its character definitions at >800, and this allows us to perform reading and writing of screen characters without that nagging offset. Those who've done work involving Assembly and XB will notice that we've done a BLWP @GPLLNK. XB does not supply such a link vector. Our LOAD program supplies one of those, as well as a DSRLNK. The utility vectors (GPLLNK & DSRLNK) we use are those written and published some time ago by Craig Miller.

That leads into another topic, the use of utility vectors and routines. If a single program is to operate in both the E/A and XB environments, one must also overcome the fact that the nice easy REFs provided by E/A's Option 3 are not available. If the program was designed for XB, one can arrange to provide the XB utilities when operating under E/A. Conversely, one can design so that the XB version uses the E/A utilities.

In different programs, we've used both these approaches to closing the utility gap between XB and E/A. That's a topic we plan to explore at some length later in this series. For now, we'll just say that on a disk here at Harrison, we have a file called EAUT and a file called XBUT, so we can get the whole set of either into one of our programs.

When exiting from our WP program, we undo the things done on entry. We reset the VDP to graphics mode by putting a byte of >E0 at location >83D4, then LI R0,>01E0, and perform a BLWP @VWTR.

That's important, because E/A expects the screen to be in graphics mode when it resumes control. If you omit doing this, then return to E/A, the message PRESS ENTER TO CONTINUE, instead of being centered at the bottom of the screen, will be moved right so much that the UE of CONTINUE will be on a separate line. No real harm is done by this, but it's annoying to the user, and so we feel it should be avoided.

That's about all we'll cover today. It's a lot to digest for one sitting, anyway. For those who are serious students of Assembly, we recommend trying the two programs in today's Sidebar. Should you encounter difficulty, or need help with understanding what we're doing, please feel free to call us anytime between 9 AM and Midnight Eastern time at (301) 277-3467. We'll do our best to help you over the hurdles.

In our next article, we'll get into the subject of Loaders, of the sort we mentioned in passing here. For our own programs, we make customized loaders in each case, and take some liberties with the structure of our memory image files (no file headers), so our methods may be controversial, but they do work.

```
* TWO PROGRAMS
* PROGRAM #1
*
* A DEMO PROGRAM FOR THE SUBROUTINE CRSIN
* (INCLUDED IN PREVIOUS ARTICLE)
* THIS IS IN EFFECT A SHELL THAT ONE CAN USE TO TEST THE CRSIN SUBROUTINE
*
* REQUIRED REFERENCES
       REF  KSCAN,VMBW,VMBR,VSBW,VSBR
* DEFINE PROGRAM ENTRY POINT
       DEF  START
*
* REQUIRED EQUATES
STATUS EQU  >837C
WS     EQU  >20BA
GPLWS  EQU  >83E0
*
*
START  LWPI WS           LOAD WORKSPACE
       LI   R0,32*8+>800 SET R0 TO POINT TO SPACE CHARACTER DEFINITION
       LI   R1,TEMSTR    POINT R1 AT OUR TEMPORARY STORAGE
       LI   R2,8         EIGHT BYTES TO GET
       BLWP @VMBR        GET EIGHT BYTES
       S    R2,R0        STEP BACK ONE CHARACTER, TO THE EDGE CHARACTER
       BLWP @VMBW        WRITE EIGHT BYTES
       LI   R0,32*9+2    SET R0 FOR ROW 10, COLUMN 3
       LI   R4,20        TWENTY CHARACTERS TO ACCEPT
       LI   R15,RTNSTK   SET OUR RETURN STACK IN R15
       BL   @CRSIN       ACCEPT 20 CHARACTERS STRING
       LI   R0,32*14+2   SET FOR ANOTHER SCREEN LOCATION
       MOV  R2,R2        CHECK VALUE IN R2
       JEQ  SKIP         IF ZERO, JUMP AHEAD
       BLWP @VMBW        ELSE WRITE THE ACCEPTED STRING HERE
SKIP   LI   R4,20        RESET FOR 20 CHARACTERS
       BL   @CRSIN       RE-ENTER SUBROUTINE
       LWPI GPLWS        LOAD GPL WORKSPACE
       CLR  @STATUS      CLEAR THE STATUS
       B    @>006A       RETURN TO GPL INTERPRETER
*
* DATA SECTION FOR PROGRAM 1
```

```
*
TEMSTR BSS  21
* THE NUMBER IN THIS BSS MUST BE ONE MORE THAN THE LARGEST STRING LENGTH
* EXPECTED IN THE PROGRAM'S EXECUTION
* FOR THIS TEST, IT WAS SET AT 21 FOR A TWENTY CHARACTER INPUT STRING
       EVEN              SET PROGRAM COUNTER TO EVEN LOCATION
RTNSTK BSS  2            RETURN STACK ADDRESS AT AN EVEN LOCATION
       END
* END OF PROGRAM #1


* PROGRAM #2
* SETS PRINTER CONNECTED TO PIO PORT
* WILL AUTO-START AND RUN LABEL SKIPIT
* ENTRY AT LABEL DOUBLE WILL SET PRINTER TO DOUBLE STRIKE.
* REQUIRED REFERENCES
       REF  VMBW,DSRLNK,VSBW
* DEFINE ENTRY POINTS
       DEF  SKIPIT,DOUBLE
* REQUIRED EQUATES
*
PABPNT EQU  >8356        POINTER LOCATION FOR DSRLNK
STATUS EQU  >837C        GPL STATUS BYTE LOCATION
PAB    EQU  >1000        LOCATION FOR PAB IN VDP RAM
PABBUF EQU  >1050        BUFFER FOR BYTES TO BE SENT (VDP RAM ADDRESS)
GPLWS  EQU  >83E0        GPL WORKSPACE
*
* MAIN CODE SECTION FOR PROGRAM 2
*
DOUBLE MOV  R11,@SAV11   STASH CURRENT R11 VALUE INTO MEMORY AT LOCATION SAV11
       LWPI >20BA        LOAD USER WORKSPACE
       LI   R1,DSBYTE    SET R1 TO POINT TO DOUBLE STRIKE CHARACTERS
       JMP  PRN0         THEN JUMP
SKIPIT MOV  R11,@SAV11   STASH CURRENT R11 VALUE INTO MEMORY
       LWPI >20BA        LOAD USER WORKSPACE
       LI   R1,PRNBYT    SET R1 TO POINT TO SKIP-OVER PERFS CHARACTERS
PRN0   LI   R0,PABBUF    SET R0 TO CHARACTER BUFFER LOCATION
       LI   R2,3         THREE BYTES TO WRITE TO VDP RAM
       BLWP @VMBW        WRITE BYTES
       LI   R0,PAB       SET R0 FOR PERIPHERAL ACCESS BLOCK (PAB)
       LI   R1,PAB2DT    POINT R1 AT DATA FOR PAB
       LI   R2,16        SIXTEEN BYTES TO WRITE
       BLWP @VMBW        WRITE PAB TO VDP RAM
       AI   R0,9         ADD NINE TO POINT TO DESCRIPTOR LENGTH BYTE
       MOV  R0,@PABPNT   PLACE THAT VALUE AT >8356
* THE FOLLOWING LINE OPENS THE FILE
       BLWP @DSRLNK      PERFORM LINKAGE TO DEVICE SERVICE ROUTINE
       DATA 8            DATA FOR DSR LINKAGE
       LI   R1,>0300     PLACE WRITE OPCODE IN R1
       LI   R0,PAB       SET R0 FOR PAB LOCATION
```

```
       BLWP @VSBW        WRITE THE "WRITE" OPCODE INTO FIRST BYTE OF PAB IN VDP
       AI   R0,9         ADD NINE
       MOV  R0,@PABPNT   PLACE AT >8356
       BLWP @DSRLNK      WRITE THE BYTES FROM PABBUF TO PERIPHERAL (PIO PORT)
       DATA 8            REQUIRED DATA FOR DSRLNK
       LI   R1,>0100     PLACE CLOSE FILE OPCODE IN R1
       LI   R0,PAB       RESET R0 TO PAB
       BLWP @VSBW        WRITE THE CLOSE FILE OPCODE TO PAB
       AI   R0,9         ADD NINE
       MOV  R0,@PABPNT   MOVE TO >8356
       BLWP @DSRLNK      PERFORM CLOSE FILE OPERATION
       DATA 8            REQUIRED DATA
       LWPI GPLWS        LOAD GPL WORKSPACE
       MOV  @SAV11,R11   PUT RETURN ADDRESS BACK AT R11 OF GPL WORKSPACE
       CLR  @>STATUS     CLEAR STATUS
       RT                RETURN (BRANCH TO ADDRESS IN R11)
*
* DATA SECTION FOR PROGRAM 2
*
SAV11  DATA 0            PLACE TO SAVE R11 AT ENTRY
* FOLLOWING TWO LINES ARE THE REQUIRED DATA FOR A PERIPHERAL ACCESS BLOCK
* TO OPEN A D/V 80 FILE TO THE PIO PORT WITH THE .CR OPTION
* THE NUMBER >5003 IS PRELOADED SO AS TO PRINT ONLY THREE BYTES - THIS IS A
* SHORTCUT METHOD, NOT FOR GENERAL USE
*
PAB2DT DATA >0012,>1050,>5003,>0000,>0006
       TEXT 'PIO.CR'
DSBYTE BYTE 27,71,13     BYTES FOR DOUBLE STRIKE, PLUS A CARRIAGE RETURN
PRNBYT BYTE 27,78,10     BYTES TO SKIP OVER PERFS ON PRINTER
* NOTE - THE LAST BYTE ABOVE, WHICH WE SET AT 10, GIVES THE NUMBER OF
* LINES TO SKIP - THAT NORMALLY RANGES FROM ABOUT 4 TO 10
* WE USE 10 BECAUSE WE NORMALLY START OUR PRINTER WITH THE TOP EDGE OF
* THE SHEET JUST ABOVE THE PAPER BAIL TO PUT A BUILT-IN TOP MARGIN ON
* EACH SHEET, THUS MUST MAKE THE NUMBER HERE LARGER
*
       END SKIPIT
* END OF PROGRAM #2 - PLACING THE LABEL SKIPIT AFTER THE END DIRECTIVE
* MAKES THE PROGRAM RUN IMMEDIATELY AFTER LOADING FROM E/A OPTION 3
*
```

1.7. The Art Of Assembly — Part 7. Why A Duck?

By Bruce Harrison

This month's installment is not for the faint of heart. It will be heavy seas, high winds, rough waters. The subject is loaders. A loader is a program whose primary job is to load another program. Much of this article will be difficult to understand, and you may feel a little like Chico Marx in the movie The Cocoanuts, when he keeps asking Groucho "Why a duck?". Groucho was of course speaking of a Viaduct.

Putting first things first, we should answer the question "Why a Loader?" There are two answers to that question. One is speed, the other is memory allocation. One can, for example, write a loader that performs certain "once only" chores, then is mostly replaced by the program it loads, thus freeing up memory space for that program to use. The example we'll use today is taken from our Word Processor. In the Sidebar is the source code for a loader we use so that the program may be used with the E/A module or the TI Writer module. The file it creates is an Option 5 type program file called UTIL1. That is the default filename that both E/A Option 5 and TIW Option 3 will look for on DSK1.

We said the going would get rough, and here's the opening blast of the hurricane. This source code actually makes two programs in one. The object file containing both programs is loaded into memory under E/A Option 3, along with TI's SAVE utility. We enter the program by typing GETUT as a Program Name, thus entering the first of the two programs at that label. This little program gets a memory image file containing the Extended Basic utilities and stashes that code within the memory space occupied by the second program, then exits to TI's SAVE utility, so we can save the second program to disk as UTIL1. "Why a Duck?", you ask. Well, over here is the Viaduct. Just kidding! The reason is simple. The Word Processor was first developed as an E/A Option 3 program, mainly for our own use. When we decided to market it as a commercial product, we adapted it to run under Extended Basic with a custom loader submerged under an XB LOAD program. This meant that all the utilities such as VMBW, KSCAN, and so on were handled as Equates to the locations of these vectors when XB is in place. Much later, we decided, as part of a general upgrade, to add the ability to load from either E/A or TIW. Still, all the bulk of the program relied on finding utilities where XB places them. Thus we had to give our UTIL1 loader the capability of putting all those utilities in the right place before turning the computer over to the Word Processing program.

We then wrote a small Assembly program which would run under XB and capture the XB utilities for us in a memory image file called XBUT. Having that gave us the means to use the source code shown in the Sidebar along with TI's SAVE utility to create our own custom Option 5 loader.

And that is part of the reason for a custom loader. Back when we decided to make the program operate from Extended Basic, we needed a loader so that we would not have to load the object file with XB's CALL LOAD. The Word Processor proper fills nearly all of both the Low and High portions of the 32K memory. The object file (uncompressed) fills 394 sectors of disk space, and cannot be loaded by the E/A Option 3 loader because it AORGs into space used by that loader itself. Loading that object code under Extended Basic takes all of seven minutes. That's a long time to look at a screen which says "LOADING MAIN PROGRAM" and "PLEASE STAND BY". Thus we included in the WP code a custom "save" utility, so that the WP program could save itself as five separate memory image files. Then we wrote a loader which would be embedded under an XB LOAD program, and would load in these five memory image files from the disk. That way, from selecting XB to the Main Menu of WP on screen takes about 25 seconds from a floppy drive, or about four seconds from Ramdisk.

Much of the code in today's Sidebar is derived from that original Assembly loader we wrote for submerging under the XB LOAD program. It's not the prettiest code we've shown. In fact it's probably the ugliest we'll ever show you. We have violated many of our own rules in throwing this together. For example, there are data sections mixed in among the code sections. Also, there are places where we could have used our own methods to save memory space, but haven't done so. Instead, we followed the maxim "First, get it to work".

When we looked at the source code to prepare it for this article, we found "dead code" sections, subroutines that were never called, and unused data lines in it. Our only excuse is that this was mostly borrowed from the loader written for the XB version, hastily thrown together for a show deadline, and we quit looking at it once we got it to work. The version shown here will still work, but has been cleaned up considerably, and of course has been annotated so you can follow what it's doing line by line.

Before we look at the source code in detail, let's look at the Memory mapping for the WP program. (See Figure 1) In Low memory we have the XB utilities, then the section of code which starts up the program and puts the menu on the screen. Starting at >2A66 we have the code used for printing documents, plus the part that finds and reads a user's configuration file if he's configured his copy. That all ends at >39FE.

In High memory we have three arbitrarily divided sections of the code, which ends at >F1C0.

These five sections of code are stored on the disk in five Memory Image files, named as shown in Figure 1. The part called PRINTCODE could have been combined with the MENUCODE in one file, except that the section PRINTCODE is on occasion overwritten by either utility programs loaded as overlays, or by text from a Move or Copy text operation. Thus there are times during the program's operation that it must re-load PRINTCODE to print a document.

As you can see, the Loader sits at the higher addresses in High memory. At present, there's a gap between the end of the main program's code and the loader's beginning. If we wanted to, we could have the last section of code in the main program overlap all but a small portion of the loader without any harm. As it is, we have no plans presently for using up that remaining space.

Now into the source code. The REFs at the beginning refer to the E/A utilities which are available to us when the object code is loaded under Option 3. The code between label GETUT and the line reading B @SAVE is all that gets executed after the Option 3 loading. This code establishes a Peripheral Access Block using the data at label SAVDT, then uses DSRLNK to bring the memory image file XBUT into a VDP RAM buffer area. It then performs a VMBR operation to place that file's contents at label DATALD, within the part of the code that will be saved as UTIL1.

The TI SAVE utility takes everything between label SFIRST and SLAST and stores that as an Option 5 program file, which we name UTIL1. Thus the section at GETUT allows us to embed the Extended Basic utilities within that Option 5 program file before it's saved to disk.

We hope that's all clear, because if it isn't, then what follows will be very muddy indeed.

We now plunge into the murky waters of how the actual loader, saved as UTIL1, works. The very first thing it does is stash R11, which probably could be dispensed with, but it's there. Next it loads a temporary workspace, rather than our usual >20BA. The reason for this is simple. We will be writing to the area in low memory that includes >20BA, using registers as pointers, and we can't overwrite the workspace we're using. Thus we have a temporary workspace at label WS16 within the UTIL1 program's space.

The program now moves the 1262 bytes containing the XB utilities from label DATALD to their proper place in low memory, using the loop at label PUTUT. From this point on, the program will use the XB equates for its utility vectors such as VMBW. We've given these equates different names so they won't conflict with the REFs used in the GETUT section.

At label OPEN, we perform what's called "Boot Tracking". This section of code finds out what disk drive the UTIL1 program was loaded from, and passes that information into the PAB data that it will use to load the main program's five memory image files. Thus if one has the Word Processor disk in drive 2, or any other drive including Ramdisk, the UTIL1 program will go to that same drive to find its files and load them. This program, incidentally, has not been made compatible with hard disk systems, but will load and run from any floppy or Ramdisk, regardless of what its drive number or letter is.

After that, we branch to label MENU, where we load the final workspace at >20BA. Next we perform a little operation that's only needed when we've entered with the TI-Writer module. We simply capture the character definition for the space character, then use that to define the zero character [CHR$(0)] to look like a space. The main WP program will load its own character set beginning at character one [CHR$(1)], and extending through character 144, but the TIW module would leave the zero character defined, and we want it to look like a space.

Now we do some VWTR operations to set the screen to text mode, set up the colors for text mode, and to insure that VDP will look for character definitions at >800, then clear the screen. Now we give the user two messages on the screen, and get on with the real work of loading the five memory image files.

This would get repetitious, so we'll just go through the process of loading the first such file, called MENUCODE. First, we set R0 to point to the PAB area in VDP, R1 to point to the data, R2 to the length of that data, then write the PAB into VDP RAM. The PAB contains the opcode 05, which is used to load a memory image file. Next in the PAB is the VDP buffer address into which the bytes from that file will be dumped by the DSR. The third word in the PAB is always 0 for this kind of file, and the fourth indicates the maximum number of bytes to be taken from the file. This number must be equal to or greater than the actual file content. In our case, we've made it exactly equal to the number of bytes found in MENUCODE. Finally, there's a byte set to zero, then the length of the file descriptor, followed by the file descriptor text 'DSKx.MENUCODE'. The x is there to indicate that the 1 of DSK1 will have been replaced by the boot tracking process.

Once we've moved the PAB address plus nine into >8356 and cleared the STATUS, we proceed to get the file into the VDP RAM buffer by a BLWP @DSRLNJ. DSRLNJ? Why not DSRLNK, you ask? The utility vector DSRLNK has been overwritten when we moved the XB utilities into Low memory, so we can't use it. Instead, we've included a DSR Linkage (thanks to Doug Warren/Craig Miller) which has been renamed DSRLNJ so it won't upset the Assembler. This is the same DSRLNK that was included in Barry Traver's column some time ago. That link vector and its associated code is part of our program UTIL1, which is kept in the highest available addresses in High memory.

When our main program is running, it too uses this linkage vector to perform file accesses. It also uses the GPLLNK included in our UTIL1 program, just before DSRLNJ.

Okay, so now we've got the section called MENUCODE in VDP RAM. The next step is to put that into Low memory at the correct address. That address is >24F4+128, or >2574. The 128 bytes between >24F4 and >2574 are used when we enter from XB to stash some system data which XB will need when we exit the program, hence the actual content of our WP program starts at >2574. Moving of the code into its proper place is done by a BLWP @VMBRA, which is of course one of those XB utilities we put in place earlier.

This process continues until we've loaded all five memory image files. These files are true memory image, with no file headers attached to them. Many authors will go to some trouble about making file headers, but our reasoning was that, since the only possible use these files have is to be loaded by our own loader, there was no point in providing them with headers. There are days when we regret that decision, especially when we make changes in the main program which require changing the addresses equated in our loader, which then must be re-assembled just because one of those addresses changed by two bytes. The very next time we write a WP program, we'll put a file header of some kind in, giving us some information about the length of the file's content, and then we won't need to change and assemble the loader each time we change the main program.

The last couple of things this program does is to place the last section of code (WORDCODE3) into high memory, send the drive designator byte to location >FD0E+13, then branch to address >2840, the entry point of the main program. Location >FD0E+13 is meaningless in this program per se, but it happens to be the location where the XB loader has the drive designator byte, so we park that byte at that location, where the main program can get the information about what drive the program disk resides in.

This loader is fairly messy, and we really should get around to cleaning it up, but the idea in today's article was to give you some ideas to play with, and to show some essential things a loader must do. We understand that there are some "General Purpose" loaders available in the TI community, but our experience has been that, since our own style of doing things is unique, we're stuck with writing our own loaders. After doing a couple of them, one can always take an old one, make some changes to the source code, and generate a new custom loader with very little effort. We hope you now know "Why a Duck".

Our next article will go into the topic of file accesses in more depth, with emphasis on trapping errors in file operations.

```
* SOURCE CODE FOR UTIL1 LOADER
* TO LOAD HARRISON'S WP UNDER EDITOR/ASSEMBLER OPTION 5
* THIS IS ACTUALLY TWO PROGRAMS IN ONE
* THE FIRST GETS AND STOWS THE XB UTILITIES WITHIN THE SECOND,
* THEN BRANCHES TO TI'S SAVE UTILITY
*
       AORG >F690         SET MEMORY LOCATION FOR THIS CODE
       DEF  GETUT         DEFINE ENTRY POINT FOR FIRST PROGRAM
       REF  SAVE          REFERENCE THE LABEL SAVE IN THE TI SAVE UTILITY
       REF  VMBW,VMBR,VSBW,VSBR,DSRLNK
GETUT  MOV  R11,@>8300    STASH REGISTER 11 AT >8300
       LWPI WS16          LOAD A WORKSPACE WITHIN OUR OWN CODE
       LI   R0,PAB1       SET POINTER FOR PERIPHERAL ACCESS BLOCK
       LI   R1,SAVDT      POINT TO THE DATA FOR THAT PAB
       LI   R2,19         NINETEEN BYTES IN THE PAB DATA
       BLWP @VMBW         WRITE 19 BYTES TO VDP RAM
       AI   R0,9          ADD 9 TO R0
       MOV  R0,@PABPNT    PLACE THAT NUMBER AT >8356
       CLR  @STATUS       CLEAR THE GPL STATUS BYTE
       BLWP @DSRLNK       GET THE XB UTILITIES INTO VDP BUFFER FROM DISK FILE
       DATA 8             DATA FOR DSR LINKAGE
       LI   R0,>1020      POINT TO BUFFER SPACE IN VDP RAM
       LI   R1,DATALD     POINT R1 TO LOCATION WITHIN PROGRAM TO BE SAVED
       MOV  @SAVDT+6,R2   GET LENGTH OF FILE INTO R2
       BLWP @VMBR         READ THE XB UTILITIES INTO MEMORY
       B    @SAVE         BRANCH TO THE TI SAVE UTILITY
SAVDT  DATA >0500,>1020,0,>24F6->2008,>0009
       TEXT 'DSK4.XBUT'
*
* END OF FIRST PROGRAM
*
* START OF SECOND PROGRAM
* THE PART FROM HERE TO THE END IS SAVED BY THE TI SAVE UTILITY AS FILE UTIL1
* THIS PART IS WHAT LOADS THE FIVE MEMORY IMAGE FILES COMPRISING THE WP PROGRAM
*
       DEF  SFIRST,SLAST,SLOAD   DEFINED LABELS REQUIRED BY TI'S SAVE UTILITY
SFIRST
SLOAD
       MOV  R11,@>8300   STASH R11
```

```
       LWPI WS16          LOAD TEMPORARY WORKSPACE
       LI   R9,DATALD     POINT AT DATA FROM SAVED XB UTILITIES
       LI   R10,>2008     POINT AT START OF XB UTILITY VECTOR AREA
       LI   R4,>24F6->2008  SET R4 FOR NUMBER OF BYTES IN XBUT
PUTUT  MOV  *R9+,*R10+    MOVE A WORD INTO LOW MEMORY AREA, INCREMENT POINTERS
       DECT R4            DECREMENT COUNT BY TWO, SINCE WE'RE MOVING A WORD
       JNE  PUTUT         IF NOT ZERO, REPEAT
       B    @OPEN         BRANCH TO NEXT SECTION OF CODE
WS16   BSS  32            TEMPORARY WORKSPACE
DATALD BSS  >24F6->2008   STORAGE AREA FOR XB UTILITIES
FSTEND EQU  >2A66         END OF FIRST SECTION OF MAIN PROGRAM
ENDCNF EQU  >39FE         END OF CONFIGURATION SETTING CODE
DEFPRN EQU  >F0F8+200     END OF HIGH MEMORY PART OF WP
FAC    EQU  >834A         FLOATING POINT ACCUMULATOR
WS     EQU  >20BA         REAL WORKSPACE
VSBRA  EQU  >2028         THE XB VSBR VECTOR'S ADDRESS
OPEN
* THE SECTION HERE AT LABEL OPEN PERFORMS "BOOT TRACKING"
* THAT IS, IT TELLS OUR PROGRAM WHICH DRIVE IT WAS LOADED FROM
       MOV  @>83D0,R12    GET THE CRU BASE IN R12
       MOV  @>83D2,R9     GET THE ROM ADDRESS FOR DEVICE
       LDCR @ONES,0       ENABLE THE ROM
       AI   R9,4          ADDING FOUR PUTS US AT THE LENGTH BYTE
       MOVB *R9+,R4       PLACE THAT IN R4 AND INCREMENT R9
       SRL  R4,8          RIGHT JUSTIFY LENGTH IN R4
       LI   R10,SAVDT3+10 POINT TO TEXT BUFFER
MOVIT  MOVB *R9+,*R10+    MOV ONE BYTE FROM ROM TO TEXT BUFFER
       DEC  R4            FINISHED?
       JNE  MOVIT         NO, DO ANOTHER BYTE
       LDCR R4,0          DISABLE THE ROM (R4 IS ZERO AT THIS POINT)
       MOVB @SAVDT3+13,R1 MOVE DRIVE NUMBER (OR LETTER) INTO R1
       MOVB R1,@MENUDT+13 THEN MOVE INTO THE PAB DATA LINES
       MOVB R1,@WRD1DT+13
       MOVB R1,@WRD2DT+13
       MOVB R1,@WRD3DT+13
       B    @MENU         BRANCH TO NEXT SECTION OF CODE
ONES   DATA >0101         WORD TO TURN ON ROM IN CRU
*
VMBWA  EQU  >2024         XB'S VMBW VECTOR LOCATION
VMBRA  EQU  >202C         XB'S VMBR LOCATION
VSBWA  EQU  >2020         XB'S VSBWA
VWTR   EQU  >2030         XB'S VWTR LOCATION
KEYADR EQU  >8374         KEY-UNIT ADDRESS
STATUS EQU  >837C         GPL STATUS BYTE
SCRMO  EQU  >83D4         STORAGE LOCATION FOR SCREEN MODE BYTE
PABPNT EQU  >8356         POINTER LOCATION FOR DSR LINKAGE
PAB1   EQU  >400          FIRST PAB ADDRESS
TEXMO  BYTE >F0           TEXT MODE BYTE
BLNKLN TEXT '                                        '
OPMSG  TEXT 'LOADING IN MAIN PROGRAM' TEXT MESSAGE
```

```
PSBMSG TEXT 'PLEASE STAND BY'
CRITE  BSS  8
* FOLLOWING DATA SECTION CONTAINS THE PAB DATA FOR THE SECTIONS OF MAIN PROGRAM
MENUDT DATA >0500,>1000,0,FSTEND->2574,>000D
       TEXT 'DSK1.MENUCODE'
SAVDT3 DATA >0500,>1000,0,ENDCNF-FSTEND,>000E
       TEXT 'DSK1.PRINTCODE'
WRD1DT DATA >0500,>0D00,0,9983,>000E
       TEXT 'DSK1.WORDCODE1'
WRD2DT DATA >0500,>0D00,0,9983,>000E
       TEXT 'DSK1.WORDCODE2'
WRD3DT DATA >0500,>0D00,0,DEFPRN+2->A000-9983-9983,>000E
       TEXT 'DSK1.WORDCODE3'
* MAIN PART OF LOADER BEGINS HERE
MENU   LWPI WS           SETS UP WORKSPACE
       MOVB @TEXMO,@SCRMO MOVE BYTE >F0 INTO >83D4
       CLR  @KEYADR      CLEAR WORD AT >8374
* THE NEXT SIX LINES ARE HERE TO CLEAR OUT THE DEFINITION OF CHARACTER ZERO.
* THAT CHARACTER IS DEFINED WHEN WE ENTER FROM TI-WRITER MODULE, SO WE SET IT
* UP TO LOOK LIKE A SPACE CHARACTER.
* THIS IS NECESSARY SINCE OUR WP MAKES USE OF CHARACTER ZERO, AND WE WANT IT TO
* LOOK LIKE A SPACE ON THE SCREEN
       LI   R0,32*8+>800 SET R0 TO SPACE CHARACTER DEFINITION
       LI   R1,CRITE     USE A STORAGE SPACE
       LI   R2,8         EIGHT BYTES TO GET
       BLWP @VMBRA       READ EIGHT BYTES
       LI   R0,>800      POINT TO CHARACTER ZERO DEFINITION
       BLWP @VMBWA       WRITE EIGHT BYTES
TEXT   LI   R0,>01F0     PREPARES FOR TEXT MODE
       BLWP @VWTR        SETS SCREEN IN TEXT MODE
       LI   R0,>074E     SETS COLORS
       BLWP @VWTR        FOR TEXT MODE
       LI   R0,>0401     PREP TO SET CHARACTER TABLE AT >800
       BLWP @VWTR        SET IT THERE
CLS    CLR  R0           POINT R0 TO SCREEN ORIGIN
       LI   R4,24        24 ROWS TO CLEAR
       LI   R1,BLNKLN    POINT TO 40 SPACES TEXT
       LI   R2,40        40 CHARACTERS PER ROW TO WRITE
LOOP   BLWP @VMBWA        WRITE 40 SPACES
       A    R2,R0        MOVE WRITE ADDRESS 1 LINE (ADD 40 TO R0)
       DEC  R4           DECREASE COUNT OF ROWS
       JNE  LOOP         IF NOT ZERO,LOOP BACK AND DO ANOTHER
       LI   R0,9*40+9    SET R0 FOR ROW 10, COLUMN 10
       LI   R2,23        23 CHARACTERS IN MESSAGE
       LI   R1,OPMSG     POINT R1 AT MESSAGE
       BLWP @VMBWA       WRITE MESSAGE "LOADING IN MAIN PROGRAM" TO SCREEN
       LI   R0,11*40+12  SET R0 FOR ROW 12, COLUMN 13
       LI   R2,15        15 BYTES IN MESSAGE
       LI   R1,PSBMSG    WRITE "PLEASE STAND BY"
       BLWP @VMBWA       TO THE SCREEN
```

```
LI   R0,PAB1       POINT R0 TO PERIPHERAL ACCESS BLOCK VDP ADDRESS
LI   R1,MENUDT     POINT TO FIRST PAB DATA BLOCK
LI   R2,23         23 CHARACTERS IN BLOCK
BLWP @VMBWA        WRITE PAB TO VDP RAM
AI   R0,9          ADD NINE TO ADDRESS
MOV  R0,@PABPNT    MOVE THAT VALUE TO >8356
CLR  @STATUS       CLEAR STATUS BYTE
BLWP @DSRLNJ       USE DSR LINKAGE VECTOR
DATA 8             DATA FOR DSR LINK
LI   R0,>1000      POINT TO BUFFER AREA
MOV  @MENUDT+6,R2 GET FILE LENGTH INTO R2
LI   R1,>24F4+128 POINT AT LOW MEMORY LOCATION FOR FIRST CODE SECTION
BLWP @VMBRA        READ THE SECTION MENUCODE INTO LOW MEMORY
LI   R0,PAB1       POINT TO PAB LOCATION
LI   R1,SAVDT3     SAVDT3 IS SECOND PAB DATA PORTION
LI   R2,24         24 BYTES TO WRITE
BLWP @VMBWA        WRITE PAB INTO VDP
AI   R0,9          ADD NINE TO ADDRESS
MOV  R0,@PABPNT    MOVE THAT TO >8356
CLR  @STATUS       CLEAR GPL STATUS BYTE
BLWP @DSRLNJ       USE DSR LINKAGE VECTOR
DATA 8             REQUIRED DATA
LI   R0,>1000      POINT TO BUFFER
LI   R1,FSTEND     POINT TO END OF FIRST CODE SECTION
MOV  @SAVDT3+6,R2 LENGTH OF CODE SECTION IN R2
BLWP @VMBRA        MOVE THE FILE PRINTCODE INTO LOW MEMORY
LI   R1,WRD1DT     POINT TO NEXT PAB DATA
LI   R0,PAB1       SET R0 TO PAB
LI   R2,24         24 BYTES TO WRITE
BLWP @VMBWA        WRITE DATA TO PAB
AI   R0,9          ADD NINE
MOV  R0,@PABPNT    MOVE TO >8356
CLR  @STATUS       CLR STATUS
BLWP @DSRLNJ       DSR LINK
DATA 8             REQ'D DATA
LI   R0,>0D00      SET TO BUFFER LOCATION
LI   R1,>A000      POINT R1 TO START OF HIGH MEMORY
LI   R2,9983       9983 BYTES IN FILE
BLWP @VMBRA        READ THIS SECTION INTO HIGH MEMORY
LI   R0,PAB1       RESET TO PAB
LI   R1,WRD2DT     SECOND HIGH MEMORY PART
LI   R2,24         24 BYTES
BLWP @VMBWA        WRITE PAB
AI   R0,9          ADD 9
MOV  R0,@PABPNT    TO >8356
CLR  @STATUS       CLR STATUS
BLWP @DSRLNJ       DSR LINK
DATA 8             DATA
LI   R0,>0D00      POINT TO BUFFER
LI   R1,>A000+9983 ADDRESS FOR SECOND SECTION OF HIGH MEMORY CODE
```

```
       LI   R2,9983      9983 BYTES TO READ
       BLWP @VMBRA       READ INTO HIGH MEMORY
       LI   R0,PAB1      SET TO PAB
       LI   R1,WRD3DT    THIRD HIGH MEMORY PART
       LI   R2,24        24 BYTES PAB DATA
       BLWP @VMBWA       WRITE TO VDP
       AI   R0,9         ADD 9
       MOV  R0,@PABPNT   TO >8356
       CLR  @STATUS      CLR STATUS
       BLWP @DSRLNJ      DSR LINK
       DATA 8            DATA
       LI   R0,>0D00     SET TO BUFFER
       LI   R1,>A000+9983+9983 ADDRESS FOR LAST SECTION OF CODE
       MOV  @WRD3DT+6,R2 LENGTH OF CODE SECTION
       BLWP @VMBRA       READ CODE INTO HIGH MEMORY
       MOVB @SAVDT3+13,@>FD0E+13 "MAILBOX" THE DRIVE DESIGNATOR FOR MAIN PGM
       B    @>2840       BRANCH TO MAIN PROGRAM ENTRY POINT
*  GENERAL PURPOSE GPL AND DSR LINKS
*  FOR USE UNDER EXTENDED BASIC
*  THIS CODE BY DOUG WARREN/CRAIG MILLER OF MILLER'S GRAPHICS
GPLWS  EQU  >83E0
GR4    EQU  GPLWS+8
GR6    EQU  GPLWS+12
STKPNT EQU  >8373
LDGADD EQU  >60
XTAB27 EQU  >200E
GETSTK EQU  >166C

       AORG >FF2C        SET MEMORY LOCATION FOR LINKAGE ROUTINES
* THIS AORG IS SET SO THAT UTILITIES WIND UP AT THE SAME LOCATION AS WITH THE
* EXTENDED BASIC LOADER
GPLLNK DATA GLNKWS
       DATA GLINK1
RTNAD  DATA XMLRTN
GXMLAD DATA >176C
       DATA >50
GLNKWS EQU  $->18
       BSS  >08
GLINK1 MOV  *R11,@GR4
       MOV  *R14+,@GR6
       MOV  @XTAB27,R12
       MOV  R9,@XTAB27
       LWPI GPLWS
       BL   *R4
       MOV  @GXMLAD,@>8302(R4)
       INCT @STKPNT
       B    @LDGADD
XMLRTN MOV  @GETSTK,R4
       BL   *R4
       LWPI GLNKWS
```

```
       MOV  R12,@XTAB27
       RTWP
*
*
PUTSTK EQU  >50
TYPE   EQU  >836D
NAMLEN EQU  >8356
VWA    EQU  >8C02
VRD    EQU  >8800
GR4LB  EQU  >83E9
GSTAT  EQU  >837C

DSRLNJ DATA DSRWS,DLINK1
DSRWS  EQU  $
DR3LB  EQU  $+7
DLINK1 MOV  R12,R12
       JNE  DLINK3
       LWPI GPLWS
       MOV  @PUTSTK,R4
       BL   *R4
       LI   R4,>11
       MOV  R4,@>402(R13)
       JMP  DLINK2
       DATA 0
       DATA 0,0,0
DLINK2 MOVB @GR4LB,@>402(R13)
       MOV  @GETSTK,R5
       MOVB *R13,@DSRAD1
       INCT @DSRADD
       BL   *R5
       LWPI DSRWS
       LI   R12,>2000
DLINK3 INC  R14
       MOVB *R14,@TYPE
       MOV  @NAMLEN,R3
       AI   R3,-8
       BLWP @GPLLNK
DSRADD BYTE >03
DSRAD1 BYTE 00
       MOVB @DR3LB,@VWA
       MOVB R3,@VWA
       SZCB R12,R15
       MOVB @VRD,R3
       SRL  R3,5
       MOVB R3,*R13
       JNE  SETEQ
       COC  @GSTAT,R12
       JNE  DSREND
SETEQ  SOCB R12,R15
DSREND RTWP
```

```
SLAST  END
* SLAST MARKS THE END OF WHAT'S SAVED IN THE FILE UTIL1
```

1.8. The Art Of Assembly — Part 8. File Handling Tips

By Bruce Harrison

This month's column will offer a few useful tips and hints drawn from our experience with file handling in Assembly language.

There are some fundamental decisions that must be made in any file structure you're going to use in a program. Perhaps the most fundamental is whether to use a Fixed or Variable record size. This of course brings implications along for the ride. With Fixed record length, you get the ability to access any record in the file without reading those that came before it. In a game program that we wrote some time ago, we used a fixed record length so that we could take a random number and use that to read a record in the file selected at random. In another case, for our Golf Score Analyzer, we used a fixed record length of 56 bytes, which corresponded to the records in memory relating to each round of golf entered. This made our whole file-access problem much easier to deal with than it could have been.

Variable record length is, in general, more efficient in use of disk space, because the disk controller will squeeze as many records as possible into each sector of the disk. Thus if your file organization is Variable 80, there could be twenty such records in a single 254 byte sector on the disk, provided the actual length of the records was quite short. You can't, of course, put more than eighty characters in any record.

In either case, this choice of fixed or variable is not to be taken lightly, as the choice made will have implications later on. We've had occasion to regret some decisions we've made along this line, and so advise a lot of "thinking through" for this decision.

By far the worst "killer" in file handling operations is error handling. Errors generally fall into two major categories. The first is the case in which the file failed to open. One common cause for this kind of error is a mistake of one kind or another in the device name. If the user types in "DKS1" instead of "DSK1", the file will not open. We have found one reliable way of detecting that this kind of error has occurred. If an invalid device name is used, the status register will have its bit two set. We have found that, at least with TI's built-in DSRLNK routine, the status byte at >837C does not get bit two set. (Page 298 of the E/A manual says that the byte at >837C does get bit 2 set.) Thus one must examine the status register upon return from an Open File operation to discover whether or not such an error has happened. In source code:

```
            BLWP @DSRLNK            Use DSR link to open file
            DATA 8                  Required data for DSRLNK
            STST R14                Store the status register in R14
            ANDI R14,>2000          Mask all but bit 2 in R14
            JEQ  NEXTOP             If zero, file has opened - proceed
            B   @OPNERR             Else file has not opened, report error
NEXTOP      (program continues)
```

We have run extensive tests using this method, and have not found it to fail to detect a "bad device name". It will, when used in combination with the error reporting scheme shown below, also detect errors of other kinds when opening a file, such as "bad attribute" errors. Using a special program we prepared just for testing, we found we could type in such errors as DKS1, SR232, POI, and so forth, and the OPEN would always indicate a bad device name using the above code. This will also report the same error if, for example, you type DSK6 when you only have DSK1 thru DSK5 on your system (including Ramdisks).

When an open error has been detected, you can branch to the error reporting code at OPNERR and produce a screen message such as "BAD DEVICE NAME" or "BAD ATTRIBUTE", to alert the user.

There are other errors that will be reported when opening a file, such as the case where your external Drive 2 is turned off, or the drive door is open. These will report "DEVICE ERROR" using the source code included here.

Once a file has been opened, there are other possible errors that can be found by attempting to read or write records in the file. The only way we've found to properly report such errors is as follows:

```
            BLWP @DSRLNK            Attempt to read or write
            DATA 8                  Required data for DSR
            LI   R0,PAB+1           Point to second byte in PAB
            BLWP @VSBR              Read that into left byte R1
            SRL  R1,13              Shift R1 right by 13 bits
            JEQ  CONTOP             If zero, operation OK
            B   @FILERR             Else branch to error report
CONTOP      (program continues)
```

The Editor/Assembler book gives definitions for the errors you can find by this method, on page 299. Note that the first given applies only in the case of a bad device name, and that must actually be found in the OPEN operation. This snippet of source code assumes you have a Peripheral Access Block (PAB) in the VDP RAM at address PAB. When a write or read error has been found on an opened file, you can construct a lookup table in your error reporting scheme, and report errors in plain English on screen. This is preferable to simply reporting errors by code numbers, since it gives the user a clue to what may be wrong. There will be cases, of course, that fall into the catchall "OTHER FILE ERROR" category, and that message won't help much.

Here's the source code for a lookup table method to report file errors in a reasonably user-friendly fashion, assuming the errors have been detected as we've shown above:

```
OPNERR      LI   R0,22*32+2         row 23, column 3 of screen
            LI   R1,FNOMSG          Point to message string
            BL   @DISSTR            Use subroutine to display
            LI   R0,PAB+1           Point to PAB+1
            BLWP @VSBR              Read into left byte R1
            SRL  R1,13              Shift R1 right 13 bits
FILERR      SLA  R1,1               Double number in R1
            AI   R1,LUT             Add lookup table address
            MOV  *R1,R1             Get address of text string
            LI   R0,23*32+2         Row 24, column 3
            BL   @DISSTR            Use subroutine to display
            BL   @KEYLOO            Stop at key loop for reading
            B    (somewhere else in program)
```

The line BL @KEYLOO is simply a means of stopping anything from happening so the user can read the message on the screen. The subroutine KEYLOO was included in Part 2 of this series, and may be used as shown there.

In the data section, include the following:

```
BADDEV      BYTE 15
            TEXT 'BAD DEVICE NAME'
WRPROT      BYTE 15
            TEXT 'WRITE PROTECTED'
BADATT      BYTE 13
            TEXT 'BAD ATTRIBUTE'
ILLOP       BYTE 17
            TEXT 'ILLEGAL OPERATION'
OUTSP       BYTE 19
            TEXT 'OUT OF BUFFER SPACE'
ENDFIL      BYTE 11
            TEXT 'END OF FILE'
DEVERR      BYTE 12
            TEXT 'DEVICE ERROR'
FILBAD      BYTE 16
            TEXT 'OTHER FILE ERROR'
LUT         DATA BADDEV,WRPROT,BADATT
            DATA ILLOP,OUTSP,ENDFIL
            DATA DEVERR,FILBAD
FNOMSG      BYTE 17
            TEXT 'FILE DID NOT OPEN'
```

You'll also need the small subroutine which we've called DISSTR to display the appropriate string on the screen. For example:

```
DISSTR      MOVB *R1+,R2            Get length into left byte R2
            SRL  R2,8               Right justify byte in R2
            JEQ  DISX               If R2 zero, get out
            BLWP @VMBW              Write text to screen
DISX        RT                      Return
```

We'll digress for just a moment here to discuss that line JEQ DISX in this subroutine. The utility VMBW does not check to see whether R2 is zero before performing its job, so we must do that before calling the utility. In the code we've shown here, R2 would never be zero at this point, but this subroutine can be used for other purposes, and might encounter a null string to display. If one calls VMBW with R2 zero, the utility will attempt to write 65,535 bytes into VDP RAM, with disastrous results.

There will be situations where you'll want to do something different about an end of file error, rather than reporting that to the screen. If you are reading the entire file into memory, you may want an error where R1 is five to simply return to a menu or go on to some other place in the program. You could, after each read operation, insert a CI R1,5 after the line JEQ CONTOP, then jump or branch to somewhere else if R1=5.

There are some things said in the TI E/A manual that, to borrow a phrase from Gershwin, "Ain't necessarily so!" We mentioned one of those earlier, concerning the status byte at >837C. Here's another. The book says that the Status test only applies if the three leftmost bits of the byte at PAB+1 are all zero. Not true! For Open operations, we recommend that you first test for the Status register, then check the PAB+1 byte, as we did at OPNERR. There will be instances, for example, when this second test will indicate "BAD ATTRIBUTE" when a file did not open.

Our tests have been as exhaustive (and exhausting) as we could manage in preparing for this article. Our system has a Horizon Ramdisk with Drives 3, 4, and 5 thereon, plus floppy drives 1 and 2, connected to a TI Controller. We have even tried writing to a full disk, and sure enough when we opened a file for Output to that disk, we got a "FILE DID NOT OPEN" message plus the "OUT OF BUFFER SPACE" message. We did find instances where the Ramdisk reports a different error than the TI controller does for the same situation.

There was one peculiar thing that we found on Ramdisk. Our test program was set up for D/V 80 files. The disk catalog file (e.g. DSK1.) is not that kind of file, and the TI Controller will dutifully give you a BAD ATTRIBUTE indication and will not open the file when you try it as a D/V 80 file. The Ramdisk, however, will open the file DSK5. and allow you to read from it, even though the attributes in our PAB clearly do not match that file. In other cases, such as trying to open a D/F 80 file with a PAB set up for D/V 80, the Ramdisk does report errors correctly.

This is of course not something you'd normally do on purpose, but it did confuse us to see the Ramdisk catalog being accessed as a D/V 80 file. Maybe Gary Bowser planned it that way, but in any case it won't bother the average user.

Incidentally, there's nothing really magic about the messages we've used in today's source code. You could make your messages shorter or more elaborate. "OUT OF BUFFER SPACE" could just as well read "OUT OF DISK SPACE", since that's the more likely cause for this kind of error. For our own convenience, we made each message in the form of a string, with its length byte first, then content in a TEXT line. That allowed us to use a rather simple subroutine (DISSTR) to display any message chosen from the lookup table. This simple subroutine will not work with XB, but does just fine in E/A Option 3 programs like the one we used to do testing for this article. We also tested this error reporting scheme using the GPLLNK and DSRLNK shown in last month's Sidebar, with results identical to those using TI's DSRLNK under the E/A Option 3.

Yes, we've gone on forever about the business of error checking without even showing all the steps required to open a file, let alone read or write to one.

We believe, however, that with a little patience studying the E/A manual, one can learn fairly quickly how to manage those steps, but the business of error trapping and reporting has caused us much anguish, so we thought it deserved lots of "ink".

In our next installment, we'll continue with more pointers on file handling, including some of the common types of files such as D/V 80, D/F 80, and Memory Image files (a.k.a. Program files). We'll open, read, write, and close some files, too.

1.9. The Art Of Assembly — Part 9. More File Handling Tips

By Bruce Harrison

Last month we spent most of our space dealing with file errors. This month we'll get into ways to deal with the normal situation of files that do open and get read or written.

Since space is limited, we'll concentrate on commonly used file types, like Display/Variable 80, Display/Fixed 80, and so-called Program files, also known as Memory Image files. After you've read this article, you should know how to change the source code to handle other file types.

One of the primary requirements for doing anything with files on the TI is to set aside some space in the VDP RAM for your Peripheral Access Blocks (PABs) and the Buffers you'll need to send or receive data from your files. To some extent, you are free to use many different locations in VDP, but must take care not to overlap areas important to your program's execution. If you're operating in Graphics Mode on the VDP (this is the normal mode when you **ENTER** from E/A Option 3) you'll find that any address above >1000 and below >37D7 can be used. In many cases we've put our PAB at >1000 and our buffer at >1050. It's important to insure that the buffer won't overlap either the PAB or >37D7. For most devices, the entire PAB, including the file descriptor, will not exceed 25 bytes in length. If, however, you're dealing with Hard Drive files, the descriptor may occupy many more bytes, including directory and sub-directory names.

If you are operating in TEXT mode on the VDP, another area in VDP RAM is open for your use, between >400 and >800. In our Word Processor, which operates in TEXT mode, we use that area for four separate PABs and their associated buffers.

If only one file at a time is opened, you can "recycle" and use the same PAB and Buffer area for any and all files you use. Otherwise, you'll need a separate PAB and Buffer for each of the files that are open simultaneously.

The practice we use for establishing PABs is fairly simple. The fixed data for the first ten bytes is placed in the data area of our source code, with a small area (usually 15 bytes) reserved beyond that for the file descriptor, which comes from user-entered data. An example is shown in the Sidebar at label PAB1DT. This PAB data is preset for opening a D/V 80 file type. The code at label START shows how one might use our subroutines CRSIN and MOVSTR (given in previous articles of this series) to get the name of the file in place as a user input. In this case, the full version of CRSIN should be used, including the lower case to upper case conversion lines, so the user won't have to worry about having **ALPHA LOCK** engaged to enter a file name with all upper-case letters. One note of caution — after the call to CRSIN, it's wise to see whether the user has left the input blank, and issue him an error message if he has. CRSIN leaves the length of the input string in R2, so a simple MOV R2,R2 followed by a JEQ to jump somewhere and report an error will do the trick.

The code at label OPNF1 shows how to load that data into the PAB area in VDP, and then open the file. First, the file mode must be placed in the PAB1DT area. We've shown in the example a file to be opened for INPUT by moving a byte called INMD to PAB1DT+1. Incidentally, we don't recommend using UPDATE mode for Variable record length files. It is too easy to create an unusable file by trying to write to such a file in UPDATE mode. For Fixed record length files, UPDATE will allow you to modify records in the file at random without messing up the rest of the file.

It's important at this stage that the first byte in this PAB be 00, so that the file will OPEN. The next step is to write this data into VDP at the correct place, and to insure that the right number of bytes are written to include the complete file descriptor. The first example in the Sidebar shows the right way to do this, so that the actual descriptor length is used to write all the necessary bytes into the VDP. If you're not dealing with the longer descriptors needed for Hard Drives or RS232, you can use the second, or shortcut, method, which writes 25 bytes regardless of descriptor length. (25 bytes will include the 10 of the PAB, plus 5 for the device name and period, plus 10 for the maximum acceptable file name length.)

Once the PAB has been written to the VDP, one needs to place the address PAB1+9 at location >8356, which we call PABPNT. The DSRLNK must have this address in that location so it can find the file descriptor in VDP RAM. We normally include a CLR @STATUS before calling DSRLNK, but we're not sure it's necessary. In some cases, we've forgotten to do it with no ill effects. Be brave, and leave that line out.

In all cases, the BLWP @DSRLNK line must be followed by a line reading DATA 8. Maybe there was supposed to be another use for DSRLNK in which some number other than 8 would be used, but we don't know about that. In any case, forgetting the DATA 8 will cause DSRLNK to fail. We've shown here the steps necessary to detect an error on opening the file, and that branches to the code we included in last month's article.

Given that the file opens, we can read records from the file using the code at label RDF1. The bytes READF, WRITEF, and CLOSEF will work for any kind of file except Memory Image type, in which they're not needed.

For illustrative purposes only, we've parked the contents of each record we read at location TEMSTR, which in this case has been set to a block of 81 bytes, 1 for the length of the record, plus 80 for the maximum possible record length. In a real application, you'd want to move that string to somewhere else before reading the next one from the file.

Here we've also included the error detection needed for read operations, but made the exception for error code 5, End Of File. If we've reached the end of file, we simply jump ahead to the close file operation at CLSF1. We should mention for those skilled in Extended Basic programming that this End of File error does not work exactly like the EOF function in XB. XB reports EOF when the last record in the file is read, while this error does not report until you try to read a record beyond the last record. Let's say for example there were forty records in the file opened as #1, and we're reading with XB. As soon as we've read the fortieth, XB will report EOF(1). In our Assembly case, Error 5 will not be reported until we try reading the forty-first record.

Incidentally, the XB Manual states that the EOF function will not work for Fixed Record length files. That's wrong. EOF works just the same for Fixed or Variable in XB. Error 5 in Assembly also works for both Fixed and Variable record lengths.

While we're at comparisons to XB, we should say that the method we've shown for reading into a string (TEMSTR) is essentially equal to a LINPUT function in XB, in that it places the entire contents of the record at location TEMSTR. This may be important if you've created the file with more than one variable stored in the same record, as you'll have to sort out the contents of the record for yourself after they're dumped into TEMSTR. In a later article, we'll try to give some pointers on how to separate different variables in such a case. We've done that when reading the Catalog file of a disk for our Word Processor, and it's not really difficult.

The final step in file operations is to close the file. The code to accomplish that is shown at CLSF1. We normally recommend that files be closed as soon as you've finished reading or writing them. There are possible exceptions, such as the case of a Fixed record length file in which you want to skip around and read random records. In that case, you can with reasonable safety leave the file open while other functions are performed, then close it when exiting your program.

Trying to close a file that hasn't opened will in general do no harm, but will result in an error. We haven't bothered to show detection or handling of that error. What we recommend is that the error trapping that shows a file has not opened should also cause the program not to try closing it. The adept student will no doubt invent a simple way to do this.

Now that you know how to open a file and read it, you'll easily determine how to open one for output and write to it. The sample code shown at OPNF2 will serve well. Studying that annotated code should give you all you need, so we won't dwell on it here. Also in today's Sidebar are PAB setups and mode bytes for D/F 80 type files.

Our final topic for today is that of the special kind of files called Memory Image files, or, as they list in a disk catalog, Program files. The first thing you should know about them is that the name PROGRAM is often a misnomer. In our Word Processor, we use a file called WPCHARACT, which will list on a disk catalog as PROGRAM, but is in reality a character set which we read into VDP directly at >808 to set up character definitions for characters from 1 through 144.

True PROGRAM files, such as those made by the XB SAVE operation or by the TI SAVE utility, have file headers so the computer can detect what kind of files they are, and where they belong in memory. Thus if you try to load in and run an XB program under E/A Option 5, you'll get an error once the E/A loader reads the file header information from the file.

We very often use the dangerous practice of creating memory image files without bothering to place headers on them, since they're used in ways that don't need headers. It isn't really dangerous unless some thoughtless user tries to load them as XB or Option 5. Disaster may then ensue.

We just made a little experiment along that line, and it turns out that, while XB is forgiving, and reports I/O ERROR 50 for our "headerless" files, E/A Option 5 simply goes bonkers given a "program" file that isn't one. Maybe not always, but it just did that in two out of two tries with memory image but non-type 5 files. We must include that subject at more length in a future article, which we plan to subtitle "Off the End of the World".

In Part 7, we showed some source code used in our Word Processor's E/A Option 5 loader, which should serve as a good example of how we can bring our headerless memory image files into memory, then branch into the program placed in memory that way.

In today's Sidebar, there's the inverse case, showing how the "saver" part of our Golf Score Analyzer works to save that program into two memory image files. Like the Word Processor, the GSA's object file can only be loaded initially by the CALL LOAD process under Extended Basic. (We suffer the tiresome delay, so our users won't have to.) Once we've done that, we can CALL LINK ("SAVIT"), and thus exercise the code shown at that label in the Sidebar. SAVIT first saves our INSTALL program for the GSA, which is part of the code when loaded from the object file by XB. That part gets used as an overlay into the memory normally reserved for user data, and performs steps necessary to install GSA on a Ramdisk. Next, SAVIT takes the part of GSA that resides in Low memory and saves that in a file called DSK1.GOLFCODE, and then takes the part residing in High memory into a file called DSK1.GOLFCODE1. It also tells us on-screen the length of each part as it's being saved. We use that information to update the two loaders that we supply with GSA, one for XB, the other for E/A or TIW. In GSA's normal use, all High memory from >B000 through >FFE6 is set aside for user data.

The setup in the PAB for these memory image operations is very simple. There's no need for record size or file type data. The only bytes that count in the PAB data are the first one, which is 06 for a save and 05 for a load, the third and fourth, which point to the buffer in VDP RAM, and bytes six and seven, which must contain the number of bytes to be saved in the file. From byte 9 onward, things are the same as for any other file type, with the length of the descriptor in byte 9, followed by the descriptor itself.

It is important to transfer the stuff you're saving from its memory location into the buffer in VDP RAM, and to make sure there's not more than the buffer area can handle. If our buffer were at >1020, for example, we could save no more than 10,167 bytes in a file, since any more would overwrite data at VDP RAM address >37D7, which is needed by the computer.

Once that's been done, and the PAB data written to VDP at some lower location than >1020, and the PAB+9 address passed to >8356, just a DSRLNK call makes the file on the disk. No OPEN operation or CLOSE operation is necessary.

The code section we've shown for the Golf Score Analyzer's SAVIT is somewhat convoluted. It also calls a subroutine called INTDI1, which has not been in any of our articles so far. We'll pass that along very soon. The saving code is not saved as part of the main program, but as part of the INSTALL program. This is done so that INSTALL can make changes in the main program itself, then save the two main files to a Ramdisk drive with the modifications in place. The part called GETINS is saved as part of the main program, but is only used when the user performs an installation process. Otherwise, the part at GETINS is overwritten by user data. Okay, so that was clear as mud, but it all works the way it was intended to, and that's what really counts.

Information for dealing with other file types, such as Internal, is available in the E/A Manual in pages 291 through 304. The setup information for byte 1 of the PAB, on page 293, is given bit-by-bit, so one must do some tedious work to figure out what the correct hex code for that byte should be. We use a hand-held calculator from Radio Shack that converts binary to octal or hex or decimal at the press of a key, and that comes in handy. (Ours is model EC-4030, which is probably out of production by now.)

None of the code in today's Sidebar is complete, nor will the whole thing even assemble correctly, but it's a series of pieces that you can use in your own programs. All of what's shown is taken from real programs we've written, and it worked when integrated into those programs.

We're not sure just yet what our next article will cover. We are writing these things many months ahead of publication, so maybe we'll take a vacation from writing until more of them have appeared in print. (As we write this, Part 2 has not yet appeared.) Perhaps we'll have some reader feedback questions to cover in Part 10. Then again, maybe we'll get into that rather amusing topic of "Off the End of the World", in which we'll relate some of the funny things that can happen when programming in Assembly.

```
* SOME CODE FRAGMENTS FOR FILE HANDLING OPERATIONS
* THESE ARE BITS AND PIECES TO BE INTEGRATED INTO PROGRAMS
* TO PERFORM FILE OPENINGS, READING, WRITING, AND CLOSINGS
* ALL PUBLIC DOMAIN SOURCE CODE
*
* REQUIRED REFERENCES
       REF  VMBW,VMBR,VSBW,VSBR
       REF  DSRLNK
*
* REQUIRED EQUATES
STATUS EQU  >837C
WS     EQU  >20BA
GPLWS  EQU  >83E0
PAB1   EQU  >1000
BUF    EQU  >1050
PABPNT EQU  >8356
*
*
* CODE AT LABEL START COULD BE USED TO GET A FILE NAME INPUT FROM THE
* USER.  IT USES SUBROUTINES WE'VE SUPPLIED PREVIOUSLY
*
START  LI   R15,RTNSTK   SET STACK FOR HIGH LEVEL SUBROUTINE
```

```
       LI   R0,3         ROW 1, COLUMN 4
       LI   R4,15        15 CHARACTERS WILL BE ACCEPTED
       BL   @CRSIN       USE CRSIN SUBROUTINE
       LI   R9,TEMSTR    POINT AT TEMPORARY STRING
       LI   R10,PAB1DT+9 POINT R10 AT FILE DESCRIPTOR LENGTH BYTE
       BL   @MOVSTR      MOVE STRING FROM TEMSTR TO PAB DATA
*  TWO WAYS TO OPEN ARE SHOWN
*  USE ONLY ONE OF THESE, DEPENDING ON YOUR NEED
*  FIRST IS THE LONG AND ACCURATE METHOD
*  SECOND IS A SHORTCUT
OPNF1  MOVB @INMD,@PAB1DT+1 OPEN WILL BE INPUT MODE
       LI   R0,PAB1      SET WRITE ADDRESS IN R0
       MOVB @PAB1DT+9,R2 GET DESCRIPTOR LENGTH BYTE INTO LEFT BYTE R2
       SRL  R2,8         RIGHT JUSTIFY SO R2 IS A WORD OF LENGTH
       AI   R2,10        ADD 10 TO INCLUDE THE PAB1DT LINE PLUS DESCRIPTOR
       LI   R1,PAB1DT    POINT R1 AT PAB DATA
       BLWP @VMBW        WRITE BYTES TO PAB LOCATION IN VDP RAM
       AI   R0,9         ADD NINE TO ADDRESS IN R0
       MOV  R0,@PABPNT   PLACE THAT ADDRESS AT >8356
       CLR  @STATUS      CLEAR GPL STATUS
       BLWP @DSRLNK      USE DSRLNK UTILITY
       DATA 8            REQUIRED DATA
       STST R14          STORE STATUS REGISTER IN R14
       ANDI R14,>2000    MASK ALL BUT BIT #2 IN R14
       JEQ  RDF1         IF ZERO, GO AHEAD TO READ FILE
       B    @OPNERR      ELSE TO OPNERR (SHOWN IN LAST ARTICLE)
* SHORTCUT METHOD FOR FILES OTHER THAN HARD DISK OR RS232 TYPE
OPNF1  MOVB @INMD,@PAB1DT+1 OPEN WILL BE INPUT MODE
       LI   R0,PAB1      SET WRITE LOCATION
       LI   R1,PAB1DT    SET SOURCE FOR PAB DATA
       LI   R2,25        25 BYTES - MAX FOR MOST PURPOSES
       BLWP @VMBW        WRITE DATA TO VDP
       AI   R0,9         ADD NINE
       MOV  R0,@PABPNT   MOVE TO >8356
       CLR  @STATUS      CLEAR STATUS
       BLWP @DSRLNK      USE LINKAGE VECTOR
       DATA 8            REQUIRED DATA
       STST R14          STORE STATUS REGISTER IN R14
       ANDI R14,>2000    MASK ALL EXCEPT BIT 2
       JEQ  READF1       IF ZERO, PROCEED TO READ
       B    @OPNERR      OPNERR SHOWN LAST ARTICLE
RDF1   MOVB @READF,R1    MOVE READ OPCODE INTO LEFT BYTE R1
       LI   R0,PAB1      PAB ADDRESS IN VDP
       BLWP @VSBW        WRITE ONE BYTE INTO PAB
       AI   R0,9         ADD NINE
       MOV  R0,@PABPNT   MOVE TO >8356
       CLR  @STATUS      CLEAR GPL STATUS
       BLWP @DSRLNK      USE DSRLNK
       DATA 8            REQUIRED DATA
       LI   R0,PAB1+1    SET TO SECOND BYTE OF PAB IN VDP
```

```
       BLWP @VSBR        READ INTO LEFT BYTE R1
       SRL  R1,13        SHIFT R1 RIGHT BY 13 BITS
       JEQ  READON       IF ZERO, NO ERROR IN DSR OPERATION
       CI   R1,5         IF ERROR = 5, END OF FILE HAS BEEN REACHED
       JEQ  CLSF1        IF SO, CLOSE THE FILE
       B    @FILERR      ELSE SOME OTHER ERROR, REPORT THAT TO USER
READON LI   R0,PAB1+5    POINT AT PAB+5 IN VDP RAM
       BLWP @VSBR        READ THAT BYTE INTO LEFT BYTE R1
* FOR D/V FILES, THE BYTE AT PAB+5 IS THE LENGTH OF THE RECORD JUST READ
       MOVB R1,R2        MOVE BYTE INTO R2
       SRL  R2,8         RIGHT JUSTIFY LENGTH IN R2
       MOVB R1,@TEMSTR   MOVE BYTE TO TEMSTR
       LI   R0,BUF       POINT TO BUFFER LOCATION IN VDP
       LI   R1,TEMSTR+1  CONTENT GOES TO TEMSTR+1
       BLWP @VMBR        READ CONTENT OF RECORD FROM VDP BUFFER
* CODE WOULD BE INSERTED HERE TO MOVE THE RECORD FROM TEMSTR, OR DISPLAY
* THE RECORD ON THE SCREEN, OR ANY OTHER OPERATION YOU DESIRE
       JMP  RDF1         JUMP BACK TO READ NEXT RECORD
CLSF1  LI   R0,PAB1      POINT TO PAB ADDRESS
       MOVB @CLOSEF,R1   GET CLOSE OPCODE IN LEFT BYTE R1
       BLWP @VSBW        WRITE OPCODE TO PAB
       AI   R0,9         ADD NINE
       MOV  R0,@PABPNT   PLACE AT >8356
       CLR  @STATUS      CLEAR STATUS
       BLWP @DSRLNK      CALL DSRLNK
       DATA 8            REQUIRED DATA
* FROM HERE, GO ON TO NEXT PROGRAM OPERATION
*
* CODE BELOW OPENS A D/V 80 FILE FOR WRITING, THEN WRITES THE RECORD STASHED
* AT TEMSTR TO THE FILE
*
OPNF2  MOVB @OUTMD,@PAB1DT+1 OPEN WILL BE OUTPUT MODE
       LI   R0,PAB1      SET WRITE ADDRESS IN R0
       MOVB @PAB1DT+9,R2 GET DESCRIPTOR LENGTH BYTE INTO LEFT BYTE R2
       SRL  R2,8         RIGHT JUSTIFY SO R2 IS A WORD OF LENGTH
       AI   R2,10        ADD 10 TO INCLUDE THE PAB1DT LINE PLUS DESCRIPTOR
       LI   R1,PAB1DT    POINT R1 AT PAB DATA
       BLWP @VMBW        WRITE BYTES TO PAB LOCATION IN VDP RAM
       AI   R0,9         ADD NINE TO ADDRESS IN R0
       MOV  R0,@PABPNT   PLACE THAT ADDRESS AT >8356
       CLR  @STATUS      CLEAR GPL STATUS
       BLWP @DSRLNK      USE DSRLNK UTILITY
       DATA 8            REQUIRED DATA
       STST R14          STORE STATUS REGISTER IN R14
       ANDI R14,>2000    MASK ALL BUT BIT #2 IN R14
       JEQ  WRTF2        IF ZERO, GO AHEAD TO WRITE FILE
       B    @OPNERR      ELSE TO OPNERR (SHOWN IN LAST ARTICLE)
WRTF2  MOVB @TEMSTR,R1   GET LENGTH OF RECORD IN LEFT BYTE R1
       LI   R0,PAB1+5    POINT TO RECORD LENGTH BYTE OF PAB
       BLWP @VSBW        WRITE LENGTH TO PAB
```

```
       MOVB R1,R2        PLACE LENGTH IN LEFT BYTE R2
       SRL  R2,8         RIGHT JUSTIFY LENGTH IN R2
       LI   R1,TEMSTR+1  POINT TO STRING CONTENT
       LI   R0,BUF       POINT AT BUFFER IN VDP
       BLWP @VMBW        WRITE RECORD CONTENTS TO VDP
       MOVB @WRITEF,R1   GET WRITE OPCODE IN R1
       LI   R0,PAB1      POINT TO START OF PAB
       BLWP @VSBW        WRITE THE OPCODE BYTE TO VDP
       AI   R0,9         ADD 9
       MOV  R0,@PABPNT   MOVE TO >8356
       CLR  @STATUS      CLEAR GPL STATUS BYTE
       BLWP @DSRLNK      CALL DSR LINKAGE
       DATA 8            REQUIRED DATA
       LI   R0,PAB1+1    POINT TO SECOND BYTE OF PAB
       BLWP @VSBR        READ THAT BYTE INTO R1
       SRL  R1,13        SHIFT R1 RIGHT 13 BITS
       JEQ  WRTON        IF ZERO, NO ERROR, SO GO ON
       B    @FILERR      ELSE BRANCH TO ERROR HANDLING
WRTON
* THIS LABEL WOULD GET ANOTHER RECORD READY AT TEMSTR, THEN BRANCH BACK TO
* WRTF2, OR ELSE IF FINISHED WOULD BRANCH BACK TO CLSF1 TO CLOSE THE FILE
* THIS FILE, SINCE IT'S USING THE SAME PAB AND BUFFER, CAN'T BE OPEN WHILE
* FILE 1 IS ALSO OPEN
*
* REQUIRED DATA SECTION
* THE FOLLOWING DATA SOURCE LINES ARE REQUIRED
*
* THIS PAB DATA AND MODE BYTES APPLY TO D/V 80 FILES
PAB1DT DATA >0014,BUF,>5000,>0000,>000F
       BSS  15
INMD   BYTE >14          BYTE FOR INPUT OF DISPLAY/VARIABLE FILE
OUTMD  BYTE >12          BYTE FOR OUTPUT OF DISPLAY/VARIABLE FILE
APPMD  BYTE >16          BYTE FOR APPEND OF DISPLAY/VARIABLE FILE
UPDAMD BYTE >10          BYTE FOR UPDATE MODE OF D/V FILE -NOT RECOMMENDED
WRITEF BYTE 3            OPCODE FOR WRITE OPERATION
READF  BYTE 2            OPCODE FOR READ OPERATION
CLOSEF BYTE 1            OPCODE FOR CLOSE OPERATION
*
* THE DATA BELOW IS A PAB SETUP PLUS THE MODE BYTES
* FOR A D/F 80 FILE
*
PAB2DT DATA >0001,BUF,>5050,>0000,>000F
       BSS  15
FINMD  BYTE >05          INPUT MODE BYTE FOR DISPLAY/FIXED FILE TYPE
FOUTMD BYTE >03          OUTPUT MODE BYTE FOR D/F FILES
FAPPMD BYTE >07          APPEND MODE BYTE FOR D/F FILES
FUPDMD BYTE >01          UPDATE MODE BYTE FOR D/F FILES
*
* BELOW IS CODE AND DATA SAMPLE FOR MAKING MEMORY IMAGE FILES
* DERIVED FROM OUR GSA PROGRAM
```

```
*
       AORG >B000        THIS CODE STARTS AT >B000
INSTDT DATA >0500,>1020,0,ENINST-SAVDT,>000C
       TEXT 'DSK1.INSTALL'
SAVBYT BYTE 6
LDBYTE BYTE 5
* THIS SECTION IS USED TO LOAD THE INSTALL CODE WHEN NECESSARY
GETINS
       MOVB @LDBYTE,@INSTDT PUT LOAD OPCODE IN FIRST BYTE OF PAB DATA BLOCK
       LI   R0,PAB1      POINT TO PAB LOCATION
       LI   R1,INSTDT    POINT R1 AT DATA BLOCK
       LI   R2,22        22 BYTES IN PAB
       BLWP @VMBW        WRITE TO VDP
       AI   R0,9         ADD 9
       MOV  R0,@PABPNT   TO >8356
       CLR  @STATUS      CLEAR STATUS
       BLWP @DSRLNK      LOAD FILE INTO VDP BUFFER
       DATA 8            REQD DATA
       LI   R0,>1020     POINT AT BUFFER
       MOV  @INSTDT+6,R2 LENGTH OF FILE INTO R2
       LI   R1,SAVDT     FILE STARTS AT LOCATION SAVDT
       BLWP @VMBR        GET CODE INTO MEMORY
       B    @INSTAL      THEN BRANCH TO NOW-INSTALLED INSTALL CODE
ENDAUX EQU  $
*  SAVER - STASHES PROGRAM
*  AS MEMORY IMAGE FILE
*  15 JUN 89
       DEF  SAVIT        DEFINED ENTRY POINT
SAVDT  DATA >0600,>1020,0,ENMAIN-GPLLNK,>000D
       TEXT 'DSK1.GOLFCODE'
SAVDT1 DATA >0600,>1020,0,ENDAUX->A000,>000E
       TEXT 'DSK1.GOLFCODE1'
SAVIT
       MOV  R11,@>8300   STASH REGISTER 11
       LWPI WS           LOAD OUR WS
       LI   R15,RTNSTK   SET R15 TO RETURN ADDRESS
       BL   @CLS         CLEAR SCREEN
       LI   R9,INSTDT+9  DISPLAY FILE DESCRIPTOR
       LI   R0,SCRWID*5+4 AT ROW 6, COLUMN 5
       BL   @DISLI       USING SUBROUTINE
       MOV  @INSTDT+6,R5 BRING LENGTH OF INSTALL SECTION INTO R5
       LI   R0,SCRWID*7+15 SCREEN LOCATION ROW 8, COLUMN 16
       BL   @INTDI1      DISPLAY INTEGER NUMBER ON SCREEN
       MOVB @SAVDT,@INSTDT SET FOR SAVING INSTALL PART
       MOV  @INSTDT+6,R2 GET LENGTH IN R2
       LI   R1,SAVDT     POINT TO START OF CODE TO BE SAVED
       LI   R0,>1020     BUFFER ADDRESS
       BLWP @VMBW        WRITE TO BUFFER
       LI   R0,PAB1      SET TO WRITE PAB
       LI   R2,22        22 BYTES
```

```
        LI   R1,INSTDT    PAB DATA FOR INSTALL SECTION
        BLWP @VMBW        WRITE TO PAB IN VDP RAM
        AI   R0,9         ADD 9
        MOV  R0,@PABPNT   AT >8356
        CLR  @STATUS      CLEAR
        BLWP @DSRLNK      PERFORM WRITING OF FILE INSTALL TO DSK1
        DATA 8            DATA
SAVPT2
        LI   R9,SAVDT+9   GET DESCRIPTOR FOR GOLFCODE FILE
        LI   R0,SCRWID*9+4 ROW 10, COLUMN 5
        BL   @DISLI       DISPLAY FILE DESCRIPTOR
        MOV  @SAVDT+6,R5  GET LENGTH OF LOW-MEMORY CODE SECTION IN R5
        LI   R0,SCRWID*11+15  ROW 12, COLUMN 16
        BL   @INTDI1      DISPLAY INTEGER
        LI   R0,>1020     POINT TO BUFFER
        LI   R1,GPLLNK    GPLLNK IS AT START OF PROGRAM'S LOW MEM PORTION
        LI   R2,ENMAIN-GPLLNK ENMAIN IS END OF LOW MEM PART OF CODE
        BLWP @VMBW        WRITE TO BUFFER
        LI   R0,PAB1      SET FOR PAB
        LI   R1,SAVDT     POINT TO DATA
        LI   R2,23        23 BYTES
        BLWP @VMBW        WRITE PAB TO VDP
        AI   R0,9         ADD 9
        MOV  R0,@PABPNT   TO >8356
        CLR  @STATUS
        BLWP @DSRLNK      WRITE FIRST SECTION OF CODE TO FILE
        DATA 8
        LI   R9,SAVDT1+9  GET DESCRIPTOR FOR SECOND FILE
        LI   R0,SCRWID*15+4 SCREEN ROW 16, COLUMN 5
        BL   @DISLI       DISPLAY THE DESCRIPTOR
        MOV  @SAVDT1+6,R5 GET LENGTH OF HIGH MEMORY SECTION IN R5
        LI   R0,SCRWID*17+15   ROW 18, COLUMN 16
        BL   @INTDI1      DISPLAY LENGTH (AS DECIMAL NUMBER)
        LI   R0,>1020     POINT TO BUFFER
        LI   R1,>A000     START OF HIGH MEMORY
        LI   R2,ENDAUX->A000 LENGTH OF HIGH MEM PORTION
        BLWP @VMBW        WRITE INTO BUFFER
        LI   R0,PAB1      SET FOR PAB
        LI   R1,SAVDT1    POINT TO PAB DATA
        LI   R2,24        24 BYTES TO WRITE
        BLWP @VMBW        WRITE PAB TO VDP
        AI   R0,9         ADD 9
        MOV  R0,@PABPNT
        CLR  @STATUS
        BLWP @DSRLNK      WRITE FILE GOLFCODE1 TO DISK
        DATA 8
GEXIT   LWPI GPLWS        LOAD GPL WORKSPACE
        MOV  @>8300,R11   REPLACE R11
        RT                RETURN
INSTAL
```

```
* THE CODE FOR THE INSTALLATION PROCESS FOLLOWS HERE (NOT SHOWN)
* IT ENDS AT A LABEL CALLED ENINST
TEMSTR BSS  81           TEMPORARY STORAGE LOCATION FOR RECORD
* THE NUMBER IN THIS BSS MUST BE ONE MORE THAN THE LARGEST STRING LENGTH
* EXPECTED IN THE PROGRAM'S EXECUTION
RTNSTK DATA 0            RETURN ADDRESS STACK NEEDED BY CRSIN
```

1.10. The Art Of Assembly — Part 10. Off The End Of The World

By Bruce Harrison

When we started out with our TI Home computer, we had only the console, with Extended Basic module. Storage was by cassette tape, and painfully slow. There are times when we consider those days to be golden.

Programming in Extended Basic remained our mode of operation for several years, even after adding the PE Box, 32K memory, and of course a disk drive. Frustration set in when we kept running into the limitations imposed by the high-level language. Speed of execution became our greatest desire, and the hardest thing to achieve.

For all its drawbacks, however, Extended Basic had one big advantage. The Interpreter was always in control of the machine. One could make all kinds of mistakes in the "program" fed to that interpreter, but it would always be able to stop the program and make that awful "BOOP" sound. We were protected and safe, like Christopher Columbus sailing around in the Mediterranean, there was no chance of sailing off the end of the world.

Of course frustration got to us, and we chose to seek the shorter route to India, through Assembly language. Talk about uncharted seas! No more BOOP! Once something assembled correctly, no more "SYNTAX ERROR". Just plain crashes, lockups, and screen displays from outer space.

In this new venture we found many ways to send the computer sailing "Off the End of the World". We'll recount a couple of those today.

One of the easiest ways to put the computer into a bottomless abyss is to try a VMBW operation when R2 is zero. Nobody ever made spectacular graphics like that can do. The screen fills with all kinds of patterns, multiple colors, changing apparently at random. It usually ends with the screen looking as if sync has been lost, and retrace lines are faintly visible, but flashing on and off at random positions. If we're lucky, it eventually gets back to somewhere stable, and then **FCTN = (QUIT)** becomes active so we can escape from the abyss without resorting to the ON-OFF switch.

Another such experience happened when we were trying to get some of our music programs to run on Geneve. In these programs, we had to supply a DSRLNK to operate our Assembly program under Extended Basic. Early testing showed that the DSRLNK we use for the TI would cause a lockup on the Geneve. One of our customers then supplied us the source code for a DSRLNK that he knew would work on his Geneve.

Eagerly we accepted, and placed this different DSRLNK in our source code. It assembled without error, so we loaded the Object file for testing on our faithful TI. Everything worked fine until we made a selection from the menu. The DSRLNK worked, in that it did bring in a file, and music played. There was just one small problem. The DSRLNK had turned our screen totally blank. It seems video output had been completely disabled, so the monitor was just a black screen free-running.

Nevertheless, the music continued to play until the number was over. We wondered if the Menu was on our invisible screen. The only way to tell was to pretend it was there and make a selection. To our complete surprise, the number selected loaded in and played, just as if the menu were visible.

Of course we didn't think it would be wise to ask our TI customers to put up with the blank menu just so our Geneve customers would be happy. We abandoned that DSRLNK. Actually, we first studied its source code for many hours, trying to decipher how it managed to blank the screen. No dice.

Much later, we discovered that the original DSRLNK we used on the TI would work on the Geneve if we entered our menu via an Option 5 loader, instead of through Extended Basic. We still don't know why that works! It has also been our experience that what works on one Geneve may not work on all of them.

Another fine mess we've created for ourselves was in the business of doing a VMBR operation without having set R2 to the correct value. This can have disastrous results, especially if you're reading into a data area that's ahead of most of your code. Zap, you've overwritten part of your program, and the computer rapidly gets "lost in space". We've done this particular trick on a number of occasions, and while it doesn't make any spectacular screen displays, it does very effectively lock up the computer so that only the ON-OFF switch has any effect whatsoever.

On one notable occasion, we were trying to bring in eighty characters from a VDP File buffer. As it happened, we had the number eighty handy at a data location somewhere in low memory (i.e. EIGHTY DATA 80). We could have written MOV @EIGHTY,R2, or LI R2,80. No, we made a slight mistake.

We had placed in our source code a sort of mixed metaphor, so it read LI R2,EIGHTY. That put the address of the data item EIGHTY into R2, not the value 80. The address, being a number something like >29E2, made our read operation from VDP bring 10,722 bytes from VDP into memory, wiping out our entire menu driver, and of course locking the computer up solid.

We stared at the printout of that source code for days on end without noticing that error. We even took the printout with us to Philadelphia while visiting a friend there. Near the end of our stay in Philadelphia, away from our beloved computer, we noticed the error, and of course then everything made sense.

The three hour drive back from Philly was filled with tension. Was that the only error? Would it work okay? Was there yet another error somewhere that we hadn't found? We couldn't bear the waiting to get home and try this fix. We did try it, and of course that cured the problem. We were able to sleep that night after all.

Sleepless nights are just another hazard in this profession of computer programmer. There is a kind of tension that builds up when something isn't working right which is more deadly to sleep than 1000 cups of coffee. "There must be a way to make this work! I know there's a way! What if I did another small operation before proceeding with. . ."

Sleep won't come, but tiredness does, and even while you're struggling with that one possible error, your lack of sleep is inducing others. Now you start getting syntax errors during the assembly run. You typed LI R0.14 instead of LI R0,14. Now you go back and fix that, but you're still not closing in on the original problem, and every load, save, and assemble is eating up the wee hours without any payoff.

Sometimes the exhaustion gets the better of us, and we "sleep on" the problem. This usually helps, and a fresh look at the source code in the morning brings a solution. It's always a good idea to take a sleep break, hard though that may be to do.

The hardest problems of all to find are the ones where some key step has been omitted. If something that's there is wrong, that's much easier to spot than something that's not there. Of course there is a debugger supplied with the E/A package. Why not use that to isolate the problem? We seldom resort to trying TI's debugger. Here's why.

First, one must know the actual addresses to use when the program is resident in memory. To be perfectly accurate in tracing a problem, this means running a fully listed assembly, generating a mountain of fan-fold paper which will be useful only until the problem is corrected. Second, one must make an educated guess as to where the problem occurs, and set a breakpoint before the suspect operation. If one could then single-step up until the disaster, the debugger would have earned its keep. One can't do that with TI's debugger, so the whole process becomes a tiresome series of stopping, reloading, taking another guess, then trying again.

Finally, many of the things we do involve situations where the debugger can't even be loaded into memory, because there's not enough room after our program loads, or else it will overwrite something we AORG'd when it loads.

Now and then our ability to create havoc on the computer is enhanced by the presence in our house of small animals. We have cats, and some of them have discovered our TI Computer console. One likes to curl up in a tight ball on top of the console while we're using it. She likes the warmth from its electronic innards. Sometimes her body sort of overlaps the keys, making it impossible to press the keys **1**, **2** or **3**.

Once your author committed the error of leaving the room while this darling little cat was atop the console. During his absence, the cat decided to walk across the keyboard, and when he returned, nothing from the Ramdisk menu would work. It took some time, but eventually a scenario emerged. The computer had been left with the Ramdisk menu on-screen. The cat must have hit key 5, which activates the Horizon Configure program. She'd then pressed any key, and put us into configure. The next key she must have stepped on was D for drives, then Q. This renamed drive #5 as drive Q. By the time we returned to the room, the screen had blanked. We hit **FCTN = (QUIT)** to reset, and the menu came back, but those programs listed on the menu as being on drive 5 could not be accessed, since we no longer had a drive 5 on the system.

Pretty clever cat, that. It took quite a while to figure out what had happened, but only a minute to straighten out.

Another of our cats learned how to play a game with our old Star Delta 10 printer. She loved watching the printhead zoom back and forth, and seeing the paper advance. After a while, she figured out how to step on the On-Line button, then she'd just keep one little paw on the Line Feed button and watch as half a ream of paper headed down the back of the printer. We stopped using the Delta-10 some time ago, and she hasn't learned yet how to do this trick on the Star NX-1000.

As we write this, it's late on a summer evening, and the air conditioner is making the room a little chilly, so there's a cat curled up beside the console, under the desk lamp. The lamp keeps her warm, but now and then she shifts positions, and knocks papers off the desk and onto the floor.

But she is cute, cuddly, and purrs so sweetly when petted. (We are suckers for purring cats.) Besides, humans exist just to pick up papers, straighten out Ramdisks, and of course to feed cats. Don't believe this? Ask any cat.

Enough of this rambling. Today's Sidebar has a couple of little subroutines you may find useful. As we promised, the subroutine INTDIS we mentioned last time is here, and in two versions. One version displays a one-word integer regardless of sign, so its screen output ranges from zero through 65535. The other takes the sign bit into account, and displays positive numbers from 0 through 32767 and negative numbers from -1 through -32768.

These of course can't be used together in a program as they're shown here, because labels would be duplicated. With changes to the labels, one could use both in a program.

The method used in these subroutines was suggested by one for the PC, in one of Peter Norton's books. First, we place the number right justified in the register pair R5-R6, by moving R5 to R6 and then clearing R5. We successively divide the number by ten. After each division, R5 contains the integer quotient and R6 contains the remainder. The first such divide makes R6 (remainder) equal the least significant digit. Dividing the quotient from this operation by ten makes the remainder equal the next significant digit, and so on. This continues until we find the quotient in R5 is zero, which means the remainder then in R6 is the most significant digit in the number. Of course Peter Norton had the luxury of doing this on a PC, so all he had to do to save each digit was to "PUSH DX" onto the established stack, then "POP" the digits back into a register for display. Here, we had to create our own stack in memory and used R14 as a pointer to that stack. The code shown in the Sidebar is well annotated, so you should be able to follow its operation. On entry, R0 should point to a screen location where the decimal point would be if there were one. That is, the spot just beyond where the least significant digit will appear. The value in R5 will be destroyed by the subroutine. The screen address placed in R0 before entry will be restored upon exit.

The other subroutine is for those cases where you want to see the hex content of some word. The method used here was borrowed from another book of subroutines for the PC. By isolating the right-most nybble each time, and successively shifting a register to the right, we place the four "digits" on a stack, then strip them off and display them in correct order.

We hope these subroutines will be useful in your programs. We also hope that we've gotten a few chuckles from our readers. Those who've done some programming in Assembly will no doubt have stories of a similar nature to recount.

In our next article we promise to remain on a serious note throughout, and to pass along some useful tips.

```
* THREE USEFUL SUBROUTINES - TWO THAT DISPLAY INTEGERS IN DECIMAL,
* ONE THAT DISPLAYS A WORD VALUE IN HEX NOTATION
*
* THIS FIRST VERSION OF INTDIS SHOWS INTEGER AS POSITIVE OR NEGATIVE NUMBER
* WILL DISPLAY NUMBERS FROM 0 THRU 32767 POSITIVE, OR THROUGH -32768 NEGATIVE
* NUMBER DISPLAYED IS RIGHT JUSTIFIED FROM POSITION POINTED TO BY R0 ON ENTRY
* TWO ENTRY POINTS ARE PROVIDED:
* USE INTDIS IF R5 POINTS TO INTEGER (WORD) IN MEMORY
* USE INTDI1 IF R5 CONTAINS INTEGER TO DISPLAY
* ON ENTRY, R0 POINTS TO LOCATION ONE SPOT BEYOND LEAST SIGNIFICANT DIGIT
* ON EXIT, R0 POINTS TO SAME LOCATION
* LEADING ZEROS ARE SUPPRESSED
* R1, R5, R6, AND R14 ARE USED AND MODIFIED BY SUBROUTINE
*
INTDIS MOV  *R5,R5       ENTRY WHEN R5 POINTS TO NUMBER - MOVE VALUE TO R5
INTDI1 LI   R14,INTSTK   ENTRY WHEN R5 CONTAINS NUMBER - POINT R14 TO STACK
       MOV  R5,R6        MOVE R5 VALUE TO R6
       ANDI R6,>8000     MASK OFF ALL BUT SIGN BIT
       MOVB R6,@NEGFLG   MOVE LEFT BYTE R6 TO FLAG BYTE
       JEQ  INTLOP       IF ZERO, JUMP
       NEG  R5           ELSE MAKE R5 POSITIVE VALUE
INTLOP MOV  R5,R6        MOVE R5 VALUE TO R6
```

```
       DEC  R0           DECREMENT SCREEN LOCATION
       CLR  R5           CLEAR R5
       DIV  @TEN,R5      DIVIDE R5-R6 REGISTER PAIR BY 10
       SWPB R6           GET REMAINDER IN LEFT BYTE R6
       AB   @NUMBER,R6   ADD NUMBER MASK BYTE (>30)
       MOVB R6,*R14+     PLACE BYTE IN STACK AND INCREMENT POINTER
       MOV  R5,R5        SEE IF QUOTIENT WAS ZERO
       JNE  INTLOP       IF NOT, GO BACK FOR NEXT DIGIT
       MOVB @NEGFLG,R1   MOVE FLAG BYTE INTO LEFT BYTE R1
       JEQ  DISLOP       IF ZERO, JUMP AHEAD
       DEC  R0           ELSE DECREMENT SCREEN LOCATION
       MOVB @MINUS,R1    GET MINUS SIGN IN LEFT BYTE R1
*      AB   @OFFSET,R1   COMMENTED OUT - NEEDED IF RUN FROM XB
       BLWP @VSBW        DISPLAY MINUS SIGN
       INC  R0           POINT TO NEXT SPOT ON SCREEN
DISLOP DEC  R14          DECREMENT STACK POINTER
       MOVB *R14,R1      MOVE BYTE FROM STACK INTO LEFT BYTE R1
*      AB   @OFFSET,R1   COMMENTED OUT - USE ONLY IF RUN FROM EXTENDED BASIC
       BLWP @VSBW        WRITE DIGIT TO SCREEN
       INC  R0           INCREMENT SCREEN LOCATION
       CI   R14,INTSTK   COMPARE R14 TO BEGINNING OF STACK
       JGT  DISLOP       IF STILL GREATER, DISPLAY ANOTHER DIGIT
       RT                ELSE RETURN TO CALLING PROGRAM
*
*
* SECOND VERSION OF INTDIS - IGNORES SIGN
* NUMBER MAY RANGE FROM 0 THROUGH 65,535 (ONE WORD)
* TWO ENTRY POINTS ARE PROVIDED
* USE INTDIS WHEN R5 POINTS TO A WORD IN MEMORY TO DISPLAY
* USE INTDI1 WHEN NUMBER TO BE DISPLAYED IS ALREADY IN R5
* ON ENTRY AND EXIT, R0 POINTS TO LOCATION ONE SPOT BEYOND LAST DIGIT
* DISPLAY RIGHT JUSTIFIES NUMBER FROM THERE
* SUPPRESSES LEADING ZEROS  (I.E. NUMBER 00045 WILL DISPLAY AS 45)
* R1, R5, R6, AND R14 ARE USED AND MODIFIED BY SUBROUTINE
*
INTDIS MOV  *R5,R5       MOVE INTEGER FROM MEMORY TO R5
INTDI1 LI   R14,INTSTK   POINT R14 AT STACK
INTLOP MOV  R5,R6        MOVE R5 VALUE INTO R6
       DEC  R0           DECREMENT SCREEN POINTER
       CLR  R5           CLEAR REGISTER 5
       DIV  @TEN,R5      DIVIDE R5-R6 REGISTER PAIR BY TEN
       SWPB R6           PLACE REMAINDER IN LEFT BYTE R6
       AB   @NUMBER,R6   ADD NUMERIC MASK (>30)
       MOVB R6,*R14+     STASH NUMBER ON STACK AND INCREMENT POINTER
       MOV  R5,R5        IS QUOTIENT ZERO
       JNE  INTLOP       IF NOT, THERE ARE MORE SIGNIFICANT DIGITS
DISLOP DEC  R14          DECREMENT STACK POINTER
       MOVB *R14,R1      MOVE BYTE FROM STACK TO LEFT BYTE R1
*      AB   @OFFSET,R1   COMMENTED OUT - USED ONLY WHEN RUN FROM EXTENDED BASIC
       BLWP @VSBW        WRITE TO SCREEN
```

```
       INC  R0           POINT AT NEXT SCREEN LOCATION
       CI   R14,INTSTK   COMPARE POINTER TO BEGINNING OF STACK
       JGT  DISLOP       IF GREATER, GO BACK FOR ANOTHER DIGIT
       RT                ELSE RETURN
*
* SUBROUTINE HEXDIS DISPLAYS A HEX NUMBER ON SCREEN
* ON ENTRY, NUMBER TO BE DISPLAYED IS IN R5
* SCREEN LOCATION FOR FIRST "DIGIT" IS IN R0
* ON EXIT, R5 CONTAINS ORIGINAL NUMBER, R0 POINTS TO SPOT BEYOND LAST "DIGIT"
* R4, R6, R0, R1, R14 ARE USED AND CHANGED BY THE SUBROUTINE
*
HEXDIS
       LI   R4,4         FOUR "DIGITS" TO DISPLAY
       LI   R14,INTSTK   POINT AT STACK LOCATION
STKLP  MOV  R5,R6        MOVE R5 VALUE TO R6
       ANDI R6,>000F     MASK ALL BUT LAST NYBBLE
       SWPB R6           PLACE IN LEFT BYTE R6
       CB   R6,@TEN+1    COMPARE TO 10
       JLT  NUM          IF LESS, USE NUMBER MASK
       AB   @ALP,R6      ELSE ADD "A"-10 TO BYTE
       JMP  STCKIT       THEN JUMP
NUM    AB   @NUMBER,R6   ADD >30 TO BYTE
STCKIT MOVB R6,*R14+     PLACE BYTE IN STACK, INCREMENT POINTER
       SRC  R5,4         SHIFT R5 FOR NEXT NYBBLE
       DEC  R4           DECREMENT NYBBLE COUNT
       JNE  STKLP        IF NOT ZERO, GO BACK
       LI   R4,4         ELSE RESET R4 TO FOUR
HEXLOP DEC  R14          DECREMENT STACK POINTER
       MOVB *R14,R1      MOVE BYTE FROM STACK INTO R1
*      AB   @OFFSET,R1   COMMENTED OUT - FOR XB USE
       BLWP @VSBW        WRITE TO SCREEN
       INC  R0           INCREMENT SCREEN LOCATION
       DEC  R4           DECREMENT CHARACTER COUNT
       JNE  HEXLOP       IF NOT ZERO, WRITE ANOTHER CHARACTER
       RT                RETURN TO CALLING PROGRAM
*
*
* REQUIRED DATA SECTION - GOOD FOR ALL THREE SUBROUTINES
*
TEN    DATA 10           JUST THE NUMBER 10 AS A WORD
ALP    BYTE 55           ASCII VALUE FOR "A" WITH TEN SUBTRACTED (65-10=55)
INTSTK BSS 5             RESERVED SPACE FOR DIGITS (FIVE)
MINUS  TEXT '-'          MINUS SIGN READY FOR USE
NUMBER BYTE >30          NUMERIC VALUE MASK = ASCII FOR ZERO
NEGFLG BYTE 0            FLAG BYTE FOR NEGATIVE NUMBERS
OFFSET BYTE >60          USED ONLY IF SUBROUTINE RUN UNDER XB
```

1.11. The Art Of Assembly — Part 11. Structuring Data

By Bruce Harrison

Back in Number 3 of this series, we showed a Menu Driver, derived from our Golf Score Analyzer. To make that one driver work for a number of different menus, we had to organize the data for the menus in a particular fashion, so that the driver could, among other things, auto-center the menu items between the top and the bottom of the screen, and use a loop to display the items in the body of the menu. If you'll recall, the data also was organized as strings, so that the display subroutine would know how many characters were to be displayed on each menu line.

That was just one example of how making a specific structure for the data could make the code much more efficient in both memory and speed. In this article, we'll extend that concept a bit, showing how we structured the data for those "Fill-in" screens in the Golf Score Analyzer. By entering this data in a carefully structured manner, we were able to provide one small subroutine that could produce any of a number of screens full of prompts. This saved memory, and also made the display of whole screens full of data happen very quickly.

The Sidebar shows the code for invoking the DISSCR (Display Screen) subroutine, the subroutine itself, and the data for one of those screens. There are two pointers required, (R1 and R6) and two separate data sections for each screen. The "Location" block (LOCTB1) is a set of words of data, each giving the starting screen location for a corresponding text string. The group of text strings (TXTTB1) is just that, each string having a length byte, followed by the text to be displayed.

The subroutine is nothing but a simple loop which continues displaying the strings at the desired locations until it finds the "stop" screen location. We used >FFFF as our stop code because that's an impossible number for screen location. This subroutine uses DISLI to display each string on the screen, and takes advantage of the fact that DISLI leaves the pointer at the length byte of the next string each time it displays one. The version of DISLI shown here is a very simplified one for use with programs to run under the E/A module. When used with Extended Basic, a more complex version of DISLI was used, (see #3 in this series) so that the XB Offset could be added to each character in the string before displaying it.

Harking back for a moment, you'll see here again our old friend SCRWID used to designate a screen row of characters. In the Golf Score Analyzer, everything is done on the normal Graphics screen, so SCRWID EQU 32 was included early in the source code.

Again we've used the Assembler to do math for us, so an entry like SCRWID*4+2 will be computed by the Assembler to the correct value for Row 5, Column 3 of the screen. This made it much easier for us, during development of the program, to adjust our screen positions when necessary, since there's an easy mnemonic for translating our source code to Row, Column form.

In this particular case, we didn't display anything on the top row of the screen, so the first three entries in LOCTB1 have SCRWID plus one less than the desired column. If we were to have these displayed on Row 1 the entries would read DATA 2,14,21.

In its application, this particular "fill-in" screen is used in several places, both for user entry and for displaying data from a golfer's old records. The subroutine DISSCR is of course used in other places, with other data, to produce screens for adding and editing course data, for example.

There's another hidden asset in this data structure, in that some strings which were needed in other places than within the fill-in screen were given labels. This made it easy to use small parts of this big group of strings wherever we needed them. For example, if we wanted the word "PUTTS" to appear at Row 6, column 5, we could accomplish that by:

```
            LI   R0,SCRWID*5+4
            LI   R1,PUTSTR
            BL   @DISLI
```

That idea was of course used in the program, to "recycle" parts of that long array of strings.

As you're probably all too aware by now, we are very fond of doing things with loop structures in our code. We were not always so skillful in using them, and some of our earliest attempts at programming in Assembly used very long sequences of operations where simple loops would have done the job.

Let's just for the moment assume that the fill-in screen data we've shown in today's Sidebar were not organized in this fashion. Let's suppose it was just text lines, not strings.

Let's also assume we didn't make that table of screen locations for the various strings. Our text part would take up fewer bytes in memory without the length byte before each text line:

```
TXTTB1      TEXT 'COURSE:'
            TEXT 'PAR:'
            TEXT 'DATE:'
            (and so on)
```

Now our code would not be able to operate on a simple loop basis, but would need separate steps to display each part. For example:

```
DISSCR      LI   R0,SCRWID+2
            LI   R2,7
            LI   R1,TXTTB1
            BLWP @VMBW
            A    R2,R1
            LI   R0,SCRWID+14
            LI   R2,4
            BLWP @VMBW
            (and so on)
```

Also, of course, each fill-in screen we used would need its own tailor-made code section like that shown above. A DISLI subroutine would be unnecessary, but at a very high cost.

We think this small example will once and for all make our point, that bytes spent in a well-organized data structure can be saved many times over when they permit a simple loop in the code section to display a bunch of data.

We've applied this concept of looping and data structuring in our music programs, and this has contributed materially to our ability to cram tons of music onto our disks. In the music, the data is organized into measures, and labels are placed at the beginnings and ends of measures. Then our "action" part of the code can use loops to repeat playing of sections of the music, and even to perform what musicians call a "Da Capo" (That's Italian for, literally "From the Head", or more colloquially, "From the Top"), which repeats the entire piece.

Like anything done in Assembly, our method won't feel comfortable to some programmers, nor is it necessarily the "best" way to accomplish the task. We use our own methods as examples of how structure in both data and code can become your servant, and invite our readers to modify and improve on our methods to their heart's content.

There are things referenced in today's Sidebar that are not included there, such as the CLS subroutine, for which we gave two examples earlier, and the High level subroutine return SUBRET, also given previously. (See #2 of this series in the July 1991 issue.)

This would be kind of a short article if we stopped here, yet we don't want to open a new and major topic either. Instead, let's go into a discussion on our design philosophy for developing programs. This may cause gnashing of teeth in some circles, but the following are our opinions, not those of *MICROpendium* or anyone else. (Try to imagine the word "COMMENTARY" flashing in your mind while reading this.)

We believe that programmers serving the TI community should, first and foremost, serve the person with the minimal system. For Assembly programs, this means the person with 32K, XB or E/A or TIW module, and one SS/SD disk drive. Does that mean we preclude serving those with three or four drives? No, it just means that we think the guy with the minimal system should be able to use any of our products.

What we won't do is get caught up in what we might call the "Hardware Mania" that currently infects the TI Community. What do we mean by "Hardware Mania?" This: We don't own a Hard Drive for either of our TI machines. That means, among other things, that we can't effectively program for things involving hard drive. In our opinion, nobody who owns a TI should invest in putting Hard Drive on it. The cost is high, and the payoff very questionable, mainly because the TI system was not designed for it. We do have a hard drive on one of our three PC computers, and find it very useful in the environment of MS-DOS.

When we program things on the PC, we take advantage of service routines provided by the BIOS and DOS that are omnipresent on those machines. Thus our programs have no problem with finding out what the current drive and directory are, because we can simply invoke a DOS interrupt service to get that information. We used that service in the PC version of our Golf Score Analyzer so that all the files it creates automatically reside on the same drive and in the same directory as the program.

On the TI, there is no equivalent of the DOS services. We can and do find out the device name from which a program was loaded, but if that turns out to be WDS1, we're stumped about what directory or sub-directory we loaded from. Incidentally, if any of our readers does have a way of doing this, we'd be happy if he or she would share that information with others through User Notes or any other handy forum.

Should we decide that we must serve that subset of the TI community that has hard drives, we would have to invest a lot of money and valuable time learning how to program for it. In a good year, that might increase our sales by ten disks or so, which would mean that in only 25 or 30 years our hard drive would pay for itself.

We feel the same way about such wonderful gimmicks as 80 column cards. We don't have those, either, and can't see enough extra sales potential out there to ever make acquiring one a viable financial investment.

We do have Horizon Ramdisk on one of our two TIs, but that's mainly to speed up the process of Assembling source code. The Horizon is something we use every day, and thanks to some very excellent work by its developers, it does very faithfully emulate the normal floppy drive, except of course that it performs many times faster than the floppy drive. Our Word Processor, which takes something like 27 seconds to load into memory from floppy disk, loads from Ramdisk in about 3 1/2 seconds.

We also have Horizon's P-Gram cards in both of our TIs, because those got us out of the unreliable cartridge connection. This way, we have Editor/Assembler and Extended Basic just a keystroke away.

That, however is all we do with the Ramdisk and P-Gram. We don't have any "Rambo" or "Plus" memory on either of these devices. Again, that's a matter of economics for us. Writing programs that require Rambo or P-Gram+ would be foolish, since it would severely limit our market. Perhaps we could program cleverly enough so that our programs would work without those devices, but work better if they were present. Nevertheless, buying them just to reach that marginal extra market would never, in our opinion, pay off.

Okay, we're back off our soapbox. We hope people will understand why this column will not get heavily into how to program for RAMBO or 80-column cards, or any other exotic new gadgets that come along. We've got enough problems just making sound programs for the "Baseline" TI system, and we'll concentrate on helping our readers with those problems.

Our next article will give some ideas on how to make effective use of the 10,198 bytes of VDP RAM (>1000 thru >37D6) that's normally not used by the E/A module. We may touch other topics, but that will be the main one.

```
* EXAMPLE OF DATA STRUCTURING FOR A SCREENFUL OF STUFF
* FIRST, THE CODE TO INVOKE THE SUBROUTINE
*
FILSCR BL   @CLS         CLEAR THE SCREEN
       LI   R6,LOCTB1    SET R6 TO LOCATION TABLE
       LI   R1,TXTTB1    SET R1 TO TEXT ADDRESS
       BL   @DISSCR      CALL SUBROUTINE
       (GO ON TO NEXT OPERATION)
*
* THE SUBROUTINE DISSCR DISPLAYS A WHOLE SCREENFUL
*
DISSCR MOV  R11,*R15+    STASH RETURN ADDRESS ON STACK
DISCR1 MOV  *R6+,R0      GET SCREEN LOCATION INTO R0, INCREMENT R6 BY TWO
       CI   R0,>FFFF     IS THAT "END OF SCREEN" CODE?
       JEQ  DISCRX       IF SO, GET OUT OF SUBROUTINE
       BL   @DISLI       ELSE DISPLAY STRING POINTED TO BY R1
       JMP  DISCR1       THEN JUMP BACK FOR NEXT ITEM
DISCRX B    @SUBRET      USE HIGH-LEVEL SUBROUTINE RETURN
*
*  DISLI IS SHOWN FOR REFERENCE
*  IN THE ORIGINAL PROGRAM, A MORE COMPLEX VERSION
*  OF DISLI WAS USED, FOR COMPATIBILITY WITH EXTENDED BASIC
*
DISLI
       MOVB *R1+,R2      GET LENGTH BYTE INTO R2
       SRL  R2,8         RIGHT JUSTIFY IN R2
       JEQ  DISLIX       IF R2 = 0, GET OUT OF SUBROUTINE
       BLWP @VMBW        ELSE DISPLAY STRING CONTENT
       A    R2,R1        THEN ADD R2 TO ADDRESS IN R1
DISLIX RT                RETURN TO CALLING PROGRAM
*
* DATA SECTION
*
* LABEL LOCTB1 IS FOR ENTIRE FILL-IN SCREEN
* EACH ENTRY GIVES SCREEN LOCATION FOR ONE STRING
*
LOCTB1 DATA SCRWID+2,SCRWID+14,SCRWID+21
       DATA SCRWID*4+2,SCRWID*4+10,SCRWID*4+20
       DATA SCRWID*6+1,SCRWID*7+1
       DATA SCRWID*8+1,SCRWID*9+1
       DATA SCRWID*11+2,SCRWID*11+10,SCRWID*11+20
       DATA SCRWID*13+1,SCRWID*14+1,SCRWID*15+1
       DATA SCRWID*16+1,SCRWID*18+1,SCRWID*18+9
       DATA SCRWID*18+23,SCRWID*19+2,SCRWID*19+11
       DATA SCRWID*19+21,SCRWID*20+2,SCRWID*20+11
       DATA SCRWID*21+2,SCRWID*22+2
```

```
       DATA >FFFF        END OF SCREEN INDICATOR
*
* TXTTB1 IS COLLECTION OF STRINGS FOR FILL-IN SCREEN
*
TXTTB1 BYTE 7
       TEXT 'COURSE:'
       BYTE 4
       TEXT 'PAR:'
       BYTE 5
       TEXT 'DATE:'
OUTSTR BYTE 3
       TEXT 'OUT'
       BYTE 5
       TEXT 'PUTTS'
       BYTE 3
       TEXT 'PAR'
       BYTE 2
       TEXT 'HL'
       BYTE 2
       TEXT 'PA'
       BYTE 2
       TEXT 'SC'
       BYTE 2
       TEXT 'PU'
INSTR  BYTE 2
       TEXT 'IN'
PUTSTR BYTE 5
       TEXT 'PUTTS'
PARSTR BYTE 3
       TEXT 'PAR'
       BYTE 2
       TEXT 'HL'
       BYTE 2
       TEXT 'PA'
       BYTE 2
       TEXT 'SC'
       BYTE 2
       TEXT 'PU'
GRSSTR BYTE 3
       TEXT 'GRS'
HCSTR  BYTE 10
       TEXT 'HI      HC'
NETSTR BYTE 3
       TEXT 'NET'
       BYTE 5
       TEXT 'BIRDS'
       BYTE 6
       TEXT 'EAGLES'
       BYTE 5
       TEXT 'PUTTS'
```

```
BYTE 4
TEXT 'PARS'
BYTE 6
TEXT 'BOGIES'
BYTE 13
TEXT 'DOUBLE BOGIES'
BYTE 4
TEXT 'CMNT'
```

1.12. The Art Of Assembly — Part 12. Getting The Most From VDP RAM

By Bruce Harrison

As we mentioned at the close of #11 in this series, there is a great and largely untapped resource of memory in the 10,198 bytes of VDP RAM (>1000 thru >37D6) that are not normally used by the E/A module. We have devised a number of ways to use this resource that were never mentioned by TI. Today we'll share some of those ideas with our readers, in hopes they'll become more innovative in this regard.

Several years ago, when we started to write our own Word Processor on the TI, it very quickly became obvious that we would run out of memory if we tried keeping the working text in a main memory buffer space. That was of course what both the E/A editor and TI-Writer did, and doing so placed limits on what features the editor could contain. It also contributed to the need for a separate "formatter" program to do printing with TI-Writer.

Also early in the development of the WP we decided that a single document should be able to be many pages in length while still being part of one document. The answer to our dilemma lay in the VDP RAM memory. We designed the WP so that only one page would be kept in the computer at one time, and so that the text of that page would actually exist only in VDP RAM, as a series of screens. Thus the first screenful of a document page resides at >1000 in VDP RAM, the next screenful at >1400, then >1800, and so on. As the user scans through a page, we "flip" the screens by a simple VWTR operation, setting VDP Register 2 to different values so that the output to the monitor is different sections of the VDP RAM memory.

This same concept was used in our game program "SCUD BUSTERS" for getting and displaying the on-screen instructions for the user. As the file of instructions was being read from the disk, the lines of text were "displayed" into 40-character screens starting at >1000 and moving up by >400 for each new screenful. Once all the text was present in the higher VDP RAM locations, we switched VDP Register 2 so that the screen at >1000 would be visible, then switched upward by >400 for each successive screen.

Another potential use for VDP RAM that we haven't tapped yet is the concept of having multiple character sets loaded there, and thus being able to "instantly" switch in a whole new character set by writing to VDP Register 4. The Pattern Table address advances in >800 increments, so one could conceivably load one character set of 256 definitions at >800 (the default location under E/A), load another set at >1000, another at >1800, and so on. Then one could switch whole character sets by a VWTR operation. Let's just say we had an alternate character set at >1000, and wanted to instantly switch over to that set. The code to do so would look like this:

```
LI   R0,>0402
BLWP @VWTR
```

Then to switch back to the normal character set, one could:

```
LI   R0,>0401
BLWP @VWTR
```

Simple, eh? We mentioned that we haven't tried this, and we haven't. We have done such a switch on a one-time basis when entering an Assembly program through an Extended Basic loader, so that the dreaded "offset" would not be in effect on our characters going to the screen. The "LOAD" program for our Word Processor does just that. It uses a GPLLNK service to load the standard characters in the >800 area, then switches to text mode and changes VDP Register 4 to use the character set at >800 before putting the "LOADING MAIN PROGRAM" message on screen.

Before we go further with this treatment of VDP RAM, let's issue a "warning" to XB Programmers. XB has its own uses for VDP RAM space, and any program subroutine that must return to XB should not tamper with the space above >1000. In one of our first attempts to mix Assembly subroutines with an XB program, we ran afoul of this problem. We were using an Assembly subroutine to do a sound effect by the "Sound List" method, placing our sound list data at what we thought was a safe location in VDP RAM. Before long, strange effects were happening in the Extended Basic part of the program. Weird characters were appearing on the screen, and string variables were getting lost. After trying a number of different locations for our sound list, we concluded that, while XB was in command, there was no safe place to tamper with VDP RAM.

We finally resorted to changing the subroutine so that, just before putting the sound list (132 bytes) into VDP RAM, it would do a VMBR into our own low-memory space, thus stashing the contents of that area. After the sound effect ended, we did a VMBW to put back what was in that place in VDP RAM before our sound effect. This worked.

Sound lists are another potential use for the unused VDP RAM space. Normally, we don't use the "sound list" process at all, because the limitation of working with note durations in 60ths of a second is a killer for the classical music we've programmed. In the game SCUD BUSTERS, we made an exception to our normal rule so that sound effects like the Patriot's launching sound could proceed in the "background" while the CPU was busy checking for coincidence between the sprite representing the Patriot and that representing the Scud. We used sound lists for all the "noises" in that game, but the Theme at the beginning was done by our "normal" method of sending bytes directly to the sound generator. The Theme would have been impossible in sound list format, because of its rapid pace, quick-decaying drumbeats, and so on.

Of course VDP RAM is also commonly used for "PABs and Buffers" as mentioned in the E/A manual. Some care must be taken with the choice of locations for PAB and Buffer, particularly if Memory Image files are being loaded, since these will fill large blocks at the buffer location.

In a couple of our smaller utility programs, such as the ones we use to transfer text files from and to our PC computers, we actually used the lower part of the viewable screen as the buffer. This is somewhat unorthodox, but it had the singular advantage that one could actually see the text lines being transferred right on the screen. In that case the PAB was located at an "off screen" location, since much of the PAB content is "undefined" characters.

If push really came to shove, one could even put sections of data into VDP RAM instead of in the main memory. The only drawback to this use is that access to these data items would take considerably more time than getting them from main memory takes. One could even do a kind of "dynamic allocation" in this way, where different sections of data could be brought into main memory while they're needed, then written back to VDP for storage while another section of data is brought in for another part of program operation.

The same concept could be used for program "overlays", in that several small sections of code could be brought into VDP RAM as a memory image file, then transferred to main memory for execution when needed. Again this is a concept that we haven't actually tried, but it should be able to work.

Okay, dear readers, put your thinking caps on, and see if you can actually apply some of these ideas in your programs, or perhaps come up with more ways to use VDP RAM and share them with others.

Today's is a relatively short article, with no Sidebar full of source code, but we hope this "food for thought" will take hold. Next time we're going to delve into the mysteries of Random Number generation, and will provide some directly useful source code for that purpose.

1.13. The Art Of Assembly — Part 13. Randomly Speaking

By Bruce Harrison

There's nothing so rare as a good sequence of Random Numbers. Today we'll explore random number sequences on the computer and discuss some of the pitfalls waiting for you when you do random number sequences on your friendly little TI.

First, a word about terminology. In the pure sense, nothing we do in today's article will produce true random numbers. If you're going to throw dice, there's no substitute for real ones. What we can do on the computer is generate sequences of numbers that appear to be randomly selected, in the sense that knowing what the most recent number was gives no hint at what the next one will be, unless one knows both the algorithm used and the seed number in use. What we'll create, from a mathematical point of view, is a sequence of Pseudo-Random numbers. Each such sequence is really mathematically deterministic, given a particular starting number or "seed". The method we'll show can create 65,536 different sequences, and unless one were willing to do a lot of arithmetic by pencil and paper, one will have no way of predicting what number will arise next from the sequence. In this article, we'll use the term "Random Number" to mean a member of such a pseudo-random sequence.

The keys to making good random numbers are two: First, one must have an algorithm for determining what the next number in sequence will be; Second, one must have a high quality means of getting the seed, or starting number for the sequence. Let's put this second thing first, since this must be done first in any practical application.

As you probably know, there is a memory location in the TI called Random Number Seed (>83C0). Under the simplest conditions on the TI, this location in memory will be loaded with a word that's unpredictable. There are, however, exceptions to this rule. If, for example, one selects Extended Basic and that finds a LOAD program on DSK1, the seed will always be left with the same number, and thus RANDOMIZE will be ineffective. If one has a Horizon Ramdisk, and the computer boots up to the menu provided by that device, then the Random Number seed will always contain the number >02B0. This too will render the seed useless.

Fortunately for us, there are numbers in the TI's memory that change as a function of time, and can be captured conveniently to provide a seed number. One memory location is >8379, and it's only one byte, but under the right conditions it will provide a number in the range of >00 through >FF, and there's no way of predicting its contents before reading it. Even this has caveats attached.

For reasons unknown to us, the Editor/Assembler module, while it's in control, limits the range of this number to >00 through >14. The probable reason for this is that the counter is being used to time the blinking of the cursor, and gets cleared every time it reaches >14. That gives us only twenty-one possible states for the number. What do we do about that? We'll get to that in a moment.

The second counter that keeps changing is the Screen Timeout counter at >83D6. This number keeps incrementing by two on each VDP interrupt, until it blanks the screen, or until a key is struck on the keyboard. The fact that this counts by two is important to the discussion that follows.

Today's Sidebar source code is a fragment from our game program SCUD BUSTERS. This code puts a message on the screen (UNDO ALPHA LOCK, <ENTER>) and then grabs the state of the VDP Interrupt counter into R10. Since we may have entered from Editor/Assembler, and thus the counter might be limited in its range of numbers, we use only its lowest order bit at this point. That's accomplished by the ANDI R10,>0001 instruction.

Now before scanning the keyboard, we grab the screen timeout counter's state from >83D6 into R5. This must be done before KSCAN, because this number gets cleared when a key is struck. The way we've constructed this loop, we will have the value before the keystroke in R5 after the key has been struck. Now it becomes important to know that this number counts by twos, so the number in R5 after the keystroke is always even. (Its least significant bit is always a 0.) The LIMI 2 and LIMI 0 instructions in the key loop are there so that the timers will keep advancing while we await the keystroke input.

We then take this number from R5 and place it at the seed location, >83C0. Now we grab the byte at >8379 and place that in the high-order byte at >83C0. At this point >83C0 contains two bytes of unpredictable content, but it will always be an even number because the low byte was derived from a counter that's counting by twos. Now to be sure that we give our seed the full possible range, we take that one bit that was stashed in R10 and XOR the word at >83C0 with it. All but the lowest bit of R10 will always be zero at this point, but the state of that one bit will determine whether our seed number is odd or even. Moving R10 back to >83C0 completes the seeding operation. The number thus placed at >83C0 will have been selected based on the relationship between the states of the two counters and the time at which **ENTER** was actually pressed.

By doing this, we've established a number unknown to the user that may take on any value from 0 through 65,535. That's 65,536 possible numbers for the seed used in our random number process.

Once we have this number established, the sequence of numbers to be produced by our random generation algorithm is actually determined. Given that same starting number, the algorithm will produce exactly the same sequence of numbers every time. What makes it useful is of course that the user won't know which of those 65,536 sequences is being produced, and so the numbers will appear to be quite random.

The algorithm shown in today's Sidebar (subroutine RANDNO) was adapted from one used in TI's Tombstone City game. It's very simple but effective, and produces very good random sequences at blazing speed. On each pass, it takes a large number into R4, multiplies by whatever happens to be at the random number seed location (>83C0), adds a large number to the low-order word of the result (in R5), and places that new number in location >83C0. The number then in R5 is our "new" random number, 16 bits in length.

In most cases we won't actually want a number in the range >0000 through >FFFF, but will want a more manageable range of numbers, like perhaps 0 through 255, 1 through 100, or something like that. To get that number tailored to a range we want, we use a trick borrowed from a book on PC Assembly language. Take a number one more than the range desired. Clear register 4 so the word in R5 is treated as a double-length word in R4-R5, then divide R4 by the number that's one more than our desired range. When we do this division, the number in R4 will be the quotient, which we ignore, and the number in R5 will be the remainder. This will always be a number between 0 and one less than the number we divided by. Voila! That's the desired result.

To make this easier for our game, which uses random numbers for many purposes, we preload R3 with the desired range before calling the subroutine, then let the subroutine perform the divide operation, so that when we return from the subroutine R5 always contains an integer in the range we want.

In the Sidebar we've shown just one example of this, the little loop we use to randomly scatter 26 "stars" in the night sky. We preload R3 with the value 32*19, so the stars will be scattered over 19 rows of the screen, then we add 64 after return from the subroutine, so the stars will begin at Row 3 of the screen and extend through row 22.

Before the reader mail comes in, we'll answer one "Why Don'tcha" question, to wit: "Why don't we just take the word from >83D6, add that bit from R10 to make it odd or even, then move it to >83C0?" Answer: In many cases (in this program too) there will have been a recent keystroke before we entered the key loop, and therefore the high order byte of the Timeout Counter would be zero most of the times we took our seed value. Thus we use the byte from >8379, which advances much more rapidly (60 times per second) as the high order part of the seed.

Now before we get deluged with questions other than that one, let's state for the record that your author is not an expert on Random Numbers per se. The quality of randomness can be tested, I'm told, by algorithms designed for that purpose. There are people who work near my home at a place called the National Security Agency (NSA) who could probably run tests on my "random number" process and shoot it full of holes. I'm not even sure how often a sequence of full-word numbers generated my way would repeat itself. Is it once every 32,767 numbers, once every 65,535 numbers, or once every 10,000?

Before we did this game program, we ran tests galore to make sure we got the most "random" possible performance from our algorithm. We never found a sequence repeating in our tests. In fact, once one does that divide operation to tailor the number to a specific range, there appears to be no way of predicting what comes next in the sequence. That's so because there are many different raw "whole word" numbers that will produce the same output "ranged" number, and a different number will follow that each time, so far as we could tell.

What seemed important was that the apparent degree of randomness would be more than enough to keep the user guessing as to where and when the next SCUD would enter the screen, and in what direction and at what horizontal speed it would be moving. With 65,536 possible sequences of numbers, we felt this would be good enough for us, even if it weren't good enough for the folks at NSA. After all, the security of the nation's codes and ciphers does not depend on my random numbers. (That's probably fortunate for us.)

As a by-product of all that testing we did before producing SCUD BUSTERS, we made some subroutines for use with Extended Basic to provide seeding and to make very fast random numbers for XB programmers. One of those, for example, will fill an array of dimension 500 with tailored random integers in about one second. Making 500 tailored (ranged) random numbers with RND in Basic or XB takes a very long time indeed. (Use a calendar instead of a stopwatch.)

These utilities have been released to Public Domain, and made available through User Groups and other means of "free" distribution. Tigercub (156 Collingwood Ave, Whitehall, OH, 43213) has them available as part of their TI-PD offerings. If you'd like a collection of these assembly subroutines with demo programs, and can't find it through other sources, I'll send one (SS/SD disk) if you'll send a check or money order for $2.00 to Harrison Software, 5705 40th Place, Hyattsville MD 20781. Ask for the "Random XB Utilities" disk. Please try getting it from Tigercub, your User Group or BBS first. Also, please don't mail me a blank disk. Disks cost me less than you'll pay to mail me a blank one.

We're not sure just now what our topic for next month will be, so let's just leave that as a "surprise" in your next *MICROpendium*.

```
* SOURCE CODE FRAGMENT FROM SCUD BUSTERS
* ILLUSTRATES RANDOM NUMBER PROCESS
*
* FIRST SECTION PLACES A MESSAGE ON-SCREEN
* AND BEEPS TO ALERT USER
       LI   R0,32*22+3   POINT AT R0W 23, COLUMN 4
       LI   R1,UNALP     MESSAGE "UNDO ALPHA LOCK, <ENTER>"
       LI   R2,24        24 CHARACTERS
       BLWP @VMBW        WRITE MESSAGE TO SCREEN
       BLWP @GPLLNK      USE GPLLNK
       DATA >34          TO PRODUCE "BEEP" TONE
* NEXT SECTION PLACES AN UNPREDICTABLE NUMBER IN THE SEED LOCATION
* BASED ON THE STATES OF THE VDP INTERRUPT COUNTER AND SCREEN TIMEOUT COUNTER
       MOV  @>8378,R10   GET THE VDP INTERRUPT TIMER INTO R10
       ANDI R10,>0001    USE ONLY THE LOWEST ORDER BIT
SEED   MOV  @>83D6,R5    GET SCREEN TIMEOUT COUNTER IN R5
       CLR  @STATUS      CLEAR GPL STATUS
       BLWP @KSCAN       SCAN THE KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            STOP INTERRUPTS
       CB   @ANYKEY,@STATUS  HAS A KEY BEEN STRUCK?
       JNE  SEED         IF NOT, REPEAT LOOP
       CB   @KEYVAL,@ENTERV WAS THAT <ENTER> KEY
```

```
       JNE  SEED         IF NOT, REPEAT LOOP
       MOV  R5,@>83C0    PLACE LAST STATE OF TIMEOUT AT >83C0 (SEED)
       MOVB @>8379,@>83C0 PUT VDP INTERRUPT BYTE INTO HIGH BYTE OF SEED
       XOR  @>83C0,R10   XOR SO LOW BIT OF PRIOR INTERRUPT COUNT IS USED
       MOV  R10,@>83C0   MOVE RESULTING WORD FROM R10 TO SEED
* THIS SECTION USES SUBROUTINE RANDNO TO PLACE 26 STARS IN THE SKY
* AT RANDOMLY SELECTED POSITIONS
       LI   R9,26        COUNT OF STARS TO MAKE
       CLR  R1           CHARACTER 0 IS THE STAR
RNDSTR LI   R3,32*19     R3 DETERMINES RANGE OF NUMBER DESIRED
       BL   @RANDNO      FROM RANDNO
       MOV  R5,R0        MOVE RANDOM NUMBER FROM R5 TO R0
       AI   R0,64        ADD TWO ROWS "BOTTOM" OF RANGE
       BLWP @VSBW        WRITE ONE STAR TO SCREEN
       DEC  R9           DECREMENT COUNTER
       JNE  RNDSTR       IF NOT ZERO, GET ANOTHER NUMBER
* SUBROUTINE RANDNO MAKES RANDOM NUMBER
* ON ENTRY, R3 CONTAINS DESIRED RANGE
* ON EXIT, R5 CONTAINS A NUMBER 0 THRU NUMBER GIVEN IN R3
RANDNO
       LI   R4,28645     PLACE A LARGE NUMBER IN R4
       MPY  @>83C0,R4    MULTIPLY BY THE SEED NUMBER
       AI   R5,31417     ADD A LARGE NUMBER TO RESULT IN R5
       MOV  R5,@>83C0    MOVE THAT RESULT BACK TO SEED
       CLR  R4           CLEAR R4 SO NUMBER IS RIGHT JUSTIFIED IN R4-R5
       INC  R3           INCREMENT SO R3 IS ONE MORE THAN DESIRED RANGE
       DIV  R3,R4        DIVIDE BY DESIRED RANGE
       RT                RETURN TO CALLING PROGRAM
```

1.14. The Art Of Assembly — Part 14. Crossing The Bridge To Option 5

By Bruce Harrison

In all our Assembly work so far in this series, we've concentrated our attention on programs to execute as Option 3 (LOAD AND RUN) from the E/A menu. We have made some excursions into the mysteries of Loaders, but not much in the area of using TI's SAVE utility to make our programs into Option 5 directly.

Let's start by simply stating that, in the E/A manual, there are lots of things TI forgot to tell you about this process of converting from Option 3 to Option 5. In some ways, it's a little like crossing from Staten Island to Brooklyn by first building the Verrazano Narrows bridge.

Let's say you've written and debugged an Assembly program, and everything is working just the way you want it under Option 3. Now you want to convert it to Option 5, so you get out the big book and start at page 420. You put in the labels SFIRST, SLOAD, SLAST, make sure your first instruction at label SFIRST is an executable one, save your source code to disk and Assemble it. Now you load in your object file and the SAVE utility, and enter at label SAVE. Everything looks fine. You give the Option 5 program a file name "DSK1.ANYPROG", and SAVE makes one or more memory image files.

Now you run an immediate test of this by loading DSK1.ANYPROG under Option 5. So long as you haven't used GPLLNK, everything will work. (We'll get into the strange case of GPLLNK in a little bit.) You go on about your business, happy that what TI said was okay.

Later, you start up your computer anew, and try DSK1.ANYPROG from Option 5. Nothing works. The screen goes green, no displays appear, the keyboard does not respond except to **FCTN = (QUIT)**, and you're mystified. The problem is actually quite a simple one. Remember those "Utility" vectors like VMBW, VSBW, KSCAN, and so on that you used in your Option 3 program? They haven't been placed in low memory for you by Option 5. When you made that first trial just after the SAVE operation, those utilities were still in place from the Option 3 loading operation. Now that you've made a clean start, there's nothing there, so the first place in your program that calls for one of those will simply lock up the computer.

You have at this point fallen victim to something TI forgot to mention on pages 420 and 421 of the manual, namely that Option 5 does not load those utilities into low memory for you.

You have a few options at this point in your program's development. You could forget about Option 5 and simply keep your program running as Option 3 only. That's okay for your own use, but the program will take longer to load each time you want to use it, and you'll always feel that this program is unfinished.

You could rework your program so that it doesn't use those low-memory utilities, but that's really a big pain. You could write your own utilities to perform these functions, but that's an even bigger pain. You could use a tool like Art Green's Linker and Library system, which provides Art's own versions of those utilities embedded in your program file. (We have a copy of Art Green's Linker disk, purchased from him several years ago at Ottawa, but have never used it. Judging only by its documentation, it looks as if it would work nicely, but we can't say for certain.)

The method that we've used in all of our Option 5 conversions has been simply to stash the needed utilities in our own program space as a data block, then put them in their proper place in memory when the program starts. We take 806 bytes from low memory and place that at the very end of our program before branching to the SAVE utility.

The first part of today's Sidebar shows how we do this, using what one might call a "wrapper" around our main program. There's a short section of code at the very beginning of the program, to put the utilities in place, and a short section of code after SLAST that is used to get the utilities into our own memory space before branching to TI's SAVE utility. Once our program has been assembled this way, we load the object file from Option 3, then load the SAVE utility. Now instead of using the Program Name SAVE, we use the Program Name SAVIT, which is like a mini-program within our own program. This stashes the utilities, then branches to TI's utility to actually make the memory-image file(s).

Another important note: If your program itself loads into low memory, as with an AORG, TI's SAVE utility can't be used to make it an Option 5 Program File, because that utility itself loads into low memory, and will overwrite the program you've just loaded. For that instance, we use our own "SAVE" utility, tacked onto the back end of our main program. This method was used in our Smart Connect programs, to let all of High Memory remain available for stashing a file being brought in from a PC. Source code for that is in the second part of today's Sidebar.

We mentioned earlier the "strange case" of GPLLNK, and perhaps now would be a good time to explain that statement.

TI's built-in GPLLNK will work just fine so long as we're operating in the straight Option 3 environment, without an Auto-Start label in our program. Once we're out of that "envelope", the TI GPLLNK seems to run amok. Even saving all the utilities as we've shown in today's Sidebar will not allow our Option 5 program to use the TI GPLLNK. The symptom that shows up is easy to describe, but so far we've not found any way to explain it. Let's say for example that we've used BLWP @GPLLNK followed by DATA >34 to make a beep when our program is ready for a user input. In normal Option 3 mode, where we've typed in the entry point as a PROGRAM NAME, all is well. The beep happens as expected, and as long as we've remembered to include LIMI 2 and LIMI 0 in our key acceptance loop, the beep stops itself after the normal duration.

If, however, we used the Auto-Start option in our Option 3 program, the beep results in taking us immediately out of our program. It places us at the PROGRAM NAME input location (without the prompt), and our program is essentially dead. We can type in the Program Name at the prompt and re-start, but that defeats the purpose of having the Auto-Start option in the first place. The same symptom will show up in the case of an Option 5 program that uses TI's GPLLNK. Why, you ask, does this happen? Sorry, but we don't know. We've tried numerous ways to get this effect not to happen, but always to no avail. We've resorted to using a separate GPLLNK routine (see #7 in this series) to provide the GPLLNK services in our programs.

If any of our readers can shed some light into this dark corridor of TI operation, we'd be happy to get some feedback on the matter. A letter to the FEEDBACK column would be most appreciated, or even a letter to us at 5705 40th Place, Hyattsville, MD 20781, would be cheerfully accepted. In either case, we'd like to know the why of this little problem, and any solution short of providing a separate GPLLNK.

In the course of our many experiments to come up with a viable method of making the utilities available under Option 5, we've made a "mapping" of the locations of all these E/A utilities, which is probably worth sharing with our readers. Here it is:

VECTOR	*ADDR*	*WORKSPACE/CODE*
GPLLNK	>2100	>2094/>21C4
XMLLNK	>2104	>2094/>2196
KSCAN	>2108	>2094/>21DE
VSBW	>210C	>2094/>21F4
VMBW	>2110	>2094/>2200
VSBR	>2114	>2094/>220E
VMBR	>2118	>2094/>221A
VWTR	>211C	>2094/>22B2
DSRLNK	>2120	>209A/>22B2
LOADER	>2124	>20DA/>23BA

Most of these utilities use the workspace at >2094, and, except for the object code LOADER, their code sections end at >23BA.

A quick glance at our Sidebar will reveal that we save all of the area from >2094 through >23BA into our program space. We don't save the loader's code, which extends some distance beyond >23BA, since we don't generally need an object code loader in our programs. We do save the contents of the workspace at >2094, since this seems necessary for correct operation of TI's DSRLNK. Thus 806 bytes are saved and replaced by our SAVIT and PUTUT loops.

Note here, as in some examples we've shown previously, that we use a temporary workspace while saving and loading the utilities, since those envelop our usual workspace at >20BA. Within the programs themselves we "recycle" parts of the memory used to stash the utilities as general purpose memory. For example, if we need a couple of 80 character spaces, we will use EAUT for one of them and EAUT+80 for another. We sometimes use that space for our subroutine return address stack.

The second part of today's Sidebar is a bit more tricky to use. In this case, you have to put the name of the planned Option 5 file directly in the source code, and remember to change the length byte at the end of SPABDT. In the example we've put in the name "DSK1.ANYPROG", which has 12 characters in it, and made the length byte >0C at the end of SPABDT.

There are reminders in the source code that bear highlighting here as well. When you wrap this "sandwich" with the source code shown, you must be sure to remove the END directive from the end of your main program, else the assembler will never assemble the second slice of bread. Also, in the case of the low memory programs, there will be an AORG at the start of the main program. This must be removed so that the AORG in the first part of our "sandwich" will define the memory location for the program. We've used >2678 in that AORG directive, since that's usually the first available low memory location when we start up under E/A.

In this second part of source code, we've used absolute numbers for the utility vectors VMBW and DSRLNK, and the absolute address for the GPL STATUS byte and GPL Workspace. This was done simply to avoid conflicting with the REF statements you may have included in the main program's code. We'd create Assembly errors if we also REF'd the same utilities, so we took advantage of our "mapping" to put the absolute numbers in here.

Before we leave this subject, we'll call your attention to the DATA at the line labeled HEADER in the source code. When TI's SAVE utility is used to make Option 5 files, it takes care of placing the header on the file for you. Here, where we are in effect making our own "SAVE" utility, we compose the header in three data statements. The first data word is zero to indicate that this file is not "chained" to others. Thus the Option 5 loading process will not bother looking for another file named "DSK1.ANYPROH" to load after "DSK1.ANYPROG".

The second word of data in the header is the length of the actual memory image content of the program. We've made the Assembler calculate that number for you by placing the expression SLAST-SFIRST in the data line. (A similar trick was used in the PAB data, with +6 added to account for the header.) The last of the three words is the origin of the program in memory, and this we've set to >2678. If you've used some other origin, you might want to change that to read SFIRST instead of an absolute number.

If your low-memory program is tight for memory space, you could place an AORG >A000 directive just before label GETUT, so that the second slice of bread will appear in high memory instead of using valuable low-memory space.

Please remember that the method shown in today's article is only one way of getting across from Option 3 to Option 5. There are probably a hundred or so other ways to accomplish this mission, but we've given you a way that we know works reliably, because we've used these methods in our own programs. We'd be very pleased if readers who know "better" ways would share them through letters or items for the "USER NOTES" column.

In our next article, we'll try to wrestle with some problems of making programs compatible with Geneve. That effort will be hampered somewhat by the fact that we don't own one of those, but our tips will be rather generic anyway.

```
* TWO WAYS TO CONVERT FROM OPTION 3 TO OPTION 5
* FIRST IS FOR NORMAL HIGH MEMORY LOADED PROGRAMS
*
* EXAMPLE OF "WRAPPER" FOR OPTION 5 PROGRAM FILE
* TAKEN FROM OUR CRYPTOGRAM GAME PROGRAM
       DEF  SFIRST,SLOAD,SLAST  DEFINITIONS NEEDED BY SAVE UTILITY
SFIRST
SLOAD
       LWPI WST          LOAD TEMPORARY WORKSPACE
       LI   R9,EAUT      POINT R9 AT STORED UTILITIES
       LI   R10,>2094    POINT R10 AT >2094 IN LOW MEMORY
       LI   R4,>23BA->2094  LOAD R4 WITH NUMBER OF BYTES TO MOVE
PUTUT  MOV  *R9+,*R10+   MOVE ONE WORD, INCREMENT POINTERS BY TWO
       DECT R4           DECREMENT COUNT BY TWO
       JNE  PUTUT        IF NOT ZERO, REPEAT OPERATION
*
* MAIN PROGRAM'S CODE STARTS HERE
*
* THE MAIN PROGRAM AND ITS DATA SECTION ARE
* "SANDWICHED" HERE BETWEEN THE PART THAT REPLACES
* THE UTILITIES AND THE PART THAT SAVES THEM INTO
* MEMORY THAT WILL BECOME PART OF THE MEMORY IMAGE FILE
*
* MAIN PROGRAM'S DATA SECTION ENDS HERE
*
       EVEN              INSURE THAT EAUT IS AT AN EVEN MEMORY LOCATION
EAUT   BSS >23BA->2094   LENGTH OF BSS IS 806 BYTES
       REF  SAVE         REFERENCE TI'S SAVE UTILITY
       DEF  SAVIT        DEFINE OUR SAVIT ENTRY POINT
SLAST
* SLAST MARKS THE END OF WHAT THE SAVE UTILITY WILL PUT IN MEM-IM FILE
SAVIT
       MOV  R11,@>8300   STASH R11
       LWPI WST          LOAD OUR TEMPORARY WORKSAPCE
       LI   R9,>2094     POINT R9 AT BEGINNING OF AREA TO BE SAVED
       LI   R10,EAUT     POINT R10 AT MEMORY LOCATION ABOVE
       LI   R4,>23BA->2094  LOAD R4 WITH NUMBER OF BYTES TO MOVE (806)
GETLP  MOV  *R9+,*R10+   MOVE ONE WORD AND INCREMENT POINTERS BY TWO
       DECT R4           DECREMENT COUNT BY TWO
       JNE  GETLP        IF NOT ZERO, REPEAT AT LABEL GETLP
       B    @SAVE        BRANCH DIRECTLY TO TI'S SAVE UTILITY
WST    BSS  32           OUR TEMPORARY WORKSPACE
       END
* END OF FIRST "SANDWICH" SOURCE CODE
*
* SECOND "SANDWICH"
```

```
* BUILT-IN SAVER FOR LOW-MEMORY PROGRAMS
* ADD THESE PROGRAM LINES TO CONVERT PROGRAM
* FROM OPTION 3 TO OPTION 5
* AFTER ASSEMBLY, LOAD AS OPTION 3, THEN
* ENTER WITH PROGRAM NAME GETUT TO SAVE PROGRAM
* AS OPTION 5 PROGRAM FILE
*
* THIS PART GOES BEFORE MAIN BODY OF PROGRAM
       DEF  GETUT
       AORG >2678        ABSOLUTE ORIGIN AT >2678
SFIRST
SLOAD
       LWPI TWS          LOAD TEMPORARY WORKSPACE
       LI   R9,SAVUT     POINT AT MEMORY BLOCK WITH UTILITIES
       LI   R10,>2094    POINT AT DESTINATION FOR UTILITIES
       LI   R4,>23BA->2094  LOAD R4 WITH LENGTH OF BLOCK
PUTUT  MOV  *R9+,*R10+   MOVE ONE WORD AND INCREMENT POINTERS BY TWO
       DECT R4           DECREMENT COUNT BY TWO
       JNE  PUTUT        IF NOT ZERO, MOVE ANOTHER WORD
*
* MAIN PROGRAM GOES HERE - BE SURE TO REMOVE AORG AND END FROM MAIN PROGRAM
* NEXT PART GOES AFTER END OF MAIN PROGRAM
*
       EVEN
SAVUT  BSS  >23BA->2094  SPACE FOR SAVING UTILITIES
SLAST
GETUT  MOV  R11,@>8300   STASH REGISTER 11
       LWPI TWS          LOAD TEMPORARY WORKSPACE
       LI   R9,>2094     POINT AT START OF UTILITIES
       LI   R10,SAVUT    POINT AT MEMORY SPACE
       LI   R4,>23BA->2094  LOAD R4 WITH LENGTH OF BLOCK
GETLP  MOV  *R9+,*R10+   MOVE ONE WORD AND INCREMENT POINTERS BY TWO
       DECT R4           DECREMENT COUNT BY TWO
       JNE  GETLP        IF NOT ZERO, REPEAT
       LI   R0,>1000     POINT R0 AT PERIPHERAL ACCESS BLOCK IN VDP
       LI   R1,SPABDT    POINT TO DATA FOR PAB
       LI   R2,25        25 BYTES TO WRITE
       BLWP @>2110       WRITE PAB DATA INTO VDP (>2110 IS VMBW VECTOR)
       AI   R0,9         ADD NINE TO R0
       MOV  R0,@>8356    MOVE PAB+9 ADDRESS TO >8356
       LI   R0,>1020     POINT AT VDP BUFFER LOCATION
       LI   R2,6         SIX BYTES TO WRITE
       LI   R1,HEADER    FOR FILE HEADER
       BLWP @>2110       WRITE HEADER TO BUFFER
       A    R2,R0        ADD HEADER LENGTH TO VDP ADDRESS
       LI   R2,SLAST-SFIRST   LENGTH OF ENTIRE PROGRAM IN R2
       LI   R1,SFIRST    R1 POINTS TO START OF PROGRAM
       BLWP @>2110       WRITE MEMORY IMAGE TO VDP BUFFER
       CLR  @>837C       CLEAR GPL STATUS BYTE
       BLWP @>2120       CALL DSR LINKAGE VECTOR AT >2120
```

```
       DATA 8            DATA MUST BE 8
       LWPI >83E0        LOAD GPL WORKSPACE
       CLR  @>837C       CLEAR GPL STATUS BYTE
       B    @>006A       RETURN TO GPL INTERPRETER
TWS    BSS  32           TEMPORARY WORKSPACE
HEADER DATA 0,SLAST-SFIRST,>2678
* CHANGE FILE NAME IN SPABDT AS NEEDED
* BE SURE LENGTH BYTE AT SPABDT+9 MATCHES LENGTH OF NAME
SPABDT DATA >0600,>1020,0,SLAST+6-SFIRST,>000C
       TEXT 'DSK1.ANYPROG   '
       END
* END OF SECOND "SANDWICH"
```

1.15. The Art Of Assembly — Part 15. Compatibility With The Geneve

By Bruce Harrison

We have said this before, but let's just be sure our readers know at the outset of this article that we do not own a Geneve. We will, however, relate some of the compatibility problems we've encountered, and offer solutions to some, but not all.

In last month's article we discussed getting from an Option 3 program to an Option 5 environment, and one of the suggestions made was to capture the E/A utilities from low memory, embed them into a program's space, and then have the Option 5 program put them back in low memory when the program starts.

That process seems to help on the Geneve, especially if your program uses DSRLNK. In an earlier article, we showed the source code for a general purpose DSRLNK and GPLLNK to be used when programs had to load from Extended Basic. For reasons we've never pinned down, the DSRLNK given there will not always work on a Geneve. (The GPLLNK will.) If you have done the process described last month, however, your program will be using the E/A DSRLNK, and that seems to work just as well on a Geneve as on a TI.

In some of our programs, we have made the program able to load from either XB or E/A, but advised Geneve owners that only the E/A entry method will work on their machines. That seems to be borne out by recent tests.

The other big problem that we encountered was that timing loops set up to run on the TI speed up considerably on the Geneve, even at its slowest clock rate. A customer named Aaron West, who owns both a Geneve and a TI, helped us find a solution to that problem, so that some of our Assembly music disks could be made so that the music would play at the same pace on either machine, and regardless of the clock speed setting on the Geneve.

From that experience, which took many mailings of disks between Maryland and Connecticut, we were able to devise a fairly efficient way of calibrating timing loops for the faster Geneve. Today's Sidebar shows our Calibrate routine, which measures the speed of the computer it's running on and then allows us to modify the timing loop counts in the program. This process will also work for "bus" modified TI consoles, in which the 32K memory works on a 16-bit basis, and so executes much faster than the normal 32K expansion.

Our friend Dan Eicher ran some benchmark tests using this calibration code, and found that the numbers tracked accurately on all three systems (normal TI, BUS modified, and Geneve). We have not tested this code on a TI with the new Accelerator installed.

We suspect that some of what we've done may not work all that well for the Accelerator, because the speed difference may be more than we can handle. The problem occurs not in the calibration run itself, but in the multiply and divide operation that's necessary for adjusting loop counts. If the ratio between the TI speed and that of the running machine is too great, then the results of the divide operation will not fit in a 16-bit word, resulting in an unrecoverable error condition.

We ran into this particular problem when trying to make our Assembly music work on Tandy computers. Most of the Tandy PCs would handle it just fine, but we did discover that some of their newer ones were so much faster than our 1000SX that a "divide overflow" error would happen on long-duration notes, stopping the PC dead. The divide overflow won't stop the Geneve or TI, but timing loops can still go crazy.

The source code shown in the Sidebar uses the CRU clock, which runs at the same rate on either TI or Geneve, and so gives us a "constant" by which to measure the execution speed of the machine we are running on. The code as shown provides a complete E/A Option 3 program that will run a test for you and display results on the screen. When integrating this into a program of your own, you would omit all the code beginning at label DISPLY, the two lines immediately following label CALIB, and all the data except labels CALNUM and TINUM.

In essence what happens here is that we load up a count into the CRU, turn on the CRU clock, and then run a time-wasting loop 50 times before we stop the CRU clock. Our loop has taken some amount of time, which will differ by the speed of the computer, while the CRU clock has provided us an invariant time measuring count. When we stop the CRU clock and recover the count left in the CRU, we have a measure of the speed at which that 50-times loop repetition occurred. On a standard TI, the count we report out to CALNUM will be 199, just like the constant value we have used at TINUM. On a Geneve, that number reported to CALNUM will always be less than what we got on the TI.

As we mentioned, Dan Eicher was able to get access to a TI, a Bus-modified TI, and a Geneve for testing this routine. The resulting CALNUM values he got were:

STANDARD TI - 199 (of course)
"BUS" TI - 139
GENEVE - 48

Dan also ran a slightly modified version of this routine. He changed the Workspace to >8300, so that the registers used in the delay loop would be accessed as 16-bit words. That made no difference on the Geneve or the Bus Modified TI, but changed the number to 165 on a standard TI. This tells us that the Geneve, like the Bus Modified TI, has a 16-bit path to its 32K memory as well as to CPU RAM PAD. We learn something every day in this business!

Now a little honest confession from your Assembly columnist: I don't really know for certain how this works. Aaron West knows, I think, but his explanation to me wasn't exactly clear. What I do know is that this works, and I have annotated the source code in the Sidebar to indicate how I think it works, but please don't accept the annotations as Gospel.

Once the number CALNUM is established, we can use it to modify the delay-loop counts in our program, so the actual time delays these loops provide will be nearly constant regardless of the machine they run on. Now let's suppose that you have a delay loop count value built into your program somewhere like this:

```
DLYCNT      LI   R4,>0200
```

To fix that up for another machine, you have to do something like this:

```
CALDLY      MOV  @DLYCNT+2,R7
            MPY  @TINUM,R7
            DIV  @CALNUM,R7
            MOV  R7,@DLYCNT+2
```

Let's examine this one line at a time. The first line gets the immediate value used for the delay loop into Register 7. Next, we multiply by the number at TINUM (199). The result will be some number in the R7-R8 register pair. Now we divide that number by CALNUM. If we are on a standard TI, we just multiplied and divided by the same number, so R7 will contain the original immediate value. If we are on a Geneve, and CALNUM is therefore 48, we will divide R7-R8 by that smaller number, so R7 will contain >084A after the division. Thus when we move this number from R7 back into the immediate value location, the delay loop, when it executes, will execute more than four times as many passes though the loop. Thus the timing of the actual loop will be compensated for the speed of the machine, so the time delay imposed will be about the same as on the original TI for which the program was developed. Of course this correction of the timing count will have to be done early in the execution of your program, before the timing loop itself has to execute. Our practice has been to do the calibration part at the very beginning of the program, then do the Multiply and Divide operation for each timing constant immediately after CALIB, thus doing these things before any of the delay loops executes.

As we understand it, the Geneve can be run at different speeds by changing the setting of its CPU clock. People who have tried out some of our music that was made Geneve Compatible have reported that this made no difference whatever in the tempo of the music. It should not, therefore, make any difference in the performance of the calibration business we just covered, except that the numbers we've shown in our example would be different, but with the same self-regulating result.

Somehow you all knew there would be some words of caution coming here, and you were right! The delay counts that you start with have to be set up so that they won't "overflow" when the multiply and divide operation are performed. In the previous example, we started with number >0200 for our delay loop on the TI. That's a safe number, in that even if it got multiplied by ten in the calibration process, it would still be only >1400, well short of the limit (>FFFF) before causing an overflow condition. Without having a Geneve to run exhaustive tests on, we can only guess at what the actual limits are. In our previous example, where the Geneve was running slightly over four times the TI speed, we could have started with >3DBF in the delay loop and not encountered an overflow. What we recommend, without being overly cautious, is that the original delay loop counts be kept to >2000 or less, so that no risk of overflow is present.

Suppose >2000 does not give you enough delay? You could always make a nested loop for the delay, such as this:

```
DLYCNT      LI   R5,4
SECOND      LI   R4,>2000
DLYLOP      SRC  R15,15
            SRC  R15,1
            DEC  R4
            JNE  DLYLOP
            DEC  R5
            JNE  SECOND
```

Please note that in this case you would modify the immediate value at SECOND+2 through that multiply and divide operation, and that the outer loop count (4, in this case) would not be modified for this nested situation.

The advent of the Accelerator puts a whole new wrinkle into all of our calculations. It's reported to multiply the inherent CPU speed on the TI by ten. In that delay loop situation we just covered, the starting value for the count would have to be below >1999 (>FFFF divided by ten) in order to avoid overflow. We suspect that using the Accelerator may make many pieces of existing software unusable. Anything involving timing loops will execute much too fast to be manageable. We don't plan on modifying our TIs that way.

There is of course another whole approach to timing loops, in which one uses the VDP Interrupt Timer to perform delay timing. This is a convenient way around the whole problem, assuming that counting by 60ths of a second gives accurate enough results. This could become the topic for a whole article, but let's just give a quick example of how this can be made to work. Let's assume you want a two second delay for the user to see something on the screen. The delay loop could be constructed like this:

```
            CLR  @>8378
DLYLOP      LIMI 2
            LIMI 0
            MOV  @>8378,R4
            CI   R4,120
            JLT  DLYLOP
```

The instructions LIMI 2 and LIMI 0 are very important in this particular case, because without those, this becomes an infinite delay. The loop will continue to execute until the VDP Interrupt counter at >8378 becomes equal to or greater than 120, which makes the overall delay 120 60ths of a second, or in simpler numbers, two seconds. This same technique can be applied for other delay amounts just by changing the number in the CI statement. Three seconds would require CI R4,180, and so on. The number is limited to 255, and that would give a delay of 4.25 seconds. If longer delays are needed, the loop could be nested as shown above.

We hope this column has shed some light into the dark corners. We realize that there are some wonderful things that can be done with a Geneve that could not be done on a TI, but our focus is of necessity on the things that won't work the same on the Geneve. It appears that, over time, Geneve owners are finally getting software written just for their machines, and that's a hopeful sign. If Geneve owners have enough Geneve-exclusive software, they won't have to worry about trying to run stuff written for the TI on their machines.

Next month's topic is still undecided at this writing. We are writing these many months ahead of publication, and watching Reader Feedback every month for questions or comments from our readers, which we'll try to handle in future columns.

```
* CALIBRATION - MEASURES EXECUTION SPEED OF THE MACHINE THIS RUNS ON
* CODE BY BRUCE HARRISON
* PRINTS NUMBERS ON-SCREEN FOR EXAMINATION
* TOP NUMBER IS FOR CURRENT MACHINE, BOTTOM NUMBER IS FOR STANDARD TI-99/4A
* RELEASED TO PUBLIC DOMAIN
*
GPLWS  EQU  >83E0        GPL WORKSPACE
STATUS EQU  >837C        GPL STATUS BYTE
WS     EQU  >20BA        USER WORKSPACE
KEYADR EQU  >8374        KEY-UNIT ADDRESS
KEYVAL EQU  >8375        STRUCK KEY VALUE ADDRESS

       REF  VSBW,KSCAN   REFERENCED UTILITIES
       DEF  CALIB        DEFINE OUR ENTRY POINT
CALIB
       MOV  R11,@>8300   STASH RETURN ADDRESS
       LWPI WS           LOAD USER WORKSPACE
       CLR  R12          SET CRU BASE 0
       SETO R3           SET R3 TO ONES
       LDCR R3,15        PUT 15 BITS INTO CRU
       SBZ  0            ACTIVATE CRU CLOCK
       LI   R4,50        DELAY COUNT
DLY
       CLR  R5           ALL THIS JUST
       LI   R6,>FFFF     KILLS SOME TIME
       LI   R9,256       UNTIL WE READ
       DIV  R9,R5        THE CLOCK
       DEC  R4           DECREMENT LOOP COUNT
       JNE  DLY          IF NOT ZERO, REPEAT LOOP
       CLR  R12          SET CRU BASE 0
       SBO  0            STOP CRU CLOCK
       STCR R3,15        GET 15 BITS INTO R3
       CLR  R12          ZERO CRU BASE
       SBZ  0            RE-ACTIVATE CLOCK
       ORI  R3,>8000     MAKE NEGATIVE VALUE3
       SRA  R3,1         CUT VALUE IN HALF
       INV  R3           INVERT ALL BITS IN R3
       MOV  R3,@CALNUM   STASH AT CALNUM
```

```
       MOV  @CALNUM,R5   PUT IN R5
       LI   R0,11*32+16  SET SCREEN LOCATION
       BL   @INTDIS      DISPLAY INTEGER
       AI   R0,64        MOVE DOWN TWO SCREEN LINES
       MOV  @TINUM,R5    PUT TINUM IN R5
       BL   @INTDIS      DISPLAY THAT
KEY    CLR  @STATUS      THIS SECTION
       BLWP @KSCAN       SIMPLY WAITS FOR
       CB   @ANYKEY,@STATUS A KEY TO BE PRESSED
       JNE  KEY          ELSE REPEAT SCAN
       LWPI GPLWS        GET BACK TO GPL WORKSPACE
       MOV  @>8300,R11   PUT R11 VALUE BACK
       CLR  @STATUS      CLEAR STATUS
       RT                RETURN
INTDIS
       LI   R14,INTSTK   POINT R14 AT STACK
INTLOP MOV  R5,R6        PLACE R5 NUMBER IN R6
       DEC  R0           DECREMENT R0
       CLR  R5           CLEAR R5
       DIV  @TEN,R5      DIVIDE BY TEN
       SWPB R6           GET REMAINDER IN LEFT BYTE R6
       AB   @NUMBER,R6   ADD NUMBER MASK
       MOVB R6,*R14+     STASH ON STACK (LEAST SIGNIFICANT DIGIT FIRST)
       MOV  R5,R5        IS R5 0 YET
       JNE  INTLOP       IF NOT, GO BACK
DISLOP DEC  R14          POINT TO MOST SIGNIFICANT DIGIT FIRST
       MOVB *R14,R1      MOV IT TO R1
       BLWP @VSBW        WRITE DIGIT TO SCREEN
       INC  R0           MOVE ONE CHARACTER ON SCREEN
       CI   R14,INTSTK   ARE WE AT BEGINNING
       JGT  DISLOP       NO, GO BACK FOR NEXT DIGIT
       RT                FINISHED, RETURN
INTSTK BSS  5            FIVE DIGIT MAX INTEGER (65535)
TEN    DATA 10           THE NUMBER 10 AS A WORD
NUMBER BYTE >30          HEX FOR ZERO CHARACTER
ANYKEY BYTE >20          SPACE CHARACTER
CALNUM DATA 0            DATA FOR NUMBER FOUND
TINUM  DATA 199          NUMBER THIS YIELDS ON A TI
       END
```

1.16. The Art Of Assembly — Part 16. Maximizing Speed Of Execution

By Bruce Harrison

Today's topic is speed, but it is also choices. Once we are doing our programming in Assembly, we have a big speed advantage going for us compared to the interpreted Basic or Extended Basic languages. None the less, when operations are performed over many repetitions, as in a loop, the choices we make as to how to perform our operations can make a big difference in the speed of completing them.

We have touched on the topic of integer math operations before, and mentioned how using integers rather than floating point numbers can make things go faster. Here is a concrete example from our Golf Analyzer program. When that was reviewed by Bill Gaskill, he remarked on the speed at which the program performed the calculation of Handicaps. That calculation involves taking as many as twenty rounds of golf, adding up the scores, multiplying and dividing many times, and so on, yet when the program performed that operation there was a barely perceptible delay between starting and ending the process.

One thing that made such speed possible was the range of numbers involved. Course ratings, for example, which must be used in the calculation, have two parts (as we treated them). There is an integer part and a single decimal place. (e.g. 72.4) The integer part will not go over 100, and the decimal will always be just one digit. Thus we were able to streamline all our math operations that involve the rating by first multiplying the integer part by ten, then adding the decimal to that. In the example above, 72.4 would become 724. The gross scores, from which ratings are subtracted, were also multiplied by ten before doing the subtraction. Thus the whole of the calculations could be performed using the integer math instructions, and only at the very end of the calculations was the result divided by ten and then rounded to produce a "handicap". The code that performs this handicap calculation is shown in today's Sidebar.

The source code will be difficult to follow, even with copious annotation, but we think it will serve as an example of how to take maximum advantage of the range of a number and thus apply the much faster integer math operations rather than the floating point operations.

It is not important that the reader understand all the operations in the Handicap program section, but let's briefly describe what happens. The program has a file of golf rounds loaded in memory. It works backward from the round that has just been entered by the user, and will look at the round just entered plus the 19 or fewer rounds which precede it. Rounds are stored in date order, so the program will use the most recent 20 rounds.

For each such round, the program uses a subroutine to add up the gross score for that round. It uses other subroutines to find the course data for the round, including the Course Rating and Course Slope. (Slope is a number which ranges from about 100 through 130 or so, with 113 being the "normal" course slope.) The rating is actually stored in the course record as two bytes, one which contains the integer part, and one which contains the single decimal place. Early in the planning stage for GSA we saw that the rating multiplied by ten would always be far less than 32767, and so could always be treated as a word value in integer math operations.

The program then subtracts the rating times ten from the gross score times ten for each of up to 20 rounds, then performs a multiply and divide with the number 113 and the course slope. The result is a "differential" for the handicap calculation. These differentials are stashed in a table in memory for examination later in the process. If twenty prior rounds are available, the program will use the ten lowest differentials from that table to perform the handicap calculation. If there are more than four rounds available but less than 20, a lookup table will be used to decide how many differentials are used.

The ten lowest differentials are all added together (these are still ten times the actual numbers), multiplied by .96 (really multiplied by 96 and divided by 100). This result is then divided by the number of differentials used, and divided by ten to produce a handicap index number. The same result is then multiplied by slope, divided by 113, divided by ten and rounded to produce the handicap for the course just played.

This is probably all as clear as mud, but it may help those who are trying to follow what the source code does.

We don't really know how much impact there would be by performing all these operations in floating point math, because we have not tried doing this. What we intended to point out in all this is that programmers should take advantage of the situation they are presented with, and use integer operations where possible.

Another potential time-waster is in dealing with strings. In the same program, we had to find the course data for each round played, as the course data was stashed in a different part of memory. The course name was included in the record for each round played, but the pars for each hole on that course, and its rating and slope were stored elsewhere. This meant that for each round encountered, we had to look up the course information. That operation involved a special kind of string comparison to find the corresponding course record.

We were comparing the course name stored with the round with each course name among the courses in memory. That could involve considerable time, but we made a special kind of string comparison routine to minimize time spent. For the course names to be a match, they had to be equal in all respects, including length. Therefore since the length was there as the first byte in each string, we compared that first. If the lengths were unequal, then there was no point in comparing the content, so we got out of the comparison loop immediately. If length were equal, we would compare the content part of the string only until we found an unequal character, and then get out. Thus only the matching string would require comparison of all the bytes in the string to reach a decision.

In other words, our comparison would end as soon as it was possible to end it. This principle should be applied in all cases when strings are being compared.

In a more generalized case of string comparison, one can't make any decision on the length alone, because the content of the strings will determine which is smaller or larger. For that case, we have shown in the Sidebar a general purpose string comparison routine that will determine which string is larger, or whether they are equal. If they are identical up to the length of the shorter one, the longer string will be called larger. This routine is drawn from our "Easy Data" source code, and was actually used in the sorting of strings from XB DATA statements.

Of course what is shown here is really a fragment, since it does not show what actually happens when strings are found to be bigger, less, or equal. In the actual application, this code resulted in storing the address of the least of the two strings, or the address of the first one if they were equal. We purposely left out that part from the Sidebar so it would not confuse you.

In the real application, this string comparison was used as part of a very involved sorting routine. That routine was called from Extended Basic, and could sort 55 records of six fields each by two criteria in about three seconds. That time includes the finding of the strings in XB DATA statements and the assignment of those 330 strings into XB Array variables. If such a sort were done by Extended Basic, the time would be measured in minutes, not seconds.

Before we get too caught up in our own hype, though, we should say that what we are showing today are things that we have devised for specific circumstances. They have worked, and quite well, but that doesn't mean these samples are the "best" way to do things. Also, we are not showing today any "wrong" way to do things. The only thing we're attempting to say and show is a reasonably efficient and fast way to do some selected operations, and we are leaving the details of implementation to the user. The best way is still the one that you can use and understand, not necessarily our way.

There's another interesting tidbit in today's Sidebar called the ROUND subroutine. There were places in the calculations where the formula called for a rounded result, so we devised this small subroutine to do the job for us. That is why we sometimes placed the divisor in R3 before doing a divide operation, instead of simply dividing by some word in memory.

The idea of rounding is simple enough in concept. If the remainder is equal to or more than half the divisor, then the quotient is incremented by one.

The first step in the rounding is to double the remainder in R6. Now we simply compare R6 to R3. If R6 is less than R3, then we do not increment the quotient. Otherwise we do increment the quotient, so it is now properly rounded. None of the cases where this was used needed the remainder for anything after this operation, so we left R6 alone as we exited. Had we needed R6 restored to its original value after rounding, we could put in one line at label ROUNDX to SRL R6,1. This would put R6 back to the value it had upon entry to this subroutine.

This rounding process is not always exact, but is good enough for the cases in which we have applied it.

Suppose, for example, we divided 12 by 5. That would give us a quotient of 2 and a remainder of 2. Doubling the remainder would make that 4, which is less than 5, so the quotient would not be incremented. Had we divided 13 by 5, we would get a quotient of 2 and remainder of 3. Doubling 3 would make that more than 5 so we would (correctly) increment the quotient by one. We will leave it for the reader to play around with other numbers and see how accurate this process is. We chose this really simple method mainly for its speed, not for its ultimate accuracy.

The thoughtful reader will now look at our source code and see that in many instances we appear to have violated our own general rules. For example, we seem to be using labeled places in memory to stash values away, rather than keeping those values in registers, as we advocate.

Appearances can be deceiving. There are many subroutines called by the HCAP routine, and those alter the values in many of the registers. We chose to make room in our data segment of the program to stash things so we wouldn't have to worry about the registers that the subroutines use.

In the PC version of this same program, we handled things a bit differently. In the PC's Assembly language, one does not have the luxury of sixteen registers, nor the ability to do a "context switch" to another set of them. Instead, the PC has a readily useable "stack" segment in memory, into which one can "push" values and from which one can "pop" values whenever necessary. There are only four general purpose registers on the PC, called AX, BX, CX, and DX. There are also a number of special purpose registers which can't be used except for purposes like keeping track of memory segments and as pointers. A subroutine in PC assembly normally begins with a series of push operations to save the entry values of the registers, then at its exit ends with corresponding pops to restore the registers to their previous values. PUSH causes a value to be placed on the stack and the stack pointer register to be decremented by two. POP does the opposite. It looks something like this:

```
SAMPLE_SUB:
      PUSH AX
      PUSH BX
      PUSH CX
      PUSH DX
      (perform operations)
      POP DX
      POP CX
      POP BX
      POP AX
      RET
```

This saves and then restores the four "general purpose" registers available on the PC. The special purpose registers on the PC, such as DS, ES, SI, and DI may also be pushed and popped in a similar fashion. Of course the disadvantage of this method is that there is a lot of program memory used up as "overhead" for the subroutines, not to mention that all this pushing and popping eats into execution speed as well. Be grateful for the more efficient way that the TI lets us handle registers. While on this topic, we should also point out that not all operations on a PC can be performed with any of the four "General Purpose" registers. Indexed addressing, for example, can be performed using BX, but not AX, CX, or DX.

We hope this little digression into the PC realm will not upset you too much. It's there simply to show you what a good thing you have going in the TI's Assembly language.

Next month's topic is undecided at this point. We'd like to thank those readers who've told us they enjoy this series. We enjoy writing them, too.

```
* SOME ILLUSTRATIVE CODE SECTIONS
* FROM OUR OWN PROGRAMS
* ALL CODE SHOWN IS PUBLIC DOMAIN
*

* HCAP SHOWS AN INTERESTING USE OF INTEGERS TO HANDLE NUMBERS
* THAT HAVE ONE DECIMAL PLACE ATTACHED
HCAP
       MOV  R11,*R15+    STASH RETURN ADDRESS
       CLR  @COUNT       CLEAR A DATA WORD
       CLR  @GTPIN       AND ANOTHER
       LI   R1,SCRLI     POINT R1 AT A BUFFER SPACE IN MEMORY
HCP0   BL   @GETRAT      GETRAT GETS THE COURSE RATING AND SLOPE INFORMATION
       MOV  @RATINT,R5   PLACE THE RATING'S INTEGER PART IN R5
       MOV  @TEN,R3      GET THE VALUE TEN IN R3
       MPY  R3,R5        MULTIPLY THE RATING'S INTEGER PART BY TEN
       A    @RATDEC,R6   ADD THE DECIMAL PART OF THE RATING
       C    @ACES,@NINE  WAS THIS A NINE-HOLE ROUND?
       JNE  HCAPA        IF NOT, JUMP AHEAD
* THE FOLLOWING LINES ARE USED TO COMBINE THE RESULTS OF TWO NINE HOLE ROUNDS
* INTO ONE 18 HOLE ROUND
       C    @GTPIN,@ONE
       JEQ  HCAPB
       MOV  @ONE,@GTPIN
       MOV  R6,@GTNET
       MOV  @RNDTOT,@GTSCR+2
       MOV  @SLOPE,@GTPAR
       JMP  HCAP1
HCAPB
       A    @GTSCR+2,@RNDTOT
       A    @GTNET,R6
       SRL  R6,1
       MOV  @SLOPE,R5
       A    @GTPAR,R5
       SRL  R5,1
```

```
        MOV  R5,@SLOPE
HCAPA   MOV  R6,@GTSCR    STASH R6 IN MEMORY
        CLR  @GTPIN       CLEAR A WORD
        MOV  @RNDTOT,R5   MOV THE TOTAL GROSS SCORE INTO R5
        MPY  R3,R5        MULTIPLY THAT BY TEN
        S    @GTSCR,R6    SUBTRACT THE RATING (X10) FROM THE SCORE (X10)
        JGT  HCAP0        IF POSITIVE RESULT, JUMP
        JMP  HCAP1        ELSE SKIP NEXT PART
HCAP0   MOV  R6,R5        PLACE R6 BACK INTO R5
        MPY  @ONE13,R5    MULTIPLY BY 113
        MOV  @SLOPE,R3    MOVE THE COURSE SLOPE INTO R3
        DIV  R3,R5        DIVIDE R5-R6 PAIR BY SLOPE
        BL   @ROUND       ROUND THE RESULTING NUMBER
        MOV  R5,*R1+      STASH THE RESULT IN A STACK POINTED TO BY R1
        INC  @COUNT       INCREMENT THE COUNT OF ROUNDS USED
HCAP1   MOV  @CURREC,R9   GET CURRENT ROUND RECORD POINTER IN R9
        AI   R9,-56       SUBTRACT LENGTH OF ONE RECORD
        CI   R9,FILORG    SEE IF THAT'S BEFORE OUR FIRST RECORD
        JLT  HCAP3        IF SO, WE HAVE RUN OUT OF PRIOR ROUNDS
        MOV  R9,@CURREC   ELSE PLACE R9 AS CURRENT RECORD POINTER
        C    @COUNT,@TWENTY  HAVE WE DONE TWENTY ROUNDS?
        JEQ  HCAP3        IF SO, JUMP TO NEXT PART
        BL   @GETCN       ELSE GET COURSE NAME FOR NEXT PRIOR ROUND
        BL   @FNDCRS      THEN FIND THE COURSE RECORDS
        MOV  @CURREC,R9   AND SET R9 BACK TO ROUND RECORD START
        LI   R10,TEMREC   POINT AT TEMPORARY RECORD STORAGE
        LI   R4,56        56 BYTES TO GET
        BL   @MOVBTS      MOVE THE CURRENT ROUND RECORD TO TEMREC
        BL   @RNDCMP      THEN COMPUTE THE ROUND'S SCORE
        JMP  HCP0         AND JUMP BACK TO INCLUDE THIS ROUND
HCAP3
        MOV  @COUNT,R3    GET THE NUMBER OF ROUNDS FOUND
        MOV  R3,@GTOT     GTOT HAS NUMBER OF ROUNDS AVAILABLE
        CI   R3,5         COMPARE TO FIVE
        JLT  HCZX         IF LESS, NO HANDICAP ISSUED
        CI   R3,19        COMPARE TO 19
        JGT  HCAP4        IF GREATER, JUMP
        AI   R3,-5        ELSE SUBTRACT FIVE FROM THE NUMBER
        MOVB @RULUT(R3),R1 AND USE LOOKUP TABLE
        SRL  R1,8         RIGHT JUSTIFY NUMBER FROM LUT
        JMP  HCAP5        THEN JUMP AHEAD
HCAP4   MOV  @TEN,R1      IF TWENTY ROUNDS FOUND, WE'LL USE TEN OF THEM
HCAP5   CLR  @GTSCR       CLEAR A MEMORY WORD
        MOV  R1,@COUNT    COUNT HAS NUMBER OF ROUNDS TO BE USED
        CLR  R8           CLEAR REGISTER 8
        CLR  R5           AND 5
HCAP6   CLR  R13          CLEAR REGISTER 13
        CLR  R2           AND 2
        LI   R7,>7FFF     PLACE HIGHEST POSITIVE NUMBER IN R7
        MOV  @GTOT,R4     GET NUMBER OF ROUNDS FOUND IN R4
```

```
HCAP8  MOV  @SCRLI(R13),R2 GET A DIFFERENTIAL INTO R2
       JEQ  HCAP7          IF ZERO, SKIP THIS ONE
       C    R2,R7          ELSE COMPARE TO R7
       JGT  HCAP7          IF GREATER, SKIP
       MOV  R2,R7          ELSE R2 IS THE LOWEST DIFFERENTIAL
       MOV  R13,R9         SAVE POINTER IN R9
HCAP7  INCT R13            MOVE AHEAD IN STASHED DIFFERENTIALS BY A WORD
       DEC  R4             DECREMENT COUNT
       JGT  HCAP8          IF GREATER THAN ZERO, GO BACK
       A    R7,@GTSCR      ELSE ADD R7 TO TOTAL OF DIFFERENTIALS
       CLR  @SCRLI(R9)     AND CLEAR THAT MEMBER OF THE DIFFERENTIAL SET
       INC  R5             INCREMENT R5 COUNT
       C    R5,@COUNT      COMPARE TO TOTAL
       JLT  HCAP6          IF LESS, JUMP BACK
HCAPX  MOV  @GTSCR,R5      MOVE TOTAL OF DIFFERENTIALS TO R5
* THE OPERATION IN THE NEXT FOUR INSTRUCTIONS ESSENTIALLY MULTIPLIES
* THE TOTAL OF DIFFERENTIALS BY .96
       LI   R3,96          PLACE 96 IN R3
       MPY  R3,R5          AND MULTIPLY R5 BY 96
       LI   R3,100         NOW LOAD 100 INTO R3
       DIV  R3,R5          AND DIVIDE R5-R6 PAIR BY 100
       MOV  R5,R6          MOVE THAT QUOTIENT TO R6
       CLR  R5             AND CLEAR R5
       DIV  @COUNT,R5      DIVIDE BY THE NUMBER OF DIFFERENTIALS
       MOV  R5,R10         AND STASH QUOTIENT IN R10
       CLR  R9             CLEAR R9
       DIV  @TEN,R9        DIVIDE R9-R10 PAIR BY TEN
       MOV  R9,@HCINT      R9 IS THE INTEGER PART OF HANDICAP INDEX
       MOV  R10,@HCDEC     R10 IS THE DECIMAL PART
       MOV  R5,@GTSCR      NOW STASH R5 IN MEMORY
       MOV  @SAVREC,@CURREC GET NEWLY ADDED RECORD'S ADDRESS BACK
       BL   @GETCN         GET ITS COURSE NAME
       BL   @FNDCRS        FIND THE COURSE DATA
       BL   @GETRAT        GET THE RATING AND SLOPE INFORMATION
       MOV  @GTSCR,R5      BRING BACK R5
       MPY  @SLOPE,R5      MULTIPLY BY SLOPE OF COURSE
       DIV  @ONE13,R5      THEN DIVIDE BY 113
       MOV  @TEN,R3        BRING TEN BACK INTO R3
       MOV  R5,R6          MOVE R5 VALUE TO R6
       CLR  R5             CLEAR R5
       DIV  R3,R5          DIVIDE R5-R6 BY TEN
       BL   @ROUND         ROUND THE RESULT
HCAPX1 B    @SUBRET        THEN EXIT THE SUBROUTINE
HCZX   CLR  R5             CLEAR R5
       JMP  HCAPX1         THEN SHORTCUT TO EXIT
*
* ROUNDING SUBROUTINE FOR INTEGER DIVIDE OPERATIONS
* INCREMENTS QUOTIENT IF REMAINDER IS >= HALF OF DIVISOR
* ON ENTRY, QUOTIENT IS IN R5, REMAINDER IN R6, DIVISOR IN R3
*
```

```
ROUND  SLA  R6,1          DOUBLE REMAINDER
       C    R6,R3         COMPARE R6 TO DIVISOR
       JLT  ROUNDX        IF LESS, SKIP
       INC  R5            ELSE REMAINDER IS => .5, INC QUOTIENT
ROUNDX
       RT                 RETURN TO CALLING SEGMENT
*
*
* SPECIAL PURPOSE STRING COMPARISON SUBROUTINE
* DETERMINES ONLY WHETHER STRINGS ARE IDENTICAL OR NOT
*
* ON ENTRY, R9 AND R10 POINT AT THE LENGTH BYTES
* OF THE TWO STRINGS TO BE COMPARED
*
STRCMP
       MOVB *R9,R3        GET LENGTH INTO R3
       SRL  R3,8          RIGHT JUSTIFY
       INC  R3            INCREMENT TO INCLUDE LENGTH BYTE
STRCP0 CB   *R9+,*R10+    COMPARE ONE BYTE
       JNE  NOCMP         IF NOT EQUAL, GET OUT OF HERE
       DEC  R3            ELSE DECREMENT COUNT
       JNE  STRCP0        IF NOT ZERO, COMPARE NEXT BYTE PAIR
NOCMP  RT                 RETURNS FROM SUBROUTINE R3>0 MEANS STRINGS UNEQUAL

* GENERAL PURPOSE STRING COMPARISON SUBROUTINE
* ON ENTRY, R9 AND R10 POINT TO THE LENGTH BYTES OF
* TWO STRINGS TO BE COMPARED
*
CMPSTR
       MOVB *R9+,R4       GET LENGTH FIRST STRING IN R4
       MOVB *R10+,R5      LENGTH OF SECOND IN R5
       SRL  R4,8          RIGHT JUSTIFY
* THE FOLLOWING LINE WAS NOT NEEDED IN THE ORIGINAL APPLICATION
       JEQ  LESS          A NULL STRING WILL BE LESS
       SRL  R5,8          RIGHT JUSTIFY
* THE FOLLOWING LINE WAS NOT NEEDED IN THE ORIGINAL APPLICATION
       JEQ  BIG           IF THIS IS A NULL, THEN R9'S STRING IS BIGGER
CMP910 CB   *R9+,*R10+    COMPARE THE BYTES POINTED BY R9 AND R10
       JGT  BIG           IF R9'S BYTE IS BIGGER, JUMP
       JLT  LESS          IF R9'S BYTE IS LESS, JUMP
       DEC  R4            DECREMENT COUNT OF R9' STRING LENGTH
       JNE  DEC5          IF NOT ZERO, DECREMENT R5
       CI   R5,1          ELSE SEE IF R5=1
       JEQ  EQUAL         IF SO, STRINGS ARE EQUAL
* IF R5=1 AT THIS POINT, THE STRINGS BEING COMPARED ARE IDENTICAL
* IF R5 IS >1 AT THIS POINT, IT MEANS THE STRING POINTED BY R10
* IS IDENTICAL UP TO THE LENGTH OF THE STRING POINTED BY R9,
* BUT THE R10 STRING HAS MORE CHARACTERS, SO THE R9 STRING IS
* SMALLER BY DEFINITION
       JMP  LESS          ELSE STRING POINTED BY R9 IS LESS
```

```
DEC5   DEC  R5           DECREMENT OTHER COUNT
       JNE  CMP910       IF NOT ZERO, COMPARE NEXT BYTE
* IF WE REACH HERE, IT MEANS THE STRING POINTED BY R10 HAS RUN
* OUT OF CHARACTERS, AND ALL ITS CHARACTERS WERE EQUAL TO THOSE IN R9'S STRING
* THEREFORE R9'S STRING IS BIGGER BY DEFINITION
BIG    (PERFORM SOME OPERATION)
       JMP  RETRN        THEN JUMP TO EXIT
LESS   (PERFORM SOME OPERATION)
       JMP  RETRN        THEN JUMP TO EXIT
EQUAL  (SOME OTHER OPERATION)
RETRN  RT                RETURN TO CALLING PROGRAM
```

1.17. The Art Of Assembly — Part 17. Structure Can Be Good — But

In the very first of these articles we touched on the matter of structured programming, and haven't touched it since. Some time back we received a letter from one of our readers, who was annoyed that we had not taught our readers how to properly interface between "modules". There are a couple of good reasons (in our opinion) for not teaching that to TI programmers. On only about two occasions in the time that we've been writing Assembly for the TI have we found it necessary to link separately loaded object code "modules", and in both those cases the interface between them consisted of one DEF in the first one and one REF in the second. Big deal.

We have used the concept of modules in a non-trivial way on the PC computer. In one such instance there were some 20-odd separately assembled object files, with complex interactions required between them, and with very carefully designed interfacing to pass whole data segments as parameters from one module to another. The result, after the LINK process, was a single EXE file of some 117K bytes. The whole thing worked very well, even though it was written partly by one author and partly by another, with your columnist tying the whole together.

If we were in the unfortunate position of writing programs that required four or five programmers, with multiple overlays and such, then the idea of "modules" would make perfect sense, and the overhead involved in parameter passing and such would become a necessary evil. On the TI, however, there are very few programs that need that kind of approach, and that's good, because we don't have the memory to throw away on parameter passing schemes.

In our opinion, the very concept of separate object modules to be linked by a Linker or linking loader is an example of "mainframe thinking". On the PC it sometimes becomes absolutely necessary because the Assembler will run out of Symbol Table space long before it runs out of space for the object code. The PC Assembler will tell you it's run out of memory, then report that 18K remains. That means it has used up the part of a 64K segment reserved for the symbol table, even though it still has 18K left for object code. The TI is far better than the PC in this respect. We have written some very complicated stuff on this machine, with what seemed like far too many labels involved, but never ran the TI out of symbol table space during Assembly. Conversely, our old Golf Score Analyzer, which assembled very nicely in one object file on the TI, had to be split into two object modules on the PC to perform exactly the same job. This quickly became a real pain just keeping track of what labels had to be declared "PUBLIC" and which "EXTRN" to make the modules link correctly. (PUBLIC and EXTRN on the PC are equivalent to DEF and REF on the TI, respectively.)

Even our Word Processor, which occupies most of both the Low and High portions of the 32K memory, assembles as just one object file. On floppy disks, it takes thirty minutes or so to assemble, and that's a pain, but it's still less painful than separating it into modules would be. (Assembling on Ramdisk takes only ten minutes.)

Okay, so you say you must make your program into separate object modules. Our advice, then, is to keep the parameter passing as simple as you can possibly manage. You may have noticed in our previous examples that we most often pass parameters to our own subroutines by loading registers with data and addresses before the BL instruction. That will work in separate modules also, provided only that there is a REF and DEF relationship between the calling part in one module and the subroutine in another. For example, module 1 could have need of a subroutine in module 2, called PRLINE. So long as there was a REF PRLINE in the first one and a DEF PRLINE in the second, registers could be set before the BL to carry over the necessary parameters.

There are two other approaches to doing this, both of which still require the REF and DEF, but which will perhaps give more flexibility in some circumstances. Let's say that the subroutine needs three parameters to operate. We could pass the parameters this way:

IN MODULE 1

```
REF  PRLINE             External referenced label
BL   @PRLINE            Call the subroutine
DATA SCRROW             desired row on screen
DATA SCRCOL             desired column on screen
DATA STRADR             address of string to print
(program continues)
```

IN MODULE 2

```
DEF  PRLINE             define entry point
MOV  *R11+,R4           get first parameter in R4
MOV  *R11+,R5           get second into R5
MOV  *R11+,R6           get third parameter in R6
(subroutine continues)
RT                      Return to calling program
```

This will work, and the + included in the MOV *R11 instructions will correct the return address to return at the right point in the calling program, but we could just as easily have loaded R4, R5, and R6 before the BL, and that would work just as well. Each LI instruction takes four bytes. Each DATA takes two bytes, and each MOV *R11,RX takes two bytes, so the memory use is exactly the same, and performance should be identical.

Another approach, in case you want to leave your main program's registers alone, is to do the subroutine itself as a BLWP operation. This is a tad more complex, and requires another workspace of 32 bytes, but it can be done like this:

IN MODULE 1

```
            REF  PRLINE
            BLWP @PRLINE            Branch and load workspace pointer
            DATA SCRROW             screen row
            DATA SCRCOL             screen column
            DATA STRADR             string address
            (program continues)
```

IN MODULE 2

```
            DEF  PRLINE
PRLINE      DATA SUBWS              workspace address
            DATA PRCODE             code address
PRCODE      MOV  *R14+,R4           get first parameter
            MOV  *R14+,R5           get second
            MOV  *R14+,R6           get third
            (subroutine continues)
            RTWP                    Return to calling program and WS
SUBWS       BSS 32                  Subroutine's workspace
```

This too will work, but notice that considerably more memory is used. There are four extra bytes for the "Vector" location, and then there are the 32 bytes for the SUBWS. Of course that could be a single workspace shared by all subroutines in the module, and might therefore be "affordable" in the memory budget. Its main selling point is that the registers being used in the main program are not modified by the subroutine. Of course you must then remember not to change registers 13, 14, and 15 within the subroutine, as all those are needed for the RTWP to execute correctly. In this approach, you can also acquire data from the calling program's workspace registers, since R13 of the subroutine's workspace points to R0 of the calling workspace. You could then get the contents of the callers R2 into the subroutine's R9 by:

```
            MOV  @4(R13),R9
```

As you can see, this is already getting complicated. Are you really sure this is necessary? There's an old expression for our usual guiding principle call KISS, or Keep It Simple, Stupid. We find it much easier to follow that principle by making programs assemble into just one object file in the first place.

Of course this BLWP idea can be used in single object module programs as well as in those using multiple object files. We've shown this concept merely to illustrate how making the interface "clean" can introduce complications of its own for the programmer. This is not to say that one can't adopt such a way of doing all the subroutines. One certainly can, but must also keep in mind that the extra overhead involved in parameter passing may make the difference between a program that will fit in the TI's memory and one that will not. That, in essence, is why in our own programs we keep to the more risky but simpler way of interfacing to our subroutines.

We did use the BLWP method in one instance recently, where we were making a special subroutine to be called via a user-defined interrupt from Extended Basic. In that particular case, we did some operations in the GPL Workspace to see whether **FCTN 7** was being pressed at the keyboard. If not, we simply returned to XB with an RT instruction. If we found **FCTN 7** was being pressed, we did a BLWP into a routine to dump the screen's contents to the printer. Using the BLWP in this case made sure we did not upset anything critical to the GPL interpreter in its own workspace. There were no parameters that needed to be passed in that instance, so the BLWP and RTWP instructions were all we needed to get into and out of our screen dump routine.

Some time back, we mentioned in this column that we use Art Green's RAG Assembler, and mentioned the advantages of that one over the TI Assembler. At the time when we purchased that software package from Art at an Ottawa Faire, he insisted on throwing in his Linker at no extra cost. It's still here in a drawer, never used. We have read over the documentation, but never found a situation where it would be useful to us. In those rare instances where our TI programs had to be loaded as separate modules, the linking capability of the E/A Object Loader itself was good enough. Certainly Art had something in mind when he wrote that Linker, and he provided a library of pre-assembled modules to go with it, but we've never really seen the need for either in our work.

Calling someone else's subroutines from a library through a linker requires a lot of study and consideration. We have always found it easier to use our own subroutines, and to simply copy them into the source code where needed rather than make them part of a "library".

Several years ago, your columnist was working for the Government, and using a Wang PC at the office. There was a project in which we found it necessary to do lots of work with floating point numbers. Unfortunately there was no available library of routines available for that machine except in the Basic Compiler's link library. Months of effort went into figuring out how to access and pass our own parameters into those routines from our own Assembly modules. When it was finished, we had a "demo" program that was a real knockout, in that it could make calculations and display numbers on the screen about 20 times the speed of a compiled basic program using the same floating point routines. When it was shown to the head man, he asked "Why do this?" and the project ended. We could have done this whole thing in one afternoon by simply writing the program in Basic to start with, instead of insisting on making it an Assembly program. The point of this is that sometimes it just isn't worth the effort to try for the most perfect way of doing things. A way that works is enough in most cases.

At the risk of repeating ourselves, we'll say once more that there are as many ways of doing something in Assembly as there are people trying to do it. Each programmer finds a method that he can deal with, and then sticks to that method so long as it works. Our purpose is not to convert our readers to only our way of doing things, but that's how our source code examples are done because they are our own stuff, not somebody else's. If making your program into ten object modules to be linked by the linking loader or Art Green's Linker is what works for you, then by all means keep doing that! We promise not to scold you for it in this column. Well, maybe we will, but we won't mention your name while we scold you in this column.

Last month we showed some methods to speed up your programs. Next month we will skip back a little and discuss some ways to slow down operations when you need to, so that the humans who operate your programs can keep up with the machine.

1.18. The Art Of Assembly — Part 18. Whoa! Slow Down! Hold On A Sec!

The month before last, this column had some words about various delay loops, with the concentration on how to make those self-correcting when run on a Geneve or other faster machine.

Today we will concentrate on delays that do not need correcting, since they use the built-in VDP Timing, and so will work exactly the same on TI, Geneve, and "bus" modified TI. There are some things about the guts of the machine that we'll touch on first, after a short story.

Some years ago, your author was attending a "Computer Graphics" show, and saw a very nice computer driven display for showing the weather. It looked much like the kind of displays that are used today by every TV station, but this was before the era of the TI Home Computer. The guy demonstrating it was one of its design engineers. It really looked good until he told it to put a new display on the screen. While the new display was being formed, there was an effect on the screen like that caused by ignition noise on a TV set. Your author looked at this and said to the engineer, "I see you're stealing some of the display time to write to your refresh memory." The man's jaw dropped, and he replied, "I've been at this show two whole days and you're the first person who knew what caused that noise on the screen."

In the TI, we don't get that kind of effect because the TI synchronizes itself to the 60 Hz field rate of its video output, and writes to its refresh memory during the Vertical Interval, when the video output is blanked. The fact of doing things this way in the TI gives us some fringe benefits. It provides us a handy little "Timer" called the VDP Interrupt Timer, which counts time in 60th of a second intervals. (On models sold in Europe, we're told, it counts in 50th of a second intervals, to be compatible with the PAL video systems and 50 Hz power in use there.)

In today's Sidebar, we start with a very simple delay loop that uses the VDP Interrupt timer to create a delay. This might be used to allow a couple of seconds to view a title screen, or any such simple delay to stop the computer for a known period of time. The parameter you need for this delay is simply a number in 60ths of a second. As shown, one could set the delay to two seconds by placing 120 in register 4 before entry to this loop. The counter affects only the single byte at >8379, so this loop will not work for numbers more than 255, or about 4 1/4 seconds. If there's a familiar ring to that number, it's because 4 1/4 seconds is the maximum duration for a CALL SOUND in Basic or Extended Basic. The timing of sounds uses the same counter, which is just one byte, and thus limits to the same 255 60ths duration.

Please note that in all cases where we have shown LIMI 2 followed by LIMI 0 in today's source code, these are essential to the operation of the code. Without those, the delay loops become infinite.

The second version in the Sidebar is for those cases where you want the user to be able to abort the delay by pressing a key. This simply includes a KSCAN in the delay loop, so that any keypress will make the loop terminate before the counter times out. You can introduce slight variations on this theme so that, for example, only the **ENTER** key would terminate the delay.

Another use that can be made of the VDP interrupt timer is to blink the cursor during an input cycle. Some time back we showed a method for doing that without using the VDP Interrupt Timer, but in that case the flashing rate of the cursor would be faster on Geneves. The version shown here will have the same flash rate on either TI or Geneve. The values you put in the data for ON and OFF will determine how long the cursor stays on and off the screen. A value of 20 will give about 1/3 of a second, for example. It will of course be slightly different on European models, because of their 50 Hz vertical sync, but that should keep it close enough to the "ballpark" that it won't be annoying. Our preference in such situations is to have the cursor on the screen less than half the time, and that's why we've shown different values for "OFF" and "ON".

Way back when we were first learning Extended Basic, we learned a technique from Millers Graphics to use CALL SOUND statements as timers in our programs. That can also be done in Assembly, by dumping a sound list into VDP RAM and starting a loop that waits for the sound processing to finish. The sound itself can be "silent" if you like. Again this times in 60ths of a second, but by making the sound list itself longer, one can exceed the 4 1/4 second limitation. Of course one can also do that by double-looping the first example we showed, running multiple passes through a single delay loop.

Now let's switch gears for a moment, and catch up with some things our readers have told us about. In number 10 of this series, we showed, among other things, a routine to put a one word integer on the screen in decimal notation. Our thanks to Mr. Merle Vogt for passing along an undocumented feature that can be accessed through GPLLNK. This will take a one-word integer and convert it to a string in decimal notation for you. The sign bit is ignored, so the resulting string will represent a number as 0 through 65535. A sample source code section for using this GPLLNK feature is shown in the Sidebar. Both Harry Wilhelm and Mr. Vogt have pointed out the screen scrolling feature using GPLLNK. Moving the current screen contents up by one row and producing a blank bottom row can be done this simply:

```
BLWP @GPLLNK
DATA >4D00
```

This can only be used in the normal 32-character screen mode, but works equally well from E/A or Extended Basic environments, provided GPLLNK is available. You may have noticed that we have not used screen scrolling in any of our examples, but that's just our personal choice. We prefer a "Display At" kind of operation rather than a scrolling.

While we are on other subjects, let's skip backward a bit again. In an early number of this series, we showed a method we used for displaying a string to the screen with the "offset" for XB added to each character. Our friend Harry Wilhelm has passed along a more compact version, that works as a BLWP subroutine. (Harry is the author of the astounding "Missing Link" that Texaments sells.) His uses the workspace at >2038, which is used by VMBW and VSBW in XB. It also uses a subroutine that's part of the XB utilities area to set the VDP Write address and to get the parameters from the User's workspace. We have shown this with two entry points, one for the normal "string" setup, at entry point PRSTR, and another where the number of characters to display has been preset in the callers R2. This second entry point (VMBW60) works exactly like a normal VMBW, except that it adds the offset for you. Please bear in mind that this only works if you entered through Extended Basic. Our sincere thanks go to Harry for these little gems, and for giving us permission to pass them along to you. We'll certainly use them, and hope some of our readers will find them useful too.

We suspect that there may be hundreds of undocumented features available in our machines, like the one Merle Vogt kindly offered. If any of our readers know of more such "goodies", we'll promise to pass them along in this column if our readers will send them to us at 5705 40th Place, Hyattsville, MD 20781. We will check them out first to be sure they work as advertised. It takes time, because we work so far ahead of publication dates, but we will work those things in.

Last week (April 1992) we had a letter from one of our more avid readers asking about information on using the Joysticks and Sprite applications in Assembly programs. We sent that reader some sample source code from Scud Busters, but the thought occurred to us that we had not covered either subject in this column. Next month's column will delve into those two subjects, and provide the rest of our readers with similar source code, abridged somewhat so as not to fill up a whole issue with Harrison's Sidebar. After all, if there weren't room for the ads, we'd all have to do without our favorite mag! See you next month.

```
* SOURCE CODE EXAMPLES
* BY B. HARRISON EXCEPT WHERE NOTED
*
* FIRST, A DELAY SUBROUTINE THAT'S CONSTANT ON TI, GENEVE, OR "BUS" TI
* THE CALLING PROGRAM WOULD LOOK LIKE:

       LI   R4,120       FOR A TWO-SECOND DELAY (MAX VALUE 255)
       BL   @DELAY       CALL THE DELAY SUBROUTINE
       (PROGRAM CONTINUES)
*
* SUBROUTINE LOOKS LIKE THIS:
*
DELAY  CLR  @>8378       CLEAR THE VDP INTERRUPT COUNTER
DLY1   LIMI 2            PERMIT INTERRUPTS
       LIMI 0            SHUT THEM OFF AGAIN
       C    R4,@>8378    COMPARE VALUE IN R4 TO COUNTER
       JGT  DLY1         IF R4 IS GREATER, REPEAT LOOP
       RT                ELSE RETURN, DELAY IS FINISHED
*
* SECOND VERSION OF SUBROUTINE NEEDS:
```

```
       REF  KSCAN        REFERENCE THE KSCAN VECTOR

* CALL LOOKS LIKE THIS:
*
       LI   R4,120       LOAD R4 FOR TWO SECONDS
       BL   @DELAY2      USE SUBROUTINE
       (PROGRAM CONTINUES)
* SUBROUTINE THEN IS:
*
DELAY2 CLR  @>8378       CLEAR THE VDP INTERRUPT COUNTER
DLY2   LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN STOP THEM
       BLWP @KSCAN       SCAN THE KEYBOARD
       CB   @ANYKEY,@>837C HAS ANY KEY BEEN STRUCK?
       JEQ  DLYX         IF SO, EXIT THE SUBROUTINE
       C    R4,@>8378    ELSE COMPARE R4 TO COUNTER VALUE
       JGT  DLY2         IF R4 IS GREATER, REPEAT LOOP
DLYX   RT                ELSE RETURN TO CALLING PGM
*
* DATA SEGMENT WILL NEED THIS:
*
ANYKEY BYTE >20
*
* FOLLOWING IS A KEY-ENTRY SUBROUTINE WITH FLASHING CURSOR
* ON ENTRY, R0 SHOULD BE SET TO THE SCREEN ADDRESS AT WHICH INPUT WILL APPEAR
* ON EXIT, R8 CONTAINS THE STRUCK KEY'S ASCII VALUE
* THE CALLING PROGRAM WILL HAVE TO DECIDE WHETHER TO PUT THE STRUCK
* KEY ON THE SCREEN, OR PUT THE OLD CHARACTER (ALTKEY) BACK
* (AS FOR A CURSOR MOVEMENT KEYSTROKE)
*
       REF  VSBW,VSBR,KSCAN REFERENCE UTILITIES
KENTRY
       BLWP @VSBR        READ THE BYTE FROM CURRENT SCREEN POSITION
       MOVB R1,@ALTKEY   PLACE THAT BYTE AT ALTKEY
KI1
       MOV  @ON,R4       PLACE "CURSOR ON" DURATION IN R4
       LI   R1,>1E00     PUT CURSOR CHARACTER IN R1
       INC  @CURFLG      SET FLAG TO INDICATE CURSOR IS ON
KI1A
       BLWP @VSBW        WRITE CHARACTER TO SCREEN
       CLR  @>8378       CLEAR THE VDP INTERRUPT COUNTER
KI2
       BLWP @KSCAN       SCAN THE KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN SHUT THEM OFF
       CB   @ANYKEY,@>837C  HAS A KEY BEEN STRUCK?
       JNE  KI3          IF NOT, SKIP AHEAD
       MOV  @KEYADR,R8   ELSE PLACE KEY'S VALUE IN R8
       RT                THEN RETURN
KI3    C    R4,@>8378    COMPARE VALUE IN R4 TO VDP INTERRUPT COUNTER
```

```
       JGT  KI2          IF R4 IS GREATER, SCAN KEYBOARD AGAIN
CHNG
       MOV  @CURFLG,R8   CHECK TO SEE IF CURSOR IS ON
       JEQ  KI1          IF NOT, JUMP TO PUT CURSOR ON
       MOV  @OFF,R4      ELSE PLACE "CURSOR OFF" VALUE IN R4
       MOVB @ALTKEY,R1   MOVE THE CURRENT SCREEN CHARACTER TO R1
       CLR  @CURFLG      CLEAR TO INDICATE CURSOR IS OFF
       JMP  KI1A         GO BACK TO CONTINUE LOOP
*
* DATA SECTION NEEDS THE FOLLOWING:
ON     DATA 15           ABOUT 1/4 SECOND CURSOR "ON" TIME
OFF    DATA 25           LONGER TIME FOR CURSOR "OFF"
CURFLG DATA 0
ANYKEY BYTE >20          THE SPACE CHARACTER FOR COMPARISONS
ALTKEY BYTE 0            BYTE TO STORE THE CURRENT SCREEN CHARACTER
*
*
* THE FOLLOWING CODE SUPPLIED BY MERLE VOGT, CONVERTS A WORD TO ASCII STRING
* ON ENTRY, R12 POINTS TO A WORD LOCATION IN MEMORY
* ON EXIT, R12 POINTS AT THE STRING'S LENGTH BYTE
* THE SIGN BIT IS IGNORED, SO DECIMAL RANGES FROM 0 THRU 65535
* R10 IS MODIFIED BY THE SUBROUTINE
*
BINASC CLR  @ASC         CLEAR FIRST TWO BYTES AT ASC
       MOV  *R12,@>835E  MOVE THE WORD TO >835E
       BLWP @GPLLNK      USE GPLLNK
       DATA >2F7C        WITH THIS ADDRESS DATA
       MOVB @>8361,@ASC  GET LENGTH BYTE FOR STRING
       MOVB @>8367,R10   GET LOW BYTE OF STRING ADDRESS
       SRL  R10,8        RIGHT JUSTIFY
       AI   R10,>8300    COMPLETE THE ADDRESS
       LI   R12,ASC+1    POINT AT FIRST CONTENT BYTE
LOOP   MOVB *R10+,*R12+  MOVE ONE BYTE AND INCREMENT POINTERS
       CI   R10,>8367    ARE WE FINISHED?
       JL   LOOP         IF LOW, KEEP GOING
       LI   R12,ASC      POINT R12 AT STRING
       CLR  @>837C       CLEAR GPL STATUS BYTE
       RT                RETURN TO CALLER

* DATA SECTION NEEDS FOLLOWING:
       EVEN
ASC    BSS  6
*
*
* FOLLOWING CODE IS FROM HARRY WILHELM
* USE FOR DISPLAYING STRINGS OR OTHER STUFF IN EXTENDED BASIC ENVIRONMENT

* CALLING SEGMENT FOR A STRING WOULD LOOK LIKE THIS:

       LI   R1,TSTR      POINT R1 TO THE STRING'S ADDRESS
```

```
      LI   R0,11*32+3   POINT R0 AT DESIRED SCREEN LOCATION (E.G ROW 12, COL 4)
       BLWP @PRSTR       BLWP TO THE SUBROUTINE
       (PROGRAM CONTINUES)

* SUBROUTINES ARE SET UP AS BLWP VECTORS
*
PRSTR  DATA >2038,PRSTR1
VMBW60 DATA >2038,VMBW61
*
PRSTR1 BL   @>24CA       USE A SUBROUTINE TO PASS PARAMETERS FROM CALLING WS
       MOVB *R1+,R2      GET STRING LENGTH BYTE INTO R2
       SRL  R2,8         RIGHT JUSTIFY
       JEQ  VMBW6X       IF ZERO, SKIP THE STRING, IT HAS NULL LENGTH
       JMP  VMBW62       ELSE JUMP INTO VMBW62
VMBW61 BL   @>24CA       USE SUBROUTINE TO GET PARAMETERS
VMBW62 MOVB *R1+,R3      MOVE A BYTE INTO R3
       AI   R3,>6000     ADD THE OFFSET FOR XB
       MOVB R3,@>8C00    PLACE AT VDPWD LOCATION
       DEC  R2           DECREMENT CHARACTER COUNT
       JNE  VMBW62       IF NOT ZERO, SEND ANOTHER CHARACTER
VMBW6X RTWP              ELSE RETURN TO CALLERS WS AND CODE
*
* NOTE - WE ADDED THE LINE JEQ  VMBWX JUST IN CASE THE STRING YOU POINTED TO
* BEFORE CALLING THE SUBROUTINE HAS ZERO LENGTH

* SAMPLE DATA THAT COULD BE USED WITH THE ABOVE
TSTR   BYTE 21           LENGTH OF THE STRING
TSTRC  TEXT 'THIS IS A TEST STRING'  FOLLOWED BY ITS CONTENT

*  IF THE LENGTH IS A CONSTANT (NON-ZERO) VALUE, YOU CAN DISPLAY IT THIS WAY:

       LI   R0,11*32+3   R0 POINTS AT DESIRED LOCATION (E.G. ROW 12, COL 4)
       LI   R1,TSTRC     R1 POINTS AT TEXT
       LI   R2,21        R2 CONTAINS LENGTH OF TEXT
       BLWP @VMBW60      BLWP TO SUBROUTINE VECTOR
       (PROGRAM CONTINUES)
*
* FOR THE CURIOUS READER, HERE'S WHAT'S AT >24CA, WHICH WE JUST USED ABOVE
* THE CODE ENTRY POINT AT >24CA IS LABELED HERE AS GETWRT
* THE CODE ENTRY POINT AT >24D0 IS USED FOR READ OPERATIONS, AND IS SKIPPED
* OVER FOR WRITING OPERATIONS (SHOWN HERE AS LABEL GETRD)
* WE HAVE ASSIGNED LABELS TO MAKE IT EASIER TO FOLLOW

GETWRT LI   R1,>4000     WRITING OPERATION MASK TO R1
       JMP  PARSET       SKIP AHEAD
GETRD  CLR  R1           READING OPERATION
PARSET MOV  *R13,R2      MOVE CALLERS R0 INTO SUB'S R2
       MOVB @>203D,@>8C02 SEND LOW BYTE OF SUB'S R2 TO VDPWA LOCATION
       SOC  R1,R2        PUT THE MASK ON R2 FOR READ OR WRITE OPERATION
       MOVB R2,@>8C02    SEND HIGH ORDER BYTE TO VDPWA
```

```
       MOV  @2(R13),R1   GET CALLERS R1 INTO SUB'S R1
       MOV  @4(R13),R2   GET CALLERS R2 INTO SUB'S R2
       RT                RETURN TO MAIN ROUTINE

* THE ABOVE IS USED WITH ITS TWO ENTRY POINTS BY ALL FOUR OF THE VXBX ROUTINES,
* NAMELY VMBW, VSBW, VMBR, VSBR
* IT IS SHOWN HERE FOR REFERENCE ONLY, AND SHOULD NOT BE COPIED INTO YOUR SOURCE
```

1.19. The Art Of Assembly — Part 19. Sticks And Sprites May Break My. . .

One of the truly frustrating things to try in Extended Basic is to create some kind of game using joysticks and sprites together. Everything simply takes too much time. Defining the characters that you want for your sprites takes time, but that can sometimes be "covered" by putting a title screen or instructions on-screen while it's happening. Once you have gotten into the game part, however, there is no hiding the fact that things are just too slow. The fact that you must separately CALL JOYST to see if the stick has moved, and then also CALL KEY to see if the fire button is being pressed is an annoyance and a time-waster. Then of course there's the little matter of CALL COINC. One can work for many hours trying to "tighten" the loop that includes this instruction, and still find that the game's action will fail because COINCs are missed while the computer is doing something else. Usually, one must resort to slowing down the speeds of the sprites so that COINC will be detected.

In Assembly, much of that goes away. For openers, one can construct rather large data sections to define the character shapes, then dump these very quickly into VDP RAM with VMBW. One can also set the VDP registers so that the Sprites have their own separate character definition area in VDP, and so have 256 characters available just for the sprites, without giving up any of the normal screen characters. For example, one can use the default character definition area starting at >800 for the "stationary" screen characters, then use the area at >1000 for 256 sprite shapes. (Do that by LI R0,>0602 then BLWP @VWTR.)

Then one can change shapes of sprites "on the fly" by writing a single byte to VDP RAM at the correct place. One can also "instantly" change their horizontal or vertical velocities, their positions on the screen, and so on. Also, one now has 32 sprites available at any time, and can place all of them in automatic motion if desired.

Perhaps the most powerful difference of all is that with a single call to the KSCAN routine, one can get the joystick information, the fire button sensing, and "keypress" data for the "split" keyboard scan. This saves so much time that one usually needs to build in delays on purpose to keep the action of the game controllable by human operators.

In the Sidebar is a little "snippet" from our game SCUD BUSTERS, in which we are placing a sprite on the screen and moving it around within a defined area based on the joystick. To simplify the example, we have shown only Joystick #1 being used. This is just a loop operation, but there are delays built into it so the sprite will not move too quickly. If this loop is performed without the delay, the sprite being controlled "zips" to the edge of the screen as soon as the joystick is moved.

The delay can of course be modified to make the game respond at any chosen pace. In this game, for example, there were three levels of skill involved, with the joystick response speed tailored to each skill level. The single call to KSCAN with the byte at >8374 set to a value of one gives us the pressing of a key from the left side of the keyboard, or the fire button, reported into the byte at >8375, and gives us the Y and X positions of the stick in the bytes at >8376 and >8377. In other words, that call to KSCAN gives us all the information we need to control the game's action from either joystick or keyboard. In this instance, we ignored the keyboard except for the Q key, which would give us the same key value as the fire button.

To make the action proceed smoothly in single pixel advances of the sprite, we divided the bytes from >8376 and >8377 by four, so that 0 still is 0, but +4 became +1, and -4 became -1. We then simply got the sprite's current position from VDP RAM and added the appropriate byte to that value, then moved it back to VDP RAM to move the sprite by one pixel. Checks were included in the process to keep the sprite always in the screen area designated as our playing field. Diagonal motion was also allowed here, and of course that's made easier by taking both X and Y inputs on one scan.

While all this was going on, there was a sprite in motion by itself, falling in a ballistic arc toward the bottom of the screen. This sprite's motion was modified every 26 passes through the loop, so it would appear to be influenced by the acceleration of gravity. Its constant horizontal velocity and accelerating vertical velocity makes the appearance of a parabolic path like that of a falling ballistic missile. To keep that sprite in motion, we included the LIMI 2 and LIMI 0 instructions in the loop to allow the VDP to produce automatic motion.

Let's digress for just a moment here to discuss what the user sees at this point in the game. There's a black object (the SCUD) falling in an arc toward the bottom of the screen. There's a white object (the Aimpoint) that the user is controlling position of with his joystick. The user must place this aimpoint somewhere between the SCUD and the bottom of the screen, then fire his interceptor missile (Patriot) to meet the Scud in its flight and destroy it. Both the position of the aimpoint and the timing of the firing must be right to make an intercept happen. The speed of the Patriot will be directly proportional to the distance between the aimpoint and the launch point at the lower left corner of the screen. We're a little ahead of ourselves here, so let's get back to the source code.

When the fire button is pressed, the program moves on to a new section of code at label PATFIR. First, it determines the horizontal and vertical speeds for the Patriot, by doing some math on the distance to the aimpoint.

That's all done very quickly by integer math operations. Next the shape is selected from a group of preloaded shapes so the missile will look like it's flying in the correct attitude for its path. (At least approximately.) The position, velocities, and character information are loaded into VDP RAM, and that launches the Patriot. There's also a sound effect.

Once the Patriot is launched, we enter a new loop operation. This one has no delays in it. It's purposes are to see whether the Scud or Patriot has left the active screen area on the sides, bottom, or top, and to see whether the two sprites have come within a predetermined distance of each other. This distance criterion is a larger number (8 pixels) for beginner's skill level than for more advanced players (5 pixels). Thus the loop is doing nothing but reading out from VDP RAM the horizontal and vertical positions of the two sprites that are moving independently, and making comparisons of those numbers. We did not bother looking at the VDP Status byte, as that would not give us the information we needed, and would simply have wasted time.

We've omitted from this source code the part that makes results happen, but here's how it goes. If the SCUD reaches the bottom of the screen, there's a loud explosion sound effect, and a "miss" is scored for the player. If the SCUD leaves the field by the side of the screen, it doesn't count. (The game is designed to make this a very rare happening.) If the Patriot flies off the screen without intercepting the Scud, it simply disappears into the night, and the Scud continues to the ground. If the two sprites get into "coincidence", then an "air burst" explosion occurs, and the two sprites disappear into a "fireball" of three sprites overlapping one another. A hit is scored for the player, then the game goes back to decide by a random number when to launch its next Scud.

The code in this section of the game gets rather involved, but we think that with the annotations, you'll be able to follow the action if you really want to. It also may appear that many operations are performed in each run through the "Coinc" loop, and that is so, but execution takes very little time. The "patriot" sprite can fly very quickly, but we have never seen a case where a coincidence has been missed in playing the game. As we look at it now, there probably are some unnecessary steps taken in there, but it doesn't slow anything down enough to cause trouble. That, of course, is the ultimate test. If it works, we are going to leave it alone.

What we have shown is of course just a fragment of the code for the game, and without its other parts it won't assemble. The point of showing this is to illustrate using an actual section of code that does work as intended in its context.

We expect you'll quickly reach the conclusion that dealing with Sprites and Joysticks in Assembly is a tedious business, with all those tiny steps to be performed. It's certainly a long way from the Extended Basic CALL SPRITE, CALL JOYST, CALL KEY, and CALL COINC to what's shown here. Maybe it's not worth all that, but you can find out only by doing it yourself and seeing that the performance of a program is worth the effort.

If you are programming for yourself, not for commercial sales, then it will always be worth the effort to prove that it can be done. On the other hand, writing commercial stuff, sometimes a program that eats months of time to create will sell five or six copies and then die. That can become a real pain! It gets worse! At the 92 TICOFF show, there was a youngster who kept re-visiting our table and playing SCUD BUSTERS for what seemed hours at a time. His father came by, looked at what was going on, but instead of offering to buy this for his child, made the assertion that "That's on the BBS." We pointed out to him that this program was a unique copyrighted product of Harrison, and that it was definitely not on anyone's BBS! He was sure something like this was available on the BBS. No sale!

Next month's topic is undecided at this time. We'll leave you with this little joke, which originally applied to farming, but seems appropriate in this business. "How do you make a small fortune writing software?" "Start out with a large fortune and write software for a while."

```
* FRAGMENT OF SOURCE CODE FROM "SCUD BUSTERS" GAME
* EXAMPLE OF USE OF SPRITES AND JOYSTICKS
* FIRST SECTION SETS UP AN "AIMPOINT" AS
* SPRITE #0, MOTIONLESS AT THE MIDDLE OF THE SCREEN
*
* EQUATES
MOTBL  EQU  >780
ATTLST EQU  >300
DESTBL EQU  >800
COLTBL EQU  >380
KEYADR EQU  >8374
KEYVAL EQU  >8375
YVAL   EQU  >8376
XVAL   EQU  >8377
STATUS EQU  >837C
* NOTE - THE SPRITE DESCRIPTOR TABLE WAS SET TO EQUAL
* THE CHARACTER DEFINITION TABLE BY:
       LI   R0,>0601
       BLWP @VWTR
* THIS SETS VDP REGISTER 6 TO LOOK FOR SPRITE CHARACTERS STARTING AT >800
NEWSCN
       INC  @FIRFLG      PERMIT ONE FIRING
       LI   R0,ATTLST    POINT AT START OF SPRITE ATTRIBUTE LIST IN VDP RAM
       LI   R1,>6000     MIDDLE OF SCREEN IN "Y" POSITION
       BLWP @VSBW        WRITE THAT TO Y-POSITION FOR SPRITE 0
       INC  R0           POINT AT X-POSITION
       LI   R1,>8000     MIDDLE HORIZONTAL POSITION
       BLWP @VSBW        WRITE THAT TO X-POSITION SPRITE 0
       INC  R0           POINT TO CHARACTER FOR SPRITE 0
       LI   R1,>0000     MAKE IT CHARACTER 0
       BLWP @VSBW        WRITE THE CHARACTER NUMBER
       INC  R0           POINT AT COLOR BYTE FOR SPRITE 0
       LI   R1,>0F00     MAKE IT WHITE
       BLWP @VSBW        WRITE THAT TO SPRITE 0 COLOR BYTE
       LI   R3,2         SET FOR SPRITE 2
       BL   @DELX        DELETE THAT ONE
* NOW A RANDOM DELAY BETWEEN 1/2 AND 1 1/2 SECONDS OCCURS BEFORE
* THE SCUD APPEARS
       LI   R3,60        SET R3 TO 60
       BL   @RANDNO      RANDOM NUMBER WILL BE 0-60
       AI   R5,30        ADD 30 TO THE RANDOM NUMBER, SO IT'S 30-90
       CLR  @>8378       CLEAR THE VDP INTERRUPT TIMER
DLY1
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            TURN THEM OFF
       C    @>8378,R5    COMPARE TIMER TO R5
```

```
       JLT  DLY1         IF LESS, CONTINUE LOOPING
*
* FOLLOWING ESTABLISHES THE STARTING POSITION AND MOTION FOR THE SCUD
* THEN STARTS THE SCUD FALLING FROM THE TOP OF THE SCREEN
*
       LI   R0,ATTLST+4  POINT AT Y-POSITION BYTE FOR SPRITE 1 (SCUD)
       LI   R1,>0A00     VERTICAL POSITION AT DOT-ROW 10
       BLWP @VSBW        PLACE THE SPRITE THERE
       INC  R0           POINT TO X-POSITION SPRITE 1
       LI   R3,226       SET RANGE FOR RANDOM NUMBER AT 226
       BL   @RANDNO      GET A RANDOM NUMBER 0-226
       AI   R5,10        ADD 10 SO NUMBER IS 10-236
       SWPB R5           SWAP SO NUMBER IS IN LEFT BYTE R5
       MOVB R5,R1        MOVE THAT BYTE TO R1
       MOVB R5,R7        AND STASH IT IN R7
       BLWP @VSBW        WRITE THE HORIZONTAL POSITION FOR SPRITE 1
       INC  R0           POINT AT CHARACTER BYTE SPRITE 1
       LI   R1,>5300     SET CHARACTER VALUE
       BLWP @VSBW        WRITE THAT
       INC  R0           POINT AT COLOR BYTE SPRITE 1
       LI   R1,>0100     SET FOR BLACK
       BLWP @VSBW        WRITE THAT
       INC  R0           POINT TO Y-POSITION FOR SPRITE 2
       LI   R1,>D000     SET FOR "DELETE" POSITION
       BLWP @VSBW        WRITE THAT BYTE
       CLR  R1           CLEAR REGISTER 1
       LI   R0,MOTBL     POINT TO SPRITE MOTION TABLE
       BLWP @VSBW        WRITE 0 TO Y-MOTION FOR SPRITE 0
       INC  R0           INCREMENT TO X-MOTION SPRITE 0
       BLWP @VSBW        MAKE THAT ZERO TOO
       AI   R0,3         POINT AHEAD TO Y-MOTION FOR SPRITE 1
       MOVB @INIV,R1     MOVE AN INITIAL VELOCITY INTO THAT BYTE
       BLWP @VSBW        WRITE INITIAL Y-MOTION
       LI   R3,15        SET FOR RANDOM NUMBER
       CLR  R10          CLEAR REGISTER 10
       BL   @RANDNO      GET A RANDOM NUMBER 0-15
       INC  R5           ADD ONE SO IT'S 1-16
       CB   R7,@HEX80    SEE WHICH HALF OF SCREEN SPRITE 1 IS IN HORIZONTALLY
       JLE  POSXV        IF LOW OR EQUAL, JUMP
       NEG  R5           ELSE MAKE R5 = - R5
       INC  R10          INCREMENT R10
POSXV  SWPB R5           SWAP BYTES IN R5
       MOVB R5,R1        PLACE BYTE IN R1
       INC  R0           POINT AT X-MOTION FOR SPRITE 1
       BLWP @VSBW        WRITE THE X-MOTION
       MOVB @MOTION,@>837A SET NUMBER OF SPRITE THAT MAY HAVE MOTION
       SWPB R5
* FOLLOWING SECTION CHOOSES A SHAPE FOR THE SPRITE
* BASED ON ITS DIRECTION OF MOTION
       MOV  R10,R10
```

```
       JEQ  CLEAR4
       NEG  R5
CLEAR4 CLR  R4
       DIV  @SIX,R4
       LI   R1,99
       MOV  R10,R10
       JEQ  ADDFOR
       S    R4,R1
       JMP  SWAONE
ADDFOR A    R4,R1
SWAONE SWPB R1
       LI   R0,ATTLST+6
       BLWP @VSBW
NEWLOP MOV  @KEYDLY,R6
* HERE WE BEGIN THE LOOP THAT LOOKS FOR THE JOYSTICK AND FIRE BUTTON
* INPUTS.  R6 IS PRESET TO 26
*
       MOVB @ONE,@KEYADR SET FOR KEYBOARD SCAN UNIT ONE
LOOP   CLR  @STATUS      CLEAR GPL STATUS BYTE
       BLWP @KSCAN       SCAN KEYBOARD AND JOYSTICK
       LIMI 2            PERMIT INTERRUPTS
       LIMI 0            THEN STOP THEM
       CB   @KEYVAL,@FIRE HAS FIRE KEY BEEN STRUCK?
       JNE  CMPN1        IF NOT, JUMP AHEAD
       MOV  @FIRFLG,R1   IS FIRING PERMITTED?
       JEQ  CMPN1        IF NOT, JUMP
       B    @PATFIR      ELSE FIRE THE PATRIOT
* HERE WE SET UP AND ENTER A DELAY LOOP THAT SIMPLY WASTES TIME TO
* KEEP THE JOYSTICK FROM RESPONDING TOO QUICKLY

CMPN1  MOV  @RPTDLY,R4   PUT A DELAY FACTOR IN R4
DLY    SRL  R5,15        WASTE SOME TIME
       LIMI 2            PERMIT INTERRUPTS
       LIMI 0            THEN SHUT THEM OFF
DEC4   DEC  R4           DECREMENT COUNTER
       JNE  DLY          IF NOT ZERO, REPEAT LOOP
       LI   R0,ATTLST+5  POINT AT X-POSITION SPRITE 1
       BLWP @VSBR        READ THAT BYTE
       CB   R1,@RTLIM    COMPARE TO RIGHT LIMIT OF SCREEN
       JH   GONEW        IF HIGH, JUMP
       CB   R1,@LFTLIM   COMPARE TO LEFT LIMIT
       JL   GONEW        IF HIGH, JUMP
* EVERY 26TH TIME THROUGH TO THIS POINT, THE VERTICAL
* VELOCITY OF THE SCUD IS INCREASED BY FIVE
*
DEC6   DEC  R6           DECREMENT OUTER LOOP COUNT
       JNE  REPRT        IF NOT ZERO, JUMP
       LI   R0,MOTBL+4   ELSE POINT AT Y-MOTION BYTE SPRITE 1
       CLR  R1           CLEAR R1
       BLWP @VSBR        READ Y-MOTION INTO LEFT BYTE R1
```

```
       AB   @FIVE,R1     ADD FIVE
       CI   R1,>7F00     COMPARE TO MAXIMUM VELOCITY
       JH   REPRT        IF HIGH, SKIP
       BLWP @VSBW        ELSE WRITE NEW Y-MOTION FOR SPRITE 1
       MOV  @KEYDLY,R6   RESET COUNT IN R6
REPRT  LI   R0,ATTLST+4  LOOK AT Y-POSITION SPRITE 1
       BLWP @VSBR        READ THAT BYTE
       CB   R1,@DWNLIM   COMPARE TO BOTTOM LIMIT
       JL   REPRT3       IF NOT AT BOTTOM, JUMP
       JMP  STOPIT       ELSE "MISS" PROCESS
REPRT3 INC  R0           POINT AT X-POSITION SPRITE 1
       BLWP @VSBR        READ THAT
       CB   R1,@RTLIM    CHECK RIGHT LIMIT
       JL   REPRT2       JUMP IF NOT OKAY
GONEW  B    @NEWSCN
REPRT2 CB   R1,@LFTLIM   CHECK LEFT LIMIT
       JH   REPRT1       JUMP IF OKAY
       B    @NEWSCN      ELSE CANCEL SCUD
* IN THIS SECTION THE JOYSTICK INPUTS ARE USED TO CHANGE
* THE POSITION OF THE AIMPOINT SPRITE
* POSITION IS CHECKED TO KEEP THE POINTER WITHIN THE
* RECTANGULAR AREA OF THE SCREEN DESIGNATED FOR PLAY
REPRT1 LI   R0,ATTLST    POINT AT Y-POSITION SPRITE 0 (AIMPOINT)
       BLWP @VSBR        READ THAT INTO R1
       MOVB @YVAL,R2     MOVE THE BYTE FROM >8376 INTO R2
       SRA  R2,2         DIVIDE BY FOUR
       SB   R2,R1        SUBTRACT FROM POSITION IN R1
       CB   R1,@UPLIM    COMPARE TO UPWARD LIMIT
       JL   XCHK         IF NOT OKAY, JUMP AHEAD
       CB   R1,@DWNLIM   ELSE COMPARE TO LOWER LIMIT
       JH   XCHK         IF NOT OKAY, JUMP
       BLWP @VSBW        ELSE WRITE NEW Y-POSITION FOR SPRITE 0
XCHK   INC  R0           POINT AT X-POSITION BYTE FOR SPRITE 0
       BLWP @VSBR        READ THAT
       MOVB @XVAL,R2     MOVE THE BYTE FROM >8377 INTO R2
       SRA  R2,2         DIVIDE BY FOUR
       AB   R2,R1        ADD THAT TO X-POSITION IN R1
       CB   R1,@RTLIM    COMPARE TO RIGHT SIDE LIMIT
       JH   LOOP         IF NOT OKAY, SKIP BACK
       CB   R1,@LFTLIM   COMPARE TO LEFT SIDE LIMIT
       JL   LOOP         IF NOT OKAY, JUMP BACK
       BLWP @VSBW        ELSE WRITE NEW X-POSITION SPRITE 0
       JMP  LOOP         THEN GO BACK TO START OF LOOP
STOPIT
* THE SECTION AT STOPIT MAKES DECISIONS ABOUT WHETHER A HIT OR MISS
* HAS HAPPENED, UPDATES SCORE, DISPLAYS SCORE, AND SO ON
* SCUDC - THIRD PART OF MAIN CODE
*
PATFIR
       LIMI 0            SHUT OFF INTERRUPTS
```

```
FIRIT
       CLR  @FIRFLG      ONLY ONE FIRING PER SCUD PERMITTED
       LI   R0,ATTLST    POINT AT Y-POSITION SPRITE 0 (AIMPOINT)
       BLWP @VSBR        GET THAT BYTE IN R1
       INC  R0           POINT AT X-POSITION SPRITE 0
       SWPB R1           SWAP BYTES IN R1
       BLWP @VSBR        LEFT BYTE IS XPOS SPRITE 0
*                        RIGHT BYTE IS YPOS SPRITE 0
       MOV  R1,R3        TEMPORARILY STORE R1 IN R3
       MOVB @LFTLIM,R1   GET LEFT EDGE OF SCREEN IN R1
       SWPB R1           SWAP R1
       MOVB @DWNLIM,R1   PUT BOTTOM OF SCREEN IN R1
       LI   R0,ATTLST+8  POINT AT Y-POSITION SPRITE 2 (PATRIOT)
       BLWP @VSBW        PLACE THAT BYTE
       SWPB R1           SWAP BYTES
       INC  R0           POINT AT X-POSITION SPRITE 2
       BLWP @VSBW        WRITE THAT (LEFT SIDE OF SCREEN)
       MOV  R1,R2        LEFT BYTE IS XPOS SPRITE 2
*                        RIGHT BYTE IS YPOS SPRITE 2
      LI   R1,>680F     >68 IS STARTING CHARACTER FOR SPRITE 2, HEX F IS COLOR
       INC  R0           POINT AT CHARACTER BYTE SPRITE 2
       BLWP @VSBW        WRITE THAT
       SWPB R1           SWAP BYTES IN R1
       INC  R0           POINT AT COLOR BYTE SPRITE 2
       BLWP @VSBW        WRITE COLOR WHITE
       SWPB R3           SWAP BYTES IN R3
       SWPB R2           AND IN R2
       MOV  R2,R4        STASH R2 IN R4
       MOV  R3,R1        GET R3 BACK IN R1
       SRL  R2,8         RIGHT JUSTIFY R2
       SRL  R1,8         RIGHT JUSTIFY R1 ALSO
       S    R2,R1        SUBTRACT THESE NUMBERS
       SRA  R1,1         NOW CUT NUMBER IN R1 IN HALF
       JLT  LDYV2        IF LESS THAN ZERO, JUMP
       NEG  R1           ELSE MAKE R1= -R1
LDYV2  SWPB R1           NOW SWAP BYTES
       MOVB R1,R9        SAVE THE BYTE IN R9
       LI   R0,MOTBL+8   POINT AT Y-MOTION BYTE FOR SPRITE 2 (PATRIOT)
       BLWP @VSBW        WRITE THAT MOTION
       MOV  R4,R2        GET R4 BACK INTO R2
       SWPB R3           SWAP R3
       SWPB R2           AND R2
       MOV  R3,R1        PUT R3 IN R1
       SRL  R1,8         RIGHT JUSTIFY
       SRL  R2,8         AND R2 AS WELL
       S    R2,R1        SUBTRACT THESE NUMBERS
       SRA  R1,1         CUT R1 IN HALF
       JGT  LDXV2        IF POSITIVE, JUMP
       NEG  R1           ELSE MAKE R1= -R1
LDXV2  SWPB R1           SWAP THE BYTES
```

```
       MOVB R1,R7        STASH BYTE IN R7
       INC  R0           POINT AT X-MOTION FOR SPRITE 2
       BLWP @VSBW        WRITE THE X-MOTION
* FOLLOWING CODE CHOOSES A SHAPE FOR SPRITE 2 BASED ON
* ITS DIRECTION OF MOTION
       LI   R1,>6A00
       SRA  R9,8
       JEQ  PUTPAT
       NEG  R9           R9 IS POSITIVE #
       LI   R1,>6600
       SRL  R7,8
       JEQ  PUTPAT
       CLR  R8
       DIV  R7,R8
       CI   R8,8
       JGT  PUTPAT
       AB   @ONE,R1
       CI   R8,2
       JGT  PUTPAT
       AB   @ONE,R1
       CI   R8,1
       JEQ  PUTPAT
       SLA  R9,1
       C    R9,R7
       JGT  PUTPAT
       AB   @ONE,R1
       SLA  R9,2
       C    R9,R7
       JGT  PUTPAT
       AB   @ONE,R1
PUTPAT LI   R0,ATTLST+10 POINT AT CHARACTER BYTE FOR SPRITE 2
       BLWP @VSBW        WRITE SELECTED SHAPE
       LI   R10,>2200    POINT AT "IN FLIGHT" SOUND EFFECT
       MOV  R10,@>83CC   SET THAT FOR SOUND PROCESSING BY VDP
       SOCB @ONE,@>83FD  AND START THE SOUND
       MOVB @ONE,@>83CE  EFFECT
* THE LOOP STARTING AT LABEL COINC KEEPS CHECKING THE POSITIONS OF SPRITES 1 &
2
* TO SEE IF EITHER SCUD OR PATRIOT LEAVES ACTIVE SCREEN AREA
* OR IF THEY "MEET" WITHIN THE PRESET LIMITS
COINC  LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN STOP THEM
       LI   R0,ATTLST+4  POINT AT Y-POSITION SPRITE 1 (SCUD)
       BLWP @VSBR        READ THAT BYTE
       SWPB R1           SWAP R1
       INC  R0           POINT AT X-POSITION SPRITE 1
       BLWP @VSBR        READ THAT BYTE
       CB   R1,@RTLIM    HAS SPRITE GONE OFF TO RIGHT?
       JL   CHKL         IF NOT, CHECK LEFT
       B    @STOPIT      ELSE GET OUT OF HERE
```

```
CHKL    CB   R1,@LFTLIM   HAS SPRITE GONE OFF TO LEFT?
        JH   CHKX         IF NOT, JUMP AHEAD
        B    @STOPIT      ELSE GET OUT OF HERE
*
* IN THE NEXT SECTION, THE DIFFERENCES BETWEEN THE POSITIONS OF SPRITES 1 & 2
* ARE COMPARED TO THE TOLERANCES UPTOL AND DWNTOL, TO SEE IF COINCIDENCE HAS
* OCCURRED.  AT BEGINNER LEVEL, UPTOL AND DWNTOL ARE SET TO +8 AND -8,
* WHILE AT HIGHER SKILL LEVELS THEY ARE +5 AND -5, RESPECTIVELY.
*
CHKX    MOV  R1,R3        STASH R1 IN R3
        AI   R0,3         R3 HAS X,Y SPRT 1
        BLWP @VSBR        READ Y POSITION SPRITE 2 (PATRIOT)
        SWPB R1           SWAP R1
        INC  R0           POINT AT X POSITION SPRITE 2
        BLWP @VSBR        READ THAT BYTE
        MOV  R1,R2        R1 HAS X,Y SPRT 2
        SB   R3,R2        SUBTRACT THE X POSITIONS OF THE TWO SPRITES
        CB   R2,@UPTOL    COMPARE RESULT TO UPWARD TOLERANCE
        JLE  CHKY         IF LOW OR EQUAL, CHECK Y TOLERANCE
        CB   R2,@DWNTOL   ELSE COMPARE X POSITION TO DOWN TOLERANCE
        JHE  CHKY         IF HIGH OR EQUAL, JUMP
        JMP  POSCKS       ELSE JUMP AHEAD, NO COINCIDENCE
CHKY    SWPB R2           SWAP THE BYTES IN R2
        SWPB R3           AND IN R3
        SB   R3,R2        SUBTRACT
        SWPB R3           SWAP R3 AGAIN
        CB   R2,@UPTOL    COMPARE Y POSITION DIFFERENCE
        JLE  EXOUT        IF LOW OR EQUAL, COINCIDENCE HAS HAPPENED
        CB   R2,@DWNTOL   ELSE COMPARE TO DOWN TOLERANCE
        JHE  EXOUT        IF HIGH OR EQUAL, COINCIDENCE HAS OCCURRED
POSCKS
        CB   R1,@RTLIM    COMPARE SPRITE 2 POSITION
        JH   PAST         IF HIGH, DELETE PATRIOT
        CB   R1,@LFTLIM   COMPARE TO LEFT LIMIT
        JL   PAST         IF LOW, SAME ACTION
        SWPB R1           SWAP R1
        CB   R1,@UPLIM    PAST UPPER LIMIT?
        JL   PAST         IF SO, DELETE PATRIOT
        CB   R1,@DWNLIM   DOWN LIMIT?
        JH   PAST         IF SO, DELETE PATRIOT
* NEXT SECTION DOES SIMILAR COMPARISONS FOR THE SCUD (SPRITE 1)
*
        CB   R3,@RTLIM
        JH   CANSCU
        CB   R3,@LFTLIM
        JL   CANSCU
        SWPB R3
        CB   R3,@UPLIM
        JL   PAST
        CB   R3,@DWNLIM
```

```
       JH   PAST
       JMP  COINC        BOTH SPRITES CONTINUE "FLYING", JUMP BACK TO REPEAT
EXOUT
* CODE FROM THIS POINT ON IS DEVOTED TO SCORING, MAKING SOUND EFFECTS, ETC
```

1.20. The Art Of Assembly — Part 20. The Sounds Of The TI

By Bruce Harrison

One of the precious "free" gifts included in our beloved TI is the sound chip. This little jewel, with its three main voices and its "noise" generator, makes a wide variety of sound effects possible without any additional hardware required. Only two PC manufacturers have seen fit to include such capability: IBM in the now orphaned PCjr, and Tandy in their 1000 series of PCs. Both of those chose to use the exact same TI chip that's in the 99/4A.

On the TI, one can make some very good sounds, and of course even music, from Basic or Extended Basic. Even though the sound is made in durations of 1/60 second, some really good music has been produced that way. (On the above-mentioned PCs, sounds in basic are timed in increments of 1/18.2 second, making decent music virtually impossible.)

In the Assembly realm, we have considerably more flexibility available than in the Basic and XB realms. Here, we can choose not only what sounds we want, but can choose to time their durations in many ways, and even produce simulated "instrument" effects, like harpsichord, flute, snare drum, and so on. The techniques we've used to produce instrument effects could fill more than one installment of this series by themselves, and we'll get to that one day, but for today we'll concentrate on simpler uses for the "sound chip".

1.20.1. The Sound List Method

Let's start with a rather simple application, in which we want a noise or a series of musical notes to occur while something else is happening. TI provided an automatic sound processing capability in the VDP so that one could "have his cake and eat it too". The VDP can be given a list of sounds to produce, and told to start making those, then the computer can go on with other business, looking for keystrokes from the keyboard, or looking for joystick inputs, sprite coincidences, and so on, while the sound list executes "on background". That's certainly a handy feature. It does require the instructions LIMI 2 and LIMI 0 to let the VDP continue its sound processing on an interrupt basis, but that's a small price to pay for the capabilities that it gives us.

Sound lists may be placed in VDP RAM at the beginning of a program, and then activated when needed, without needing to be re-loaded.

The first part of today's Sidebar shows one such application, in which the sound lists for three different effects are pre-loaded early in a program and then activated when the program needs them. These particular lists are from the game Scud Busters. In that case, the "in flight" sound can be interrupted at any time by one of the "explosions" depending on what happens to the sprites that are in motion on the screen. The interruption process is very simple. Note that we need not stop the "in flight" effect, but just put the right addresses in place to start the explosion, and processing of the "in flight" effect stops. This will not always be the case, depending which generators were being used by the first sound being processed. There are a couple of "safe" ways of dealing with that possibility. Perhaps the easiest is to put a "shut up" sound at the beginning of each sound list, with a duration of 1. That will shut down all four of the generators for 1/60th of a second before the new sound starts. You can also put "silence" bytes in your first "note" for the new sound to shut off any generators not used by that sound effect, and that will make an "instant" interruption of the previous sound effect. That's the method used in our Sidebar source code.

The explosion sound effects are allowed to run for their full duration in all cases, and serve thereby as timers to allow the user time to see the explosion screen display. We mentioned back in number 18 the potential use of sound lists as timers, and here is a practical example of that use.

Each sound list should end with a "zero duration" sound so that it will end without leaving a generator "hanging" when the intended sound ends. You'll notice that each of the sound lists shown ends that way. The content part of a sound list is outlined well enough in the E/A book itself, but you will see some tricks used in our implementation that are not covered in the book. The list beginning at ENDSND, for example, includes a note for generator 3 in the bytes >C2 and >0E, but then sets that generator's volume at silence by >DF. It then sets the noise generator to produce a noise subharmonic of the generator 3 note at maximum volume by sending >E3 and >F0 bytes. During the rest of the list, it alternates the noise generator's response by sending bytes of >F5 and >F3 in successive "notes". Also, the volume of the noise is decayed by changing the noise generator's volume from >F0 down to >FA before the final line in the list, at which all generators are set to silence. This alternation of the "note" and decaying of volume creates a kind of "pulsing" explosion sound with two distinct noises heard while the volume decays. Of course this particular list was the result of a good deal of experimenting to get just the effect we wanted. The byte >E3 is equivalent to Basic's -4 sound, while the >E5 is equivalent to the Basic -6 sound. Knowing that can let you use Basic or XB to experiment with sounds before you try them in a sound list.

Of course if you're going to do that, you must also bear in mind the relationship between durations in Basic and Assembly sound lists. In this case, the Basic and Extended Basic books have lied to you. Those books indicate that you can specify sound durations in milliseconds. This is just not true. Any number of milliseconds you indicate up to and including 16 will produce exactly the same duration of sound, namely 1/60th of a second. Indicating 17 will get you 2/60ths, as will 18, 19, 25, or 30 milliseconds. The crossover point from one actual duration to the next is every 16.666. . . (sixes all the way across the page if you like) milliseconds. We'll leave the math to you, but you can successfully experiment with your sounds in Basic or XB as long as you remember that the real durations are in 1/60ths of a second, and translate into "milliseconds" for Basic. We never said this would be easy!

As it happens, the examples we've shown all involve use of the noise generator, not the musical voices of generators 1 through 3, but that shouldn't hinder your efforts if you understand the principles involved.

All of the above presupposes that you have an area of the VDP RAM memory that can remain available for the duration of your program. In this case, we simply assigned small blocks of VDP RAM at addresses >2050, >2100, and >2200 for our sound lists, then left those areas untouched during execution of the program. If our program had disturbed those areas, we would have had to reload our sound lists each time we wanted to use them. In most cases you'll find any address above >1000 will do, so long as you don't go beyond >37D7.

There will be cases where the sound list method won't do the job, and for that reason we'll now show at least one more method for "doing sound". Let's start with the assumption that you are using VDP RAM for some purpose that will preclude setting any of it aside for sound lists. You can still use them, but in a different manner. Unfortunately, these methods will not permit a true "background" process for sound, but will require timing loops of some kind in your own code.

1.20.2. Direct To The Generator

You can send sound defining bytes directly to the sound chip at address >8400, then use your own method to time the durations. For openers, we'll consider a method that uses the exact same sound list as shown in the previous example, but will not load the sound list into VDP RAM. This method will still time the sounds in 1/60th second increments using the VDP Interrupt timer, but will do that timing in the "foreground" program.

As shown in the Sidebar starting at label METH2, you'll need a pointer set to the beginning of the sound list. We've used R9 here, but any register that's handy will do. The first byte in the sound list is the number of bytes that constitute the "note" being sent. We take that first byte into R4 and then use R4 as a counter. Each of "count" bytes is then sent to the sound chip at address >8400. The byte immediately after the last "generator" byte is the duration, and here we've put that byte in R4, then right justified this number in that register. If that number is zero, we are at the end of the sound list, so we simply jump out of the sound section of code. Otherwise, we clear the VDP Interrupt counter, then simply start looping with a LIMI 2 and LIMI 0, and a comparison between R4 and the VDP Interrupt counter. As long as R4 is greater than the value in the counter, we keep repeating the loop. Once the counter gets equal to or more than R4, the sound "note" is finished, so we jump back to process the next "note" in the list. We have used the expression "note" here to mean a set of instructions passed to the sound chip, which includes both note values for the generators and volume values for those generators. As we've mentioned before in this series, the duration here for any one note may not exceed 255, or >FF, which makes a note last 4 1/4 seconds.

There is one trick required to use the direct method. We'll pass that along without knowing why it's so: At the very beginning of the program, before loading your own workspace, you must execute a MOV R11,@ANYWRD instruction. ANYWRD here means just that. You can move R11 to >8300, for example, or to some word location in your own data section. You won't need it again, but if you don't execute that MOV instruction before loading your own workspace, the direct method will not work properly. Please don't ask why this is so. It just is!

Since this method uses the VDP Interrupt counter to time the durations of the notes, the same duration values that were used when we placed the sound list in VDP RAM will work. As before, the last "note" must have a duration of zero to signal that we're at the end of a sound list. We recommend a "note" like this be the last in the list:

```
BYTE 4,>9F,>BF,>DF,>FF,0
```

That will "shut down" all four generators in the chip by setting each to a silent volume level.

The code shown for this method can also be set up as a subroutine starting at label NXTNOT, with an RT instruction at label SNDEX. Then one could play different sound lists by:

```
LI   R9,SNDLST
BL   @NXTNOT
```

The subroutine will modify the values in R9 and R4, but will leave all others alone.

1.20.3. More Exotic Methods

There are other ways to do the sounds, and once you've crossed the bridge into sending bytes directly to the sound chip, you can do things that were impossible in the "sound list" method. We'll just touch on those today.

First, let's suppose that 1/60th second is too long a duration for you. You want a succession of very swift notes to play, like the strumming of a guitar or lute, or you want some noise like automatic weapons firing in rapid succession. These cases are where the real power of the "direct" method comes into play. Instead of timing durations with the VDP Interrupt timer, you can construct a simple delay loop of your own, and use a word value instead of a byte to do the timing. This way, you can make incredibly short sounds and incredibly long ones without strain, since the "duration" can run from 1 through 65,535, and the amount of time each loop takes can be tailored to your own needs by inserting "time wasters" into the loop. One of our favorites is to do a DIV operation within the timing loop. That wastes time very nicely, and can be used for other purposes related to the sound you're creating. The SRC instruction can also be used for this purpose, and it will serve well.

We of course have used this third method for our "Assembly Music" products, and have been able to fine tune the response of the generators to simulate musical instruments of various kinds. We made changes to the volumes and notes on the generators while a "note" was playing. Thus an instrument like a piano or harpsichord could be simulated by using an exponential decay in volume during each note played. Barry Boone has carried that concept even farther with his SOUND F/X product, to produce spectacular effects and even spoken words in a recognizable voice without a speech synthesizer.

Next month we'll go on with this subject, revealing some of our "trade secrets" used in the Assembly Music that we are known for. That will include at least one of our most advanced "instrument" subroutines.

```
* TWO METHODS FOR USING SOUND LIST DATA TO PRODUCE SOUND EFFECTS
* FIRST CAN ALLOW SOUNDS TO PROCESS IN BACKGROUND WHILE THE PROGRAM
* PERFORMS OTHER ACTIONS
* CODE BY B. HARRISON
* PUBLIC DOMAIN
*
* THE FOLLOWING HAPPENS AT THE BEGINNING OF THE PROGRAM, TO PRELOAD THE
* SOUND LISTS INTO THE VDP RAM AREAS
*
       LI   R0,>2050     POINT AT FIRST LIST AREA
       LI   R1,SNDDAT    BEGINNING OF FIRST SOUND LIST
       LI   R2,ENDSND-SNDDAT LENGTH OF FIRST SOUND LIST
       BLWP @VMBW        WRITE THAT TO VDP RAM
       A    R2,R1        ADD LENGTH OF FIRST LIST
       LI   R0,>2100     POINT TO VDP RAM AREA FOR SECOND LIST
       LI   R2,LSOUND-ENDSND  LOAD R2 WITH LENGTH OF SECOND LIST
       BLWP @VMBW        WRITE THAT
       A    R2,R1        ADD LENGTH OF SECOND LIST
       LI   R0,>2200     POINT AT ADDRESS FOR THIRD LIST
       LI   R2,BLANK-LSOUND  LENGTH OF THIRD LIST
       BLWP @VMBW        WRITE THAT TO VDP RAM
* PROGRAM CONTINUES
*
* FOLLOWING CODE SECTION STARTS THE "IN FLIGHT" SOUND EFFECT FOR THE PATRIOT
* WHEN THE PATRIOT MISSILE IS LAUNCHED
*
INFLT  LI   R10,>2200    POINT AT "IN FLIGHT" SOUND LIST
       MOV  R10,@>83CC   MOVE THAT ADDRESS TO >83CC
       SOCB @ONE,@>83FD  TURN ON
       MOVB @ONE,@>83CE  VDP SOUND PROCESSING
COINC  LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN SHUT THEM OFF
* A LOOP HERE LOOKS FOR SPRITE COINCIDENCE
* AND FOR THE SCUD TO REACH BOTTOM OF SCREEN
* DEPENDING WHICH HAPPENS, PROGRAM JUMPS TO EITHER CRASH OR CRASH2
*
CRASH
*
* CODE THAT PLACES A GROUND BURST EXPLOSION EFFECT ON SCREEN GOES HERE
```

```
*
       LI   R10,>2100    POINT AT VDP ADDRESS FOR "GROUND BURST" EFFECT
       JMP  CRASH1       THEN JUMP
CRASH2
*
* CODE THAT MAKES AN AIR BURST ON SCREEN GOES HERE
*
       LI   R10,>2050    POINT AT VDP ADDRESS FOR "AIR BURST" EFFECT
CRASH1 LIMI 0            STOP INTERRUPTS FOR NOW
       MOV  R10,@>83CC   PLACE SOUND LIST ADDRESS AT >83CC
       SOCB @ONE,@>83FD  THEN START
       MOVB @ONE,@>83CE  VDP SOUND PROCESSING
       LIMI 2            PERMIT INTERRUPTS
SNDLOP MOVB @>83CE,R10   TIMING LOOP FOR THE SOUND
       JNE  SNDLOP       CONTINUES LOOPING UNTIL SOUND LIST HAS FINISHED
       LIMI 0            DISCONTINUE INTERRUPTS
* PROGRAM CONTINUES WHEN EXPLOSION SOUND ENDS
*
* SECOND METHOD, USES "SOUND LIST" IN MAIN MEMORY, DOES NOT REQUIRE USE OF
* VDP RAM FOR THE SOUND LIST
*
SOUND  EQU  >8400        DEFINE THE SOUND CHIP ADDRESS
*
*
METH2
       LI   R9,SNDDAT    POINT AT "SOUND LIST" IN RAM
NXTNOT
       MOVB *R9+,R4      GET THE "COUNT" BYTE INTO R4
       SRL  R4,8         RIGHT JUSTIFY IN R4
       JEQ  SNDEX        IF ZERO, GET OUT OF PROCESS
MOVSND
       MOVB *R9+,@SOUND  MOVE A SOUND DEFINING BYTE TO THE CHIP
       DEC  R4           DECREMENT COUNT OF BYTES IN THIS NOTE
       JNE  MOVSND       IF NOT ZERO, REPEAT PROCESS
       MOVB *R9+,R4      ELSE GET THE "DURATION" BYTE INTO R4
       SRL  R4,8         RIGHT JUSTIFY IN R4
       JEQ  SNDEX        IF ZERO, THIS IS END OF SOUND LIST, SO GET OUT
       CLR  @>8378       ELSE CLEAR THE VDP INTERRUPT COUNTER
SNDLOP
       LIMI 2            ALLOW INTERRUPTS BRIEFLY
       LIMI 0            THEN SHUT THEM OFF
       C    R4,@>8378    COMPARE R4 TO VDP INTERRUPT COUNT
       JGT  SNDLOP       IF R4 IS GREATER, WE'RE NOT FINISHED WITH THIS NOTE
       JMP  NXTNOT       ELSE WE ARE FINISHED, GO BACK FOR NEXT NOTE
SNDEX
* PROGRAM CONTINUES HERE

*
* IN DATA SECTION, THREE SOUND LISTS
* FIRST MAKES "AIR BURST", SECOND "GROUND BURST", THIRD IS "IN FLIGHT"
```

```
*
SNDDAT
       BYTE 5,>9F,>BF,>DF,>E5,>F2,3
       BYTE 2,>E5,>F0,9
       BYTE 2,>E5,>F2,8
       BYTE 2,>E5,>F4,6
       BYTE 2,>E5,>F6,4
       BYTE 2,>E5,>F8,2
       BYTE 2,>E5,>FA,1
       BYTE 1,>FF,0
ENDSND
       BYTE 7,>9F,>BF,>C2,>0E,>DF,>E3,>F0,3
       BYTE 2,>E5,>F0,15
       BYTE 2,>E3,>F2,3
       BYTE 2,>E5,>F2,12
       BYTE 2,>E3,>F4,2
       BYTE 2,>E5,>F4,10
       BYTE 2,>E3,>F6,2
       BYTE 2,>E5,>F6,8
       BYTE 2,>E3,>F8,1
       BYTE 2,>E5,>FA,6
       BYTE 4,>FF,>DF,>BF,>9F,0
LSOUND BYTE 5,>E4,>F0,>9F,>BF,>DF,12
       BYTE 1,>F1,10
       BYTE 1,>F3,8
       BYTE 1,>F5,7
       BYTE 1,>F8,6
       BYTE 1,>FC,5
       BYTE 1,>FF,0
BLANK  DATA 0            DATA SECTION CONTINUES HERE
```

1.21. The Art Of Assembly — Part 21. Sound Trade Secrets

By Bruce Harrison

Last month we promised to share some of our "trade secrets" with you, and today we'll make good on that promise. We warn you in advance, though, that this won't be easy going for the reader. It wasn't easy for us either, when first we decided to make a "piano" sound on the TI. Today's Sidebar contains lots of source code, all derived from one of our Assembly Music products. What's shown is not a complete picture, since that would eat up much of this issue of the magazine. It is, though, an example taken directly from our actual source code, and annotated to make it possible for you to follow what it's doing.

The bulk of today's Sidebar is the source code for our "piano" subroutine, which uses data put together by Dolores P. Werths, our resident musician. A small segment of that data source is shown here also. The principle involved in the piano subroutine is to send a note to the sound chip at >8400, then cause an exponential decay of the volume for that note while the generators are playing it. The exponential decay is simulated by using a "decay factor" located among the "variables" at the end of the subroutine's code. This is normally set at a fraction of the duration of a 64th note, the value of which is set by a "TEMPO" file, shown here. The value of a 64th note is just a number, typically ranging from about 180 to about 400. That sets the speed at which the music plays. The decay timing is self-modifying during each note's duration, so that the note decays quickly at first, then more gradually as it's duration goes on. That's accomplished by doubling the temporary decay number each time the volume is reduced.

To put that in perhaps more easily understood terms, let's say that the note's duration calls for 4000 passes through the timing loop, and that the decay factor starts at a value of 50. The first 49 passes through the timing loop will thus be made without changing the volumes. On the 50th pass, the volumes for all generators will be reduced by 2 decibels, and the decay number will double, so the next reduction will happen at the 100th pass through the loop. At that pass, the decay will go to 200, then 400, then 800, and so on. Each time the decay happens, the next one is moved later in the time duration of the note. This makes a "curved" decay process with respect to time, and simulates the way the sound produced by a piano string decays after the string is struck by the piano's hammer.

On each decay cycle, we check the current volume level of each generator to see if it's reached the silence level, and leave it alone if it has reached silence. Otherwise, we increment the volume being sent to that generator by one, thus lowering the volume output by two decibels. We're getting ahead of ourselves a bit.

The code section shown in today's Sidebar is complex, to say the least, and is not really complete in any sense, else it would fill many pages. It's a small section of the source code for one of the "Nannerl's Notebook" compositions by Leopold Mozart. The "Piano" subroutine is complete as shown, but only a small portion of the other necessary files is shown to serve as an example of how this subroutine is used. Some of what's shown here actually creates no output to the object file. The sections called "NOTES" and "NAN11T" simply establish mnemonic labels so that the musician can create data source files in a quasi-musical notation. The portion of "NOTES" shown covers only one octave of generator one's notes. The full file carries this scheme through seven octaves for all three generators. This same file establishes simple mnemonic labels for the volumes of notes, starting with V11 being the loudest volume for generator 1.

The "NAN11T" part is shown here in full, and it establishes the duration values for mnemonic labels in "musical" notation based on the value TEMP, which is the duration for a 64th note. To adjust speed of playing, the musician changes only the value for TEMP, and the assembler calculates all the others.

In the musical data part, we have shown only one measure of the piece. There would also be an "action" section of code, consisting only of parts like this:

```
LI   R9,M001
LI   R13,E005
BL   @LDMEAS
```

That would cause measures 1 through 5 to be played by the subroutine. Many measures can be played with just one call to the subroutine, since it will continue playing until the R9 pointer to the data equals or exceeds the R13 pointer. Where portions of the music repeat, loops are used to count the number of times a section plays. Registers 3, 4, and 5 are not used by the subroutine, so that single, double or triple nested loops can be counted using those three registers in the "action" part of the piece's source code. In some cases, like at label M001A in the Sidebar, extra labels are employed so that parts of a measure can be re-used elsewhere in the piece.

This particular implementation of the "piano" subroutine has a special feature to make it easy for the musician to have notes play as normal durations, or as staccato or legato notes. Legato means the note plays for the full duration, staccato means it plays for 3/4 duration, while "normal" means 7/8ths duration. To do that, this subroutine takes advantage of two "facts". First, that specifying the notes and volumes for three generators won't take more than 9 bytes, and second that a single byte can range up through 255. Thus the "tens" place in the "number of bytes" byte and one digit in the "hundreds" place could be used by the musician to signal things to the subroutine.

The code for this signaling works like this: If one or more generators are to play "legato", then there's a 100 in the number. The subroutine simply divides the "number of bytes" by 100, and sends the quotient to the Legato flag word (LEGFLG). It then divides the remainder by 10, and uses the quotient from that division to decide which generators are affected by legato or staccato. If the quotient of the division by ten is zero, then all generators will play at "normal" duration, which is 7/8ths of the stated duration. The encoding of the "tens" place works like this:

VALUE	*Meaning*
1	gen 1
2	gen 2
3	gen 3
4	gens 1 and 3
5	gens 2 and 3
6	gens 1, 2, and 3
7	gens 1 and 2

The subroutine "decodes" this portion of the byte and sets flag bytes in the table at SETFLG to indicate which generators are affected by the staccato or legato condition. The remainder from the division, left in R8, is the number (1-9) of bytes to be sent to the sound chip.

Thus if the musician is specifying a new note and duration for each of the three generators, and wants a legato on generators 2 and 3, with generator one playing a normal duration, the "number of bytes" byte would be given as 159. For a staccato on generators 2 and 3, the "bytes" byte is entered as 59.

Using decimal entries in this way made it much easier for the musician to comprehend what was going on and to control the actions of the subroutine. Of course the Assembler converts all these numbers to hex notation in the object code.

When this kind of data is assembled, there will be extra bytes left in the object code, and those will be set at zero by the assembler. For example, if the source looks like this:

```
BYTE 3
DATA AZ1
BYTE V31
DATA SX
```

The Assembled values, assuming we started at an even location, would look like this in hex notation:

```
0300 803C 9200 015D
```

The subroutine detects these zero valued bytes and simply ignores them, since 0 is not a legal value for the sound chip. It also checks before taking in the duration bytes to be sure it's at an even location, since of course the left byte of the duration could be zero, and that must not be ignored.

As always, when we look at source code in preparing these articles, we see room for improvement. For example, at label START, it would work just as well and consume less memory if it looked like this:

```
START   INC  R9
        COC  @ONE,R9
        JEQ  START
        MOV  *R9+,R1
```

That would save us a couple of instructions, since at the point where the original code says MOVB *R9+,R1, we know we're at an even location, and thus could simply move the word as shown above. Of course this isn't going to mean we re-assemble all the music stuff just to save those few bytes, but once again it shows the fallible human qualities of your author. As you learn Assembly, you will always find such things in stuff you wrote last year, and may be tempted to kick yourself for them. Don't. Everybody who does assembly code will tell you this is common.

We're not going to bore everyone silly at this point with a line-by-line examination of the Sidebar. That should make everyone happy, including the Publishers. By now, if you've followed this series, you can figure out our contorted logic from the Sidebar itself.

As you may know, we have released all our Assembly Music products to Public Domain, which means that many of them are available through User Group libraries, and all of them are available from TIGERCUB's PD catalog. (Catalog for a refundable $1.00 from TIGERCUB SOFTWARE, 156 Collingwood Avenue, Columbus, OH, 43213.) We have also made up a disk of the complete source code for one number (Tchaikovsky's March from the Nutcracker Suite) that we offer at nominal cost ($3.00 incl. S&H) for those who want to explore the realm of Assembly music. (Harrison Software, 5705 40th Place, Hyattsville MD 20781, ask for the March source code disk.)

Next month's topic is undecided. Perhaps we'll pursue a reader's suggestion, or some crazy idea of our own. We do appreciate having heard from some of our readers. It brightens our sometimes dull days.

```
* ASSEMBLY MUSIC SOURCE CODE EXCERPTS
* THESE ARE FRAGMENTS FOR ILLUSTRATION
*
* CODE BY B. HARRISON EXCEPT WHERE NOTED
*
* FIRST PARTS ARE ALL EQUATES TO MAKE MNEMONIC LABELS AVAILABLE
*
*  NOTES - EQUATES FOR MUSIC PROGRAMS
*  SAVED AS "NOTES"
*
A1     EQU  >893F        FIRST OCTAVE
AZ1    EQU  >803C        GENERATOR ONE
BJ1    EQU  >803C
B1     EQU  >8A38
C1     EQU  >8735
CZ1    EQU  >8732
DJ1    EQU  >8732
D1     EQU  >8A2F
DZ1    EQU  >8F2C
EJ1    EQU  >8F2C
E1     EQU  >872A
F1     EQU  >8128
FZ1    EQU  >8D25
GJ1    EQU  >8D25
G1     EQU  >8B23
GZ1    EQU  >8B21
```

```
A1J1   EQU  >8B21        2ND OCTAVE
A11    EQU  >8C1F        GENERATOR ONE
* SIMILAR ENTRIES CONTINUE FOR ADDITIONAL OCTAVES ON ALL GENERATORS

*  VOLUME LEVELS
*
V11    EQU  >0090        MAX LOUD GEN ONE
V21    EQU  >0091
V31    EQU  >0092
V41    EQU  >0093
V51    EQU  >0094
V61    EQU  >0095
V71    EQU  >0096
V81    EQU  >0097
V91    EQU  >0098
V101   EQU  >0099
V111   EQU  >009A
V121   EQU  >009B
V131   EQU  >009C
V141   EQU  >009D
V151   EQU  >009E
V161   EQU  >009F        SILENCE GEN ONE
* SIMILAR ENTRIES FOR GENERATOR 2 V12, V22, ETC. CONTINUE

* NANNERL'S NOTEBOOK #11 MARCHE IN F
*  FILE NAN11T - TEMPO VARIABLES
*  THE VALUE AT LABEL TEMP GOVERNS PLAYING SPEED FOR THE MUSIC
*
TEMP   EQU 349      VALUE 4 THRU 2000
SX     EQU TEMP     64TH NOTE DURATION
T      EQU 2*TEMP   32ND NOTE DURATION
S      EQU 4*TEMP   16TH NOTE
E      EQU 8*TEMP   8TH NOTE
Q      EQU 16*TEMP  QUARTER
H      EQU 32*TEMP  HALF
TT     EQU S/3      TRIPLET 32ND NOTE
TS     EQU E/3      TRIPLET 16TH
TS2    EQU TS/2     HALF OF THAT
TE     EQU Q/3      TRIPLET EIGHTH
TE2    EQU TE/2     HALF THAT
TE3    EQU TE/3     ONE THIRD OF THAT
TE4    EQU TE/4     ONE QUARTER OF THAT
STP    EQU SX/4     VERY SHORT NOTE

* PROGRAM ENTRY POINT SHOULD CONTAIN:

ENTRY  MOV  R11,@>8300   MOVE REGISTER 11
       LWPI WS           THEN LOAD YOUR WORKSPACE
*
* CODE TO INVOKE THE PIANO FOR ONE MEASURE LOOKS LIKE THIS
```

```
       LI   R9,M001      BEGINNING OF A MEASURE
       LI   R13,E001     END OF A MEASURE
       BL   @LDMEAS      CALL THE SUBROUTINE TO PLAY THIS SECTION
*
* THE PIANO SUBROUTINE IS THIS:
* "GRAND PIANO" SUBROUTINE
* MODIFIED FOR AUTO STACCATO AND LEGATO
*
SOUND  EQU  >8400
LDMEAS
LM1    MOVB *R9+,R1      GET THE "NUMBER" BYTE INTO R1
       JEQ  LM1          IF ZERO, TRY NEXT BYTE
       SRL  R1,8         RIGHT JUSTIFY IN R1
       MOV  R1,R8        MOVE THE "NUMBER" WORD TO R8
       CLR  R7           CLEAR REGISTER 7
       CLR  @SETFLG      CLEAR A FLAG WORD
       CLR  @SETFLG+2    AND THE WORD AFTER THAT
       DIV  @DECHUN,R7   DIVIDE R7-R8 PAIR BY 100
       MOV  R7,@LEGFLG   R7 WILL = 1 FOR A LEGATO NOTE
       CLR  R7           CLEAR R7 AGAIN
       DIV  @DECTEN,R7   DIVIDE R7-R8 PAIR BY TEN
       MOV  R7,R7        CHECK TO SEE IF R7 IS ZERO
       JEQ  NORMA0       IF SO, JUMP AHEAD
       CI   R7,7         ELSE COMPARE TO 7
       JEQ  SET2         IF SO, JUMP
       CI   R7,3         COMPARE R7 TO 3
       JLT  SET2         IF LESS, JUMP
       MOVB @ONE+1,@SETFLG+2 ELSE SET FOR STACCATO/LEGATO ON GENERATOR 3
       AI   R7,-3        SUBTRACT 3 FROM R7
       JEQ  NORMA0       IF ZERO, JUMP AHEAD
SET2   CI   R7,2         ELSE COMPARE TO 2
       JLT  SET1         IF LESS, JUMP
       MOVB @ONE+1,@SETFLG+1 ELSE SET FLAG FOR GENERATOR 2
       AI   R7,-2        SUBTRACT 2
       JEQ  NORMA0       IF ZERO, JUMP
SET1   MOVB @ONE+1,@SETFLG ELSE SET FLAG FOR GENERATOR 1
       CLR  R7           CLEAR R7
NORMA0 MOV  R8,R1        MOVE R8 BACK TO R1
LM2    CB   *R9,@ZERO    ARE WE POINTING AT A ZERO BYTE?
       JEQ  INC9         IF SO, JUMP
       CLR  R8           ELSE CLEAR R8
       MOVB *R9,R8       MOVE A "NOTE" BYTE INTO R8
       SRL  R8,12        SHIFT BY 12 SO ONLY THE LEFT NYBBLE IS USED
       COC  @ONE,R8      IS THIS AN ODD NUMBER?
       JNE  BLAH         IF NOT, JUMP AHEAD
       AI   R8,-9        ELSE SUBTRACT 9
       JLT  BLAH         IF LESS THAN ZERO, JUMP
* AT THIS POINT, WE KNOW THAT THE BYTE POINTED TO BY R9 IS A VOLUME BYTE
       SRL  R8,1         ELSE DIVIDE NUMBER BY 2
       CB   *R9,@SNDOFF(R8) SEE IF IT'S A "SILENCE" BYTE
```

```
       JEQ  BLAH1        IF SO, JUMP AHEAD
       CB   *R9,@STACOT(R8) SEE IF IT'S A "STACCATO" VALUE
       JNE  LM7          IF NOT, JUMP
       MOVB @SNDOFF(R8),@SOUND ELSE SILENCE THE GENERATOR
       MOVB @SNDOFF(R8),@VOLUME(R8)  AND PUT SILENCE IN THE VOLUME TABLE
       JMP  BLAH1        THEN JUMP
LM7    MOVB *R9,@VOLUME(R8) PLACE THE BYTE IN VOLUME TABLE
BLAH   MOVB *R9,@SOUND   MOVE THE BYTE TO THE SOUND CHIP
BLAH1  DEC  R1           DECREMENT BYTE COUNT
       JEQ  START        IF ZERO, JUMP AHEAD
INC9   INC  R9           ELSE INCREMENT R9
       JMP  LM2          THEN JUMP BACK TO PROCESS NEXT BYTE
START  INC  R9           POINT AT NEXT BYTE IN DATA
       COC  @ONE,R9      IS R9 AN ODD LOCATION?
       JEQ  START        IF SO, MOVE ON TO NEXT BYTE
       MOVB *R9+,R1      GET HIGH BYTE OF DURATION INTO R1
       SWPB R1           SWAP
       MOVB *R9+,R1      GET LOW BYTE OF DURATION INTO R1
       SWPB R1           SWAP SO R1=DURATION WORD FROM DATA
*
* THE FOLLOWING FIVE LINES ARE PART OF A "CALIBRATION" PROCESS TO MAKE THE MUSIC
* PLAY AT THE SAME PACE ON A GENEVE AS IT DOES ON A TI - TO MAKE THIS EFFECTIVE,
* A "DUMMY" RUN THROUGH THE SUBROUTINE'S INNER LOOP (LM4) WAS DONE AT THE
* BEGINNING OF THE PROGRAM.  THESE LINES ARE COMMENTED OUT BECAUSE THEY CAN'T
* BE USED WITHOUT THAT DUMMY RUN HAVING BEEN PERFORMED.
*
*      MOV  R1,R0        MOVE R1 INTO R0
*      CLR  R1           CLEAR R1
*      MPY  @TINUM,R0    MULTIPLY R0 BY THE TI CALIBRATION NUMBER
*     DIV  @CALNUM,R0   THEN DIVIDE R0-R1 PAIR BY THE PRESENT MACHINE'S NUMBER
*      MOV  R0,R1        MOVE THE QUOTIENT BACK TO R1
       MOV  R1,@SHTCNT   PLACE THAT FOR NOW AT LABEL SHORT COUNT
       MOV  R1,R8        AND MOVE IT INTO R8
       MOV  @LEGFLG,R7   CHECK FOR LEGATO INDICATOR
       JNE  STK2         IF FLAG SET, JUMP
       MOVB @SETFLG,R7   CHECK FLAG FOR GEN 1
       JNE  STK4         IF NOT ZERO, JUMP
       MOVB @SETFLG+1,R7 CHECK FLAG FOR GEN 2
       JNE  STK4         IF NOT ZERO, JUMP
       MOVB @SETFLG+2,R7 CHECK FLAG FOR GEN 3
       JEQ  STR1         IF ZERO, JUMP
STK2   MOV  R1,R7        ELSE PLACE DURATION IN R7
       SRL  R7,3         DIVIDE BY 8
       JMP  STK3         THEN JUMP
STK4   MOV  R1,R7        PLACE DURATION IN R7
       SRL  R7,2         DIVIDE BY 4
STK3   S    R7,R8        SUBTRACT 1/8 OR 1/4 FROM DURATION
       MOV  R8,@SHTCNT   THEN MOVE THE RESULT TO SHORT COUNT
STR1   MOV  R1,@DURAT    AND MOVE FULL DURATION TO DURAT
       MOV  @ONE,R1      THEN SET R1 AT VALUE 1
```

```
       MOV  @DECNT,R15   PLACE THE DECAY VALUE IN R15
LM4    MOV  R1,R8        MOVE THE CURRENT COUNT TO R8
       CLR  R7           CLEAR R7
       DIV  R15,R7       DIVIDE R7-R8 PAIR BY DECAY COUNT IN R15
       MOV  R8,R8        IS THERE ANY REMAINDER?
       JNE  DEC1         IF REMAINDER NON-ZERO, JUMP AHEAD
       LI   R14,VOLUME   ELSE POINT R14 AT VOLUME TABLE
       LI   R7,4         FOUR GENERATORS TO CHECK
INLP   CLR  R8           CLEAR R8
       MOVB *R14,R8      GET A BYTE FROM VOLUME TABLE
       SLA  R8,4         SHIFT LEFT TO STRIP OFF GENERATOR NYBBLE
       CB   @SILENT,R8   COMPARE BYTE TO "SILENCE" VALUE
       JEQ  INLP5        IF SILENT, JUMP AHEAD
       AB   @ONE+1,*R14  ELSE ADD ONE TO THE BYTE IN THE TABLE
INLP5  MOVB *R14,@SOUND  AND MOVE THAT BYTE TO THE SOUND CHIP
INC14  INC  R14          INCREMENT POINTER TO NEXT BYTE IN VOLUME TABLE
       DEC  R7           DECREMENT COUNTER
       JNE  INLP         IF NOT ZERO, REPEAT
       SLA  R15,1        ELSE MULTIPLY DECAY COUNT IN R15 BY 2
DEC1   INC  R1           INCREMENT LOOP COUNT IN R1
       C    R1,@SHTCNT   COMPARE TO SHORT DURATION COUNT
       JNE  DEC1C        IF NOT EQUAL, JUMP
       C    @SHTCNT,@DURAT ELSE COMPARE SHORT COUNT TO FULL DURATION
       JEQ  DEC1C        IF EQUAL, JUMP AHEAD
       CLR  R8           ELSE CLEAR R8
       LI   R7,4         PUT 4 IN R7
       CLR  R14          CLEAR REG 14
       MOV  @LEGFLG,R8   MOVE LEGATO COUNT INTO R8
       JNE  DEC1A        IF NOT ZERO, JUMP
INLP2A MOVB @SETFLG(R14),R8 ELSE MOVE A FLAG BYTE TO R8
       JEQ  INC14A       IF ZERO, JUMP
       MOVB @SNDOFF(R14),@SOUND ELSE SILENCE THIS GENERATOR
       MOVB @SNDOFF(R14),@VOLUME(R14) AND THE VOLUME TABLE BYTE
INC14A INC  R14          INCREMENT POINTER IN R14
       DEC  R7           DECREMENT COUNTER
       JNE  INLP2A       IF NOT ZERO, REPEAT
       JMP  DEC1C        ELSE JUMP
DEC1A  CLR  R8           CLEAR REGISTER 8
       MOVB @SETFLG(R14),R8 MOVE A FLAG BYTE TO R8
       JEQ  INC14B       IF ZERO, JUMP AHEAD
       MOVB @SNDOFF(R14),@SOUND ELSE SILENCE THE GENERATOR
       MOVB @SNDOFF(R14),@VOLUME(R14) AND THE BYTE IN VOLUME TABLE
INC14B INC  R14          INCREMENT POINTER
       DEC  R7           DECREMENT COUNT
       JNE  DEC1A        IF NOT ZERO, REPEAT
DEC1C  C    R1,@DURAT    SEE IF DURATION COUNT IN R1 = FULL DURATION
       JNE  LM4          IF NOT, GO BACK TO START OF LOOP
       C    R9,R13       SEE IF WE'RE AT END OF POINTED DATA SECTION
       JGT  RETRN        IF GREATER, GET OUT OF SUBROUTINE
       JEQ  RETRN        IF EQUAL, SAME ACTION
```

```
       B    @LDMEAS      ELSE GO BACK FOR NEXT NOTE
RETRN  RT                EXIT THE SUBROUTINE
* END OF SUBROUTINE
* DATA SECTION INCLUDES THE FOLLOWING:
*
DECNT  DATA TEMP/2+10    DECAY COUNT
ONE    DATA 1            VALUE ONE AS A WORD
DECHUN DATA 100          100 AS A WORD
DECTEN DATA 10           10 AS A WORD
DURAT  DATA 0            NOTE DURATION
LEGFLG DATA 0            LEGATO FLAG WORD
SETFLG BYTE 0,0,0,0      GENERATOR FLAG BYTES
SHTCNT DATA 0            SHORTENED DURATION COUNT
STACOT BYTE >9E,>BE,>DE,>FE OLD STACCOTO METHOD
VOLUME BYTE >9F,>BF,>DF,>FF GENERATOR VOLUME TABLE
SNDOFF BYTE >9F,>BF,>DF,>FF "SILENCE" BYTE TABLE
*
* FOLLOWING IS A DATA EXCERPT FOR THE FIRST MEASURE OF ONE PIECE
* NANNERL'S NOTEBOOK #11 MARCHE IN F
* MEASURE ONE ONLY
* MUSICAL DATA BY DOLORES P. WERTHS
*
M001   BYTE 9            NINE BYTES IN THIS NOTE
       DATA G21,A22,F13  2ND OCT G GEN 1, 2ND OCT A GEN 2, 1ST OCT F GEN 3
       BYTE V31,V52,V53  VOL 3 GEN 1, VOL 5 GEN 2, VOL 5 GEN 3
       DATA TT           TRIPLET 32ND DURATION
       BYTE 3            THREE BYTES IN NEXT NOTE
       DATA F21          2ND OCT F GEN 1
       BYTE V31          VOL 3 GEN 1
       DATA TT           TRIPLET 32ND
       BYTE 3
       DATA E21
       BYTE V31
       DATA TT
       BYTE 113          LEGATO ON GEN 1, 3 BYTES
       DATA F21
       BYTE V31
       DATA E
       BYTE 163          LEGATO GENS 1,2, AND 3 , 3 BYTES
       DATA G21
       BYTE V31
       DATA S
M001A  BYTE 119          LEGATO ON GEN 1, 9 BYTES
       DATA A31,A22,F13
       BYTE V31,V52,V53
       DATA S+T
       BYTE 115          LEGATO ON GEN 1, 5 BYTES
       DATA G21
       BYTE V31,V152,V153
       DATA T
```

```
        BYTE 119
        DATA A31,A22,F13
        BYTE V31,V52,V53
        DATA S+T
        BYTE 115
        DATA B3J1
        BYTE V31,V152,V153
E001    DATA T
```

1.22. The Art Of Assembly — Part 22. The Business End

By Bruce Harrison

From time to time in this column we've broken away from the drier aspects of Assembly Language programming. Today we thought we should devote a whole column to a topic we have never seen touched, namely the business of making a business out of this programming "hobby". Like the Assembly programming itself, this isn't easy.

1.22.1. The Rules

Rule number one is that there are no rules. Each company that chooses to serve this very special market seems to take its own unique approach. Some fall by the wayside using "sound" business practices, while some succeed in spite of very bad practices. Our own practices in such areas as record keeping are shamefully lax, yet we have been known to show a profit in some years.

For us, the only real Rule number one is to serve our customers. We exist as a business to serve them, not to serve ourselves. We'll stay around and make modest profits only so long as people in the community can trust us to serve their needs. There in the last part of that sentence is the biggest single problem for any software business, since to serve the needs of TI owners, one must know what those needs are. Most of the time, this boils down to guessing what might be needed, and then keeping one's goals in perspective.

There's always someone who expresses needs like "What I need is a MacIntosh emulation program for my TI". Laugh if you like, but some of the letters in *MICROpendium* come pretty close to that kind of need statement. It certainly might be a big seller, but could also become a bottomless pit for the programmer.

1.22.2. Think Big

If anything can kill off a TI programming effort, this is it. Anyone remember PRESS? That, in our opinion, was a victim of the THINK BIG syndrome. We see this all over the place in the PC market, where for example a popular word processor occupies five Megabytes on the hard disk, and needs somewhere around one Megabyte of memory to operate. In that market, of course, such products actually succeed, because there's continuous pressure for people to upgrade to "bigger and better" machines. As the machines get bigger and faster, the programs for them get bigger and slower. Programmers for the PC have no incentive to write efficient programs.

For those of us trying to stay "within bounds" on the TI, such thinking must be studiously avoided. After some time, there is a kind of "sixth sense" we develop which keeps us from trying to make Lotus Symphony for the TI. (Somebody will probably try that soon. Write it in C, and it will all be easy.)

1.22.3. Lucky Guesses

In a lot of cases, we've done things because of our own needs, then found that there were others with that same need, and therefore a market existed. Two examples should serve to illustrate this point.

Some time ago, we had a need to move files back and forth between our PCs and our TIs. Since we have only single density disk capability on our TIs, PC Transfer was not an option. Thus we decided to create a little "utility" to transfer text files (actually source code) via a simple RS-232 connection. Once that was working, my partner Dolores suggested that I should "dress it up" a bit and release it as a commercial product. Months (maybe 12 of them?) went by, with her suggesting and my resisting, but when the "dressing up" was done, we had Smart Connect, which was one of our all-time best sellers.

Likewise our Word Processor, which was originally written to satisfy our own needs, found a good market among others who hate TI-Writer. When that product was reviewed by Stan Krajewski, who himself always disliked the TI-Writer approach, we suddenly found there were lots of people who felt the same way. Sales soared. This made us, and presumably a lot of others, happy people.

Conversely, some of our worst disasters as commercial products have been those that we imagined someone else would need, even though we didn't need such things ourselves. That's one reason our catalog has shrunk from time to time, as products we introduced were quietly discontinued.

1.22.4. Think Small

This little bon mot was not our invention, but sound advice given us by Jim Peterson. We're told that Leopold Mozart gave similar advice to his son Wolfgang Amadeus. There are plenty of small things people with TIs need. Many of these small things have found their way into our public domain utility disks, and into Jim Peterson's catalog. A few have resulted in commercial products like Easy Data. Not all small things are good ones, of course, but when we hit upon something which fills a "niche", it's good for the whole community. Of course a small product that sells many copies is good for our bottom line as well.

1.22.5. Ship The Product

In Philadelphia, where your author grew up, there was a bakery with a little reminder to the clerk on a sign beside the cash register. The sign said "Register first, Wrap after". Today, there is a tendency among some software companies to carry that idea to extremes. Checks are deposited (Register first) months before the products are shipped (Wrap after). We don't think that's right, especially in the TI "family". Over all the years we have been selling products to TI users, we have never (knock on wood) had a customer's check bounce. We've therefore adopted the opposite policy from that espoused by the bakery. Ship the product first, then take the check to the bank.

Yes, that's a radical idea, but one we think should be common among firms serving the TI community. Customers do notice when disks arrive before their checks clear, and they'll feel much better about ordering your products when their orders get prompt service. Customers also notice little things like enclosing a letter with each product shipped, and having the company's phone number offered up for product assistance.

While we are on the subject of radical ideas, we think it's also a good idea in some cases to offer a full refund if the customer is unable to use the product, or misunderstood its purpose, or just plain dislikes it. It's better for your business in the long run to refund $5.00 or $10.00 to a customer than to have that customer forever feeling cheated. We have given refunds on a number of occasions, and will continue to follow that practice.

1.22.6. Don't Quit Your Day Job

Your author should be the last one to offer this advice, having "quit his day job" over a year ago. Of course that wasn't quitting, but retirement after 33 years with the Government, and there is an annuity income to keep the wolf from our door. In spite of our own status, we recommend that programmers just starting a business should make it an "evenings and weekends" operation for some time. Unlike the PC market, which is still growing, the TI market must inevitably shrink a little with each passing year, as more TIs get shoved aside by the ubiquitous PCs and Macs. It's not quite at the "buggy whips" stage yet, but there will be no new Fortune 500 companies arising from the TI Software market. Keep your day job, and get ready to spend endless hours at the machine. Say goodbye to friends and family, if you have any. The business will consume all the excess energy you've got. If your products start to sell, it gets tougher, because you'll have no time to develop new products while selling the existing ones. My partner went through a serious case of pneumonia thanks to that kind of pressure.

There are ways around this dilemma. Our friend Chris Bobbitt, for example, rarely writes any software himself, but concentrates his efforts on selling programs written by others. That has, for the most part, worked out well for him. Asgard will remain one of the "giants" in our cottage industry as long as there are some diehard TI users around.

Other companies, like Texaments, also rely mainly on what might be called "guest authorship" for the bulk of their software products. Our little operation may be among the minority in selling only what our own in-house staff of two people can create. If you're a gifted programmer, and like most, not a businessman, the idea of becoming an "author" for Asgard or Texaments may be better than forming your own company. Of course we did not follow this advice either, but maybe another adage will fit here. An old friend of ours in Philadelphia, who's a very successful businessman, gave us this sage advice: "It's good to learn from your mistakes, but it's even better to learn from other people's mistakes." We are forever grateful to Sal Roggio for that one.

1.22.7. People Are Funny

Here we go again with thanks, this time to Art Linkletter. For our younger readers, he didn't always sell Contour Chairs for a living, but had a radio (and later TV) show with the above title. Of course in the case of a software business, it should read "Customers are funny". There's another old saw that says "The Customer is Always Right". In some respects that's true, but sometimes the customer needs some educating, or just more information than he started out with.

Customers can sometimes do really funny things. Recently, one of our customers received a shipment from us and promptly returned the disks because they were SS/SD disks. That was a "first" in our experience. We offer most of our products on SS/SD disks because that's the one format that every TI system will work with. Our customer was thoughtful enough to include his phone number in the note he sent with the disks. We called. "What kind of system do you have?" "Myarc mini-system." "Did you try out the disks?" "No, my TI system isn't set up just now." He'd bought a PC, and didn't have room for both systems to be operational. It seems this fellow does everything on his TI in DS/DD format, and was unaware that the SS/SD disks could be read by his drives. We sent the disks back to him with a plea that he try using them, or copying them onto DS/DD initialized disks if he'd like. Of course we offered to send DS/DD format if his system really wouldn't read SS/SD. That will be a problem, since our company doesn't own a TI system with DS/DD capability, but we'll find a way if we must. We do have some friends in the business.

Other examples abound, but we'll only throw in one more. A customer who had bought Smart Connect called to say that he couldn't get the PRINTINST program to work, and so could not print the instructions for using the program. "What happens when you try running PRINTINST?" "We get an error message I/O error 00." A sudden chill crept through my body. "Does the disk drive light come on before this error report?" "Just a minute, I'll try it again and see." A pause while the customer walked from the phone to the TI and back again. "No, the drive light doesn't come on, and the message says I/O error 00 in 100." By now we had our own TI fired up, with a copy of Smart Connect in Drive 1 and with PRINTINST loaded under XB. Line 100 says OPEN #2:"PIO.LF",OUTPUT. We all know how PIO behaves, and this certainly isn't it. Error code 00 means BAD DEVICE NAME. Hmmmm.

After a few more minutes of discussion, we determined that his printer was connected via RS-232, and that this particular RS-232 card had no PIO port on it. We gave the customer some instructions for editing line 100 to send the instructions to his printer on the serial port, then rung off. Presumably this worked, as no more phone calls came back. We've never seen an RS-232 card for the TI which lacked a PIO port, but it appears there are such beasts around. You learn something every day.

In all fairness, that particular customer was not himself a TI owner, but had bought Smart Connect for a relative who was a TI owner with a newly acquired PC. Still, nobody thought of looking at what line 100 contained before calling, much less of editing it for RS-232 output.

1.22.8. What Do Those Tea Leaves Mean?

The success of our little Smart Connect product has got us to wondering why it's so popular. Obviously, a lot of TI owners now have PCs as well as TIs. There are two distinct schools of thought about why so many PC/TI owners want to transfer files between the two machines.

Our opinion is that this is step one of two steps, with the second being to place the TI on the auction block or the garbage heap, after transferring all "important" text files to the PC. If we're right, that means the beginning of the end for us as a business. Maybe that makes us the ultimate pessimist, but we've always said that the advantage of pessimism is that we may be surprised, but are rarely disappointed at what happens. Of course our own example is different from that dismal scenario. We have a total of three PCs which coexist with our two TIs, and your author still uses the TI for everything except those things that must be done on the PC. But we are not typical of the "average" user anyway.

Our good friend Barry Traver takes another message from the evidence we just cited, that people are looking for some kind of coexistence between the two machines. He and we agree that there are capabilities in the TI that no PC offers at any price. Maybe, he reasons, the TI is being kept for its strengths, while the text files are being transferred to the PC so it can serve for taking care of more mundane matters like correspondence.

Who's right? Who knows? This time perhaps nobody's right. At one time we would have predicted that we'd "closet" our TIs within a few months of getting our first PC, yet here we are years later, still banging these articles out on the little 99/4A keyboard. I think it was Mark Twain who said something about the reports of his death being "greatly exaggerated". Our TI seems to be saying the same thing.

1.22.9. What To Do

How would we know? The only way to find out whether you can become a giant of industry is to try doing it. You may be the one who comes up with the next major breakthrough, making the TI emulate the MacIntosh. You might become the "Microsoft" of the TI world. This reminds us of another little anecdote, with which we'll end today's business advice.

There is a TV commercial currently running on our local channels in which a customer approaches a hot dog stand and asks for a hot dog with mustard. The vendor is heard to reply "I got ketchup and onions." The customer, astounded, says: "You sell hot dogs and you don't have mustard?" The vendor repeats, "I got ketchup and onions, ketchup and onions." The customer, resignedly, "Okay, okay. Ketchup and onions. Who knows, could be good!." Maybe your version of Lotus Symphony for the TI will be the next "Hot dog with ketchup and onions". Could be good! If that commercial can sell haircuts (yes, it's a commercial for a hair cutting salon) then anything is possible.

Next month we promise no more business advice. With a little advice from us, AT&T could be in Chapter 11. It will be back to the serious programming stuff, with a good healthy dose of source code on the side.

1.23. The Art Of Assembly — Part 23. Sorting Out Sorts

By Bruce Harrison

Sorting things into alphabetical or numerical order is of course one of those things computers do better and faster than humans. At least that's what they're supposed to do. Sometimes, when we have run sorts in Extended Basic, it has seemed that the human could have beaten the machine at this task, particularly for string variables in arrays.

There are two keys to making an effective and efficient sort. First is having a quick and memory-efficient way of determining which of two things is bigger than the other. The second is to have an efficient "algorithm" for using that information to re-order the groups of numbers or strings we're dealing with. Recently, we've done some work on sorting, mainly aimed at helping the Extended Basic programmer who's frustrated with the slow response of Extended Basic in performing sorts.

1.23.1. Use What's There

There are helping routines already built into your TI, and in some instances these can speed things along dramatically. For sorting numbers, there are two possible ways. If the numbers are all integers in the range of 0 through 65535, one can simply use the compare instruction, then be careful about whether the sign of the numbers is taken into account. To include the sign, a JLT or JGT instruction is used after the comparison, while to ignore the sign, one uses JL or JH after the compare. Let's say for example that R9 and R10 are pointing to members of an array of integers, and that we want to find out whether the one pointed by R9 is bigger than the one pointed by R10. We could proceed like this:

```
C    *R9,*R10          Compare two words
JGT  BIGGER            If R9's word is bigger, jump
```

That would take the state of the sign bit into account, so that for example >8001 would not be bigger than >7FFF, but smaller, because it has a sign bit of one. To ignore sign, we'd use this:

```
C   *R9,*R10           Compare the integers
JH   BIGGER            If R9's logically higher, jump
```

In this case, >8001 compared to >7FFF would jump to BIGGER.

Of course this is the most simple example of comparing numbers. If the numbers are floating point numbers instead of simple integers, they can still be handled simply, by using the "Floating Point Compare" routine through the XMLLNK utility vector.

To do that, one must first know the correct address data to supply to the XMLLNK vector. There lies one of the "rubs" in this business. When one is working in "pure" Assembly, to be run under the E/A module, there is the address >0A00 for FCOMP to be supplied to the XMLLNK, which itself is REF'd in the source code. If on the other hand one is working on a routine to be linked from Extended Basic, both the XMLLNK itself and the address for FCOMP are different. In that case XMLLNK is at address >2018, and the FCOMP is at address >0D3A.

Thus in pure Assembly, we'd have:

```
            REF  XMLLNK             Required reference
            BLWP @XMLLNK            Use the utility
            DATA >0A00              Data for FCOMP routine
```

But if we're linked from XB, we'd need:

```
XMLLNK      EQU  >2018              XB's XML link vector address
            BLWP @XMLLNK            Use the utility
            DATA >0D3A              Address for FCOMP routine
```

It's maddening, this kind of thing, especially if you're trying to make a program that will run in either XB or E/A environments, as we often do. The XMLLNK utility for E/A is different from the one used by XB. In E/A's version, it uses the >0A00 to access a lookup table, where it gets the address >0D3A for the FCOMP routine. Things get even more complex for some other routines accessed by XMLLNK, but this one is bad enough to illustrate the problem.

In either case, the FCOMP routine requires that one of its two numbers be located at FAC (>834A), and the other at ARG (>835C). Each of course must be eight bytes representing a floating point number in Radix 100 form. Once the comparison is done, one must examine the GPL Status byte at >837C to determine the results of the comparison. Only two bits of that byte are important, and they must be examined separately. One way of doing this is to put the Status byte into two registers, then mask off all but the bits we want. For example:

```
            BLWP @XMLLNK            Use utility
            DATA FCOMP              Floating point comparison
            MOVB @>837C,R5          Move GPL status byte to R5
            MOVB @>837C,R6          And to R6
            ANDI R5,>4000           Mask all but the "greater" bit
            JNE  BIGGER             If not zero, jump
            ANDI R6,>2000           Mask all but "equal" bit
            JNE  EQUAL              If not zero, jump
```

This code will jump to BIGGER if the number at ARG is arithmetically bigger than the number at FAC, and will jump to label EQUAL if they're equal. If both tests fail, then the number at ARG is smaller than the number at FAC.

What is done at labels BIGGER and EQUAL is of course another problem, but remember it's important to isolate the individual bits from the STATUS byte as we've shown above. In many instances you can eliminate the business of finding out if the two are equal, since that may not matter, depending how your sort proceeds from there. Often the EQUAL status can be ignored to save time and trouble.

1.23.2. Make Your Own

For strings, there is no equivalent to the FCOMP we used for floating point numbers, so one must compare byte by byte, walking through the string by incrementing pointers. Our convention is to use R9 and R10 as pointers, and to use the auto-increment option in the compare instruction. Thus to compare one byte of each string and move the pointers to the next byte, we simply:

```
CB   *R9+,*R10+          Compare one byte
```

In most cases involving strings, we don't have characters beyond 127 to worry about, so a simple JGT or JLT operation will be used as the decision making process. In today's Sidebar is a complete program that shows a string comparison.

1.23.3. What To Do About It

Once we know whether a particular string or number is bigger than another, we must decide what to do next, or how to proceed with the sorting. One could, for example, proceed with a "bubble" sort, in which each successive pair of the array is compared, and pairs are switched with one another if required. This takes many passes through the array to get it all sorted. Our preferred method, which was used in the MULTISORT routine for our Easy Data disk, is to scan through the entire array once, find either the biggest or the smallest, then re-arrange things to put that member where it belongs, and proceed to repeat this process until all members have been put where they belong. Today's Sidebar shows a complete program example which sorts an array of 75 strings in just about one and a half seconds. That's fast. Our MULTISORT routine, which interfaces with XB DATA statements, sorts 55 records of six fields each by two criteria in about 3 1/2 seconds. Try sorting that in XB. Takes forever.

In the MULTISORT routine, we were able to unload our variables into the care of XB by the NUMASG and STRASG links. If we're working in pure Assembly, that luxury is not available to us, so we need another way of handling the "where to put it" problem. Were we working in the PC, for example, we could assign a complete 64K byte segment of memory to put our sorted array in, and simply leave the original data alone in its original segment of memory.

On the TI, we don't have that luxury, so we must do something different. If we're dealing with Floating Point numbers, re-arranging them in an array is made simpler because they all have the same length (8 bytes each). With Strings, the length byte gives us the length of each string in the array, but of course each may have a different length. In the Sidebar is one "suggested" way of handling this particular problem, while making efficient use of memory.

This sort algorithm scans through all the strings in the "array", and finds the address of the least of the strings. It then stores that string in a temporary buffer (TEMSTR). Next it takes all the strings that are physically before that one in memory, and moves that entire block down by the length of the string found plus one. It's plus one to account for the length byte. Now the pointer for the start of the "unsorted" array is incremented by that same amount, and the "least" string is moved in at the head of the list. This process continues until there's only one string left, at which point we know they're all in correct order.

1.23.4. The Ordered Table

Another approach to the process, which doesn't take as much time, is to first build a table of the addresses of all the strings, then re-arrange this table instead of the strings themselves. Of course this method requires that enough memory be available for both the strings themselves and the table of addresses. That approach was used to an extent in the MULTISORT routine, where a table of addresses for the actual content of the DATA in the XB program was compiled in high memory, in the space left unused by the XB program. In a pinch, the address table can be built in VDP RAM instead of the normal expansion memory.

1.23.5. The Presort Concept

We've so far ignored the idea of where the strings or numbers you're sorting came from in the first place. In the case of MULTISORT, the data comes from the Extended Basic program. In the test program shown in the Sidebar with this article, the string data was embedded in the source code itself, so it would be instantly available once the program was finished loading. In many cases, however, the data will either be entered from the keyboard or brought in from a file on disk. In such cases, it's possible to do the sorting "on the fly" as the records are entered or read from the disk. We call this process "presort", because it sorts as the data is received, so that as soon as all the data is entered or read, the records are in correct order. We've done this for XB programmers with little subroutines that are on our Utilities Volume 3 disk.

The complete program shown in today's Sidebar can be typed in, assembled, and run as an Option 3 program under E/A. The program name is SORTEM. As we said, this will sort its 75 self-contained strings in about 1 1/2 seconds. Of course once it has run, the strings are in their proper order, so re-running will in effect not do any sorting. To perform the sort again, one must re-load the program from Option 3. This program should be easy enough to understand from the annotated source code. We think this will serve as a good example of how to perform a quick and memory-efficient sort.

We hope today's column will inspire some of our readers to take a new look at the idea of sorting data with Assembly routines. Perhaps some creative thinking on your part will improve on our methods, and make life easier for all the rest of us in the TI community.

Next month's topic is undecided at this time. We're open to suggestions.

```
* TEST SORT ROUTINE
* BY B. HARRISON
* 18 JUN 1992
* PUBLIC DOMAIN
```

```
*
       REF  VMBW,GPLLNK,KSCAN  REFERENCE UTILITY VECTORS
       DEF  SORTEM       DEFINE ENTRY POINT
*
* CODE SECTION
*
SORTEM
       LWPI WS           LOAD OUR WORKSPACE
       CLR  @COUNT       CLEAR COUNTER
       LI   R4,STSTR     GET ADDRESS OF START OF STRING ARRAY
       MOV  R4,@LOWEND   MOVE THAT TO LOW END ADDRESS STORAGE
       MOV  R4,@LEAST    AND FOR NOW INTO VARIABLE LEAST
COUNTM
       MOVB *R4+,R5      GET LENGTH BYTE OF A STRING
       SRL  R5,8         SHIFT TO RIGHT JUSTIFY IN R5
       JEQ  STSCAN       IF ZERO, WE'RE AT END OF ARRAY
       INC  @COUNT       ELSE INCREMENT COUNT
       A    R5,R4        ADD LENGTH TO ADDRESS POINTER
       JMP  COUNTM       JUMP BACK FOR NEXT STRING
STSCAN MOV  @LOWEND,R9   POINT R9 AT LOW END OF UNSORTED STRINGS
NEXT   MOV  R9,R10       MOVE THAT ADDRESS TO R10
       MOVB *R9,R7       GET THE LENGTH OF FIRST STRING IN R7
       SRL  R7,8         RIGHT JUSTIFY
       A    R7,R10       ADD TO R10
       INC  R10          INCREMENT SO R10 POINTS TO LENGTH OF NEXT STRING
MOV9   MOV  R9,R14       SAVE PRESENT R9 IN R14
       MOV  R10,R15      SAVE PRESENT R10 IN R15
       MOVB *R9+,R4      GET LENGTH FIRST STRING IN R4
       MOVB *R10+,R5     LENGTH OF SECOND IN R5
       SRL  R4,8         RIGHT JUSTIFY
       JEQ  MOVIT        IF ZERO, WE'RE THROUGH SCANNING
       SRL  R5,8         RIGHT JUSTIFY
       JEQ  MOVIT        IF ZERO, WE'RE THROUGH SCANNING
CMP910 CB   *R9+,*R10+   COMPARE THE BYTES POINTED BY R9 AND R10
       JGT  BIG          IF R9'S BIGGER, JUMP
       JLT  LESS         IF R9'S LESS, JUMP
       DEC  R4           DECREMENT COUNT
       JNE  DEC5         IF NOT ZERO, DECREMENT R5
       CI   R5,1         ELSE SEE IF R5=1
       JEQ  BIG          IF SO, STRINGS ARE EQUAL
       JMP  LESS         ELSE STRING POINTED BY R9 IS LESS
DEC5   DEC  R5           DECREMENT OTHER COUNT
       JNE  CMP910       IF NOT ZERO, COMPARE NEXT BYTE
BIG    MOV  R15,@LEAST   STASH OLD ADDRESS AT LEAST TABLE LOCATION
       MOV  R15,R9       RESET R9 TO POINT AT CURRENT LEAST STRING
       JMP  NEXT         THEN JUMP TO CONTINUE SCAN
LESS   MOV  R14,@LEAST   STASH OLD ADDRESS AT LEAST TABLE LOCATION
       MOV  R14,R9       RESET R9 TO PRESENT LEAST STRING
       MOV  R15,R10      GET OLD R10 BACK
       MOVB *R10+,R7     GET STRING LENGTH INTO R7
```

```
       SRL  R7,8          RIGHT JUSTIFY
       A    R7,R10        ADD LENGTH TO POINTER
       JMP  MOV9          THEN JUMP BACK TO TEST NEXT STRING
* THE SECTION BEGINNING AT MOVIT TAKES THE LEAST STRING FROM THIS SCAN, MOVES IT
* UP TO THE BEGINNING OF THE ARRAY
MOVIT
       MOV  @LEAST,R9     GET ADDRESS OF LEAST STRING
       LI   R10,TEMSTR    POINT AT TEMPORARY STORAGE LOCATION
       MOVB *R9,R4        GET LENGTH OF LEAST INTO R4
       SRL  R4,8          RIGHT JUSTIFY
       INC  R4            INCREMENT TO INCLUDE LENGTH BYTE ITSELF
       MOV  R4,R8         STASH IN R8
       MOV  R4,R5         AND IN R5
       C    R9,@LOWEND    IS THIS ALREADY IN THE RIGHT PLACE?
       JEQ  NOMOVE        IF SO, SKIP MOVING IT
MOVOUT MOVB *R9+,*R10+    ELSE MOVE ONE BYTE TO TEMPORARY STORAGE
       DEC  R4            DECREMENT COUNT OF BYTES
       JNE  MOVOUT        IF NOT ZERO, MOVE ANOTHER
*
* THIS NEXT SECTION MOVES ALL STRINGS ABOVE THE LEAST IN THE UNSORTED ARRAY DOWN
* TO MAKE ROOM FOR THE LEAST TO GO TO THE TOP OF THE UNSORTED PART OF THE ARRAY
*
       MOV  R9,R10        PUT ADDRESS BEYOND END OF LEAST STRING INTO R10
       MOV  @LEAST,R9     PUT LEAST ADDRESS BACK IN R9
       MOV  R9,R4         STASH THAT IN R4
       DEC  R9            POINT AT BYTE JUST BEFORE LEAST'S LENGTH
       DEC  R10           AND AT LAST BYTE OF LEAST STRING IN ARRAY
       S    @LOWEND,R4    SUBTRACT BOTTOM OF UNSORTED PORTION
MOVREV MOVB *R9,*R10      MOVE ONE BYTE UPWARDS
       DEC  R9            DECREMENT R9 POINTER
       DEC  R10           DECREMENT R10 POINTER
       DEC  R4            DECREMENT COUNT OF BYTES TO MOVE
       JNE  MOVREV        IF NOT ZERO, REPEAT
*
* AT THIS POINT ALL THE STRINGS THAT WERE ABOVE THE LEAST HAVE BEEN MOVED
* DOWNWARD IN THE ARRAY BY THE LENGTH OF THE LEAST STRING AND ITS LENGTH BYTE
*
       LI   R9,TEMSTR     POINT AT TEMPORARY STORAGE
       MOV  @LOWEND,R10   AND AT BOTTOM OF UNSORTED LIST
MOVIN  MOVB *R9+,*R10+    MOVE ONE BYTE OF LEAST STRING INTO PLACE
       DEC  R5            DECREMENT COUNT
       JNE  MOVIN         IF NOT ZERO, MOVE ANOTHER
NOMOVE A    R8,@LOWEND    ADD THE LENGTH OF LEAST TO ADDRESS POINTER
       DEC  @COUNT        DECREMENT NUMBER OF STRINGS LEFT TO SORT
       MOV  @COUNT,R4     MOVE THAT TO R4
       CI   R4,1          COMPARE TO ONE
       JGT  STSCAN        IF MORE THAN ONE LEFT, START ANOTHER SCAN
*
* IF THERE'S ONLY ONE STRING LEFT, WE'RE FINISHED SORTING, BECAUSE THAT ONE IS
* ALREADY IN THE CORRECT PLACE
```

```
*
       LI   R1,STSTR     POINT AT START OF SORTED ARRAY
       CLR  R4           CLEAR R4
SHOW   MOVB *R1+,R2      GET LENGTH BYTE INTO R2
       JEQ  EXIT         IF ZERO, ALL HAVE BEEN DISPLAYED
       SRL  R2,8         ELSE RIGHT JUSTIFY
       LI   R0,23*32+2   POINT AT ROW 24, COL 3
       BLWP @VMBW        WRITE THE STRING TO SCREEN
       CLR  @>837C       CLEAR GPL STATUS BYTE
       BLWP @GPLLNK      USE GPLLNK
       DATA >4D00        TO SCROLL SCREEN UP ONE ROW
       INC  R4           INCREMENT COUNT OF STRINGS DISPLAYED
       CI   R4,23        ARE 23 ON SCREEN?
       JLT  SHOWON       IF LESS, CONTINUE DISPLAYING
KEY    BLWP @KSCAN       ELSE SCAN THE KEYBOARD
       CB   @ANYKEY,@>837C HAS A KEY BEEN STRUCK?
       JNE  KEY          IF NOT, SCAN AGAIN
       CLR  R4           ELSE CLEAR COUNT OF DISPLAYED STRINGS
SHOWON A    R2,R1        ADD LENGTH TO POINTER IN R1
       JMP  SHOW         THEN JUMP BACK TO SHOW NEXT STRING
EXIT   BLWP @KSCAN       SCAN THE KEYBOARD
       CB   @ANYKEY,@>837C HAS A KEY BEEN STRUCK?
       JNE  EXIT         IF NOT, RE-SCAN KEYBOARD
       LWPI >83E0        ELSE LOAD GPL WORKSPACE
       B    @>6A         THEN RETURN TO E/A CONTROL
*
* DATA SECTION
*
WS     BSS  32           OUR WORKSPACE
COUNT  DATA 0            NUMBER OF STRINGS IN ARRAY
LEAST  DATA 0            ADDRESS OF LEAST STILL IN UNSORTED LIST
LOWEND DATA 0            BOTTOM OF UNSORTED LIST
TEMSTR BSS  256          TEMPORARY STORAGE FOR STRING TO BE PLACED
ANYKEY BYTE >20          BYTE FOR KEYSTROKE DETECTION
*
* FROM HERE ON ARE THE STRINGS TO BE SORTED
* THERE MUST BE A ZERO BYTE AT END TO INDICATE END OF ARRAY
* NULL STRINGS MUST NOT BE INCLUDED IN THE LIST
*
STSTR  BYTE 6            LENGTH OF FIRST STRING
       TEXT 'ZEBRAS'     CONTENT
       BYTE 9
       TEXT 'AARDVARKS'
       BYTE 7
       TEXT 'ANIMALS'
       BYTE 9
       TEXT 'YESTERDAY'
       BYTE 9
       TEXT 'COMPUTERS'
       BYTE 7
```

```
TEXT 'WINDAGE'
BYTE 14
TEXT 'DRIVING SCHOOL'
BYTE 11
TEXT 'DAILY PAPER'
BYTE 12
TEXT 'COMPUTATIONS'
BYTE 7
TEXT 'FRIENDS'
BYTE 9
TEXT 'XYLOPHONE'
BYTE 7
TEXT 'MONITOR'
BYTE 10
TEXT 'TELEVISION'
BYTE 6
TEXT 'FROSTY'
BYTE 10
TEXT 'MICROPHONE'
BYTE 6
TEXT 'K-MART'
BYTE 8
TEXT 'CIRCULAR'
BYTE 8
TEXT 'SYMPHONY'
BYTE 8
TEXT 'CONCERTO'
BYTE 8
TEXT 'ASSEMBLY'
BYTE 9
TEXT 'ANTIPHONS'
BYTE 8
TEXT 'CALOMINE'
BYTE 12
TEXT 'FRIENDLINESS'
BYTE 10
TEXT 'SATURATION'
BYTE 10
TEXT 'BRIGHTNESS'
BYTE 7
TEXT 'ANTENNA'
BYTE 7
TEXT 'COLUMNS'
BYTE 5
TEXT 'ROWED'
BYTE 9
TEXT 'BEAUTIFUL'
BYTE 8
TEXT 'UNICORNS'
BYTE 7
```

```
TEXT 'JAGUARS'
BYTE 7
TEXT 'JOINTLY'
BYTE 6
TEXT 'MARCEL'
BYTE 8
TEXT 'JEAN-GUY'
BYTE 9
TEXT 'POTHOLDER'
BYTE 6
TEXT 'VERITY'
BYTE 9
TEXT 'VERITABLE'
BYTE 8
TEXT 'DREADFUL'
BYTE 7
TEXT 'WEAVING'
BYTE 8
TEXT 'SCRUPLES'
BYTE 8
TEXT 'OPTIONAL'
BYTE 6
TEXT 'XANADU'
BYTE 9
TEXT 'INTENSIVE'
BYTE 9
TEXT 'WATERGATE'
BYTE 6
TEXT 'MOVIES'
BYTE 6
TEXT 'BABIES'
BYTE 8
TEXT 'CHILDREN'
BYTE 8
TEXT 'CHILDISH'
BYTE 10
TEXT 'PERCENTAGE'
BYTE 8
TEXT 'RELIABLE'
BYTE 9
TEXT 'CAPRICORN'
BYTE 12
TEXT 'MOBILIZATION'
BYTE 9
TEXT 'ANTIQUITY'
BYTE 10
TEXT 'BARBARIANS'
BYTE 8
TEXT 'SANDOVAL'
BYTE 9
```

```
       TEXT 'SAN DIEGO'
       BYTE 9
       TEXT 'OBLIVIOUS'
       BYTE 7
       TEXT 'ANCIENT'
       BYTE 8
       TEXT 'POLYGONS'
       BYTE 10
       TEXT 'MARGARITAS'
       BYTE 9
       TEXT 'MINISTERS'
       BYTE 8
       TEXT 'RESEARCH'
       BYTE 9
       TEXT 'POLYANDRY'
       BYTE 9
       TEXT 'MARVELOUS'
       BYTE 11
       TEXT 'GRANDMOTHER'
       BYTE 9
       TEXT 'GREATNESS'
       BYTE 12
       TEXT 'FAITHFULNESS'
       BYTE 8
       TEXT 'MIRTHFUL'
       BYTE 9
       TEXT 'MERRIMENT'
       BYTE 9
       TEXT 'NATURALLY'
       BYTE 11
       TEXT 'OZONE LAYER'
       BYTE 11
       TEXT 'MASTERPIECE'
       BYTE 10
       TEXT 'DEDICATION'
       BYTE 11
       TEXT 'NONETHELESS'
       BYTE 9
       TEXT 'SUITCASES'
ENDSTR BYTE 0            ZERO LENGTH BYTE IS END OF ARRAY
       END
```

1.24. The Art Of Assembly — Part 24. Another Sort Of Sort

By Bruce Harrison

Last month we showed an entire program to sort a self-contained list of 75 strings. Toward the end of that column, we introduced the idea that we call "Pre-sort" for sorting things as they are received from a source. It occurred to us that this left another "shoe to drop" on the subject, so today we're dropping that other shoe.

In the Sidebar is another complete E/A Option 3 program that you can type in, assemble, and run for yourself. It will prompt for an input file name, then will take the named input file, treating each record as a string, and sort the entire file as received. When it's done, it will prompt for an output file name, and given one it will save the sorted contents of the input file to this output file.

The sort process is fairly simple. As each record is gotten from the file, it will be compared to each record already in memory. When a record is found that's bigger than the incoming record, all of the records currently in the array from there to the end are moved downward by one more than the length of the incoming record, then the incoming record is slipped into its proper place in the array. A zero byte is placed just after the last string in the array to mark the end. If no "bigger" string is found in the array, the incoming one is placed at the end of the array.

We have of course tested this program extensively, and it seems to work exactly as advertised. Error traps are built in to protect the user from errors of almost any kind, including trying to sort a file that's too big for the 24K of high memory. We'll get into what happens on errors in just a bit, but first let's discuss the performance of the program.

1.24.1. Performance

The first test we performed was to take a file made from the 75 strings that were in last month's program. From a normal floppy disk drive, it takes about six to seven seconds to sort that list of words, including the time to open the file. The sort produces an output file that has all 75 of those records in alphabetical order.

Having succeeded with that short file, we decided to try something bigger. We took the D/V 80 file of one of these columns (Number 3) which has 231 records, and occupies 65 sectors on a DS/SD disk. This took about 1 minute to read and sort. The result was a rather interesting output file. All the blank lines (Blank lines are records consisting of a single space.) came first. These were followed by a group of lines indented by 26 spaces, then those indented by less than 26 spaces, then those that had the five character indent for paragraph beginnings, and finally a large group of lines which had no indent.

Within each group, the sorting was done as you'd expect. Lines that had characters like parentheses or numbers came first, then those starting with capital letters, then those starting with lower case letters. In other words, the sort worked exactly as it should. One minute didn't seem like much time to read and sort this file. For comparison purposes, we went into the E/A editor and loaded the same file from the floppy disk. It took about 28 seconds to simply load in for editing, or nearly half the time our program took to load and sort the file. Doing this sort operation on a file from Ramdisk takes much less time.

1.24.2. The Error Reports

There are two major kinds of errors that will be trapped by this program. First of course are file errors. These are reported to the user on rows 23 and 24 of the screen. After the error report, pressing any key takes the user back to the prompt for the appropriate file name. If an error occurs on the input file, you return to that prompt to try again. If an error occurs on the output file, you return to that file name input field. The sorted file in such cases is still in memory, so you get a second try at saving the sorted file to disk.

The other possible error is running out of memory. As the program builds the array of sorted strings in high memory, it checks with each added string for the end of usable memory at >FFE7. If that would be exceeded, the error is reported, and after a keypress you're at the output file name prompt so you can save what's in memory. In general, any D/V 80 file that's small enough to be edited by E/A's editor will not run this program out of memory. The capacity we allow is 24,550 bytes. This is because the program itself is all in low memory, so that all the high memory from >A000 through >FFE6 can be used for the string array.

1.24.3. User Guidance

The user interface in this program is very simple. At the file name prompts, function key presses may be used to delete (**FCTN 1**) or toggle into insert (**FCTN 2**), or move the cursor left and right (**FCTN S** or **FCTN D**). Function-9 is reserved for two purposes. If you're at the output name prompt, **FCTN 9** takes you back to the input name prompt. If you're at the input name prompt, **FCTN 9** takes you out of the program. When a sort has finished being saved to disk, the program returns to the input file name prompt, so you can sort a series of files without exiting the program, if you wish.

1.24.4. Embellishments

Since we've provided all the source code in today's Sidebar, you can add some "touches" of your own to this rather primitive program. For example, just before the BL @CRSIN lines, you could put in a "beep" sound via GPLLNK, and you could add a "boop" sound in the error traps just before the BL @KEYLOO lines. You could also expand the input file name fields to allow for hard disk path names.

1.24.5. Program Construction

Those who've followed this column will see a lot of familiar stuff in the Sidebar. There's CRSIN from number 5, for example. That was lifted "as is" for use here. The error trapping for file operations came right out of number 8, and other subroutines were "imported" from other sources. The opening, reading, writing and closing of files came from number 9. We mention all this just to show that this is something you too can do, taking old subroutines from previous programs or from our "Sidebars" and re-cycling them to make a new program. Thus the bulk of this "new" program is re-cycled subroutines, not newly written source code. Doing it this way allowed this whole program to be put together and tested in a single afternoon. This column to accompany the source code took only a couple of hours the next morning. (Hope John and Laura don't notice this last.)

We keep a "battery" of such routines here on dusty old floppy disks, and in some cases in multiple versions, so that routines for linkage from Extended Basic, which are slightly different, co-exist with those written for E/A use. If we had time, we'd make some kind of index of all those floppy disks, so a particular routine would be easier to find, but it seems we can always find time to search through all the disks, but never find the time to index them. It reminds me of another expression from my days in Civil Service, concerning "rush" projects: "We never have enough time to do it right, but we always have time to do it over."

We hope you'll find today's program useful. It's written rather crudely, but is completely functional. You can make changes in it to your heart's content, for different kinds of files, and so on. You could also adapt it to work from Extended Basic, or make it an "Option 5" program file. Next month we promise we'll write about something other than sorts. Two columns in succession on that topic is enough.

```
* FILE SORTING PROGRAM
* BY B. HARRISON
* 22 JUN 1992
* PUBLIC DOMAIN
*
* REQUIRED REFERENCE VECTORS
*
       REF  VMBW,VMBR,VSBW,VSBR
       REF  DSRLNK,KSCAN
*
* REQUIRED EQUATES
*
STATUS EQU  >837C
WS     EQU  >20BA
GPLWS  EQU  >83E0
SCRWID EQU  32
PAB1   EQU  >1000
BUF    EQU  >1050
PABPNT EQU  >8356
KEYADR EQU  >8374
KEYVAL EQU  >8375
*
       DEF  START        DEFINE ENTRY POINT
```

```
*
       AORG >2678        SET ORIGIN IN LOW MEMORY
*
* CODE SECTION - MAIN PROGRAM
*
START
       LWPI WS           LOAD USER WORKSPACE
       LI   R15,RTNSTK   SET STACK FOR HIGH LEVEL SUBROUTINE
       LI   R0,3         ROW 1, COLUMN 4
       LI   R4,3*SCRWID  THREE ROWS TO CLEAR
       BL   @CLRFLD      CLEAR THEM
       LI   R1,INFSTR    SET FOR INPUT FILE PROMPT
       BL   @DISSTR      DISPLAY PROMPT
       AI   R0,SCRWID    DOWN TO ROW 2
       LI   R4,15        15 BYTES
       BL   @CRSIN       USE CRSIN SUBROUTINE
       CI   R8,15        HAS F-9 BEEN STRUCK?
       JNE  PLCFN        IF NOT, GO ON
       B    @EXIT        ELSE EXIT PROGRAM
PLCFN
       LI   R9,TEMSTR    POINT AT TEMPORARY STRING
       LI   R10,PAB1DT+9 POINT R10 AT FILE DESCRIPTOR LENGTH BYTE
       BL   @MOVSTR      MOVE STRING FROM TEMSTR TO PAB DATA
       LI   R4,>A000     SET R4 TO >A000
       MOV  R4,@ENDSTR   MOVE THAT TO END OF ARRAY LOCATION
       CLR  *R4          MAKE >A000 EQUAL 0
OPNF1
       LI   R0,2*SCRWID+2 SET FOR ROW 3, COL 3
       LI   R1,SFSTR     "SORTING FILE" MESSAGE
       BL   @DISSTR      DISPLAY THAT
       MOVB @INMD,@PAB1DT+1 OPEN WILL BE INPUT MODE
       LI   R0,PAB1      SET WRITE ADDRESS IN R0
       MOVB @PAB1DT+9,R2 GET DESCRIPTOR LENGTH BYTE INTO LEFT BYTE R2
       SRL  R2,8         RIGHT JUSTIFY SO R2 IS A WORD OF LENGTH
       AI   R2,10        ADD 10 TO INCLUDE THE PAB1DT LINE PLUS DESCRIPTOR
       LI   R1,PAB1DT    POINT R1 AT PAB DATA
       BLWP @VMBW        WRITE BYTES TO PAB LOCATION IN VDP RAM
       AI   R0,9         ADD NINE TO ADDRESS IN R0
       MOV  R0,@PABPNT   PLACE THAT ADDRESS AT >8356
       CLR  @STATUS      CLEAR GPL STATUS
       BLWP @DSRLNK      USE DSRLNK UTILITY
       DATA 8            REQUIRED DATA
       STST R14          STORE STATUS REGISTER IN R14
       ANDI R14,>2000    MASK ALL BUT BIT #2 IN R14
       JEQ  RDF1         IF ZERO, GO AHEAD TO READ FILE
       BL   @OPNERR      ELSE TO OPNERR
       B    @START       THEN BACK TO START
RDF1
       MOVB @READF,R1    MOVE READ OPCODE INTO LEFT BYTE R1
       LI   R0,PAB1      PAB ADDRESS IN VDP
```

```
        BLWP @VSBW          WRITE ONE BYTE INTO PAB
        AI   R0,9           ADD NINE
        MOV  R0,@PABPNT     MOVE TO >8356
        CLR  @STATUS        CLEAR GPL STATUS
        BLWP @DSRLNK        USE DSRLNK
        DATA 8              REQUIRED DATA
        LI   R0,PAB1+1      SET TO SECOND BYTE OF PAB IN VDP
        BLWP @VSBR          READ INTO LEFT BYTE R1
        SRL  R1,13          SHIFT R1 RIGHT BY 13 BITS
        JEQ  READON         IF ZERO, NO ERROR IN DSR OPERATION
        CI   R1,5           IF ERROR = 5, END OF FILE HAS BEEN REACHED
        JEQ  CLSF1          IF SO, CLOSE THE FILE
        BL   @FILERR        ELSE SOME OTHER ERROR, REPORT THAT TO USER
        B    @START         BACK TO START
READON  LI   R0,PAB1+5      POINT AT PAB+5 IN VDP RAM
        BLWP @VSBR          READ THAT BYTE INTO LEFT BYTE R1
        MOVB R1,R2          MOVE BYTE INTO R2
        SRL  R2,8           RIGHT JUSTIFY LENGTH IN R2
        MOVB R1,@TEMSTR     MOVE BYTE TO TEMSTR
        LI   R0,BUF         POINT TO BUFFER LOCATION IN VDP
        LI   R1,TEMSTR+1    CONTENT GOES TO TEMSTR+1
        BLWP @VMBR          READ CONTENT OF RECORD FROM VDP BUFFER
        MOV  R2,R8          STASH STRING LENGTH IN R8
        INC  R8             INC TO INCLUDE LENGTH BYTE
        LI   R10,>A000      POINT AT START OF ARRAY MEMORY
CMPSTR
        LI   R9,TEMSTR      POINT AT INCOMING STRING
        MOV  R10,R14        SAVE R10 ADDRESS IN R14
        C    R10,@ENDSTR    COMPARE R10 TO END OF ARRAY
        JLT  CMPON          IF LESS, PROCEED WITH COMPARISON
        JMP  NOSORT         ELSE NO SORT NECESSARY
CMPON   MOVB *R9+,R4        GET INCOMING STRING LENGTH IN R4
        MOVB *R10+,R5       GET AN ARRAY STRING'S LENGTH
        SRL  R4,8           RIGHT JUSTIFY R4
        SRL  R5,8           AND R5
CMP910  CB   *R9+,*R10+     COMPARE ONE BYTE
        JGT  BIG            IF R9'S IS GREATER, JUMP
        JLT  SMALL          IF R9'S IS LESS, JUMP
        DEC  R4             ELSE DECREMENT COUNT
        JEQ  SMALL          IF ZERO, R9'S STRING IS LESS
        CI   R5,1           COMPARE R5 TO 1
        JNE  DEC5           IF NOT EQUAL, JUMP
        JMP  BIG            ELSE R9'S STRING
DEC5    DEC  R5             DECREMENT OTHER COUNT
        JNE  CMP910         IF NOT ZERO, COMPARE ANOTHER
BIG     MOV  R14,R10        GET ORIGINAL ADDRESS BACK IN R10
        MOVB *R10+,R7       TAKE LENGTH BYTE INTO R7
        SRL  R7,8           RIGHT JUSTIFY
        A    R7,R10         ADD LENGTH TO R10
        JMP  CMPSTR         THEN COMPARE TO NEXT STRING
```

```
SMALL
       MOV  @ENDSTR,R10  POINT AT END OF ARRAY
       MOV  R10,R9       POINT R9 AT PRESENT END
       MOV  R10,R4       MOVE PRESENT END TO R4
       S    R14,R4       SUBTRACT START OF HIGH STRING
       A    R8,R10       ADD LENGTH OF STRING TO BE ADDED
       CI   R10,>FFE7    ARE WE AT END OF MEMORY
       JLT  DEC9         IF NOT, PROCEED
       LI   R0,22*SCRWID+2 ELSE SET FOR ROW 23, COL 3
       LI   R1,OOMSTR    "OUT OF MEMORY"
       BL   @DISSTR      DISPLAY THAT
       BL   @KEYLOO      WAIT FOR KEYSTROKE
       JMP  GETOFN       THEN MOVE ON
DEC9   DEC  R9           DECREMENT R9
       DEC  R10          AND R10
MOVREV MOVB *R9,*R10     MOVE ONE BYTE
       DEC  R9           DECRMENT POINTER
       DEC  R10          AND OTHER POINTER
       DEC  R4           DECREMENT BYTE COUNT
       JNE  MOVREV       IF NOT ZERO, REPEAT
       JMP  MOVIN        ELSE JUMP AHEAD
NOSORT MOV  @ENDSTR,R14  MOVE END OF ARRAY ADDRESS INTO R14
MOVIN  LI   R9,TEMSTR    POINT AT INCOMING STRING
       MOV  R14,R10      DESTINATION ADDRESS IN R10
       BL   @MOVSTR      MOVE STRING INTO ARRAY
       A    R8,@ENDSTR   ADD LENGTH TO END ADDRESS
       MOV  @ENDSTR,R4   MOVE THAT TO R4
       MOVB @ONE,*R4     PUT A ZERO BYTE THERE
       B    @RDF1        THEN READ NEXT RECORD
*
CLSF1  BL   @CLSF2       USE SUBROUTINE
*
GETOFN
       LI   R0,3         SET TO ROW 1, COL 3
       LI   R4,3*SCRWID  THREE ROWS
       BL   @CLRFLD      CLEAR
       LI   R1,OUTSTR    "OUTPUT NAME" PROMPT
       BL   @DISSTR      DISPLAY
       AI   R0,SCRWID    DOWN ONE ROW
       LI   R4,15        15 BYTES
       BL   @CRSIN       GET NAME
       CI   R8,15        F-9 STRUCK?
       JNE  GETOF1       IF NOT, MOVE ON
       B    @START       ELSE BACK TO START
GETOF1 LI   R9,TEMSTR    POINT AT TEMPORARY STRING
       LI   R10,PAB1DT+9 AND FILE NAME LOCATION
       BL   @MOVSTR      MOVE THE STRING INTO PLACE
*
OPNF2
       MOVB @OUTMD,@PAB1DT+1 OPEN WILL BE OUTPUT MODE
```

```
        LI   R0,PAB1       SET WRITE ADDRESS IN R0
        MOVB @PAB1DT+9,R2  GET DESCRIPTOR LENGTH BYTE INTO LEFT BYTE R2
        SRL  R2,8          RIGHT JUSTIFY SO R2 IS A WORD OF LENGTH
        AI   R2,10         ADD 10 TO INCLUDE THE PAB1DT LINE PLUS DESCRIPTOR
        LI   R1,PAB1DT     POINT R1 AT PAB DATA
        BLWP @VMBW         WRITE BYTES TO PAB LOCATION IN VDP RAM
        AI   R0,9          ADD NINE TO ADDRESS IN R0
        MOV  R0,@PABPNT    PLACE THAT ADDRESS AT >8356
        CLR  @STATUS       CLEAR GPL STATUS
        BLWP @DSRLNK       USE DSRLNK UTILITY
        DATA 8             REQUIRED DATA
        STST R14           STORE STATUS REGISTER IN R14
        ANDI R14,>2000     MASK ALL BUT BIT #2 IN R14
        JEQ  WRTF2         IF ZERO, GO AHEAD TO WRITE FILE
        BL   @OPNERR       ELSE TO OPNERR
        JMP  GETOFN        THEN JUMP BACK
WRTF2
        LI   R9,>A000      POINT AT START OF ARRAY
WRTNXT  LI   R10,TEMSTR    POINT R10 AT TEMSTR
        BL   @MOVSTR       MOVE THE STRING
        MOVB @TEMSTR,R1    GET LENGTH OF RECORD IN LEFT BYTE R1
        JEQ  CLSFO         IF ZERO LENGTH, WE ARE AT END OF ARRAY
        LI   R0,PAB1+5     POINT TO RECORD LENGTH BYTE OF PAB
        BLWP @VSBW         WRITE LENGTH TO PAB
        MOVB R1,R2         PLACE LENGTH IN LEFT BYTE R2
        SRL  R2,8          RIGHT JUSTIFY LENGTH IN R2
        LI   R1,TEMSTR+1   POINT TO STRING CONTENT
        LI   R0,BUF        POINT AT BUFFER IN VDP
        BLWP @VMBW         WRITE RECORD CONTENTS TO VDP
        MOVB @WRITEF,R1    GET WRITE OPCODE IN R1
        LI   R0,PAB1       POINT TO START OF PAB
        BLWP @VSBW         WRITE THE OPCODE BYTE TO VDP
        AI   R0,9          ADD 9
        MOV  R0,@PABPNT    MOVE TO >8356

        CLR  @STATUS       CLEAR GPL STATUS BYTE
        BLWP @DSRLNK       CALL DSR LINKAGE
        DATA 8             REQUIRED DATA
        LI   R0,PAB1+1     POINT TO SECOND BYTE OF PAB
        BLWP @VSBR         READ THAT BYTE INTO R1
        SRL  R1,13         SHIFT R1 RIGHT 13 BITS
        JEQ  WRTNXT        IF ZERO, NO ERROR, SO GO ON
        BL   @FILERR       ELSE BRANCH TO ERROR HANDLING
        BL   @CLSF2        CLOSE FILE
        B    @GETOFN       THEN BRANCH
CLSFO   BL   @CLSF2        CLOSE THE FILE
        B    @START        THEN BACK TO START
EXIT    LWPI GPLWS         LOAD THE GPL WORKSPACE
        B    @>6A          THEN BACK TO E/A
*
```

```
* CODE SECTION - SUBROUTINES
*
OPNERR
       MOV  R11,*R15+    STACK R11
       LI   R0,22*32+2   ROW 23, COL 3
       LI   R1,FNOMSG    FILE NOT OPENED
       BL   @DISSTR      DISPLAY THAT
       LI   R0,PAB1+1    POINT AT PAB + 1 IN VDP
       BLWP @VSBR        READ THAT BYTE
       SRL  R1,13        SHIFT RIGHT 13 BYTES
       JMP  FILER1       THEN JUMP
FILERR MOV  R11,*R15+    STACK R11
FILER1 SLA  R1,1         DOUBLE NUMBER IN R1
       AI   R1,LUT       ADD LOOKUP TABLE ADDRESS
       MOV  *R1,R1       GET ERROR MESSAGE ADDRESS INTO R1
       LI   R0,23*32+2   POINT AT ROW 24, COL 3
       BL   @DISSTR      DISPLAY ERROR MESSAGE
       BL   @KEYLOO      STOP AT KEY LOOP
       LI   R0,22*SCRWID LOAD R0 FOR ROW 23
       LI   R4,2*SCRWID  TWO ROWS
       BL   @CLRFLD      CLEAR THOSE
       B    @SUBRET      THEN RETURN
*
DISSTR  MOVB *R1+,R2     GET LENGTH BYTE
        SRL  R2,8        RIGHT JUSTIFY
        JEQ  DISX        IF ZERO LENGTH, SKIP IT
        BLWP @VMBW       ELSE WRITE STRING TO SCREEN
DISX    RT               RETURN TO CALLER
*
CLSF2  LI   R0,PAB1      POINT TO PAB ADDRESS
       MOVB @CLOSEF,R1   GET CLOSE OPCODE IN LEFT BYTE R1
       BLWP @VSBW        WRITE OPCODE TO PAB
       AI   R0,9         ADD NINE
       MOV  R0,@PABPNT   PLACE AT >8356
       CLR  @STATUS      CLEAR STATUS
       BLWP @DSRLNK      CALL DSRLNK
       DATA 8            REQUIRED DATA
       RT                RETURN
*
*
CRSIN
       MOV  R11,*R15+    STACK RETURN ADDRESS
       CLR  @INSFLG      CLEAR OUR INSERT FLAG
       MOV  R0,@PGNUM    STASH R0 IN MEMORY LOCATION
       DEC  R0           DECREMENT R0
       MOVB @EDGE,R1     PLACE EDGE CHARACTER IN LEFT BYTE R1
       BLWP @VSBW        WRITE EDGE CHARACTER TO SCREEN
       INC  R0           RESET R0 TO ORIGINAL VALUE
       A    R4,R0        ADD NUMBER OF CHARACTERS TO ACCEPT
       BLWP @VSBW        WRITE AN EDGE CHARACTER TO SPOT BEYOND FIELD
```

```
       MOV  R0,@ENDOC    SAVE THIS LOCATION IN MEMORY
       S    R4,R0        RESET R0 TO ORIGINAL VALUE
       MOV  R4,@SAV4     STASH R4 IN MEMORY
CRSI0A BLWP @VSBR        READ THE CHARACTER POINTED TO BY R0
       MOVB R1,@ALTKEY   STASH THAT CHARACTER AT LOCATION ALTKEY
CRSI0  BL   @CURFRC      FORCE THE CURSOR ONTO THE SCREEN
       BL   @KI2         USE THE SCANNING SUBROUTINE WITH FLASHING CURSOR
       CI   R8,9         HAS RIGHT ARROW BEEN STRUCK?
       JEQ  CRSRT        IF SO, JUMP
       CI   R8,8         HAS LEFT ARROW BEEN STRUCK?
       JEQ  CRSBK        IF SO, JUMP
       CI   R8,10        DOWN ARROW?
       JLT  CRSC4        IF LESS, JUMP
       CI   R8,15        HAS FCTN-9 BEEN STRUCK?
       JEQ  CRSDMY       IF SO, JUMP
       CI   R8,13        HAS ENTER KEY BEEN STRUCK?
       JLT  CRSDMY       IF LESS, JUMP
CRSC4  CI   R8,4         HAS FCTN-2 (INSERT) BEEN STRUCK?
       JNE  CRSENT       IF NOT, JUMP
       INC  @INSFLG      ELSE SET INSERT FLAG
       JMP  CRSI0        THEN JUMP BACK
CRSENT CB   @KEYVAL,@ENTERV HAS ENTER BEEN STRUCK?
       JEQ  CRSDMY       IF SO, JUMP
       CI   R8,3         HAS FCTN-1 (DELETE) BEEN STRUCK?
       JEQ  CRSDEL       IF SO, JUMP
       CI   R8,32        SPACE BAR
       JLT  CRSI0        IF LESS, JUMP
* THE FOLLOWING FIVE LINES ARE NEEDED ONLY IF ONE WANTS LOWER CASE
* CHARACTERS CONVERTED TO UPPER CASE.  IF NOT, OMIT THESE FIVE LINES
       CI   R8,122       COMPARE TO LOWER CASE Z
       JGT  CRSI0        IF GREATER, JUMP
       CI   R8,97        COMPARE TO LOWER CASE A
       JLT  CRSI1        IF LOWER, JUMP
       SB   @ANYKEY,@KEYVAL ELSE SUBTRACT >20 FROM KEYSTROKE
CRSI1
       MOV  @INSFLG,R1   TEST IF INSERT FLAG ON
       JEQ  CRSI1A       IF NOT, JUMP
       MOVB @ALTKEY,R1   ELSE WRITE CURRENT CHARACTER
       BLWP @VSBW        TO CURRENT SCREEN POSITION
       MOV  @ENDOC,R2    MOVE LIMIT ADDRESS INTO R2
       S    R0,R2        SUBTRACT CURRENT R0 POSITION
       LI   R1,TEMSTR    POINT TO TEMSTR LOCATION
       BLWP @VMBR        READ CHARACTERS FROM SCREEN
       DEC  R2           DECREMENT CHARACTER COUNT
       JEQ  CRSI1A       IF R2 IS ZERO, NO INSERT - WE'RE AT LAST POSITION
       INC  R0           INCREMENT SCREEN POSITION
       BLWP @VMBW        WRITE CHARACTERS BACK
       DEC  R0           POINT BACK ONE SPOT
CRSI1A MOVB @KEYVAL,R1   MOVE THE KEY STRUCK INTO LEFT BYTE R1
       BLWP @VSBW        WRITE KEY VALUE TO SCREEN
```

```
        INC  R0           POINT AT NEXT CHARACTER POSITION
        BLWP @VSBR        READ CHARACTER THAT'S THERE
        CB   R1,@EDGE     IS THIS AN EDGE CHARACTER?
        JNE  CRSI0A       IF NOT, JUMP
        DEC  R0           ELSE BACK UP ONE CHARACTER
        JMP  CRSI0A       THEN BACK FOR ANOTHER KEY INPUT
CRSRT   MOVB @ALTKEY,R1   TAKE CURRENT SCREEN CHARACTER INTO LEFT BYTE R1
        BLWP @VSBW        WRITE CHARACTER TO SCREEN
        CLR  @INSFLG      CLEAR THE INSERT FLAG
        INC  R0           MOVE TO NEXT SPOT
        BLWP @VSBR        READ THE CHARACTER THERE
        CB   R1,@EDGE     IS THAT EDGE CHARACTER?
        JEQ  CRSRT1       IF SO, JUMP
        MOVB R1,@ALTKEY   ELSE STASH CURRENT SCREEN CHARACTER
        BL   @CURFRC      FORCE CURSOR ONTO SCREEN
        BL   @KI2A        GO SCAN KEYBOARD
        CB   @KEYVAL,@RITEV IS RIGHT ARROW STILL HELD DOWN?
        JEQ  CRSRT        IF SO, KEEP GOING RIGHT
        CB   @KEYVAL,@NOKEY HAS NO KEY BEEN STRUCK?
        JEQ  CRSRT2       IF SO, JUMP
CRSRT1  DEC  R0           BACK TO PREVIOUS SPOT
CRSRT2  MOVB @ONOFF,@KI2A+2 RESTORE DELAY CONSTANT
        MOVB @ALTKEY,R1   GET CHARACTER INTO LEFT BYTE R1
        BLWP @VSBW        WRITE TO SCREEN
        JMP  CRSI0        THEN JUMP BACK FOR ANOTHER KEY
CRSBK   MOVB @ALTKEY,R1   GET CURRENT CHARACTER IN R1
        BLWP @VSBW        WRITE TO SCREEN
        CLR  @INSFLG      CLEAR INSERT FLAG
        DEC  R0           BACK ONE SPOT
        BLWP @VSBR        READ CHARACTER FROM SCREEN
        CB   R1,@EDGE     IS THAT EDGE CHARACTER?
        JEQ  CRSBK1       IF SO, JUMP
        MOVB R1,@ALTKEY   ELSE STASH CHARACTER AT ALTKEY
        BL   @CURFRC      FORCE CURSOR ONTO SCREEN
        BL   @KI2A        GO GET KEYSTROKE
        CB   @KEYVAL,@LEFTV  IS LEFT ARROW STILL HELD DOWN?
        JEQ  CRSBK        IF SO, GO BACK AGAIN
        CB   @KEYVAL,@NOKEY HAS NO KEY BEEN STRUCK
        JEQ  CRSRT2       IF SO, JUMP
CRSBK1  INC  R0           MOVE TO NEXT SPOT
        JMP  CRSRT2       THEN JUMP
CRSDMY  JMP  CRSIX        THIS IS A DUMMY JUMP TO KEEP JUMPS IN RANGE
CRSDEL  MOV  R0,R7        STASH R0 IN R7
        CLR  @INSFLG      CLEAR INSERT FLAG, SINCE WE'RE DELETING
        MOV  @ENDOC,R2    END OF FIELD ADDRESS IN R2
        S    R0,R2        SUBTRACT CURRENT CHARACTER ADDRESS
        INC  R0           POINT TO NEXT CHARACTER
        DEC  R2           DECREMENT R2 COUNT
        JEQ  CRSD1        IF R2 ZERO, PRINT SPACE - WERE AT LAST POSITION
        LI   R1,TEMSTR    POINT R1 AT TEMSTR FOR TEMPORARY STORAGE
```

```
       BLWP @VMBR         READ CHARACTERS INTO LOCATION TEMSTR
       MOV  R7,R0         PUT BACK R0
       BLWP @VMBW         WRITE CHARACTERS FROM TEMSTR TO SCREEN
CRSD1  MOVB @ANYKEY,R1    PUT A SPACE IN LEFT BYTE R1
       MOV  @ENDOC,R0     GET LIMIT SPOT INTO R0
       DEC  R0            DECREMENT BY ONE
       BLWP @VSBW         WRITE A SPACE TO SPOT JUST BEFORE LIMIT
       MOV  R7,R0         GET R0 BACK AGAIN
CRSD0  B    @CRSI0A       BRANCH BACK TO BEGINNING
CRSIX  MOVB @ALTKEY,R1    MOVE CURRENT CHARACTER TO R1
       BLWP @VSBW         WRITE TO SCREEN
       MOV  @ENDOC,R0     SET LIMIT POSITION IN R0
       DEC  R0            DECREMENT BY ONE
       MOV  @SAV4,R2      MOVE MAX NUMBER OF CHARACTERS INTO R2
CRSIX1 BLWP @VSBR         READ THE CHARACTER AT CURRENT R0 POSITION
       CB   R1,@ANYKEY    IS THAT A SPACE?
       JNE  CRSIXX        IF NOT, WE'VE REACHED CONTENT OF STRING
       DEC  R0            ELSE MOVE BACK ONE SPOT
       DEC  R2            DECREASE CHARACTER COUNT BY ONE
       JGT  CRSIX1        IF GREATER THAN ZERO, JUMP BACK
CRSIXX MOV  @PGNUM,R0     GET ORIGINAL R0 POSITION BACK
       SWPB R2            PUT CHARACTER COUNT IN LEFT BYTE R2
       MOVB R2,@TEMSTR    PLACE THAT AT TEMSTR
       SWPB R2            REVERSE R2 AGAIN
       JEQ  CRIX          IF R2=0, JUMP
       LI   R1,TEMSTR+1   ELSE SET R1 TO POINT TO STRING CONTENT STORAGE
CRSIX2 BLWP @VMBR         READ THE STRING FROM THE SCREEN
CRIX
SUBRET DECT R15           DECREMENT STACK POINTER BY TWO
       MOV  *R15,R11      GET RETURN ADDRESS BACK
       RT                 RETURN
*
*
KI2    CLR  @STATUS       KEY-IN WITH ALTERNATING
       BLWP @KSCAN        CHARACTER AND CURSOR
       LIMI 2             ACTIVATE INTERRUPTS
       LIMI 0             SHUT OFF INTERRUPTS
       DEC  R4            ENTER AFTER R4 SET TO >0200
       JEQ  CHNG          AND R1 TO >1E00 AND VSBW
       CB   @ANYKEY,@STATUS HAS A KEY BEEN STRUCK?
       JNE  KI2           IF NOT, RE-SCAN KEYBOARD
       MOV  @KEYADR,R8    ELSE PUT KEY'S VALUE IN R8
       RT                 THEN RETURN
CHNG   CI   R1,>1E00      IS R1 SET TO CURSOR CHARACTER?
       JEQ  L1            IF SO, JUMP
       LI   R1,>1E00      ELSE SET LEFT BYTE R1 TO CURSOR
       BLWP @VSBW         WRITE CURSOR TO SCREEN
       MOVB @ONOFF,R4     PLACE TIMING IN LEFT BYTE R4
       JMP  KI2           GO BACK TO SCANNING KEYBOARD
L1     MOVB @ALTKEY,R1    PLACE ALTERNATING CHARACTER IN LEFT BYTE R1
```

```
       MOVB @ONOFF+1,R4  PLACE ALTERNATE DELAY IN LEFT BYTE R4
       BLWP @VSBW        WRITE CHARACTER TO SCREEN
       JMP  KI2          GO BACK TO SCANNING KEYBOARD
*
* THE FOLLOWING IS A SPECIAL KEY INPUT FOR REPEATING OPERATION OF
* THE RIGHT AND LEFT ARROW KEYS
* THIS SUBROUTINE INCLUDES SELF-MODIFYING CODE
*
KI2A   LI   R5,>0280     LOAD R5 WITH DELAY FACTOR
KI2B   CLR  @STATUS      CLEAR GPL STATUS
       BLWP @KSCAN       SCAN KEYBOARD
       CB   @KEYVAL,@NOKEY  HAS NO KEY BEEN STRUCK?
       JEQ  KI2C         IF SO, JUMP
       LIMI 2            SET INTERRUPTS ON
       LIMI 0            SET INTERRUPTS OFF
       DEC  R5           DECREMENT DELAY COUNTER
       JNE  KI2B         IF NOT ZERO, SCAN AGAIN
       MOVB @ONE,@KI2A+2 ELSE MODIFY DELAY COUNT
KI2C   RT                THEN RETURN
*
* THE FOLLOWING SUBROUTINE FORCES THE CURSOR CHARACTER ONTO THE SCREEN
*
CURFRC LI   R1,>1E00     PUT CURSOR CHARACTER IN LEFT BYTE R1
       LI   R4,>0100     SET DELAY FACTOR IN R4
       BLWP @VSBW        WRITE CURSOR TO SCREEN
       RT                RETURN
*
* FOLLOWING SUBROUTINE CLEARS AN INPUT FIELD
* BEGINNING AT R0 POSITION, EXTENDING NUMBER OF CHARACTERS IN R4
*
CLRFLD
       MOV  R4,R2        PLACE VALUE OF R4 IN R2
       MOV  R0,R3        SAVE R0
       MOVB @ANYKEY,R1   PUT SPACE CHARACTER IN LEFT BYTE OF R1
CLRFL1 BLWP @VSBW        WRITE ONE SPACE IN FIELD
       INC  R0           POINT TO NEXT CHARACTER SPOT
       DEC  R2           DECREMENT COUNT OF SPACES
       JNE  CLRFL1       IF NOT ZERO, REPEAT WRITING OPERATION
       MOV  R3,R0        REPLACE ORIGINAL VALUE OF R0
       RT                RETURN
KEYLOO CLR  @STATUS      CLEAR GPL STATUS BYTE
       BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN SHUT THEM OFF
       CB   @ANYKEY,@STATUS HAS KEY BEEN STRUCK?
       JNE  KEYLOO       IF NOT, SCAN AGAIN
       RT                ELSE RETURN
MOVSTR
       MOVB *R9,R4       MOVE LENGTH BYTE TO R4
       SRL  R4,8         SHIFT RIGHT
```

```
       INC  R4           INCREMENT TO INCLUDE LENGTH BYTE
MOVBTS MOVB *R9+,*R10+   MOVE ONE BYTE, INC POINTERS
       DEC  R4           DECREMENT COUNT
       JNE  MOVBTS       IF NOT ZERO, REPEAT
       RT                RETURN
*
* REQUIRED DATA SECTION
* THE FOLLOWING DATA SOURCE LINES ARE REQUIRED BY THESE SUBROUTINES
*
ONE    DATA 1            ONE AS A WORD
ENDOC  DATA 0            END OF INPUT FIELD
INSFLG DATA 0            INSERT ACTIVE FLAG
PGNUM  DATA 0            STORAGE FOR ONE WORD
SAV4   DATA 0            STORAGE FOR ANOTHER WORD
ONOFF  DATA >0201        ON-OFF BYTES FOR CURSOR
ENDSTR DATA 0            ADDRESS OF END OF STRING ARRAY
* THIS PAB DATA AND MODE BYTES APPLY TO D/V 80 FILES
PAB1DT DATA >0014,BUF,>5000,>0000,>000F
       BSS  15
BADDEV BYTE 15
       TEXT 'BAD DEVICE NAME'
WRPROT BYTE 15
       TEXT 'WRITE PROTECTED'
BADATT BYTE 13
       TEXT 'BAD ATTRIBUTE'
ILLOP  BYTE 17
       TEXT 'ILLEGAL OPERATION'
OUTSP  BYTE 19
       TEXT 'OUT OF BUFFER SPACE'
ENDFIL BYTE 11
       TEXT 'END OF FILE'
DEVERR BYTE 12
       TEXT 'DEVICE ERROR'
FILBAD BYTE 16
       TEXT 'OTHER FILE ERROR'
       EVEN
LUT    DATA BADDEV,WRPROT,BADATT
       DATA ILLOP,OUTSP,ENDFIL
       DATA DEVERR,FILBAD
FNOMSG BYTE 17
       TEXT 'FILE DID NOT OPEN'
EDGE   BYTE >1F
INMD   BYTE >14          BYTE FOR INPUT OF DISPLAY/VARIABLE FILE
OUTMD  BYTE >12          BYTE FOR OUTPUT OF DISPLAY/VARIABLE FILE
APPMD  BYTE >16          BYTE FOR APPEND OF DISPLAY/VARIABLE FILE
UPDAMD BYTE >10          BYTE FOR UPDATE MODE OF D/V FILE -NOT RECOMMENDED
WRITEF BYTE 3            OPCODE FOR WRITE OPERATION
READF  BYTE 2            OPCODE FOR READ OPERATION
CLOSEF BYTE 1            OPCODE FOR CLOSE OPERATION
ANYKEY BYTE >20          KEYSTROKE COMPARE BYTE
```

```
NOKEY  BYTE >FF          NO KEY STRUCK
ALTKEY BYTE 0            STORAGE FOR SCREEN CHARACTER
ENTERV BYTE 13           "ENTER" KEY CODE
RITEV  BYTE 9            RIGHT ARROW
LEFTV  BYTE 8            LEFT ARROW
INFSTR BYTE 15
       TEXT 'INPUT FILE NAME'
OUTSTR BYTE 16
       TEXT 'OUTPUT FILE NAME'
SFSTR  BYTE 23
       TEXT 'SORTING FILE - STAND BY'
OOMSTR BYTE 19
       TEXT 'OUT OF MEMORY SPACE'
TEMSTR BSS  81           TEMPORARY STORAGE LOCATION FOR RECORD
       EVEN
RTNSTK DATA 0            RETURN ADDRESS STACK
       END
```

1.25. The Art Of Assembly — Part 25. Comparative Languages

By Bruce Harrison

As we write this, it's June, and we have recently seen Barry Traver's presentation from the Lima show on video. Barry was discussing the comparisons between the languages TI Extended Basic and PC Quick Basic. That spurred us to pursue an idea we'd had, to compare the Assembly languages between the two machines. We have touched upon some aspects of the PC language from time to time, but never in any depth. Today's the day to really dig into this topic.

1.25.1. No Clear Winner

In our opinion, there is no clear winner between the TI's Assembly language and the PC's. They are different, but each has its strengths and weaknesses. That will, we think, become self-evident by the end of this article. There are of course things that the TI can do which are near impossible on the PC, regardless of language used. There are things that can be done much more easily on the PC than on the TI, partly because of the ease of getting access to information about the workings of the machine and its operating system software. Our nearby public library has several shelves full of books on the PC. It is much more difficult to probe the inner workings of our beloved TI, and many of its quirks are still not fully understood.

1.25.2. Simple Operations

We'll start with some very basic little operations, like MOV operations. In the TI, one can move either a word or a byte from memory to register, register to register, register to memory, or memory to memory. On the PC, one can't do memory to memory with the MOV operation, but the other three are okay. One can, however, move immediate values into either registers or memory on the PC, while immediate values can only be moved (LI) into registers on the TI. Suppose, for example, we had a word in memory labeled COUNT, and wanted to set that word to >FE84. On the TI, we could do it this way:

```
LI   R1,>FE84
MOV  R1,@COUNT
```

On the PC, we could accomplish this more simply by:

```
MOV COUNT,0FF84h
```

Right away, we see another difference which can cause confusion when switching back and forth between these two languages. On the TI, the order of appearance for MOV operations (Not LI operations) is OPCODE SOURCE, DESTINATION, while on the PC it's OPCODE DESTINATION, SOURCE. This makes it imperative to remember on which machine you're working at all times. Note also that the syntax for the immediate value is different. On the TI, we use the symbol > to indicate that what follows it is a hex number. On the PC, the letter H is tacked on at the end of the number to designate hex, but a zero is added at the front to tell the assembler that what follows is an immediate numeric value, not a label name. For immediate values that begin with a numeric character, or those written in decimal, that leading zero can be omitted, as in MOV COUNT,58.

In both machines, things get a bit more complicated for byte MOV operations than for word operations. Let's continue with the supposition that COUNT is a word in memory, but suppose that we want to move a byte value of >1F into the low-order byte of that word. On the TI, we could:

```
LI   R1,>1F00
MOVB R1,@COUNT+1
```

On the PC, we could do this:

```
MOV COUNT,1Fh
```

hat, though, would also affect the high order byte at COUNT, because the assembler would take the 1Fh as a word value because the label COUNT was defined as a word. To move the value >1F into the low order byte without affecting the high order byte, we'd have to write:

```
MOV BYTE PTR COUNT,1Fh
```

The words BYTE PTR tell the assembler to move something into just the byte at that label.

1.25.3. The Back-words Problem

You'll notice in the section above that the low order byte at the word location COUNT was at COUNT+1 on the TI, while on the PC, the byte at COUNT was the low-order byte. The PC stores words "backward" compared to the TI. On the TI, a word is stored as High byte, Low byte. On the PC, it's always Low byte, High byte. This won't really matter unless you're trying to take source code from one machine and "translate" onto the other. In the above example, if we wanted to move the value >1F into the High byte on the PC, we'd have to:

```
MOV BYTE PTR COUNT+1,1Fh
```

It's just one more "memory test" for the programmer who switches back and forth between TI and PC. Your author has been known to forget, on occasion.

1.25.4. The Registers

The TI would appear to be a clear winner in the matter of workspace registers. We have the luxury of sixteen registers, can place them anywhere in memory, and can "context switch" by a BLWP operation to a whole new set of registers when we wish to, or simply LWPI to another set if we need more register space. On the PC, there are only four "General Purpose" registers, and they can't all be used for all purposes. On the other hand, one can treat them as words or bytes on the PC, but not on the TI. The PC's four "general purpose" registers are: AX, BX, CX, DX. They can each be "subdivided" into byte registers, so that AX can be treated as two byte registers AH and AL, BX as BH and BL, and so on. Thus we can move a byte value into the low byte of AX by MOV AL,1Fh. On the TI, if we wanted to do that without disturbing the high byte, we'd have to do some fancy shuffling, or at least more moves than just that one. For example, we could MOVB R1,R15, LI R1,>1F00, SWPB R1, then MOVB R15,R1.

We said somewhere along the line that things would get complicated, didn't we?

1.25.5. Loops And Reps

We've all used loops in our source code for one reason or another, and there's a sort of standard structure to them. Let's suppose we want to perform some operation five times. On the TI, we do a loop like this:

```
            LI   R5,5               set for five times loop
DOOPER      (Perform operation once)
            DEC  R5                 decrement count
            JNE  DOOPER             if not zero, repeat
```

On the PC, such loops can be performed more simply by using the special nature of the CX register. One can do the loop like this

```
            MOV CX,5                ;set CX register to 5
DOOPER:     (perform operation once)
            LOOP DOOPER             ;loop back
```

The single instruction LOOP DOOPER, on the PC, performs the operations of DEC CX, CMP CX,0 and JNE DOOPER. That's handy, so long as CX is available at this point in your program. If that's not so, for example in the case of two nested loops, one can still use this structure with the help of the PUSH and POP operations that the PC offers. The PC maintains a stack for temporary storage of word values from any source. The instruction PUSH places what follows on the stack, and decrements the stack pointer by two. POP does the opposite.

Let's assume we needed to nest two loops in a PC program, and needed CX for counting both. We could do it like this:

```
            MOV CX,10               ;set outer loop count
OUTLP: (perform steps)
            PUSH CX                 ;Stash CX on stack
            MOV CX,5                ;set inner loop count
INLP: (perform operation of inner loop)
            LOOP INLP               ;Five times
            POP CX                  ;Get old value of CX back
            LOOP OUTLP              ;repeat 10 times
```

On the TI, we'd just use two registers for the two loop counts, and that might be much easier, but then PUSH and POP are easy too.

The TI offers no equivalent of the PC's REP operation. There are some operations on the PC where we can use a count value in the CX register to repeat without the loop instruction. One good example is the case of moving some group of bytes from one location in memory to another.

To make this more understandable, we'll have to introduce the concept of the Source and Destination registers. These two registers, called SI (source index) and DI (destination index) are used to point to memory locations. The SI register is understood to point to the DATA SEGMENT, and the DI to the EXTRA SEGMENT, but let's for the moment assume that these are one and the same. To use the REP instruction, we must first set up SI, DI, and CX. Let's say we want to move fifty bytes from a location labeled STRING1 to a location labeled STRING2. On the PC, we can do this:

```
            MOV CX,50               ;set count in CX
            MOV SI,OFFSET STRING1   ;address string1
            MOV DI,OFFSET STRING2   ;address string2
            REP MOVSB               ;repeat MOVSB CX times
```

Here we've used another new word in the PC lexicon, OFFSET. This simply means that we're moving the address of STRING1 and STRING2 into SI and DI, not the value located at that address. The single instruction REP MOVSB will move a byte from the location pointed by SI to the location pointed by DI, will increment both SI and DI, decrement CX, compare CX to zero, and loop back to MOVSB if CX is not zero. That's a powerful instruction, but must be used with care. If one makes the mistake of invoking it when CX happens to be zero to start with, a whole 64K segment will get moved before the REP stops. This can be disastrous.

Just a bit ago, we mentioned the DATA SEGMENT and EXTRA SEGMENT without explanation. There is no equivalent to these on the TI, since the entire memory address space on the TI is one 64K "segment". On the PC, one may set aside segments of any number of bytes up to 64K. The program's code can occupy one or more such segments. Subroutines can also occupy more than one code segment. Data can be loaded into two or more such segments. One can, for example, keep all the program's variables and constants in one segment, and set aside an entire 64K for the user's data. Two registers are used to "point" to these two segments. These are called DS and ES for DATA and EXTRA. In many cases both DS and ES point to the same physical segment of memory, but they can be treated separately.

Like other registers, these can be "pushed" and "popped" to and from the stack, which has its own segment. Yes, this eats memory like crazy, but that's the nature of doing things on the PC anyway. Let's assume we want to set the ES to the same as the DS for some operation, then put it back later to its original value. We could:

```
PUSH ES                 ;save original ES
PUSH DS                 ;save DS
POP ES                  ;set ES to DS from stack
(perform operation)
POP ES                  ;restore original ES
```

The problem with using this stack business is to match up all the pushes and pops (each PUSH must have a corresponding POP) so that one doesn't mess up the stack pointer or accidentally get the wrong value into a register.

We've used a lot of this space on the PC's language, but of course most of this whole series has been dedicated to the TI, so we hope you'll excuse our going on somewhat about the PC this time.

1.25.6. Initializing Data

As we mentioned before, our common practice when writing Assembly source code is to keep all our data in a separate part of the source file. On the PC, the data has its own segment, so it must be kept apart from the code. (There are ways to "cheat" on this process, and have some data accessed from within the code segment, but we're not going to teach cheating here.) On the TI, one identifies data as belonging to one of three kinds, DATA (a word), BYTE (a byte) or TEXT (a string of alphanumeric characters). On the PC, there are only two categories for data statements, DW (a word) and DB (a byte). (There's also DD for a Double Word of DATA, but let's skip that for now.) How then does one introduce the text for prompts and such?

It's easy. One uses the DB notation, but with single quote marks, just like the TEXT in a TI source statement. For example, a string can be done like this:

```
PAKMSG      DB 25,'PRESS ANY KEY TO CONTINUE'
```

Here we've taken advantage of the situation to stick in the length byte and the string's content in a single source code line. On the TI, we accomplish the same effect with two lines:

```
PAKMSG      BYTE 25
            TEXT 'PRESS ANY KEY TO CONTINUE'
```

That takes a bit more work, but accomplishes the same result, and takes up exactly the same amount of memory.

1.25.7. BIOS And DOS Services

By now you're all familiar with the concept of Utility Vectors on the TI, like VMBW, KSCAN, DSRLNK, and such. There are similar things on the PC, but they are called Interrupts. There are two sets of them loaded into the lower part of the PC's memory when it's "booted". The first set are the BIOS (Basic Input/Output System) Interrupts. These are sort of "primitive" routines, but can be extremely useful. The second set are called DOS (Disk Operating System) Interrupts. These are more "advanced", but in some ways harder to use than the BIOS ones. For many of these interrupts, more than one service is available within one interrupt. You tell the interrupt which service is desired by placing a number in the AH register before calling the interrupt itself. One might, for example, do this:

```
MOV AL,'A'
MOV AH,14
XOR BH,BH
INT 10h
```

Interrupt 10h is part of the BIOS. The call to INT 10h would place the character "A" on the screen, at whatever position the cursor happened to be, and would advance the cursor by one position on the screen. We XOR'd BH with itself here so that the character would go to the normal screen. Positioning the cursor uses a different service from INT 10h, (AH = 2) with the DH and DL registers indicating the Row and Column on the screen, and BH indicating which screen "page" is being written to. INT 10h is one of the most useful of all the Interrupts available, since it can be used for many "video" related purposes. One could think of it as a combination of TI's VMBW, VSBW, VWTR, and VSBR. It will do more than these four TI services. Other BIOS interrupts will read the keyboard, read and write disk sectors, and so on.

In its "TEXT" mode, the PC has four screens (numbered 0 thru 3) that we can write to. We could pre-write a whole screenful of text to screen 1 while the monitor is showing screen 0, then use a BIOS call to switch screens and reveal screen 1 "instantly" on the monitor. (A few months ago, we explained how this can be done on the TI in VDP RAM by using the VWTR utility.)

The Interrupts that are part of the DOS are generally more complex, and vary considerably from one version of DOS to another. When we write for the PC, we use DOS interrupts belonging to Version 2 of DOS, as these will work on all but the oldest DOS versions, and will also work with later versions.

Like the BIOS Interrupts, the DOS ones have multiple functions built into each interrupt handler. Just the single Interrupt 21h from DOS has perhaps 50 functions. (Maybe more than that by now. We don't really know, since we haven't tried Version 5.0 of DOS. Each new version adds more services to INT 21h.)

There are some good DOS services available, and of course we've used them in our PC work. Opening, writing, reading, and closing files are all easily accomplished with INT 21h, at least from version 2.0 onwards. There are some services offered in DOS which Peter Norton correctly describes as "screwball" services. Some of these were carry-overs from pre-DOS operating systems like CP/M. Our practice is to avoid the DOS services except for disk file operations, and use BIOS calls for operations involving the screen and keyboard. In addition to these BIOS calls being simpler and more flexible, they also execute faster. Many of the DOS services make repeated calls to the BIOS services to perform their tasks, and this makes them execute much more slowly than direct calls to the BIOS services.

This article could go on for many more pages, but we'll stop here having just provided a glance inside the PC's Assembly language. It's definitely not "superior" to the TI's, but it has some really powerful instructions, and in experienced hands it can make truly wonderful things happen. See you next month with an "all TI" column.

1.26. The Art Of Assembly — Part 26. Odds And Ends

By Bruce Harrison

Today we're picking up some loose ends left behind from previous parts in this series, and passing along some insights. We'd like to start with a couple of sincere "Thank You" notes.

First goes to Mr. Harley Ryan, Jr. of Whitehall, OH. He got us started on the path to solving the "strange case" of TI's GPLLNK, which we talked about in the July 1992 issue. Mr. Ryan found a solution for the Auto-Run Option 3 program, which led us down quite a path. Thanks also to Millers Graphics, for publishing the solution passed along to us by Mr. Ryan.

Second thanks goes to Mr. Merle Vogt, of Von Ormy, TX. He passed along a different way of solving the utilities problem for Option 5 programs, plus another suggestion for the opening and closing sections of an Assembly program, for Option 5 or Option 3.

1.26.1. The Strange Case. . .

Back in Part 14, which appeared in July 1992, we reported that we'd never been able to get TI's GPLLNK to work correctly under either an Auto-Start Option 3 or an Option 5 situation. Shortly after that article appeared, Mr. Ryan sent a letter which gave a solution to the problem for Auto-Start Option 3 programs. He found his information in the manual for the Millers Graphics Explorer. It seems the TI GPLLNK depends on the state of the GROM address when it's called, and that is set differently by Option 3 depending on whether the program loaded has auto-start or not.

Miller's suggestion was to read the GROM address in the manner given by the E/A manual, then add >63 if your program is to auto-start and write that back through GRMWA. This works. In today's Sidebar is the short test program we put together immediately after getting Mr. Ryan's letter. This will work for Auto-start Option 3 programs with both the E/A module and the Mini-Memory. Needless to say, we were delighted. Our delight was, however, short-lived, because Option 5 turned out to be quite a different can of worms.

If one tries adding >63 to the GROM address for an Option 5 program, the GPLLNK does not work! After doing some detective work, inserting a "HEXDIS" routine into our source code, we found that we could make an Option 5 program work with GPLLNK under certain conditions by adding >2E4 to the GROM address. That was not all. We also had to stash and restore all of the low memory area from >2000 through >2676 instead of the shorter utilities portion from >2094 through >23BA, as we'd suggested in Part 14.

That worked. It worked as long as we entered our program from E/A Option 5 itself. Part two of today's Sidebar shows the source code for that solution. Of course we couldn't stop trying things, so, since we have a Ramdisk on our system which allows direct running of Option 5 programs from its menu screen, we tried that. We also have a P-Gram, which allows us to select either Extended Basic or Editor/Assembler with just the press of a key. This opened yet another can of worms for us. If, for example, the menu selection at the bottom of the Ramdisk menu said Extended Basic when we direct ran our Option 5 program, the call to GPLLNK placed us in XB without the normal character set. A mess. Typing in BYE **ENTER** got us back to the Ramdisk menu, but taught a valuable lesson. Adjusting the GROM address in this way couldn't be done under those circumstances.

1.26.2. The "Final Solution"

Finally we knew why Doug Warren and Craig Miller developed their own GPLLNK, rather than try using TI's version. To make our Option 5 test program work with the P-Gram and Ramdisk situation we just described, we had to devise a means of first finding the right GROM in the P-Gram. This we did by a trial and error method of comparing certain key bytes in each GROM address from >6000 up, in >2000 steps. Once we'd found the correct GROM, (in our case at >E000) we set up the correct GROM address as >E892, wrote that through GRMWA, and everything worked. This also works when run from Option 5 of E/A, either using P-Gram or using an E/A module. As you can see in Part 3 of the Sidebar, this takes a whale of a lot of work. It's also just possible that no E/A is actually in the P-Gram, so we put in an escape exit to the vector at >0000 in case the E/A GROM was not found.

The why, then, for Warren and Miller's GPLLNK is probably that using the TI GPLLNK can be a lot more trouble than it's worth. We've shown that it can be done, but all the code that was necessary to make it work adds up to more memory usage than just putting the Warren/Miller GPLLNK into our source file to begin with. Any of our readers can excerpt the source code shown in part 3 of the Sidebar and prove for himself that it works, but we really don't recommend this approach for general use. (See Part 14 in the July '92 issue for how to use this source code, should you wish to try it out.)

1.26.3. Other Ins And Outs

Merle Vogt is a devoted reader of this column, and has more than once come up with suggestions for ways of doing things that are different from our ways. We enjoy this, and will pass along two of his suggested methods today. First, a different way of handling the entry and exit problem for any Assembly program. At the beginning, try this:

```
START       STWP R12                STASH THE INCOMING WORKSPACE
            MOV  R12,@E1+2          MOVE THAT TO LOCATION BELOW
            MOV  R11,@E2+2          MOVE THE RETURN ADDRESS
            LWPI MYWS               LOAD YOUR WORKSPACE
            (program continues)
```

Then at the exit:

```
EXIT        CLR  @>837C            CLEAR GPL STATUS BYTE
E1          LWPI 0                 DUMMY ADDRESS FOR LWPI
E2          B @0                   DUMMY RETURN ADDRESS
            MYWS BSS 32
```

This works as follows: The STWP at the very beginning places the workspace pointer for whatever workspace was in use before your program started into R12 of that workspace. Next, the contents of that R12 are moved into the spot two bytes beyond label E1, replacing the zero that was there in the source code. The return address from R11 of the original workspace will then be placed two bytes past label E2, replacing the zero that was there. If, for example you entered with the GPL workspace, and a return address of >0070, the code at E1 and E2 will be self-modified to:

```
E1          LWPI >83E0             LOAD GPL WORKSPACE
E2          B    @>0070            BRANCH TO >70 ADDRESS
```

On some occasions we have tried this kind of approach with success, so we're sure Merle is right when he says this will work for just about any way of running your Assembly program.

1.26.4. The Utilities Problem

We said more than once in this column that there are nearly as many ways of doing anything in Assembly as there are people trying to do it. Merle Vogt has once again proved that to be true. He came up with a way of saving and restoring the E/A utilities for Option 5 programs that we'd never have thought of, but it's an effective way of doing the deed. With another thanks to Merle, here's that way.

Start by making this very simple source file:

```
            DEF  SFIRST,SLOAD,SLAST
SFIRST      EQU  >2000
SLOAD       EQU  >2000
SLAST       EQU  >2676
            END
```

Assemble this and name it MODULE2/O or MODULE2OBJ, or whatever you like. Now load this under Option 3, then load TI's SAVE utility and run program name SAVE. Call the memory image file MODULE2.

Now place the labels SFIRST, SLOAD and SLAST in the appropriate places in your main program's source file. Assemble that, then load it under Option 3, load TI's SAVE utility, and SAVE that as MODULE1.

Here's where it gets a tad tricky. Using a sector editor, find the first sector of the MODULE1 file and change its first two bytes from >0000 to >FFFF. Write the modified sector back to the disk. Make sure that MODULE1 and the MODULE2 file you made previously are on the same disk.

Choose Option 5 and load MODULE1 from whatever drive the disk happens to be in. Because the header of MODULE1 has that >FFFF in it, the Option 5 loader will look on the same drive for a file called MODULE2, and will load that into memory as well. Voila! The utilities area from the E/A will be restored into place in low memory just where it belongs and your program can use those utilities by simple REFs.

If your main program itself is large enough to become more than one file when saved by TI's SAVE utility, then you must re-name the MODULE2 to be the next number in line, (e.g. MODULE3 or MODULE4) and make that sector edit in whatever was the last file in your main program's sequence.

We haven't actually tried this yet, but have no doubt that it will work. Normally we try to use and recommend methods that don't assume our readers have such programs as sector editors handy, but if you have one, as many people do, then Merle's approach may be just the thing for building your bridge to Option 5.

1.26.5. Nobody's Perfect

Some time back, we offered some business advice in this column, and boldly asserted that in the years since 1988, when we first started selling software to the TI community, we'd never had a check bounce. It's August 1992 now, and we can't say that any more. An overseas customer wrote us a check on a U.S. bank, and that came bouncing back to us as "ACCOUNT CLOSED". Sooner or later it had to happen, but that's only one check out of hundreds that we've received, so the TI community still will get prompt shipments from us. Of course if it happens again. . .

Today's column has been a real mixed bag of stuff to chew on. Next month we'll try to stick to one topic, and beat that one to death instead of skipping around the map.

```
* PART ONE - AUTO-START
* OPTION 3 FROM E/A MENU
* TEST PROGRAM - PUBLIC DOMAIN
* BY B. HARRISON
*
       REF  GPLLNK,GRMRA,GRMWA,KSCAN
       REF  VMBW
       DEF  START
START
       LWPI >20BA        LOAD USER WORKSPACE
       MOVB @GRMRA,@GRMSAV READ HIGH BYTE GROM ADDRESS
       NOP               KILL TIME
       MOVB @GRMRA,@GRMSAV+1 READ LOW BYTE GROM ADDRESS
       NOP               KILL TIME
       DEC  @GRMSAV      DECREMENT THE ADDRESS
       A    @HEX63,@GRMSAV ADD >63
       MOVB @GRMSAV,@GRMWA WRITE HIGH BYTE BACK
```

```
       NOP               KILL TIME
       MOVB @GRMSAV+1,@GRMWA WRITE LOW BYTE BACK
       NOP               KILL TIME
       LI   R0,22*32+8   POINT TO ROW 23, COL 9
       LI   R1,TEST      MESSAGE
       LI   R2,15        LENGTH OF MESSAGE
       BLWP @VMBW        WRITE TO SCREEN
       BLWP @GPLLNK      CALL GPLLNK
       DATA >34          FOR BEEP SOUND
KEYLOO BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN SHUT OFF INTERRUPTS
       CB   @ANYKEY,@>837C KEY STRUCK?
       JNE  KEYLOO       IF NOT, KEEP SCANNING
       LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         RETURN TO GPL INTERPRETER
GRMSAV DATA 0            WORD TO STASH GROM ADDRESS
HEX63  DATA >63          VALUE HEX 63
ANYKEY BYTE >20          KEYBOARD TEST BYTE
TEST   TEXT 'THIS IS A TEST ' MESSAGE
       END  START



* PART TWO - FOR E/A OPTION 5
* THIS DOES NOT WORK ALL CASES
*
       REF  GPLLNK,GRMRA,GRMWA,KSCAN
       REF  VMBW
       DEF  SFIRST,SLOAD,SLAST
SFIRST
SLOAD
       LWPI WS           LOAD OUR WORKSPACE
       LI   R9,DATALD    POINT AT SAVED UTILITIES
       LI   R10,>2000    POINT AT LOW MEMORY
       LI   R4,>2676->2000 BYTES TO MOVE
PUTUT  MOV  *R9+,*R10+   MOVE ONE WORD
       DECT R4           DECREMENT COUNTER BY TWO
       JNE  PUTUT        IF NOT ZERO, REPEAT
       MOVB @GRMRA,@GRMSAV READ HIGH BYTE GROM ADDRESS
       NOP               KILL TIME
       MOVB @GRMRA,@GRMSAV+1 READ LOW BYTE GROM ADDRESS
       NOP               KILL TIME
       DEC  @GRMSAV      DECREMENT THE ADDRESS
       A    @HEX2E4,@GRMSAV ADD HEX 2E4
       MOVB @GRMSAV,@GRMWA WRITE HIGH BYTE BACK
       NOP               KILL TIME
       MOVB @GRMSAV+1,@GRMWA WRITE LOW BYTE BACK
       NOP               KILL TIME
       LI   R0,22*32+8   POINT TO ROW 23, COL 9
```

```
       LI   R1,TEST      MESSAGE
       LI   R2,15        LENGTH OF MESSAGE
       BLWP @VMBW        WRITE TO SCREEN
       BLWP @GPLLNK      CALL GPLLNK
       DATA >34          FOR BEEP SOUND
KEYLOO BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN SHUT OFF INTERRUPTS
       CB   @ANYKEY,@>837C KEY STRUCK?
       JNE  KEYLOO       IF NOT, KEEP SCANNING
       LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         RETURN TO GPL INTERPRETER
WS     BSS  32           OUR WORKSPACE
GRMSAV DATA 0            WORD TO STASH GROM ADDRESS
HEX2E4 DATA >2E4         VALUE HEX 2E4
ANYKEY BYTE >20          KEYBOARD TEST BYTE
TEST   TEXT 'THIS IS A TEST ' MESSAGE
       EVEN
DATALD BSS  >2676->2000  STORAGE AREA FOR UTILITIES
SLAST
       DEF  SAVIT        DEFINE SAVIT ENTRY POINT
       REF  SAVE         REF TI SAVE UTILITY
SAVIT
       LWPI WS           LOAD OUR WORKSPACE
       LI   R9,>2000     POINT AT START OF LOW MEMORY
       LI   R10,DATALD   POINT AT DATA SPACE FOR UTILITIES
       LI   R4,>2676->2000 BYTES TO MOVE
GETUT  MOV  *R9+,*R10+   MOVE ONE WORD
       DECT R4           DECREMENT COUNTER BY TWO
       JNE  GETUT        IF NOT ZERO, REPEAT
       B    @SAVE        BRANCH TO SAVE UTILITY
       END

* PART 3 - GENERAL CASE OPTION 5
* WILL ALSO WORK FROM RAMDISK MENU RUN OPTION
* AND P-GRAM WITH E/A INCLUDED


       REF  GPLLNK,GRMRA,GRMWA,KSCAN
       REF  VMBW,GRMRD
       DEF  SFIRST,SLOAD,SLAST
SFIRST
SLOAD
       LWPI WS           LOAD OUR WORKSPACE
       LI   R9,DATALD    POINT AT SAVED DATA
       LI   R10,>2000    AND AT START OF LOW MEMORY
       LI   R4,>2676->2000 BYTES TO MOVE
PUTUT  MOV  *R9+,*R10+   MOVE A WORD
       DECT R4           DEC R4 BY TWO
       JNE  PUTUT        NOT ZERO, REPEAT
```

```
       LI   R3,>6000     SET R3 TO >6000
TGROM  BL   @WGA         WRITE ADDRESS IN R3 TO GROM ADDRESS
       BL   @RGD         READ ONE BYTE
       CB   R5,@HEXAA    IS THAT HEX AA?
       JNE  NXTGRM       IF NOT, SKIP AHEAD
       INC  R3           ELSE INCREMENT ADDRESS IN R3
       BL   @WGA         SET FOR THAT ADDRESS
       BL   @RGD         READ BYTE
       CB   R5,@HEX01    COMPARE TO HEX01
       JNE  NXTGRM       IF NOT EQUAL, JUMP AHEAD
       INC  R3           ELSE LOOK AT NEXT ADDRESS
       BL   @WGA         SET THAT
       BL   @RGD         READ FROM GROM
       CB   R5,@HEX01    COMPARE
       JNE  NXTGRM       IF NOT EQUAL, JUMP
       AI   R3,5         ADD 5 TO R3 ADDRESS
       BL   @WGA         SET THAT ADDRESS
       BL   @RGD         READ A BYTE
       CB   R5,@HEX10    COMPARE TO HEX 10
       JEQ  GRMOK        IF EQUAL, WE'VE FOUND E/A IN GROM
NXTGRM ANDI R3,>F000     MASK ALL BUT FIRST NYBBLE
       AI   R3,>2000     ADD >2000
       CI   R3,>0000     IS THAT NOW ZERO?
       JEQ  ALTEX        IF SO, NO E/A FOUND
       JMP  TGROM        ELSE GO LOOK AT NEXT POSSIBLE ADDRESS
GRMOK  ANDI R3,>F000     MASK OFF ALL BUT LEFT NYBBLE
       AI   R3,>0892     ADD >892
       BL   @WGA         WRITE THAT ADDRESS TO GRMWA
       LI   R0,22*32+8   POINT TO ROW 23, COL 9
       LI   R1,TEST      MESSAGE
       LI   R2,15        LENGTH OF MESSAGE
       BLWP @VMBW        WRITE TO SCREEN
       BLWP @GPLLNK      CALL GPLLNK
       DATA >34          FOR BEEP SOUND
KEYLOO BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN SHUT OFF INTERRUPTS
       CB   @ANYKEY,@>837C KEY STRUCK?
       JNE  KEYLOO       IF NOT, KEEP SCANNING
       LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         RETURN TO GPL INTERPRETER
ALTEX  LIMI 2            ESCAPE EXIT POINT
       LWPI >83E0        LOAD GPL WORKSPACE
       BLWP @0           RETURN TO VECTOR AT 0

WGA    MOVB R3,@GRMWA WRITE HIGH BYTE OF ADDRESS
       SWPB R3           SWAP BYTES R3 AND KILL TIME
       MOVB R3,@GRMWA WRITE LOW BYTE OF ADDRESS
       SWPB R3           SWAP BYTES R3 AND KILL TIME
       RT                RETURN
```

```
RGD
       MOVB @GRMRD,R5    READ A GROM BYTE INTO R5
       NOP               KILL TIME
       RT                RETURN
WS     BSS  32           OUR WORKSPACE
GRMSAV DATA 0            WORD TO STASH GROM ADDRESS
HEXAA  BYTE >AA          HEX VALUE AA
HEX01  BYTE >01          VALUE 1
HEX10  BYTE >10          VALUE HEX 10
ANYKEY BYTE >20          KEYBOARD TEST BYTE
TEST   TEXT 'THIS IS A TEST ' MESSAGE
       EVEN
       DEF  SAVIT        DEFINE SAVIT ENTRY
       REF  SAVE         REF THE SAVE UTILITY
DATALD BSS >2676->2000   UTILITY STORAGE AREA
SLAST
SAVIT
       LWPI WS           LOAD OUR WORKSPACE
       LI   R9,>2000     POINT AT START OF LOW MEM
       LI   R10,DATALD   AND AT STORAGE AREA
       LI   R4,>2676->2000 NUMBER OF BYTES
GETUT  MOV  *R9+,*R10+   MOVE A WORD
       DECT R4           DEC COUNT BY TWO
       JNE  GETUT        IF NOT ZERO, REPEAT
       B    @SAVE        BRANCH TO TI SAVE UTILITY
       END
```

1.27. The Art Of Assembly — Part 27. Another Potpourri

By Bruce Harrison

We've revisited the Geneve subject on occasion in this series, and today here we go again. Some months back, we had a small crisis on our hands, with people who bought our Word Processor and then found out it wasn't compatible with their Geneve computers. We scrambled about a bit, trying to help our customers. Reworked the source code so that the DSRLNK would be taken from the TI E/A utilities, fixed up the loaders, and so on. We then shipped off a copy to our fellow columnist Stan Krajewski, whom we knew had a Geneve handy for testing.

Stan reported that everything seemed to work okay. Had we at last slain the dragon? It was another of those things that are too good to be true, and the untrue part came when we shipped copies to our two customers. On one of those two machines, nearly everything worked, except that the machine would lose the "index" file when saving a document to disk, so that afterwards the program wouldn't recognize that document as existing on the disk. The page files would show up on the disk, but not the vital index file. Why? We still don't know. This version worked perfectly on our TI. On the other Geneve, many of the important Function and Control key combinations wouldn't work, and it too failed to save the index files for documents.

Much later, we decided to ask Stan a few questions about his Geneve, to see if there was some rationale to this phenomenon. It turned out that Stan's Geneve was using a TI disk controller instead of a Myarc one. That could explain the business of the index files, but not the sensing and processing of keyboard inputs. Sorry, but we have no answers. Stan suggests it may have something to do with the EEPROMS, which in his case are the very latest versions. Maybe so, but the inconsistency from one Geneve to another makes it darned near impossible to be confident of making something compatible. We refunded the full purchase price to our customers, and gave up.

1.27.1. String Magic

Those who program in assembly get used to the idea that we can do pretty much anything with any of the computer's memory, and that's true so long as we need not return to some other program, like Extended Basic. Your author of course did a lot of programming in Extended Basic before starting to learn assembly. One of the frustrations in XB is the business of using string variables. As many of you already know, XB uses the VDP RAM to store string variables, and provides no other way to store them. You might write a program that takes only 2K of the high memory to load and run under XB, but if it uses lots of strings, XB is likely to run out of what it calls "Stack" space in VDP RAM, and thus your program will not be able to run.

Our good friend Jim Peterson complained to us about this, and we thought about it for a while. Later, one of our customers (Bill Harms of Chino, CA) wrote us a letter suggesting that something be done about this problem. We thought some more, then started working on source code. Both Bill Harms and Jim Peterson helped in the testing of this stuff, and both found cases where the resulting product could be used.

1.27.2. Stashing In Memory

The idea was simple. If your XB program itself uses very little of the High memory, we create a space in the leftover memory to stash an array of strings. If your XB program is large, so there's little or no high memory available, we provide another utility to stash strings in the available low memory. The TI provides pointers we can use to figure out how much of high or low memory is available, so we don't wind up corrupting anything belonging to your XB program. In today's Sidebar is the source code for the utility that stashes strings in high memory. There are four entry points defined in this source code, each of which uses parameters from a CALL LINK in XB. The high memory one shown has entry points SETHI, PUTHI, GETHI, and AVHI. There are lessons to be learned from this code, and we'll try to make this a "learning experience" for you, so don't think of this as a "commercial" for one of our products.

At SETHI, the assembly program does several things: First, it finds out from the CALL LINK parameter how many strings the XB program wants to make room for. Next, it makes a "lookup table" in low memory for the addresses of the strings it will place in high memory. Finally, it creates an array of null strings in the high memory space that's available. While it's doing this, it checks to see whether enough space is there. If you ask SETHI to make room for 6000 strings, it will run out of low memory space for the lookup table, and will report "NOT ENOUGH LOW MEMORY" on your screen. If this is the only utility being used by your XB program, there will be room for about 2900 strings' addresses in the lookup table.

Of course when you start actually assigning non-null strings into the array, more high memory will get used. As each PUTHI linkage is performed, the end point of the array is checked against the limit of available high memory, and if that's going to be exceeded, the error report "NOT ENOUGH HIGH MEMORY" will be issued on screen before that string is assigned. Just to give you the idea how powerful this thing is, we'll take an example from Jim Peterson. Jim used the routine with a multi-column printing program he'd written. Without the routine, this would run out of "stack" space before much of a multi-column page could be composed in a string array. With the routine in place, Jim's program could make 5 columns of sixty lines each at 28 characters per line (for a program listing) without running out of the high memory. With the length bytes, that's 8700 bytes in 300 strings.

The way strings are actually stashed in VDP RAM by XB is somewhat of a mystery. We did a little experiment that stored 300 strings of 28 bytes each in a string array variable under XB, and found that the amount of "stack" space reported by SIZE changed by 10,221 bytes, not 8700. Obviously there is some kind of lookup table space that's also kept in VDP RAM, and that accounts for the "overhead" of 1521 bytes. That array of 300 28-byte strings would fit in VDP RAM if it were the only string variable used in the program, but it wouldn't take many more string variables to cause the XB program to crash with an "out of memory" error.

Among other things, XB reserves space for string variables in blocks that are larger than the actual string length. If you start in XB command mode with nothing and take SIZE, you find 11840 bytes of stack free. If you then enter a statement like X$="", you'll find SIZE reports only 11830 bytes free. That null string has eaten up ten bytes. This no doubt makes things simpler for XB to deal with, but not for the programmer who's trying to cram lots of strings into VDP RAM.

In our own routine for storing in high memory, we keep the space occupied by the strings themselves limited to exactly the required amount, so that a string of zero characters occupies only one byte (the length byte), while a ten-byte string occupies exactly 11 bytes, and so on. Each time a string in the array is replaced, the space in memory beyond that one is "closed up" so the string put in place just fits in memory. By keeping the lookup table for addresses of the strings in low memory, we make all of the available high memory useful for strings in the array.

Bill Harms, after testing an early version, suggested that the XB programmer should be able to find out how much string storage space our routine still had available while his XB program was running. That seemed like a worthwhile idea, so we added the CALL LINK for AVHI, which can be used at any time either from Command mode or while a program is running. It reports the memory still available into a numeric variable named in the CALL LINK. We mention this mainly to illustrate a point, that having a "third party" try out your program at an early stage is a good idea. In this case it meant a lot of disks shipped from coast to coast, and a number of interesting phone calls, but made a much improved product possible.

At one point, we thought we had the project finished, only to find that one of our error traps didn't work as we thought it should. Bill Harms tried something we'd never thought to try, and sure enough it crashed his TI. That's another advantage to having somebody else try things out. The guy who wrote the code never thinks of doing things that have the potential for disaster. Thanks to Bill, that problem was detected and then solved.

Bill Harms made another important contribution, when he pointed out that our "internal" error traps in the Assembly routine would exit to XB as if no error had occurred, thus allowing XB to perhaps keep executing a loop that was needless. What Bill wanted was some way that XB could recognize that an error had occurred in the CALL LINK statement. After some hard thinking, we came up with one of those simple but elegant solutions to this problem. None of the calls to the routine uses more than two parameters. We thus were able to add a BLWP @NUMASG to the error trap's exit, and by setting R1 for a non-existent third parameter, made XB recognize and report an error. Given this, the XB programmer can build in an ON ERROR situation in his XB program, or can simply let the XB program stop when the error happens. That's not all, though. By doing the error report to XB through a third parameter, we allowed the XB programmer to have XB ignore the error by the simple expedient of putting a third parameter as a numeric variable into the CALL LINK.

It works like this. Suppose XB is getting a string from the array into X$: CALL LINK("GETHI",X$,Y) will take the Yth string from the array in high memory into X$. If Y is too big a number, or zero, or there's some other error detected by the Assembly error trap, XB will also report a "BAD ARGUMENT IN ZZZ" when it regains control. To make XB ignore the error, we'd CALL LINK("GETHI",X$,Y,X). Here that third parameter X will get a nonsense value assigned to it, but XB will not recognize any error. Thus the XB programmer can have it both ways, either having XB recognize the error or ignore it. This was one of those "serendipity" results which sometimes happen in programming work, where a simple solution to a problem becomes "elegant programming". We didn't originally plan it that way, but it's a rather nice outcome anyway.

We point out all these things mainly to emphasize the need for an "independent" test agent. For some of our software projects, my partner Dolores serves in that role. For our Word Processor, she served as "guinea pig" almost from its inception, and could always find the bugs in each new version. In many cases, the bugs would never have been found by the programmer. When a person tests his own software, there's a small voice in the back of his head saying "don't try pressing **FCTN 9** when you're at this point". Something in his intimate knowledge of the source code makes him avoid that keystroke which will crash the software. The "other" user will of course try anything and everything that the software should be able to handle, and will thus find the bugs every time. Some years back, your author was purchasing a rather large and complex system for the Navy. Part of the hardware was delivered before the software was fully developed, and the customer quickly learned how to crash the program with a single keystroke. This went on for some time, until finally the programmers got the software to be "robust" enough that sitting on the keyboard wouldn't cause a problem. We knew the project was really finished when the only complaint the customer could come up with was that the little bulbs for the backlit function keys burned out too frequently. Then your author signed the final delivery form so the contractor could be paid his $5 million.

1.27.3. Finishing Touches

There's a tendency sometimes in our work on the TI to rush a project, especially when a deadline is nearing. This happens to us when we are trying to get a new product ready for either the Lima Faire or the Chicago Faire. There's an urgency that may make us forget some things like proper error traps, or even some of the "nice to have" features that should be added to a program even though they're not essential to the main function. Many times this also means skipping the step of having someone else try out the software.

Perhaps the most embarrassing example of this happened when we first introduced our Smart Connect disk. We rushed so much to have that ready for Chicago that we had only tested it with one of our three PC computers. Upon return from Chicago, we found customer complaints waiting for us. Sure enough, what we'd rushed to get ready would only work on that one of our three PCs. After some rather hectic re-working of the PC programs, we were able to get the program fixed, but this became a costly error, since we had to send revised disks to all those customers who'd bought the program at Chicago. Since then, we've had very few calls from customers who purchased that package, and in those cases we were able to resolve the problems over the phone.

It's late September of 1992 as we write this, and this year we've resigned ourselves to offering at Chicago only those products that have been thoroughly tested. There are about three or four projects in various stages of incompletion right now, but none of them is going to be "rushed" for Chicago. Maybe one or two will be ready for Lima in 1993, but we're making no commitments, even to ourselves.

Last month we promised a "single topic" column, and obviously we've missed the mark here, so we are making no promises for next month, except to try to make our column interesting for all our readers, and illuminating for those trying to make programs or routines in Assembly language.

```
* HISTRING/S
* AUX STRING STORAGE ROUTINE
* FOR HIGH MEMORY STRING STORAGE
* CODE BY BRUCE HARRISON
* FOR USE UNDER XB WITH ALSAVE
* 23 AUG 92
*
* EQUATES FOR XB UTILITIES, ETC.
VMBW   EQU  >2024        VDP MULTI-BYTE WRITE VECTOR
VSBW   EQU  >2020        VDP SINGLE-BYTE WRITE
VMBR   EQU  >202C        VDP MULTI-BYTE READ
STRASG EQU  >2010        STRING VARIABLE ASSIGNMENT VECTOR
STRREF EQU  >2014        STRING REFERENCE VECTOR
NUMREF EQU  >200C        NUMERIC VARIABLE REFERENCE VECTOR
NUMASG EQU  >2008        NUMERIC ASSIGNMENT VECTOR
XMLLNK EQU  >2018        XML LINKAGE VECTOR
CFI    EQU  >12B8        CONVERT FLOATING POINT TO INTEGER
CIF    EQU  >20          CONVERT INTEGER TO FLOATING POINT
FAC    EQU  >834A        FLOATING POINT ACCUMULATOR
KEYVAL EQU  >8375        KEY VALUE ADDRESS
KSCAN  EQU  >201C        KEYBOARD SCAN VECTOR
STATUS EQU  >837C        GPL STATUS BYTE
SCRWID EQU  32           SCREEN WIDTH (32 CHARACTERS)
GPLWS  EQU  >83E0        GPL WORKSPACE ADDRESS
FIRLO  EQU  >2002        FIRST AVAILABLE LOW MEM ADDRESS
LASLO  EQU  >2004        START OF DEF TABLE IN LOW MEM
LASHI  EQU  >8386        HIGHEST AVAIL ADDRESS IN HIGH MEM
*
* END EQUATES, BEGIN SUBROUTINE CODE
*
       DEF  SETHI,PUTHI,GETHI DEFINE ENTRY POINTS
       DEF  AVHI
*
*
SETHI
       LWPI WS           LOAD OUR OWN WORKSPACE
       CLR  R0           CLEAR R0 - NOT ARRAY
       LI   R1,1         POINT TO FIRST PARAMETER
       BLWP @NUMREF      GET THAT NUMBER
       BLWP @XMLLNK      USE XML LINKAGE
```

```
        DATA CFI           TO CONVERT TO INTEGER
        MOV  @FAC,@MAXNUM STASH AT MAXNUM LOCATION
        JGT  SET0A         IF GREATER THAN ZERO, JUMP
        B    @BADPRM       ELSE IF ZERO OR NEGATIVE, REPORT ERROR
SET0A   MOV  @FIRLO,R9     GET FIRST AVAIL ADDRESS LOW MEM
        ANDI R9,>FFFE      INSURE EVEN NUMBER
        INCT R9            POINT TO NEXT WORD
SET0    MOV  R9,@STTBL     STASH THAT ADDRESS AWAY
        MOV  @MAXNUM,R4    GET NUMBER IN R4
        SLA  R4,1          DOUBLE FOR WORD COUNT
        LI   R3,>A000      SET R3 TO >A000
OKAY0   MOV  R3,*R9+       PLACE R3 NUMBER AT R9 ADDRESS
        C    R9,@LASLO     COMPARE TO END OF LOW MEM
        JEQ  LOWERR        IF EQUAL, JUMP
        JGT  LOWERR        OR GREATER, JUMP
        INC  R3            INC NUMBER IN R3
        DECT R4            DECREMENT COUNT BY TWO
        JNE  OKAY0         IF NOT ZERO, OKAY
        MOV  R9,@ENDTBL    PLACE R9 AT ADDRESS ENDTBL
        MOV  @MAXNUM,R4    GET MAXNUM BACK
        MOV  R4,R10        PLACE IN R10
        LI   R9,>A000      LOAD R9 FIRST HI-MEM ADDRESS
        A    R9,R10        ADD TO R10
        C    R10,@LASHI    COMPARE TO HIGHEST HI ADDRESS AVAIL
        JLT  OKAY1         IF LESS, PROCEED
        B    @HIGHNG       ELSE ISSUE ERROR MESSAGE
LOWERR  B    @LOWNG        ISSUE LOW MEMORY ERROR
OKAY1
        MOVB R0,*R9+       MOVE A ZERO BYTE, INCREMENT POINTER
        DEC  R4            DEC COUNTER
        JNE  OKAY1         IF NOT ZERO, REPEAT
        MOV  R9,@ENDLST    PLACE R9 AT LIMIT ADDRESS
        B    @QEXIT        THEN EXIT ROUTINE
PUTHI
        LWPI WS            LOAD OUR WORKSPACE
        CLR  R0            CLEAR R0, NOT ARRAY
        LI   R1,1          POINT FIRST PARAMETER
        LI   R2,TEMSTR     POINT AT TEMPORARY STRING STORAGE
        MOVB @MAXLEN,*R2   SET FOR MAXIMUM LENGTH (255 BYTES)
        BLWP @STRREF       GET INCOMING STRING
        MOVB *R2,R3        GET LENGTH BYTE IN R3
        SRL  R3,8          RIGHT JUSTIFY
        INC  R1            POINT TO SECOND PARAMETER
        BLWP @NUMREF       GET THE NUMERIC VALUE
        BLWP @XMLLNK       USE XML LINKAGE
        DATA CFI           TO CONVERT NUMBER TO INTEGER
        MOV  @FAC,R4       MOVE RESULTING INTEGER TO R4
        JEQ  NOK4          IF ZERO, NOT VALID
        C    R4,@MAXNUM    COMPARE TO MAXIMUM NUMBER SET UP
        JLE  OKAY4         IF LOW OR EQUAL, OKAY
```

```
NOK4    B    @BADPRM       THEN ISSUE ERROR MESSAGE
OKAY4   DEC  R4            ZERO-BASE THE NUMBER
        SLA  R4,1          THEN DOUBLE TO INDEX BY WORDS
        MOV  @STTBL,R5     GET START OF ADDRESS TABLE IN R5
        A    R4,R5         R5 HAS TABLE ADDRESS
        MOV  *R5+,R6       R6 HAS ADDRESS PRESENT STRING
        MOV  R6,R12        STASH ADDRESS IN R12
        MOVB *R6+,R1       R6 POINTS TO CONTENT PRESENT STRING
        SRL  R1,8          R1 HAS PRESENT STRING LENGTH
        MOV  R3,R7         R7 HAS INCOMING STRING LENGTH
        S    R1,R7         R7 HAS DIFFERENCE INCOMING-PRESENT LENGTH
        JLT  OKAY5         IF A NEGATIVE NUMBER, JUMP
        JEQ  NOMOVE        IF ZERO, JUMP
        MOV  @ENDLST,R8    ELSE GET END OF STRING LIST IN R8
        A    R7,R8         ADD LENGTH DIFFERENCE
        C    R8,@LASHI     COMPARE TO MEMORY LIMIT
        JLT  OKAY6         IF LESS, PROCEED
        B    @HIGHNG       THEN ISSUE ERROR MESSAGE
OKAY5
        A    R1,R6         ADD LENGTH OF PRESENT STRING
        MOV  @ENDLST,R4    GET END OF STRING LIST INTO R4
        S    R6,R4         SUBTRACT ADDRESS
        MOV  R6,R9         SET MOVE SOURCE
        MOV  R9,R10        MOVE TO R10
        A    R7,R10        ADD LENGTH DIFFERENCE
MOVBT   MOVB *R9+,*R10+    MOVE ONE BYTE
        DEC  R4            DECREMENT COUNT
        JNE  MOVBT         IF NOT ZERO, REPEAT
        JMP  OKAY7         ELSE JUMP AHEAD
OKAY6
        MOV  @ENDLST,R4    GET END ADDRESS IN R4
        MOV  R4,R9         MOVE THAT TO R9
        MOV  R9,R10        AND R10
        S    R12,R4        SUBTRACT ADDRESS OF CURRENT STRING
        A    R7,R10        ADD LENGTH DIFFERENCE
MOVREV  MOVB *R9,*R10      MOVE ONE BYTE
        DEC  R9            DECREMENT SOURCE POINTER
        DEC  R10           AND DESTINATION POINTER
        DEC  R4            DECREMENT COUNT
        JNE  MOVREV        IF NOT ZERO, REPEAT
OKAY7
        A    R7,*R5+       ADD LENGTH DIFFERENTIAL TO NEXT TABLE ADDRESS
        C    R5,@ENDTBL    ARE WE FINISHED?
        JLT  OKAY7         IF NOT, REPEAT
        A    R7,@ENDLST    ADD LENGTH DIFFERENCE TO END OF STRING LIST

NOMOVE
        MOV  R12,R10       GET R12 INTO R10
        INC  R3            INC TO INCLUDE LENGTH BYTE ITSELF
MOVIN   MOVB *R2+,*R10+    MOVE ONE BYTE INTO STRING POSITION
```

```
       DEC  R3           DECREMENT COUNT
       JNE  MOVIN        IF NOT ZERO, REPEAT
       JMP  QEXIT        ELSE JUMP TO EXIT
AVHI
       LWPI WS           LOAD OUR WORKSPACE
       CLR  R0           NOT ARRAY VARIABLE
       MOV  @LASHI,@FAC  GET LAST AVAILABLE HIGH MEM ADDRESS INTO FAC
       S    @ENDLST,@FAC SUBTRACT END OF ARRAY
       BLWP @XMLLNK      USE XML LINKAGE
       DATA CIF          TO CONVERT INTEGER TO FLOATING POINT
       LI   R1,1         FIRST LINK PARAMETER
       BLWP @NUMASG      ASSIGN THE NUMBER
       JMP  QEXIT        THEN JUMP TO EXIT
GETHI
       LWPI WS           LOAD OUR WORKSPACE
       CLR  R0           CLEAR R0, NO ARRAY
       LI   R1,2         POINT TO SECOND PARAMETER
       BLWP @NUMREF      GET NUMBER
       BLWP @XMLLNK      USE XML
       DATA CFI          CONVERT TO INTEGER
       MOV  @FAC,R3      MOVE TO R3
       JEQ  NOK          IF ZERO, NOT VALID
       C    R3,@MAXNUM   COMPARE TO MAXIMUM NUMBER
       JLE  OKAY3        IF LOW OR EQUAL, OKAY
NOK
       B    @BADPRM      THEN ISSUE ERROR MESSAGE
OKAY3  DEC  R3           ZERO-BASE THE NUMBER
       SLA  R3,1         THEN DOUBLE FOR WORD INDEXING
       MOV  @STTBL,R2    GET START OF ADDRESS TABLE
       A    R3,R2        ADD INDEX NUMBER
       MOV  *R2,R2       POINT R2 AT STRING DESIRED
       DEC  R1           POINT TO FIRST PARAMETER
       BLWP @STRASG      ASSIGN THE STRING
QEXIT
       LWPI GPLWS        LOAD GPL WORKSPACE
       B    @>006A       RETURN TO GPL INTERPRETER
* END OF MAIN CODE SECTION
* FOLLOWING IS ERROR-HANDLING CODE
BADPRM
       BL   @CLR23       CLEAR ROW 23
       LI   R0,22*SCRWID+4 POINT AT ROW 23, COL 5
       LI   R9,OORSTR    MESSAGE ADDRESS IN R9
       BL   @DISLI       DISPLAY MESSAGE
       JMP  ERREX        JUMP TO ERROR EXIT
LOWNG
       BL   @CLR23       CLEAR ROW 23
       LI   R0,22*SCRWID+4 ROW 23, COL 5
       LI   R9,LMFSTR    MESSAGE ADDRESS
       BL   @DISLI       DISPLAY THAT
       JMP  ERREX        THEN JUMP
```

```
HIGHNG
       BL   @CLR23       CLEAR 23
       LI   R0,22*SCRWID+4 ROW 23, COL 5
       LI   R9,HMFSTR    MESSAGE
       BL   @DISLI       SHOW THAT
ERREX
       BL   @CLR24       CLEAR ROW 24 OF SCREEN
       LI   R0,23*SCRWID+3 POINT AT ROW 24, COLUMN 4
       LI   R9,PAKSTR    PRESS ANY KEY MESSAGE
       BL   @DISLI       DISPLAY THAT
ERRLOO CLR  @STATUS      CLEAR STATUS
       BLWP @KSCAN       SCAN KEYBOARD
       CB   @ANYKEY,@STATUS KEYSTRUCK?
       JNE  ERRLOO       IF NOT, SCAN AGAIN
       LI   R0,22*SCRWID ROW 23, COL 1
       LI   R1,BOTTOM    SAVED BOTTOM OF SCREEN
       LI   R2,2*SCRWID  TWO ROWS WORTH
       BLWP @VMBW        WRITE OLD STUFF BACK
       CLR  R0           NOT ARRAY VARIABLE
       LI   R1,3         THIRD PARAMETER
       BLWP @NUMASG      ASSIGN WHATEVER IS AT FAC
       JMP  QEXIT        THEN EXIT

* END OF ERROR-HANDLING CODE
*
* FOLLOWING SECTION ARE SUBROUTINES
CLR23
       LI   R0,22*SCRWID ROW 23, COL 1
       LI   R1,BOTTOM    DATA SAVING LOCATION
       LI   R2,2*SCRWID  TWO ROWS WORTH
       BLWP @VMBR        GET CURRENT CONTENT
       LI   R9,BLNKLN    32 SPACES
       JMP  DISLI        THEN JUMP
CLR24  LI   R0,23*SCRWID POINT AT ROW 24, COLUMN 1
       LI   R9,BLNKLN    32 SPACES
DISLI  LI   R10,SCRLI    POINT AT CHARACTER BUFFER
       MOV  R10,R1       MAKE R1=R10
       MOVB *R9+,R4      GET STRING LENGTH INTO R4
       SRL  R4,8         RIGHT JUSTIFY
       MOV  R4,R2        PLACE VALUE IN R2 ALSO
DIS1   MOVB *R9+,*R10    MOVE ONE BYTE OF STRING TO BUFFER
       AB   @OFFSET,*R10+  ADD >60 OFFSET
       DEC  R4           DECREMENT LENGTH COUNT
       JNE  DIS1         IF NOT ZERO, REPEAT
       BLWP @VMBW        ELSE USE VMBW TO DISPLAY STRING
DISLIX RT                RETURN
*
* END OF SUBROUTINE CODE
*
* FOLLOWING IS REQUIRED DATA FOR SUBROUTINE
```

```
*
WS     BSS  32           OUR OWN WORKSPACE
BOTTOM
TEMSTR BSS  256          STORAGE FOR INCOMING STRING
ENDLST DATA >A000        END OF STRING LIST
STTBL  DATA 0            START OF TABLE
ENDTBL DATA 0            END OF TABLE
MAXNUM DATA 0            MAX NUMBER OF STRINGS FROM SETUP
SCRLI  BSS  SCRWID       SCREEN LINE CHARACTER BUFFER
MAXLEN BYTE 255          MAX POSSIBLE STRING LENGTH
ANYKEY BYTE >20          HEX VALUE 20
OFFSET BYTE >60          XB CHARACTER OFFSET
LMFSTR BYTE 21
       TEXT 'NOT ENOUGH LOW MEMORY'
HMFSTR BYTE 22
       TEXT 'NOT ENOUGH HIGH MEMORY'
OORSTR BYTE 22
       TEXT 'PARAMETER OUT OF RANGE'
PAKSTR BYTE 25
       TEXT 'PRESS ANY KEY TO CONTINUE'
BLNKLN BYTE SCRWID
       TEXT '                                '
       END
```

1.28. The Art Of Assembly — Part 28. Never Do This!

By Bruce Harrison

There are a lot of things that could fit the title of today's column. The words come to us from an old friend named Chester Majewski. He would use a hammer to bang a car's battery terminal lug onto the post, all the while repeating "Never do this". This column could also be sub-titled "I can't believe I ever did that!"

Today's column was inspired by our going back into the source code for our Word Processor to make some improvements. The source code hadn't been worked on in about a year, and then only for relatively minor adjustments. This time the aim was to make some significant additions after correcting a bug. The bug is a story in itself, but let's tell that very briefly.

My partner Dolores was the chief "guinea pig" for the Word Processor, but rarely uses it now, preferring to do her writing on one of our PC computers. One day a couple of weeks ago she decided to use her TI and my WP to write a letter. "It Crashed!"

These dreaded words reached my ears, but I was busy in the next room, and couldn't investigate just then. Later, she told me she'd made several attempts to create a document, and each time the computer went bonkers. This started me thinking. I asked whether the disk she was using for the document had a name. Back when the WP was being developed, we used only the TI Disk Manager II module to initialize disks, and that won't let the user proceed without having entered a disk name. Now, we both use the DM-1000 program to work with disks, and Dolores almost never assigns a name to the disks she initializes.

A quick check of the source code showed that indeed the Word Processor program would crash if it encountered a disk whose name was a null string. In a couple of hours, the problem was fixed, tests were run, and Dolores had a new version of WP that would tolerate the "un-named" disk. Having thus nibbled on the edges of this mass of source code, we decided that more long-needed improvements should be tackled. Ugh! More long days and nights at the computer, more endless Assembler runs and testing sessions.

1.28.1. The Ugly Truth

There is nothing worse for an Assembly programmer than to have to go back into source code that was written a long time ago. We forget, over time, how crudely made the older stuff really was, and now see hundreds of places where what was done can be materially improved. Just the simple little things like setting a register as a word to a byte value taken from memory can easily be improved. In many places, it was like this:

```
CLR  R4                 Clear a register
MOVB @LABL,R4           Move the byte
SWPB R4                 Swap the bytes
```

It should be done like this:

```
MOVB @LABL,R4           Move the byte
SRL  R4,8               Right-justify in the register
```

That takes fewer bytes to accomplish the same job. Of course the Word Processor has thousands of lines of source code (about 150 pages of listing) and perhaps a hundred or so instances where the above could be done. If that were all, it would be easy to track them all down and make the change, but of course that's not all.

When we start looking at old source code, we find truly amazing things. As our programs are developed, we make changes all over the place to fix problems or to head off potential bugs. This process, with its extreme labor-intensive nature, leaves little time for re-checking parts of the code once they seem to work correctly. In a couple of instances, we found that whole processes were being performed twice, because a change had been made in a subroutine, and no checking had been done in the part of the code that called the subroutine. We found that some files were being closed in a subroutine, and then closed again in the main code. No harm was being done, but memory and execution time were being wasted.

There were other problems to be attacked. Most of the Word Processor's code was written while your author was a relative novice in Assembly language. The handling of file errors was done crudely, and some needed error traps for file handling just didn't exist. To make matters worse, the whole matter of handling file operations was scattered all over the place in various source files. Nesting of subroutines was an intricate maze, making it difficult to track down just what subroutines were being called, and from where. Of course the human memory is a truly amazing thing, so after a while the patterns of the original source begin to make some kind of sense, albeit a rather screwball kind of sense.

This sort of thing strikes one as similar to a Marx Brothers movie. The whole thing may be absurd, but there's some consistency to it, so that each ridiculous thing that happens fits into its own "universe".

Every once in a while when re-vamping this kind of source code, we are tempted to just start over with a clean sheet of paper. We never give in to that temptation, partly because we realize that doing so would mean maybe a year of effort just to get back to where we were when we started. That's impractical in the extreme. Thus we struggle with our old stuff, trying to make it work better, faster, and more efficiently.

The work proceeds, but very slowly. Each small change we make bears the risk of disaster. Thus as each small change is made in the source code, another process of assembling, loading, and testing must be performed. To speed things up, we keep all the main source code and the object file on our Horizon Ramdisk. This way, it takes only about ten minutes to assemble, but of course the testing can take much longer. We never can be quite sure whether a small "fix" has messed up something that used to work, so each time through the process we exercise all the functions of the program. Sometimes this seems a thankless task, but every now and then we make a discovery that helps our future work. One of those involves the use of the PIO port. We've found a way out of the situation when there's nothing attached to that port or when the printer is turned off. As you all know, in such situations the activity light on the RS-232 card comes on and stays on. What we found, though, is that the machine will still sense the **FCTN 4** keystroke, and this will cause our program to "escape" from servicing the PIO port. On exit, there is an error reported in the Peripheral Access Block in VDP RAM. Getting the byte at PAB+1 and shifting it right by 13 bits gives us an error value of 6. By placing an error trap in our code, we were able to use that **FCTN 4** keystroke as an emergency exit from a printing operation.

1.28.2. Error Traps Revisited

Back in number 8 of this series, we showed source code for making "plain English" error reports on the screen when file operations created the error. That code works very nicely, but in our playing around with the Word Processor, we decided to take advantage of a slightly more efficient way of doing the screen reporting. Today's Sidebar shows both the old and new ways. The difference in the new way is that, by making all the report texts the same length (16 bytes), and using the fact that 16 is an exact power of two, we could eliminate the lookup table used in our earlier method, and make the reporting more streamlined. This means of course that the error messages in some cases are a bit terse, but still better than "I/O ERROR 26". In the Word Processor, we supplemented this "bare bones" message with additional screen information like "PRINTING FILE DID NOT OPEN".

The method works like this. We set up the whole series of messages as sixteen character text lines with a label at the beginning of the first message. That message corresponds to the situation where the reported error code is zero, and we only know from the status register test that an error has happened. After reading the byte at PAB+1 into R1 with VSBR, we shift it right by 13 bits so that R1 contains the actual error code. Now to index into our table of messages, we shift the register left by four bits, effectively multiplying by sixteen. Adding the address of the beginning of the table of messages makes R1 point to the correct message for the error code. Now setting R2 to the common message length of 16 bytes, and R0 to the desired screen location sets us up to put the correct message on screen with VMBW. That's pretty simple, but effective. Of course if we wanted to apply the same method and make the messages longer, we'd have to make them all 32 bytes long and shift R1 left by five bits instead of four. That would chew up the memory we saved by not having a lookup table, and thus defeat the purpose of using this method in the first place.

Take your choice, as either method will produce excellent results, and will give the user a bit more information than some "error code" report, or the infamous "FATAL FILE ERROR". We always like to tease our friend Mike Maksimik about this last way of treating errors. It's one of the minor annoyances in Mike's MIDI-Master 99 that any file error gets reported on the screen as simply "FATAL FILE ERROR". We like to point out to Mike that at no time, so far as we know, has anyone died because of a file error on his TI. (At least we hope nobody has.)

We mentioned the business of a small fix causing a big problem, and here's an example. We went into the section of code called HOMIT, which allows the user to get to the top of a page with just a single keystroke (**FCTN H**). There was a problem when trying to use this on pages with only one screenful of text. We'll skip the details, but our first attempt at fixing this one little thing caused a really big problem that we didn't discover for several days. During a test run, we tried moving a paragraph from one part of a page up to the top, and used the **FCTN H** to put the cursor at the beginning of the page. When we then executed **FCTN 9** to complete the move, most of the page became scrambled. Words appeared to be scattered in all the wrong places. We got out of edit mode without saving the changes to this garbled mess, and went right back to the "drawing board" in the source code.

After some groping around, we found that we'd skipped a step, so that part of our program didn't know that the cursor was sitting at the top of the page, and thus the completion of the move operation was using the wrong starting point in the screen. We were able to correct the problem, so that "homing" the cursor no longer messed things up, but another lesson had to be learned yet again. Never assume that a quick fix will be okay until all the functions have been exercised.

1.28.3. A Fine Bowl Of Spaghetti

Sometimes we find that we've made a real mess of things even when everything works. The freedom that Assembly gives us can be its own pitfall. The source code for the Word Processor is much too big to be kept in one file, so it's "organized" into editable files from WORDA through WORDP. During its development, many subroutines were developed, and sometimes these were simply placed in whatever of the source files was the smallest, regardless of this subroutine's relationships to other parts of the program. That, plus the "nesting", can make tracing through some operations a real nightmare.

For example, the main code for getting into the edit mode is in the WORDG file. This calls a subroutine that's in WORDJ, then that subroutine calls one that's in WORDG, which in turn calls subroutines in WORDD, and so on. In programming parlance, this is called "spaghetti code", since it's all mixed up and twisted upon itself. If we did this in Extended Basic, where our users could list the program and see how messy it really is, we'd never release it. In Assembly, we can hide behind the fact that it all works correctly, and few of our users will try to disassemble the code once it's working.

As we write this, it's "take a break" time in our assembly work on the Word Processor. Jim Peterson has just issued another of those little "challenges" that we sometimes take up, so maybe we'll put the WP to bed again for a while, release Version 2.9, and start on Jim's challenge. Of course we already had about four other projects waiting in line before starting this round with the WP. Maybe we'll keep trying for Geneve compatibility, but then there's this game that our little boy wants me to create for him, and then again there's . . .

Now perhaps you see what we mean by "Never Do This!" Maybe that should apply to the whole idea of being a programmer.

```
* SIDEBAR 28
*
* FILE ERROR HANDLING
* THIS IS NOT COMPLETE CODE, JUST SNIPPETS
* FIRST, A SAMPLE FILE READING
*
*
* CODE BY B. HARRISON
* PUBLIC DOMAIN
*
* THIS OPENS AND READS A FILE RECORD
*

OPNF   LI   R0,PAB       POINT TO PAB IN VDP
       LI   R1,PABDT     AND PAB DATA
       LI   R2,25        25 BYTES TO WRITE
       BLWP @VMBW        WRITE PAB DATA
       AI   R0,9         ADD NINE
       MOV  R0,@>8356    STASH ADDRESS
       BLWP @DSRLNK      USE DSR LINKAGE
       DATA 8
       STST R14          STORE STATUS REGISTER
       ANDI R14,>2000    MASK ALL BUT TWO BIT
       JEQ  RDFI         IF ZERO, READ RECORD
GOERR  B    @OPNERR      ELSE REPORT OPEN ERROR
RDFI   MOVB @READF,R1    SET TO READ
       LI   R0,PAB       POINT AT PAB
       BLWP @VSBW        WRITE BYTE
       AI   R0,9         ADD NINE
       MOV  R0,@>8356    STASH ADDRESS
       BLWP @DSRLNK      USE DSR
       DATA 8
       LI   R0,PAB+1     POINT AT PAB PLUS ONE
       BLWP @VSBR        READ A BYTE
       SRL  R1,13        SHIFT RIGHT 13 BITS
       JEQ  READON       IF ZERO, MOVE ON
GOERR2 B    @FILERR      ELSE REPORT ERROR
READON (PROGRAM CONTINUES)
*
```

```
* NEXT SECTION IS "OLD" METHOD FOR REPORTING ERRORS
*
OPNERR LI   R0,22*32+3   POINT AT ROW 23, COL 4
       LI   R1,OPNMSG    "FILE NOT OPENED"
       BL   @DISSTR      DISPLAY STRING WITH SUBROUTINE
       LI   R0,PAB+1     POINT AT PAB PLUS ONE
       BLWP @VSBR        READ A BYTE
       SRL  R1,13        SHIFT RIGHT 13 BITS
FILERR SLA  R1,1         SHIFT LEFT ONE BIT (DOUBLE NUMBER)
       MOV  @LUT(R1),R1  GET MESSAGE ADDRESS INTO R1
       LI   R0,23*32+4   POINT AT ROW 24, COL 5
       BL   @DISSTR      DISPLAY SELECTED MESSAGE
       BL   @KEYLOO      STOP FOR A KEYPRESS
       B    (SOMEWHERE ELSE)
*
* DATA SECTION FOR "OLD" METHOD
*
PABDT  DATA >0014,BUF,>5050,>0000,>000F
       TEXT 'DSK1.ANYOLDFILE'
*
READF  BYTE 2
*
* ERROR MESSAGE STRINGS
*
BADDEV BYTE 15
       TEXT 'BAD DEVICE NAME'
WRPROT BYTE 15
       TEXT 'WRITE PROTECTED'
BADATT BYTE 13
       TEXT 'BAD ATTRIBUTE'
ILLOP  BYTE 17
       TEXT 'ILLEGAL OPERATION'
OUTSP  BYTE 19
       TEXT 'OUT OF BUFFER SPACE'
ENDFIL BYTE 11
       TEXT 'END OF FILE'
DEVERR BYTE 12
       TEXT 'DEVICE ERROR'
FILBAD BYTE 16
       TEXT 'OTHER FILE ERROR'
*
* LOOKUP TABLE
*
       EVEN
LUT    DATA BADDEV,WRPROT,BADATT,ILLOP
       DATA OUTSP,ENDFIL,DEVERR,FILBAD
OPNMSG BYTE 17
       TEXT 'FILE DID NOT OPEN'
*
* SECOND "NEW" WAY OF REPORTING FILE ERRORS
```

```
*
OPNERR
       LI   R0,22*32+6   POINT AT ROW 23, COL 7
       LI   R1,FNOTXT    FILE NOT OPENED
       LI   R2,17        17 BYTES TO WRITE
       BLWP @VMBW        WRITE MESSAGE
       LI   R0,PAB+1     POINT AT PAB PLUS ONE
       BLWP @VSBR        READ A BYTE
       SRL  R1,13        SHIFT RIGHT 13 BITS
FILERR SLA  R1,4         SHIFT LEFT FOUR BITS (MULTIPLY BY 16)
       AI   R1,FERMSG    ADD START OF MESSAGE TABLE
       LI   R2,16        16 BYTES IN EACH MESSAGE
       LI   R0,23*32+7   POINT AT ROW 24, COL 8
       BLWP @VMBW        WRITE MESSAGE
       BL   @KEYLOO      PAUSE FOR KEYSTROKE
       B    (SOMEWHERE ELSE)
*
* DATA SECTION FOR "NEW" METHOD
*
FERMSG TEXT 'BAD DEVICE NAME '  EACH TEXT LINE 16 BYTES
       TEXT 'WRITE PROTECTED '
       TEXT 'BAD ATTRIBUTE   '
       TEXT 'BAD OPERATION   '
       TEXT 'DISK IS FILLED  '
       TEXT 'END OF FILE     '
       TEXT 'DEVICE ERROR    '
       TEXT 'OTHER FILE ERROR'
FNOTXT TEXT 'FILE DID NOT OPEN' 17 BYTES LENGTH
```

1.29. The Art Of Assembly — Part 29. Pastafazool, etc.

By Bruce Harrison

This will be another departure from the main stream, and may stir up controversy among other language proponents, but after all this column is the "proponent" for Assembly, so we'll brag about our favorite language here with apologies to nobody.

Some time ago, we bought our first PC, which was a Tandy 1000-SX model. For those who follow the PC market, let's say right off that this is an old fashioned PC, with an Intel 8088 microprocessor, a mere 640 K memory capacity, and so on. By today's standards, a veritable dinosaur in that realm. One thing we wanted to find out about that machine was how fast it would do things that are important to us, like display things on the screen. We also wanted to explore the languages we had available on that machine, including Basic, Compiled Basic, and Assembly. To do this quickly, we made a group of "nonsense" programs that we called the "pastafazool" series. The purpose of these small programs was simply to measure speed of putting something on the screen. Each program in the pastafazool series does the same job, namely print the word "pastafazool" as many times as it can fit on the screen (6 per row times 24 rows, or 144 times on the PC's screen). The program would clear the screen first, of course, then fill and re-clear for a total of five cycles. (Five cycles gave us a measurable time span.)

Results were amazing in many ways. We found, for example, that compiled basic was much slower than interpreted basic at performing this task! That surprised us, because basic in compiled form is supposed to run faster. In Assembly, we had several choices of how to do the actual display operations, and the choices made a big difference. The slowest Assembly method was using the DOS Interrupt services. That performed only slightly faster than Interpreted Basic. The next best was using the BIOS Interrupt services. That was an improvement, but still not what we'd call "blazing" in speed. As on the TI, Assembly on the PC lets us do things in more ways, including the option of writing things directly to the memory section that serves the screen. (Segment address B800H, for you PC buffs.) This produced by far the fastest result, outstripping all other ways by a wide margin. Speed was measured using the PC's built in timer, and displayed on screen after the five repeats of filling the screen with "pastafazool" were finished.

To summarize, the results went like this:

Compiled Basic	20 seconds
Interpreted Basic	9 seconds
DOS service Assembly	7 seconds
Direct-to-screen Assembly	<1 second

This last is given as <1, since the time printed on the screen would sometimes report as one second and sometimes as zero second, depending how the timing of start and finish happened to catch the phase of the timer.

Okay, so what does all this have to do with the TI, you ask? We wanted to try a similar experiment on the TI, and did so. Of course we did not have so many options open, since there's no compiler for TI Basic (We used XB in the test) and there's no direct equivalent of either the DOS Interrupts or the "Direct to screen" capability. In today's Sidebar are two things. A listing of the very simple Extended Basic program that writes "pastafazool, pastafazool" on each row of the screen five times, and two Assembly source code files that do the same thing. The one called PASTAB/S uses the VSBW utility vector, and the one called PASTA/S uses the VMBW vector.

The Extended Basic version is written as a three-line program, with all the action in line 10. Line 1 simply clears the VDP Interrupt Timer. Line 12 reads that timer, then prints the count with 255 added because the counter will have "rolled over" once, divided by 60 to report the time elapsed in seconds. This is listed in 28 column lines just as it would appear on the screen. The two Assembly versions are shown as annotated source code, so you should be able to easily follow their operation. Given what's in today's Sidebar, you should be able to conduct your own tests at home. Our results went as follows:

Extended Basic	6.4 seconds
VSBW Assembly	0.65 second
VMBW Assembly	0.2 second

Timing of the two Assembly versions is done using the VDP Interrupt counter at >8378. This yields a number in 60ths of a second, which is displayed on the screen after the five cycles are finished. Both versions then use a simple key scan loop to allow you to read the number. Pressing any key gets you back to Editor/Assembler. The VSBW version reported between 39 and 40 60ths on the screen, or about 0.65 seconds. For the VMBW version, time for execution ranged from 10 to 14 60ths, so we've taken that as averaged to 12 60ths, or 0.2 seconds. That's a 32:1 speed ratio between the slowest and fastest. You may notice that we've used an "undocumented" feature of GPLLNK to convert the count number to a string for screen display. Our thanks again to Merle Vogt for that little gem.

Using the VDP Interrupt counter in this way introduces a kind of "Heisenberg uncertainty" into our measurement, since the program has to execute LIMI 2 and LIMI 0 to allow the timer to function. That's done after each row is written, but performing those actions eats time in itself, so we can't really discover how fast the operation would complete without the LIMI operations being performed. We tried it without, and it doesn't appear to seriously hamper the speed, but we can never be sure. Heisenberg applied his principle in the field of particle physics, but the idea works here as well, in that the things we do to measure something affect the thing we're trying to measure, so we can never be sure what the case would be without our measurement.

Now to change subjects again, we'll go back to an old theme, namely things that are just plain wrong in TI's Assembly book. Back in the December 1992 issue of this magazine, there was a reader-to-reader plea from Mr. Ian J. Howle concerning use of the Bitmap Mode on the TI's VDP. We sent Mr. Howle a short piece of source code that corrected some of the errors induced by his reading the book, so that he'd at least have a non-crashing starting point. The trouble is that, while what we sent to him won't crash, it won't seem to do what the book indicates it should do.

Before we go on, please understand that there are things we don't use on our TI, and Bitmap Mode is one of those. If it did work just as given in the book, perhaps we'd consider using it, but the amount of VDP space that it requires is so great, and the amount of code and data required to manipulate the screen display in that mode is also daunting. In our experimenting with it, we tried many variations on the theme as to where the screen image table, color table, and character table should be placed. Results varied, but the overall pattern was that the definitions we put in the character table for characters 0 through 7 would simply repeat for the rest of the characters 8 through 255, in eight character groups. That is, 8 through 15 would have the same pattern as 0 through 7, 16 through 23 would repeat those patterns, and so on. Of course we know that other programmers have used Bitmap Mode with success, so there has to be an answer, but it's most certainly not what's given in the E/A manual.

It has sometimes occurred to us that we should "document" all the known errors in the big book, but that's really a huge job, and our time keeps getting eaten up by creating new products from what we know how to use, leaving no time for the arduous task of documenting errors found. There are others in the "community" who have no doubt found errors and found corrections to make things work. Harry Wilhelm's name comes to mind, as one who not only found out how to use Bitmap Mode, but also how to make it accessible from Extended Basic, complete with automatic sprite motion, which the book implies can't be done. Harry probably also knows how to use half-bit mode, whatever that is. (One of our customers asked us about that, and we had to plead ignorance about the whole subject.)

Here then is another plea to our readers, many of whom have written to us. If you've discovered a case where the book is wrong, and a solution to make something work, pass it along to us at 5705 40th Place, Hyattsville MD 20781, and we'll try to work it into a future column. In past columns, we have passed along some "undocumented" features and some "trade secrets" of our own, but we are certainly not the ultimate source of such information, as our knowledge is limited to what we've tried doing. Even at that, we don't always succeed in "breaking the code" to make certain functions work. (e.g. Bitmap Mode)

Once again we are not sure about next month's topic. We are always working on little projects of one kind or another, and that is where the material for these columns comes from. The mysteries of the TI are still many, even with all the probing done by its users over the years. There's a rumor about that somebody has obtained the source code for Extended Basic, and that may make more of TI's "trade secrets" accessible. The whole topic of GPL language is gradually being unraveled, and that's probably good, even though some of us have no plans to write in that language. At least there are practitioners exploiting that realm now, and giving still more power to the users of our beloved orphan.

One more tidbit before we close. The last few of these columns were written on a laptop PC instead of on our TI. That's not because we're losing faith in out TI, but just because the Ramdisk on the TI was filled up with source code for one of our current projects, and the PC had plenty of disk space available.

```
* SIDEBAR 29
* FIRST, THE XB PROGRAM PASTABAS
* LISTED IN 28 COLUMNS
*
*
1 CALL INIT :: CALL LOAD(-31
879,0)
10 FOR J=1 TO 5 :: CALL CLEA
R :: FOR I=1 TO 24 :: DISPLA
Y AT(I,3):"PASTAFAZOOL, PAST
AFAZOOL" :: NEXT I :: NEXT J
12 CALL PEEK(-31879,A):: PRI
NT (A+255)/60
*
*
* NEXT, THE SLOWER ASSEMBLY VERSION
*
* PASTAB/S - SOURCE FOR VSBW VERSION
* CODE BY B. HARRISON
* PUBLIC DOMAIN
*
       DEF  START2       DEFINE ENTRY
       REF  VSBW,VMBW,KSCAN,GPLLNK
START2
       LWPI >20BA        LOAD USER WS
       CLR  @>8378       CLEAR VDP INT COUNTER
       LI   R4,5         5 SCREENS
NEXSCR
       LI   R2,768       768 CHARS
       LI   R1,>2000     SPACE CHAR
       CLR  R0           SCREEN ORIGIN
CLRLP
       BLWP @VSBW        WRITE ONE SPACE
       INC  R0           NEXT ADDRESS
       DEC  R2           DEC COUNT
       JNE  CLRLP        IF NOT 0, RPT
       LI   R0,4         ROW 1, COL 5
       LI   R3,24        24 ROWS
PRLIN  LI   R5,PASTA     POINT AT STRING
       MOVB *R5+,R2      GET LENGTH
       SRL  R2,8         MAKE WORD
       MOV  R2,R6        SAVE LENGTH
PRCHR  MOVB *R5+,R1      GET ONE BYTE
       BLWP @VSBW        WRITE THAT
       INC  R0           INC SCRN ADDR
       DEC  R2           DEC COUNT
       JNE  PRCHR        IF NOT 0, RPT
       AI   R0,32        ADD ROW LENGTH
       S    R6,R0        SUB STRING LEN
       LIMI 2            ALLOW INTERRUPTS
```

```
       LIMI 0            THEN STOP THEM
       DEC  R3           DEC ROW COUNT
       JNE  PRLIN        IF NOT 0, RPT
       DEC  R4           DEC SCRN COUNT
       JNE  NEXSCR       IF NOT 0, RPT
       MOV  @>8378,R10   GET VDP INTERRUPT COUNT
       CLR  @>837C       CLEAR GPL STATUS BYTE
       BLWP @GPLLNK      USE GPL LINK
       DATA >4D00        TO SCROLL SCREEN
       AI   R0,-32       MOVE BACK TO ROW 24
       MOV  R10,@>835E   MOVE TIME COUNT TO >835E
       CLR  @>837C       CLEAR GPL STATUS
       BLWP @GPLLNK      USE GPL LINK
       DATA >2F7C        TO CONVERT INTEGER TO STRING
       MOVB @>8361,R2    GET STRING LENGTH
       SRL  R2,8         RIGHT JUSTIFY
       MOVB @>8367,R1    GET STRING ADDRESS LOW BYTE
       SRL  R1,8         RIGHT JUSTIFY
       AI   R1,>8300     ADD HIGH BYTE
       BLWP @VMBW        WRITE "TIME" STRING TO SCREEN
SCAN   BLWP @KSCAN       SCAN KEYBOARD
       CB   @ANYKEY,@>837C KEY STRUCK?
       JNE  SCAN         IF NOT, REPEAT
       LWPI >83E0        LOAD GPL WS
       B    @>6A         RETURN TO GPL
PASTA  BYTE 24
       TEXT 'PASTAFAZOOL, PASTAFAZOOL'
ANYKEY BYTE >20
       END
*
* LAST, THE FASTER ASSEMBLY VERSION
*
* PASTA/S - SOURCE FILE FOR VMBW VERSION
* CODE BY B. HARRISON
* PUBLIC DOMAIN
*
       DEF  START        DEFINE ENTRY
       REF  VMBW,KSCAN,GPLLNK
START
       LWPI >20BA        LOAD USER WS
       CLR  @>8378       CLEAR VDP INT COUNTER
       LI   R4,5         COUNT IN R4
NEXSCR
       LI   R3,24        24 ROWS
       LI   R2,32        32 COLUMNS
       LI   R1,BLNKLN    32 SPACES
       CLR  R0           SCREEN ORIGIN
CLRLP
       BLWP @VMBW        WRITE 32 SPACES
       A    R2,R0        ADD 32
```

```
       DEC  R3           DEC ROW COUNT
       JNE  CLRLP        IF NOT 0, RPT
       LI   R0,4         ROW 1, COL 5
       LI   R3,24        24 ROWS
PRLIN  LI   R1,PASTA     TEXT STRING
       MOVB *R1+,R2      GET LENGTH
       SRL  R2,8         MAKE A WORD
       BLWP @VMBW        WRITE STRING
       AI   R0,32        DOWN 1 ROW
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN STOP THEM
       DEC  R3           DEC ROW COUNT
       JNE  PRLIN        IF NOT 0, RPT
       DEC  R4           DEC SCREEN COUNT
       JNE  NEXSCR       IF NOT 0, RPT
       MOV  @>8378,R10   GET VDP INTERRUPT COUNTER
       CLR  @>837C       CLEAR GPL STATUS
       BLWP @GPLLNK      USE GPL LINK
       DATA >4D00        TO SCROLL SCREEN
       AI   R0,-32       MOVE BACK TO ROW 24
       MOV  R10,@>835E   STASH TIME COUNT AT >835E
       CLR  @>837C       CLEAR GPL STATUS
       BLWP @GPLLNK      USE GPLLNK
       DATA >2F7C        TO COVERT INTEGER TO STRING
       MOVB @>8361,R2    GET STRING LENGTH
       SRL  R2,8         RIGHT JUSTIFY
       MOVB @>8367,R1    GET STRING ADDRESS LOW BYTE
       SRL  R1,8         RIGHT JUSTIFY
       AI   R1,>8300     ADD HIGH BYTE
       BLWP @VMBW        WRITE STRING TO SCREEN
SCAN   BLWP @KSCAN       SCAN KEYBOARD
       CB   @ANYKEY,@>837C HAS A KEY BEEN STRUCK?
       JNE  SCAN         IF NOT, RPT
       LWPI >83E0        LOAD GPL WS
       B    @>6A         RETURN TO GPL
BLNKLN TEXT '                              '
PASTA  BYTE 24
       TEXT 'PASTAFAZOOL, PASTAFAZOOL'
ANYKEY BYTE >20
       END
```

1.30. The Art Of Assembly — Part 30. Using What's There

By Bruce Harrison

Today's topic is using what's there, which means using the routines that come with your TI instead of inventing your own. The routines provided include some that are "documented" and some that are not. Today we'll concentrate most of our attention on routines that work with floating point numbers.

1.30.1. Floating Point Numbers

We've said before in this column that we use integer math whenever possible to speed up the execution time in our programs. There are cases, however, where nothing but floating point numbers can do the job. In a recent case, we were doing some work on a "tool" for users of Mike Maksimik's MIDI-Master. The job involved taking numbers from a file in the form of ASCII strings, multiplying or dividing by a user-input number such as 2.5, then writing the numbers back as strings into an output file record. (Portions of the source code are shown in the Sidebar.)

The numbers themselves are all integers to start with, and if the divisor or multiplier were always 2.5, we could have made the calculations in integer math. However, where the user could input numbers of any kind, such as 2.3742, the problem of handling things as integers would get very complex indeed. Thus we had an opportunity to take advantage of the routines provided by TI right inside the computer.

In the TI, floating point numbers are represented by eight byte blocks in Radix 100 notation. Radix what, you ask? There is an explanation in the Editor/Assembler manual, but it's about as clear as mud, so we'll try to explain in more simple terms. The Radix 100 notation uses an offset, or multiplier, in powers of 100. This multiplier occupies one byte. That leaves seven bytes for the number itself. Each byte of these seven can take on values from 0 through 99, thus each byte represents two digits of our common decimal number notation. That allows fourteen significant digits of accuracy in all the TI's floating point number operations.

The "multiplier" is biased by >40 (64 decimal), so that the multiplier can provide both positive and negative powers of 100. In other words, 100 raised to the zero power makes the multiplier byte equal >40. Perhaps an example or two would help. Here are a couple:

Decimal number bytes in FP format

1.0000	>40,>01,>00,>00,>00,>00,>00,>00
0.5000	>3F,>32,>00,>00,>00,>00,>00,>00
105.2	>41,>01,>05,>14,>00,>00,>00,>00
89,999.999	>42,>08,>63,>63,>63,>5A,>00,>00

We might find this easier to see if the bytes themselves were expressed in decimal notation instead of hex. These four would look like this:

1.0000	64,1,0,0,0,0,0,0
0.5000	63,50,0,0,0,0,0,0
105.2	65,1,5,20,0,0,0,0
89,999.999	66,8,99,99,99,90,0,0

Still clear as mud? Then think of it like scientific notation. In that notation, 8.9999999E+4 would represent the last number shown above, where the number after the E is the power of ten that multiplies the number before the E. In this case 10 raised to the fourth power, or 10000. In the Radix 100 case, the number used as an exponent is a power of 100, so that 10000 is represented by 2, not four. (10000 is 100 raised to the second power.)

We could go on like this all day and all night, but if you've got the idea by now, you can see the rest yourself. If you haven't got it by now, thirty more pages might not make it clear. In almost every case that you'll encounter, the problem of creating numbers in Radix 100 notation can be performed for you by the TI's internal routines anyway, so you won't need to understand it. (Aren't you glad we didn't discuss negative numbers in Radix 100?)

Perhaps we should start with a routine that allows the TI to make the floating point number for you, so you need not wrestle with the notation yourself. Suppose you have an integer in the range of 32767 through -32768. This could be a word in memory, for example. Converting to floating point format is simple. Place the word at FAC, which is at >834A. (FAC is an abbreviation for Floating Point ACcumulator.)

Once the number is at FAC, XMLLNK can be used to convert the integer to a floating point number by:

```
BLWP @XMLLNK
DATA >2300
```

This will convert the number at FAC (one word) into a floating point number, with the sign as appropriate. That is, numbers from 0 through >7FFF (0 through +32767) will become positive floating point numbers, while numbers from >FFFF through >8000 (-1 through -32768) will become negative floating point numbers. As we'll see later, we can in some circumstances either ignore or not ignore the sign of integer numbers. In a test we just ran, for example, we tried two different ways to convert an integer into a string on the screen, and one of those paid heed to the sign of the number, while another did not. Whoops, we're getting ahead of our story. We must stop doing that!

When we are going to deal in floating point numbers, we often put some equates into the beginning of our source code to give us mnemonic labels, so that it will be easier to understand what we're doing. A typical set of equates would look like this:

```
FAC         EQU >834A         Floating point accumulator
ARG         EQU >835C         Floating point argument
CIF         EQU >2300         Convert Integer to floating point
CFI         EQU >1200         Convert Floating point to Integer
CNS         EQU >0014         Convert number (FP) to string
CSN         EQU >1000         Convert string to number (FP)
FADD        EQU >0600         Floating point addition
FSUB        EQU >0700         Floating point subtraction
FMUL        EQU >0800         Floating point multiplication
FDIV        EQU >0900         Floating point division
FCOM        EQU >0A00         Floating point compare
INT         EQU >0022         Compute greatest integer
```

Of these, all except CNS and INT are accessed through XMLLNK. CNS and INT are accessed with GPLLNK. When using the floating point math routines, such as FADD and FMUL, the two floating point numbers to be added or multiplied are first placed at FAC and ARG. Each must be a valid eight byte floating point number. After any of the math operations, the result is a number at FAC. For adding and multiplying, it's of no importance which of the numbers is at FAC and which at ARG before the operation, except of course if one must keep one of those numbers handy, in which case ARG would be the proper place for that number, since FAC gets modified by the operation. For division or subtraction, it's important to know which number goes at which location. For subtraction, FAC is the number to be subtracted from ARG, with the result placed at FAC. For division, FAC is the divisor of the number at ARG, and again FAC gets the result. Put another way, for subtraction, we could write that FAC=ARG-FAC, while for division FAC=ARG/FAC.

In various of our projects over the years, we have used most of these utility routines. (The only one we can't remember using is FSUB.) They all work, and quite well. In one case that's memorable, we did something that really proved to us the incredible accuracy of the TI's floating point routines.

1.30.2. Amazing Accuracy

Once, we were creating a little utility for XB users, designed to place musical note values into an array variable. The XB program would give us a starting note value and the number of octaves to fill in the array, then we would take that starting number and multiply it repeatedly by the twelfth root of two to make all the half-tone steps. The twelfth root of two is the number 1.059463094. We expressed that number in Radix 100 notation in our source file with ten significant digits. What happens with repeated multiplication by this number is that every twelfth time, the number doubles. (e.g. 110 multiplied by the twelfth root of two twelve times equals 220, or 24 times equals 440, and so on.) We found that, using the TI's FMUL routine, making a seven-octave run, or multiplying 84 times in succession, then rounding the result, the result was exactly correct. Of course without the rounding, there is some cumulative error in the last couple of decimal places, but you can use the TI's own utilities to get results more accurate than most PC basic programs would yield. The rounding was done by adding .5 to the result number, then using our own "int" function, which worked faster than the function provided by the "greatest integer" function with GPLLNK. Just looking at raw numbers, without rounding, the PC basic on our trusty laptop gave a result of 14,080.05 after a seven-octave run, while the TI gave the result as 14,079.9996, which is a much more accurate number, off by only .0004 from the exact expected result.

1.30.3. How To Use Them

In today's Sidebar are a couple of small examples taken from some of our recent work on tools for MIDI-Master. Note that in these cases we didn't have to create any floating point numbers ourselves, except for the value 0.5 used in rounding, but just used TI's routines. Taking numbers from the screen or from anywhere in VDP memory was very easy using the CSN routine. We provide a pointer to the location of the string in VDP RAM, and let the routine do the work of turning that into a floating point number. This routine reads the number from VDP, and stops when it finds a space or some other character that's not a valid numeric character. If the string presented is not a valid number, the routine returns 0 at location FAC, and this can be used as an indication that no valid number was present, unless of course 0 is a valid entry anyway. In the example shown for a user input number, 0 was not a valid entry, so finding that FAC = 0 would cause our program to replace the default string in the input field and make the user re-enter either the default or some other valid number.

This conversion routine will also work correctly if the string is a "scientific notation" number, such as 9.00E3, which will yield 9000 as a floating point number. It works equally well for negative numbers, and so on.

The companion service, convert number to string, also comes in awfully handy. This gives us a really quick way to put floating point numbers on-screen as strings, without having to develop any such routines for ourselves. One small caution must be observed about the strings created by CNS. This service reserves a leading space for the minus sign, and thus for positive numbers the string will have a space in its first character. In the example shown, we were working with numbers that were always positive, and MIDI-Master would balk if there were a leading space before the number, so we simply incremented our pointer before displaying the string, so as to eliminate the space. Doing this can also perform a kind of "absolute value" function, by simply ignoring the minus sign if it's there as the first string character.

Integer numbers can also be converted into strings, and in two ways. The integer can first be placed at FAC and then converted to FP, and then to a string. This will produce a numeric string that recognizes the sign of the integer you started with, and reports strings as 0 through 32767 or -1 through -32768. You can get the computer to ignore the sign and produce a string ranging from 0 through 65535 by using the undocumented feature provided us by Merle Vogt. For this, the integer is placed at >835E, then GPLLNK is called to make the conversion as shown in the Sidebar. In this case, the string is always 0 or positive, and there's no leading space to contend with.

For those interested, the details for use of these routines is on pages 254 through 261 in the E/A manual. There is a warning provided on page 261 concerning the use of the CIF routine when working from an Extended Basic environment. This warning can safely be ignored under some circumstances. We have been able to use these routines safely even when an Option 5 Program File was loaded and run from Extended Basic, provided only that we dumped the E/A utilities starting at >2000 and running through >23BA into low memory. (We covered how to do this in an earlier column in this series.) The fact that this all worked was one of those pleasant little surprises that come along every now and then. Another example for the "we learn something every day" category.

We've gone on at some length about this topic, but still have the feeling of having just started. Perhaps we'll continue this subject in a future article.

```
*  SIDEBAR 30
*
*  SOME EXAMPLES OF USING EXISTING
*  CONSOLE ROUTINES
*  PUBLIC DOMAIN
*
*  CODE BY B. HARRISON EXCEPT WHERE NOTED
*
*
*  FIRST, SOME EQUATES USED
*
*
CNS    EQU  >0014   CONVERT NUMBER TO STRING (W/GPLLNK)
CFI    EQU  >1200   CONVERT FLOATING POINT TO INTEGER
FAC    EQU  >834A   FLOATING POINT ACCUMULATOR
ARG    EQU  >835C   FLOATING POINT ARGUMENT
CSN    EQU  >1000   CONVERT STRING TO NUMBER (W/XMLLNK)
INT    EQU  >0022   FIND LARGEST INTEGER IN F.P. NUMBER (W/GPLLNK)
FADD   EQU  >0600   FLOATING POINT ADDITION (XMLLNK)
FMUL   EQU  >0800   FLOATING POINT MULTIPLY (XMLLNK)
FDIV   EQU  >0900   FLOATING POINT DIVIDE   (XMLLNK)
FAC11  EQU  >8355   LSB ADDRESS FOR CNS
FAC12  EQU  >8356   STRING LEN FOR CNS
*
*  FOLLOWING DISPLAYS A NUMBER USING AN
*  UNDOCUMENTED FEATURE WITH GPLLNK
*  BY CONVERTING THE INTEGER PLACED AT
*  >835E INTO A STRING, WHICH IS THEN
*  DISPLAYED AT ROW 12, COLUMN 7
*  THE NUMBER IS DISPLAYED AS AN UNSIGNED
*  INTEGER FROM 0 THROUGH 65535
*  THANKS TO MERLE VOGT FOR THIS ONE!
*
*
       LI   R0,11*32+6   SET R0, ROW 12, COL 7
       MOV  @NUMBER,@>835E  PLACE NUMBER AT >835E
       CLR  @>837C       CLEAR GPL STATUS BYTE
       BLWP @GPLLNK      USE GPLLNK
       DATA >2F7C        TO CONVERT INTEGER TO STRING (UNDOCUMENTED)
       MOVB @>8361,R2    GET STRING LENGTH
       SRL  R2,8         RIGHT-JUSTIFY
       MOVB @>8367,R1    GET LOW BYTE OF ADDRESS
       SRL  R1,8         RIGHT-JUSTIFY
       AI   R1,>8300     ADD >8300 HIGH BYTE
       BLWP @VMBW        WRITE STRING TO SCREEN
*
*
```

```
*  FOLLOWING GETS A NUMBER FROM A DISPLAYED STRING,
*  THEN CONVERTS THAT TO AN INTEGER IN R5
*  (ON ENTRY, R0 POINTS TO NUMBER'S SCREEN LOCATION)
*  (ON EXIT, INTEGER VALUE IS IN R5 AND AT FAC)
*
GETNUM
       MOV  R0,@FAC12    PLACE SCREEN ADDRESS AT >8356
       BLWP @XMLLNK      USE XML LINKAGE
       DATA CSN          TO CONVERT STRING TO NUMBER
*  AT THIS POINT, FAC CONTAINS NUMBER IN FLOATING POINT
*  FORMAT, AS EIGHT BYTES IN RADIX 100 NOTATION
       BLWP @XMLLNK      USE XML AGAIN
       DATA CFI          TO CONVERT FLOATING POINT TO INTEGER
       MOV  @FAC,R5      MOVE INTEGER TO R5
       RT                RETURN
*
*
*  ANOTHER WAY TO DISPLAY A NUMBER
*  THIS ONE DISPLAYS A FLOATING POINT NUMBER
*  BY CONVERTING IT TO A STRING
*
*
       LI   R0,11*32+6   POINT AT ROW 12, COL 7
       MOVB @ZERO,@FAC11 PLACE A ZERO AT >8355
       CLR  @STATUS      CLEAR THE GPL STATUS BYTE
       BLWP @GPLLNK      USE GPL LINKAGE
       DATA CNS          TO CONVERT NUMBER AT FAC TO STRING
       MOVB @FAC12,R2    GET STRING LENGTH
       SRL  R2,8         RIGHT JUSTIFY
       MOVB @FAC11,R1    GET LOW BYTE OF STRING ADDRESS
       SRL  R1,8         RIGHT JUSTIFY
       AI   R1,>8300     ADD >8300 AS HIGH BYTE
*  FOLLOWING TWO LINES ARE USED IF
*  WE WANT TO IGNORE THE SIGN OF THE NUMBER
*  WITHOUT THESE TWO LINES, POSITIVE NUMBERS
*  ARE DISPLAYED WITH A LEADING SPACE,
*  NEGATIVE NUMBERS WITH A MINUS SIGN
       INC  R1           POINT TO 2ND CHARACTER IN STRING
       DEC  R2           DECREMENT LENGTH BY ONE
       BLWP @VMBW        DISPLAY THE STRING
*
*
* THE FOLLOWING ARE EXCERPTS FROM "TOOL4"
* AN ASSEMBLY PROGRAM THAT PROVIDED "DELAY CONVERSION"
* FOR MIDI-MASTER MUSIC SOURCE FILES
*
*
*  FIRST PART PLACES A DEFAULT CONVERSION FACTOR (2.5)
*  ON THE SCREEN, THEN ACCEPTS A USER INPUT
*
```

```
*
       LI   R0,8*32+20   POINT AT ROW 9, COL 21
CHO3   LI   R1,T5STR     POINT AT DEFAULT NUMBER (2.5)
       BL   @DISSTR      DISPLAY THAT
       LI   R4,6         SIX CHARACTERS MAX
       BL   @CRSIN       ACCEPT INPUT
       MOV  R0,@FAC12    MOVE START ADDRESS TO >8356
       BLWP @XMLLNK      USE XML LINK
       DATA CSN          TO CONVERT STRING TO FLOATING POINT NUMBER
       MOV  @FAC,R4      MOVE THE WORD AT FAC
       JEQ  CHO3         IF THAT'S ZERO, INVALID ENTRY
       LI   R9,FAC       ELSE POINT TO FAC
       LI   R10,TWOPT5   AND TO STORAGE PLACE
       LI   R4,8         EIGHT BYTES
       BL   @MOVBTS      MOVE THAT FP NUMBER TO STORAGE
*
* THE PROGRAM READS THE MUSIC SOURCE FILE INTO MEMORY
*
* THE PROGRAM IS WRITING THE FILE FROM MEMORY BACK TO DISK
* WHERE WE RE-JOIN THE ACTION
*
* AT THIS POINT, WE'VE FOUND A FILE RECORD THAT
* CONSISTS OF "#DELAY=" PLUS A NUMBER
* AND PLACED THAT RECORD AT LABEL TEMSTR
* THE PROGRAM WILL EITHER MULTIPLY OR DIVIDE
* THE NUMBER AFTER "#DELAY=" BY THE USER INPUT
* OBTAINED IN THE PREVIOUS "SNIPPET"
*
       LI   R0,21*32+8   POINT AT ROW 22, COL 9
       LI   R4,24        24 CHARACTER FIELD
       BL   @CLRFLD      CLEAR THAT FIELD
       LI   R1,TEMSTR    POINT AT TEMPORARY STRING
       BL   @DISSTR      DISPLAY THE STRING
       AI   R0,7         ADD 7 TO POINT AT NUMERIC PART
       MOV  R0,@FAC12    PLACE THAT ADDRESS AT >8356
       BLWP @XMLLNK      USE XML LINKAGE
       DATA CSN          TO CONVERT STRING TO F.P. NUMBER
       LI   R9,FAC       POINT AT NUMBER IN FAC
       LI   R10,ARG      AND AT ARG LOCATION
       LI   R4,8         EIGHT BYTES TO MOVE
       BL   @MOVBTS      MOVE F.P. NUMBER FROM FAC TO ARG
       LI   R9,TWOPT5    POINT AT USER'S INPUT CORRECTION NUMBER
       LI   R10,FAC      AND AT FAC
       LI   R4,8         EIGHT BYTES
       BL   @MOVBTS      PLACE USER'S NUMBER AT FAC
       MOV  @BFLG,R4     CHECK FOR MULT OR DIV
       JEQ  DIV          IF ZERO, DIVIDE
*
* BFLG SIMPLY INDICATES WHETHER A MULTIPLY OR DIVIDE IS NEEDED
* DEPENDING ON AN EARLIER USER INPUT (NOT SHOWN)
```

```
*
       BLWP @XMLLNK      ELSE USE XML
       DATA FMUL         TO MULTIPLY
       JMP  RNDOFF       THEN JUMP AHEAD
DIV    BLWP @XMLLNK      USE XML
       DATA FDIV         TO DIVIDE ARG BY FAC
*
* CODE STARTING AT RNDOFF CORRECTLY ROUNDS THE RESULTING NUMBER
* BY FIRST ADDING 0.5, THEN TAKING THE INT OF THAT NUMBER
*
RNDOFF
       LI   R9,FPHALF    POINT AT FLOATING POINT NUMBER 0.5
       LI   R10,ARG      AND AT ARG
       LI   R4,8         EIGHT BYTES
       BL   @MOVBTS      MOVE 0.5 F.P. INTO ARG
       BLWP @XMLLNK      USE XML
       DATA FADD         TO ADD .5 TO RESULT OF ABOVE
       CLR  @STATUS      CLEAR GPL STATUS BYTE
       BLWP @GPLLNK      USE GPLLNK
       DATA INT          TO FIND LARGEST INTEGER
*
*  CODE FROM HERE ON CONVERTS THE NEW NUMBER TO A STRING,
*  PLACES THE NEW NUMBER STRING INTO THE FILE RECORD FOR
*  OUTPUT, AND WRITES THAT RECORD TO THE OUTPUT FILE
*
*
* PORTION OF DATA PART OF TOOL4 SOURCE CODE
*
FPHALF BYTE 63,50,0,0,0,0,0,0 THE NUMBER 0.5 IN FLOATING POINT FORMAT
TWOPT5 BSS  8              STORAGE FOR USER'S CONVERSION FACTOR AS F.P. NUMBER
TEMSTR BSS  81             TEMPORARY STRING STORAGE LOCATION
DELSTR BYTE 7              STRING LENGTH
       TEXT '#DELAY='      COMPARISON SUB-STRING TO FIND RECORDS OF INTEREST
T5STR  BYTE 3              STRING LENGTH
       TEXT '2.5'          DEFAULT CONVERSION FACTOR STRING
*
```

1.31. The Art Of Assembly — Part 31. "This Is A Football"

By Bruce Harrison

This month we'll start with a short story, just to have a laugh before the serious stuff starts. The story goes that the legendary football coach Vince Lombardi once got so disgusted with his team's performance in a game that he called a Monday morning meeting and announced that "We are going back to the fundamentals". He held up a football and spoke very slowly and clearly. "Gentlemen, this is a football." A voice from a player in the back of the room piped up, "Wait a minute, coach, you're goin' too fast." There was a moment of fearful silence from all the players, then coach Lombardi just started to laugh, and all joined in.

This month we're starting at the beginning, for those who have never tried Assembly. In the rest of this column, we will make every effort not to "go too fast" for any of our readers. We all know that Assembly is not an easy language to learn, and that mastering it is a term we simply don't use. Many readers have contacted us to say that, while they enjoy reading these columns, they don't really understand the subject matter. We hope this one will be easy to grasp from beginning to end.

1.31.1. Just What Is Assembly, Anyway?

Back in the bad old days when digital computers were first being produced, there was no such thing as Assembly Language. Programmers had to write their code in actual machine language, making each instruction in binary notation. This was a very tedious way of making programs, and since binary is a very unfriendly system of numbers for humans to deal with, errors in coding were the rule rather than the exception. Minor improvements came along, with use of either octal or hexadecimal coding, but still the work was tedious and error-prone.

The introduction of Assembly was a real blessing, since it allowed the use of mnemonic symbols that were human-readable to represent quantities and addresses in the source code. Now the programmer could work with a sort of words consisting of letters that meant something. It's a lot easier, after all, to recognize the word FAC in a source file than to pick out the address written as the binary number 10000011 01001010. (That's >834A, incidentally.) Assembly allowed the programmer to work with a kind of quasi-English language, then let the Assembler convert the code into the binary numbers that the computer understands.

As with labels for addresses, Assembly allowed mnemonics to represent the machine instructions as well, so that an instruction in source code could read LI R0,-1, instead of the binary 00000010 00000000 11111111 11111111 or the hex 0200 FFFF. As mysterious and terse as Assembly source code may be when compared to languages like Basic, it's still light years ahead of the binary or hex representations of the same addresses or instructions. Assembly, then is a language readable by humans, which can be readily translated by the Assembler into machine code in the binary notation.

1.31.2. Some Terminology

One of the things one encounters when trying to learn any language is the difference between nouns and verbs, adjectives and adverbs, and so on. In human languages these are called "parts of speech". In Assembly, we'll just use the word terminology. Here's a short glossary of terms that are used every day in Assembly programming:

LABEL — a short human-readable quasi-English word that can represent a location (address) in memory or a quantity. The programmer usually invents his own labels. On the TI, these must begin with a capital letter, and may not be longer than six characters.

INSTRUCTION — an abbreviation that represents an operation to be performed by the computer.

EXPRESSION — usually a number or a combination of numbers or labels that the assembler can calculate into an address.

ADDRESS — a specific place in the computer's memory, either a byte location or a word location.

BIT — contraction for Binary Digit. A two-state quantity that may only have the values 1 or 0.

BYTE — a group of eight bits treated as a unit. May take on values from 0 through 255.

WORD — a group of sixteen bits treated as a unit. May take on values from 0 through 65,535. (On some computers, such as the big "mainframe" units, the word length can be 30, 32, 60 or 64 bits.)

DIRECTIVE — an instruction to be executed by the Assembler, not the assembled program.

FIELDS — the different parts of a line in the source code.

LABEL FIELD — the very beginning of a line in the source code. (Most lines do not contain a label.)

OPCODE FIELD — the instruction part of the line.

OPERAND FIELD — the numbers, labels, or expressions upon which the operation is to be performed.

REGISTER — a special purpose word of memory, on which certain operations can be performed that cannot be performed on words in ordinary memory

WORKSPACE — On the TI, this is a set of sixteen registers. The user can select any part of the addressable RAM memory as a workspace for his use. (On most other computers, workspace registers are not assignable, but are considered part of the CPU's private reserve of memory.)

There are additional terms sometimes used, but we'll try to explain those when used, rather than extend the glossary.

1.31.3. Uses For Labels

Those who program in Basic are familiar with the concepts of line numbers and variables. The Labels used in Assembly language can serve as line numbers, variable names, or even as constants, depending how they're used in the source code context. Here are three examples to illustrate these three uses:

EXAMPLE 1 — label used like a line number

```
KEYIN       BLWP @KSCAN             Scan the keyboard
            CB   @STATUS,@ANYKEY    see if a key is struck
            JNE  KEYIN              If not, go back to label KEYIN
```

EXAMPLE 2 — label used like a variable name

```
            INC  @COUNT             add one to the variable count
            ...
COUNT       DATA 0                  One word of memory reserved as variable
```

EXAMPLE 3 — label used as a constant

```
BUF         EQU  >1050              Make the word BUF represent >1050
            ...
            LI   R0,BUF             load register zero with the value BUF
```

Each of these examples is of course just a fragment of source code, and would result in just one tiny operation when assembled. Example 1 would simply cause the computer to loop endlessly between the label KEYIN and the JNE instruction until the user strikes a key on the keyboard. The labels STATUS and ANYKEY are used here as a constant and a "variable", respectively. The label KSCAN is a constant that's usually set by a REF directive.

The small loop in Example 1 is equivalent to an Extended Basic line like this:

```
120 CALL KEY(0,K,S):: IF S<>1 THEN 120
```

Example 2 is one of the simplest operations possible, since it simply adds one to the value of a variable stored at the word location named COUNT. In XB, it would be COUNT = COUNT + 1.

Example 3 has no direct equivalent in XB, because XB doesn't allow the user to mess around with its workspace registers. What the example does is to place the value >1050 into the first word in the workspace, which is identified by the name R0. Each word in the workspace can be identified in the source code by a simple label like R0, R1, . . . R15. The Assembler will allow the R to be omitted from these labels, so that R15 could be shown as simply 15. (We find this too confusing for us, even if it's not confusing to the Assembler, so we always use those Rs, and use the R option on the Assembler.)

1.31.4. A Little Exercise

Are we "goin' too fast"? We hope not. Here's a little exercise for your fingers, to get you into the "swim" a bit. Try typing in the following source file with the E/A Editor. Save it to some disk as TEST/S, then run the Assembler on it, giving the object file name as DSKx.TEST/O, using the R option, then use the LOAD AND RUN option (3) from E/A to run it. The PROGRAM NAME is START (very imaginative). When typing in, place the label field starting at the very leftmost column.

```
            DEF START               define our entry point
            REF VMBW,KSCAN          refer to predefined labels
STATUS      EQU >837C               the GPL Status byte
WS          EQU >20BA               the User Workspace
GPLWS       EQU >83E0               the GPL interpreter's workspace
START
            LWPI WS                 set the workspace pointer at >20BA
            LI R0,11*32+2           set R0 to point at row 12, column 3
            LI R1,MSG               set R1 to point at label MSG
            LI R2,27                set length of MSG in Register 2
            BLWP @VMBW              write multiple bytes to screen

KEYIN
            BLWP @KSCAN             scan keyboard
            CB @ANYKEY,@STATUS      test for keypress
            JNE KEYIN               if none, repeat scan
            LWPI GPLWS              else set workspace for GPL
            B @>006A                then branch back to E/A

MSG         TEXT 'Welcome Assembly Programmer'
ANYKEY      BYTE >20
            END
```

Okay, so that didn't do much, but consider it a "first" program written in Assembly. It's a complete program, and will do something with your computer. More importantly, it will teach in an "object lesson" some of the basic concepts we've been trying to make clear. It illustrates all the uses of labels, for example, plus some directives. We could have made it more complex, and made it do more, but that would defeat our purpose of making an extremely simple Assembly program.

We'll try now to explain what each line of this source code does in some detail, so you'll start to get a "feel" for how this mysterious language works.

The first line says DEF START. This is a directive for the Assembler, and it tells the Assembler to identify the label START as an entry point, so that when the Option 3 object file is loaded, the E/A module will know where to start executing this program.

The next line is a REF directive. The two labels listed after this directive (VMBW and KSCAN) are pre-defined as addresses by the E/A module. When the object file is loaded, each occurrence of VMBW and KSCAN will be replaced by the actual address of the utility "vector" named in the source file.

The next three lines all establish "constants" for the Assembler. After the Assembler encounters these lines, it will look for those three labels in the source file, and will place the number found after the EQU directive into the object file in place of that label. These are placed early in the program, so that their values are established before they need to be used. You could think of these as non-variable variables. Sometimes in XB, you'll find a variable identified like COLCOUNT=32, which is then never changed during the rest of the program. The difference between this XB example and the EQU directive in Assembly is that, in Assembly, once a label has been made a constant with EQU, it cannot be changed later in the program.

After the three EQU lines, we find the first label at which something is actually to be performed by the computer. Notice that we have placed this label on a line by itself. That's done simply to make it look better. The instruction LWPI WS could be on that same line, and this would make absolutely no difference to the Assembler. The mnemonic LWPI means Load Workspace Pointer Immediate. When the computer encounters this operating instruction, it will take whatever is in the very next word location after the instruction and place that in the workspace pointer, so that from that point on, your set of registers will start at memory location >20BA.

From here on, the computer will interpret the label R0 as the word of memory at >20BA, R1 as the word at >20BC, and so on. In the next three lines, we set values into the registers R0, R1, and R2. These lines use the LI instruction, which means Load Immediate. On the TI, only registers can be loaded with values through this directive. (On PC computers, other memory locations can be loaded with immediate values.) The instruction causes the contents of the next word to be loaded into the register identified in the first part of the operand field. In the first of these three lines, we have used what's called an "expression" as the immediate value, rather than a simple number. What happens here is that the Assembler evaluates the expression, and places just one number into the word following the instruction. In other words, the Assembler does the math for you, taking 11, multiplying that by 32, then adding two to that result. Thus, after Assembly, the single number 354 [(11*32)+2] will be the "immediate" value loaded into memory as part of the program. This will result in display of our message at Row 12, Column 3 on the 32-character screen.

The next line sets the register R1 to the address of the label MSG, through a LI instruction. Wherever that line of text happens to be in memory, the number representing that location will be contained in R1 after this instruction is performed. The line after this sets R2 to the value 27, which is the length of the message at label MSG. All three of these Load Immediate operations are done to prepare the registers for execution of the next line.

The instruction BLWP means Branch and Load Workspace Pointer. This can be thought of as similar to a subprogram call in XB, but here the registers serve to transfer all needed parameters to the subprogram. When the BLWP is performed, the workspace pointer will be changed, so that a set of registers associated with the "subprogram" will be used, and the workspace we were using (at >20BA) will not be altered during the VMBW operation. VMBW means VDP Multi-Byte Write. In all cases, this operation takes the number of bytes indicated by R2, from the address pointed to by R1, and writes that number of bytes into the location in VDP RAM indicated by R0.

The result in this particular case will be that the words "Welcome Assembly Programmer" will suddenly appear at row 12, column 3 on the screen. Notice that we did not have to clear the screen here, as the Load And Run option from E/A did that for us.

From here on, the program is just a loop like the one we showed in an example before. The computer will scan the keyboard repeatedly until it finds that a key has been struck, and then will leave our program by re-loading the workspace pointer to the Graphics Programming Language (GPL) workspace, then turning control over to the GPL Interpreter at location >006A.

The lines in what we'll call the DATA section contain only labels and directives. At label MSG, the directive TEXT tells the Assembler to insert the characters following that are between the ' marks into the object file as ASCII codes. Thus the first byte at that point in the memory would be the number 87, which is the ASCII value for upper case W. That would be followed by the ASCII value for lower case e, and so on until all 27 bytes of the message were included. The directive at label ANYKEY causes the BYTE at that memory location to be equal to >20. This is normally left alone, since it's there only to compare to the GPL status byte in checking for a keystroke.

The word END in the last line is not related to the same word as used in XB. Here, it's a directive to the Assembler, to tell the assembler it can stop assembling the source file. The actual end of the program in the XB sense of the word END comes at the line containing the instruction B @>006A, since that's where this program relinquishes control of the computer.

1.31.5. Extended Basic Equivalent

This whole program could be written in two lines of XB program, like this:

```
100 DISPLAY AT(12,1):"Welcome Assembly Programmer"
110 CALL KEY(0,K,S) :: IF S<>1 THEN 110 ELSE END
```

That seems like an awful lot of Assembly to replace just a little piece of Extended Basic, doesn't it? Yes, it is, but over time you'll come to appreciate the differences in terms of execution speed and memory use that all fall on the side of Assembly.

For the convenience of those who subscribe to *MICROpendium* on disk, we have provided John with the little program above as a separate file named TEST/S, so you can avoid the hassle and time spent typing in the source code.

As always, we're just scratching the surface, and have already written too much for one sitting. Next month we'll continue this lesson for the beginners.

1.32. The Art Of Assembly — Part 32. Some Instructions

By Bruce Harrison

Last month we went back to fundamentals, to give those just starting in Assembly a first lesson in the language. Today we continue with the very primitive operations that are called "instructions". We'll cover some of the most commonly used ones, and where possible relate them to instructions in Basic.

The instruction set built into the TI-99/4A is a very rich one. Many instructions are provided, so that the microprocessor can perform a wide range of operations. This chip has lots of work to do in following the instructions we give to it. It takes each instruction apart bit by bit to determine what's to be done, and what needs to be fetched from memory, placed back into memory, and so on. We could start with almost anything, so we might as well start with the MOVE instructions.

1.32.1. A Moving Experience

One of the most commonly used instructions in our programs is the MOV (move) instruction. It's provided in two forms. MOV means we will move a word (16 bits) from one place to another. MOVB means (as you might guess) that we'll move a byte (8 bits) from one place to another. Each time we use either form of the move instruction, we must provide two operands. These operands may be labels, indicating a memory location, or they may be registers, or even combinations of both labels and registers.

Let's start with a simple case that can be related to an operation done in Basic. Suppose we have a label for a word of data called FLAG1, and we want that word moved into a word used as a variable called VAR1. The instruction in source code would look like this:

```
MOV  @FLAG1,@VAR1
```

This is equivalent to an operation in Basic such as:

```
VAR1 = FLAG1
```

In both cases, the value of the variable FLAG1 is unchanged after the operation, but the value of VAR1 is set equal to whatever was the value of FLAG1. Notice that, in Assembly, the variable that's being changed in value is after the comma, while in Basic, it's before the equal sign. In the parlance of Assembly, the operands are in the order SOURCE,DESTINATION. The destination operand gets modified, while the source operand remains the same as it was. In this first case, both operands were memory words identified by labels. Move operations can also be performed using the Workspace Registers. MOV R4,R5 makes R5 contain whatever number was in R4. This kind of move can be very useful to temporarily "stash" the contents of a register before some operation that will destroy that register's current value. After an operation that changed the value in R4, we could get back it's original value by MOV R5,R4 if we'd stashed R4 in R5 before that.

Move operations can also work in either direction between registers and variables. Thus we could put the value currently in VAR1 into R4 by MOV @VAR1,R4. Conversely, we can move the contents of a register into a labeled memory location like our FLAG1 by MOV R5,@FLAG1.

The MOVB instruction works exactly the same, except that when registers are involved, the MOVB always involves the high order byte of the register. If, for example, R4 contained the number >F3E4, doing a MOVB R4,@SAVBYT would move only the eight bits containing >F3 into the location labeled SAVBYT.

1.32.2. Integer Math Instructions

We all know that one of the computer's most basic purposes is to compute. That is, do mathematical functions. The simplest of these operations are of course to add and subtract. The instructions in source code are simply the upper case letters A and S. Like the move instructions, each of these needs two operands, a source and a destination. These can be either registers or labels used as variables, or combinations of the two. Let's start with the simplest possible example, to add the values in two registers. That looks like this:

```
A    R2,R1              add the value in register 2 to that in R1
```

As in the MOV case, the source operand remains unchanged, while the destination is changed. Let's say that, before this instruction R2 contained 6 and R1 contained 4. After the operation, R2 will still contain 6, but R1 will contain 10. The same can be done with labels used as variables, as for example the labels PLUS and COUNT. In Assembly, we could write:

```
A    @PLUS,@COUNT       add plus to count
```

This is the equivalent of the Basic instruction COUNT = COUNT + PLUS. The result is that COUNT contains the sum of its previous value and the value at PLUS, while PLUS retains its previous contents.

Subtract, with the opcode S, works in a similar fashion, with results going to the destination operand, and the source operand remaining unchanged.

1.32.3. The Tougher Stuff

In grade school, we learned that the operations of multiplying and dividing numbers were much tougher than adding and subtracting. In Basic, these operations are just as easy as the others. In Assembly, we're back to grade school, and these operations get tough again.

For openers, we can't use anything but Registers for the destination operand in MPY and DIV operations. (The reason for this will soon be clear.) The source operand may be either a register or a label used as a variable. In either case, the MPY and DIV operations affect two adjacent registers, not just one. This is done because the results of either MPY or DIV need two registers to contain the outcome. Let's say, for example, that R3 contains a number (let's say 3) that we want to multiply by 5. We could load another register with 5, then perform the operation, like this:

```
            LI   R2,5               place the number 5 in register 2
            MPY  R2,R3              multiply register 3 by register 2
```

The result of this operation will be a double-word number in registers R3 and R4. This has to be done because the result of a multiply can be a number too large to fit in one register. In this case, R3 would contain 0 after the multiply, while R4 would contain 15. If R3 contained a number like 32000 in the previous example, then the result would be 160,000. That's too big a number for one register, so it's placed in the pair of registers R3 and R4, with the lower order part in R4, and the higher order part in R3.

Divide operations start with a pair of registers to contain the number that's to be divided. The instruction DIV R1,R3 would divide the double word quantity in R3-R4 by the value in R1. Thus if the original quantity we wanted to divide were only a one-word value, we'd place that in R4 and clear R3, so that the value to be divided would be correctly represented in the two registers R3-R4.

The results of a divide will appear in the register pair used as the destination, but as two separate numbers. The first register in the pair will contain the quotient, while the second contains the remainder. Let's take a simple example. Assume that R3 contains 0, R4 contains 14, and R1 contains 3. If we then perform DIV R1,R3, the results in R3-R4 will be that R3 contains 4, which is the largest number of times 3 goes into 14, while R4 will contain 2, which is the remainder. Think of it like this:

```
04
3/14
-12
2
```

That's the operation in simple arithmetic, in which we take the trial quotient 4, multiply by the divisor, then subtract that product from the dividend to find the remainder. So long as the remainder is less than the divisor, we are finished with this stage. That's how the result shows up for a DIV operation, with the quotient in the first register and the remainder in the second.

1.32.4. Compare And Jump Instructions

The jump instructions are related in some respects to the Basic GOTO instruction, but most of them actually perform an IF-THEN kind of operation, and then a GOTO operation. In many cases these "conditional jumps" are used in conjunction with the compare instruction, so we'll treat them together.

Let's start with a simple situation, in which we want to see whether a register contains a number greater than some limit, and to continue the program somewhere else if that's so. In such a case, we could write:

```
            C    R5,@LIMIT          compare R5 to the variable LIMIT
            JGT  BIG                if greater, jump to label BIG
            (else continue program here)
BIG         CLR  R5                 clear register 5
```

For the moment, we'll ignore the fact that one of the operands for compare is a register, and state what would be an equivalent in Basic:

```
300 IF REG5 > LIMIT THEN 500
310 (program executes if REG5 is equal to or less than LIMIT)
...
500 REG5=0
```

Here, the program lines starting at 310 would execute if the variable REG5 were less than or equal to the variable LIMIT, but if variable REG5 were greater than LIMIT, then line 500 would be executed after the IF-THEN.

The compare instruction (C) needs two operands, and these can be registers, labels, or a combination of the two. After a compare, conditional jumps can be made on any of several results of the comparison. JEQ means Jump if Equal, JLT means Jump if Less Than, and of course JGT means Jump if Greater Than. There are other conditionals, such as JNE, for Jump if Not Equal. For some operations, we can use the comparisons on a logical basis rather than an arithmetic basis. Instructions such as JH and JL work that way. To understand the difference, let's take an example, in which we are comparing R3, which contains >8000, and R4, which contains >0005. If we use C R3,R5 followed by JGT, the jump will not happen, because >8000 is treated as the negative number -32768, and that's less than 5. If we used the logical equivalent JH, the jump will occur, since >8000 is logically higher than >0005.

In many cases, the conditional jumps can be used without the compare instruction. This is so because the bits in the status register are affected by many operations, so the "compare" is not necessary. For example, we can perform the equivalent of a FOR-NEXT loop this way:

```
            LI   R4,4               load register 4 with the number 4
DOIT        (perform some operation)
            DEC  R4                 decrement the count in R4
            JNE  DOIT               if not zero, repeat.
            (else perform next operation)
```

This works because when we DEC or INC or MOV something, there is an implied comparison to zero. Thus in the above case, each time through the loop we DEC R4, and the result is compared to zero by the microprocessor, so that a conditional jump after the DEC will behave as if a comparison between the register and zero had been performed. In the above case, the operations between label DOIT and the JNE instruction will be performed four times, until the DEC results in R4 becoming zero.

There are two other forms for the compare instruction. CB means compare just one byte, instead of the word that's compared by the C instruction. CI means Compare Immediate, and this special form must have a register as its first operand and an immediate value as its second operand.

The final case for jumps is the unconditional jump, given simply by the instruction JMP. That's like the unconditional GOTO you're familiar with in Basic. There are limits on the range of jumps, but that's beyond the scope of the present discussion. (For cases beyond the range of jumps, there's the unconditional B or Branch instruction, which we'll cover in another lesson.)

In our next installment, we'll cover some more instructions, and will try to give our readers some idea how to create real programs with this powerful language.

1.33. The Art Of Assembly — Part 33. More Instructions

By Bruce Harrison

Today we'll continue with the mini-series for beginners in Assembly language. Last month we covered the use of some of the TI's machine instructions, and today we will cover more of the most often used ones, as well as closing some topics we began last month.

1.33.1. Jump Range

We mentioned last month that the range of jump instructions is limited, without elaboration. Now's the time to explain ourselves. A jump instruction itself occupies only one word in memory. Within that single word are bits to indicate the kind of jump that's intended (JEQ, JNE, JGT, etc.) and the address offset to which that jump is to take place. The space for this offset is only eight bits (one byte). That means the number itself can't go beyond the range of -128 through +127. In hex, that's the range from >80 through >7F. The microprocessor interprets this number as a differential address (in words, not bytes), starting from the present state of the program counter. "The WHAT?", you ask. Inside the microprocessor chip are special registers that we can't access directly. (Some hardware person will probably write us a letter to say something about whether these registers are "on-chip" or not. Save your ink, we don't care!) The three important ones for the present discussion are the Workspace Pointer, the Program Counter (or pointer) and the Status Register. We already discussed the Workspace Pointer, which simply keeps track of where the set of registers we're using is located in memory. The Program Counter keeps track of where the next Instruction to be performed is located in memory. This advances in word steps, always pointing at even-numbered addresses. Each time an instruction is fetched from memory, the program counter is incremented by two, so it points to the next possible location for an instruction.

We know this is already getting pretty heavy, but there's just no substitute for really explaining what happens so that you'll understand how Jump instructions work. Let's look at a concrete example. Suppose the computer is executing the following sequence of instructions:

```
            LI   R4,5               place five in register 4
REPEAT      MOVB *R9+,*R10+         move a byte
            DEC  R4                 subtract one from R4
            JNE  REPEAT             if not zero, jump back.
            INCT R4                 (next operation)
```

For the moment, we will ignore most of these instructions, and start with what happens when the microprocessor is executing the instruction JNE REPEAT. Once that instruction has been fetched from memory for execution, the chip will advance the program counter by two bytes, so the program counter now points to the location in memory where INCT R4 is located. The computer executes the JNE instruction by comparing the last completed operation to zero. Actually, that's not quite accurate. It does the comparison to zero as part of the execution of the DEC instruction, so that the status register bits are already set before the JNE instruction is fetched from memory. If the DEC R4 resulted in R4 becoming zero, the "EQUAL" bit will be set (made to equal 1) in the Status Register. If R4 did not become zero, then the EQUAL bit in the Status Register will be zero. This latter condition, where the EQUAL bit is "turned off", will cause the microprocessor to look at the second byte in the JNE instruction, and to add twice that "offset" to the contents of the Program Counter. In this particular case, the second byte in the JNE instruction will contain the value -3, so that when this is doubled and added to the program counter, the next instruction to be fetched will be the one at label REPEAT, instead of the instruction INCT R4. Thus a loop operation will be performed until the number in Register 4 decrements to zero. On that pass through the loop, the jump will not happen because the "EQUAL" bit in the status register will have been turned on, so the computer will go ahead to the next instruction after JNE, or INCT R4.

If there were a couple of pages of source code between label REPEAT and the JNE instruction, the offset byte stored in the JNE instruction word might have to be a larger negative number than -128. If that's the case, the Assembler will tell us that by issuing an error during the Assembly process. The error will be reported as "RANGE ERROR in line XXX". This condition is hard to predict when looking at source code, because the number of words in memory that each instruction line occupies may vary from one word to three. (In the example shown above, all of the instructions used were one-word kinds, so our calculation of the jump offset as -3 was easy.)

When one encounters this situation, the normal "workaround" is to switch over the logic involved, then use a B or Branch instruction to go back to label REPEAT. The revised code would look like this:

```
            LI   R4,5
REPEAT      MOVB *R9+,*R10+
            (lots more instructions included in loop)
            DEC  R4
            JEQ  NEXTOP                 if zero, jump ahead
            B    @REPEAT                else branch back to REPEAT
NEXTOP      INCT R4
```

What we've done is to invert the logic of our Jump so that it will make the short jump ahead to label NEXTOP when R4 becomes zero, else it will perform the operation of an unconditional branch to repeat the loop. Because the TI measures jump ranges in words, not bytes, the acceptable range for jumps is actually quite broad. (On PC computers, jump ranges are in bytes, which means they have only half the range potential of those on the TI. This is not so for the unconditional JMP on the PC, which has a range anywhere within a 64K byte segment.)

Our "normal" practice, when there's any doubt about the range of a jump, is to put the jump instruction in anyway, then let the Assembler tell us whether that's okay. This sometimes has bad results, but it often saves us bytes in our programs by running the risk on jump ranges.

You'll notice that this revised source code needs more instructions, and one more label than the first example. The instruction B @REPEAT takes four bytes in itself. That's because there is a two-byte word just for the B instruction, then another two-byte word for the address of the label REPEAT. This leads us nicely into another topic for today.

1.33.2. The Branch Instructions

There are three important forms of Branch instructions, so let's cover all three here. Each is easy to spot, because it starts with the letter "B". The plain and simple Branch has just that letter as its opcode. The other two forms are the BL (Branch and Load) and the BLWP (Branch and Load Workspace Pointer) opcodes. Each requires just one operand, which is an address to which the branch is to happen. Any of the forms can branch to anywhere in the 64K address space. (This, while true, must not be taken too literally, as branching to some unknown place in the memory space can have disastrous results.)

Now that you know what the Program Counter is, we can discuss these branches in the more correct sense of how the computer executes them. For the unconditional Branch, the computer takes the second word of the B instruction, places that in the program counter, then continues execution with whatever instruction is at that location. (There are situations where the B instruction or the BL instruction occupies only one word in memory. We'll get to one of those in a few moments.) No "return address" is saved anywhere, so there's no automatic way of "returning" from a B. In most senses, this is a direct equivalent of the Basic GOTO instruction.

The second form, called BL, is roughly equivalent to the Basic GOSUB instruction. The return address is stashed in Register 11 of the current workspace, and the program counter is loaded with the word following the BL instruction, then execution continues with the instruction at that location. The difference here is that a "return" to the instruction following the BL is possible by having an RT (return) instruction at the end of the subroutine.

Before going on to BLWP, let's digress for just a moment into the alternate ways of providing the address for a Branch operation. This can be, as we've shown in the example above, simply a label preceded by the @ symbol. It can also be an address contained in one of the workspace registers. This allows the program to dynamically change the place to which the branch goes. Let's suppose that we have a situation in which, depending on what has happened in the program, there may be a need to BL to any of several different subroutines. (Basic equivalent is ON-GOSUB) If we arrange to place the address of the chosen subroutine in, say, register 5, then we can write a source instruction like this:

```
BL    *R5
```

The asterisk tells the computer to use whatever number is in Register 5 as the address for the BL instruction. Needless to say, this is a very powerful form of the instruction, because it allows us to perform the equivalent of ON-GOTO or ON-GOSUB operations based on what happens during program execution.

We'll now go headlong into the really complex case of the BLWP instruction. As we pointed out in an earlier part of this mini-series, the BLWP allows us to change the workspace pointer for the duration of a "subprogram". This is similar in many respects to the use of subprograms in Extended Basic, but not with respect to "variables" in the data portion of the program.

To really understand a BLWP, we must now define another term, the Vector. A Vector is nothing but two words of DATA in memory, which provide the information the computer needs to execute a BLWP. Let's look at an example:

```
UTVEC       DATA UTWS,UTSUB         subprogram workspace & code addresses

UTWS        BSS  32                 32 bytes set aside for workspace
UTSUB       (first instruction in subprogram)
            (subprogram code continues)
            RTWP                   return from subprogram

USEUT       BLWP @UTVEC            use the subprogram specified at UTVEC
```

The BLWP is a two-word instruction, where the second word is the address of the VECTOR, and is not the address at which program execution will continue. The vector itself is two words of data. The first of these is the address of the workspace to be used by the subprogram, and the second is the address of the first instruction in the subprogram.

As the name implies, the instruction BLWP both branches and loads the workspace pointer. It actually does more than that. If we BLWP @UTVEC, as above, many things happen all in that one instruction. First, the microprocessor takes the first word at the vector address to use for its workspace pointer. Next it takes the current contents of the workspace pointer and places that number into R13 of the new workspace. It takes the current contents of the program counter (pointing to the instruction after the BLWP's two words), and places that in R14 of the new workspace. Next it takes the current contents of the status register and places that in R15 of the new workspace. Once all this is done, the workspace pointer is changed to the new workspace location, and the program counter is loaded with the second word of the vector, so that execution will continue with the first instruction in the subprogram.

1.33.3. Many Happy Returns

Both the BL and the BLWP instructions set up means for the program to return to whatever comes after the branching instruction's one or two words. The return instructions are different, with RT being used in the BL case, and RTWP in the BLWP case.

In the case of BL, the "return address" is placed in R11 of the workspace, so that a subroutine can be ended with the instruction RT (return). RT actually is just a pseudonym for the instruction B *R11. Both source statements (RT and B *R11) produce exactly the same result when assembled. That is the hexadecimal number 045B. The 045 part means Branch to the address in a register, while the B tells the computer that register 11 (B is the number 11 in hex notation) contains the address to which it's to branch. For this return method to work properly, it's important that the subroutine should not tamper with the contents of Register 11 during its execution, so that the return address is preserved until it's needed.

In the BLWP case, the instruction RTWP causes a whole bunch of things to happen. The old Workspace Pointer is taken from R13, the program counter is re-loaded with the contents of R14, and the status register is re-loaded with the contents of R15. Execution then continues at the instruction that follows the BLWP instruction. As in the case of BL subroutines, it's important that the contents of R13, R14, and R15 not be tampered with during the subprogram's execution.

In both situations, there are cases in which the contents of these "return" registers can safely be altered, but those are beyond the scope of our "beginner's" lessons. Earlier columns in this series have dealt with ways of handling such alterations of the registers for the more advanced student.

That's a lot of material for one sitting, so perhaps we'd better quit writing now. Next month we'll conclude this little digression into the beginner's realm, and after that it'll be back to the more advanced and esoteric stuff. Of course some of you may consider this member of the series to be advanced and esoteric, but we can't help that. There's just no substitute for taking the time to understand what things really do.

1.34. The Art Of Assembly — Part 34. Cramming Time!

By Bruce Harrison

It's getting near "final exam" time for our beginner students of Assembly Language, so we're going to "cram" in this month's installment. We'll try to cover as much remaining ground as possible in one session. We'll start by trying to cover a few more important instructions.

1.34.1. Immediates Galore

Back in the first part of this mini-series for beginners, we showed examples of two "immediate" instructions, namely Load Immediate (LI) and Load Workspace Pointer Immediate (LWPI). The word "immediate" has nothing in particular to do with suddenness of the operation, but simply means that the word in memory that immediately follows the instruction is to be used as data in performing the instruction. Thus LI R2,27 means load register 2 with the next word in memory, which in this case contains the number 27. As we showed in that column, the immediate value itself can be represented in source code by an expression, which the Assembler computes into a single word value.

There are some other important "immediate" instructions, all of which involve a register as the first operand, with an immediate value as the second. In most of these instructions, the usual order of operands for TI Assembly is reversed, in that the first operand (register) is the Destination, and the second (Immediate Value) is the Source. We'll cover the exception in a couple of moments, but first let's simply list all the "Immediate" instructions.

LWPI	Load Workspace Pointer Immediate
LI	Load a register with Immediate value
AI	Add an Immediate value to a register
ANDI	AND the register with an Immediate value
ORI	OR the register with an Immediate value
CI	Compare the register to an Immediate Value

Add Immediate does exactly what its name implies. If register 2 held a value of 45, and we perform the instruction AI R2,15, then R2 will contain 60 after the operation. Note that there is no such thing as "subtract immediate", but you can make a subtraction happen by placing a minus sign before the immediate value. Thus performing AI R2,-15 in the above case would make R2 become 30.

ANDing an immediate value is a logical operation which is usually used to isolate particular bits in the register for examination. An easy example is hard to come by, but just suppose that it's important to know whether the number in R2 is an odd or an even number. Odd numbers in Binary always have a 1 as their least significant bit, while even numbers always have their LSB set at 0. (Trust us that this is so.) The easy way to tell about the number in Register 2, then, would be simply to ANDI R2,1. The immediate value 1 has only its least significant bit set to 1. (In binary, that's 00000000 00000001.) Thus when an AND operation is performed, R2 will contain 1 if and only if its LSB was 1 to start with. If it was an even number, the result of ANDI R2,1 will be zero. The test could look like this in source code:

```
            ANDI R2,1
            JEQ  EVEN              if result is zero, jump
            (else it was an odd number)
EVEN        (do whatever follows for an even number)
```

If the value in R2 were important to begin with as such, then one should move that value to someplace else before the ANDI instruction, because only the LSB of R2's content will survive this operation.

The OR Immediate does effectively the opposite of ANDI, in that it will force a 1 into each bit of the destination register wherever there's a 1 in the Immediate value. Let's take a similar situation involving R2, but this time we want to make sure that the number in R2 is forced to be odd. We can do this by the simple instruction ORI R2,1. If R2 were already odd, this would have no effect, and its value would be the same as before the operation. If, however, it were even, its value would be incremented by one, making it an odd number.

These are not the most common uses for ANDI and ORI, but we chose to use these illustrations because they are simple enough to give the beginner a feel for the instructions.

The final Immediate instruction, Compare Immediate, is the one exception to the unusual Destination,Source relationship for Immediate instructions. If for example, we perform an operation like this:

```
            CI   R8,78
            JGT  BIG
```

We will jump to BIG if the value in Register 8 is larger than the immediate value 78. As with all Compare operations, neither Operand is changed by the comparison. (The microprocessor performs a subtraction internally, but does not "output" the results except into the Status Register.)

1.34.2. A Shifty Business

Way back when, those special memory words that we call Registers were called by the name Shift Registers. One of the operations that is exclusive to the registers in the TI is the series of instructions called Shift Instructions. These are hard to visualize, even in hex numbers, and virtually impossible to understand in decimal numbers, since they are operations that happen to the bits in the registers. Perhaps then you'll forgive us if we move back for a minute or two into that mysterious world of binary numbers.

Let's make this as easy as possible by starting with a register that contains the value 1. In binary, the content of the register would be 00000000 00000001. Now we can perform some shift operations on this register (let's say it's Register 2). For openers, let's shift it left by one bit. That's done by the source statement SLA R2,1. The result will be that the single one bit in the register will move over one position to the left, and its original place will become a zero. It would then look like this in binary: 00000000 00000010. The effect in this case is that the register that contained 1 now contains 2. Within limits, the SLA instruction can be used as a multiply operation, multiplying by two for each bit position that the register is shifted to the left. If the situation above were SLA R2,4, then the register that contained 1 would contain 16.

Before getting too involved, let's try an opposite case, where we shift to the right instead of to the left. If we take the register as starting with 16, then SRL R2,4, we wind up with the value 1 in the register.

The menu for shift instructions goes like this, with left shifts consisting of just one item, while right shifts have three choices:

SLA	Shift Left Arithmetic
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SRC	Shift Right Circular

The difference between Logical and Arithmetic shifts involves what happens to the Most Significant Bit in the register, known also as the Sign Bit.

In integer arithmetic operations, any quantity in which the leftmost bit is a one is treated as a negative number. Let's say that a register contains 10000000 00000000 in binary. That's a negative number (8000 in hex, -32768 in decimal) to the computer. Similarly, the number expressed in binary as 11111111 11111111 is also negative (-1 in decimal, or FFFF in hex). If we simply change that leftmost bit to a zero, as 01111111 11111111 it becomes the positive number 32767. If we do a SRL operation, the leftmost bit will be replaced by zero regardless of what its previous value was. If, on the other hand, we SRA, the leftmost bit will remain whatever it was, so that a negative number will stay negative after the shift is performed. In other words, for an Arithmetic shift, the sign of the number is preserved, while for a Logical shift, it can be changed.

That leaves only SRC to discuss. Shift Right Circular means that when a bit is shifted out on the right side, it "circles" back to the left side. Let's take our old friend R2, load it with just 1, and then do an SRC R2,1 operation on it. Before, it has 00000000 00000001, and after this operation it will have 10000000 00000000. In decimal values, that one instruction changed 1 to -32768. Quite a shift!

We'll leave this subject here, because there's lots more ground to cover, and there are excellent descriptions of what happens in these shift instructions in the E/A Manual.

1.34.3. Other Instructions

As a quick check in the Assembly Manual will reveal, there are a lot of instructions we haven't touched. Some are very simple, like the instruction CLR, which has only one operand. CLR makes its one operand equal zero, which can come in very handy indeed. The operand can be any memory location or a register. Other handy one-operand instructions are INC and INCT, DEC and DECT. These add or subtract one or two from their destination operands.

Once you've started playing around with your own Assembly programs, and gotten some things working, you'll find that the explanations of the instructions as given in the Manual will start to make sense, and you'll be able to apply them for yourself.

1.34.4. Addressing Modes

In most of the lessons in this mini-series, we have stuck to the simplest modes of addressing memory. The TI has a number of very interesting and useful ways to address memory. Here's a brief summary and a couple of examples.

The addressing modes are these:

Direct addressing
Register indirect addressing
Register indirect auto-increment
Indexed

Direct addressing simply means that the operand named in the instruction is the address for the source or destination. This can be either a register or any location in memory. For example, we can write MOVB R1,@PABDT+1. This will move the high order byte from R1 into the memory location one byte past the label PABDT.

Register indirect addressing means that the register's contents are to be used as the address. For example, if we write MOV *R9,*R10, the asterisk tells the Assembler that whatever numbers are in R9 and R10 are addresses for the source and destination operands, and the registers themselves will not be changed by this operation.

In addition to the asterisk in front of the register numbers, we can add a plus (+) sign after the register to cause the number in the register to auto-increment after the operation. (e.g. MOV *R9+,*R10+) In the case just cited the plus would mean that after moving the word from the location pointed to by R9 into the word pointed to by R10, both registers would be incremented by two, so they'd point at the next word. For MOVB operations, the auto-increment will advance the values in the registers by only one, so they'll point at the next byte in memory.

Finally, there's indexed addressing. This uses a combination of direct addresses and registers. Suppose there's a lookup table, at label LUT, and R1 contains the member of the table we want to access. We can write MOVB @LUT(R1),R2. This will add the value in R1 to the address LUT, then move the byte at that location into the high byte of R2. This mode can be a very powerful means of accessing arrays of data in memory.

All these addressing modes have been used in our own Assembly work, and they've all had their advantages depending on what we were trying to do. We trust that you too will come to have great respect for the designers who built all this power into that tiny little computer chip.

1.34.5. Final Exams

Sorry, but there will be no final exams. This headline was just to get your attention. If you're reading this, it worked, so we can try to start wrapping up our beginners' mini-series. As you all knew at the start, this introductory material would not create "instant assembly programmers" from neophytes. Its purpose is to give you all a "feel" for the language, and in some degree to get you ready to tackle the "big book", which can be very confusing at times. Also, those of you who've kept all your back issues of *MICROpendium* can go through some of the earlier parts of this series and perhaps understand them this time. If you have *MICROpendium* on disk, you'll find many useful "snippets" of source code in the Sidebars to these articles. Using those, either "as is" or just as examples should move you well along toward becoming an Assembly programmer.

1.34.6. Parting Thoughts

During this mini-series, we have made some very general comparisons between various operations as represented in Assembly and "similar" operations in Basic. The similarities are mostly superficial. When we perform an operation like A = A + B in Basic, what really happens is that two eight-byte floating point numeric quantities are added, and the resulting eight byte floating point number is stored in memory by Basic at the location reserved for the variable A. In our Assembly "equivalent", A @B,@A the quantities being added are one-word integers, with a range of values limited to -32768 through +32767. In the Basics, the statements placed in the program need to be interpreted by the computer, and even a very small statement like that above results in the execution of hundreds of machine instructions to first interpret and then execute the desired operation. This interpretation that takes place when Basic is running is both a strength and a weakness. It's a strength because of the simplicity for the programmer, but the need to interpret and then perform all those steps makes the operations slower by many fold when compared to similar operations coded in Assembly.

We hope that at least some of you will "stay the course" in learning Assembly. There's an enormous effort involved to become proficient in this language, and even your author, who's been doing Assembly programming for some years on both TI and PC computers, still has things to learn. The work is difficult at times, but the reward, in having programs that do "impossible" things at incredible speed, makes it all worth the pain.

1.35. The Art Of Assembly — Part 35. Even More's There!

By Bruce Harrison

Our production of this series has been stopped for a short while as we wrote the "beginner" mini-series. This month, we take up where Part 30 left off, with more on the use of features built into the TI, many of which are undocumented. We'll be overlapping into our friend Barry Traver's territory somewhat, in that much of what's shown this month is designed for operation in the Extended Basic "environment". That is, with a 32 character screen, the offset needed for characters written to the screen, and so on. Today's Sidebar is, in fact, the source code for a couple of routines we made for use under Extended Basic. The key "what's theres" in this case came from Harry Wilhelm, to whom we owe yet another debt of gratitude.

1.35.1. Clearing The Decks

Let's start with something really simple, which will get the computer into a "Basic" environment very quickly. One little BLWP to GPLLNK will do all of the following:

1. Clear the screen.
2. Restore the normal Basic character sets, both upper and lower case, including the cursor and edge characters.
3. Set the random number seed.
4. Delete any and all sprites.
5. Set the screen color to cyan.
6. Set the colors for all character sets to black on cyan.
7. Set all VDP registers to their Basic values.

Pretty good for just one call to GPLLNK. The magic number is >27E3. Or, as it shows up in source code:

```
BLWP @GPLLNK
DATA >27E3
```

That's it! All seven of those things happen in a trice, and you're set up with a cleared cyan screen and essentially in a Basic environment. The singular drawback to all this (you knew there had to be one) is the ever-popular offset for the screen characters. Adding >60 to each character sent to the screen and subtracting >60 from any character read from the screen is a real pain, but that's part of the price we pay for the "efficient" arrangement of the character definitions in Basic. The other part of the price we pay is that characters below the cursor (>1E, or >7E with the offset) cannot be used, since the space that would be used for their definitions is part of the screen.

We've seen cases where a programmer got around that limitation by using >3800 as his screen image table, but that's not a practice we recommend. We won't condemn you for doing that, but don't complain to us if that screws up something else you were trying to do.

Included in today's Sidebar is another gift from Harry Wilhelm, in the form of two utility vectors (PRSTR and VMBW60) that will help you over the problem of writing things to the screen with the offset. These will help so long as you keep in mind that the characters from number 29 downward are verboten. (German for Forbidden.)

Pardon a digression, but that word verboten has such a neat ring to it. It always reminds us of the opening scene of the TV show Hogan's Heroes. As the prisoners were dashing out into the snow from their quarters, we noticed a sign beside the door. The sign was in German, so don't ask us what it said, except that the word in large letters at the very top was VERBOTEN, and the list of items in very small letters stretched down to the ground. That kind of thing always has special meaning for those who've worked for the Government. The list of what's verboten always fills many more volumes than the list of what's permitted. Now I've been reminded of another story, concerning the use of Festive Decorative Materials, but I'd better save that story and get on with the business at hand, or this column will be verboten. (That's five times we've used that word. Enough!)

Where were we? Oh yes, the lead-in to the Sidebar should come next. In that Sidebar, the first part is source code for what's called the RSXB routine. As you can see, this is a very simple but complete sub-program for use with CALL LINK from Extended Basic. (Presumably it could also be used from console Basic if the E/A cartridge is available.) What makes it complicated is the need to include one's own GPLLNK, since TI in its infinite wisdom decided not to provide that for XB users. (Thanks again, TI!)

1.35.2. The Line Editor

Harry Wilhelm has provided us with another undocumented GPL routine, the Line Editor. So far as we can tell, this is the routine that's used by the Command mode in Basic and Extended Basic to allow you to type in commands or program lines, and of course to edit existing program lines. As soon as we'd tinkered around with this a bit, an old idea came flashing back into our consciousness. That idea was something we call the "Ultimate Accept At". We always liked TI Extended Basic's ACCEPT AT, for the neat way it allowed us to take user inputs from anywhere on the screen. Our main gripe with it was that no more than 28 characters could be input. String Variables can hold as many as 255 characters in XB, but not if entered through ACCEPT AT.

Having the TI Line Editor available, with its many allowed parameters, gave us the freedom to invent that desired routine which permits an effective Accept At with room for 255 characters if desired. That's real Power, there in that Line Editor which TI didn't bother to mention to us. One can place the start and stop locations for input anywhere on the screen. (The stop location can even be off the bottom of the screen, if we'd like.)

The second part of today's Sidebar is the source code for the Ultimate Accept At. This uses TI's Line Editor by the simple source statements:

```
BLWP @GPLLNK
DATA >285A
```

Of course this little routine does a lot of work getting prepared for that call, including the reading of parameters that XB passes along in the Call Link, clearing the necessary screen area for our input field if desired, and placing a prompt on the screen if needed. The routine also checks to see whether there's enough room on the screen for the desired length of input field, and moves its ACCEPT up by enough rows to insure screen space.

This routine, designed for the Extended Basic programmer, is available as Public Domain software from a number of sources, including the Lima and Chicago User's Groups. The disk includes the complete source code, object files, instructions on how to use the routines, and demo XB programs to show how they work. If you can't get it anywhere else, we'll provide it directly for a paltry $1.50 to cover media and mailing. (5705 40th Place, Hyattsville MD 20781 USA)

1.35.3. Other Undocumenteds

No, you can't hire these to look after your children, or to serve as domestic workers. There are two more for today's column that are worth mentioning. Back in Part 30, we showed a GPL routine that would take an integer number and convert it into a string, which we then displayed on the screen. Our friend Harry has an even better deal for us. Once again, this uses the offset for screen characters, but we don't need to add the offset, or even do the displaying. All we do is take the one-word integer, move it into location >835E, put our desired screen location at >8320, then call the routine through GPLLNK. Looks like this:

```
LI   R0,32*11+2         (row 12, col 3, for example)
MOV  R0,@>8320          put that at >8320
MOV  @NUMBER,@>835E     place the integer at >835E
BLWP @GPLLNK            use GPL linkage
DATA >2842              with this DATA number.
```

Voila! The number is on the screen, with offset, in decimal notation, at row 12, column 3. Neat, eh?

That's another small miracle. No doubt there are hundreds of these that we don't yet know about. Last on today's list is the matter of scrolling the screen. Some time ago, we showed the undocumented way to do that using GPLLNK:

```
BLWP @GPLLNK
DATA >4D00
```

That one will work from E/A, or Basic, or Extended Basic. The newest addition to our growing list is actually a documented one (Page 416 of the E/A Manual). This is a scroll that can be used only in the XB environment, and through XMLLNK, not GPLLNK. It looks like this:

```
BLWP @XMLLNK
DATA >0026
```

This is, as we said, documented on page 416 of the E/A manual, as part of a list of unexplained equates,

all of which are to be used with XMLLNK, except that TI forgot to mention that this is how they are to be used. That list includes:

```
CNS         EQU  >06          convert number to string
VPUSH       EQU  >0E          PUSH a value?
VPOP        EQU  >10          POP a value?
ASSGNV      EQU  >18          does what?
CIF         EQU  >20          convert integer to floating point
SCROLL      EQU  >26          Scroll screen
VGWITE      EQU  >34          does what?
GVWITE      EQU  >36          does what?
```

The meanings of the "does whats" in this list is unclear. If any of our readers has used these equates for anything, please let us know, and we'll pass the information along. We suspect that the VPUSH and VPOP have something to do with using a value stack in VDP RAM for floating point math operations, but can't at this time confirm or deny that suspicion. The SCROLL in this list reportedly works faster than the >4D00 scroll, but this one apparently works only from the Extended Basic environment.

Here it is, getting on toward ten years since the end of production of the TI-99/4A, and still there are mysteries buried in the "system" software needing to be unraveled. Our friend Harry Wilhelm uses the old book "The TI Intern" to dig out some information, but having looked at that, we still don't see how he comes up with useful routines from it.

That's enough for today. There's a rather big Sidebar for today's column, with all that annotated source code to chew on, so this will be it for now. Next month's topic is still undecided, so we'll once again surprise you (and ourselves) next month.

```
* SIDEBAR 35
*
* FIRST PART IS "RSXB" - RESETS MANY
* PARAMETERS TO DEFAULT CONDITIONS
* AS EXTENDED BASIC CALL LINK("RSXB")
* PUBLIC DOMAIN
*
STATUS EQU  >837C
GPLWS  EQU  >83E0
GR4    EQU  GPLWS+8
GR6    EQU  GPLWS+12
STKPNT EQU  >8373
LDGADD EQU  >60
XTAB27 EQU  >200E
GETSTK EQU  >166C
       DEF  RSXB
RSXB
       LWPI WS           LOAD OUR WORKSPACE
       CLR  @STATUS      CLEAR GPL STATUS
       BLWP @GPLLNK      USE GPL LINKAGE
       DATA >27E3        CLEAR OUT TO XB DEFAULTS
```

```
       LWPI GPLWS        LOAD GPL WORKSPACE
       B    @>6A         BRANCH TO GPL INTERPRETER
WS     BSS  32           OUR WORKSPACE
*
* FOLLOWING IS GPLLNK BY DON WARREN/CRAIG MILLER
*
GPLLNK DATA GLNKWS
       DATA GLINK1
RTNAD  DATA XMLRTN
GXMLAD DATA >176C
       DATA >50
GLNKWS EQU  $->18
       BSS  >08
GLINK1 MOV  *R11,@GR4
       MOV  *R14+,@GR6
       MOV  @XTAB27,R12
       MOV  R9,@XTAB27
       LWPI GPLWS
       BL   *R4
       MOV  @GXMLAD,@>8302(R4)
       INCT @STKPNT
       B    @LDGADD
XMLRTN MOV  @GETSTK,R4
       BL   *R4
       LWPI GLNKWS
       MOV  R12,@XTAB27
       RTWP
       END
*
* END OF THE "RSXB" ROUTINE
*
* SECOND PART IS SOURCE CODE FOR THE
* ULTIMATE ACCEPT AT ROUTINE
*
* PUBLIC DOMAIN
* USE WITH EXTENDED BASIC CALL LINK
* USE BY CALL LINK("ULTACC",R,C,CLR,"PROMPT",NOCHRS,VAR[,"BEEP"])
* WHERE: R,C ARE ROW AND COLUMN
* CLR MAY BE 0, 1 OR 2 (0 DOES NOT CLEAR SCREEN, 1 DOES, 2 RESETS XB)
* PROMPT MAY BE ANY STRING, INSERT "" FOR NO PROMPT
* NOCHRS IS NUMBER OF CHARACTERS TO ACCEPT.
* MAKING NOCHRS NEGATIVE WILL PUT EXISTING VALUE OF VAR ON SCREEN AS DEFAULT
* THE VARIABLE MAY BE ANY NUMERIC OR STRING VARIABLE
* THE "BEEP" IS AN OPTIONAL PARAMETER - PLACING ANYTHING
* AFTER THE VAR PARAMETER WILL MAKE A BEEP OCCUR
* IF THE WORD BEEP IS USED, IT MUST BE SURROUNDED BY QUOTES
*
VMBW   EQU  >2024        VDP MULTI-BYTE WRITE
VSBW   EQU  >2020        VDP SINGLE-BYTE WRITE
VMBR   EQU  >202C        VDP MULTI-BYTE READ
```

```
VSBR   EQU  >2028        VDP SINGLE-BYTE READ
KSCAN  EQU  >201C        KEYBOARD SCAN
CFI    EQU  >12B8        CONVERT FLOATING POINT TO INTEGER
CIF    EQU  >0020        CONVERT INTEGER TO FLOATING POINT
CNS    EQU  >0014        CONVERT NUMBER TO STRING
CSN    EQU  >11AE        CONVERT STRING TO NUMBER
ARG5ID EQU  >8305        LOCATION FOR TYPE OF ARGUMENT 5
ARGNUM EQU  >8312        LOCATION FOR NUMBER OF ARGUMENTS
SCRWID EQU  32           SCREEN WIDTH
STRASG EQU  >2010        ASSIGN STRING VARIABLE
STRREF EQU  >2014        GET STRING VARIABLE
NUMREF EQU  >200C        GET NUMERIC
NUMASG EQU  >2008        ASSIGN NUMERIC VARIABLE
XMLLNK EQU  >2018        XML LINKAGE VECTOR
FAC    EQU  >834A        FLOATING POINT ACCUMULATOR
FAC11  EQU  FAC+11       PLUS ELEVEN BYTES
FAC12  EQU  FAC+12       PLUS TWELVE BYTES
SCROLL EQU  >0026        XB SCROLL ROUTINE (W/XMLLNK)
GPLWS  EQU  >83E0        GPL WORKSPACE
GR4    EQU  GPLWS+8      GPL REGISTER 4
GR6    EQU  GPLWS+12     GPL REGISTER 6
STKPNT EQU  >8373        STACK POINTER
LDGADD EQU  >60
XTAB27 EQU  >200E
GETSTK EQU  >166C
STATUS EQU  >837C        GPL STATUS BYTE
       DEF  ULTACC,ULTCLR
ULTCLR
       LWPI WS           LOAD OUR WORKSPACE
       BL   @CLRXB       USE CLRXB SUBROUTINE
       B    @EXIT        EXIT THIS ROUTINE
ULTACC
       LWPI WS           LOAD OUR WORKSPACE
       CLR  @NUMFLG      CLEAR NUMBER FLAG
       CLR  @DEFFLG      CLEAR DEFAULT FLAG
       CLR  R0           NOT ARRAY
       LI   R1,1         FIRST PARAMETER (ROW)
       BLWP @NUMREF      GET NUMBER
       BLWP @XMLLNK      USE XML VECTOR
       DATA CFI          TO CONVERT TO INTEGER
       MOV  @FAC,R4      PUT ROW IN R4
       JLT  ROWERR       IF NEG, ERR
       JEQ  ROWERR       IF ZERO, ERROR
       CI   R4,25        COMPARE TO 25
       JLT  GETCOL       IF <25, OKAY
ROWERR B    @ROWNG       ELSE REPORT ERROR
COLERR B    @COLNG       REPORT COLUMN ERROR
GETCOL
       MOV  R4,@ROW      PLACE ROW # AT LOCATION ROW
       INC  R1           NEXT PARAM (COLUMN)
```

```
        BLWP @NUMREF      GET NUMBER
        BLWP @XMLLNK      USE XML
        DATA CFI          CONVERT TO INTEGER
        MOV  @FAC,R4      GET INTO R4
        JLT  COLERR       IF NEG, ERROR
        JEQ  COLERR       IF ZERO, ERROR
        CI   R4,28        COMPARE TO 28
        JGT  COLERR       IF >28, ERROR
        MOV  R4,@COL      MOVE TO COL
GETCLR  INC  R1           NEXT PARAM
        BLWP @NUMREF      CLEAR SCREEN?
        BLWP @XMLLNK      USE XML
        DATA CFI          CONVERT TO INTEGER
        MOV  @FAC,R4      MOVE TO R4
        MOV  R4,@CLRFLG   PLACE AT FLAG LOCATION
        JEQ  NOCLR        IF ZERO, DONT
        CI   R4,1         IS VALUE 1?
        JGT  CLRALL       IF GREATER, JUMP
        CLR  R0           ELSE CLEAR R0
        LI   R1,BLNKLN    POINT AT BLANK LINE
        LI   R2,32        32 CHARACTERS
        LI   R4,24        24 ROWS
CLSLP   BLWP @VMBW60      WRITE 32 SPACES TO SCREEN
        A    R2,R0        MOVE DOWN 1 ROW
        DEC  R4           DEC ROW COUNT
        JNE  CLSLP        IF NOT ZERO, REPEAT
        CLR  R0           RE-CLEAR R0
        JMP  NOCLR        JUMP AHEAD
CLRALL
        BL   @CLRXB       USE SUBROUTINE
NOCLR
        LI   R1,4         PROMPT PARAMETER
        LI   R2,PRMSTR    POINT AT PROMPT STRING
        MOVB @MAXLEN,*R2  PLACE MAX LENGTH NUMBER THERE
        BLWP @STRREF      GET THE STRING
        MOVB @PRMSTR,R4   GET ACTUAL LENGTH IN R4
        SRL  R4,8         RIGHT JUSTIFY
        CI   R4,29        COMPARE TO 29
        JLT  LENOK        IF LESS, LENGTH IS OKAY
        B    @PRMNG       ELSE BRANCH TO ERROR REPORT
LENOK   MOV  R4,@PRMLEN   STASH LENGTH
        INC  R1           NEXT PARAM
        BLWP @NUMREF      ALLOWED INPUT LENGTH
        BLWP @XMLLNK      USE XML
        DATA CFI          CONVERT TO INTEGER
        MOV  @FAC,R4      MOV TO R4
        JGT  C256         IF POSITIVE, JUMP
        JEQ  CHRERR       IF ZERO, ERROR
        INC  @DEFFLG      SET DEFAULT FLAG
        NEG  R4           NEGATE R4 (MAKES IT POSITIVE NUMBER)
```

```
C256   CI   R4,256       COMPARE TO 256
       JLT  ARG4OK       IF LESS, OKAY
CHRERR B    @CHRNG       ELSE ERROR
ARG4OK
       MOV  R4,@MAXCHR   STASH ALLOWED
       INC  R1           NEXT PARAMETER
GARG5  MOVB @ARG5ID,R4   GET PARAMETER TYPE IN R4
       SRL  R4,8         RIGHT JUSTIFY
       CI   R4,4         COMPARE TO 4
       JLT  FNDTY        IF LESS, PARAMETER IS OKAY
       B    @PTNG        ELSE ISSUE ERROR
FNDTY
       ANDI R4,1         MASK ALL BUT LAST BIT
       JNE  GETSTR       IF NOT ZERO, JUMP
       INC  @NUMFLG      ELSE THIS IS A NUMERIC PARAMETER
       MOV  @THIR2,@MAXCHR SET MAX CHARACTERS AT 32
       BLWP @NUMREF      GET NUMBER FROM XB
       MOVB @ONE,@FAC11  SET FAC+11 TO ZERO
       CLR  @STATUS      CLEAR GPL STATUS BYTE
       BLWP @GPLLNK      USE GPL LINK
       DATA CNS          TO CONVERT NUMBER TO STRING
       MOVB @FAC12,R4    GET STRING LENGTH
       LI   R10,TEMSTR   POINT AT TEMPORARY STORAGE
       MOVB R4,*R10+     MOVE LENGTH TO THAT LOCATION
       SRL  R4,8         RIGHT JUSTIFY LENGTH
       MOVB @FAC11,R9    GET LOW BYTE OF ADDRESS
       SRL  R9,8         RIGHT JUSTIFY
       AI   R9,>8300     ADD >8300 HIGH BYTE
       CB   *R9,@MINUS   SEE IF A NEGATIVE NUMBER
       JEQ  MOVB         IF SO, JUMP
       INC  R9           ELSE POINT AHEAD BY ONE BYTE
       DEC  R4           DECREMENT LENGTH COUNT
       MOVB @WS+9,@TEMSTR  MOVE LOW BYTE OF R4 TO LENGTH BYTE TEMSTR
MOVB   MOVB *R9+,*R10+   TAKE ONE BYTE OF STRING
       DEC  R4           DECREMENT LENGTH COUNT
       JNE  MOVB         IF NOT ZERO, REPEAT
       JMP  PRMOK        ELSE JUMP AHEAD
GETSTR LI   R2,TEMSTR    POINT AT TEM STRING SPACE
       MOVB @MAXLEN,*R2  PUT MAX LENGTH THERE
       BLWP @STRREF      GET THE STRING FROM XB
PRMOK
       MOVB @TEMSTR,R4   GET BACK STRING LENGTH
       SRL  R4,8         RIGHT JUSTIFY
       C    R4,@MAXCHR   COMPARE TO MAX NUMBER CHARACTERS
       JEQ  SLOK         IF EQUAL, OKAY
       JLT  SLOK         IF LESS THAN, OKAY
       MOV  @MAXCHR,R4   ELSE TRUNCATE STRING
SLOK   MOV  R4,@DEFLEN   MOVE R4 TO DEFAULT STRING LENGTH
       SWPB R4           SWAP BYTES
       MOVB R4,@TEMSTR   PLACE AT TEMP STRING
```

```
       MOV  @ROW,R0      GET ROW VALUE INTO R0
       MOV  @PRMLEN,R1   MOVE PROMPT LENGTH TO R1
       MOV  @MAXCHR,R2   MOVE MAX CHARACTERS TO R2
       MOV  @DEFLEN,R3   GET DEFAULT LENGTH IN R3
       C    R2,R3        COMPARE
       JGT  R2OK         IF GREATER, JUMP
       MOV  R3,R2        ELSE REPLACE R2 WITH R3
R2OK
       A    R1,R2        ADD PROMPT LENGTH
       A    @COL,R2      ADD COLUMN
       INC  R2           THEN ADD ONE
* R2 HAS SUM OF COLUMN, AND LENGTHS
       MOV  R2,@TOTLEN   MOVE TO TOTAL LENGTH WORD
       CLR  R1           CLEAR R1
       DIV  @TWEN8,R1    DIVIDE R1-R2 BY 28
       MOV  R2,R2        CHECK REMAINDER
       JEQ  R1OK         IF ZERO, OKAY
       INC  R1           ELSE INCREMENT R1
R1OK
* R1 HAS NUMBER OF ROWS REQUIRED FOR
* PROMPT AND INPUT
       MOV  R1,@TOTROW   MOVE TO TOTAL ROWS LOCATION
       A    R0,R1        ADD START ROW
       CI   R1,25        WILL THIS GO PAST 24?
       JLT  ROWOKY       IF LESS, OKAY
       AI   R1,-24       ELSE SUBTRACT 24
       S    R1,@ROW      THEN SUBTRACT THAT FROM ROW
ROWOKY
       MOV  @ROW,R9      GET ROW INTO R9
       DEC  R9           DECREMENT
       MPY  @THIR2,R9    MULTIPLY BY 32
       MOV  R10,R14      SAVE PRODUCT IN R14
       MOV  @COL,R9      GET COLUMN
       INC  R9           INCREMENT
       A    R9,R10       ADD TO PRODUCT OF ROW-1 X 32
       MOV  R10,R0       MOVE TO R0 - STARTING SCREEN POSITION
       MOV  R10,@STPRM   STASH AT LOCATION STPRM
       MOV  @TOTLEN,R2   GET TOTAL LENGTH INTO R2
       MOV  @TOTROW,R4   MOVE TOTAL ROWS TO R4
       LI   R1,4         LOAD 4 INTO R1
       MPY  R1,R4        MULTIPLY R4 BY 4
       A    R5,R2        ADD PRODUCT TO R2
       MOVB @SPACE,R1    LOAD R1 WITH SPACE BYTE
       MOV  R0,R13       STASH R0 IN R13
CLRLP  BLWP @VSBW        WRITE ONE SPACE
       INC  R0           INCREMENT SCREEN LOCATION
       DEC  R2           DECREMENT COUNT
       JNE  CLRLP        IF NOT ZERO, REPEAT
       DEC  R0           DECREMENT LOCATION
       MOV  R0,@ENDOC    STASH AT ENDOC
```

```
       AI   R14,SCRWID   ADD SCREEN WIDTH TO R14
       DECT R14          THEN SUBTRACT 2
       MOV  R14,R0       MOVE THIS TO R0
       MOV  @TOTROW,R4   MOVE TOTAL ROWS TO R4
       LI   R1,EDGE      POINT AT EDGE CHARACTERS
       LI   R2,4         FOUR OF THEM
EDGELP
       BLWP @VMBW        WRITE FOUR EDGE CHARS
       AI   R0,32        MOVE DOWN ONE ROW
       CI   R0,23*32     COMPARE TO ROW 24
       JLT  DEC4         IF LESS, JUMP
       DECT R2           ELSE SUBTRACT 2 FROM R2
       JMP  LASEDG       THEN JUMP
DEC4   DEC  R4           DECREMENT ROW COUNT
       JNE  EDGELP       IF NOT ZERO, JUMP
LASEDG BLWP @VMBW        WRITE LAST TWO EDGE CHARACTERS
       DEC  R0           DECREMENT POSITION
       MOV  R0,@LASPOS   STASH AT LASPOS
* NEXT SECTION PUTS PROMPT ON SCREEN
       MOV  @STPRM,R0    GET PROMPT START POSITION
       LI   R9,PRMSTR    POINT AT PROMPT STRING
       MOVB *R9+,R2      GET ITS LENGTH
       SRL  R2,8         RIGHT-JUSTIFY
       JNE  PRDLP        IF NOT ZERO, OKAY
       MOV  @STPRM,@STDOC ELSE MOVE START OF PROMPT TO STDOC
       JMP  PUTDEF       THEN JUMP AHEAD
PRDLP
       BLWP @VSBR        READ THE CHARACTER AT CURRENT SCREEN POSITION
       CB   R1,@EDGE     IS THAT EDGE CHARACTER?
       JEQ  SKIPW        IF SO, SKIP IT
       MOVB *R9+,R1      ELSE GET PROMPT CHARACTER
       AB   @OFFSET,R1   ADD OFFSET
       BLWP @VSBW        THEN WRITE
       DEC  R2           DEC CHARACTER COUNT
       JEQ  PRDX         IF ZERO, EXIT
SKIPW
       INC  R0           INC SCREEN LOCATION
       JMP  PRDLP        THEN REPEAT ABOVE
PRDX
       INC  R0           INC SCREEN LOCATION
       MOV  R0,@STDOC    MOVE TO STDOC
PUTDEF
       MOV  @STDOC,R0    GET STDOC BACK INTO R0
CKED   BLWP @VSBR        READ CHARACTER
       CB   R1,@EDGE     IS THAT EDGE?
       JNE  PDEF1        IF NOT, GO AHEAD
       INC  R0           ELSE POINT AT NEXT SCREEN LOCATION
       JMP  CKED         THEN REPEAT
PDEF1
       MOV  R0,@STDOC    STASH R0 AT STDOC
```

```
        MOV  @DEFFLG,R4   GET DEFAULT FLAG IN R4
        JNE  PDEF0        IF NON-ZERO, JUMP
        MOV  R0,@STDOC    AND STASH
        JMP  GETIN        THEN JUMP AHEAD
PDEF0   LI   R9,TEMSTR    POINT AT DEFAULT STRING
        MOVB *R9+,R2      GET LENGTH INTO R2
        SRL  R2,8         RIGHT JUSTIFY
        JEQ  GETIN        IF ZERO, JUMP AHEAD
DEFLP
        BLWP @VSBR        READ CHARACTER FROM SCREEN
        CB   R1,@EDGE     IS THAT EDGE?
        JEQ  SKIPD        IF SO, SKIP
        MOVB *R9+,R1      ELSE GET STRING CHAR INTO R1
        AB   @OFFSET,R1   ADD OFFSET
        BLWP @VSBW        WRITE THAT
        DEC  R2           DEC COUNT
        JEQ  GETIN        IF ZERO, JUMP AHEAD
SKIPD   INC  R0           INCREMENT SCREEN POS
        JMP  DEFLP        THEN JUMP BACK

GETIN
        MOV  @STDOC,R0    SET FOR START OF INPUT FIELD
        MOV  @MAXCHR,R2   MOVE MAX CHARACTERS TO R2
GETINL  BLWP @VSBR        READ CHARACTER
        INC  R0           INCREMENT POINTER
        CB   R1,@EDGE     IS THAT EDGE?
        JEQ  SKIP3        IF SO, SKIP
        DEC  R2           ELSE DEC COUNT
        JEQ  GETIN0       IF ZERO, JUMP
SKIP3
        JMP  GETINL       ELSE REPEAT
GETIN0
        MOVB @ARGNUM,R4   GET NUMBER OF ARGUMENTS INTO R4
        SRL  R4,8         RIGHT JUSTIFY
        CI   R4,7         COMPARE TO 7
        JLT  NOBEEP       IF LESS, NO BEEP ISSUED
        CLR  @STATUS      CLEAR GPL STAT
        BLWP @GPLLNK      USE GPLLNK
        DATA >34          FOR BEEP SOUND
NOBEEP
        DEC  R0           DECREMENT SCRN LOCATION
        MOV  R0,@ENDOC    MOVE TO ENDOC
        MOV  R0,@>832A    MOVE TO HEX 832A
        MOV  R0,@>835E    AND HEX 835E
        MOV  @STDOC,R0    GET START OF INPUT POSITION
        MOV  R0,@>8320    MOVE TO HEX 8320
        MOVB R0,@>8361    MOVE HIGH BYTE TO HEX 8361
        MOVB @WS+1,@>8362 LOW BYTE TO HEX 8362
        CLR  @STATUS      CLEAR GPL STATUS
        BLWP @GPLLNK      USE GPL LINK
```

```
       DATA >285A         LINE EDITOR FUNCTION
       MOV  @ENDOC,R0     GET END OF INPUT FIELD INTO R0
FLCLP
       C    R0,@STDOC     COMPARE TO START OF FIELD
       JLT  NULIN         IF LESS, WE HAVE A NULL INPUT
       BLWP @VSBR         READ THE CHARACTER
       CB   R1,@EDGE      IS THAT EDGE?
       JEQ  DECZER        IF SO, JUMP
       CB   R1,@SPACE     IS IT A SPACE
       JNE  SETEND        IF NOT, JUMP AHEAD
DECZER DEC  R0            ELSE MOVE R0 BACK ONE BYTE
       JMP  FLCLP         THEN JUMP TO REPEAT
SETEND
       MOV  R0,@ENDOC     R0 MARKS END OF INPUT CONTENT
       MOV  @STDOC,R0     GET START OF INPUT FIELD
       LI   R9,TEMSTR+1   POINT AT TEMP STRING PLUS ONE
       CLR  R2            CLEAR REG 2
RDLP
       BLWP @VSBR         READ ONE CHARACTER
       CB   R1,@EDGE      COMPARE TO EDGE
       JEQ  SKPRD         IF EQUAL, SKIP IT
       SB   @OFFSET,R1    SUBTRACT OFFSET
       MOVB R1,*R9+       MOVE TO TEMSTR LOCATION POINTED BY R9
       INC  R2            INC COUNT OF CHARACTERS
SKPRD  INC  R0            POINT AT NEXT SCREEN LOCATION
       C    R0,@ENDOC     COMPARE TO END OF INPUT
       JGT  STROUT        IF GREATER, JUMP
       JMP  RDLP          ELSE READ MORE
NULIN  CLR  R2            SET R2 TO 0
STROUT
       SWPB R2            SWAP BYTES
       MOVB R2,@TEMSTR    MOVE TO LENGTH BYTE AT TEMSTR
       MOV  @NUMFLG,R4    GET NUMERIC FLAG
       JNE  NUMOUT        IF NOT ZERO, JUMP
       CLR  R0            ELSE CLEAR R0
       LI   R1,6          PARAMETER 6
       LI   R2,TEMSTR     POINT AT TEMSTRING
       BLWP @STRASG       ASSIGN TO VARIABLE IN XB
       JMP  EXIT          THEN EXIT ROUTINE
NUMOUT SWPB R2            SWAP SO R2=LENGTH
       MOV  R2,R2         MOVE R2 TO ITSELF
       JNE  GDNUM         IF NON-ZERO, GOOD NUMBER
       CLR  @FAC          ELSE CLEAR FLOATING POINT ACCUMULATOR
       CLR  @FAC+2        AND THE SECOND WORD
       JMP  ASG           THEN JUMP AHEAD
GDNUM  LI   R0,>1000      POINT R0 AT >1000
       MOV  R2,R9         MOVE R2 INTO R9
       LI   R1,SV1000     POINT AT TEMPORARY STORAGE
       LI   R2,33         33 CHARACTERS
       BLWP @VMBR         READ 33 CHARACTERS
```

```
        LI   R1,TEMSTR+1  POINT AT CONTENT OF STRING
        MOV  R9,R2        GET LENGTH BACK INTO R2
        BLWP @VMBW        WRITE STRING TO VDP
        MOV  R0,@FAC12    PLACE ADDRESS >1000 AT FAC+12
        A    R2,R0        ADD LENGTH TO R0
        MOVB @ANYKEY,R1   PUT A SPACE IN R1
        BLWP @VSBW        WRITE A SPACE AFTER STRING IN VDP
        BLWP @XMLLNK      USE XML LINK
        DATA CSN          TO CONVERT TO NUMBER
ASG     CLR  R0           CLEAR R0
        LI   R1,6         6TH PARAMETER
        BLWP @NUMASG      ASSIGN NUMBER AT FAC TO XB VARIABLE
        LI   R0,>1000     POINT AT >1000
        LI   R1,SV1000    STASHED CONTENT
        LI   R2,33        33 BYTES
        BLWP @VMBW        WRITE BACK
EXIT    LWPI GPLWS        LOAD GPL WORKSPACE
        B    @>006A       BRANCH TO GPL INTERPRETER
ERREX
        BLWP @PRSTR       PRINT ERROR MESSAGE
        LI   R0,23*32+3   SET ROW 24, COL 3
        LI   R1,PAK       POINT TO "PRESS ANY KEY"
        BLWP @PRSTR       DISPLAY THAT
KEY     BLWP @KSCAN       SCAN KEYBOARD
        CB   @ANYKEY,@STATUS HAS KEY BEEN PRESSED?
        JNE  KEY          IF NOT, SCAN AGAIN
        LI   R0,22*32     ELSE SET TO ROW 23
        LI   R2,64        TWO ROWS TO WRITE
        LI   R1,SAVBOT    FROM SAVBOT
        BLWP @VMBW        WRITE BACK
        CLR  R0           CLEAR R0
        MOVB @ARGNUM,R1   GET NUMBER ARGS
        SRL  R1,8         RIGHT JUSTIFY
        INC  R1           INC - NON EXISTENT PARAMETER
        BLWP @NUMASG      ASSIGN NUMBER
        JMP  EXIT         THEN JUMP TO EXIT
ROWNG
        BL   @CLRBOT      CLEAR BOTTOM OF SCREEN
        LI   R0,22*32+5   POINT AT ROW 23, COL 6
        LI   R1,ROWSTR    AT ROW MESSAGE
        JMP  ERREX        JUMP BACK
COLNG
        BL   @CLRBOT
        LI   R0,22*32+3   ROW 23, COL 4
        LI   R1,COLSTR    COLUMN ERROR
        JMP  ERREX        JUMP
PRMNG
        BL   @CLRBOT
        LI   R0,22*32+3   ROW 23, COL 4
        LI   R1,PTLSTR    PROMPT ERROR
```

```
       JMP  ERREX
CHRNG
       BL   @CLRBOT
       LI   R0,22*32+3
       LI   R1,TMCSTR    TOO MANY CHARACTERS (OVER 255)
       JMP  ERREX
PTNG
       BL   @CLRBOT
       LI   R0,22*32+4
       LI   R1,PARSTR    WRONG PARAMETER TYPE
       JMP  ERREX
*
* FOLLOWING ARE SUBROUTINES USED BY THE
* MAIN CODE SECTION ABOVE
*
CLRBOT
       LI   R0,22*32     POINT AT ROW 23, COL 1
       LI   R1,SAVBOT    AND AT MEM LOCATION
       LI   R2,64        TWO ROWS
       BLWP @VMBR        READ WHAT'S THERE
       LI   R1,BLNKLN    POINT AT BLANK LINE
       SRL  R2,1         CUT R2 IN HALF
       BLWP @VMBW60      WRITE ONE BLANK ROW
       A    R2,R0        ADD ONE ROW
       BLWP @VMBW60      WRITE ANOTHER
       RT
CLRXB  CLR  @STATUS      SUBROUTINE
       BLWP @GPLLNK      CLEAR
       DATA >27E3        TO XB DEFAULTS
       RT                RETURN
*
* PRSTR UTILITY - COURTESY HARRY WILHELM
*
PRSTR  DATA >2038,PRSTR1 UTILITY BLWP VECTOR
VMBW60 DATA >2038,VMBW61 UTILITY BLWP VECTOR
*
PRSTR1 BL   @>24CA       USE A SUBROUTINE TO PASS PARAMETERS FROM CALLING WS
       MOVB *R1+,R2      GET STRING LENGTH BYTE INTO R2
       SRL  R2,8         RIGHT JUSTIFY
       JEQ  VMBW6X       IF ZERO, SKIP THE STRING, IT HAS NULL LENGTH
       JMP  VMBW62       ELSE JUMP AHEAD
VMBW61 BL   @>24CA       USE SUBROUTINE TO PASS PARAMETERS
VMBW62 MOVB *R1+,R3      MOVE A BYTE INTO R3
       AI   R3,>6000     ADD THE OFFSET FOR XB
       MOVB R3,@>8C00    PLACE AT VDPWD LOCATION
       DEC  R2           DECREMENT CHARACTER COUNT
       JNE  VMBW62       IF NOT ZERO, SEND ANOTHER CHARACTER
VMBW6X RTWP              ELSE RETURN TO CALLERS WS AND CODE
*  GENERAL PURPOSE GPL LINK
*  FOR USE UNDER EXTENDED BASIC
```

```
*
* (INCLUDE HERE THE GPLLNK SHOWN ABOVE)
* (BE SURE NOT TO COPY THE END DIRECTIVE)
*
* FOLLOWING IS THE DATA SECTION
* FOR THE ABOVE CODE
*
EDGE   DATA >7F7F,>7F7F  EDGE CHARS WITH OFFSET
WS     BSS  32           OUR WORKSPACE
ENDOC  DATA 0            STORAGE FOR END OF INPUT FIELD
STDOC  DATA 0            STORAGE FOR START OF INPUT
STPRM  DATA 0            START OF PROMPT
ROW    DATA 0            ROW OF SCREEN
COL    DATA 0            COLUMN OF SCREEN
ONE    DATA 1            THE NUMBER ONE
NUMFLG DATA 0            FLAG FOR NUMERIC VARIABLE
THIR2  DATA 32           THIRTY TWO
DEFFLG DATA 0            FLAG FOR DISPLAY OF DEFAULT
LASPOS DATA 0
DEFLEN DATA 0            LENGTH OF DEFAULT STRING
PRMLEN DATA 0            LENGTH OF PROMPT
TWEN8  DATA 28           TWENTY-EIGHT
MAXCHR DATA 0            MAX CHARACTERS TO ACCEPT
TOTROW DATA 0            TOTAL ROWS TO BE USED
TOTLEN DATA 0            TOTAL LENGTH
CLRFLG DATA 0            FLAG FOR SCREEN CLEARING
BLNKLN TEXT '                                ' 32 SPACES
PRMSTR BSS  30           STORAGE FOR PROMPT STRING
TEMSTR BSS  256          STORAGE FOR IN-OUT STRING
MINUS  TEXT '-'          MINUS SIGN
SPACE  BYTE >80          SPACE CHARACTER WITH OFFSET
OFFSET BYTE >60          OFFSET FOR BASIC
MAXLEN BYTE 255          MAXIMUM STRING LENGTH
ANYKEY BYTE 32           KEY STROKE COMPARISON
TMCSTR BYTE 26           LENGTH OF MESSAGE
       TEXT 'CHARACTERS 1 THRU 255 ONLY'  MESSAGE
PAK    BYTE 25           LENGTH
       TEXT 'PRESS ANY KEY TO CONTINUE' MESSAGE
PTLSTR BYTE 25           LENGTH
       TEXT 'PROMPT STRING IS TOO LONG' MESSAGE
ROWSTR BYTE 21
       TEXT 'ROW MUST BE 1 THRU 24'
COLSTR BYTE 24
       TEXT 'COLUMN MUST BE 1 THRU 28'
PARSTR BYTE 23
       TEXT 'PARAMETER IS WRONG TYPE'
SAVBOT BSS  31           MEMORY TO SAVE SCREEN BOTTOM TWO ROWS
SV1000 BSS  33           SAVING >1000 BYTES
       END
```

1.36. The Art Of Assembly — Part 36. We Keep Learning

By Bruce Harrison

We've all seen advertisements from institutions of higher learning for their "continuing education" programs. Many of these are aimed directly at people like your author, too old to start over, and too tired to go for the Doctorate degrees, yet willing and able to learn some new skill or other. We can resist the temptation to pay anyone money to teach us new things, because we learn new tricks every day in the business of Assembly, and that's enough "continuing education" for us.

Sometimes the things we learn are those that we "should have known" all along, but never did learn because we were not paying enough attention, or just failed to make a connection among various facts. Today we'll start with one of those cases.

1.36.1. Do We Feel Stupid?

Yes, today we feel stupid. This is to correct ourselves on something stupid that we did in this column, dating all the way back to Part 8 of the series. In Part 8, we discussed the detection and handling of file errors. In that column, we boldly asserted that the way we used to find an "open" error was the only sure way to do so. Sometimes it pays to go back to the fundamentals, and that's what we did in writing our mini-series for beginners. To briefly refresh your memory, what we did in Part 8 was to show how to detect an error when opening a file by doing this:

```
            BLWP @DSRLNK            use DSR linkage
            DATA 8                  requ'd data
            STST R14                get status register in R14
            ANDI R14,>2000          mask all but Equal bit
            JEQ  OPNOK              if zero, file opened
            B    @OPNERR            else an error in open operation
OPNOK       (program continues)
```

DUMB! That was really DUMB! STUPID! IDIOTIC! WASTEFUL! The lines with STST and ANDI were not necessary. This accomplishes the same thing:

```
            BLWP @DSRLNK
            DATA 8
            JNE  OPNOK              if not equal, okay
            B    @OPNERR            else an error in open operation
OPNOK       (program continues)
```

All we really needed was that JNE instruction. The DSR linkage routine sets the status register for us by setting bits in its own R15 before executing its RTWP instruction. If an error of any kind has occurred in the DSR operation, the "EQUAL" bit will be turned on, else that bit will be off. Thus JNE is all we needed to do, since having the equal bit turned off means that no error occurred.

We would perhaps never have thought of trying this had it not been for the exercise of thinking through just exactly how the conditional jumps use the status register. We were forced to do that thinking in order to explain to beginners what those jumps are all about in detail. Thanks to those of you who asked for a "beginners" lesson, your author learned something himself. Who says "You can't teach an old dog. . ."? This 53 year old dog is still learning things.

To prove this to ourselves, we went back to the test routines that were used in preparing Part 8, made the change shown above, and ran some more tests. This worked perfectly. This same test can detect errors in any DSRLNK operation, including read or write operations. Finding the kind of error still requires looking at the PAB+1 byte from VDP RAM, but detecting the presence of an error is as simple as the second snippet shown above. In those new tests, we used the TI DSRLNK that's built into the E/A module's utilities, and also the Don Warren/Craig Miller DSRLNK routine that's used when we're operating from Extended Basic CALL LINK. In both situations, this worked perfectly.

For your convenience, we've put this new method into today's Sidebar, along with the updated error reporting method shown earlier in Part 28 of this series, and even slightly modified that to improve its memory efficiency. The example as given opens a D/V 80 file for input, but of course it will work every bit as well for other file types depending on what's in the PAB data. (See part 9 of this series for data on using other file types.)

1.36.2. The Free Eye Test

Back when your author was "gainfully employed", we used to joke about our Employer having offered a free eye test, but "I didn't see the notice". (I also never heard the announcement on the public address system about the free hearing test.) In some ways, the E/A Manual is an "eye test". There are things in that book that one may not notice until the fiftieth reading. Such an experience happened recently, again as part of our work on the "beginner" series.

Some time back, we offered some advice about taking the keystroke from a single keypress into a register, so that it can be compared to various numbers. The example we were showing had to do with Yes/No keypress situations. The recommendation was that the contents of location KEYVAL (>8375) be put into R8 as part of the key input subroutine, then the main code could test for Y or y like this:

```
            CI   R8,89              compare to ASCII for u.c. Y
            JEQ  YES
            CI   R8,110             compare to ASCII for l.c. y
            JEQ  YES
            (else answer is taken as no)
YES         (answer was Y or y for yes)
```

What we didn't notice was that we didn't really have to remember the ASCII codes for Y and y. We could have written our source code this way:

```
        CI   R8,'Y'
        JEQ  YES
        CI   R8,'y'
        JEQ  YES
        (and so on)
```

That simple trick was right there in front of our eyes in the part of the manual dealing with the DATA directive, and we should have seen it and connected it to the Immediate instructions. What will happen in the above cases is that the Assembler will make the immediate values 89 and 110 for us. (We had done this in PC assembly language, and it worked there, but had never thought of trying it on our TI.)

1.36.3. Getting Even

In most cases, there is no difference between the following two ways of labeling an item:

```
THREE       DATA 3
FOUR
            DATA 4
```

There is usually no difference between putting the label on the same line as the directive or instruction, or putting the label on a line by itself. We used that word usually on purpose, because we're about to show an exception. If the location count was left at an odd address before the DATA directive, then there's a difference. Suppose we had the following two cases:

```
THREE       DATA 3
ANYKEY      BYTE >20
FOUR        DATA 4

THREE       DATA 3
ANYKEY      BYTE >20
FOUR
            DATA 4
```

In the first of these two cases, the label four will be assembled as an even address because it defines a word in the memory by the DATA directive. In the second case, the label FOUR will be assembled as the odd address one byte beyond ANYKEY. There will, in both cases, be a "wasted" byte between the byte that's set to >20 and the word that's set at the value 4, but the address of the label FOUR will be different. Let's say that the label ANYKEY is at address >A04E. The memory contents would look exactly the same for both cases:

>A04E contains >20, >A04F contains 0, >A050 contains 0, and >A051 contains 4. What's different is that, in the first case the label FOUR is the address >A050, while in the second case that label will be >A04F. This will make a terrible difference if we then use an instruction like this:

```
        MOV  @FOUR,R3
```

If our data was set up as the first example above, the value placed in R3 will be 4, which of course was our intent. In the second case, the computer will notice that this is a word move, and that the address given is an odd number. It will therefore take the two bytes starting at the next lowest address into R3, so this time R3 will be loaded with the two bytes starting at >A04E, and its value will become >2000. That's very definitely not what we intended.

Having been "bitten" once by this problem in our Code Breakers program, we are very "shy" about this kind of problem. There is one other solution to this problem, and that is to put in the EVEN directive before the label FOUR, like this:

```
THREE       DATA 3
ANYKEY      BYTE >20
            EVEN
FOUR
            DATA 4
```

Once again, the results in actual memory locations is the same as we showed earlier, except that the EVEN directive makes the Assembler's location counter advance to >A050 before it sets the location value for the label FOUR, so this method will result in correct operation for our MOV instruction.

The lesson in all this is that the EVEN directive is almost never necessary. Just making sure that the DATA or BYTE or TEXT directive is in the same source line as its label will insure that the labels which should be at even locations will in fact be assembled as even addresses. In today's Sidebar are partial listings from the Assembler, showing the three cases in detail, so you can see for yourself how the addressing works out.

As we've said earlier, our normal practice is to separate all of the data in our programs into one place in the source code, and that's usually at the end of the source, after all the code parts. Within that data section, we normally group all the DATA directives together at the beginning of the data section, so that we need never worry about there being an odd byte made into a wasted byte of zero value. (We also put any BSS directives that have even-numbered arguments into that same part of the data section, as for example our workspace as WS BSS 32.)

To be fair to TI, they did mention this very important difference between the label-only line and the label being on the same line as the DATA directive, but again this was not easily noticed where it was buried in that giant book. (It's in the last paragraph on page 47, after a "However".)

1.36.4. The Challenge

This is number 36 of The Art Of Assembly, which means that three years' worth of these columns have been written. It's hard to imagine how this can be, since it seems like only yesterday when we were writing Part 1. The challenge for someone who's been doing this for so long is to keep the column both interesting and relevant for the practitioners of this strange and difficult "art". Much material has been covered, and we are still learning things and passing those along to our readers, but we need some more "feedback" to keep going. Your author avidly reads the Reader Feedback column in *MICROpendium*, as well as the "Reader to Reader" column. That's how the "beginners" parts were inspired. If you're looking for something on our subject, and are too shy to have your concerns aired in print, you can always write to us directly at 5705 40th Place, Hyattsville MD 20781. We answer all letters, and will make every effort to find a solution to your problem. We may not always succeed, but we'll always try.

```
* SIDEBAR 36
*
* PART ONE
*
* FILE ERROR HANDLING RE-REVISITED
* THIS IS NOT COMPLETE CODE, JUST SNIPPETS
* SIMPLER METHODS FOR BOTH ERROR
* DETECTION AND ERROR REPORTING
* CODE BY B. HARRISON
* PUBLIC DOMAIN
*
* THIS OPENS AND READS A FILE RECORD
*

OPNF   LI   R0,PAB       POINT TO PAB IN VDP
       LI   R1,PABDT     AND PAB DATA
       MOVB @PABDT+9,R2  GET DESCRIPTOR LENGTH BYTE
       SRL  R2,8         RIGHT JUSTIFY IN R2
       AI   R2,10        ADD TEN FOR THE PAB ITSELF
       BLWP @VMBW        WRITE PAB DATA
       AI   R0,9         ADD NINE
       MOV  R0,@>8356    PLACE ADDRESS
       BLWP @DSRLNK      USE DSR LINKAGE
       DATA 8            REQUIRED DATA
       JNE  RDFI         IF NOT "EQUAL", OKAY
       B    @OPNERR      ELSE REPORT OPEN ERROR
RDFI   MOVB @READF,R1    SET TO READ
       LI   R0,PAB       POINT AT PAB
       BLWP @VSBW        WRITE BYTE
       AI   R0,9         ADD NINE
       MOV  R0,@>8356    PLACE ADDRESS
       BLWP @DSRLNK      USE DSR
       DATA 8            REQUIRED DATA
       JNE  READON       IF NOT "EQUAL", OKAY
       B    @FILERR      ELSE REPORT ERROR
READON (PROGRAM CONTINUES)
```

```
*
*
OPNERR
       LI   R0,22*32+6   POINT AT ROW 23, COL 7
       LI   R1,FNOTXT    FILE NOT OPENED
       LI   R2,17        17 BYTES TO WRITE
       BLWP @VMBW        WRITE MESSAGE
FILERR LI   R0,PAB+1     POINT AT PAB PLUS ONE
       BLWP @VSBR        READ A BYTE
       SRL  R1,13        SHIFT RIGHT 13 BITS
*
* AT THIS POINT YOU MIGHT WANT TO INSERT A CI R1,5
* AND JEQ TO SOMEPLACE ELSE FOR AN END-OF-FILE ERROR
*
       SLA  R1,4         SHIFT LEFT FOUR BITS (MULTIPLY BY 16)
       AI   R1,FERMSG    ADD START OF MESSAGE TABLE
       LI   R2,16        16 BYTES IN EACH MESSAGE
       LI   R0,23*32+7   POINT AT ROW 24, COL 8
       BLWP @VMBW        WRITE MESSAGE
       BL   @KEYLOO      PAUSE FOR KEYSTROKE
       B    (SOMEWHERE ELSE)
*
* DATA SECTION
*
PABDT  DATA >0014,BUF,>5050,>0000,>000F
       TEXT 'DSK1.ANYOLDFILE'
*
FERMSG TEXT 'BAD DEVICE NAME '  EACH TEXT LINE 16 BYTES
       TEXT 'WRITE PROTECTED '
       TEXT 'BAD ATTRIBUTE   '
       TEXT 'BAD OPERATION   '
       TEXT 'DISK IS FILLED  '
       TEXT 'END OF FILE     '
       TEXT 'DEVICE ERROR    '
       TEXT 'OTHER FILE ERROR'
FNOTXT TEXT 'FILE DID NOT OPEN' 17 BYTES LENGTH
*
*
* SIDEBAR PART TWO
* ASSEMBLY LISTINGS TO SHOW THE "LONE LABEL"
* EFFECT ON A DATA ITEM'S ADDRESS
* (THESE ARE PARTIAL LISTINGS, EDITED FOR CLARITY)
*
* EXAMPLE ONE - WITH LABEL AND DIRECTIVE
* ON THE SAME SOURCE LINE
* (THE RIGHT WAY)

LOCATION CONTENT LABEL  OPCODE/DIRECTIVE

0048     C0E0    MOV4   MOV  @FOUR,R3
```

```
004A     0050'
*
* NOTE THAT LABEL FOUR IS AN EVEN ADDRESS (>50)
* AND EQUALS THE LOCATION OF THE DATA ITSELF
*
004C     0003    THREE  DATA 3
004E       20    ANYKEY BYTE >20
0050     0004    FOUR   DATA 4

* EXAMPLE TWO - WITH LABEL ON SEPARATE
* SOURCE LINE FROM THE DIRECTIVE
* (THE WRONG WAY)

LOCATION CONTENT LABEL  OPCODE/DIRECTIVE

0048     C0E0    MOV4   MOV  @FOUR,R3
004A     004F'
*
* NOTE THAT LABEL FOUR IS AN ODD ADDRESS (>4F)
* AND NOT THE ADDRESS OF THE DATA ITSELF
*
004C     0003    THREE  DATA 3
004E       20    ANYKEY BYTE >20
                 FOUR
0050     0004           DATA 4

* EXAMPLE THREE - WITH LABEL ON SEPARATE
* SOURCE LINE FROM THE DIRECTIVE
* BUT USING EVEN DIRECTIVE (ANOTHER RIGHT WAY)

LOCATION CONTENT LABEL  OPCODE/DIRECTIVE

0048     C0E0    MOV4   MOV  @FOUR,R3
004A     0050'
*
* NOTE THAT LABEL FOUR IS AN EVEN ADDRESS (>50)
* AND EQUALS THE ADDRESS OF THE DATA ITSELF
*
004C     0003    THREE  DATA 3
004E       20    ANYKEY BYTE >20
                        EVEN
                 FOUR
0050     0004           DATA 4
```

1.37. The Art Of Assembly — Part 37. Option 5 Revisited

By Bruce Harrison

Some time ago, we presented in this column a couple of ways to make your Assembly program over into an Option 5 Program File format. This makes programs load much faster, and of course more conveniently than is the case for Option 3 loading. The drawback is that the utility subprograms like VMBW, VSBW, and such, do not get loaded into low memory under Option 5 as they do in an Option 3 loading process. (Funnelweb's loader for Option 5 does set things up properly, but TI's does not. Once again Tony McGovern did the right thing.) In our earlier columns on this subject, we used very crude but effective methods to load the low memory with the utilities.

Today we'll present two more methods for getting your Option 5 program to load its utilities. The first method came from work we were doing to create a compiler for Extended Basic programs. (That will be covered further next month.) In the process of our "research" toward a compiler, we were playing around with the idea of executing CALL operations from within Assembly routines. To execute a call, we used a DSRLNK, complete with a PAB and the DATA >A. What we found, in our little experiments, was that we could write an Option 3 program that would perform a CALL INIT that re-loaded everything in low memory to the normal "startup" situation for an Option 3 load. That is, the low memory would look as if no loading of the Option 3 program had taken place. It then occurred to us that this might be a way to get things set up for Option 5 programs also.

Let's digress for a moment to describe how our own TI is configured. We use a "normal" TI with two floppy drives, but have added a Horizon Ramdisk and Horizon P-Gram card. The P-Gram is kept loaded with Extended Basic and Editor/Assembler. The P-Gram also gives us the very handy CALL PG, which allows us to inspect RAM, ROM, GROM, and VDP memories very easily. Thus when we got our little CALL INIT experiment put together, we added a CALL PG to the program, so that we could go and inspect the memory as it was left by the CALL INIT operation. (The High Memory expansion and parts of VDP memory get changed by the CALL PG, but low memory is mostly left alone.) Sure enough, a quick examination of the low memory showed that after our little Option 3 program had run, low memory looked just the way it would if we went into console basic, did a CALL INIT, then a CALL PG, with the singular exception that the device name PG showed up in the low memory after the DSRLNK to perform the CALL PG.

1.37.1. Making A Call

As you can see from PART ONE of the Sidebar, the call is actually made by a DSR linkage operation, just like a file operation, except that this time the DATA statement that follows the BLWP to DSRLNJ is DATA >A, not the usual DATA 8. The reason for this is simple. In the Device ROMs and GROMs, there are linked lists for the device drivers present in that ROM or GROM. The pointer to the beginning of the list for "disk" type devices is at 8 bytes from the start of the ROM or GROM space. The list pointer for the "call" type services is at 10 (>A) bytes from the start of the ROM or GROM space. In the case of the TI Disk controller, for example, the ROM space (CRU BASE >1100) starts at >4000. The ROM contains both Disk device drivers and Call drivers, so there is a pointer at >4008 to the start of the disk devices list, and a pointer at >400A to the start of the Call drivers. In the case of the TI Disk Controller, the Call drivers include some which take care of direct sector-access operations and CALL FILES, while the Disk devices list takes care of devices DSK, DSK1, DSK2 and DSK3.

The DSR linkage actually searches these linked lists for the named device or call, as appropriate, then executes when it finds the address for the desired device or call driver. In the Sidebar, we are showing the Warren/Miller DSRLNK, which actually uses a GROM routine through GPLLNK to do this searching for us, so the searching in this case is "transparent" from the point of view of our source code. The fact that our CALL INIT worked correctly presented something of a mystery in itself.

Here's why. Our P-Gram card has all of the GROM for both Extended Basic and Editor/Assembler loaded into it. The XB part starts at >6000 and runs through >DFFF, while the E/A GROM starts at >E000 and runs through >FFFF. Why, then, when we executed our DSRLNK looking for INIT, did the DSR not find the INIT that's in the XB's GROM and execute that, instead of doing what we intended, executing the CALL INIT that's in the E/A GROM? A little probing around in the GROM has given us a potential answer to this mystery.

The linked lists of calls in all but the XB GROM are set up like this: The first pointer points to a group of bytes that looks, typically, like this:

```
>6B3E,>6B82,>04,INIT,>6B50,>6BD8,>04,LOAD...
```

In each such grouping, the first two bytes point to the next entry in the list (0000 for the last entry). The next two bytes are the address of the named routine for this entry, then there's a byte for the length of the name, followed by the name of the routine. The numbers we've shown here are the actual numbers found in an E/A's GROM. For ease of reading, we've shown the names in "Plain English", but of course in the GROM they'd show up as hex characters.

In Extended Basic's GROM, the numbers are arranged in a different order, so an entry similar to the above would look like this:

```
>C02B,>04,INIT,>C2BA,>C034,>04,PEEK,>C2CA...
```

Here, the pointer to the next entry is followed immediately by the name length, then the name, and after that is the address for the routine. Thus, when the search is taking place, the DSR doesn't find the name where it should be in the linked list, so each item found in the XB GROM gets ignored, and the search continues until it finds INIT "correctly" listed in the E/A's GROM. Obviously the XB interpreter must execute its own search for call names in a different fashion, so it can find its own calls, not those in other GROMs. (The calls linked list in Basic's GROM are arranged like those in E/A's GROM.)

It was probably arranged this way on purpose by TI, so that XB would skip over the console Basic version of any CALL name, and find its own version for execution. This difference turned out to be convenient for us, in that it meant our new method for dealing with Option 5 would work even though both the XB and E/A GROMs are "present" in our P-Gram. For ROM calls, XB seems to be able to find those even though their linked lists are constructed like the Basic and E/A GROM lists. Please don't ask just how XB manages that. We don't know!

After proving to ourselves that we could execute a CALL INIT and then a CALL PG from within an Option 3 program, we added the labels SFIRST, SLOAD, and SLAST to our source file, performed a SAVE operation to make an Option 5 Program File called OPT5, and then ran that. That worked perfectly, getting us quickly into our PG call. It worked when run directly from our Ramdisk menu, just as well as when run from Option 5 under E/A. Having had some success, we decided to try another experiment, and found a flaw in our "perfect" solution.

Instead of doing a CALL PG, we decided to try simply putting some text on the screen, executing a "key loop" to let us see that, then returning to E/A control. DISASTER! Everything worked until we pressed a key, then our return to E/A went bonkers. A bunch of stuff appeared, scrolled up the screen, and then the screen went blank and eventually returned us to "startup" conditions, with our Ramdisk menu on the screen.

The only "safe" return from this little experiment was to BLWP @0. That did what it should do, returning us to the Ramdisk menu without any intervening trouble. We assume that this return problem has something to do with the GROM address, but saving and restoring that did absolutely nothing. Obviously, TI's "hidden agenda" has struck us again, and we don't know why.

So long as you don't need to return to E/A at the end of your program, the CALL INIT process will work just fine to load the E/A utilities for your Option 5 program. Just be sure that any exit point executes a BLWP @0, and you'll go safely back to startup condition, with the appropriate title screen or menu, depending how you're equipped.

Having the P-Gram available makes our life easier in many ways, and we'd like to thank Bud Mills and company for this nifty little card, and especially for the PG call that makes probing into TI's secrets much easier. While we were at it, we decided to find out just where E/A's CALL INIT gets its data to dump into the low memory. It was easy enough to find, using the "search" function from the PG call.

We found the data that's dumped into the low memory starting at >F000 in our P-Gram's GROM memory. In the normal E/A cartridge, we found this data at GROM address >7000. The data is there in the form of three strings, each with a "length" word, then a target location word, followed by the string of data itself. The word at >7000 (or >F000) contains the number 8, followed by a word containing >2000, then the eight bytes that the INIT routine places at that location. The next two bytes are the length of the second string (>654), followed by the destination for that string (>2022), and then the 1620 bytes that go into low memory starting at >2022. Finally, after those 1620 bytes is the length (>00C8) and destination (>3F38) for the initial state of the REF/DEF table that goes at the high end of the low memory. This is followed by those definitions for all the pre-defined labels like VMBW, VSBW, DSRLNK, and so on.

1.37.2. Method Two

Having thus found the data that's used by CALL INIT, and doped out how it's organized, we have a second method for putting this data into low memory. We can take the length from the first two bytes of the GROM at >7000 (or >F000), place that into a register, take the target address from the next two bytes, place that in another register, then simply loop a GROM read cycle for the number of times in that first register, and we'll have what we were after for the first "string". When that finishes, we'll be "pointing" at the first of the two bytes for the second group's length. Thus we can make a nested loop to take the three strings needed as shown in today's Sidebar. This will work fine for the case of a user with an actual E/A cartridge, and will not cause a problem upon return to E/A, but will cause a problem when the user has a P-Gram with E/A starting at a different address in GROM space.

Since we don't know, in general, whether the user of our program will be using an actual E/A module, or some device like the P-Gram, we've added a routine in PART TWO of the Sidebar to search the GROM address space until the E/A GROM is found, and then advance the address to the "equivalent" of >7000 in the real E/A module.

What's shown there is a kind of shortcut method. Starting at >6000, it first checks the 0th byte in that block for >AA, and if that's okay, it gets the address from the 06 and 07 bytes for the title "lookup table" starting point. Since the E/A GROM has only one title, we can take that address from the 06/07 bytes, then just add five so we're pointing at the title itself. Here there's a comparison loop that's another shortcut, in that it checks the title from the GROM for only the first four letters, to wit "EDIT". That's more than enough to distinguish the E/A module from TI EXTENDED BASIC or TI-WRITER, and such. The loop that compares the title is set up so that as soon as a byte is found not to match the text at label EATITL, the program will exit the comparison loop and go on to the next >2000 group in the GROM space.

As we were getting ready to put a "wrap" on today's Sidebar, one of those "almost forgots" sprang to mind. The astute reader will now go back to Part 26 with us. If the program that's going to use this "startup" includes a REF to TI's GPLLNK, then we have to correct the GROM address for GPLLNK before the program can BLWP to it. Thus we've pulled that little trick out of Part 26 and added it to today's Sidebar PART TWO, so that the user's program can BLWP to TI's GPLLNK without any problem. At least that's what we thought, until we tried testing part two of the Sidebar with an actual GPLLNK call in it. Another disaster! It took quite a while to unravel this new mystery, but eventually we got the answer.

In our older methods, we had "captured" the low memory contents after an Option 3 loading process, so there had to be something else of great importance that we were missing. After much probing around, we found that the Option 3 load sets the word of memory at >2030 to the value >061C. That address is part of the UTLTAB area, and is called "GPL Return Address" in the big book. When we executed our Part Two code to get low memory loaded from the GROM, address >2030 was left with zero as its value. Thus we added the short section of code that puts >061C into R5, then moves that value into >2030. Hooray! This did the trick. As shown in the Sidebar, then, Part Two is a complete program that works from Option 5, puts the word "EDIT" on the screen, beeps, and then waits for a keypress, after which it exits to E/A's PRESS ANY KEY TO CONTINUE prompt. In other words, it does exactly what we intended. We tested this from Option 3, from Option 5 under E/A, with both our P-Gram E/A and a real module, and as run directly from our Ramdisk menu. Worked in all cases.

1.37.3. The Exceptions

You always know that in our column we'll point out carefully any known "exceptions" to what we've presented as a "solution", and sure enough, here's a "biggie". In the previous methods we showed for setting up the utilities in low memory, the data needed was "embedded" in the program as saved to the disk. That way, the program could be loaded (assuming some loader was at hand) and run without the E/A module being available. The method we've covered today is not like that. In the Sidebar, (PART TWO) we've shown an "emergency exit" that will get out of the program if the E/A module is not found, and that's an essential feature in any program file that works this way. We ran a test in which only the XB module was available, and sure enough the program worked as expected, exiting back to startup. Either of the two methods shown in today's Sidebar will work just fine so long as the E/A GROM is available somewhere in the GROM address space. Neither will work if E/A is not present.

We should also point out that the Part One section, which uses the Warren/Miller DSRLNK to execute the CALL INIT, will probably not work on the Geneve. So far as we know, the second method shown today will work just as well on Geneve as on a TI. We'll ask a friend who has a Geneve to test this, and in due course will let you know.

Next month we'll write some about our "impossible" project, making a Compiler for Extended Basic. If you thought this month's topic was "heavy", just wait one month, and this will look like "duck soup".

```
* SIDEBAR 37
* EXPERIMENTS WITH "INIT"
* USING THE E/A MODULE
* CODE BY B. HARRISON
*
***********************************
* FIRST PART - USING INIT AS A CALL *
* 16 JUL 1993                       *
* PUBLIC DOMAIN                     *
***********************************
*
PAB    EQU  >1000        PAB LOCATION IN VDP RAM
STATUS EQU  >837C        GPL STATUS BYTE
GPLWS  EQU  >83E0        GPL WORKSPACE
GR4    EQU  GPLWS+8      GPL REGISTER 4
GR6    EQU  GPLWS+12     GPL REGISTER 6
STKPNT EQU  >8373        STACK POINTER
LDGADD EQU  >60          LOAD GROM ADDRESS
XTAB27 EQU  >200E        STORAGE SPOT
GETSTK EQU  >166C        GET STACK
GRMRA  EQU  >9802        GROM READ ADDRESS
GRMWA  EQU  >9C02        GROM WRITE ADDRESS
       REF  VMBW,DSRLNK  REF UTILITIES
       REF  VDPWA,VDPWD  REF ADDRESSES
       DEF  PGX          DEFINE ENTRY POINT
PGX    LWPI WS           LOAD OUR WORKSPACE
       LI   R0,PAB       POINT AT PAB LOCATION
       ORI  R0,>4000     MAKE IT A WRITE ADDRESS
       SWPB R0           SEND LOW ORDER BYTE FIRST
       MOVB R0,@VDPWA    TO VDPWA
       SWPB R0           SWAP BYTES
       MOVB R0,@VDPWA    SEND HIGH ORDER BYTE
       LI   R1,PABDT     POINT AT PAB DATA
       LI   R2,15        15 BYTES TO SEND TO VDP
WMB    MOVB *R1+,@VDPWD  WRITE ONE BYTE
       DEC  R2           DECREMENT COUNT
       JNE  WMB          IF NOT ZERO, REPEAT
       LI   R0,PAB+9     GET PAB +9 ADDRESS
       MOV  R0,@>8356    PUT AT >8356
       CLR  @STATUS      CLEAR GPL STATUS
       BLWP @DSRLNJ      USE MILLER DSRLNK
       DATA >A           WITH DATA >A FOR "CALL" OPERATION
*
* CODE FROM HERE ON MAKES A CALL TO PG IN THE P-GRAM CARD
* THIS MUST BE LEFT OUT IF YOU DON'T HAVE THAT CARD
*

CALLPG LI   R0,PAB       POINT TO PAB AGAIN
       LI   R1,PABDT2    DATA FOR CALL PG
       LI   R2,15        15 BYTES
```

```
       BLWP @VMBW        USE THE VMBW THAT INIT HAS PUT IN PLACE
       AI   R0,9         ADD 9
       MOV  R0,@>8356    PUT AT >8356
       CLR  @STATUS      CLEAR GPL STATUS
       BLWP @DSRLNK      USE E/A DSRLNK
       DATA >A           WITH DATA >A TO CALL PG
*
* YOUR MAIN PROGRAM CODE WOULD GO HERE
* ANY GPLLNK CALLS MUST USE THE MILLER GPLLNK
* THAT'S ALREADY INCLUDED BELOW (NO REF TO GPLLNK)
* IT MUST EXIT WITH THE FOLLOWING:
*
EXIT   BLWP @0
*
* DATA FOR PART ONE
*
WS     BSS  32           OUR WORKSPACE
PABDT  DATA 0,0,0,0,>0004 PAB DATA FOR 'INIT'
       TEXT 'INIT  '     FILE NAME
PABDT2 DATA 0,0,0,0,>0002 PAB DATA FOR 'PG'
       TEXT 'PG    '     FILE NAME
*
* FOLLOWING IS THE WARREN/MILLER GPLLNK AND DSRLNK
*
GPLLNK DATA GLNKWS       UTILITY VECTOR - WORKSPACE
       DATA GLINK1       VECTOR CODE WORD
RTNAD  DATA XMLRTN       R9 OF GLNKWS
GXMLAD DATA >176C        R10 OF GLNKWS
       DATA >50          R11 OF GLNKWS
GLNKWS EQU  $->18
       BSS  >08
GLINK1 MOV  *R11,@GR4    MOV @>50,@GR4
       MOV  *R14+,@GR6   MOV DATA INTO GR6
       MOV  @XTAB27,R12  STASH VALUE FROM >200E
       MOV  R9,@XTAB27   PLACE ADDRESS XMLRTN AT >200E
       LWPI GPLWS        LOAD GPL WS
       BL   *R4          BL
       MOV  @GXMLAD,@>8302(R4) PLACE >176C AT >8302 +
       INCT @STKPNT      INC BY TWO AT >8373
       B    @LDGADD      B @ >60
XMLRTN MOV  @GETSTK,R4   MOV @>166C, R4
       BL   *R4          BL THERE
       LWPI GLNKWS       LOAD GLNKWS
       MOV  R12,@XTAB27  REPLACE VALUE FROM >200E
       RTWP              RETURN WITH WORKSPACE POINTER
PUTSTK EQU  >50
TYPE   EQU  >836D
NAMLEN EQU  >8356
VWA    EQU  >8C02
VRD    EQU  >8800
```

```
GR4LB  EQU  >83E9
GSTAT  EQU  >837C

DSRLNJ DATA DSRWS,DLINK1  RE-NAMED DSRLNJ, NOT TO CONFLICT

DSRWS  EQU  $
DR3LB  EQU  $+7
DLINK1 MOV  R12,R12
       JNE  DLINK3
       LWPI GPLWS
       MOV  @PUTSTK,R4
       BL   *R4
       LI   R4,>11
       MOV  R4,@>402(R13)
       JMP  DLINK2
       DATA 0
       DATA 0,0,0
DLINK2 MOVB @GR4LB,@>402(R13)
       MOV  @GETSTK,R5
       MOVB *R13,@DSRAD1
       INCT @DSRADD
       BL   *R5
       LWPI DSRWS
       LI   R12,>2000
DLINK3 INC  R14
       MOVB *R14+,@TYPE
       MOV  @NAMLEN,R3
       AI   R3,-8
       BLWP @GPLLNK
DSRADD BYTE >03
DSRAD1 BYTE 00
       MOVB @DR3LB,@VWA
       MOVB R3,@VWA
       SZCB R12,R15
       MOVB @VRD,R3
       SRL  R3,5
       MOVB R3,*R13
       JNE  SETEQ
       COC  @GSTAT,R12
       JNE  DSREND
SETEQ  SOCB R12,R15
DSREND RTWP
       END
*
*************************
****  END OF PART ONE  ****
*************************
*
*********************************
* SECOND PART - GETTING UTILITIES *
```

```
* DIRECTLY FROM THE E/A GROM      *
* 18 JULY 1993                    *
* PUBLIC DOMAIN                   *
***********************************
*
GRMRA  EQU  >9802        THE GROM READ ADDRESS
GRMWA  EQU  >9C02        THE GROM WRITE ADDRESS
GRMRD  EQU  >9800        THE GROM "READ DATA" ADDRESS
       REF  KSCAN,VMBW,GPLLNK REFERENCE VECTORS
       DEF  SFIRST,SLAST,SLOAD  DEFS FOR OPTION 5
       DEF  INITX3       DEFINE ENTRY POINT
*
* THE "LONE LABELS" SFIRST AND SLAST GET THE SAME
* ADDRESS AS INITX3, SO THAT THE "SAVE" UTILITY
* WILL WORK CORRECTLY
*
SFIRST
SLOAD
INITX3 LWPI WS           LOAD OUR WORKSPACE
       LI   R3,>6000     POINT AT >6000 IN GROM
NXTGRM MOVB R3,@GRMWA    WRITE HIGH BYTE OF ADDRESS
       SWPB R3           SWAP
       MOVB R3,@GRMWA    WRITE LOW BYTE OF ADDRESS
       SWPB R3           SWAP AGAIN
       MOV  R3,R4        MOVE R3 INTO R4
       CB   @GRMRD,@VALBYT CHECK FOR BYTE >AA AT START
       JNE  ADDGRM       IF NOT EQUAL, SKIP AHEAD
       AI   R4,6         ELSE ADD 6 TO ADDRESS IN R4
       MOVB R4,@GRMWA    SEND THE HIGH BYTE
       SWPB R4           SWAP
       MOVB R4,@GRMWA    SEND LOW BYTE
       SWPB R4           SWAP BACK
       MOVB @GRMRD,R4    READ A BYTE FROM X006 IN GROM
       SWPB R4           SWAP
       MOVB @GRMRD,R4    READ NEXT BYTE
       SWPB R4           SWAP AGAIN
       MOV  R4,R4        MOVE R4 TO ITSELF TO SET STATUS REGISTER
       JEQ  ADDGRM       IF ZERO, SKIP THIS GROM
       LI   R5,4         ELSE LOAD R5 WITH 4
       AI   R4,5         ADD 5 TO ADDRESS IN R4
       LI   R9,EATITL    POINT R9 AT TITLE "EDIT"
*
* AT THIS POINT R4 HAS ADDRESS OF THE TITLE OF THIS GROM
* WHILE R9 POINTS AT THE WORD "EDIT"
* AND R5 HAS THE NUMBER OF LETTERS TO COMPARE
*
       MOVB R4,@GRMWA    SET TO READ FROM
       SWPB R4           SWAP R4
       MOVB R4,@GRMWA    ADDRESS IN R4
       SWPB R4           SWAP R4
```

```
READTL CB   @GRMRD,*R9+  COMPARE ONE BYTE TO "EDIT"
       JNE  ADDGRM       IF NOT EQUAL, WRONG GROM
       DEC  R5           DECREMENT COUNT
       JNE  READTL       IF NOT ZERO, TRY NEXT BYTE FROM GROM
       ANDI R3,>F000     ELSE WE'VE FOUND EDITOR/ASSEMBLER
       MOV  R3,R7        STASH "ROOT" GROM ADDRESS IN R7
       AI   R3,>1000     ADD >1000 TO ROOT ADDRESS OF GROM
       JMP  GETUT        THEN GO GET THE UTILITIES
ADDGRM AI   R3,>2000     POINT AHEAD BY 2000
       JNE  NXTGRM       IF NON-ZERO, JUMP TO INSPECT NEXT BLOCK
       BLWP @0           ELSE WE HAVE NO E/A GROM
GETUT  MOVB R3,@GRMWA    LOAD GROM ADDRESS HIGH BYTE
       SWPB R3           SWAP BYTES
       MOVB R3,@GRMWA    LOAD GROM LOW ADDRESS
       LI   R5,3         SET COUNTER AT 3 (THREE BLOCKS OF DATA)
GQTY   MOVB @GRMRD,R4    GET ONE BYTE FROM GROM
       SWPB R4           SWAP
       MOVB @GRMRD,R4    GET LOW ORDER BYTE
       SWPB R4           SWAP AGAIN
       MOVB @GRMRD,R3    GET LOCATION HIGH ORDER BYTE
       SWPB R3           SWAP
       MOVB @GRMRD,R3    GET LOCATION LOW ORDER
       SWPB R3           SWAP
GETGR  MOVB @GRMRD,*R3+  GET A BYTE OF DATA, PLACE AT ADDRESS IN R3
       DEC  R4           DECREMENT NUMBER OF BYTES THIS BLOCK
       JNE  GETGR        IF NOT ZERO, DO ANOTHER
       DEC  R5           DECREMENT NUMBER OF BLOCKS
       JNE  GQTY         IF NOT ZERO, START NEXT BLOCK
*
* NEXT SECTION SETS UP THE CORRECT GROM ADDRESS FOR USING
* THE TI GPLLNK - WITHOUT THIS, OPTION 5 PROGRAMS CAN'T
* USE THAT UTILITY EVEN THOUGH IT'S LOADED INTO LOW MEMORY
*
       AI   R7,>0892     ADD OFFSET >892 TO ROOT GROM ADDRESS
       MOVB R7,@GRMWA    WRITE HIGH BYTE
       SWPB R7           SWAP BYTES
       MOVB R7,@GRMWA    WRITE LOW BYTE
       SWPB R7           SWAP BACK
       LI   R5,>061C     PUT STANDARD GPL RETURN ADR IN R5
       MOV  R5,@>2030    PLACE THAT AT >2030
*
* FROM THIS POINT ON, ALL E/A UTILITIES ARE AVAILABLE
* FOR USE BY YOUR PROGRAM, INCLUDING TI'S GPLLNK
* YOUR PROGRAM MAY EXIT THROUGH THE CODE AT QEXIT, SHOWN BELOW
*
DISPL  LI   R0,11*32+13  POINT AT ROW 12, COL 14
       LI   R1,EATITL    THE WORD "EDIT"
       LI   R2,4         FOUR CHARACTERS
       BLWP @VMBW        WRITE THOSE
       CLR  @>837C       CLEAR GPL STATUS BYTE
```

```
       BLWP @GPLLNK      USE TI'S GPLLNK
       DATA >34          TO MAKE A BEEP
SCAN   BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN SHUT THEM OFF
       CB   @ANYKEY,@>837C  HAS A KEY BEEN STRUCK?
       JNE  SCAN         IF NOT, SCAN AGAIN
QEXIT  LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         BRANCH TO GPL INTERPRETER
*
* DATA SECTION FOR PART TWO
*
WS     BSS  32           OUR OWN WORKSPACE
VALBYT BYTE >AA          VALIDATION BYTE
EATITL TEXT 'EDIT'       COMPARISON NAME FRAGMENT
ANYKEY BYTE >20          COMPARISON FOR KEYSTROKE
SLAST  EQU  $            END OF "SAVE" FOR OPTION 5
       END
*
**************************
****  END OF PART TWO  ****
**************************
```

1.38. The Art Of Assembly — Part 38. Trying The Impossible

By Bruce Harrison

It's September 1993 as we write this, and we are in the middle of an endless project. For some time now, it's been the opinion of many skilled TI programmers that it would be impossible to make a compiler for TI Extended Basic. We think that they've been wrong, and are trying to prove it. The only way to prove the experts wrong in such matters is of course to do the very thing that was declared impossible.

1.38.1. The Compiler Problem

We have used compilers for Basic programs on PC computers, and found that there's one problem that those compilers have in common. The size of the program we end up with is huge compared to what we started with. In a typical case, one may start with a program of 4000 bytes, perform the compiling and linking steps, and find the .EXE file takes over 30,000 bytes. In the most extreme example, we tried just the simplest possible kind of Basic program, with only one line, like this:

```
10 PRINT "Hello"
```

When compiled and linked, this simple program became a 29,000 byte monster. (As we recall, the original one line Basic program took all of about ten bytes.) How, you ask, can this happen? The answer lies in the approach taken to the problem of making the compiler, and in the nature of what a "program" in Basic really is.

1.38.2. What Is The Program?

From the beginning of Basic, there's a misconception that's created in our minds, to the effect that the collection of Basic instructions which we write and save to the disk is a program for the computer. It's NOT! What's called a Basic program is really just a collection of data. That data is used by a program called the Basic Interpreter to make the computer do things. In the PC case, the Basic Interpreter is a program called BASIC.EXE, of about 78,000 bytes, which must be loaded into the computer's memory before any Basic "program" can be loaded or run. The Interpreter typically includes a large number of routines in machine code, which are used in ways determined by the content of the Basic "program" that it's "running". What's really running is the interpreter itself, which looks at the contents of each statement in this data, determines what routines and with what parameters need to be executed, and executes those routines.

On the TI, the Basic Interpreter does not need to be loaded from any disk, but resides in the computer itself, or in the Cartridge for Extended Basic. The TI's case is complicated by the fact that there are really two interpretation steps required. First, the data which we call the Basic program is "translated" into a series of GPL instructions, then those are used by the GPL Interpreter. This two-stage process may have been needed to keep the memory requirements within bounds, as the GPL code is very compact, and thus allowed even the "console" Basic to be a very rich language. We could "second guess" that idea, but there's nothing we can do about it now. Just remember that in the case of the TI, the program that's actually running when we're in Basic or Extended Basic is the GPL Interpreter.

In the PC compilers we've used, there are two possible ways to "link" the output program. The first is to use what's called a "RUNTIME LIBRARY" for execution. This limits the final product to a series of calls to another set of routines, which the computer loads into memory along with the .EXE program. (In this method, the RUNTIME library file must be kept on the same disk as the compiled program.) The second method for compiling is to create a standalone .EXE program, in which case the routines from a linking library are combined directly into the .EXE program itself, so no separate file is needed to run the resulting program. In either case, there is a huge amount of memory required, so that the routines that emulate similar ones from the Interpreter will be in memory when needed.

1.38.3. Why Impossible?

Taking an approach like that done on the PCs to making a compiler for the TI's Basics is clearly impossible, because the required machine code to emulate what the interpreter provides would make even short Basic "programs" become too large to fit in memory when compiled. The answer, if there ever is one, will be to make the compiler work in conjunction with what's already built into the computer. TI did its level best to make that "solution" itself impossible, by keeping all the inner workings of the Basic and GPL interpreters secret.

At this stage, (early September 1993) the fundamental framework of the compiler is looking like this: The "source" XB program will be saved in MERGE format. The compiler will load from Extended Basic. It will read the MERGE format file and produce three output files. The first will be a "shell" XB program in merge format. This shell will allow the compiled program to present itself to the computer as XB. The second output file will be an Assembly source file. The third output file will be an auxiliary data file used with the Assembly source file. A special loader provided by Harry Wilhelm will put together the mergeable "shell" and the assembled object file into what will look like an XB program. The compiler will be able to perform all of its steps without the need for an E/A cartridge.

We are making progress toward our goal, and have sent out some "samples" for testing to various people in the community. It's too early to say whether we'll succeed totally, but we have already made FOR-NEXT, PRINT, IF-THEN-ELSE, HCHAR, VCHAR, and CALL KEY into assembly routines, so that "demo" programs can perform as compiled programs under XB. Along the way, we find out some interesting tidbits of information. Just the other day, for example, we discovered a way to fool Extended Basic into thinking that we're running a line number from the original program. This is used with the ERROR function and with the BREAK to allow XB to report just the way it would if the original program were running. That way, if the user finds the compiled program stopping with an error, he'll know where in the source XB program to look for that error. We have, by the way, made the compiler so that **FCTN 4** will "break" the program, and so that CONtinue will work just as it does in XB programs, picking up just where it left off when **FCTN 4** was struck.

1.38.4. Real Source Code

Today's Sidebar shows a short XB program, then the source code generated from that file by the compiler. No, this hasn't been "fudged" in any way, but we've added annotation so you'll be able to understand what's happening. At some places in the source code, you'll see lines that say BL @TOGI. This means that the compiler will turn over the next operation to the GPL Interpreter to perform. What GPL is to perform is indicated by the DATA following the BL. In most cases that's an FLXX, where XX is the number of the FL label to be used. FL is just a mnemonic for Fake Line. This trick was not our invention, but was passed along by Harry Wilhelm. The fake line starts with the token for :: (130), then contains the tokenized form of the original program line, and ends with the tokens for :: GOTO 32767. Line 32767 of the mergeable "shell" program simply says CALL LINK("BACK"), and that returns control to the Assembly part of the compiled program right after the DATA for the TOGI line.

In the compiler's present state of development, there are many XB statements that the compiler can't handle, and when it fails to find the statement's opening token among its own routines, it branches to a section of code that generates this Fake Line and the BL and DATA for the main code section of the output source file. Thus the compiler can handle what's not included by letting XB handle it. For IF-THEN, it gets a bit more complicated, in that we first clear a flag word at TRUFLG, then BL @TOGI, but with an FI label (for Fake If) in the DATA line. The FI line starts with ::, and contains everything in the IF clause up to but not including the THEN token. The FI line continues with THEN CALL LINK("TRUE") ELSE 32767. (All in tokenized form, of course) This means that if what's in the IF clause is true, control will return to the Assembly code at label TRUE, and this will set TRUFLG to one, so that the Assembly code can determine what to do next.

If there's no ELSE, the compiler will generate an EL label at the appropriate place to handle what's to be done if the IF is not true. The compiler also checks what's after THEN, to see if there's a GOTO to skip past the ELSE part, and it inserts a GOTO if needed and as appropriate. (see B @L30 in the Sidebar)

The compiler generates labels with some idea of being consistent and mnemonic. For example, each new line of the source XB file creates a label that starts with L, followed by the line number. This way, one can very easily find the part of the Assembly source file that was derived from each line of the source file.

1.38.5. Variables

Handling the variables was an early challenge. Since we knew at the outset that some functions would forever be handled by the "Fake Line" process, it was imperative that our way of handling the variables be compatible with the way XB handles them. At the same time, we wanted some processes, like FOR-NEXT loop control, to be handled as integers only, so that speed could be improved. In essence, we wanted to "have our cake and eat it too". That's exactly what we've done in this compiler. All variables used in the original program are listed in the shell XB merge file, so that XB can perform its normal pre-scan to reserve space for them and build its symbol table in VDP RAM. In the Assembly source code, we make two tables for the variables. VARTBL includes all variables used in the original program, both string and numeric. Our own integer variables have their own table just after the table of XB's variables. Each entry in the Integer table has two words, where the first is reserved for the value of that variable, and the second is the cross-reference to the position of the corresponding variable in the main table.

When the compiled program is running, our Assembly part in the file STDOPN looks up all the variables' addresses in VDP RAM, and puts those into our VARTBL (where the 0,0 is in each variable entry) so that we can get any variable's current state from XB, and we can pass back values set in the integer variables when we need to. The compiler determines where in our Assembly part this needs doing, and BL's to either FPTIV or IVTFP as appropriate, with DATA indicating which integer variable is to be passed to or from.

1.38.6. Modularity

The Assembly support routines like FOR-NEXT, KEY, HVCHAR, and so on, are contained in separate Assembly source files, so they can be incorporated by COPY directives only if needed by the particular program. When we are at the point that we consider this product "finished", both the compiler itself and all its source code will be released to Public Domain, so that others may write additional support modules and integrate them as desired.

1.38.7. An Update

Last month we showed a way to use a CALL INIT within an Assembly program, and explained that this messed up our return to E/A. As in many such cases, there's a "brute force" solution to this problem. If one saves the entire contents of CPU RAM Pad (256 bytes starting at >8300) before the INIT call, and then restores that before ending the program, as well as setting the word at >2030 to >061C and saves and restores the GROM address, the return to E/A will work correctly. Yes, that's quite a lot of "bother", just to prove a point, but it's just another symptom of the "secrecy syndrome" which affects everything we try to do on this little machine.

1.38.8. Chapter 11

Back in part 22, we wrote about what we called "The Business End". There's an old joke in the business world that goes: "We lose 20 cents on each item we sell, but we make it up in volume." That's no joke when it happens in real life. Our company did actually make a small profit in 1992, but that profit was nothing compared to the hundreds of hours that went into the making of the products being offered for sale. A hard decision had to be made, then, as to what the "business" would do in the year 1993. The decision was that most of our commercial business would cease operation at the end of '93. The "profits" for '92, measured in percentage, were actually pretty good, coming in at somewhere around 20% of sales (before taxes), but when that "bottom line" in dollars is less than 1/4 of one month's Annuity from our Federal Retirement fund, it's simply not worth the endless hours spent in developing new products. If a product takes 200 hours to develop, then sells six copies at $10.00 "profit" each, that's an effective "salary" of only 30 cents per hour. Worse yet, that 30 cents per hour is taxed as income at about 40 percent between the IRS, the state, and the county, so the "after tax" yield is only 18 cents per hour. Our Maryland prison system pays higher hourly wages to those who make license plates. President Clinton has promised to lower this after-tax yield for us in 1993. It just wouldn't do for us to become rich from this business.

We will still offer some services, in the form of our custom program assistance for MIDI-Master users, and for those who need customized modules for use with other programs, but our standard product catalog is gone. We will continue programming, but what we develop will be aimed at the Public Domain or "Freeware" market. That is, it will be made available through user groups and such outlets for just whatever their copying fee amounts to.

This way, the users will get the benefit of our expertise, and we won't have to struggle with the question of "profits" that must be shared with the IRS, the Maryland State Comptroller, and the County of Prince Georges.

```
* SIDEBAR 38
* COMPILER INPUT AND OUTPUT
* (DOES NOT INCLUDE SUPPORTING FILES)
*
* FIRST, THE ORIGINAL XB PROGRAM
* (LISTED IN 28 COLUMNS)
*
10 FOR I=1 TO 30
20 IF I<10 THEN PRINT "I<10"
;I ELSE IF I<20 THEN PRINT "
I=>10";I ELSE PRINT "I>19";I
30 NEXT I
*
* THIS WAS SAVED WITH MERGE OPTION
* AS DSK4.IFTEMER
* COMPILER THEN PRODUCED THE FOLLOWING
*
*
* FIRST IS THE "SHELL" XB PROGRAM IFTE/M
* PRODUCED BY THE COMPILER (MERGE FORMAT)
```

```
*
1 CALL INIT
10 GOTO 100
11 I
100 CALL LINK("MAIN")
101 !@P-
32767 CALL LINK("BACK")
*
* SECOND OUTPUT IS THE ASSEMBLY SOURCE FILE
*
* ASSEMBLY SOURCE FILE
* HARRISON XB COMPILER
* DERIVED FROM:
* DSK4.IFTEMER
       COPY "DSK4.STDOPN" COPY IN THE "STANDARD OPEN" FILE
L10    BL   @SETCL       SET VALUE OF CURRENT XB LINE
       DATA 10           AT 10
       BL   @FORSET      SET UP A FOR-NEXT LOOP
       DATA 1            FROM 1
       DATA 30           THROUGH 30
       DATA 1            STEP 1
       DATA IV0          VARIABLE I (INT VARIABLE 0)
       DATA >0000        ALL PARAMETERS ARE JUST NUMBERS
LM0    DATA 0,0          RESERVE WORDS FOR LIMIT AND STEP
FR0
L20    BL   @SETCL       LABEL FR0 IS START OF LOOP
       DATA 20
       CLR  @TRUFLG      CLEAR "TRUTH FLAG" FOR IF-THEN
       BL   @IVTFP       TRANSFER INTEGER VARIABLE TO FLOATING POINT
       DATA IV0          USING VARIABLE IV0 (I)
       BL   @TOGI        USE GPL INTERPRETER
       DATA FI0          ON FAKE IF #0
       MOV  @TRUFLG,R4   MOVE THE TRUTH FLAG
       JEQ  EL0          IF IT'S ZERO, JUMP TO ELSE #0
       BL   @GETSC       GET A STRING CONSTANT
       DATA SC0          STRING CONSTANT #0
       BL   @PRNV        PRINT THAT
       DATA 180          WITH PENDING PRINT (;)
       BL   @PRNIV       PRINT INTEGER VARIABLE
       DATA IV0          NUMBER 0
       DATA 0            WITH NO PENDING PRINT
       B    @L30         GOTO LINE 30
EL0
       CLR  @TRUFLG      CLEAR TRUTH FLAG
       BL   @IVTFP       TRANSFER IV TO FP
       DATA IV0          INT VARIABLE #0
       BL   @TOGI        USE GPL INTERPRETER
       DATA FI1          FOR FAKE IF #1
       MOV  @TRUFLG,R4   CHECK THE TRUTH FLAG
       JEQ  EL1          IF NOT TRUE, GO TO ELSE #1
```

```
       BL   @GETSC       GET STRING CONSTANT
       DATA SC1          NUMBER 1
       BL   @PRNV        PRINT THAT
       DATA 180          WITH PENDING PRINT (;)
       BL   @PRNIV       PRINT INTEGER VARIABLE
       DATA IV0          NUMBER 0 (I)
       DATA 0            WITH NO PENDING PRINT
       B    @L30         GOTO LINE 30
EL1
       BL   @GETSC       GET STRING CONSTANT
       DATA SC2          NUMBER 2
       BL   @PRNV        PRINT THAT
       DATA 180          WITH PENDING PRINT (;)
       BL   @PRNIV       PRINT INTEGER VARIABLE
       DATA IV0          IV #0
       DATA 0            WITH NO PENDING PRINT
L30    BL   @SETCL       SET LINE INDICATOR
       DATA 30           LINE 30
       BL   @NXTIP       USE "NEXT" SUBROUTINE
       DATA LM0          FOR DATA AT LM#0
       DATA IV0          USING VARIABLE IV0
       B    @FR0         IF WITHIN LIMIT, BACK TO LABEL FR0
       BL   @TOGI        ELSE TO GPL INTERPRETER
       DATA FEND         WITH FAKE "END" LINE (ENDS PROGRAM)
       COPY "DSK4.STDSUB"  COPY IN STANDARD SUBROUTINES
       COPY "DSK4.FORNEXT" COPY IN FOR-NEXT ROUTINES
       COPY "DSK4.PRINT"   COPY IN PRINT ROUTINE
       COPY "DSK4.STDDAT"  COPY STANDARD DATA SECTION
       COPY "DSK4.IFTE/A"  COPY AUXILIARY DATA FILE
VARTBL BYTE 0,0,1        XB VARIABLES TABLE (0,0 BECOMES ADDRESS)
       TEXT 'I'
ENDTBL EQU $
* EACH VARIABLE IS LISTED WITH TWO BYTES RESERVED FOR
* ITS XB ADDRESS, THEN LENGTH OF ITS NAME, THEN THE NAME
       EVEN              ENSURE AN EVEN ADDRESS
IVTBL  EQU $
IV0    DATA 0,2          0 BECOMES "I", 2 IS CROSS-REFERENCE TO VARTBL
ENDIV  EQU $
       END
*
*
*
* LAST IS FILE IFTE/A, AUXILIARY DATA
* FIO AND FI1 ARE FAKE IF'S
* SC0, SC1, AND SC2 ARE STRING CONSTANTS
* CONTAINING "I<10", "I=>10" AND "I>19"
*
FI0    BYTE 130,132,73,191,200,2,49,48
* ABOVE IS TOKENIZED FOR :: IF I<10
       BYTE >B0,>9D,>C8,4,76,73,78,75
```

```
       BYTE >B7,>C7,4,84,82,85,69,>B6,>81,>C9,>7F,>FF
* ABOVE IS TOKENIZED FOR THEN CALL LINK("TRUE") ELSE 32767
SC0    BYTE 4,73,60,49,48
FI1    BYTE 130,132,73,191,200,2,50,48
       BYTE >B0,>9D,>C8,4,76,73,78,75
       BYTE >B7,>C7,4,84,82,85,69,>B6,>81,>C9,>7F,>FF
SC1    BYTE 5,73,61,62,49,48
SC2    BYTE 4,73,62,49,57
```

1.39. The Art Of Assembly — Part 39. More Mysteries Unraveled

By Bruce Harrison

In last month's column, we teased you a bit by mentioning that we'd gotten our Compiler to report errors, breakpoints, and other messages by the line numbers from the original XB program, but didn't say how that was done. Today we'll fill that gap for you, and also clear up some other "pending" mysteries we've mentioned in previous columns.

1.39.1. Fooling XB

In our work on the compiler, we became concerned about the business of error reporting, and the fact that if errors were reported, they would most often be reported as occurring in line 32767, because that's the line of the "shell" XB part of the compiled program that XB "thinks" it's executing. If we left matters like that, a user of our compiler would be left to guess where in his source XB program this error might be tracked down. Thus we set out to find some way of "fooling" XB into reporting the error as, for example "NEXT WITHOUT FOR IN 125", where 125 would be a line that doesn't exist in the compiled program, but that's where the original XB program would report this error.

We knew of course that XB has to "know" what line it's currently executing, so we got out some reference material to track down where that information gets stored. The address >832E was listed as "Pointer to current line number in line number table". That seemed a likely place to look, so we arranged a test by typing in a small XB program like this:

```
10 BREAK 20
20 GOTO 20
```

This program of course won't really do anything except run through line 10 and then stop with the report "BREAKPOINT IN 20". When that happened, we typed in CALL PG, to get into our P-Gram's program that allows us to examine memory. Sure enough, there was an address number in >832E that started with >FF, so this looked promising, as the line number table for such a short program would surely be in the >FF area. We then looked at that address, and found there another number in the >FF range. This was most certainly not the line number. It was the address of the line itself. Sure enough, if we looked two bytes back from the address in >832E, we were at the location in the line number table that contained >0014, (decimal 20). That's what we were after.

Now we suspected that XB might determine the line number for breakpoints and such by taking the address from >832E, subtracting two, and then taking the number from that address to report the line number on the screen. Thinking that such is the case and proving it are of course two very different things, but we set up an experiment in one of our test XB programs, (this one always ends with an error because it was designed to do just that) then patched up the compiler's code so that the following would happen:

1. The compiler would record the current line number as a DATA entry in the source code it creates.

2. The compiled program would place that data item at some convenient location, called CLNUM.

3. When an error was to be reported, the compiled program would load a register with CLNUM+2, then move that register to >832E before its BLWP @ERR.

In theory, this meant that the ERR routine would take that address from >832E, subtract two from it, then take the number from the resulting address and report that number on screen. The theory in this case was exactly right! The "snippets" in today's Sidebar show how this process worked for error reporting, for breakpoints, and for error or warning messages in "Fake Line" processing by the compiled program.

1.39.2. The ERR Report

Okay, here's another mystery we can clear up. On page 416 of the E/A manual, there's a long list of equates, some of which are left sort of unexplained. ERR EQU >2034 was one that baffled us for some time, especially since the next page contains a long list of possible error reports complete with addresses, but there didn't appear to be any way to connect the two things. By a mostly trial and error process, we have doped out just exactly how to use the numbers from that error message list and the ERR equate, so that when we're operating from an XB "environment", we can use XB to report the selected error message.

It's just this simple. Take that address from the table on page 417 for the message you want reported. (e.g. >1E00 for BAD VALUE) Write this into your source code:

```
LI   R0,>1E00
BLWP @>2034
```

That's it! When these two lines execute, XB will produce the "Boop" sound, and will report BAD VALUE IN XXX, where XXX will be the line number. The code that's used by the vector at >2034 takes whatever was in the caller's R0 and uses that to determine which message it will print.

1.39.3. Some Days It's Easy

Shortly after discovering the answers we've just discussed, we were fooling around with one of our compiled "test programs", in which we'd placed an INPUT statement to get a value for one of the variables used in the program. This INPUT would go into a numeric variable, which our program would then use as the limit value for a FOR-NEXT loop. In our compiler, INPUT is simply passed along to the GPL Interpreter to perform, since there's no speed advantage to be gained from replicating the INPUT process. Thus the source for the compiled program uses a BL @TOGI to execute a "Fake Line" (see last month's column) for the INPUT statement.

While running the compiled program, we decided to be very daring and put an illegal entry in at this input prompt. We typed **Q ENTER** at the prompt. What happened surprised and delighted us. We got the "boop", a report saying WARNING - INPUT ERROR IN 95, and the prompt on the screen just below that. The number 95 was indeed the line number from the original XB program, but what would happen next? We typed in **20 ENTER** to answer the prompt with a numeric response, and our compiled program went on about its business just as it normally would! Why did this work? What XB apparently did after issuing the warning was to go back to the start of our INPUT "fake line", and simply execute it again. When we answered the prompt with a legal input, XB continued the fake line, to wit :: GOTO 32767. That took us back into the Assembly code right where it should, and our program performed as expected. Flushed with success, we went back into our original XB test program, put in a line before # 95 that said "ON WARNING NEXT", then saved that in merge format for compiling.

When compiled, this too behaved exactly as it should. When the bad entry was made, no "boop" or warning message appeared, but the screen scrolled up and the input prompt was repeated. We made a correct entry, and the program continued from there. Incidentally, ACCEPT AT will behave in a similar manner, except that with the ON WARNING NEXT in effect, there will be no screen scrolling, and the input field will be cleared for a new input value.

1.39.4. And Then Other Days. . .

Actually, it was the very same day, in this case. Having proved that our faking of line number reporting worked for WARNING messages as well as for ERROR reports, we tried yet another little test, starting with the original XB program just discussed. At the input prompt, we entered **0**. The XB program then completely skipped the FOR-NEXT loop, and went on to what followed that. Unfortunately for us, that's NOT what happened in the compiled version.

Our Compiler's FOR-NEXT implementation was designed to handle almost every possible error, but in this case it went ahead and executed the FOR-NEXT loop once, then went on to the instruction after the NEXT. This happened because we were not checking the state of the index variable against the limit value until we got to the "NEXT" part of the loop. Once again it was "back to the drawing board" for our FOR-NEXT in the compiler.

That's just one small example of why things get messy when trying the "impossible". Just before this happened, we had run through a similar exercise on the PRINT function. Just when we thought the PRINT was working okay, we remembered that there's a function called TAB. (Expletive Deleted!) Two whole days were eaten up with that little problem. While we were about it, though, we also took care of the cases like this:

```
PRINT ,"Hello There"
```

This skips over halfway across the screen before printing Hello There, unless there was a comma at the end of the last PRINT statement, in which case the screen will scroll and Hello There will appear at the start of the next line. You get the message by now. Once one decides to implement an XB function, one must go in and check out all the possible variations, including PRINT with no argument, PRINT :"Something", and so on.

This can lead to some surprising results. For example, does anyone know what happens if you ask XB to do the following?

```
PRINT TAB(33);"Hello"
```

Jim Peterson probably knows, and Harry Wilhelm knows because I told him, but who else out there knows? (The Shadow Knows!) If the number in the TAB argument is greater than 28, XB will simply subtract 28 from it, and keep doing so until it's less than 28, then will execute a tab to this number position in the current line. For the above case, 33-28=5, so the H in Hello will be at the fifth position on the bottom line of the screen. TAB(61) would produce exactly the same result, since XB would subtract 28 twice to arrive at the number 5. Things like this can just drive a person crazy. Harry Wilhelm was of the opinion that if the TAB has a number greater than 28, it should cause the screen to scroll enough times to make up the numbers above 28. Instead, we put a simple loop into our PRNTAB routine so that it will exactly match the performance of XB's TAB.

1.39.5. But What If. . .?

One of the problems in this process of compiling is the very richness of the language. This means that for each new thing we try to implement as a compiled function, there are many potential variations on how the programmer might use that function, and we have to consider each and every one. In almost every situation where a number can be used in a program, for example, that can be either a simple number, a numeric variable, or numeric expression. We have further complicated things for ourselves by creating the concept of the integer variable, so there are four possible "things" that can supply the numbers for our functions.

This means that for functions like the setting up of a FOR-NEXT or CALL HCHAR, or even CALL KEY, the compiler has to determine which kind of thing each parameter is, and then has to put something into its source file so that the "runtime" routine will know this, and can properly handle each possible kind of parameter.

To let our compiled program know what's up, we use a data word in which each nybble is coded separately, thus maximizing our memory efficiency. For example, the routine that does both CALL HCHAR and CALL VCHAR has a maximum of four parameters. Each of these can be a simple number, an integer variable, a floating point (XB) variable, or a numeric expression. To tell our routine what type each parameter is, we use the four nybbles of one word. The right nybble contains 0 if the ROW is a number, 1 if it's an integer variable, 2 if it's a floating point variable, and 4 if it's a numeric expression. Each of the other three nybbles is coded in similar fashion to indicate the kind of thing that COLUMN, CHARACTER, and REPEAT parameters are. This same process is applied to the parameters for the FOR setup in FOR-NEXT, and for the parameters in CALL KEY.

1.39.6. Just One More Thing. . .

To quote Peter Falk as Columbo, just one more thing to clear up another of those little "mysteries". The Equates on page 416 of the E/A Manual include ones called VPUSH and VPOP. We thought these might mean pushing and popping numeric values to and from a stack in VDP RAM. We performed a little experiment using the source code shown in the Sidebar, and sure enough, VPUSH, executed through XMLLNK, took the floating point number from FAC and stashed it on a stack in VDP RAM. By clearing FAC, then doing a VPOP and re-assigning this value to an XB variable, we proved that this worked. But don't get too confident about such things. If you've pushed a variable in this way and then returned control to XB before coming back to POP the variable, the chances are pretty good that you'll have lost your value. This happens because XB makes its own use of the VDP value stack, so by the time you're back in control, the pointers will have been reset, and your original value on the stack will be gone.

Thus take a warning. You can use VPUSH to temporarily stash the value of a floating point number, and it can be recovered as long as your Assembly code is still "in charge". However, if you return control to XB, you should first pop out that floating point number and put it somewhere else, lest it get lost while XB is using this same stack.

We have a planned topic for next month which may be of some interest to the non-programming "public" among our readers. We have recently devised a way of taking an existing Option 3 object file and converting it to an Option 5 program file without having access to the original source code, and without doing any exotic maneuvers with Assembly. See you then!

```
* SIDEBAR 39
* FIRST IS A SMALL PIECE FROM
* SOURCE CODE CREATED BY THE
* COMPILER
*
* EACH LINE OF THE XB PROGRAM
* IS IDENTIFIED BY AN "L" PLUS
* THE LINE NUMBER BEING PROCESSED
* HERE IS SOURCE CODE CREATED BY
* LINES 20 THRU 30 OF A DEMO PROGRAM
*
L20     BL   @SETCL    USE SUBROUTINE SETCL
        DATA 20        TO SET CURRENT LINE NUMBER
        BL   @FORSET   SET UP A FOR-NEXT LOOP
        DATA 1         FROM 1
        DATA 5         TO 5
        DATA 1         STEP 1
        DATA IV0       VARIABLE IV0 (I IN XB)
        DATA >0000     ALL PARAMETERS JUST NUMBERS
LM0     DATA 0,0       STORAGE FOR LOOP COUNTING
        BL   @LIMCHK   CHECK LIMIT ON THIS LOOP
        DATA NX0       IF FINISHED, GOTO LABEL NX0
        BL   @IVTFP    CONVERT IV0 INTO FLOATING POINT I
        DATA IV0       (ALLOWS I TO BE USED BY FL #4, BELOW)
        BL   @TOGI     USE GPL INTERPRETER [READ A$(I)]
```

```
       DATA FL4       WITH FAKE LINE #4
       BL   @IVTFP    CONVERT AGAIN
       DATA IV0       IV0 TO F.P. VARIABLE I
       BL   @TOGI     USE GPL INTERPRETER
       DATA FL5       WITH FAKE LINE 5 [DISPLAY AT(I*2+5,6)...]
       BL   @INCLV    PERFORM "NEXT I"
       DATA LM0       WITH DATA AT LABEL LM0
NX0
       BL   @TOGI     USE GPL INTERPRETER
       DATA FL6       FOR FAKE LINE 6 [DISPLAY AT(24,6)...]
L25    BL   @SETCL    SET CURRENT LINE
       DATA 25        AT 25
       BL   @KEY      "CALL KEY" (SUBROUTINE KEY NOT SHOWN)
       DATA 0         0 (KEY-UNIT 0)
       DATA 18        K (KEY VARIABLE K)
       DATA 22        S (STATUS VARIABLE S)
       DATA >0220     K AND S ARE FLOATING POINT VARIABLES
       BL   @TOGI     USE GPL INTERPRETER
       DATA FI0       FOR FAKE IF #0 [IF S<1]
       MOV  R1,R1     CHECK R1 TRUTH INDICATION
       JNE  EL0       IF R1 NOT ZERO, STATEMENT FALSE
       B    @L25      IF TRUE, GOTO LINE 25 [THEN 25]
EL0
* IF STATEMENT WAS NOT TRUE, CODE FOLLOWING EL0 WILL EXECUTE
L30    BL   @SETCL       SET LINE NUMBER
       DATA 30           AT 30
       BL   @ONGTS       USE SUBROUTINE ONGTS
       DATA NE0          NUMERIC EXPRESSION #0 [K-48]
       DATA >8405        CONTROL WORD
       DATA LU0          LOOKUP TABLE #0
* CONTROL WORD DECODES AS FOLLOWS:
* 8 MEANS THIS IS ON-GOSUB, NOT ON-GOTO
* 4 MEANS A NUMERIC EXPRESSION (K-48) AS ARGUMENT
* 5 MEANS THERE ARE 5 BRANCH LINE NUMBERS
*
* LU0 IS THE LOOKUP TABLE AS FOLLOWS:
LU0    DATA L120         ADDRESS OF LINE 120
       DATA L200             "   "   "   200
       DATA L230             "   "   "   230
       DATA L300             "   "   "   300
       DATA L40              "   "   "   40
*
* THE ABOVE SECTION OF DEMO PROGRAM
* XB CODE LISTED IN 28 COLUMNS
* READS DATA TO PUT A MENU ON SCREEN
*
20 FOR I=1 TO 5 :: READ A$(I
):: DISPLAY AT(I*2+5,6):A$(I
):: NEXT I :: DISPLAY AT(24,
7)BEEP:"SELECT BY NUMBER"
```

```
25 CALL KEY(0,K,S):: IF S<1
THEN 25
30 ON K-48 GOSUB 120,200,230
,300,40
*
* NEXT PART IS FROM THE "RUNTIME" ROUTINES
* WHICH THE COMPILED PROGRAM USES
*
* SUBROUTINES USED IN COMPILED PROGRAM
* 24 SEP 93
* STORED AS STDSUB
*
TOGI   MOV *R11+,@>832C  PUT FAKE LINE ADDRESS AT >832C
       LI  R1,CLNUM+2    POINT AT CLNUM + 2
       MOV R1,@>832E     PLACE THAT ADDRESS AT >832E
       LWPI >83E0        LOAD THE GPL WORKSPACE
       B @>006A          BRANCH TO GPL INTERPRETER
*
* LABEL TRUE IS USED WITH AN IF-THEN
* SO THAT IF THE "IF" IS TRUE, OUR REGISTER 1
* WILL BE CLEARED.  OTHERWISE, THE "IF" WILL
* RETURN TO US AT LABEL BACK, AND R1 WILL STILL
* CONTAIN THE ADDRESS WE LOADED INTO IT IN TOGI
*
TRUE   CLR  @WS+2        CLEAR OUR REGISTER 1 IF STATEMENT WAS TRUE
BACK   LWPI WS           RETURN FROM A FAKE LINE TO OUR WORKSPACE
       RT                THEN BACK TO ASSEMBLY CODE
SUBRET DECT R15          SUBTRACT 2 FROM OUR STACK POINTER
       MOV  *R15,R11     MOVE WORD INTO OUR R11
RETURN LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN STOP THEM
       LWPI >83E0        LOAD THE GPL WORKSPACE
       BL   @>20         CHECK FOR FCTN-4 KEYPRESS
       JNE  RET0         IF NOT, JUMP AHEAD
       LI   R6,CLNUM+2   ELSE SET FOR BREAKPOINT
       MOV  R6,@>832E    WITH ADDRESS CLNUM+2
       LI   R6,CONLIN    SET FOR A "CONTINUE"
       MOV  R6,@>832C    WITH GPL INTERPRETER
       B    @>6A         THEN BRANCH TO GPL INTERPRETER TO BREAK
RET0   LWPI WS           LOAD OUR OWN WORKSPACE
       RT                THEN RESUME AT ADDRESS IN R11
SETCL  MOV  *R11+,@CLNUM MOVE THE DATA AFTER THE BL INTO CLNUM
       RT                THEN RETURN
GENERR LI   R1,CLNUM+2   GET LINE NUMBER POINTER
       MOV  R1,@>832E    INTO >832E FOR ERROR REPORT LINE
       BLWP @ERR         (ERR EQUATED TO >2034, NOT SHOWN)


*  ONGTS ON-GOTO / ON-GOSUB
*  HARRISON COMPILER
```

```
*  25 SEP 1993
*  STORED AS ONGTS
*
*
ONGTS  MOV  *R11+,R2     FIRST DATA INTO R2 (ARGUMENT ADDRESS)
       MOV  *R11+,R13    SECOND INTO R13    (CONTROL WORD)
       MOV  *R11+,R3     THIRD INTO R3      (LOOKUP TABLE ADDRESS)
       MOV  R11,R14      STASH R11 RETURN ADDRESS IN R14
       MOV  R13,R12      PUT R13 INTO R12
       SRL  R12,8        SHIFT RIGHT 8 BITS
       ANDI R12,7        MASK ALL BUT LOWEST THREE BITS
       JEQ  ONGTE1       IF ZERO, AN ERROR CONDITION
       CI   R12,1        COMPARE TO 1
       JNE  DECCV2       IF NOT ONE, JUMP AHEAD
       MOV  *R2,R2       ELSE GET INTEGER VARIABLE VALUE
       JMP  SETCV        THEN JUMP
DECCV2 CI   R12,2        COMPARE TO 2
       JNE  DECCV4       IF NOT, JUMP AHEAD
       AI   R2,VARTBL-2  SET UP FOR F.P. VARIABLE
       MOVB *R2+,R9      FIRST BYTE OF F.P. ADDRESS
       SWPB R9           SWAP
       MOVB *R2+,R9      SECOND BYTE OF F.P. ADDRESS
       SWPB R9           SWAP AGAIN
       BL   @MOVFAC      USE A SUBROUTINE (NOT SHOWN)
       MOV  @FAC,R2      GET INTEGER VALUE INTO R2
       JMP  SETCV        THEN JUMP
DECCV4 MOV  R2,@FAC      STASH ADDRESS OF NUMERIC EXPRESSION AT FAC
       BL   @INTERP      INTERPRET EXPRESSION
       MOV  @FAC,R2      MOVE INTEGER IN TO R2
SETCV  ABS  R2           TAKE ABSOLUTE VALUE
       MOV  R13,R12      MOVE R13 TO R12
       ANDI R12,>00FF    USE ONLY LSB
       C    R2,R12       COMPARE
       JGT  ONGTE2       IF R2 > R12, ERROR (BAD VALUE)
       DEC  R2           DECREMENT R2
       JLT  ONGTE2       IF <0 THEN BAD VALUE
       SLA  R2,1         DOUBLE R2
       A    R3,R2        ADD LOOKUP TABLE ADDRESS
       MOV  *R2,R5       GET THE GOTO-GOSUB ADDRESS IN R5
       MOV  R13,R12      MOVE R13 AGAIN
       JGT  ONGSX        IF POSITIVE, THIS IS ON-GOTO
       MOV  R14,*R15+    ELSE IT'S ON-GOSUB, SO STACK RETURN ADDRESS
ONGSX  B    *R5          BRANCH TO THE ADDRESS IN R5
ONGTE1 LI   R0,>0300     ERROR REPORT WILL BE "SYNTAX ERROR"
       JMP  ONGTEX       JUMP AHEAD
ONGTE2 LI   R0,>1E00     ERROR REPORT WILL BE "BAD VALUE"
ONGTEX B    @GENERR      BRANCH TO ERROR REPORTING CODE
*
* END OF COMPILER SUBROUTINE FRAGMENTS
*
```

```
* FOLLOWING IS SHORT XB PROGRAM
* USED TO TEST VPUSH/VPOP
* THROUGH LINKAGE TO ASSEMBLY
*
1 ! TEST OF VPUSH/VPOP
2 ! USE WITH PUPOP/O
3 ! EXTENDED BASIC ONLY
10 CALL INIT
20 CALL LOAD("DSK1.PUPOP/O")
30 A=48.254 :: PRINT A
40 CALL LINK("TEST",A)
50 PRINT A
*
* FOLLOWING IS SOURCE FILE PUPOP/S
* TO USE, ASSEMBLE INTO PUPOP/O,
* THEN RUN WITH ABOVE XB PROGRAM
*
* PUSH/POP TEST
* DSK1.PUPOP/S
* WITH XB
* ENTRY LABEL TEST
*
* REQUIRED EQUATES
*
NUMREF EQU  >200C        NUMERIC REFERENCE
NUMASG EQU  >2008        NUMERIC ASSIGNMENT
XMLLNK EQU  >2018        XML LINK VECTOR
VPUSH  EQU  >000E        VDP PUSH
VPOP   EQU  >0010        VDP POP
FAC    EQU  >834A        F.P. ACCUMULATOR
*
* CODE SECTION
*
       DEF  TEST         DEFINE ENTRY POINT
TEST   LWPI WS           LOAD OUR WORKSPACE
       BLWP @NUMREF      GET PARAMETER (A FROM XB)
       BLWP @XMLLNK      USE XML
       DATA VPUSH        TO PUSH FROM FAC ONTO STACK
       CLR  @FAC         CLEAR FAC
       BLWP @XMLLNK      USE XML VECTOR
       DATA VPOP         TO POP VALUE A FROM STACK
       BLWP @NUMASG      RE-ASSIGN TO VARIABLE
       LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         RETURN TO GPL INTERPRETER
*
* DATA SECTION
*
WS     DATA 0,1          PRELOADED REGISTERS 0 AND 1
       BSS  28           REST OF WORKSPACE
       END
```

1.40. The Art Of Assembly — Part 40. It Pays To Be . . .

By Bruce Harrison

Back in your author's early childhood, there was no such thing as television. Network shows existed on the radio, and these provided much of the entertainment in our house. There were some shows with truly ridiculous ideas behind them. (Television didn't change that!) As we were getting ready to write this, an old radio show came to mind. It was titled "It Pays To Be Ignorant". As we recall, there was a theme song that went "It Pays to be Ignorant, to be Dumb, to be Stupid, to be Ignorant. . .". It was a quiz show on which the contestant who gave the dumbest answer to the question was the winner.

That came to mind because the main topic for today's column came from a fellow who asked me what most TI programmers would call a dumb question. It happened one Sunday evening, while I was pecking away at my TI, trying to solve some knotty problem in the Compiler. The phone rang. The caller was a gentleman from Massachusetts, and he had several questions, all wrapped around the idea of taking an existing Option 3 Assembly program and converting it to Option 5. He was trying to see if our Part 14 (Crossing the Bridge. . .) would be useful, but not having any luck understanding the process. We explained that to use what's in Part 14, one needed to start with the source code for the program in question. Since he had only the object file, we told him that conversion would be impossible. Anybody who knows anything about TI Assembly knows that!

We sent the gentleman a disk with some things to solve a couple other problems he'd encountered, but his question about conversion without the source code kept nagging at us. The mind keeps working on something like this, even without much conscious thought. We kept hearing "What if we. . . No! That won't work because. . .". Maybe fifteen or twenty ideas came and went in this manner, but then finally there was an idea that just might work.

1.40.1. The Bright Idea

If we took our "sandwich" from Part 14, and separated it into two source files, inserted REFs and DEFs in various places, then assembled the two parts of the sandwich into separate object files, we could do this impossible task. We would use the "linking" feature of the E/A Object loader (as we seldom do) to give these modules information about each other and the original object file, and the "can't be done" would become a "here's how".

Several conditions must be met for this method to work. First, the object file has to be in relocatable code. Many are, since in most cases for an Option 3 file there's no need to AORG the program. The other requirement is that one must know the entry point label for the program. That seemed a likely thing, since one can't run an Option 3 from E/A without knowing the entry point. (Except for Auto-Start) We decided to try a little experiment, using a handy Option 3 file called DEBUG, which of course was supplied with the E/A package. As shown in the Sidebar, we split the sandwich into a part called SFIRST/S, which includes a ref to the label DEBUG, and SLAST/S, which didn't need any tailoring except to have a couple of DEFs added.

Here's how it worked. We Assembled SFIRST/S into SFIRST/O, and SLAST/S into SLAST/O. Now we went into E/A, selected Option 3, and started loading object files. First, we loaded DSK4.SFIRST/O. Next, DSK5.DEBUG, then DSK4.SLAST/O, and finally DSK5.SAVE. (This last is the TI SAVE utility.) Now we got to the PROGRAM NAME prompt, and typed in the word SAVIT. This meant we'd run the part of the SLAST file that captures the E/A utilities and places them within the saved Option 5 file, then branches to TI's SAVE to actually create the Option 5 program files. We answered the SAVE prompt with DSK5.DEBUGOP5, and sure enough that worked. Before the mail comes in, we'll admit openly that making an Option 5 out of the debugger doesn't, in itself, make any sense. The idea was to prove a concept, not to do something useful. DEBUGOP5 worked perfectly, whether loaded from E/A, or from our Ramdisk menu loader. Q.E.D.

1.40.2. Step By Step

Okay, let's say you have an Option 3 file, and that you know the entry point, and you know that it's relocatable. (We'll tell you how to find out in a couple of paragraphs.) Follow these steps in order:

1. Assemble SLAST/S into SLAST/O.

2. Edit SFIRST/S. Replace the two occurrences of DEBUG with the entry point name for your Option 3 program. Save that to disk and Assemble it into SFIRST/O.

3. Find a disk that has TI's SAVE utility on it. Copy that onto the same disk with SFIRST/O, SLAST/O, and the original Option 3 object file. (From here on, we'll assume that disk is in drive 1, but of course any drive will do.)

4. Get into E/A Option 3. At the FILE NAME prompt, type in DSK1.SFIRST/O **ENTER**.

5. When that finishes loading, type in the name of the original object file, then **ENTER**.

6. When that finishes loading, type in DSK1.SLAST/O **ENTER**, and follow that loading with DSK1.SAVE **ENTER**.

7. After DSK1.SAVE has loaded, leave the entry field blank and press **ENTER**. Answer the PROGRAM NAME prompt with SAVIT **ENTER**.

8. You'll now have a prompt from TI's SAVE utility on the screen. Type in the name you'll use for your Option 5 program file, (e.g. DSK1.MYOPT5). Press **ENTER**. Drive 1's activity light will come on while the Option 5 file(s) are being saved.

9. Now you'll have on disk a program file or a series of them, depending on length. This should run from any method you choose to use for loading Option 5 programs.

We promised to tell you how to tell whether an object file is Absolute or Relocatable. The answer depends on whether the object file is in compressed or uncompressed format, but you can tell in either case. Get into E/A Option 1, (EDIT), and load the object file for editing. Several things can happen here that may alarm you, but just bear with us and proceed as instructed. You may get a MEMORY FULL warning when loading the object file. Just ignore that, press **ENTER**, and then **2** to edit. For compressed object files, you'll get a CONTROL CHARACTER REMOVED warning. Ignore that too.

In edit, you'll just look first at the first record in the file (top line of the screen). The first thing in that line, for an uncompressed file, will be a number in hex notation. If that number is 00000, this file is absolute, and can't be converted by our method. Anything other than 00000 means the conversion will probably work. For compressed files, there won't be a readable number there in the first line. Instead, look down the leftmost column on the screen, where the tags are. If there's a series of Bs and/or 9s down the left column, then this is absolute code and conversion won't work. If the left column contains As and/or Cs, then this is relocatable code, and conversion will probably work.

In any case, after you've examined the object file with the editor, just get out of there, preferably with **FCTN =**, so you won't be tempted to save this file back to disk. DON'T SAVE after this "edit"! If your answer was "yes" from this check, then use the step-by-step procedure above to do your conversion. There may be cases where this method won't succeed, as for example if the Option 3 file was close to filling the memory without adding our "sandwich". Should you need help, please give us a call at (301) 277-3467, any time from 9AM through midnight Eastern time, seven days a week. This column is living proof that NO QUESTION IS TOO DUMB! We may not immediately have the solution, but don't get discouraged, we'll keep trying.

1.40.3. Another What If. . .

There's another potential hazard to this process for converting Option 3 to Option 5, and that's the case where the original Option 3 was set up with an auto-start label. If that's the case, the Option 3 file will start executing as soon as it loads from E/A, and you'll never get the chance to load either the SLAST/O or SAVE files. Yes, this too can be overcome, thanks to the genius of Tony McGovern. Funnelweb has an Option 3 loader which will simply ignore the auto-start, and allow you to continue with loading until you're ready to run something. Select LOADERS from Funnelweb's menu, then the L/R AUTO OFF choice from the menu that appears. The only trick required here is that, after loading the SAVE utility, you'll have to press **FCTN 3** to clear the entry field. Now press **ENTER**, and use the left-right arrow keys to select SAVIT from the list of DEFs then on the screen. Press **FCTN 6** with SAVIT selected, and you're in business. Here's another thanks to Tony McGovern, for a great feature in Funnelweb.

This loader that's provided by Funnelweb can also be used to find the entry point for a "mystery" Option 3 file. Simply select the LOAD/RUN or the L/R AUTO OFF choice, then enter the object file's name and let it load. When it finishes, press **FCTN 3** to delete all of the file name from the entry field, and press **ENTER**. Any and all defined labels in that file will be shown on the screen. If there's only one, then that's the entry label you'd use at two places in SFIRST/S. If there are two or more, apply some educated guess logic to figure out which is the main entry, or else just select one of them and run the program from Funnelweb. If it seems to work as you'd expect, then you've guessed correctly. If not, try this process again and make a different selection.

1.40.4. Serendipity Results

As so often happens in this business, we find out purely by accident that certain things on our TI behave in ways we didn't expect. That was the case with XB's PRINT TAB function, as we explained last month. Now our work on the Compiler has yielded another unexpected pleasant surprise. In the compiler's own subroutines, we used XMLLNK to invoke the Convert Floating Point to Integer (CFI) routine in the console. In the way we are using it, it does exactly what we needed, taking an eight-byte floating point number at FAC and converting that to a two-byte integer at FAC. This routine produces weird results for numbers outside the range of -32768 through 32767, but that was of no concern to us, since such numbers were not involved in the situation we were working with.

In one of our DEMO XB programs for the compiler, we used an INPUT statement so the user could decide how many times to execute a FOR-NEXT loop. INPUT places that number into a floating point variable, as you'd expect. In the process of using that number to control the FOR-NEXT loop, the compiled program takes that floating point number into FAC and uses CFI through XMLLNK to convert it to an integer as the loop's limit value. Just to see what would happen, we typed in a number that included a decimal part, as in 10.25. The program just ignored the decimal, and ran ten repeats of the FOR-NEXT loop. Seeing that this worked, we tried entering 10.5, and were surprised to see the loop execute 11 times, not 10. Other quick checks followed, and in all cases if the decimal part was .499999 or less, the CFI routine rounded it off to the lower integer value, while if the decimal part was .5 or greater, the CFI routine correctly rounded to the next integer.

In today's Sidebar is a little routine you can use from Extended Basic to see how this works, plus a short XB program to allow testing of this "rounding" process. In this case, we've re-converted the integer to floating point so that we could use NUMASG to report the result back into XB and print it. We have run tests, and the only anomaly we found was that for negative numbers, the rounding doesn't go to the next integer until the decimal part is >.5. Thus -10.5000 will round to -10, while -10.50000001 will round to -11.

Finally today, we apologize if we seemed to poke fun at our friend in Massachusetts, whose phone call inspired the first part of our column today. Like many of our readers, and like your author, he was struggling with trying to make his TI do something new, and had gotten frustrated enough to call for help. Many columns ago, we promised not to scold our readers by name, and we've kept that promise. Our friend in Massachusetts will no doubt recognize his case as we've described it. We hope he'll be happy that thanks to his having the courage to ask his question, an answer has been created that may help dozens of other users in our community.

Next month's topic is undecided. Perhaps we'll get another phone call that will lead to yet another discovery in our quest to know everything there is to know about this wonderful but strange machine. The only thing sure is that we will write a column for next month's issue, as this "old windbag" always has something to say.

```
* SIDEBAR 40
* FIRST PART, TWO SOURCE FILES
* FOR USE IN "CONVERTING" FROM
* OPTION 3 TO OPTION 5 WITHOUT
* HAVING ORIGINAL PROGRAM'S
* SOURCE CODE.


* SFIRST/S  FIRST SOURCE FILE
* IN THE TWO PLACES WHERE DEBUG APPEARS,
* IT SHOULD BE REPLACED BY THE ENTRY
* LABEL FROM THE OPTION 3 PROGRAM
* (SEE TEXT FOR DETAILS)
*
       DEF  SFIRST,SLOAD DEFINITIONS REQUIRED BY TI SAVE UTILITY
       REF  DEBUG,EAUT,WST REFS TO OPTION 3 AND SLAST/O
SFIRST
SLOAD
       LWPI WST          LOAD TEMPORARY WORKSPACE
       LI   R9,EAUT      POINT R9 AT STORED UTILITIES
       LI   R10,>2000    POINT R10 AT >2094 IN LOW MEMORY
       LI   R4,>2676->2000  LOAD R4 WITH NUMBER OF BYTES TO MOVE
PUTUT  MOV  *R9+,*R10+   MOVE ONE WORD, INCREMENT POINTERS BY TWO
       DECT R4           DECREMENT COUNT BY TWO
       JNE  PUTUT        IF NOT ZERO, REPEAT OPERATION
       B    @DEBUG       BRANCH TO OPTION 3'S ENTRY POINT
       END
*
* END OF SFIRST/S
*
* SLAST/S SECOND SOURCE FILE
* NO CHANGES REQUIRED
* FOR USE WITH ANY OPTION 3
*
       DEF  SLAST,EAUT,WST,SAVIT
       EVEN              INSURE THAT EAUT IS AT AN EVEN MEMORY LOCATION
EAUT   BSS >2676->2000   LENGTH OF BSS IS >676 BYTES
       REF  SAVE         REFERENCE TI'S SAVE UTILITY
SLAST
* SLAST MARKS THE END OF WHAT THE SAVE UTILITY WILL PUT IN MEM-IM FILE
SAVIT
       MOV  R11,@>8300   STASH R11
       LWPI WST          LOAD OUR TEMPORARY WORKSAPCE
       LI   R9,>2000     POINT R9 AT BEGINNING OF AREA TO BE SAVED
       LI   R10,EAUT     POINT R10 AT MEMORY LOCATION ABOVE
```

```
       LI   R4,>2676->2000  LOAD R4 WITH NUMBER OF BYTES TO MOVE
GETLP  MOV  *R9+,*R10+   MOVE ONE WORD AND INCREMENT POINTERS BY TWO
       DECT R4           DECREMENT COUNT BY TWO
       JNE  GETLP        IF NOT ZERO, REPEAT AT LABEL GETLP
       B    @SAVE        BRANCH DIRECTLY TO TI'S SAVE UTILITY
WST    BSS  32           OUR TEMPORARY WORKSPACE
       END
*
* END OF SLAST/S


* NEXT PART IS THE EXPERIMENT IN
* ROUNDING NUMBERS USING XMLLNK
* THIS MUST BE USED WITH THE XB
* PROGRAM SHOWN BELOW FOR TESTING



* TEST/S - SOURCE CODE FOR
* DEMONSTRATION OF ROUNDING
* THROUGH USE OF XMLLNK
* ENTRY LABEL TEST
*
* REQUIRED EQUATES
*
NUMREF EQU  >200C        NUMERIC REFERENCE
NUMASG EQU  >2008        NUMERIC ASSIGNMENT
XMLLNK EQU  >2018        XML LINKAGE VECTOR
CIF    EQU  >20          CONVERT INT TO F.P.
CFI    EQU  >12B8        CONVERT F.P. TO INT
FAC    EQU  >834A        F.P. ACCUMULATOR
*
* CODE SECTION
*
       DEF  TEST         DEFINE ENTRY LABEL
TEST   LWPI WS           LOAD OUR WORKSPACE
       CLR  R0           CLEAR R0, NOT ARRAY
       LI   R1,1         FIRST PARAMETER
       BLWP @NUMREF      GET PARAMETER VALUE
       BLWP @XMLLNK      USE XML LINKAGE
       DATA CFI          CONVERT TO INTEGER (ROUNDS)
       BLWP @XMLLNK      USE XML AGAIN
       DATA CIF          CONVERT BACK TO F.P.
       BLWP @NUMASG      RE-ASSIGN TO XB VARIABLE
       LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         BRANCH TO GPL INTERPRETER
*
* DATA SECTION
*
WS     BSS  32           OUR OWN WORKSPACE
```

```
      END
*
* END OF SOURCE FILE TEST/S
*
* BELOW IS LISTING OF THE XB
* PROGRAM TO BE USED WITH
* THE OBJECT FILE TEST/O, MADE
* FROM THE SOURCE ABOVE
*
1 ! TEST PROGRAM - ROUNDING
2 ! USING ASSEMBLED FILE TEST/O
3 ! PUBLIC DOMAIN
4 ! BY BRUCE HARRISON
5 ! EXTENDED BASIC ONLY!
10 CALL INIT
20 CALL LOAD("DSK1.TEST/O")
30 INPUT "A NUMBER ":A
40 CALL LINK("TEST",A)
50 PRINT A:"ANOTHER? (Y/N)"
60 CALL KEY(0,K,S)
70 IF S<1 THEN 60
80 IF K=89 OR K=121 THEN 30
```

1.41. The Art Of Assembly — Part 41. It's About Time

By Bruce Harrison

Today's subject is time. Recently, it's been important to us in several respects. In the first place, it's February 1994 as we write this, and we've just learned of the death of our dear friend Jim Peterson. This reminds us that we're all working against an unknown time limit, but that's not the kind of time constraint we're concerned with in today's column. It's the use of our beloved TI to do things that are related to "keeping time" in matters of seconds. We'll talk about two time-related assembly projects, and show how these use the internal Vertical Interval timer in the TI to provide an accurate source of "ticks" that we can count.

1.41.1. The Metronome

As many of our readers are aware, we are a "musical" family, having produced lots of "computer music" both with and without MIDI capability. Our two little boys have started learning to use "real" musical instruments, so we're getting used to having the sounds of a clarinet and violin in the house. When kids are learning to play instruments, they need a stable source of timing to keep them playing at a steady tempo. Our local music store has all kinds of metronome devices, from the simple mechanical ones to the most exotic electronic ones. They all work quite well, but all are expensive, considering that all they do is produce a repetitive "tick" at a constant rate in beats per minute.

Being a typical "TI Cheapskate", it occurred to me that our beloved computer ought to be able, with the right programming, to produce those ticks for "free". We plunged right into Assembly, and in just a couple of days we had a new Public Domain "product" ready for any and all TI users. This product, which is called METRONOME, is available through the Lima and Chicago User Groups at their normal copying fee. It's also been authorized to be placed on BBS, etc. Thus it should be readily available to anyone who needs it. Actually the disk contains two versions of the program, one for U.S. users and another for "European" users. This is necessary because the actual timing is provided by the Vertical Interval timer, and that operates at 60 Hz for U.S. systems, and at 50 Hz for "European" systems, including those in Great Britain and Australia.

We've not included the source code for the metronome in today's column, partly because it's a lot of stuff, and partly because it's already provided on the Public Domain disks. We will, however, cover some salient points.

First on our list of "musts" was that the user interface had to be extremely simple but effective. The input for the user had to be expressed in Beats per Minute, since that's what is used in the musical world. (Sheet music often indicates tempo by metronome setting, and the number given is always understood to be in Beats per Minute.) Of course nothing in the computer itself understands Beats per Minute, so we have to provide a kind of "translation" from B/M to a number expressed in 60ths (or 50ths) of a second. We also wanted all this to be done as accurately as possible, so we used Floating Point numbers to do the calculations in this mathematical "translation" process. In the following, we're describing the U.S. version, then we'll briefly cover what's different in the "European" version.

The user gets a prompt "BEATS PER MINUTE" on the screen, with an allowed range indicated (15 through 500). He enters a number in that range, let's say 120, for example. As soon as he presses **ENTER**, we use an internal service called Convert String to Number to take the number right from the screen and convert it to a floating point number stored at FAC (>834A). What we're after is a number in 60ths of a second that will provide a limit count for the Vertical Interval timer at >8378. To get there, we take the number of 60ths of a second in one minute (3600) expressed as a floating point number, and place that eight-byte quantity at location ARG (>835C) in memory. Now we use another built in service called Floating Point Divide to divide 3600 by 120. That gives us the number 30, or exactly half the number of 60ths in one second.

This is a number that can be used as the limit on our time count. We use the built-in Convert Floating Point to Integer routine, (This conversion rounds the number correctly to the nearest integer.) place that number into memory as an integer value, and start making our "ticks". First, we clear the counter at >8378, then send a few bytes to the sound generator to make a sound. We enter a loop that checks constantly the number in >8378 against the limit number (30 in this case). The sound we've sent is silenced after the first three counts, so it's just a brief "tick", after which the computer gives silence for the rest of the 30 60ths period. When that count finishes, the computer re-clears >8378, sends another tick to the sound generator, and starts counting the next period.

This gives exactly the desired result, with two ticks per second, or 120 per minute, being produced by the monitor's speaker. By using the Vertical Interval as our source of timing, we've made this able to be accurate regardless of whether it's run on a standard TI, or a "bus modified" TI, or a Geneve at any of its clock rates.

1.41.2. Range Limits

The timer we're using (at >8378) actually uses only one byte, at >8379. Thus the number we can count has an upper limit of 255 counts before it "zeros" itself. Thus there's a lower limit of 15 beats per minute, as that calculates to 240 counts. (14 would wind up with a count of 257, and that's too big.)

The other limit, 500, is arbitrary, but necessary so that there will be some "off time" between ticks even at the fastest rate. With 500 entered, the number of counts for the interval would be 7.2 (rounds to 7) so that the sound would be on for three 60ths, then off for four 60ths, and so on. The program checks the value of the count before starting the ticks, and rejects the entry if the count is less than seven or more than 240. The user will see the entry field clear, ready to accept a new input. The accepted range (15 through 500) should be enough for nearly any musical purpose, covering as it does from an extremely slow dirge through a super prestissimo.

While this ticking is going on, we flash the cursor onto the screen during each tick, so that there's a visual cue in case the musician's playing drowns out the sound of the tick. We also check for a keyboard input during the ticking, so that the user can stop the ticking by simply pressing any key. Pressing a key other than **FCTN 9** during a stopped condition will just re-start at the same rate. Pressing **FCTN 9** while the ticks are stopped will clear the entry field so a new rate can be entered.

1.41.3. The "EUR" Version

For our friends in Europe and Australia, the "EUR" version, called METROEUR, is designed to operate with their PAL video system, which has a 50 Hz vertical rate. The essential difference is that, for the 50 Hz system, there are 3000 vertical intervals per minute instead of the 3600 on the U.S. system. Thus where, in the preceding discussion, we put 3600 as a floating point number into ARG, the "EUR" version places 3000 at ARG, then divides that by the user's input number. Because of this different number, the "EUR" version allows a lower bottom limit of 12 beats per minute, but that's the only obvious difference, except that the "EUR" version has a screen legend saying "EUROPEAN VERSION". This Public Domain product is available from the Lima Users' Group as disk 870A. (Contact the group c/o Dr. Charles Good at P.O. Box 647, Venedocia, OH 45894.)

1.41.4. A Timed Input Field

Ever since we started programming on the TI (in Extended Basic) we've wished for some way to limit the time allowed for an INPUT or ACCEPT AT statement. As we've mentioned before, our friend Jim Peterson sometimes issued challenges to the assembly practitioners. Both he and Barry Traver have said that there were times when they'd like to be able to place a "time limit" on a user input, as in some quiz games they've written in Extended Basic. As you all know, once Extended Basic starts executing an INPUT or ACCEPT AT statement, it just waits until the user is finished with his entry, even if an hour goes by. This can take some of the fun out of a game program, since the user could go to the library to look up the answer, and the computer would wait till he or she got back to answer the question.

As we were fooling around with this timing for the metronome application, this problem kept haunting our thoughts. Perhaps we could keep a time count through a "user interrupt", and then find some way to terminate the INPUT or ACCEPT command after a number of vertical intervals, without the user having typed anything from the keyboard.

Before we go further, let's confess that the "user interrupt" process is one that we rarely use, and don't really understand. Thus getting the interrupt to do what we wanted was difficult. As you'll see in the Sidebar, the "final product" is fairly simple. There are two "entry points" used with CALL LINK from Extended Basic, plus the Interrupt code itself. Here's how it works.

Assuming the code shown in the Sidebar has been assembled and loaded under Extended Basic, the program that's using the code would perform a CALL LINK("SETTIM",TL), where TL is a number in seconds. Anything from 1 second through 546 seconds can be used for TL. Decimal quantities can be used, as for example 4.5 seconds. The limit 546 (about 9 minutes) is an absolute upper limit, beyond which the time counter simply won't work. (To see why, whip out your trusty calculator and multiply 546 by 60.)

The CALL LINK to SETTIM can be done at any place in the XB program. It sets the limit number in 60ths of a second, but doesn't start the count operating. To apply the time limit, insert a statement just before the input statement like this:

```
CALL LINK("ACT") :: INPUT "ANY STRING ":X$
```

This will "activate" the pre-set time limit for that input action. While the computer is waiting for input, it will be cycling through interrupts, one of which will increment the counter in our user interrupt routine every 60th of a second. (SETTIM multiplies the number given in seconds by 60.) When the count equals or exceeds the limit, the interrupt will begin executing the code at label TIMEUP. This code puts the ASCII value for the **ENTER** key into the key value address (>8375), and puts the value >20 into the GPL Status byte (>837C). The INPUT routine sees this condition and "assumes" the user has pressed **ENTER** on the keyboard, so it exits from the INPUT statement. When we first started developing this routine, it worked perfectly with INPUT, but not with ACCEPT AT. ACCEPT AT would exit with an error. We discussed the matter with Harry Wilhelm, and he quickly figured out that the problem had to do with whether an actual key scan had just taken place before we tried forcing the **ENTER** and >20 into place. As he has so many times, Harry came to our rescue, with a modification to our routine. Harry figured out how to check the GROM address to determine whether the TI has just done a key scan. Thus the forced **ENTER** can be put in at exactly the right time, so no error occurs upon return from ACCEPT AT. With this modification, the timeout can be used with any kind of input routine, including CALL KEY. With this necessity to check the GROM address, the timing is not as accurate, as the Interrupt may have to wait several 60ths of a second before it hits the correct "window" to terminate the input cycle. Harry recommended that we discard the floating point calculations, since the accuracy won't be that good. We've left that part alone, however, so that the programmer can at least try for a fraction of a second time allowance. A SETTIM link with 4.5 seconds allowed will last longer, on average, than one with 4 seconds, and since it cost very few bytes to keep the floating point capability, we left it in. The Sidebar is well annotated, so our regular readers should be able to understand it without a line-by-line explanation. The business of checking the GROM is rather complicated, so we've taken Harry's word for how that works (see below). There are two different XB Test programs listed in the Sidebar, one that uses both INPUT and ACCEPT AT, and another that uses CALL KEY.

1.41.5. Harry's Explanation

The idea behind the routine at TIMEUP is to determine if the GPL instruction SCAN has just happened. If it has then loading >8375 with a key value and SOCing >837C with >2000 will be detected as a keystroke.

The GPL byte for SCAN is >03. Also in the course of SCAN the current address of the GROM pointer is pushed onto the GROM sub stack, and then popped from the stack. If we find that one level deeper in the GROM sub stack points to the next byte scheduled to be read from GROM and that the byte before this byte is a >03 then SCAN has just occurred.

When we come to the routine at TIMEUP the GROM registers are set so that XB can read the next GROM byte from >9800 our program can read the address of that byte from GRMRA at >9802 When this address is read it is automatically incremented one byte higher than where the actual read would have taken place. Therefore, the address needs to be DEC'd to point to the byte that would have been read in the normal sequence of events. If the address is DECT'd it then points to the most recent byte to be read from GROM. Once that byte is read from >9800, the pointer will autoincrement to the same address normally expected by XB. One catch is that if reading the address at GRMRA can result in, say, >A001. With DECT this becomes >9FFF. With the autoincrement reading the GROM byte should result in the pointer becoming >A000; the expected result. However after the autoincrement, the pointer is left pointing at >8000!!!!!!! Because of this you cannot read 1 byte lower than these addresses: >0000,>2000 >4000,>6000,>8000,>A000,>C000,>E000.

Next month's topic is one that some of our readers have asked for, namely the Bitmap Mode of operation. We'll include a small sample program that gets nicely into and out of the Bitmap Mode, so you can see how that's done.

```
*  SIDEBAR 41
*  "IT'S ABOUT TIME"
*
* FIRST, TODAY'S SOURCE CODE
*
* TIME DELAY INTERRUPT (TIMER/S)
* FOR USE WITH XB
* PUBLIC DOMAIN
* by Bruce Harrison and Harry Wilhelm
*
* REQUIRED EQUATES
*
XMLLNK EQU  >2018        XML LINK VECTOR
NUMREF EQU  >200C        NUMERIC REFERENCE
GPLWS  EQU  >83E0        GPL WORKSPACE
FMUL   EQU  >0E88        FLOATING POINT MULTIPLY
FAC    EQU  >834A        FLOATING POINT ACCUMULATOR
ARG    EQU  >835C        ARGUMENT
GSTAT  EQU  >837C        GPL STATUS BYTE
*
* BEGIN CODE SECTION
*
       DEF  SETTIM,ACT        DEFINE ENTRY POINTS
*
* ACT ACTIVATES (STARTS) THE TIMEOUT COUNT
*
ACT    MOV  @CHKON,@>83C4 PLACE ADDRESS OF INTERRUPT
       CLR  @CUMNUM       CLEAR COUNTER
       CLR  @GSTAT        CLEAR GPL STATUS
       RT                 RETURN
*
* SETTIM SETS THE LIMITING TIME FOR THE USER
```

```
* THROUGH A CALL LINK FROM XB
* ALLOWED TIME IS GIVEN IN SECONDS
*
SETTIM LWPI WS            USE OUR WORKSPACE
       CLR  R0            CLEAR R0 - NOT AN ARRAY
       LI   R1,1          FIRST PARAMETER
       CLR  @GSTAT        CLEAR GPL STATUS BYTE
       BLWP @NUMREF       GET PARAMETER
       LI   R9,SIXTY      POINT AT F.P. NUMBER 60
       LI   R10,ARG       AND AT ARGUMENT
       LI   R4,8          EIGHT BYTES TO MOVE
MOV1   MOVB *R9+,*R10+    MOVE A BYTE
       DEC  R4            DEC COUNT IN R4
       JNE  MOV1          IF NOT ZERO, REPEAT
       CLR  @GSTAT        CLEAR GPL STATUS
       BLWP @XMLLNK       USE XML LINKAGE
       DATA FMUL          MULTIPLY SECONDS BY SIXTY
       CLR  @GSTAT        CLEAR GPL STATUS
       BLWP @XMLLNK       USE XML LINK
       DATA >12B8         CONVERT NUMBER TO INTEGER
       MOV  @FAC,@LIMNUM MOVE TO LIMIT NUMBER
       LWPI GPLWS         LOAD GPL WORKSPACE
       B    @>6A          GO TO GPL INTERPRETER
*
* TIMER IS THE INTERRUPT ROUTINE
*
TIMER  LIMI 0             PREVENT INTERRUPTS
       CB   @>8375,@HX0C   look for signs of life at the keyboard
       JGT  TIMEU5        IF KEYS BEING PRESSED, JUMP
TIMER1 INC  @CUMNUM       INCREMENT TIME COUNT
       C    @CUMNUM,@LIMNUM  COMPARE TO LIMIT
       JGT  TIMEUP        IF GREATER, JUMP AHEAD (TIME EXPIRED)
BACK   B    *R11          ELSE RETURN

TIMEUP MOVB @>9802,R3     Reads address of GROM byte scheduled to be read next
       SWPB R3
       MOVB @>9802,R3
       SWPB R3
       DEC R3             the true address in GROM

      CZC @HX1FFF,R3     if address that was to be read was >A000 then we can't
      JNE TIMEU2        read one byte lower because >9FFF is impossible in GROM

       MOVB R3,@>9C02     restore GROM address.
       SWPB R3
       MOVB R3,@>9C02
       JMP BACK           and return

TIMEU2 DEC R3             want to read the preceding byte
       MOVB R3,@>9C02     GROM Write Address Register
```

```
       SWPB R3
       MOVB R3,@>9C02
       SWPB R3
       INC R3
       CLR R2
       MOVB @>9800,R2    Read preceding GROM byte; with autoincrement GROM
*                        pointer will be same as before we tinkered with it
       MOVB @>8373,R4    \
       SRL R4,8           checking previous stack entry to see if we've just
       AI R4,>8302        popped from the stack
       C R3,*R4          /
       JNE TIMEU1
       CI R2,>0300       GPL for keyscan

TIMEU1 JNE BACK          if EQ then keyscan just happened

       MOVB @ENTER,@>8375
       SOC @MASK,@>837C
TIMEU5 CLR @>83C4
       JMP BACK

* DATA SECTION
*
ENTER  BYTE >0D          ASCII FOR "ENTER" KEY
HX0C   BYTE >0C          enter minus 1
MASK   DATA >2000
HX1FFF DATA >1FFF
WS     BSS  32           OUR WORKSPACE
CUMNUM DATA 0            COUNT NUMBER (60THS OF A SECOND)
LIMNUM DATA 0            LIMIT NUMBER
SIXTY  BYTE 64,60,0,0,0,0,0,0  SIXTY IN F.P. NOTATION
CHKON  DATA TIMER        ADDRESS OF MAIN INTERRUPT
ANYKEY BYTE 32           KEY PRESSED VALUE
       END
*
*
* EXTENDED BASIC TEST PROGRAMS
* FOR TESTING THE ABOVE WITH XB
* (LISTED IN 28 COLUMNS)
*
* FIRST, USING AN INPUT STATEMENT
* AND AN ACCEPT AT

10 CALL INIT
20 CALL LOAD("DSK1.TIMER/O")
30 CALL CLEAR
40 CALL LINK("SETTIM",4.5)
50 CALL LINK("ACT")
60 INPUT "ANY STRING ":X$
70 IF X$="" THEN PRINT "TIME
```

```
'S UP" ELSE PRINT X$
80 DISPLAY AT(12,1):"ANY STR
ING"
90 CALL LINK("ACT")
100 ACCEPT AT(12,15):X$
110 IF X$="" THEN PRINT "TIM
E'S UP" ELSE PRINT X$
*
* SECOND XB TEST, USING CALL KEY
* THIS WILL RUN UNTIL STOPPED BY
* FCTN-4
*

10 CALL INIT
20 CALL LOAD("DSK1.TIMER/O")
30 CALL LINK("SETTIM",1)
40 CALL LINK("ACT")
50 CALL KEY(0,K,S):: IF S=0
THEN 50
60 N=N+1 :: PRINT N
70 GOTO 30
```

1.42. The Art Of Assembly — Part 42. At Long Last Bitmap

By Bruce Harrison

Seems we're forever thanking people in this column, and today we're going to put the thanks right up front. Thanks to John C. Johnson of Cedar Rapids, Iowa. John is a regular reader of our column, and a very skilled programmer. He read our complaint about not being able to get Bit-Map mode to work, and supplied some immediate help. His help was in the best possible form, as source code files on disk. These files included a correct way of setting up the Bitmap Mode and a plotting subroutine so that we can turn on any selected pixel on the screen. FANTASTIC! This was just exactly what we were hoping to be able to do! Today's column will be the first of two parts devoted to Bitmap Mode.

1.42.1. An Old Story

John's package to us included a very old article from the IUG magazine, in which Bill Gronos explained the operation of Bit-Map mode, and supplied the subroutines that John incorporated into his own "sample" programs. Long time readers of this column will remember that way back near the beginning of this series, we gave you a series of subroutines that could be used in Graphics Mode to do things like put strings on the screen and such. Today, we will do much the same for the Bitmap Mode. The column itself will be fairly short, as the Sidebar is a complete program that does some things to illustrate the use of the subroutines. This program, among other things, goes back and forth between the Graphics and Bitmap Modes to show that this can be done smoothly. We've also used some slick moves at the beginning of the program to capture the character definitions and color tables that were in use while we're in graphics mode. The character definitions are then used while we're in Bit-Map mode to print legends on the screen.

1.42.2. The Source Code

The Sidebar is a complete program, organized into three parts. The main program is called BITEXP/S. This is supported by subroutines in BITSUBS and by a data section called BITDATA. The main file uses the others with copy directives. The heart of all this is the BITSUBS file, where we've supplied subroutines to get you gracefully into and out of Bitmap Mode, to plot lines or curves pixel-by-pixel, and to put text strings or just single ASCII characters on screen. The coordinate system for the single-pixel plotting is similar to the Dot-Row and Dot-Column system you'd use for placing sprites. For characters or text, the Row and Column are used just like for Graphics Mode. In both cases, the subroutine will accept a color instruction placed in R9 before the BL. If R9 is clear, no change will be made to whatever color that part of the screen is already set up for. For characters or strings, R9 specifies in its left byte the foreground and background colors. For single pixel plotting, the left nybble of R9 specifies the foreground color for that pixel, with the background remaining whatever it was for that part of the screen.

1.42.3. The Program

It doesn't do a whole lot, but here's what. It puts a menu on-screen in Graphics Mode, with four choices offered. Any keystroke outside the 1-4 range will be ignored. Selecting 1 through 3 will cause a switchover to the Bitmap Mode, which will set up as black-on-white colors. For BOXES, a series of single-pixel boxes will appear, each one inside the previous one, until there's room for no more. All these boxes are made blue on white, since R9 was set to >4000 before we started drawing. Then a legend "BOXES" is written as a string in white on blue at the bottom of the screen. Once that's done, the computer just waits for a keypress, then returns to the Graphics Mode and displays the menu. The graphics mode colors and characters are put back where they belong in VDP RAM as part of the SETGM routine.

Selection 2 makes a Bitmap rectangular spiral on the screen, changing colors for each leg of each time around the screen. The vertical lines are all black, lines on the bottom are red, while lines across the top are blue. The word SPIRAL gets displayed at the bottom of the screen in white on dark green. Pressing a key returns to the menu.

Selection 3 uses those many data entries in BITDATA to draw a sine curve on the screen. The X-axis is labeled at 0, 90, 180, 270, and 360 degree spots, then "A SINE WAVE" appears twice — near the bottom of the screen in black on white, then again near the top in white on dark green. This is done just to illustrate an important difference between the normal Graphics mode and the Bitmap Mode. The same characters can be set into different colors on the same screen, provided only that the characters are written in a different place.

Selection 4 exits gracefully back to Editor/Assembler, but will also exit gracefully to Funnelweb if that's how it was loaded. The EXIT routine replaces six bytes at the location pointed to by >8370 so that DSR routines will work upon return to E/A or Funnelweb. We stashed away those six bytes as part of our opening section, so that on return to E/A, everything in E/A would work correctly.

1.42.4. The Subroutines

First in the subroutines is SETBM. This is the means of getting from Graphics Mode into Bitmap. We re-arranged the order of performing these steps from the original, so that the transition to Bitmap is made at the very end, after all the conditions have been set. This makes the transition appear "seamless", so no glitches appear during the changeover. We also took the part that clears out the character definition table into a separate subroutine so that it can be used by itself as a "clear screen" while you're in the Bitmap Mode. We also use that subroutine to clear things before our transition back to graphics mode, again to minimize any cosmetic problems in that transition.

The PLOT subroutine was copied as is from what John Johnson sent, but we added the ability to color the pixel just written. The CHAR feature was added to show that it could be done. Inventive readers will find this works for any characters, even those taken from revised character sets, CHARA1 files, and such. Now that we have cracked the door open, we'll try to pass along some more techniques and tips for Bitmap operations in future columns. As we write this, we're working on a little "toy" program to draw pictures and such. The TI world doesn't need any more such programs, but we learn a lot more by writing them than by using the existing programs, and learning is what this column is all about.

Because today's Sidebar is so large, we're going to save the explanation of how all this works for next month. Keep this one handy, so next time you'll be able to study how all this little "example" program works.

```
* SIDEBAR 42
* THREE FILES FOR BITMAP EXPERIMENTS
* FIRST, THE MAIN FILE - BITEXP/S
*
* Bitmap source code     01/15/94
* Loads from EA3 or Funnelweb choice number 4.
* code derived from John C. Johnson and Bill Gronos
* modified and combined by Bruce Harrison
* PUBLIC DOMAIN
* BITEXP/S
       DEF  START        DEFINE ENTRY POINT
       REF  VWTR,KSCAN,VMBW,VMBR,VSBW,VSBR
KEYADR EQU  >8374        KEY-UNIT ADDRESS
START  LWPI WS           LOAD OUR WORKSPACE
       LI   R0,>380      POINT AT COLOR TABLE
       LI   R1,SAVCLR    AND AT STORAGE SPACE
       LI   R2,32        32 BYTES TO GET
       BLWP @VMBR        READ COLOR TABLE INTO STORAGE
       MOV  @>8370,R0    GET VDP ADDR FROM >8370
       LI   R1,ANYKEY+1  POINT AT STORAGE BUFFER
       LI   R2,6         SIX BYTES TO READ
       BLWP @VMBR        READ THOSE INTO BUFFER
       LI   R0,>800      POINT AT CHARACTER TABLE
       LI   R1,CHRTBL    AND AT BUFFER STORAGE
       LI   R2,256*8     256 CHARACTER DEFINITIONS
       BLWP @VMBR        STASH CHARACTER DEFS
       CLR  @KEYADR      CLEAR KEY-UNIT
MENU   LI   R1,MENDAT    POINT AT MENU DATA
       LI   R0,32+6      ROW 2, COL 7 OF SCREEN
       BL   @DISSTR      DISPLAY MENU TITLE
       LI   R4,4         FOUR ITEMS IN MENU
       LI   R0,32*7+6    ROW 8, COL 7 OF SCREEN
MENU0  BL   @DISSTR      DISPLAY AN ITEM
       AI   R0,64        MOVE DOWN TWO ROWS
       DEC  R4           DECREMENT COUNT
       JNE  MENU0        IF NOT ZERO, REPEAT
       LI   R0,22*32+8   ELSE POINT ROW 23, COL 9
```

```
       BL   @DISSTR       DISPLAY SELECT LEGEND
MENKEY BL   @KEY          GET A KEYSTROKE
       MOV  @KEYADR,R8    MOVE KEY'S VALUE TO R8 AS WORD
       AI   R8,-49        SUBTRACT ASCII FOR "1"
       JLT  MENKEY        IF LESS THAN ZERO, REJECT
       CI   R8,3          COMPARE TO THREE
       JGT  MENKEY        IF GREATER, REJECT
       JEQ  EXIT          IF EQUAL, EXIT
       SLA  R8,1          ELSE DOUBLE NUMBER IN R8
       MOV  @LUT(R8),R12 GET BRANCH ADDRESS INTO R12
       BL   @SETBM        SET UP BITMAP MODE
       B    *R12          BRANCH TO SELECTED FUNCTION
EXIT   MOV  @>8370,R0     GET BACK >8370 ADDRESS
       LI   R1,ANYKEY+1   POINT AT BUFFER STORAGE
       LI   R2,6          SIX BYTES
       BLWP @VMBW         WRITE THOSE BACK TO VDP
       CLR  R0            CLEAR OUR R0
       LWPI >83E0         LOAD GPL WORKSPACE
       B    @>6A          RETURN TO GPL INTERPRETER
SINWAV LI   R12,SINDAT    POINT AT SINE DATA
       LI   R9,>C000      SET COLOR TO DARK GREEN
POINT  MOV  *R12+,R7      MOVE DOT COLUMN NUMBER TO R7
       AI   R7,20         ADD 20
       MOV  *R12+,R8      MOVE DOT ROW DATA TO R8
       CI   R12,CRSOVR    CHECK FOR COLOR CHANGE POINT
       JLT  POSWV         IF LESS THAN, SKIP AHEAD
       LI   R9,>6000      ELSE SET COLOR FOR DARK RED
POSWV  NEG  R8            MULTIPLY BY -1
       AI   R8,96         ADD COORDINATE ORIGIN
       BL   @PLOT         DRAW ONE PIXEL
       CI   R12,ENDDAT    ARE WE PAST DATA?
       JLT  POINT         IF NOT, REPEAT FOR NEXT PIXEL
       LI   R8,96         POINT AT DOT-ROW 96
       LI   R7,255        DOT-COLUMN 255
       LI   R9,>4000      SET COLOR FOR DARK BLUE
LINELP BL   @PLOT         PLOT ONE PIXEL AT DOT-ROW, DOT-COLUMN
       DEC  R7            DEC COLUMN COUNT
       JNE  LINELP        IF NOT ZERO, WRITE ANOTHER
       LI   R8,14         ROW 14
       LI   R7,3          COL 3
       LI   R6,48         CHARACTER 48 ("0")
       LI   R9,>4F00      COLOR DARK BLUE ON WHITE
       BL   @CHAR         PLACE THE CHARACTER
       LI   R7,9          COL 9
       LI   R12,LEG0      STRING LEG0
       BL   @BITSTR       DISPLAY STRING ("90")
       LI   R7,16         COL 16
       BL   @BITSTR       DISPLAY NEXT STRING ("180")
       LI   R7,23         COL 23
       BL   @BITSTR       NEXT STRING ("270")
```

```
        LI   R7,30        COL 30
        BL   @BITSTR      NEXT STRING ("360")
        CLR  R9           USE EXISTING COLORS (BLACK ON WHITE)
        LI   R8,23        ROW 23
        LI   R7,11        COL 11
        BL   @BITSTR      NEXT STRING "A SINE WAVE"
        LI   R12,LEG4     POINT BACK AT "A SINE WAVE"
        LI   R9,>FC00     COLOR WHITE ON DARK GREEN
        LI   R8,2         ROW 2
        LI   R7,14        COL 14
        BL   @BITSTR      DISPLAY STRING
        BL   @KEY         WAIT FOR KEYPRESS
        BL   @CPDT        CLEAR THE PATTERN TABLE
        BL   @SETGM       RE-SET TO GRAPHICS MODE
        B    @MENU        RETURN TO MENU
SPIRAL  LI   R12,20       STARTING DOT-COL 20
        LI   R13,10       STARTING DOT-ROW 10
        LI   R14,180      STOP ROW 180
        LI   R15,240      STOP COLUMN 240
        MOV  R13,R8       PUT START ROW IN R8
LINE    MOV  R12,R7       STARTING COL IN R7
        AI   R12,10       ADD 10 TO START COL
        CLR  R9           COLORS BLACK ON WHITE
LOOP1   BL   @PLOT        DRAW ONE PIXEL
        INC  R8           MOVE DOWN ONE ROW
        C    R8,R14       COMPARE TO LIMIT
        JL   LOOP1        IF LOW, REPEAT
        LI   R9,>6000     COLOR DARK RED
LOOP2   BL   @PLOT        DRAW ONE PIXEL
        INC  R7           INC COLUMN
        C    R7,R15       COMPARE TO LIMIT
        JL   LOOP2        IF LOW, REPEAT
        CLR  R9           COLOR BLACK ON WHITE
LOOP3   BL   @PLOT        DRAW ONE PIXEL
        DEC  R8           DEC ROW
        C    R8,R13       COMPARE TO TOP LIMIT
        JH   LOOP3        IF HIGH, REPEAT
        MOV  R13,R8       PUT LIMIT IN R8
        AI   R13,10       ADD 10 TO TOP LIMIT
        LI   R9,>4000     COLOR DARK BLUE
LOOP4   BL   @PLOT        DRAW ONE PIXEL
        DEC  R7           DEC COL
        C    R7,R12       COMPARE TO LEFT LIMIT
        JH   LOOP4        IF HIGH, REPEAT
        AI   R14,-10      SUBTRACT 10 FROM LEFT LIMIT
        AI   R15,-10      AND FROM STOP COLUMN
        C    R13,R14      COMPARE TOP AND BOTTOM LIMITS
        JLT  LINE         IF LESS, BACK TO START
        LI   R8,24        ROW 24
        LI   R7,15        COL 15
```

```
       LI   R9,>FC00     COLOR WHITE ON DARK GREEN
       LI   R12,SPISTR   STRING "SPIRAL"
       BL   @BITSTR      DISPLAY THAT
       BL   @KEY         ELSE WAIT FOR KEYSTROKE
       BL   @CPDT        CLEAR PATTERN TABLE
       BL   @SETGM       SET GRAPHICS MODE
       B    @MENU        RETURN TO MENU
BOXES  LI   R12,20       STARTING COLUMN
       LI   R13,10       STARTING ROW
       LI   R14,180      STARTING BOTTOM LIMIT
       LI   R15,240      STARTING RIGHT LIMIT
       LI   R9,>4000     COLOR DARK BLUE
BOX0   MOV  R12,R7       PUT COL IN R7
       MOV  R13,R8       PUT ROW IN R8
BOX01  BL   @PLOT        DRAW ONE PIXEL
       INC  R8           INC ROW
       C    R8,R14       COMPARE TO BOTTOM LIMIT
       JL   BOX01        IF LOW, REPEAT
BOX02  BL   @PLOT        DRAW ONE PIXEL
       INC  R7           INC COL
       C    R7,R15       COMPARE TO RIGHT LIMIT
       JL   BOX02        IF LOW, REPEAT
BOX03  BL   @PLOT        DRAW ONE
       DEC  R8           DEC ROW
       C    R8,R13       COMPARE TO LIMIT
       JH   BOX03        IF HIGH, REPEAT
BOX04  BL   @PLOT        DRAW ONE
       DEC  R7           DEC COL
       C    R7,R12       COMPARE TO LIMIT
       JH   BOX04        IF HIGH, REPEAT
       AI   R12,10       ADJUST START COLUMN
       AI   R13,10       ADJUST START ROW
       AI   R14,-10      ADJUST BOTTOM ROW
       AI   R15,-10      ADJUST BOTTOM COL
       C    R13,R14      COMPARE TOP, BOTTOM LIMITS
       JLT  BOX0         IF LESS THAN, DRAW NEXT BOX
       LI   R8,24        ROW 24
       LI   R7,15        COL 15
       LI   R9,>F400     COLOR WHITE ON DARK BLUE
       LI   R12,BOXSTR   STRING "BOXES"
       BL   @BITSTR      DISPLAY THAT
       BL   @KEY         WAIT FOR KEYSTROKE
       BL   @CPDT        CLEAR PATTERN TABLE
       BL   @SETGM       SET GRAPHICS MODE
       B    @MENU        BACK TO MENU
       COPY "DSK1.BITSUBS" COPY IN SUBROUTINES
       COPY "DSK1.BITDATA" COPY IN DATA FILE
       END
* END OF BITEXP/S
*
```

```
* SECOND FILE - BITSUBS
* 15 JAN 1994
*
* SUBROUTINES FOR HANDLING BITMAP
* OPERATIONS AND TRANSITIONS
*
* FOLLOWING SECTION SETS COMPUTER INTO BITMAP MODE
*
SETBM  LI   R0,>206      SET TO WRITE VDP REGISTER 2
       BLWP @VWTR        SIT TO >1800  (SCREEN IMAGE TABLE)
       LI   R0,>403      SET TO WRITE TO VDP REG. 4
       BLWP @VWTR        PDT TO >0000  (PATTERN DESCRIPTOR TABLE)
       LI   R0,>3FF      SET TO WRITE TO VDP REG 3
       BLWP @VWTR        CT TO >2000   (COLOR TABLE)
       LI   R0,>607      SET TO WRITE VDP REG 6
       BLWP @VWTR        Sprite descriptor table to >3800
       LI   R0,>570      SET TO WRITE VDP REG 7
       BLWP @VWTR        Sprite attribute list to >3800
       LI   R0,>58       INITIALIZE SCREEN IMAGE TABLE (SIT) (AT >1800)
       MOVB R0,@>8C02    WRITE LOW BYTE VDP ADDRESS
       SWPB R0           SWAP R0
       MOVB R0,@>8C02    WRITE HIGH BYTE VDP ADDRESS
       LI   R0,3         THREE TABLES OF 256 BYTES EACH
       CLR  R1           START WITH ZERO
SIT    MOVB R1,@>8C00    WRITE TO VDP (SELF-INCREMENTING)
       AI   R1,>100      ADD 1 TO HIGH BYTE R1
       JNE  SIT          IF NOT ZERO, REPEAT
       DEC  R0           ELSE DEC COUNT
       JNE  SIT          IF NOT ZERO, REPEAT
       LI   R0,>60       INIT COLOR TABLE  (CT) AT >2000
       MOVB R0,@>8C02    WRITE LOW BYTE OF ADDRESS
       SWPB R0           SWAP R0
       MOVB R0,@>8C02    WRITE HIGH BYTE OF ADDRESS
       LI   R0,>1800     >1800 BYTES TO WRITE
       LI   R1,>1F00     COLORS ALL BLACK ON WHITE
CT     MOVB R1,@>8C00    WRITE ONE BYTE
       DEC  R0           DEC COUNT
       JNE  CT           IF NOT ZERO, REPEAT
       MOV  R11,R14      STASH RETURN ADDRESS
       BL   @CPDT        CLEAR PATTERN TABLE
       LI   R0,2         SET R0 TO WRITE 2 TO VDP REGISTER ZERO
       BLWP @VWTR        SET TO M3 MODE (BITMAP)
       B    *R14         RETURN
CPDT   LI   R0,>40       CLEAR PATTERN DESCRIPTOR TABLE  (PDT) AT >0000
       MOVB R0,@>8C02    WRITE LOW BYTE ADDR
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH BYTE ADDRESS
       LI   R0,>1800     >1800 BYTES TO WRITE
       CLR  R1           ALL ZEROS
PDT    MOVB R1,@>8C00    WRITE ONE
```

```
       DEC  R0           DEC COUNT
       JNE  PDT          IF NOT ZERO, REPEAT
       RT
*
* FOLLOWING SETS COMPUTER BACK TO NORMAL GRAPHICS MODE
*
SETGM  LI   R0,>1E0      SET TO WRITE VDP REG 1
       BLWP @VWTR        WRITE
       LI   R0,>200      SET TO WRITE VDP REG 2
       BLWP @VWTR        WRITE
       LI   R0,>401      SET TO WRITE VDP REG 4
       BLWP @VWTR        WRITE
       LI   R0,>30E      VDP REG 3
       BLWP @VWTR        WRITE
       LI   R0,>600      VDP REG 6
       BLWP @VWTR        WRITE
       LI   R0,>506      VDP REG 5
       BLWP @VWTR        WRITE
       LI   R0,>380      POINT AT COLOR TABLE
       LI   R1,SAVCLR    AND AT SAVED COLOR DATA
       LI   R2,32        32 BYTES
       BLWP @VMBW        WRITE THE COLOR TABLE BACK
       LI   R0,>800      POINT AT GRAPHICS CHAR TABLE
       LI   R1,CHRTBL    AND AT STORED CHARACTER DATA
       LI   R2,256*8     256 CHARACTERS
       BLWP @VMBW        WRITE CHARACTER DEFS BACK
       CLR  R0           PREP TO WRITE VDP REG 0
       BLWP @VWTR        WRITE THAT TO REMOVE BITMAP
       RT                RETURN
*
* FOLLOWING WRITES ONE PIXEL TO SCREEN AT LOCATION POINTED BY
* R8 (DOT ROW) AND R7 (DOT COLUMN)
*
PLOT   MOV  R7,R3        MOVE DOT COLUMN TO R3
       MOV  R8,R4        AND DOT ROW TO R4
       MOV  R4,R5        DOT ROW ALSO IN R5
       ANDI R5,7         R5 HAS DOT ROW MODULO 8
       SZC  R5,R4        SO DOES R4
       SLA  R4,5         MULTIPLY R4 BY 32
       A    R5,R4        ADD R5, SO R4 HAS DR MOD. 8 * 32 + DR MOD 8
       MOV  R3,R0        MOVE DOT COL TO R0
       ANDI R0,>FFF8     R0 HAS DC - DC MOD 8
       S    R0,R3        R3 HAS DC MOD 8
       A    R4,R0        ADD R4
       SWPB R0           SWAP BYTES
       MOVB R0,@>8C02    WRITE LOW ADDRESS BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH ADDRESS BYTE
       NOP               WASTE TIME
       MOVB @>8800,R1    READ THE BYTE
```

```
        SOCB @M(R3),R1    OVERLAY MASK FROM TABLE M
        ORI  R0,>4000     SET THE 4000 BIT IN R0
        SWPB R0           SWAP
        MOVB R0,@>8C02    WRITE LOW BYTE OF ADDRESS
        SWPB R0           SWAP
        MOVB R0,@>8C02    WRITE HIGH BYTE OF ADDRESS
        NOP               WASTE TIME
        MOVB R1,@>8C00    WRITE MODIFIED BYTE BACK TO VDP
        MOV  R9,R9        IS COLOR TO BE SET?
        JEQ  PLOTX        IF NOT, JUMP AHEAD
        ANDI R0,>3FFF     STRIP OFF "4" FROM R0
        AI   R0,>2000     ADD >2000 TO POINT AT COLOR TABLE ENTRY
        BLWP @VSBR        READ THAT BYTE INTO R1
        MOVB R1,R2        MOVE THE BYTE TO R2
        ANDI R2,>F000     STRIP ALL BUT LEFT NYBBLE
        CB   R2,R9        COMPARE TO LEFT BYTE R9
        JEQ  PLOTX        IF EQUAL, COLOR ALREADY SET
        ANDI R1,>0F00     ELSE STRIP OFF LEFT NYBBLE R1
        AB   R9,R1        REPLACE WITH LEFT NYBBLE R9
        BLWP @VSBW        THEN WRITE COLOR BYTE BACK
PLOTX   RT                RETURN
BITSTR  MOV  R11,R15      STASH R11
        MOV  R7,R13       SAVE COLUMN IN R13
        MOVB *R12+,R4     GET STRING LENGTH BYTE IN R4
        JEQ  BITSX        IF ZERO, SKIP PROCESS
        SRL  R4,8         RIGHT JUSTIFY
BITST0  MOVB *R12+,R6     MOVE ONE BYTE OF STRING TO R6
        SRL  R6,8         RIGHT JUSTIFY
        BL   @CHAR        DISPLAY THAT CHARACTER
        INC  R7           INC COLUMN
        DEC  R4           DEC LENGTH COUNT
        JNE  BITST0       IF NOT ZERO, REPEAT
        MOV  R13,R7       PUT COLUMN BACK IN R7
BITSX   B    *R15         ELSE RETURN
CHAR    MOV  R8,R0        PUT ROW COUNT IN R0
        DEC  R0           DEC TO ZERO-BASE NUMBER
        LI   R2,8         PUT 8 IN R2
        SLA  R0,5         MULTIPLY R0 BY 32
        A    R7,R0        ADD COLUMN
        DEC  R0           DEC TO ZERO-BASE COLUMN
        SLA  R0,3         MULTIPLY R0 BY 8
        MOV  R6,R1        PUT CHARACTER FROM R6 INTO R1
        SLA  R1,3         MULTIPLY BY 8
        AI   R1,CHRTBL    ADD START OF STORED CHARACTER DEFINITIONS
        BLWP @VMBW        WRITE 8 BYTES TO VDP RAM
        MOV  R9,R9        CHECK FOR COLOR CHANGE
        JEQ  CHARX        IF NONE, SKIP AHEAD
        AI   R0,>2000     ELSE ADD COLOR TABLE OFFSET
        MOVB R9,R1        MOVE COLOR BYTE TO R1
CHCL    BLWP @VSBW        WRITE ONE BYTE
```

```
       INC  R0           POINT AT NEXT LOCATION
       DEC  R2           DEC COUNT IN R2
       JNE  CHCL         IF NOT ZERO, REPEAT
CHARX  RT                RETURN
KEY    BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN TURN THEM OFF
       CB   @ANYKEY,@>837C KEY STRUCK?
       JNE  KEY          IF NOT, SCAN AGAIN
       RT                ELSE RETURN
DISSTR MOVB *R1+,R2      GET STRING LENGTH INTO R2
       SRL  R2,8         RIGHT JUSTIFY
       BLWP @VMBW        WRITE STRING TO SCREEN
       A    R2,R1        ADD LENGTH TO POINTER
       RT                RETURN
* END OF BITSUBS
*
* THIRD FILE - BITDATA
* 15 JAN 1994
*
*
* DATA SECTION
*
WS     BSS  >20          OUR WORKSPACE
M      DATA >8040,>2010,>0804,>0201  MASK DATA
*
* FOLLOWING IS DATA FOR THE SINE WAVE
* EACH PAIR OF WORDS IS ONE POINT IN DOT ROW, DOT COLUMN (ZERO BASED)
*
* THIS WAS CREATED USING AN XB PROGRAM
* TO CORRECTLY SCALE THE DATA FOR BITMAP SCREEN
*
SINDAT DATA 0,0
       DATA 1,2
       DATA 2,5
       DATA 3,8
       DATA 4,11
       DATA 5,14
       DATA 6,17
       DATA 7,20
       DATA 9,23
       DATA 10,26
       DATA 11,29
       DATA 12,32
       DATA 13,34
       DATA 14,37
       DATA 15,40
       DATA 16,43
       DATA 18,45
       DATA 19,48
```

```
DATA 20,50
DATA 21,53
DATA 22,55
DATA 23,58
DATA 24,60
DATA 25,62
DATA 27,64
DATA 28,67
DATA 29,69
DATA 30,71
DATA 31,73
DATA 32,74
DATA 33,76
DATA 35,78
DATA 36,80
DATA 37,81
DATA 38,83
DATA 39,84
DATA 40,85
DATA 41,87
DATA 42,88
DATA 44,89
DATA 45,90
DATA 46,91
DATA 47,91
DATA 48,92
DATA 49,93
DATA 50,93
DATA 51,94
DATA 53,94
DATA 54,94
DATA 55,94
DATA 56,94
DATA 57,94
DATA 58,94
DATA 59,94
DATA 60,94
DATA 62,93
DATA 63,93
DATA 64,92
DATA 65,92
DATA 66,91
DATA 67,90
DATA 68,89
DATA 70,88
DATA 71,87
DATA 72,86
DATA 73,84
DATA 74,83
DATA 75,81
```

```
        DATA 76,80
        DATA 77,78
        DATA 79,77
        DATA 80,75
        DATA 81,73
        DATA 82,71
        DATA 83,69
        DATA 84,67
        DATA 85,65
        DATA 86,63
        DATA 88,60
        DATA 89,58
        DATA 90,56
        DATA 91,53
        DATA 92,51
        DATA 93,48
        DATA 94,46
        DATA 96,43
        DATA 97,40
        DATA 98,38
        DATA 99,35
        DATA 100,32
        DATA 101,29
        DATA 102,26
        DATA 103,23
        DATA 105,21
        DATA 106,18
        DATA 107,15
        DATA 108,12
        DATA 109,9
        DATA 110,6
        DATA 111,3
        DATA 112,0
CRSOVR  DATA 114,-3
        DATA 115,-6
        DATA 116,-9
        DATA 117,-12
        DATA 118,-15
        DATA 119,-18
        DATA 120,-21
        DATA 121,-24
        DATA 123,-27
        DATA 124,-29
        DATA 125,-32
        DATA 126,-35
        DATA 127,-38
        DATA 128,-41
        DATA 129,-43
        DATA 131,-46
        DATA 132,-48
```

```
DATA 133,-51
DATA 134,-53
DATA 135,-56
DATA 136,-58
DATA 137,-61
DATA 138,-63
DATA 140,-65
DATA 141,-67
DATA 142,-69
DATA 143,-71
DATA 144,-73
DATA 145,-75
DATA 146,-77
DATA 147,-79
DATA 149,-80
DATA 150,-82
DATA 151,-83
DATA 152,-85
DATA 153,-86
DATA 154,-87
DATA 155,-89
DATA 156,-90
DATA 158,-91
DATA 159,-92
DATA 160,-92
DATA 161,-93
DATA 162,-94
DATA 163,-94
DATA 164,-95
DATA 166,-95
DATA 167,-95
DATA 168,-95
DATA 169,-95
DATA 170,-95
DATA 171,-95
DATA 172,-95
DATA 173,-95
DATA 175,-94
DATA 176,-94
DATA 177,-93
DATA 178,-93
DATA 179,-92
DATA 180,-91
DATA 181,-90
DATA 182,-89
DATA 184,-88
DATA 185,-87
DATA 186,-85
DATA 187,-84
DATA 188,-83
```

```
       DATA 189,-81
       DATA 190,-79
       DATA 192,-78
       DATA 193,-76
       DATA 194,-74
       DATA 195,-72
       DATA 196,-70
       DATA 197,-68
       DATA 198,-66
       DATA 199,-64
       DATA 201,-62
       DATA 202,-59
       DATA 203,-57
       DATA 204,-55
       DATA 205,-52
       DATA 206,-50
       DATA 207,-47
       DATA 208,-44
       DATA 210,-42
       DATA 211,-39
       DATA 212,-36
       DATA 213,-33
       DATA 214,-31
       DATA 215,-28
       DATA 216,-25
       DATA 217,-22
       DATA 219,-19
       DATA 220,-16
       DATA 221,-13
       DATA 222,-10
       DATA 223,-7
       DATA 224,-4
       DATA 225,-1
ENDDAT EQU  $            END OF SINE DATA
LUT    DATA BOXES,SPIRAL,SINWAV  MENU BRANCH TABLE
MENDAT BYTE 21
       TEXT 'BITMAP SELECTION MENU'
       BYTE 16
       TEXT '1.  BITMAP BOXES'
       BYTE 17
       TEXT '2.  BITMAP SPIRAL'
       BYTE 20
       TEXT '3.  BITMAP SINE WAVE'
       BYTE 15
       TEXT '4.  EXIT TO E/A'
       BYTE 16
       TEXT 'SELECT BY NUMBER'
LEG0   BYTE 2
       TEXT '90'
LEG1   BYTE 3
```

```
       TEXT '180'
LEG2   BYTE 3
       TEXT '270'
LEG3   BYTE 3
       TEXT '360'
LEG4   BYTE 11
       TEXT 'A SINE WAVE'
BOXSTR BYTE 5
       TEXT 'BOXES'
SPISTR BYTE 6
       TEXT 'SPIRAL'
ANYKEY BYTE >20          COMPARISON BYTE FOR KEYSTROKE
       BSS  6            STORAGE FOR DSR DATA FROM VDP RAM
SAVCLR BSS  32           STORAGE FOR GRAPHICS COLOR TABLE
CHRTBL BSS  256*8        STORAGE FOR GRAPHICS CHARACTER DEFINITIONS
```

1.43. The Art Of Assembly — Part 43. Bit-Map Explained

By Bruce Harrison

Many years ago, your author found a book in the Navy Department's library called "Television Really Explained". This turned out to be a British publication. It was intended to explain the operation of a television system in terms for the most rank beginner. One line from that book sticks in the mind to this day. It describes the motion of the electron beam in the picture tube as "moving across the screen at amazing speed". Nicely said! No need to understand microseconds, reflex scanning systems, or anything technical like that. Just "amazing speed". We could sum up this whole series of articles just that simply: Assembly Language executes at "amazing speed".

1.43.1. Last Month's Sidebar

Now's the time to get out last month's issue. We'll wait. ..... Okay, now find the page with our "Sidebar 42" on it. ..... Here we go, ready or not. Down to the label SINWAV is mostly pretty standard stuff, including a menu setup to offer a selection of actions, and an exit routine to get out of the program. Just above the label EXIT is the first thing that's different from the usual. Before branching out to the selected item, there's a BL @SETBM. That uses a subroutine to set all the required conditions for the Bitmap Mode of operations. From there on, each operation takes place in Bitmap. Let's now skip down the page a bit to label SETBM, in file BITSUBS. This, with the auxiliary subroutine CPDT, is all that's needed to take your TI from its normal "Graphics" mode into a cleared Bitmap screen.

SETBM begins with a few "write to register" operations, which set up the VDP tables to locations compatible with the Bitmap Mode. The screen image table is set to >1800, the pattern descriptor table to 0, the color table to >2000, and the sprite descriptor table and sprite attribute list are both set to >3800. (This program does not use sprites, so the tables are set there just to insure we don't get any undesired sprites by accident.) Once that's done, the table areas are filled with the appropriate bytes to create a blank screen, black-on-white color combination, and to fill the screen image table with bytes from 0 through 255, repeated three times. The screen image table, then, is left alone at all times, while the pattern table, with 3 sets of 255 eight-byte character patterns, determines what's seen on the screen. Things get written to the screen (and colorized) by writing individual bits to the pattern descriptor table and bytes to the color table.

1.43.2. The Sine Wave

Now that we've used SETBM to put us in Bitmap Mode, let's go back into the main part of our code, at label SINWAV. As the name implies, this will plot a sine wave on the screen as single pixel dots. It starts by pointing R12 at the large block of data labeled SINDAT. Next it sets R9 to >C000, which will make the dots plotted appear in Dark Green. We complicated things a little here by making the upper half of the sine wave plot in green, and the lower half in red. In the data that R12 is pointing to, each pair of words gives the X (horizontal) and Y (vertical) values for one spot on the sine curve. Thus when we MOV *R12+,R7, we've taken the X value into R7. This value is expressed in Dot-Row coordinates. We add 20 just to move the sine wave in a bit from the screen edge. Now we get the vertical coordinate into R8 by MOV *R12+,R8. At this point we check the value in R12 to see whether we're still in the positive half-cycle. If we are, we skip ahead to label POSWV, else we change the color nybble in R9 to >6000, so the points will be plotted in dark red. Now the number in R8 is set up for a zero-based vertical coordinate, but what we want is to have things centered around dot-row 96, or halfway down the screen. Thus we take the number in R8 and NEG it, then add our offset of 96, and we're ready to place one pixel on the screen.

To do that, we use the subroutine PLOT. Plot takes the numbers in R7 and R8, and calculates from them the exact bit that represents the pixel at those coordinates. It uses Modulo math processes to accomplish this, and the source code is pretty well annotated. The methods used here to calculate Modulos are short cut methods that take full advantage of the number base in use, and are not general purpose Modulo routines. Let's say we wanted to calculate 10 Modulo 3. The general purpose way to do this would be as follows:

```
LI   R3,10
CLR  R2
LI   R1,3
DIV  R1,R2
```

After this DIV, R3 would contain 10 modulo 3, which happens to equal 1, since that's the remainder after dividing 10 by 3. The quotient 3 would be left in R2, but that's of no importance. This same thing could be done in Basic, by taking this kind of steps: A=3 :: B=10 :: M=(B/A-INT(B/A))*A M would then contain the answer 1. (On some computers, M would contain .9999999. . . instead of 1, but you get the idea, or at least we hope you do.)

Dividing, however, takes time, so the subroutine PLOT uses shortcuts that take advantage of the fact that it's MODULO 8 that we're after, thus doing its calculations more quickly. Once it's done its Modulo math, the routine reads one byte from VDP, sets just the correct bit in that byte, than writes the byte back to VDP. This places one dot at the correct spot on the screen in the dot-row, dot-column coordinate system. The last part of PLOT checks to see if R9 is non-zero, which means that a color has been specified. It then checks the appropriate byte in the color table by adding >2000 to R0 and doing a VSBR. If the color specified by R9 has already been set in that byte, it's left alone. Otherwise the routine sets the color from the left nybble of R9 into the foreground nybble, and writes that byte back into the color table in VDP.

SINWAV then just keeps repeating all of this until it runs out of data, and by then the sine curve is all plotted on the screen. Now we put in a visible X-axis by plotting a line with the code at LINELP. This puts 255 dots across dot-row 96. The next trick is to print the legends on the screen. That uses the subroutines at BITSTR and CHAR.

BITSTR is similar in concept to our old routine DISSTR for displaying strings in the standard Graphics and Text modes. Before calling BITSTR, we set R8 to the desired Graphics row, R7 to the desired Graphics column, R9 to the color scheme desired, and R12 to the address of the string to be displayed. As in the case of DISSTR, the string begins with a length byte, followed by the text content of the string. BITSTR starts by stashing the R11 return address in R15, then saving the starting column number in R13. Next it gets the length of the string into R4, and if that's zero, jumps to the exit. Assuming the length is at least one byte, that number is right justified in R4 to serve as a character count for the loop at BITST0. That loop gets a byte from the string into R6, right justifies that byte, then uses the subroutine CHAR to display it on the Bitmap screen. BITST0 continues this process, incrementing the column in R7 on each pass, until the count in R4 becomes zero.

The subroutine CHAR is somewhat strange, in that it writes the eight bytes of the character pattern into the pattern table, rather than the character itself into the screen image table. Back at the very early part of this program, we grabbed all the existing character definitions for the graphics mode from VDP >800 into memory at location CHRTBL. We did that precisely for the use that CHAR will make of that data at CHRTBL. CHAR starts by getting the desired row from R8 into R0. Remember that these are not Dot-Rows, but the normal 1-24 graphics row numbers. Char decrements the row count in R0 to zero-base the number. Next, it loads R2 with 8, which we'll need later. Now the number in R0 gets multiplied by 32, so it represents a graphics screen position. The desired Graphics column from R7 is added, then R0 is DEC'd again to zero-base the column. Since the character definition table is in eight-byte chunks, we multiply the number now in R0 by eight, so it points to the correct part of the pattern definition table. Now to get the character data, we take the desired character number from R6 into R1, multiply that by eight to index into our CHRTBL block in memory, then add the address of the start of the saved character definitions. (AI R1,CHRTBL)

At this point, R0 points to the correct place in the character definition table in VDP, R1 points to the pattern for the desired character in CHRTBL, and R2 contains 8, which is the number of bytes needed to define one character. BLWP @VMBW makes that desired character appear at the desired row and column on the screen. Now R9 is checked, and the desired color scheme for this character is written to the color table if needed. By this method, we could in theory write each character in a string in a different color scheme. (The present subroutine BITSTR doesn't allow that, but the creative programmer can see how to do it.)

The ability to write strings of characters at various places on the screen with different color schemes, as this program does, is an important difference between Bitmap and the normal Graphics Mode. That and the ability to write selectively to any pixel on the screen makes Bitmap Mode a powerful tool. The one significant drawback in all this is that almost all of the VDP RAM space gets eaten up. In this particular case, that's no problem, but if your program needs VDP space for file buffers and such, then the crunch can indeed get serious.

When the sine wave is all plotted and all the legends are printed, it executes a BL @KEY, which just puts the computer into a key-sensing loop until any key has been pressed. Given a keypress, it uses the subroutine CPDT (Clear Pattern Descriptor Table) to clear the Bitmap screen, then uses SETGM (Set Graphics Mode) to put the VDP back into its "normal" graphics mode. This routine starts by re-setting all the VDP Registers except #0 to their "normal" values, then puts back the color table and character definition table data that was saved at the start of the program. Finally, with everything in place, it clears VDP Register 0, to put the VDP back in graphics mode, and returns to the main code. After BL @SETGM, the SINWAV exits by branching back to label MENU.

The other two "main" routines, called SPIRAL and BOXES, work in a similar fashion, except that no curves are plotted, and all lines drawn are straight horizontal and vertical ones. They use the same subroutines to do these jobs, so we'll spare you the description of all that, and leave analysis of SPIRAL and BOXES as an exercise for the serious student.

Now is the time to mention one small limitation to the capability of Bitmap. One can individually set pixels on and off, but the color table entries of eight bits each affect eight pixels in the pattern table. Thus if we set a color byte for a spot in the image, a total of eight pixels including that one will have this same color scheme.

Let's try to illustrate this. Imagine that we are looking at a single-byte section of the pattern descriptor table and the corresponding byte in the color table:

Pattern Table location >0508 - bits 00101100

Color Table location >2508 - byte >4F

With this setup, each of the ones in the eight bits at >0508 will result in a dark blue pixel on the screen, while each of the zeros in those eight bits will be a white spot on the screen. Thus this one byte in the color table affects eight adjacent pixels in the image we see, so when we set a color scheme for any of the pixels at >0508 by writing a color byte to >2508, the background and foreground colors we've set apply to all eight of those pixels. This means that, while we can turn individual pixels on or off one at a time, we can only set their colors in groups of eight, not by individual pixel.

After getting our feet wet in Bitmap with the program for last month's column, we tried getting even more adventurous, and created a Drawing program. In future columns, we'll pass along some more things learned from that effort, including "UNPLOT", to erase a single pixel from the Bitmap screen, and an algorithm to create straight lines from any Dot-Row, Dot-column position to any other Dot-Row, Dot-Column position. The Drawing program itself has been released to Public Domain, and is available from the Lima Users' Group. (Disk 928 in their library, which includes all the source code and sample pictures.) That program is really just a toy, but we have used it to make drawings for our childrens' science fair projects, one of which won a blue ribbon.

Next month we'll be covering two topics: The first will be how to deal with those prompts the Assembler issues, and then the much more difficult topic of doing a CALL FILES from Assembly.

Good luck with your own experiments.

1.44. The Art Of Assembly — Part 44. By Popular Request

By Bruce Harrison

Many moons ago, we got a request from one reader for a little help getting through such things as the Prompts given by the TI Assembler. There was only one letter, from one reader, but by our standards that constitutes "popular request". This took some experimenting, since we never use that Assembler. We got out the necessary disks and refreshed our tired old memory. Here's how the prompts go, and how to answer them:

LOAD ASSEMBLER? (Y/N) - This means "are you ready to load the files ASSM1 and ASSM2 from the disk in drive 1?" The E/A module expects to find those files on Drive 1, and since you might not have a disk with those files in place, this prompt gives you a chance to put one there before suffering the error report that will happen if E/A can't find those files on that drive. On our own system, we don't generally use the module, but have E/A on our P-Gram, and we've modified it to look on Drive 5 of our Ramdisk for any of its standard files.

SOURCE FILE NAME? Fill in the complete name, as in DSK1.MYPROG/S, then press **ENTER**. The Assembler will check for the existence of this file, lighting the appropriate drive briefly.

OBJECT FILE NAME? Fill in the name for the output file, as in DSK2.MYPROG/O, then press **ENTER**. This too will briefly light up a disk drive, thus assuring a disk is available.

LIST FILE NAME? Most of the time, you'll want to leave this blank. On those occasions when you need a listing, you'll most likely use the PIO port in answer to this prompt, so a listing will be printed.

OPTIONS? This causes probably the most grief. There are four possible options, each represented by one upper case letter, and they may all be entered, in any order. R means that the Assembler will expect registers to include the R in their names (e.g. LI R0,35). In our own work, we always use this option. L in the options means a listing is to be produced on the device named as LIST FILE NAME above. A listing can only be produced if both the name and the L are present. S means that the listing will include a symbol table at its end. This can be very useful, as it will list all labels used or referenced in the source file, with their addresses. Finally, there's the C option, which will make the object file in Compressed format. That saves disk space, but we never use it. First, this makes a non-readable object file, and second, compressed files can't be loaded except by the E/A or Mini-Memory loaders.

In our own work, we never use the TI Assembler anyway, mainly because its error reporting is lousy. We prefer using the RAGASM by Art Green, and keep that ever at hand on our Ramdisk. Here are the prompts for Art Green's Assembler:

MACROS: We always leave this blank, since we don't use MACROs.

SOURCE: Same as for the first prompt on TI's Assembler.

OBJECT: Same as for the second prompt on TI's Assembler.

ID/DATE: Can be used to date the object file for identification. We never use this, but just leave the entry field blank.

OPTIONS: Accepts the same four letters as the TI version, and they mean the same thing. The List file name normally defaults to PIO. This can be changed by an INSTALL procedure. In that same procedure, one may also designate options to be used by default. For example, in our own use, we've set up RAGASM so that it defaults to R option, thus saving us the trouble of typing that in for each assemble. If, however, we do put in any options, then we must include R, since any entry in this field voids the default entries.

MORE: When RAGASM finishes an Assemble, it will prompt with MORE. Any answer other than **Y ENTER** will exit the program. **Y ENTER** will take you back to the MACROS: prompt for another assembly operation.

1.44.1. CALLs Revisited

Okay, so your author is just plain stubborn about some things. Some time back, we reported making attempts to use CALLs directly from Assembly code. We'd tried to get some understanding of the process, particularly for CALL FILES, as can be used from Basic and Extended Basic. We'd learned where the code was located, at >5D5A in the TI disk controller's ROM. That information did not, however, help us with the problem of passing along the one parameter that this call needs. Being so stubborn, we just couldn't give up this idea, and so resorted to the brute force method of dis-assembling the entire ROM program, then tracing painfully through its contents. This is difficult to do, as the code contains many BLs and BLWPs, so one must plod through a mountain of code to determine what's really happening. One must also make some educated guesses about what the registers will contain at various places, since the code has lots of things like this: MOV @>0070(R9),@>0034(R9). Our guess (which proved correct) was that R9 must contain >8300 at this point, so that the resulting addresses in the example above would be >8370 and >8334.

After a lot of slogging through, including a step-by-step run with Miller's Explorer, we found that the answer to the riddle was >832C. That is, when the call process starts, that word in RAM Pad had to point to the location in VDP RAM where the length of the name FILES was located. The actual number parameter had to be encoded in tokenized form as >B7,>C8,>01,>3x,>B6. (The x is a number from 1 through 9.) This had to be written to VDP immediately following the PAB.

1.44.2. The Solution

Today's Sidebar is a complete program, written to allow us to pick a number between 1 and 9, and have CALL FILES execute to make room for that number of files in VDP RAM. By itself, this isn't really useful except to prove a point. It proves only that we can CALL FILES from Assembly, and have that execute correctly. One could, however, take pieces of today's Sidebar, incorporate them into his own program, and thus have the number of files set up correctly for that program's needs. The three essential elements are the inclusion of REF DSRLNK, the data starting with PABDT and ending with ANYKEY, and the code section between label MKCALL and the entry DATA >A, a few lines below. The labels CALPNT and PABPNT are defined as >832C and >8356, respectively.

You'll note that this PAB data does not need to contain the usual ten bytes that are required in most PABs. Just the length of the name, followed by the name FILES, then the data for the number in tokenized form. >B7 is the token for "(", C8 means that what follows is an unquoted string, >01 is the length of this string, and >3x is the number of files desired, where x is any number from 1 through 9. Finally, the token >B6 stands for ")", signaling the end of the parameter string.

As shown in the Sidebar, this little program includes a little "window dressing", in the form of prompts, a routine to display those, and a routine to display a number in decimal form. It starts with a prompt at the beginning for the number of files you want capable of being open simultaneously. This can be any number from 1 through 9. Just a single keypress answers this prompt, and any key outside the range 1 through 9 will be rejected. The selected number will appear on the screen below this prompt, then a couple more numbers will appear. The numbers displayed are taken from location >8370. The first number displayed shows the address for the highest available byte in VDP RAM after the space for file processing has been set aside. On a "cold" start, this number will be >37D7, or 14295 in decimal. That indicates that the default number (3) has been set for FILES. The number shown below that is the number after the CALL FILES has been performed. If the number of files selected is above 3, then this number will be less than 14295. If the number selected was less than 3, then this will show a number greater than 14295.

When it's done, there will be a "press any key" prompt at the bottom of the screen. Pressing any key will get you out of the program. You can then re-run this little exercise by selecting 4 from the E/A menu, then pressing **ENTER** at the Program Name prompt. If you do this, you can see by the numbers shown that the previous run has indeed left the correct number at address >8370. The limits on that number are 11187 for 9 files through 15331 for 1 file. Each file accounts for 518 bytes of space in VDP RAM.

As we said, you can excerpt the needed stuff from today's Sidebar to allow a program of your own to CALL FILES. Just put the number of files you want between the single quotes where our Sidebar shows '9' in the tokenized data following PABDT. For example, if the desired number is five, you'd change that tokenized line to look like this:

```
BYTE >B7,>C8,>01,'5',>B6
```

Putting the 5 between single quotes like this will cause the Assembler to insert >35 at that place in memory, and that's the desired value for the CALL FILES routine. For your own application, of course, that number can be anything from 1 through 9, just as in our little demo program. You must, however, take some care in the rest of your program not to write into the area of VDP reserved for the five files. CALL FILES will leave the highest available address at >8370, so you can compare to that word within your program to set a limit on your own "writes" to VDP. In many of our own programs, we limit our VDP writes to >37D7 and below, which is the "normal" number found in >8370. If we've used CALL FILES, we'll have to be more careful about this.

Finally, you should know that this little trick only works in limited cases. It won't work correctly for CALL SCREEN, for example. We don't know exactly why, but take our word for it, because we've tried that and got an endless repeating INCORRECT STATEMENT for our trouble. That's not too surprising, since one can't expect a call from Basic to work like one from the disk controller's ROM. This trick has been tested with only the TI disk controller, so we've no idea whether it will work correctly with CorComp or Myarc disk controllers. Some brave souls among our readers may want to try this with those "third-party" disk controllers, but we guarantee nothing in those cases. (Attempt at your own risk.) Should you need our help in applying this little trick, please write or phone, as we're always glad to help. That address and phone number are:

Bruce Harrison
5705 40th Place
Hyattsville MD 20781
U.S.A.
Phone (301) 277-3467

```
* SIDEBAR 44
* A COMPLETE PROGRAM THAT SOLVES
* THE OLD CALL FILES PROBLEM
*
* CALLS FILES (1-9) FROM E/A OPT 3
* 17 APR 94
* by Bruce Harrison
* Public Domain
*
       REF  VMBW,KSCAN,VMBR,VSBW,DSRLNK  REF'D UTILITIES
       DEF  START        DEFINE ENTRY POINT
*
* REQUIRED EQUATES
*
PAB    EQU  >1000        PERIPHERAL ACCESS BLOCK VDP ADDRESS
CALPNT EQU  >832C        POINTER FOR CALL FILES
PABPNT EQU  >8356        POINTER FOR DSRLNK
GPLWS  EQU  >83E0        GPL WORKSPACE
*
* MAIN CODE SECTION
*
START  LWPI WS           LOAD OUR WORKSPACE
```

```
       CLR  @>8374       CLEAR KEY-UNIT
       LI   R0,34        POINT AT ROW 2
       LI   R1,PROMPT    PROMPT FOR USER CHOICE - NUMBER OF FILES
       BL   @DISSTR      DISPLAY THE PROMPT
GTNF   BLWP @KSCAN       SCAN THE KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN STOP
       CB   @>837C,@ANYKEY HAS A KEY BEEN PRESSED
       JNE  GTNF         IF NOT, RE-SCAN
       MOV  @>8374,R8    MOVE KEY AS WORD INTO R8
       CI   R8,'1'       COMPARE TO "1"
       JLT  GTNF         IF LESS, REPEAT SCAN
       CI   R8,'9'       COMPARE TO "9"
       JGT  GTNF         IF GREATER, REPEAT SCAN
       LI   R0,3*32+14   ROW 4, COL 15
       MOVB @>8375,R1    GET THE KEY'S VALUE INTO R1
       BLWP @VSBW        WRITE THE CHOICE TO SCREEN
       MOVB R1,@PABDT+9  AND PLACE IT IN DATA FOR THE CALL
       LI   R0,6*32+6    ROW 7, COL 7
       LI   R1,ONSTR     "OLD NUMBER"
       BL   @DISSTR      DISPLAY THAT
       LI   R0,8*32+12   ROW 9,COL 13
       BL   @NUMDIS      SHOW NUMBER FROM >8370
MKCALL LI   R0,PAB       POINT AT PAB LOCATION IN VDP
       LI   R1,PABDT     AND DATA FOR PAB
       LI   R2,ANYKEY-PABDT INCLUDE THE TOKENIZED STUFF
       BLWP @VMBW        WRITE THAT TO VDP
       MOV  R0,@PABPNT   MOVE R0 TO >8356
       MOV  R0,@CALPNT   AND TO >832C
       BLWP @DSRLNK      USE DSR LINK
       DATA >A           WITH DATA FOR A "CALL" SERVICE
       MOV  @>8350,@>835E  CHECK FOR ERROR
       JEQ  NEWNUM       IF NONE, JUMP
       BL   @ERRDIS      ELSE DISPLAY ERROR CODE
NEWNUM LI   R0,13*32+6   ROW 14, COL 7
       LI   R1,NNSTR     "NEW NUMBER"
       BL   @DISSTR      DISPLAY THAT
       LI   R0,15*32+12  ROW 16, COL 13
       BL   @NUMDIS      DISPLAY NUMBER FROM >8370
       LI   R0,22*32+5   ROW 23, COL 6
       LI   R1,PAK       "PRESS ANY KEY"
       BL   @DISSTR      DISPLAY THAT
SCAN   BLWP @KSCAN       SCAN KEYBOARD
       CB   @ANYKEY,@>837C  KEY STRUCK?
       JNE  SCAN         IF NONE, REPEAT
       LWPI GPLWS        LOAD GPL WORKSPACE
       B    @>6A         BRANCH TO GPL INTERPRETER
*
* SUBROUTINE SECTION
*
```

```
DISSTR MOVB *R1+,R2      GET LENGTH BYTE
       JEQ  DISX         IF ZERO, EXIT
       SRL  R2,8         RIGHT JUSTIFY
       BLWP @VMBW        WRITE TO SCREEN
DISX   RT                RETURN
ERRDIS LI   R0,12*32+12  ROW 13, COL 13
       JMP  NDIS1        THEN JUMP
NUMDIS MOV  @>8370,@>835E  GET WORD FROM >8370 TO >835E
NDIS1  CLR  @>837C       CLEAR GPL STATUS
       BLWP @GPLLNK      USE GPL LINKAGE
       DATA >2F7C        DATA FOR INTEGER TO STRING CONVERSION
       MOVB @>8361,R2    GET STRING LENGTH
       SRL  R2,8         RIGHT JUSTIFY
       JEQ  NODIS        IF ZERO, EXIT
       MOVB @>8367,R1    GET LOW BYTE OF POINTER TO STRING
       SRL  R1,8         RIGHT JUSTIFY
       AI   R1,>8300     ADD 8300 HIGH BYTE
       BLWP @VMBW        DISPLAY THE NUMBER STRING
NODIS  RT                RETURN
*
*  GENERAL PURPOSE GPL LINK
*  BY DOUG WARREN AND CRAIG MILLER
*
GR4    EQU  GPLWS+8
GR6    EQU  GPLWS+12
STKPNT EQU  >8373
LDGADD EQU  >60
XTAB27 EQU  >200E
GETSTK EQU  >166C
*
GPLLNK DATA GLNKWS
       DATA GLINK1
RTNAD  DATA XMLRTN
GXMLAD DATA >176C
       DATA >50
GLNKWS EQU  $->18
       BSS  >08
GLINK1 MOV  *R11,@GR4
       MOV  *R14+,@GR6
       MOV  @XTAB27,R12
       MOV  R9,@XTAB27
       LWPI GPLWS
       BL   *R4
       MOV  @GXMLAD,@>8302(R4)
       INCT @STKPNT
       B    @LDGADD
XMLRTN MOV  @GETSTK,R4
       BL   *R4
       LWPI GLNKWS
       MOV  R12,@XTAB27
```

```
       RTWP
*
* DATA SECTION
WS     BSS  32           WORKSPACE
PABDT  BYTE 5            CALL NAME LENGTH
       TEXT 'FILES'      NAME
       BYTE >B7,>C8,1,'9',>B6  TOKENIZED CODE FOR (9)
ANYKEY BYTE >20          NUMBER FOR KEYSTROKE CHECKING
PROMPT BYTE 27
       TEXT 'HOW MANY FILES? PRESS (1-9)'
PAK    BYTE 21
       TEXT 'PRESS ANY KEY TO EXIT'
ONSTR  BYTE 19
       TEXT 'OLD NUMBER AT >8370'
NNSTR  BYTE 19
       TEXT 'NEW NUMBER AT >8370'
       END
```

1.45. The Art Of Assembly — Part 45. Some Surprises

By Bruce Harrison

It is May 1994, and we are once again departing from our normal format in this column. Among other things, there's no Sidebar of source code this month. This is because we've just started on another new adventure with computers, and wanted to pass along some observations that may upset or even anger the loyal TI users, but we're telling you the truth as we see it, with no apologies.

1.45.1. The Purple Heart Bargain

There are a number of "thrift" stores in our area, and being on a small budget, we shop at those stores. All the goods sold are used, sold "as is", all sales final, and so on. Still, there are occasionally real gems among the rhinestones. One day while browsing in the "Purple Heart" store in nearby Bladensburg, we found a white box that contained a Radio Shack Color Computer 2. We inquired for a price, as there was no tag attached. "Twenty dollars, AS IS" was the response. This was a 64K memory model, with Radio Shack's Extended Basic built in, so it looked like a potential bargain. We've found TI consoles (bare bones) at this same store selling for as little as $5.00, and they all worked. We scooped up the CC2 (or CoCo) and raced home to try it out. The CoCo is a nice package. It includes a built-in power supply, so there's no awkward transformer on its power cord. There were no cables included except the RF cable to connect to a TV set. The modulator, too, is built right in, so no modulator needs to dangle at the end of a cord. The RF connected through a simple adapter box that we use with Atari video games.

It worked perfectly, just as we'd hoped. The Extended Basic here is of course different in many respects from that on the TI, but we were able very quickly to get a stupid test program typed in. We used a simple FOR-NEXT loop as a quick test, like this:

```
10 FOR I=1 TO 200
20 PRINT "HARRISON'S GARBAGE REPEAT NUMBER";I
30 NEXT I
```

Having put this simple test program in place, we typed RUN **ENTER**, and were amazed! This program ran so fast that the lines scrolling up the screen were a blur. It finished the 200 repeats in just a few seconds, then we could read the lines that remained on the screen. We'd never seen any version of BASIC run so fast! To see just how fast, we fired up the PC that happened to be nearby, (Tandy 1000SX) and typed the same program into its GW BASIC. Now we teamed up to start each of these two running at the same time. The CoCo finished at about the time that the PC was printing number 100. In other words, this CoCo's Extended BASIC was executing about twice as fast as the GW BASIC on the PC. We knew from earlier experiments that the PC with GW BASIC runs much faster than our trusty old TI, so we put that same program into the TI in Console BASIC. The difference is startling. TI's Console BASIC is painfully slow compared to GW BASIC, and ridiculously slow compared to the CoCo. That same program ran a bit faster in TI Extended BASIC, but was still nowhere near a match for the GW BASIC on the PC, and lagged way behind compared to the CoCo.

1.45.2. Graphics Capabilities

We called our local Radio Shack, and discovered that the manual for the CoCo is still available (about $5.00), so we ordered that. While waiting, we checked the shelves at a local used book store, and found a book called Color Computer Graphics. This little gem contained a very good description of the graphics capabilities provided by the CoCo, with sample programs we could type in and run. Unlike the TI, the CoCo offers excellent access to its graphics capabilities directly from its Extended BASIC. Yes, there's even a Bitmap Mode (256 × 192 pixels) that's accessible from the CoCo's XB. It will also permit drawing lines, boxes, filled in boxes, circles and ellipses just with simple statements from XB! Even without the regular manual, we found that we could do some pretty amazing things on this little computer. There are some twelve modes of operation for its VDP, and all of those can be used without resorting to any Assembly tricks.

1.45.3. Limitations

There are some limitations on the CoCo that seem strange to the TI user. There are no lower case characters available, for example, but there are TWO sets of upper case characters, one set in "normal" colors, the other in "inverted" colors. Also, in many cases the ASCII codes are NOT what we're accustomed to! There is no underline character, for example. Its place is taken by a left-pointing arrow symbol. WHY? These character patterns are all in ROM, and there's no way to re-define them, so this weird arrangement of the ASCII codes is just a fact of life on the CoCo. This can be overcome for programs that execute within the CoCo, but could be a real problem if one's bringing in stuff from the serial port, for example, since the ASCII codes coming in might not provide the expected characters on the screen. There also is no capability for using the standard characters in the graphics modes! There are in-between modes of operation called "semigraphic" modes in which one can mix graphics with alphanumerics, but the alphanumerics are not available once you've put the CoCo into a "true graphics" mode.

One other "not-so-good" is the method of editing program lines in the CoCo's Extended BASIC. It's based on a method borrowed from the old Tandy TRS-80 BASIC, and is very cumbersome compared to the editing capability on the TI's BASICs. Neither TI nor CoCo can hold a candle to the editing capability of the GW BASIC on the PCs, but the CoCo is far below the TI in editing BASIC program lines. I'd tell you just how difficult it is, but you wouldn't believe it could be so bad, and it would take pages to describe.

1.45.4. Time's A Wastin'

We said the CoCo is fast, and the TI, by comparison, is slow. We got out a digital watch and timed the things. First TI Console Basic, which was the slowest, then on up. Each time is expressed in Minutes and Seconds:

TI Console BASIC	1:22
TI Extended BASIC	0:38
Compiled TI XB	0:25
Tandy 1400 FD (GW Basic)	0:19
Tandy 1000SX (GW BASIC)	0:18
CoCo Extended BASIC	0:07

Going from TI Console BASIC to the CoCo Extended Basic is nearly a 12 to 1 speed change! That Tandy 1400 FD in the list is the laptop PC that we use to write these columns. The 1000SX was once the "Top of the Line" Tandy PC, back when the Intel 8088 was the "standard" PC microprocessor. The 1400 FD also uses the Intel 8088. Both are beaten handily by the CoCo, which uses the Motorola 6809 microprocessor chip. The Extended BASIC used in the CoCo is by Microsoft, and is sort of a cross between the Microsoft BASIC used in PCs and the old Tandy TRS-80 BASIC.

Now that we've offended most of our readers by giving such a glowing account of the Radio Shack Color Computer 2, let's back off just a little for some fairness. As yet, we don't even have a cassette cable to save our CoCo programs, much less any way to use conveniences like disk drives. Also, we don't have any way to check the computer with Assembly language stuff. We don't know what internal clock speeds are used in the CoCo, but we're sure it's not in the tens of Megahertz. Probably the single biggest factor in the speed of the CoCo as compared to the TI-99/4A is the lack of that GPL Interpretation step. Thus there's no need for GROMs, and the BASIC interpreter is contained in ROM that's within the main address space. There's also no such thing as VDP RAM in the CoCo. The screen table is simply a piece of the regular RAM reserved for the screen images. This concept makes things happen faster, but at the expense of using memory for screen images that could otherwise be used for program space.

While we're on the subject of limitations, we should mention that there are only 9 colors (including black) on the CoCo, and that in the highest resolution graphics mode (256 by 192 pixels) there are two sets of two colors each, so the graphics are drawn only in black on green (ugly) or black on gray (better). In lower resolution graphics modes, there are two sets of four colors to choose from, but not all the choices are useful. We're spoiled by the ability on the TI to use 16 colors even in the Bitmap Mode at 256 by 192 pixels.

We'll keep experimenting with the CoCo, and will have more to say about it later. Have no fear, we'll not give up our trusty old TI, nor our PCs. We'll continue making those little Assembly programs that some in our "community" find useful. We'll also keep on probing into the inner workings of our TI, and telling you what we've found lurking there.

We'll be presenting some new Assembly programs at the Lima Faire. Of course by the time you read this, it'll be almost time for the 1995 Lima Faire. Sometimes it's a real pain to be writing so far ahead of our publication dates. The only real advantage is that, whenever we want, we can take a month or two (would you believe ten?) off from writing without missing a deadline.

Next month we'll go back into the realm of TI Bitmap Mode. Among other things, we'll pass along a nifty and quick way of drawing straight lines from anywhere to anywhere on the TI Bitmap screen. We'll show an algorithm that we "borrowed" from a book on PC Assembly, modified to work on the TI, making optimal straight lines and making them very quickly. See you then!

1.46. The Art Of Assembly — Part 46. Drawing A Straight Line

By Bruce Harrison

While we were developing our Drawing program, we found the need for some way to make a straight line between any two places on the Bitmap screen. That's not so easy as it might seem, especially if one wants the drawing to happen very quickly. One could use any of a host of methods, including those using sines and cosines of angles, but the burden in those methods is that there are numerous floating point calculations required, and thus the operation is slow, even when Assembly is used.

It's been a long time since High School Trigonometry, and much of that sine-cosine stuff has been lost in foggy old memories. In such cases we try to follow a rule we heard, attributed to Albert Einstein, to wit: "Never bother to memorize anything that you can look up in a book." We keep lots of books handy, and now and then find something really valuable in them. One of these is a book on PC Assembly language, which contains some interesting ideas for graphics programming. (*Assembly Language Primer for the IBM PC & XT*, by Robert Lafore, Copyright 1984 by The Waite Group, a Plume/Waite book, published by NAL Penguin, Inc., New York, NY and Scarborough, Ontario.) We pulled that book out of the mess here in our computer chamber, and looked at the graphics chapter. Sure enough, this had the answer to our need.

A very clever guy named Bresenham devised an algorithm for making straight lines on a computer screen. This algorithm takes full advantage of the discrete nature of the "pixel" structure, and uses only very simple math (add, subtract, compare) to perform all its steps. Thus it works very quickly to draw the "ideal" straight line on the screen.

1.46.1. Simplicity Itself

We enter the algorithm with four numbers, these being the X and Y coordinates of the start point, and the X and Y of the ending point. These four numbers are in "pixel" coordinates, of course, so that on the TI the X numbers range from 0 through 255, and Y from 0 through 191. Our drawing program doesn't use all of that range, but the concept is the same. The algorithm calculates where each point in the line should be drawn, pixel-by-pixel. There was some adaptation required, since the version in the book was for PC Assembly language. We found that the book version used the SI and DI registers to store some of the variables, and of course the TI doesn't have such registers. We made up for that by setting aside two words in our DATA section called ESSEYE (for SI) and DEEEYE (for DI). Also, since our drawing program already had a PLOT subroutine to place the pixels on-screen, we simply BL to that instead of using the method for plotting the points as given by the book.

1.46.2. Today's Sidebar Shows. . .

The code in today's Sidebar is not complete, but merely a fragment that you could surround with your own code. As shown, it starts with the TI already in Bitmap Mode. In parts 42 and 43 of this series, you'll find the code to get in and out of Bitmap. The code shown uses the position of a sprite to supply the coordinates for the start and end of the line. You could substitute some other method of setting these coordinates, and would not necessarily need to add the "offsets" that we show in our code. Those were necessary because the crossing of the two lines in the "+" that we use as a cursor is not at the reported sprite position.

In short, modify to your heart's content, so long as all the required variables get supplied to te algorithm. Those variables are:

X1	= start horizontal position
Y1	= start vertical position
X2	= ending horizontal position
Y2	= ending vertical position
DELX	= absolute value of X2-X1
DELY	= absolute value of Y2-Y1
ESSEYE	= sign of DELY (+1 or -1)
DEEEYE	= sign of DELX (+1 or -1)
HALFX	= half of DELX
HALFY	= half of DELY

In the Sidebar, you'll see that we've used registers R3, R4, R5, and R6 to hold these numbers while math was being performed, but put them all into the variables before the line starts being drawn. We've then put X1 and Y1 into R7 and R8, since those are used by our PLOT subroutine as the X and Y positions for plotting a point.

1.46.3. The Hard Part

The real work of the algorithm starts at label STALG, where we compare @DELX,@DELY. If DELX is lower than DELY, then we have a "steep" line to draw (above 45 degrees), else we have an "easy" line. The main difference between LEASY and STEEP is which variable determines how many points must be plotted. For a steep line, we need to plot DELY points, while for an easy one we must plot DELX points. In other words, the algorithm plots the required number of points, that being the number in DELX or DELY, whichever is the greater number. As we enter the algorithm, R3 still contains DELY, and R4 still contains DELX, so we've taken a shortcut at labels LEASY and STEEP by simply shifting right by one bit to make the appropriate HALF value.

In our implementation, we've used R10 as a temporary storage, R12 to count the number of pixels we'll plot, and R9 for the color of the line to be drawn. Thus for an "easy" slope, we put DELX in R12 to count points, while for steep lines, we put DELY in R12. The key to this whole process is the introduction of what Bresenham calls an "error term". This is a variable that tracks how far the current plotted point strays from what would be the ideal position. In our implementation, we cleared R10 at the outset, so R10 serves as the "error term" variable.

1.46.4. Step-By-Step

Let's just for the moment assume that we're going to make a line starting at 10,10 and going eight pixels to the right horizontally and six pixels upwards vertically from that point. This is a slope of the "easy" variety, so we'll start our step by step at label LEASY in the Sidebar. Since we're going up, ESSEYE will be -1, while DEEEYE will be +1, going to the right.

Our implementation is slightly complicated by the fact that we can use this same part of the code to either draw a line or erase one, but we'll assume that we're not erasing, so ERSFLG will be zero. At label LEASY, R4 still contains what we stored at DELX, so we shift that right one bit, cutting its value in half, then move that value to HALFX. Now DELY contains 6, DELX contains 8, and HALFX contains 4. We'll move DELX into R12, so we'll plot 9 points total for this line (including the start point and end point). At LDOT1, we find ERSFLG is zero, so we will MOVB @LINCLR,R9. Next we'll BL @PLOT to put a pixel on-screen at 10,10, then jump over label LEASE to LDOT2. Now we'll move to the next dot-column by adding what's at DEEEYE to R7.

Now the trick of Bresenham's algorithm. We add DELY to R10, so R10 contains 6. We compare that to HALFX, which contains 4. If R10 is greater than HALFX, (it is this time) then we subtract DELX from the error term, and add ESSEYE (-1) to R8, so our next pixel will be plotted at 9,11. Our error term in R10 is now at -2.

After plotting that second pixel, we again add 6 to R10, so it's now equal to 4. Since that's not greater than HALFX, we don't change R8, and we don't subtract anything from the error term. Thus our next point gets plotted at 9,12.

This gets tedious, doesn't it! For those who want to see the rest of this process played out, there is an Extended Basic program called BRESH which is listed in the Sidebar. This program will make a line of "pixels" using cursors for each dot, and will report at the top of the screen the variables as they change. Just press a key to make each "dot". To make it look better on the screen, we started the line at 24,10, but otherwise it's just as we described above.

For those who want to experiment beyond the one line, there's a second XB program called BRESH4, which allows you to enter your own Deltas, and then makes the line for you, starting at 24,1. Notice here that the algorithm doesn't get confused even if either Delta is zero, or even if both are zero. The limits are 27 for DELX and 23 for DELY, so we won't try doing an illegal DISPLAY AT. When BRESH4 finishes a line, it will wait for a keypress. Pressing **R** will take you back to do another line, while any other key will exit the program. You can of course do the same kind of experiments with the BRESH program, changing the parameters to see what happens, except that BRESH is only set up to handle "easy" lines, not steep ones. The variables are named to coincide with the way the Assembly version works, so that it'll be easier to follow.

1.46.5. Math Whiz Needed

We've explained what Bresenham's algorithm is, what it does, and how it does what it does, but that doesn't mean we understand why it works. The book that we took it from says the error term ". . . is related to the difference between where the pixel should go, if it could be drawn right on the line, and where it must go, since it can only occupy integer pixel locations." Yes, that's fine as far as it goes, but it still doesn't say why adding and subtracting and comparing in this way accomplishes the desired goal. We don't really know either. Perhaps somebody in our worldwide readership will be able to explain this to the rest of us! For the time being, we're very happy that it does work, producing a near-ideal straight line with discrete pixels. We've passed it along to you in the Sidebar, so others may get some use from it, and that's the best we can do.

That's all for today. Next month's topic is undecided at this point, so you'll just have to wait and see what's in next month's column.

```
* SIDEBAR 46
* A SIMPLE AND FAST METHOD
* FOR DRAWING PIXEL LINES
* BETWEEN TWO POINTS
* THIS IS A FRAGMENT FROM OUR
* DRAWING PROGRAM, NOT COMPLETE CODE
*
* AT THE ENTRY POINT, THE USER HAS PLACED
* SPRITE #0, A + SIGN ACTING AS THE DRAWING
* CURSOR, AT THE SPOT WHERE THE LINE IS TO START
* THE KEY HAS BEEN PRESSED TO INDICATE THAT,
* SO THE CODE BELOW STASHES AWAY THE COORDINATES
* AFTER ADDING OFFSETS SO THE POINT STORED IS
* THE CENTER OF THE + CHARACTER
*
       LI   R0,>3800     POINT AT START OF SPRITE POSITION TABLE
       BLWP @VSBR        READ Y OF SPRITE 0
       SRL  R1,8         RIGHT-JUSTIFY
       AI   R1,5         ADD OFFSET FOR CENTER OF +
       MOV  R1,@Y1       Y1=START Y POSITION
       INC  R0           POINT TO NEXT VDP BYTE
       BLWP @VSBR        READ X OF SPRITE 0
```

```
       SRL  R1,8          RIGHT-JUSTIFY
       AI   R1,3          ADD OFFSET FOR CENTER OF +
       MOV  R1,@X1        X1=START X POSITION
*
* BETWEEN HERE AND THE CODE BELOW,
* THE "CURSOR" SPRITE HAS BEEN MOVED
* BY THE USER TO THE DESIRED END-POINT
* FOR THE LINE SEGMENT, AND A KEY HAS BEEN
* PRESSED TO INDICATE THIS IS THE ENDPOINT
* THE CODE BELOW, THEN, STARTS BY CAPTURING
* THE END-POINT COORDINATES
*
       LI   R0,>3800      POINT AT Y OF SPRITE 0
       BLWP @VSBR         READ THAT
       SRL  R1,8          RIGHT JUSTIFY
       AI   R1,5          ADD OFFSET
       MOV  R1,@Y2        Y2=END Y POSITION
       MOV  R1,R3         R3 HAS Y2
       INC  R0            POINT AHEAD TO NEXT VDP BYTE
       BLWP @VSBR         READ X OF SPRITE 0
       SRL  R1,8          RIGHT JUSTIFY
       AI   R1,3          ADD OFFSET
       MOV  R1,@X2        X2=END X POSITION
       MOV  R1,R4         R4 HAS X2
       S    @Y1,R3        R3 HAS DELTA Y
       JLT  LD40          IF BELOW ZERO, JUMP
       LI   R5,1          ELSE R5=1
       JMP  LD41          THEN JUMP
LD40   LI   R5,-1         R5 = -1
LD41   ABS  R3            R3 HAS ABS. VAL. DELTA Y
       MOV  R5,@ESSEYE    SAVE R5 TO ESSEYE
       MOV  R3,@DELY      SAVE R3 AT DELY
*
* AT THIS POINT, DELY HAS THE DIFFERENCE IN Y POSITIONS
* ESSEYE HAS THE SIGN (+1 OR -1) OF THAT DIFFERENCE
*
       S    @X1,R4        R4 HAS DELTA X
       JLT  LD50          IF BELOW ZERO JUMP
       LI   R6,1          ELSE LOAD R6 WITH 1
       JMP  LD51          THEN SKIP AHEAD
LD50   LI   R6,-1         LOAD R6 WITH -1
LD51   ABS  R4            R4 HAS ABS. VAL. DELTA X
       MOV  R6,@DEEEYE    SAVE R6 AT DEEEYE
       MOV  R4,@DELX      SAVE R4 AT DELX
*
* AT THIS POINT, DELX HAS THE DIFFERENCE IN X POSITIONS
* DEEEYE HAS THE SIGN (+1 OR -1) OF THAT DIFFERENCE
*
       MOV  @X1,R7        PUT START X POINT IN R7 (DOT-COLUMN)
       MOV  @Y1,R8        PUT START Y POINT IN R8 (DOT-ROW)
```

```
       CLR  R9           CLEAR R9 (USED FOR COLOR OF LINE)
       CLR  R10          AND R10
STALG  C    @DELX,@DELY  COMPARE DELX,DELY
       JL   LSTEEP       IF DELX LOW, JUMP TO STEEP
LEASY  SRL  R4,1         ELSE SLOPE <45 , CUT R4 IN HALF
       MOV  R4,@HALFX    SAVE AT HALFX
       MOV  @DELX,R12    MOVE DELTA X TO R12
LDOT1  MOV  @ERSFLG,R3   CHECK FOR ERASE STATUS
       JNE  LEASE        IF ERASE, JUMP
       MOVB @LINCLR,R9   PUT LINE DRAW COLOR IN R9
       BL   @PLOT        PLOT A POINT AT COORDINATES IN R8,R7
       JMP  LDOT2        THEN JUMP AHEAD
LEASE  BL   @UNPLOT      ERASE A POINT AT R8,R7 LOCATION
LDOT2  A    @DEEEYE,R7   ADD + OR - 1 TO R7
       A    @DELY,R10    ADD DELTA Y TO R10
       C    R10,@HALFX   COMPARE TO HALF OF X
       JLT  LDOT3        IF LESS, JUMP
       JEQ  LDOT3        OR IF EQUAL, JUMP
       S    @DELX,R10    SUBTRACT DELTA X FROM R10
       A    @ESSEYE,R8   ADD PLUS OR MINUS 1 TO R8
LDOT3  DEC  R12          DECREMENT DELTA X IN R12
       JGT  LDOT1        IF POSITIVE, REPEAT
       JEQ  LDOT1        OR IF EQUAL, REPEAT
       JMP  LDREX        ELSE LINE FINISHED
LSTEEP SRA  R3,1         CUT R3 (DELTA Y) IN HALF
       MOV  R3,@HALFY    SAVE AT HALFY
       MOV  @DELY,R12    GET DELY INTO R12
LDOT4  MOV  @ERSFLG,R3   CHECK FOR ERASE
       JNE  LSTPE        IF ERASE, JUMP AHEAD
       MOVB @LINCLR,R9   ELSE PUT LINE DRAW COLOR IN R9
       BL   @PLOT        PLOT ONE POINT
       JMP  LDOT5        THEN JUMP AHEAD
LSTPE  BL   @UNPLOT      ELSE "UNPLOT" TO ERASE A POINT
LDOT5  A    @ESSEYE,R8   ADD + OR - 1 TO R8
       A    @DELX,R10    ADD DELTA X TO R10
       C    R10,@HALFY   COMPARE TO HALF OF DELTA Y
       JLT  LDOT6        IF LESS, JUMP AHEAD
       JEQ  LDOT6        IF EQUAL, JUMP AHEAD
       S    @DELY,R10    SUTRACT DELTA Y FROM R10
       A    @DEEEYE,R7   ADD DEEEYE TO R7
LDOT6  DEC  R12          DEC COUNT OF POINTS
       JGT  LDOT4        IF POSITIVE, REPEAT
       JEQ  LDOT4        OR IF EQUAL, REPEAT
LDREX  (END OF LINE DRAWING PROCESS)
       B    @KJSCAN      RETURN TO "NORMAL" DRAWING MODE
*
* SUBROUTINES PLOT AND UNPLOT - TO DRAW OR ERASE
* SELECTED PIXEL POSITION
*
* FOLLOWING WRITES ONE PIXEL TO SCREEN AT LOCATION POINTED BY
```

```
* R8 (DOT ROW) AND R7 (DOT COLUMN)
*
PLOT   MOV  R7,R3        MOVE DOT COLUMN TO R3
       MOV  R8,R4        AND DOT ROW TO R4
       MOV  R4,R5        DOT ROW ALSO IN R5
       ANDI R5,7         R5 HAS DOT ROW MODULO 8
       SZC  R5,R4        SO DOES R4
       SLA  R4,5         MULTIPLY R4 BY 32
       A    R5,R4        ADD R5, SO R4 HAS DR MOD. 8 * 32 + DR MOD 8
       MOV  R3,R0        MOVE DOT COL TO R0
       ANDI R0,>FFF8     R0 HAS DC - DC MOD 8
       S    R0,R3        R3 HAS DC MOD 8
       A    R4,R0        ADD R4
       SWPB R0           SWAP BYTES
       MOVB R0,@>8C02    WRITE LOW ADDRESS BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH ADDRESS BYTE
       NOP               WASTE TIME
       MOVB @>8800,R1    READ THE BYTE
PLOTF  SOCB @M(R3),R1    OVERLAY MASK FROM TABLE M
PLOTF0 ORI  R0,>4000     SET THE 4000 BIT IN R0
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE LOW BYTE OF ADDRESS
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH BYTE OF ADDRESS
       NOP               WASTE TIME
       MOVB R1,@>8C00    WRITE MODIFIED BYTE BACK TO VDP
       MOV  R9,R9        IS COLOR TO BE SET?
       JEQ  PLOTX        IF NOT, JUMP AHEAD
       ANDI R0,>3FFF     STRIP OFF "4" FROM R0
       AI   R0,>2000     ADD >2000 TO POINT AT COLOR TABLE ENTRY
       BLWP @VSBR        READ THAT BYTE INTO R1
       MOVB R1,R2        MOVE THE BYTE TO R2
       ANDI R2,>F000     STRIP ALL BUT LEFT NYBBLE
       CB   R2,R9        COMPARE TO LEFT BYTE R9
       JEQ  PLOTX        IF EQUAL, COLOR ALREADY SET
       ANDI R1,>0F00     ELSE STRIP OFF LEFT NYBBLE R1
       AB   R9,R1        REPLACE WITH LEFT NYBBLE R9
       BLWP @VSBW        THEN WRITE COLOR BYTE BACK
PLOTX  RT                RETURN
*
* FOLLOWING ERASES ONE PIXEL AT DOT ROW IN R8,
* DOT-COLUMN IN R7
*
UNPLOT MOV  R7,R3        MOVE DOT COLUMN TO R3
       MOV  R8,R4        AND DOT ROW TO R4
       MOV  R4,R5        DOT ROW ALSO IN R5
       ANDI R5,7         R5 HAS DOT ROW MODULO 8
       SZC  R5,R4        SO DOES R4
       SLA  R4,5         MULTIPLY R4 BY 32
```

```
       A    R5,R4        ADD R5, SO R4 HAS DR MOD. 8 * 32 + DR MOD 8
       MOV  R3,R0        MOVE DOT COL TO R0
       ANDI R0,>FFF8     R0 HAS DC - DC MOD 8
       S    R0,R3        R3 HAS DC MOD 8
       A    R4,R0        ADD R4
       SWPB R0           SWAP BYTES
       MOVB R0,@>8C02    WRITE LOW ADDRESS BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH ADDRESS BYTE
       NOP               WASTE TIME
       MOVB @>8800,R1    READ THE BYTE
       INV  R1           INVERT ALL BITS IN R1
       SOCB @M(R3),R1    OVERLAY MASK FROM TABLE M
       INV  R1           RE-INVERT WITH ONE BIT CHANGED
       ORI  R0,>4000     SET THE 4000 BIT IN R0
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE LOW BYTE OF ADDRESS
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH BYTE OF ADDRESS
       NOP               WASTE TIME
       MOVB R1,@>8C00    WRITE MODIFIED BYTE BACK TO VDP
       RT
*
* DATA SECTION
*
M      DATA >8040,>2010,>0804,>0201  MASK DATA
X1     DATA 0            STORAGE FOR START X
X2     DATA 0            STORAGE FOR END X
Y1     DATA 0            STORAGE FOR START Y
Y2     DATA 0            STORAGE FOR END Y
DELX   DATA 0            STORAGE FOR DELTA X
DELY   DATA 0            STORAGE FOR DELTA Y
HALFX  DATA 0            HALF VALUE DELTA X
HALFY  DATA 0            HALF VALUE DELTA Y
DEEEYE DATA 0            STORES SIGN DELTA X
ESSEYE DATA 0            STORES SIGN DELTA Y
ERSFLG DATA 0            ERASE MODE OFF - NON ZERO MAKES ERASE ON
LINCLR BYTE >10          DEFAULT COLOR - BLACK
*
* BELOW IS A TEST PROGRAM (BRESH)
* WRITTEN IN EXTENDED BASIC, TO SHOW
* HOW BRESENHAM'S ALGORITHM WOULD WORK
* FOR A LINE OF CURSORS FROM 24,10,
* GOING EIGHT UNITS RIGHT AND 6 UPWARD
* THE LISTING IS IN 28 COLUMNS
* R10 IS THE "ERROR TERM"
*
*
10 CALL CLEAR
20 DELX=8 :: DELY=6
```

```
30 HALFX=4 :: HALFY=3
40 R12=8 :: R10=0
50 DEEEYE=+1 :: ESSEYE=-1
60 R8=24 :: R7=10
70 DISPLAY AT(1,1):"R12=";R1
2:"R10=";R10:"R8 (ROW)=";R8:
"R7 (COL)=";R7: : : :
80 DISPLAY AT(R8,R7):CHR$(30
)
90 R7=R7+DEEEYE
100 R10=R10+DELY
110 IF R12=0 THEN DISPLAY AT
(6,1):"FINISHED" :: GOTO 130
120 DISPLAY AT(5,1):"R10+DEL
Y=";R10:"HALFX=";HALFX
130 CALL KEY(0,K,S)
140 IF S<1 THEN 130
150 IF R10<=HALFX THEN 180
160 R10=R10-DELX
170 R8=R8+ESSEYE
180 R12=R12-1
190 IF R12>=0 THEN 70
*
* FOLLOWING IS AN XB PROGRAM (BRESH4)
* THAT EMULATES THE ACTION OF
* THE ASSEMBLY VERSION
* SO YOU CAN SEE MORE CLEARLY
* WHAT HAPPENS
*
10 CALL CLEAR
20 INPUT "DELTA X (0 - 27) "
:DELX :: IF DELX<0 OR DELX>2
7 THEN 20
30 INPUT "DELTA Y (0 - 23) "
:DELY :: IF DELY<0 OR DELY>2
3 THEN 30
40 CALL CLEAR
50 HALFX=INT(DELX/2):: HALFY
=INT(DELY/2):: R10=0 :: DEEE
YE=+1 :: ESSEYE=-1 :: R8=24
:: R7=1 :: IF DELX<DELY THEN
 110
60 R12=DELX
70 DISPLAY AT(R8,R7):CHR$(30
)
80 IF DELX<DELY THEN 110
90 R7=R7+DEEEYE :: R10=R10+D
ELY :: IF R10<=HALFX THEN 10
0 ELSE R10=R10-DELX :: R8=R8
+ESSEYE
```

```
100 R12=R12-1 :: IF R12>=0 T
HEN 70 ELSE 150
110 R12=DELY
120 DISPLAY AT(R8,R7):CHR$(3
0)
130 R8=R8+ESSEYE :: R10=R10+
DELX :: IF R10<=HALFY THEN 1
40 ELSE R10=R10-DELY :: R7=R
7+DEEEYE
140 R12=R12-1 :: IF R12>=0 T
HEN 120
150 CALL KEY(0,K,S):: IF S<1
 THEN 150 ELSE IF K=ASC("R")
THEN 10
```

1.47. The Art Of Assembly — Part 47. CALL FILES From XB

By Bruce Harrison

We've talked about "CALL" operations before in this column, but today we're showing a real working solution to a real problem, not just playing around. Many of us who've used TI Extended Basic have wished that we could execute a CALL FILES from within a running program. TI did not allow that to be done, and for good reasons, which we'll get to shortly. There are two reasons why one might want to use CALL FILES within a program: First, one might have a program that needs to have more than three files open at some time during its execution. Second, one might have a program that uses so much string space (STACK memory, a.k.a. VDP RAM) that it needs to have the number of files set to 1 or 2 to allow more room for its string operations.

1.47.1. TI Had Reasons

TI Extended Basic uses VDP RAM for a number of purposes while a program is running. A portion is used for the screen display and character definitions, sprite tables, color tables and such. This stuff is stored in the lower parts of VDP RAM at addresses below >1000. Another part, in the higher addresses, is used for file operations. That normally extends from >37D8 through >3FFF, which is the end of VDP RAM. The part in between >1000 and >37D7 is used for string variables, the symbol tables for string and numeric variables, and CALLs. When a program starts running, XB performs a prescan operation, and sets up the symbol tables for variables and calls starting at >37D7 and working downward in VDP memory until all the variables and calls have been assigned places in these tables. This is the case for a "normal" situation where the default number of files (3) is allowed for. If a CALL FILES with an argument other than 3 had been performed before the program started, then the symbol tables would be formed at some address other than >37D7. Here is a partial "mapping" for VDP RAM in the default case:

VDP ADDRESSES	*CONTENT*
0000 - 1000	Screen, characters, sprites, etc.
1000 - xxxx	Reserved for strings, etc.
xxxx - 37D7	Symbol tables
37D7 - 3FFF	Reserved for file operations

Here, we've used xxxx to indicate the end of the space used for the symbol tables. When string variables are used in the program, their "content" gets placed into the space beginning at xxxx, working downward toward the limit at about >1000.

Now let's suppose that while this program is running we execute a CALL FILES(4). That action would wipe out 518 bytes of VDP before >37D7, thus destroying all or part of the symbol tables that the program needs to find its variables. After that, the program simply could not run. It would stop with SYNTAX ERROR messages when it failed to find a variable in the symbol table.

1.47.2. The Dilemma

There, in a nutshell, is the dilemma we face. If we devise an Assembly routine to get around TI's prohibition of CALL FILES within a program, we still wind up with a program that will not run under XB after the CALL LINK to our Assembly routine. How can we trick XB into doing what we want it to do? The solution is another of those things that TI forgot to tell us.

Those familiar with TI Extended Basic know that one can use the RUN command as a statement in a program. Most commonly this is used with a file name, as in RUN "DSK1.NEWPROG", to load and run another program. TI told us this would work, and it does. TI also told us that we can start the running of a program that's currently in memory at a specific line number, as in RUN 120 from Command mode. What the books didn't tell us is that the RUN command can be used in any of its forms within the program. We found that out only by trying it! We half expected that when a program contained a line like 130 RUN 150, we'd get a SYNTAX ERROR IN 130 message. That didn't happen! After a few carefully controlled experiments, we were able to convince ourselves that such a statement in an XB program does just what we'd hoped it would do. That is, it starts from scratch, performs its prescan operation, and then starts the program at the line number given in the statement. You can prove this for yourself by running the following short test program in XB:

```
100 A=15
110 RUN 130
120 A=25
130 PRINT A
```

When this program is run, it will print the value of A after executing the RUN 130 command, so you'll see 0 printed as the value of A. That's so because the prescan process that happens when line 110 executes sets all numeric variables to zero, thus eliminating the value 15 that was assigned to A in line 100. One can then prove another small point by typing in RUN 120 **ENTER** from Command mode. This time the value 25 will get printed, as you'd expect. This simply shows that in the first run, line 120 never got executed at all. Thus the solution to our dilemma was to combine the CALL FILES Assembly routine with a statement that would RUN at a specific line number after the CALL FILES had executed.

Before we go any farther, here's a warning for all those who use other versions of Extended Basic or third-party disk controllers. WE DO NOT HAVE ANY XBs OTHER THAN TI EXTENDED BASIC, NOR DO WE HAVE ANY MYARC OR CORCOMP CONTROLLERS! All of what we're presenting in today's column has been tested and worked "as advertised" in TI EXTENDED BASIC, and using a TI Disk Controller. None of it has been tested with MYARC XB, SUPER EXTENDED BASIC, MYARC or CorComp disk controllers, or any other variant. Of course it might work with those, but please don't complain to us if it doesn't.

1.47.3. The Sidebar

Today's Sidebar gives the complete source code for our utility routine CALLF, plus the listing of a demo program that will show that it works. We have distributed a Public Domain disk called CALLFILES which contains this source file, an object file, demo programs, and complete instructions for using the utility with your own XB programs. That's available through the Lima User Group for a nominal fee. Since it's Public Domain, you or your user group can make copies for anyone who needs it. The source code as shown in the Sidebar is designed solely for use as a "CALL LINK" from XB. For those doing pure Assembly programs, number 44 in this series contains all you should need to CALL FILES in those programs.

The source code starts in the normal fashion, by setting the Workspace Pointer to our own workspace. It then clears our R0 in preparation for getting the parameters from the CALL LINK. Both parameters get converted into integers by XMLLNK vector. The first parameter, which is the number of files to be permitted, is kept in R8, while the second, which is a line number, gets placed directly into the two bytes of data at FKL1+3 and FKL1+4

We found through experimentation that the FILES routine itself writes things into areas in RAM Pad, and that this can mess up the return to XB if RAM Pad is not returned to its original state. Therefore, we took the "brute force" approach of simply stashing all 256 bytes of RAM Pad in our own routine's memory space (at SAVPAD) for restoration later. The routine could probably be made to work by saving and restoring only a part of that >8300 area, but we didn't want to put in all the hours required to find out what was the minimum necessary part to save and restore.

Now we take the number of files parameter, still stored in R8, and compare it to the limits 1 and 9. If it's outside those limits, we take a shortcut to an error reporting process, so the user will know there's a bad value in the CALL LINK. We do that by setting our own R0 to >1E00, then simply BLWP to the ERR vector at >2034, which then generates the "BAD VALUE IN xxx" error report for the user.

Given that the parameter is in the correct range, we add the value >30 to it, so that it's correct as a "string" character to represent the number, and place the result (>31 thru >39) in our PAB data, replacing the '9' at PABDT+9. Now take a look at the data at label PABDT. You'll notice that this is missing the normal first eight bytes of a PAB data. We found, again by just trying it out, that only the name length, the name, and the parameter part in "tokenized" form are required. You'll note also that the name length is 5, so it represents only the length of the word "FILES", not the length of the entire block. The line in PABDT after the word "FILES" is in XB tokens, so that the FILES routine can use it just as if it were part of an XB command. Thus the opening parenthesis is the byte >B7. This is followed by a >C8 token, meaning that what follows is an unquoted string. The next byte is the length of that string, (always 1 for CALL FILES) and that's followed by the number in the range >31 thru >39 that we obtained from the parameter. The last byte is a >B6, which is the token for closing parenthesis.

We now write this whole block, from label PABDT to label FKL1, into VDP RAM at the PAB location >1000. Now since our R0 already points to the "name length" byte, we place R0 at >8356 for the DSR operation. However, the FILES routine itself begins its parsing of the data provided at the location pointed to by >832C, so we also place our R0 into that pointer before doing the DSRLNK operation. Note that the DSRLNK is followed by a DATA >A, not the DATA 8 used for file operations. This way, DSRLNK will find and execute the FILES routine in the disk controller's ROM.

When we get control back, the new value for the "highest available address in VDP RAM" will be at >8370, unless there was an error detected, in which case >8350 will be non-zero. If there's no error, we "capture" >8370 into our own R1, restore the contents of RAM Pad, put the new value back into >8370, and we're finished.

1.47.4. The Tricky Part

Now we're almost ready to return control of the computer to Extended Basic, but if we did that right at this point, our XB program might never be able to find any of its variables. Thus we must trick XB into starting the prescan process over from the top, and then make it re-start the program from a line beyond the line containing the CALL LINK("CALLF". . .). We do this by a very simple trick given to us by our good friend Harry Wilhelm. Look at the data at label FKL1. That starts with >82, which is the token for a double colon. This will tell XB that what follows is a continuation of the current program line. What follows the >82 is >A9, the token for RUN, followed by the token >C9, which means "the next two bytes are a program line number" Now remember that we placed the second parameter from the CALL LINK into the two bytes just after that >C9. Let's suppose the second parameter was 40. Then the tokens starting at label FKL1 will be equivalent to the XB statement ":: RUN 40". That, of course is what we want XB to do next. We trick XB into doing what's at FKL1 by simply placing the address of FKL1 at >832C, loading the workspace pointer to the GPL Workspace at >83E0, and branching back to the GPL Interpreter at >006A. XB will then dutifully execute a "RUN 40" just as if we'd typed that in Command mode and pressed **ENTER**.

1.47.5. Caution

The CALL LINK to CALLF should be done very early in the program, before any values are set into variables. That's important because the CALL LINK to CALLF will result in all variables being zero (for numerics) or null (for strings) after the CALL LINK is completed. In the DEMO XB program that's included in the Sidebar, we set a value on the variable A before the CALL LINK just to illustrate that A becomes 0 after the CALL LINK.

The last number printed by the demo program is the value of the word at >8370 after the CALL FILES. This will be changed in increments of 518 for each "file" allowed, as follows:

Files	*Number from >8370 (decimal)*
1	15331
2	14813
3 (default)	14295
4	13777
. . .	
9	11187

When a program that uses this CALL LINK ends, the files allowance will remain set to whatever the program did when it began. That condition may or may not be desirable for whatever you do next with XB. You can correct that by including a couple of lines at the end of the program like this:

```
3500 CALL LINK("CALLF",3,3510)
3510 END
```

The line numbers would of course be changed to whatever suits your particular program, but setting the number of files back to 3, which is the default, should allow any other XB program to behave as expected.

Our testing was done on our main TI system, which has the TI Disk Controller, two floppy drives, two Horizon Ramdisk cards, and a P-Gram installed. This whole thing might not work at all with Myarc or CorComp controllers present, and we'd be especially cautious about trying this on systems that contain Hard Drives of any kind. We'd like to hear from any readers who have Myarc or CorComp controllers if they try this out, either through "Reader Feedback" or directly.

For those who want to apply this to existing XB programs, you should know that it will also work correctly in those cases where the original XB program has a "prescan avoidance" feature in it. We were not sure whether that would mess up our "RUN (line number)" statement, but having tested this by adding the prescan avoidance feature (!@P-) to our demo programs from the Public Domain disk, we can assure you it works. (Be sure to include CALL LINK within the prescanned part of the program. The actual CALL LINK to CALLF can be beyond the prescanned part without causing trouble.) If needed, the parameters in the CALL LINK can be given as variables, provided those are assigned suitable values before the CALL LINK is executed.

Once again we've found a way to trick our favorite computer into doing something in spite of itself. We hope this will prove valuable to our XB programmers. Next month's topic is undecided. See you then.

1.48. The Art Of Assembly — Part 48. Another Use For CALL FILES

By Bruce Harrison

Forty-eight! This can't be! That means we've been turning out these columns for four years! We keep wondering where the trail will end, but just when it seems we've exhausted all the possible subjects, one of our readers comes up with a need that we find impossible to ignore. This time it was Mr. Ian J. Howle, of Seattle, WA. Mr. Howle needed some help with Bitmap, and in the course of our correspondence he mentioned that he couldn't find any simple way to load TI-Artist pictures. Another challenge was born.

1.48.1. Our Confession

TI-Artist is a very popular commercial product which we've never owned or used. We're told that the program itself is excellent, but the documentation leaves something to be desired. That's not a fact we can verify, but more than one friend of ours has made that assertion. Our own knowledge, limited as it is, comes from two sources: Harry Wilhelm provided us with some of the salient facts, such as file formats and sizes. We then obtained a copy of Notung's Disk of the Old West, which has some truly excellent examples of TI-Artist pictures made by our friend Ken Gilliland. Armed with this thin veneer of knowledge, we set some goals for ourselves. First, we made our Drawing and Video Titler programs capable of using TI-Artist pictures as inputs. That success meant we understood the files and their relationship to the VDP in Bitmap Mode.

1.48.2. But What If. . .

In both the Drawing and Titler cases, we brought the contents of the TI-Artist pictures into the 32K RAM through a DSRLNK and VMBR process, then put the VDP into Bitmap and displayed the picture by writing it from the 32K into the VDP again. Mr. Howle's question was how could we load TI-Artist pictures without tying up large chunks of the 32K RAM. Could we perhaps load the files directly into the correct places in VDP RAM directly from the disk, so that only the VDP RAM would ever need to store those files' contents? First, we have to realize that the TI-Artist picture may be one or two 25-sector files. The essential one is the file ending with _P. For example, on Disk of the Old West, there's a set of two files that comprise the Menu, as a TI-Artist picture. They are named MENU_P and MENU_C. The first is just the "black and white" content, while the second contains the color information. Most of the pictures on that disk don't have the _C file, so they have no color portion.

Now let's look for a moment at how we've set up VDP RAM for Bitmap operation in our Drawing and Titler programs. The mapping (all numbers in hex) looks like this:

VDP Addresses	*Use*
0000 - 17FF	Pattern descriptor table
1800 - 1FFF	Screen Image Table
2000 - 37FF	Color Table
3800 -	Sprite Attr Table, etc.

There would be a conflict if we just tried file operations with this setup of VDP RAM, because the block reserved for the DSR's use begins at >37D8, and that would be within the Bitmap's Color Table area. Thus we needed a way to move the DSR reserved block to an address above >3800. CALL FILES(1) does that, placing the reserved area for one file at >3BE4. Since we don't use sprites, this would be just fine. The program never has to deal with more than one file at a time, so CALL FILES(1) met all our criteria.

1.48.3. Today's Sidebar

This is a complete program, ready to be assembled. It uses some subroutines from our earlier work, including the SETBM and SETGM routines to get us into and out of Bitmap Mode, the CRSIN routine to accept file name input, etc. If you get *MICROpendium* on disk, you can just assemble the file called SIDEBAR48 directly to an object file, then run it under E/A Option 3 or from Funnelweb's Loader. The program occupies only 1553 bytes in high memory. It uses another block of 776 bytes at >2678 in low memory to stash graphics mode character definitions. The program starts with some "housekeeping" to set up the normal screen conditions. After the colors are set, we capture the character definitions from VDP RAM, and stash them away in low memory at >2678. This makes it easier to get back to Graphics mode when we need to.

Now, using essentially the same technique used in last month's column, we perform a CALL FILES(1). After that, the address kept at >8370 will be >3BE3. We'll use that later, but first we'll get a file name from the user. The file name entered will appear in memory at SPABDT+9, as a string. For TI-Artist pictures, the main file name's last two characters will be _P. In case there's a corresponding color file, we make a copy of the root file name with the _C suffix at SPABD2+9. Once that's done, we put the VDP into Bitmap Mode by BL @SETBM. Now it's time to go and get that _P file direct from the disk into the Pattern Descriptor Table for Bitmap Mode.

Notice that the data line SPABDT has zero in its second word. That, you'll recall, is the buffer location for a LOAD operation. Thus the Device Service Routine will place what it finds in the described file into VDP RAM starting at 0, and that's where the Bitmap Pattern Descriptor Table is located.

But of course we have to put our PAB data in VDP someplace, then use DSRLNK to access the file. To do that, we first set R1 to point at SPABDT, get the length of the file name from SPABDT+8 into R2, add ten, then get that number from >8370 into R0. Now by subtracting R2 from R0, we will place the PAB into VDP so that it won't overlap the area reserved for the DSR's use. Thus the PAB ends at >3BE3, and the reserved area for DSR operations begins at the next byte, >3BE4. We do our BLWP @DSRLNK, with DATA 8, and the file gets put right where we can see it. When this is done, you'll see the picture come onto the screen sector by sector. After the BLWP to DSRLNK, we perform a JNE LODOK, so that if the file was found the program will jump ahead. If the file was not found, we report the error at the bottom of the screen, then wait for a keypress.

If the file loaded correctly, we move on to the code at LODOK, which sets up to access the _C file, if that exists. The PAB Data at SPABD2 has >2000 in its second word, so that the content of the _C file will load directly into the VDP at >2000, which is the color table for Bitmap Mode. Here we don't have to trap any error, because if the file is not found, the color table will be left black on white. If the file is found, its contents will replace the previous content of the color table, which is of course the desired result.

Now the picture is on-screen, with or without color, and can be viewed. The program keeps checking the keyboard until the user presses a key. When a key gets pressed, the program uses CPDT to clear the picture away, then goes back to graphics mode through SETGM. If **FCTN 9** was pressed, the program will jump to EXIT, else it will go back to LOAD. At that prompt, you can either type a file name or press **FCTN 9** to exit.

The EXIT routine puts a '3' into the CALL FILES data at PABDT+9, then executes a CALL FILES to set things back to the default condition before branching back to the GPL Interpreter. That's all for today. SIDEBAR48 can be assembled into an object file, and run from Option 3 with the program name START. You supply the TI-Artist Pictures, we'll put them on-screen. Once again next month's topic will be a surprise. See you then.

```
* SIDEBAR 48
* A COMPLETE PROGRAM
* FOR SHOWING TI-ARTIST PICTURES
* PUBLIC DOMAIN
* CODE BY B. Harrison
       DEF  START        ENTRY POINT
       REF  VMBW,VWTR,VSBW,VSBR,VMBR
       REF  KSCAN,DSRLNK
*
* REQUIRED EQUATES
*
GPLWS  EQU  >83E0        GPL WORKSPACE
KEYADR EQU  >8374        KEY-UNIT
KEYVAL EQU  >8375        KEY VALUE
STATUS EQU  >837C        GPL STATUS
PAB    EQU  >1000        PER. ACC. BLK
CHRTBL EQU  >2678        CHR DEF STASH
CALPNT EQU  >832C        PNTR FOR CALL
PABPNT EQU  >8356        FOR DSRLNK
*
* MAIN CODE SECTION
*
START  LWPI WS           LOAD WORKSPACE
       LI   R0,>0733     SCREEN GREEN
       BLWP @VWTR        WRITE VDP REG 7
       LI   R0,>380      COLOR TABLE
       LI   R1,SAVCLR    OUR COLORS
       LI   R2,32        32 BYTES
       BLWP @VMBW        WRITE COLORS
       LI   R0,31*8+>800 EDGE CHARACTER
```

```
       LI   R1,SPCURS    BOX PATTERN
       LI   R2,8         EIGHT BYTES
       BLWP @VMBW        WRITE THAT
       LI   R0,>8F0      CHARACTER 30
       LI   R1,CHRTBL    BUFFER STORAGE
       LI   R2,>308      CHR 30 THRU 126
       BLWP @VMBR        STASH DEFS
       CLR  @KEYADR      CLEAR KEY-UNIT
* FOLLOWING DOES A CALL FILES(1)
       LI   R0,>3100     SET '1'
       MOVB R0,@PABDT+9  IN PAB DATA
       LI   R0,PAB       POINT AT PAB
       LI   R1,PABDT     AND PAB DATA
       LI   R2,ENDPDT-PABDT LENGTH
       BLWP @VMBW        WRITE TO VDP
       MOV  R0,@PABPNT   R0 TO >8356
       MOV  R0,@CALPNT   AND TO >832C
       BLWP @DSRLNK      USE DSR LINK
       DATA >A           DATA FOR "CALL"
* FOLLOWING PUTS PROMPT ON SCREEN
LOAD   BL   @CLS         CLEAR SCREEN
       LI   R0,3*32+10   ROW 4, COL 11
       LI   R1,FILIN2    PROMPT STRING
       BL   @DISSTR      DISPLAY
LOADFN LI   R0,10*32+2   ROW 11, COL 3
       LI   R4,28        28 CHARS
       BL   @CRSIN       INPUT ROUTINE
       DATA SPABDT+9     IN PAB DATA
       CI   R8,15        FCTN-9?
       JNE  NAMOK        IF NOT, JUMP
       B    @EXIT        ELSE EXIT
* FOLLOWING SETS UP FILE NAME WITH _C
NAMOK  LI   R9,SPABDT+8  NAME LENGTH WRD
       LI   R10,SPABD2+8 SECOND PAB
       MOV  *R9,*R10+    MOVE THE WORD
       MOV  *R9+,R4      LENGTH INTO R4
       JEQ  LOAD         IF ZERO, BACK
       DECT R4           REDUCE R4 BY 2
       JLT  LOAD         IF < ZERO, BACK
MOVNAM MOVB *R9+,*R10+   MOVE A BYTE
       DEC  R4           DEC COUNT
       JNE  MOVNAM       NOT ZERO, RPT
       LI   R9,UNDC      _C TEXT
MOVUDC MOVB *R9+,*R10+   MOVE THE _
       MOVB *R9,*R10     AND THE C
       BL   @SETBM       BITMAP MODE
* FOLLOWING ACCESSES THE _P FILE, LOADS
* DIRECTLY TO THE VDP STARTING AT >0000
       LI   R1,SPABDT    PAB DATA FOR _P FILE
       MOV  @8(R1),R2    LENGTH INTO R2
```

```
       AI   R2,10        10 FIXED DATA
       MOV  @>8370,R0    HIGH VDP ADDR
       S    R2,R0        SUBTRACT LENGTH
       BLWP @VMBW        WRITE TO VDP
       AI   R0,9         ADD 9 TO R0
       MOV  R0,@PABPNT   PUT AT >8356
       BLWP @DSRLNK      USE DSR LINK
       DATA 8            FOR FILE ACCESS
       JNE  LODOK        IF NE, LOAD OK
       BL   @SETGM       ELSE GRAPHICS
       BL   @CLS         CLEAR SCREEN
       LI   R1,LERMSG    "LOAD ERROR"
       LI   R0,21*32+4   ROW 22, COL 5
       BL   @DISSTR      DISPLAY MSG
       LI   R0,23*32+4   ROW 24, COL 5
       LI   R1,PAK       "PRESS ANY KEY"
       BL   @DISSTR      DISPLAY THAT
       BL   @KEY         GET KEYSTROKE
       JMP  LOAD         BACK TO PROMPT
* FOLLOWING ATTEMPTS TO LOAD _C FILE
* DIRECTLY TO THE COLOR TABLE IN VDP
LODOK  AI   R0,-9        SUBTR 9 FROM R0
       LI   R1,SPABD2    _C PAB DATA
       BLWP @VMBW        WRITE THAT
       AI   R0,9         ADD THE 9 BACK
       MOV  R0,@PABPNT   PUT AT >8356
       BLWP @DSRLNK      USE DSR LINK
       DATA 8            FOR LOADING
       BL   @KEY         SCAN KEYBOARD
       BL   @CPDT        CLEAR BITMAP
       BL   @SETGM       SET TO GRAPHICS MODE
       MOV  @KEYADR,R8   KEY INTO R8
       CI   R8,15        FCTN-9?
       JEQ  EXIT         IF SO, EXIT
       B    @LOAD        ELSE TO PROMPT
* FOLLOWING DOES A CALL FILES(3)
EXIT   LI   R0,>3300     SET '3'
       MOVB R0,@PABDT+9  IN PAB DATA
       LI   R0,PAB       POINT AT PAB
       LI   R1,PABDT     AND PAB DATA
       LI   R2,ENDPDT-PABDT LENGTH
       BLWP @VMBW        WRITE TO VDP
       MOV  R0,@PABPNT   R0 TO >8356
       MOV  R0,@CALPNT   AND TO >832C
       BLWP @DSRLNK      USE DSR LINK
       DATA >A           DATA FOR "CALL"
       LWPI GPLWS        GPL WORKSPACE
       B    @>6A         GPL INTERPRETER
*
* SUBROUTINES FOR IN AND OUT OF BITMAP
```

```
*
* FOLLOWING SETS VDP INTO BITMAP MODE
*
SETBM  LI   R0,>206      REGISTER 2
       BLWP @VWTR        SIT TO >1800
       LI   R0,>403      REG. 4
       BLWP @VWTR        PDT TO >0000
       LI   R0,>3FF      REG. 3
       BLWP @VWTR        CT TO >2000
       LI   R0,>607      REG 6
       BLWP @VWTR        SDT to >3800
       LI   R0,>570      REG 5
       BLWP @VWTR        SAT to >3800
       LI   R0,>58       SIT AT >1800
       MOVB R0,@>8C02    LOW BYTE ADDR
       SWPB R0           SWAP R0
       MOVB R0,@>8C02    HIGH BYTE ADDR
       LI   R0,3         THREE TABLES
       CLR  R1           START WITH ZERO
SIT    MOVB R1,@>8C00    WRITE TO VDP
       AI   R1,>100      INC HIGH BYTE
       JNE  SIT          NOT ZERO, RPT
       DEC  R0           ELSE DEC COUNT
       JNE  SIT          NOT ZERO, RPT
       MOV  R11,R14      STASH RET. ADDR
       BL   @CPDT        CLEAR PDT
       LI   R0,>3800     SPRITE 0
       LI   R1,>D000     USE >D0, DELETE
       BLWP @VSBW        ALL SPRITES
       CLR  @>837A       NO MOTION
       LI   R0,2         SET TO WRITE 2 TO VDP REGISTER ZERO
       BLWP @VWTR        VDP TO M3 MODE (BITMAP)
       B    *R14         RETURN
CPDT   LI   R0,>60       CT AT >2000
       MOVB R0,@>8C02    LOW BYTE ADDR
       SWPB R0           SWAP R0
       MOVB R0,@>8C02    HIGH BYTE ADDR
       LI   R0,>1800     >1800 BYTES
       LI   R1,>1F00     BLACK ON WHITE
CT     MOVB R1,@>8C00    WRITE ONE BYTE
       DEC  R0           DEC COUNT
       JNE  CT           NOT ZERO, RPT
       LI   R0,>40       PDT AT >0000
       MOVB R0,@>8C02    LOW BYTE ADDR
       SWPB R0           SWAP
       MOVB R0,@>8C02    HIGH BYTE ADDR
       LI   R0,>1800     >1800 BYTES
       CLR  R1           ALL ZEROS
PDT    MOVB R1,@>8C00    WRITE ONE
       DEC  R0           DEC COUNT
```

```
       JNE  PDT          NOT ZERO, RPT
       RT
*
* FOLLOWING SETS VDP TO GRAPHICS MODE
*
SETGM  LI   R0,>1E0      VDP REG 1
       BLWP @VWTR        WRITE
       LI   R0,>200      VDP REG 2
       BLWP @VWTR        WRITE
       LI   R0,>401      VDP REG 4
       BLWP @VWTR        WRITE
       LI   R0,>30E      VDP REG 3
       BLWP @VWTR        WRITE
       LI   R0,>600      VDP REG 6
       BLWP @VWTR        WRITE
       LI   R0,>506      VDP REG 5
       BLWP @VWTR        WRITE
       LI   R0,>380      COLOR TABLE
       LI   R1,SAVCLR    COLOR DATA
       LI   R2,32        32 BYTES
       BLWP @VMBW        WRITE COLORS
       LI   R0,>8F0      CURSOR
       LI   R1,CHRTBL    STORED PATS
       LI   R2,>308      30 THRU 126
       BLWP @VMBW        WRITE THOSE
       LI   R1,>D000     CANCEL SPRITES
       LI   R0,>300      ATTR TABLE
       BLWP @VSBW        WRITE THAT
       CLR  @>837A       STOP MOTION
       CLR  R0           VDP REG 0
       BLWP @VWTR        WRITE
       RT                RETURN
DISSTR MOVB *R1+,R2      GET STRING LEN
       SRL  R2,8         RIGHT JUSTIFY
       JEQ  DISX         IF ZERO, SKIP
       BLWP @VMBW        WRITE STRING
       A    R2,R1        ADD LENGTH
DISX   RT                RETURN
*
CLS    CLR  R0           START OF VDP
       LI   R4,24        24 ROWS
       LI   R1,BLNKLN    SPACES
       LI   R2,32        32 PER ROW
LOOP   BLWP @VMBW        WRITE 32 SPACES
       A    R2,R0        ADD 32 TO ADDR
       DEC  R4           DEC ROW COUNT
       JNE  LOOP         NOT ZERO, RPT
       RT                ELSE RETURN
*
* FOLLOWING IS A MODIFIED VERSION
```

```
* OF OUR CRSIN ROUTINE AND ITS KEY
* INPUTS KI2 AND KI2A
*
CRSIN
       MOV  R11,R15
       CLR  @INSFLG
       MOV  R0,@PGNUM
       DEC  R0
       MOVB @EDGE,R1
       BLWP @VSBW
       INC  R0
       A    R4,R0
       BLWP @VSBW
       MOV  R0,@ENDOC
       S    R4,R0
       MOV  R4,@SAV4
CRSI0A BLWP @VSBR
       MOVB R1,@ALTKEY
CRSI0  BL   @CURFRC
       BL   @KI2
       CI   R8,9
       JEQ  CRSRT
       CI   R8,8
       JEQ  CRSBK
       CI   R8,10
       JLT  CRSC4
       CI   R8,15
       JEQ  CRSDMY
       CI   R8,13
       JLT  CRSDMY
CRSC4  CI   R8,4
       JNE  CRSENT
       INC  @INSFLG
       JMP  CRSI0
CRSENT CB   @KEYVAL,@ENTERV
       JEQ  CRSDMY
       CI   R8,3
       JEQ  CRSDEL
       CI   R8,32
       JLT  CRSI0
       CI   R8,122
       JGT  CRSI0
       CI   R8,97
       JLT  CRSI1
       SB   @ANYKEY,@KEYVAL
CRSI1
       MOV  @INSFLG,R1
       JEQ  CRSI1A
       MOVB @ALTKEY,R1
       BLWP @VSBW
```

```
        MOV  @ENDOC,R2
        S    R0,R2
        LI   R1,TEMSTR
        BLWP @VMBR
        DEC  R2
        JEQ  CRSI1A
        INC  R0
        BLWP @VMBW
        DEC  R0
CRSI1A  MOVB @KEYVAL,R1
        BLWP @VSBW
        INC  R0
        BLWP @VSBR
        CB   R1,@EDGE
        JNE  CRSI0A
        DEC  R0
        JMP  CRSI0A
CRSRT   MOVB @ALTKEY,R1
        BLWP @VSBW
        CLR  @INSFLG
        INC  R0
        BLWP @VSBR
        CB   R1,@EDGE
        JEQ  CRSRT1
        MOVB R1,@ALTKEY
        BL   @CURFRC
        BL   @KI2A
        CB   @KEYVAL,@RITEV
        JEQ  CRSRT
        CB   @KEYVAL,@NOKEY
        JEQ  CRSRT2
CRSRT1  DEC  R0
CRSRT2  MOVB @ONOFF,@KI2A+2
        MOVB @ALTKEY,R1
        BLWP @VSBW
        JMP  CRSI0
CRSBK   MOVB @ALTKEY,R1
        BLWP @VSBW
        CLR  @INSFLG
        DEC  R0
        BLWP @VSBR
        CB   R1,@EDGE
        JEQ  CRSBK1
        MOVB R1,@ALTKEY
        BL   @CURFRC
        BL   @KI2A
        CB   @KEYVAL,@LEFTV
        JEQ  CRSBK
        CB   @KEYVAL,@NOKEY
        JEQ  CRSRT2
```

```
CRSBK1 INC  R0
       JMP  CRSRT2
CRSDMY JMP  CRSIX
CRSDEL MOV  R0,R7
       CLR  @INSFLG
       MOV  @ENDOC,R2
       S    R0,R2
       INC  R0
       DEC  R2
       JEQ  CRSD1
       LI   R1,TEMSTR
       BLWP @VMBR
       MOV  R7,R0
       BLWP @VMBW
CRSD1  MOVB @ANYKEY,R1
       MOV  @ENDOC,R0
       DEC  R0
       BLWP @VSBW
       MOV  R7,R0
CRSD0  B    @CRSI0A
CRSIX  MOVB @ALTKEY,R1
       BLWP @VSBW
       MOV  @ENDOC,R0
       DEC  R0
       MOV  @SAV4,R2
CRSIX1 BLWP @VSBR
       CB   R1,@ANYKEY
       JNE  CRSIXX
       DEC  R0
       DEC  R2
       JGT  CRSIX1
CRSIXX MOV  @PGNUM,R0
       MOV  *R15+,R1
       SWPB R2
       MOVB R2,*R1+
       SWPB R2
       JEQ  CRIX
CRSIX2 BLWP @VMBR
CRIX   B    *R15
*
KI2    CLR  @STATUS
       BLWP @KSCAN
       LIMI 2
       LIMI 0
       DEC  R4
       JEQ  CHNG
       CB   @ANYKEY,@STATUS
       JNE  KI2
       MOV  @KEYADR,R8
       RT
```

```
CHNG   CI   R1,>1E00
       JEQ  L1
       LI   R1,>1E00
       BLWP @VSBW
       MOVB @ONOFF,R4
       JMP  KI2
L1     MOVB @ALTKEY,R1
       MOVB @ONOFF+1,R4
       BLWP @VSBW
       JMP  KI2
*
KI2A   LI   R5,>0280
KI2B   CLR  @STATUS
       BLWP @KSCAN
       CB   @KEYVAL,@NOKEY
       JEQ  KI2C
       LIMI 2
       LIMI 0
       DEC  R5
       JNE  KI2B
       MOVB R5,@KI2A+2
KI2C   RT
*
CURFRC LI   R1,>1E00
       LI   R4,>0100
       BLWP @VSBW
       RT
*
KEY    BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            INTERRUPTS ON
       LIMI 0            THEN OFF
       CB   @ANYKEY,@STATUS KEY STRUCK?
       JNE  KEY          NOT, SCAN AGAIN
       RT                ELSE RETURN
*
* END SUBROUTINES
* START DATA SECTION
*
WS     BSS  >20          OUR WORKSPACE
NOKEY  BYTE >FF          COMPARISON BYTE
SAV4   DATA 0            STORAGE FOR R4
TEMSTR BSS  30           TEMP STRING
LERMSG BYTE 21           LENGTH
       TEXT 'ERROR IN LOADING FILE'  MESSAGE
ANYKEY BYTE >20          COMPARISON BYTE
*
* SAVCLR IS JUST 32 BYTES OF >13 TO SET
* THE GRAPHICS COLORS TO BLACK ON GREEN
*
SAVCLR DATA >1313,>1313,>1313,>1313
```

```
       DATA >1313,>1313,>1313,>1313
       DATA >1313,>1313,>1313,>1313
       DATA >1313,>1313,>1313,>1313
PGNUM  DATA 0
ENDOC  DATA 0
EDGE   BYTE 31           EDGE CHARACTER
FILIN2 BYTE 12
       TEXT 'INPUT NAME  '
PAK    BYTE 25
       TEXT 'PRESS ANY KEY TO CONTINUE'
ENTERV BYTE >0D          ENTER KEY
LEFTV  BYTE >08          FCTN-S
RITEV  BYTE >09          FCTN-D
UPKEY  BYTE 11           FCTN-E
DNKEY  BYTE 10           FCTN-X
DELKEY BYTE 3            FCTN-1
INSKEY BYTE 4            FCTN-2
INSFLG DATA 0            INSERT ON-OFF
UNDC   TEXT '_C'         UNDERLINE C
BLNKLN TEXT '                                          '
ONOFF  DATA >0201        CURSOR ON-OFF
ALTKEY BYTE 0            SAVED CHARACTER
SPABDT DATA >0500,>0000,0,>2A00,>0000
       BSS  28           FILE NAME SPACE
SPABD2 DATA >0500,>2000,0,>2A00,>0000
       BSS  28           FILE NAME SPACE
SPCURS DATA >007E,>4242,>4242,>7E00 BOX
PABDT  BYTE 5            NAME LENGTH
       TEXT 'FILES'      NAME
       BYTE >B7,>C8,1,'1',>B6 TOKEN (1)
ENDPDT EQU  $
       END
```

1.49. The Art Of Assembly — Part 49. Gripes And Delays

By Bruce Harrison

It's November 1994 as we write, and things don't look so good for the future of the TI-99/4A. One by one, vendors are dropping by the wayside. Asgard is gone, along with many software authors. Bud Mills Services still exists, but there have been no ads appearing for some months now. Western Digital is still "talking about" the SCSI card and how soon its software will be ready. We're not holding our breath! One must ask, "When it's ready, will there be any customers left?" Part of the problem is the penny-pinching TI owner himself. In recent months, Dr. Charles Good has offered many of our Public Domain disks through the Lima Users' Group at $1.00 per disk, which includes the disk itself, the software with unlimited license, and the mailing. Still, people have called me directly to ask if I can offer these disks for less! LESS THAN $1.00? In all probability, it has cost more than $1.00 to call me long distance just to ask the question. (NO, I DON'T!) At the 1994 Lima Faire, we were not selling anything, but merely allowing people to make copies at our table if they brought along blank disks. Business was brisk all day. Obviously $0.00 was the correct price for our goods.

1.49.1. Could Be Worse. . .

I like using lines from old movies, and I'm reminded here of one from the movie "Young Frankenstein". In this case, the young Doctor and his assistant Igor are in the process of robbing a grave for the body they'll use in the experiment. They're standing neck-deep in the dug-up grave. The young Doctor Frankenstein (Gene Wilder) remarks what a horrible situation they're in, and Igor (the late Marty Feldman) quips, "Could be worse." The Doctor, annoyed, points out the details of their situation and asks "How could it possibly be worse?" Igor answers in a very matter of fact tone, "Could be raining." This is followed immediately by a stroke of lightning, a clap of thunder, and a drenching rain.

Bad as our situation is, it could be worse. At least the TI community still has active user groups who "know about" each other, and of course also has *MICROpendium* to keep in touch with even the "lone" user. Back when I acquired my first Radio Shack Color Computer (CoCo), I was looking around for other users to answer my many questions. Our friend Barry Traver got me a point of contact in the Philadelphia area for a CoCo users group. The man was available by phone, and eagerly helped with many questions, but when I asked whether there was a group in the Washington DC area, he didn't know! There's no forum for the CoCo users of the world, just isolated pockets of users. We TI people should count our blessings!

1.49.2. Slowing It Down

Several times in this column we've discussed the matter of creating controlled delays in an Assembly program. In many cases Assembly's biggest strength, its speed, is a curse. In today's Sidebar, we're showing some "near perfect" methods of doing a delay in any Assembly program. The first part is a simple subroutine which uses the user interrupt process to control the amount of the delay. Except for Register 11, this method does not change any registers, so it will not interfere with what's happening in the main code. Its only drawback is that it makes delays only in increments of 1/60th second.

As you'll see in the Sidebar, the subroutine is invoked with a BL @DELAY followed by a DATA line. The DATA is the number of sixtieths of a second for the desired delay. The DATA number can run from 1 through 32767. That means the delay itself can range from 1/60 second to 546 seconds, which works out to 9.1 minutes! In the first example shown in the Sidebar, the DATA is 30, so the delay will be 1/2 second. Since this delay uses the interrupt timing, the delay is independent of the processor or memory speed, so that the delay will be the same on bus-modified TI machines or on Geneves as it is on a standard TI.

1.49.3. The European Problem

If your program is to be used primarily in Europe or Australia, where the vertical interval is in 50ths of a second, the delay timing would have to be adjusted in the source code, as for example 25 instead of 30 would be used in the DATA to make a 1/2 second delay. If that's not done, all delays in your program would actually become 6/5ths of their desired value when run on a 50 Hz machine. In some cases, we've made two versions of a program, one for U.S. use, and another for European use.

1.49.4. Variations On The Theme

In the second part of the Sidebar, there's a variation on the subroutine, which shows a method by which the machine can be made to do other things while the delay is running. Here, we've added a BLWP @KSCAN within the loop, so that the user can terminate the delay by pressing a key. This requires one more byte added to the DATA section of the program so a check can be made after the BLWP @KSCAN. Using your imagination, you can see how other functions could be combined into the delay loop. (Another "exercise for the student".) Given the speed at which Assembly executes, the computer can do lots of things in each 1/60th second, without affecting the accuracy of your delay.

1.49.5. How It Works

The key to this is the little section of code at label USRINT. That performs only one simple task, incrementing the word of memory at label TIMCNT, then returning to the interrupt servicing routine. The interrupt servicing routine is always there in the console, and activates itself once each 1/60th second during the vertical interval, provided only that a LIMI 2 is being performed now and then. Thus TIMCNT gets incremented once each 1/60th of a second, which is of course the desired case. At the label DELAY, we first take the DATA that follows the BL instruction into a word of memory at DLYTIM. This sets the "target" number for the delay. Next, the subroutine clears the word at TIMCNT, so we start with zero in that variable. The third operation puts the address of USRINT into the word at >83C4, so this interrupt will be serviced on each vertical interval. The Interrupt servicing routine uses its own workspace, so the RT in USRINT returns to the servicing routine, and does not affect your main-code registers.

Once those things are done, the subroutine enters a loop at DLY1. This loop allows interrupts briefly during each pass by the LIMI 2 and LIMI 0 instructions. It then compares the words at TIMCNT and DLYTIM to see whether the desired delay has been completed. So long as the word at TIMCNT is less than the word at DLYCNT, the loop simply repeats. Once TIMCNT becomes equal to (or greater than) DLYTIM, the subroutine clears >83C4 to de-activate USRINT, then returns to your main code after the DATA item.

1.49.6. What Have We Done?

The Interrupt driven DELAY was inspired by two things. We had previously done a Timeout routine for use with Extended Basic, so that an ACCEPT AT could be limited to a certain amount of time, measured accurately in 1/60th second increments. That routine used the User Interrupt to advance a counter, thus making an accurate timeout possible. We'd also been looking at some source code written by one of our readers. The source code was for a game program that needed delay loops, and there were several different delay subroutines because different registers needed to be preserved.

After giving it a little thought, we figured that the delay subroutine could be made using the Interrupt idea, and that only Register 11 of the program's workspace would be affected. This takes more memory than one of our reader's original subroutines, but is still more efficient than having three or four different subroutines just to do delays.

This method was then used in one of our Public Domain "products", to serve as a timer. That product is a "SLIDESHOW" program which uses TI-Artist picture files. In that application, the user can select to have each slide in a sequence appear on-screen until a key is pressed, or he may choose to use a timed sequence, in which case each slide appears for a precise amount of time before being replaced. The user of that program enters his number in seconds, ranging from .1 second through 300 seconds (five minutes). We also wanted to give the user the ability to stop the show at any time by pressing **FCTN 9**. Thus a variation of the second method in today's Sidebar was applied and tested. It worked very nicely.

1.49.7. But Then Again. . .

Having been trained as an Electronics Engineer, and having worked in that profession for 30 years, I can testify to the fact that Engineers are never content to leave well enough alone. Left to themselves, Engineers would re-design a product for 100 years before sending it into production. Back then, I would have said that nobody could do more re-thinking of things than Engineers. Then I discovered Programmers. Programmers can re-work one product for much longer than the Engineers I'd worked with. That includes your author. In the fine tradition of never leaving well enough alone, we've added Part Three to today's Sidebar. This does not need the User Interrupt, but simply depends on one that's already there, in the form of the screen timeout counter. For icing on the cake, we made this one so that it works as a BLWP vector, without affecting any of the main program's registers. Like the previous ones, this uses a DATA item that can range from 1 through 32767, and produces a delay of that number of 60ths of a second. Since this is a BLWP operation, the subroutine has its own workspace, and gets the data using its own R14. Because the screen timeout counter increments by twos, we have to double the desired number before starting the delay loop. Of course we also want to start with the counter at zero, so we CLR @>83D6 before entering the loop.

For the convenience of our readers, this PART 3 of the Sidebar is a complete program that can be typed in, assembled, and run. It'll start by putting the message DELAYING TEN SECONDS on the screen. It will then invoke the delay by doing a BLWP @DELAY, followed by DATA 600. After the ten seconds delay, the program puts a "DELAY IS FINISHED" message on the screen, then awaits a keystroke and exits when it gets one. By my digital watch, this seems to be quite accurate.

Note that the number has to be doubled, which we do by a simple SLA R0,1 operation. Doing it this way insures that the number in R0 is always even, so it will match the number at >83D6 when the desired time has expired. All the subroutine needs to do, then, is just keep allowing interrupts once on each pass through the loop, and then compare the number in R0 to the screen timeout counter at >83D6.

As in the previous case, we could include a BLWP @KSCAN within the loop, so that the delay would be stopped by any keypress. This method, with its need for another set of workspace registers, is not exactly the most efficient in memory use, but it works as advertised, so we'll forgive ourselves for taking an extra 32 bytes in this case. It could turn out that your program needs to preserve all its own registers while the delay is happening, so those 32 bytes could be worth having.

So, once again, dear readers, we leave you with the choices to make yourself. Parts one, two, and three of the Sidebar all work, so you can use any of them to advantage in your programs. You can combine features, add other possibilities, and carry this stuff beyond our wildest dreams. In any case, we hope you're successful in your pursuits.

That's all for today's lesson. The programs we're making are all being made available through our dear friend Dr. Charles Good and the Lima Users' Group. So long as people are still interested, we'll keep trying new things. We're open to suggestions from readers at any time, either for needed "products" or for help with their own work. Next month's topic is again undecided, so watch this space for another surprise.

```
* SIDEBAR49
*
* OTHER DELAY METHODS
* PART ONE - WITH A USER INTERRUPT
* USES NO REGISTERS EXCEPT R11
* MAKES POSSIBLE DELAYS FROM
* 1/60TH SECOND THROUGH 546 SECONDS (9.1 MINUTES)
* THE EXAMPLE SHOWN WOULD GIVE 1/2 SECOND
*    CODE BY:  Bruce Harrison
*    PUBLIC DOMAIN
*    25 November 1994
* IN MAIN CODE, DELAY IS INVOKED THUS:
*      BL   @DELAY       USE DELAY SUBROUTINE
*      DATA 30           NUMBER OF 60THS OF A SECOND TO DELAY
* THE DATA ITEM ABOVE MAY RANGE FROM
* 1 THROUGH 32767
*
* THE DELAY SUBROUTINE IS:
*
DELAY  MOV  *R11+,@DLYTIM GET DESIRED DELAY
       CLR  @TIMCNT       CLEAR TIME COUNT
       MOV  @INTLOC,@>83C4 SET USER INTERRUPT
DLY1   LIMI 2             ALLOW INTERRUPTS
       LIMI 0             STOP THEM
       C    @TIMCNT,@DLYTIM  COMPARE TO DESIRED DELAY
       JLT  DLY1          IF LESS, REPEAT
       CLR  @>83C4        ELSE CLEAR USER INTERRUPT
       RT                 THEN RETURN
*
* THE INTERRUPT ROUTINE IS THIS:
*
USRINT INC  @TIMCNT       INCREMENT TIME COUNT
       RT                 THEN RETURN
*
* THE FOLLOWING GOES INTO THE DATA SECTION
*
TIMCNT DATA 0
DLYTIM DATA 0
INTLOC DATA USRINT
*
* PART TWO
* ANOTHER VERSION OF DELAY SUBROUTINE
* THIS TERMINATES AS SOON AS A KEY GETS PRESSED
*
DELAY2 MOV  *R11+,@DLYTIM GET DESIRED DELAY
       CLR  @TIMCNT       CLEAR TIME COUNT
       MOV  @INTLOC,@>83C4 SET USER INTERRUPT
DLY21  LIMI 2             ALLOW INTERRUPTS
       LIMI 0             STOP THEM
       BLWP @KSCAN        SCAN KEYBOARD
```

```
       CB   @>8375,@NOKEY NO KEY PRESSED?
       JNE  DLYEX         IF NOT, TERMINATE SUBROUTINE
       C    @TIMCNT,@DLYTIM  COMPARE TO DESIRED DELAY
       JLT  DLY21         IF LESS, REPEAT
DLYEX  CLR  @>83C4        ELSE CLEAR USER INTERRUPT
       RT                 THEN RETURN

* ADD THIS TO DATA SECTION WITH THAT
* SHOWN ABOVE
NOKEY  BYTE >FF           A BYTE OF -1
*
* PART THREE - YET ANOTHER METHOD
* USING BLWP VECTOR, AND AFFECTING NONE
* OF THE MAIN PROGRAM'S REGISTERS
* THIS USES THE SCREEN TIMEOUT COUNTER
* AT >83D6, WHICH COUNTS BY TWOS
*
* DELAY3/S
* WITHOUT USER INTERRUPT
* FOLLOWING IS A COMPLETE PROGRAM
* THAT CAN BE ASSEMBLED & TESTED AS IS
* PRODUCES A TEN SECOND DELAY
*
       REF  VMBW,KSCAN   REF UTILITIES
       DEF  START        DEFINE ENTRY
START  LWPI WS           LOAD WORKSPACE
       LI   R0,12*32+5   ROW 13, COL 6
       LI   R2,20        20 CHARACTERS
       LI   R1,DLYTXT    DELAY MESSAGE
       BLWP @VMBW        WRITE THAT
       BLWP @DELAY       USE VECTOR
       DATA 600          NUMBER OF 60THS OF A SECOND TO DELAY
*
* AS IN PREVIOUS CASES, THE DATA NUMBER
* MAY RANGE FROM 1 THROUGH 32767
* HERE, IT'S 600, FOR A 10 SECOND DELAY
*
       LI   R0,14*32+7   ROW 15, COL 8
       LI   R2,17        17 CHARACTERS
       LI   R1,DOVTXT    DELAY DONE MSG
       BLWP @VMBW        WRITE THAT
KEY    BLWP @KSCAN       SCAN KEYBOARD
       CB   @>837C,@ANYKEY KEY STRUCK?
       JNE  KEY          IF NOT, REPEAT
       LWPI >83E0        LOAD GPLWS
       B    @>6A         BACK TO E/A
*
* THE DELAY SUBROUTINE IS:
*
DELAY  DATA DLYWS,DLY0   BLWP VECTOR
```

```
DLY0   MOV  *R14+,R0     GET DESIRED DELAY
       SLA  R0,1         DOUBLE THE NUMBER
*
* WE DOUBLE THE NUMBER BECAUSE THE COUNTER
* ADVANCES BY TWO EVERY 1/60th SECOND
*
       CLR  @>83D6       CLEAR TIMEOUT COUNTER
DLY1   LIMI 2            ALLOW INTERRUPTS
       LIMI 0            STOP THEM
       C    R0,@>83D6    COMPARE NUMBERS
       JNE  DLY1         IF NOT EQUAL, REPEAT
       RTWP              ELSE BACK TO MAIN PROGRAM
*
* DATA SECTION
*
WS     BSS  32           MAIN WORKSPACE
DLYWS  BSS  32           DELAY WORKSPACE
DLYTXT TEXT 'DELAYING TEN SECONDS'
DOVTXT TEXT 'DELAY IS FINISHED'
ANYKEY BYTE >20          KEYSTROKE COMPARISON BYTE
       END
```

1.50. The Art Of Assembly — Part 50. Delays Again?

By Bruce Harrison

Remember last month? We said something about Programmers being likely to re-design programs over and over. True to our word, we've made still more variations on the "delay" theme, so this month we'll pick up where we left off. Today's Sidebar has, among other things, another complete program that doesn't do much, but serves to illustrate a couple of nifty ways of doing things.

1.50.1. Delay With Option

The program DELAY4 uses a modified version of the delay subroutine used in DELAY3 last month. This time, the programmer using the subroutine has the option of allowing the delay to be terminated by a keystroke or not. The "keystroke" option is signaled by a second word of DATA following the BLWP @DELAY instruction. As before, the first word of DATA tells DELAY how many 60ths of a second to run. The second word can be either zero or any non-zero number. If it's zero, then keystrokes during the delay period will have no effect. If it's non-zero, then any keystroke during the delay will cut the delay short. In the program shown, the first BLWP @DELAY has its second DATA word as zero, so while that message "DELAYING TEN SECONDS" is on the screen, pressing keys (other than **FCTN =**) will have no effect, and the user will just have to wait for the ten seconds to expire. When the second delay starts with the message "ANOTHER TEN SECONDS" on-screen, the "keystroke" abort is active, so pressing a key will terminate the delay early. Of course if no key is pressed, the delay will continue until the full ten seconds has passed. That's so because the second DATA word on this BLWP @DELAY is non-zero.

1.50.2. Another Potential Use

After the second delay in this program, we use the delay routine in another way, to produce a "slow reveal" of a message on the screen. Here, we delay by 1/10 second after each character in the message, unless that character was a space, in which case we skip over the delay. This has a key data word of zero, so that keystrokes will have no effect during the slow printing of the "PRESS A KEY TO EXIT" message.

The source code in the Sidebar is well annotated, so our loyal readers should have no trouble following its operations. There are some tricks employed, such as taking advantage of the fact that the messages are right after one another in memory, so we can A R2,R1 to move our VMBW pointer on to the next message.

We also took full advantage here of the lengths of the messages being close, so rather than LI R2 with the new lengths, we were able to just DEC R2, DECT R2, and INCT R2. That saves some memory, but of course can only be used if the next message is only one or two characters different in length.

1.50.3. What About Extended Basic?

Having gone to such lengths to provide delays for our Assembly readers, we felt that we should do something for our friends who program in Extended Basic. The common way of doing delays in XB is of course with a simple FOR-NEXT like this:

```
FOR DELAY=1 TO 500 :: NEXT DELAY
```

That can be used in a subroutine or even a sub-program, and a variable can be used in place of the 500 shown, so that the subroutine can produce different delays. This works okay so long as it's used on the same machine it was written on. If, however, one takes such a program to a Geneve, the delays will all be too short. Conversely, if the writer of the program used a Geneve or a Bus-modified TI, all the delays will be too long when it's run on a standard TI.

Once again, Assembly language comes to the rescue. We've made a special version of the delay routine tailored for use with XB programs. This version is sort of a "do everything" routine, so that the XB programmer can choose which features get implemented by changing the way the CALL LINK is written. Let's start with the case where what's desired is a simple delay of four and a half seconds. The linkage would look like this:

```
CALL LINK("DELAY",4.5)
```

This will cause an absolute delay of 4.5 seconds, regardless of the computer in use. Nothing except **FCTN = (QUIT)** will abort the delay.

Now suppose the XB programmer wants to allow the user to terminate the delay by pressing any key. To do that, just add a second parameter to the call link:

```
CALL LINK("DELAY",4.5,1)
```

That second parameter can be any non-zero number, or even a variable. If the second parameter happens to be zero, the effect is the same as the first case, and pressing keys will have no effect.

Okay, let's carry this one step farther. Suppose you want the ability to have any keystroke abort the delay, and you also want your XB program to know which key was pressed. For that, you make sure the second parameter is non-zero, then add a variable name as the third parameter:

```
CALL LINK("DELAY",4.5,1,K)
```

In this case there are two possible outcomes. If the user does not press a key during the delay, then the delay will run its full time, and the variable K will be zero after the delay is finished. If the delay is aborted by a keystroke (other than **FCTN 4** or **FCTN =**) the ASCII value of that key will be in the variable K. If, for example, the user terminated the delay by pressing **A**, then K will equal 65 upon exit from CALL LINK. If the user presses **FCTN 6**, K will equal 12, and so on.

Any of the three parameters may be a variable instead of a number, but the third one must be a variable if it's included. Range for the first parameter is from 1/60 through 546 seconds. The lowest number, 1/60th, will probably never be noticed. The highest number, 546, will result in a 9 minute and 6 second delay. Thus we don't recommend pushing the limits on these parameters. For ordinary delays in the range of, say, 1 second through 30 seconds, no problems should be noticed. Whenever the second parameter is present and non-zero, **FCTN 4** will be active to break the program during the delay, unless the program has executed ON BREAK NEXT before the CALL LINK. In that case, pressing **FCTN 4** will abort the delay, and K will equal 2, which is the value of the **FCTN 4** key.

Regardless of any parameters, the **FCTN =** key press will perform its usual function during the delay, sending the computer back to startup conditions. This will not be affected by an ON BREAK NEXT statement in the program.

1.50.4. The Source Code

Part Two of today's Sidebar has the source code for this little routine, which we call the "ultimate delay". This has the usual features you'd find in a routine written for XB use, including the EQUates at the beginning, the use of NUMREF to get parameters, and of NUMASG to assign the key's value if required. We've also used XMLLNK to perform multiplication of floating point numbers and to convert the answer to integers for our use. Notice that we find out the number of parameters present by moving the number from >8312 into R6, then use that to govern what the routine does. If you forget to supply any parameter, the routine will simply exit without any delay. So long as one parameter is provided, and is in the correct range, all should go well.

Assuming there's one parameter, the routine gets the first one with NUMREF. Since this number is a floating point quantity in seconds, and our counting will be done in 60ths, we place the floating point number 60 in the eight bytes starting at ARG (>835C), then use an XML service to multiply the number in seconds by sixty. XML gets used again to convert that result (at FAC) to an integer word. We then move that word to R4.

Just after the MOV to R4, we check to see if the number in R4 is zero or negative, and exit if either is the case, since we needn't bother delaying by zero, and we can't delay by a negative amount of time. Given that we have an acceptable number in R4, we check R6 to see whether we need to fetch the second parameter. If there's only one, we clear R5 so we'll "know" not to look for keys being pressed.

If there's a second parameter, we fetch that with NUMREF. As always, NUMREF places that parameter as a floating point number at FAC. Since we don't care about this number except whether it's zero or non-zero, we take advantage of the fact that the F.P. representation of zero makes the word at FAC equal zero. Thus we just MOV @FAC,R5, and that gives us what we want without need to convert to an integer.

The actual delay starts at label DLY0, where we double the number in R4 and clear the word at >83D6. That word is the screen timeout counter, which gets incremented by two on each vertical interval. (That's not what the E/A manual says, but it is what happens.) Now the delay loop starts at label DLY1. Each time through the loop, we allow interrupts briefly, so that the counter can work. We then check R5, to see whether we are supposed to check the keyboard.

If R5 is zero, we skip ahead to CKDLY. If R5 is not zero, we scan the keyboard. If a key has been pressed, we capture the key's value in R3, then jump ahead to label KEXIT, so we can report out the key value if necessary. If no key was pressed, we get to label CKDLY, where we compare the number in the timeout counter to our desired delay amount in R4. If those numbers are not equal, we jump back to DLY1 and repeat the process. If they are equal, we move on to label EXIT, which clears R3. That means that if the delay has run its full time, the key value will show up as zero.

Label KEXIT right-justifies the key value in R3, then checks the number of parameters in R6. If that's less than 3, it means we don't have to report the key value to XB, so we skip ahead. If R6 was 3 or more, we go ahead and put R3 at label FAC, set R1 to 3 for the third parameter, convert the number at FAC to floating point format, and assign that to the variable named as the third parameter.

Label NKEX starts the last part of the code. Here, we clear the screen timeout counter, then load the GPL Workspace, and branch to the GPL Interpreter at >6A. That puts XB back in control of the computer, and of course what happens next depends on what's in your program right after the CALL LINK statement.

As with so many other of these utility items, we've made this routine DELAY available as part of a Public Domain disk, and have sent a copy to our friend Dr. Charles Good at the Lima Users' Group. Thus you or your group can easily obtain this routine, complete with its source code, instructions for using it, and so on. That address is:

Dr. Charles Good
P.O. Box 647
Venedocia, OH, 45894

We've called this disk The Ultimate Delay, because we think it offers all the flexibility anyone could ask for, and because it will yield reasonably accurate delay timing regardless of whether it's used on a TI or Geneve. For European users, whose systems work at 50 Hz instead of 60 Hz, there's a European version on the same disk. (in this case, Europe includes the U.K. and Australia.)

Next month we promise there will be no more delays, but will try to surprise you with some other topic. See you then.

```
* SIDEBAR 50
* PART ONE - A COMPLETE E/A PROGRAM
*
* DELAY4/S
* A FOURTH DELAY METHOD
* WITHOUT USER INTERRUPT
* WITH OR WITHOUT KEYSTROKE TERMINATE
* FOLLOWING IS A COMPLETE PROGRAM
* THAT CAN BE ASSEMBLED & TESTED AS IS
* ILLUSTRATES USE FOR SLOW REVEAL
* AND ORDINARY DELAY
*
*      Code by Bruce Harrison
*      27 November 1994
*      PUBLIC DOMAIN
*
       REF  VMBW,VSBW,KSCAN   REF UTILS
       DEF  START         DEFINE ENTRY
START  LWPI WS            LOAD WORKSPACE
       LI   R0,9*32+5     ROW 10, COL 6
       LI   R2,20         20 CHARACTERS
       LI   R1,DLYTXT     DELAY MESSAGE
       BLWP @VMBW         WRITE THAT
       BLWP @DELAY        USE VECTOR
       DATA 600           60THS TO DELAY
       DATA 0             NO KEY ABORT
       LI   R0,11*32+6    ROW 12, COL 8
       A    R2,R1         SECOND MESSAGE
       DEC  R2            19 CHARACTERS
       BLWP @VMBW         WRITE THAT
       BLWP @DELAY        USE DELAY
       DATA 600           60THS TO DELAY
       DATA 1             KEY ABORT ON
       LI   R0,13*32+7    ROW 14, COL 7
       A    R2,R1         "DELAY FINISHED"
       DECT R2            TWO SHORTER
       BLWP @VMBW         WRITE THAT
       AI   R0,63         DOWN 2 ROWS -1 COL
       A    R2,R1         "PRESS A KEY"
       INCT R2            TWO CHARS LONGER
       MOV  R1,R3         ADDRESS INTO R3
SLOWRV MOVB *R3+,R1       ONE BYTE TO R1
       BLWP @VSBW         WRITE THAT
       CB   R1,@ANYKEY    WAS IT A SPACE?
       JEQ  SLSK          IF SO, NO DELAY
       BLWP @DELAY        USE DELAY
       DATA 6             1/10 SECOND
       DATA 0             NO KEY ABORT
SLSK   INC  R0            NEXT SPOT ON SCREEN
       DEC  R2            DEC LENGTH
```

```
       JNE  SLOWRV       IF NOT ZERO, RPT
KEY    BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            INTERRUPTS ON
       LIMI 0            STOP THEM
       CB   @>837C,@ANYKEY KEY STRUCK?
       JNE  KEY          IF NOT, RPT
       LWPI >83E0        LOAD GPLWS
       B    @>6A         BACK TO E/A
*
* THE DELAY SUBROUTINE IS:
*
DELAY  DATA DLYWS,DLY0   BLWP VECTOR
DLY0   MOV  *R14+,R0     GET DESIRED DELAY
       SLA  R0,1         DOUBLE THE NUMBER
       MOV  *R14+,R1     GET KEY FLAG IN R1
       CLR  @>83D6       CLEAR TIMEOUT COUNTER
DLY1   LIMI 2            ALLOW INTERRUPTS
       LIMI 0            STOP THEM
       MOV  R1,R1        KEY TO STOP?
       JEQ  DLY2         IF NOT, JUMP
       BLWP @KSCAN       ELSE SCAN KYBRD
       CB   @>837C,@ANYKEY  KEY STRUCK?
       JEQ  DLYEX        IF SO, EXIT
DLY2   C    R0,@>83D6    COMPARE NUMBERS
       JNE  DLY1         IF NOT EQUAL, RPT
*      CLR  @>83D6       CLEAR TIMEOUT
DLYEX  RTWP              BACK TO CALLER
*
* DATA SECTION
*
WS     BSS  32           MAIN WORKSPACE
DLYWS  BSS  32           DELAY WORKSPACE
DLYTXT TEXT 'DELAYING TEN SECONDS'
SCND   TEXT 'ANOTHER TEN SECONDS'
DOVTXT TEXT 'DELAY IS FINISHED'
PAK    TEXT 'PRESS A KEY TO EXIT'
ANYKEY BYTE >20          KEYSTROKE COMPARISON BYTE
       END
*
* PART TWO - A COMPLETE SUBROUTINE
* FOR XB USERS TO MAKE DELAYS
*
* ULTDLY/S
* FOR PRECISE DELAYS FROM XB
* MAKES POSSIBLE DELAYS FROM
* 1/60TH SECOND THROUGH 546 SECONDS
* INVOKE FROM EXTENDED BASIC BY:
* CALL LINK("DELAY",T,U,K)
* WHERE T IS DESIRED DELAY IN SECONDS
*       U IS ZERO OR NON-ZERO
```

```
*       K IS VARIABLE TO REPORT KEYVAL
* U AND K MAY BE OMITTED FOR PLAIN DELAY
* K MAY BE OMITTED IF KEYSTROKE TO ABORT
* BUT NOT REPORTED BACK TO XB
*    CODE BY:  Bruce Harrison
*    PUBLIC DOMAIN
*    07 December 1994
*
       DEF  DELAY        DEFINE ENTRY POINT
*
* REQUIRED EQUATES
*
NUMASG EQU >2008         NUMERIC ASSIGN
NUMREF EQU >200C         NUMERIC REF
KSCAN  EQU >201C         KEYBOARD SCAN
XMLLNK EQU >2018         XML VECTOR
CFI    EQU >12B8         CONV F. P. TO INTEGER
CIF    EQU >20           CONV INTEGER TO F. P.
FAC    EQU >834A         FLOATING POINT ACCUM.
ARG    EQU >835C         FLOATING POINT ARGUMENT
STATUS EQU >837C         GPL STATUS BYTE
ARGNUM EQU >8312         NUM OF ARGS
FMUL   EQU >0E88         F. P. MULTIPLY
*
* MAIN CODE SECTION
*
DELAY  LWPI WS           LOAD OUR WORKSPACE
       MOVB @ARGNUM,R6   NUMBER OF ARGS TO R6
       SRL  R6,8         RIGHT JUSTIFY
       JEQ  EXIT         IF NO ARGUMENTS, EXIT
       LI   R1,1         FIRST PARAMETER
       BLWP @NUMREF      USE NUMREF
       LI   R9,SIXTY     POINT AT 60
       LI   R10,ARG      AND ARGUMENT
       LI   R4,8         EIGHT BYTES
MOVARG MOVB *R9+,*R10+   MOVE A BYTE
       DEC  R4           DEC COUNT
       JNE  MOVARG       IF NOT ZERO, RPT
       BLWP @XMLLNK      USE XML LINK
       DATA FMUL         MULTIPLY BY 60
       BLWP @XMLLNK      USE XML AGAIN
       DATA CFI          CONVERT TO INTEGER
       MOV  @FAC,R4      PUT DELAY # IN R4
       JEQ  EXIT         IF ZERO, EXIT
       JLT  EXIT         IF NEGATIVE, EXIT
       CI   R6,1         COMPARE ARGS TO 1
       JGT  GETTWO       IF GREATER, JUMP
       CLR  R5           ELSE CLEAR KEY FLAG
       JMP  DLY0         THEN JUMP AHEAD
GETTWO INC  R1           2ND PARAMETER
```

```
       BLWP @NUMREF      GET NUMBER
       MOV  @FAC,R5      MOVE TO R5
DLY0   SLA  R4,1         DOUBLE R4
       CLR  @>83D6       CLEAR TIMEOUT COUNT
DLY1   LIMI 2            ALLOW INTERRUPTS
       LIMI 0            STOP THEM
       MOV  R5,R5        CHECK R5
       JEQ  CKDLY        IF ZERO, SKIP
       BLWP @KSCAN       ELSE SCAN KEYBOARD
       CB   @ANYKEY,@STATUS KEY PRESSED?
       JNE  CKDLY        IF NOT, JUMP AHEAD
       MOVB @>8375,R3    GET KEYVAL IN R3
       JMP  KEXIT        THEN JUMP AHEAD
CKDLY  C    @>83D6,R4    COMPARE TO DESIRED NO.
       JNE  DLY1         IF NOT EQUAL, RPT
EXIT   CLR  R3           CLEAR REG 3 - NO KEY
KEXIT  SRL  R3,8         RIGHT JUSTIFY KEYVAL
       CI   R6,3         COMPARE ARGS TO 3
       JLT  NKEX         IF LESS, SKIP AHEAD
       MOV  R3,@FAC      MOVE KEYVAL TO FAC
       BLWP @XMLLNK      USE XML
       DATA CIF          CONVERT TO F.P.
       LI   R1,3         3RD PARAMETER
       BLWP @NUMASG      ASSIGN VALUE
NKEX   CLR  @>83D6       CLEAR TIMEOUT COUNTER
       LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         GO TO GPL INTERPRETER
*
* DATA SECTION
*
WS     DATA 0            PRELOADED R0
       BSS  30           R1 THRU R15
SIXTY  BYTE 64,60,0,0,0,0,0,0  SIXTY AS F.P.
ANYKEY BYTE >20          COMPARISON BYTE
       END
```

1.51. The Art Of Assembly — Part 51. Body Of Work

By Bruce Harrison

Today we're taking a different tack. We thought it might be of some interest to our readers to know what all is available from the many projects we've done on our TI. Some of these have been mentioned in this column, of course, and in some cases we've put complete programs as source code in the Sidebars. Still, it's been nearly 12 years that we've been pecking away at the TI, and lots of "products" have come from that work. Some of those products were produced for the Commercial market under the Harrison Software label. Except for the "Assembly Music" products, which have been released to Public Domain, all of that commercial stuff has been pulled from the market. The "Assembly Music" products were all released through Jim Peterson's TI-PD catalog. Since Jim's passing, we don't know exactly what's the status of that catalog, so we're not listing those products here.

1.51.1. The Public Domain Products

Since the demise of Harrison Software as a commercial venture for the TI Community, we've produced a large number of Public Domain disks. Most have some Assembly Language content, and some are complete programs, while others are just small utility routines for Extended Basic programmers. We've made sure that all of these products are available through two sources. The primary distribution point is the Lima Users' Group. Dr. Charles Good has kept all of these in the Group's Public Domain library, and makes them available by mail order at a nominal cost. Contact the group c/o Dr. Charles Good, P.O. Box 647, Venedocia, OH 45894. For those with modems, we've also supplied these disks to Barry Traver, who's made them available on Genie. (Disks may or may not be available there, as Genie has a time limit on disks kept on-line.) From time to time we've made updates, and have provided Dr. Good and Barry Traver with the latest versions as appropriate.

Some of these products were inspired or suggested by our friends in the community of TI users. Others were products of our own imagination, which we hoped somebody would find useful. In most cases, we've gotten some feedback, at least from Barry Traver or Dr. Good, to indicate that there's some need for the items. Here then is a listing and description of what's out there for use by anybody who needs it. The number after each description is the Disk number for that disk in Lima's Disk Library.

1.51.2. Extended Basic "Utilities"

Utilities Volume 1 — A collection of routines for use with Extended Basic, includes Boot Tracking routines so that an XB program can find out what disk it was loaded from, and can even modify all references to DSK1 within itself. Also includes special forms of ACCEPT AT for numeric and string variables. #836A

Utilities Volume 2 — A collection of Assembly routines that deal with DATA in the XB program. These can make quick work of, for example, assignment of DATA to array variables. Also includes a very fast MENU driver which works with DATA from the XB program and puts a nicely composed menu on-screen in a hurry. #1081B.

The Ultimate Accept AT — A special ACCEPT AT assembly routine which will provide a prompt, accept more than 28 characters into string variables, either clear the field or not, and so on. # 834A.

Time Calculator — A purely XB program that calculates time quantities in Hours, Minutes, Seconds. #863B.

Short Danny — A modified version of the old Danny Michaels Screen Dump - provides a quick loading process and allows dump to be activated via **FCTN 7** while programs are running or in Command mode. #801B.

TIMEOUT — Provides a way to have ACCEPT AT, CALL KEY, or INPUT statements run for a limited time, then exit back to XB program. Timing is done by an Interrupt, so it doesn't change with speed of the computer. #926A.

Music Background — Provides a way to have music play "on background" while the computer is doing other things. Can be activated while programs are running or in Command mode. Demos show how to provide music during an ACCEPT AT operation. #934B.

The Ultimate Delay — A delay that's unaffected by speed of the processor, and can optionally be aborted by a keystroke or not. This includes a "EUR" version for 50 Hz PAL systems. #1079B.

Checktime — Provides a means of measuring the speed of execution for XB programs. Uses an interrupt to measure time while an XB program is running, then reports elapsed time back to XB variables in minutes and seconds. (Does not require a "real-time" clock.) This is also available in a "EUR" version for 50 Hz PAL video systems. U.S. Version, # 1079A, EUR #1081A.

Font Converter — This does not use Assembly routines, but makes Assembly source files which allow conversion of Jim Peterson's Screen Fonts into CHARA1-type files. #929.

Loader — This is a combined XB and Assembly product which can load and run almost any E/A Option 5 file from Extended Basic. #922B.

Callfiles — An Assembly routine that allows an XB program to perform CALL FILES while it's running. # 957B.

Randoms — A group of utilities to provide very fast random number generation in XB programs. Includes routines for making both integer and floating point random numbers. # 1082B.

1.51.3. The All-Assembly Products

Reformat — Assembly program that takes D/V 80 text files as created by TI-Writer or Funnelweb's Text Editor, and allows the user to re-format very quickly to any number of characters per line. Also performs right-justification and margin change at user's option. #946A.

Midi Toolbox — Tools for doing things with source files for MIDI-Master. Includes many useful items, including one which will determine in advance whether a MIDI-Master music file will fit into memory when compiled, and one which will convert the durations of notes to go from Geneve timing to TI or vice versa. #1080.

Metronome — A tool for the musician or the child taking music lessons. Provides a steady "tick" at a selected number of beats per minute. Range is from 15 beats per minute through 500 beats per minute. This disk also includes a "EUR" version for use with 50 Hz PAL video systems. #869A.

Sandwich — A utility for the frustrated E/A program owner. This will allow the user to convert an E/A Option 3 object file into the more efficient and quicker-loading Option 5 format, without having access to the original source files. It won't work in all cases, but where it will, it's worth having. #869B.

Extended Basic Compiler — This is not a cure-all for every Extended Basic program in your library, but can provide improved speed of execution for many XB programs. Not recommended for very large XB programs. #1013. Source code #1014.

Drawing Program — A poor man's way to draw Bitmap pictures on your TI. Also allows use of TI-Artist Instances and Fonts. Includes printing capability. #928.

Video Titler — Allows use of either Harrison Drawing program pictures or TI-Artist pictures as titles for video taping. Provides for two complete pictures to be in memory, and allows smooth "wipes" from one picture to the next. #1011.

Font turner — Allows user to rotate the characters in a CHARA1-type file to the left, right, or upside down. Mainly intended for use with the Drawing Program. #1012B.

Slideshow — A program that allows use of TI-Artist picture files for an automated "Slide show". There is also a "EUR" version for 50 Hz PAL video systems. #1075 (U.S. version) or 1076 (European version).

Password — A very special program for those who use Horizon Ramdisks. Provides a way to secure your TI system with a private password all your own. #935A.

1.51.4. The "Extras" On The Disks

All of the disks in this "collection" include instructions, plus an Extended Basic program to print those instructions. Most also include the source files for their Assembly content, and the pure Assembly ones include an Extended Basic loader so that users who have only Extended Basic may still enjoy the Assembly programs.

Some of the disks come in more than one version. That's mostly true of the ones that involve timing with the vertical interval, in which cases there is a second version for the "European" market which uses 1/50th second timing.

1.51.5. A Word Of Caution

To those who have Geneve computers. Not all of these things will work correctly on the Geneve. Not having one, we can't test on that machine, so we can't guarantee that any of our products will be useful on that machine. The same goes for the owners of TI computers with various third-party hardware installed. Some won't work with 80-column cards, or with Myarc or CorComp disk controllers, and so on. All are compatible with Horizon Ramdisks and the Horizon P-Gram.

1.51.6. A Bud Mills Mystery!

Just recently, we experienced a problem with one of our Horizon Ramdisks. Sometimes, the problems in one Ramdisk will affect the whole system, and that seemed to be the case. The Horizon Config program was calling this particular card "unrecognizable". Having run into this kind of thing before, and knowing that what was on that card was safely backed up on Floppies, we took the ultimate step for such cases, pulling out one of the batteries from that card for a few seconds.

We put the card back into the system, turned on the card and then the system, fully expecting to have to re-initialize our drives 6 & 7. NOT SO! For reasons we can't fathom, the card behaved as if everything was still there! We re-loaded the ROS from a backup disk, but didn't need to do anything else! How did the card still retain all the data when its battery had been removed? We were under the impression that once a battery had been removed, the card would promptly "forget" everything it once "knew". If anybody knows, please don't tell us, so this can stay a mystery forever. Maybe we'll submit the case to "Unsolved Mysteries" and see how Robert Stack tells it.

1.51.7. What Will We Do Next?

That, dear readers, is largely up to you. We have been known to create special programs just to solve one particular problem for just one person in our "community". We've also taken on challenges issued by users, and developed whole products just to answer a challenge. We could keep on doing this forever, but sometimes a little inspiration helps. If you have a problem that you think we should work on, or if you just need help with one of your own projects, just drop us a line, either through *MICROpendium* or direct to:

Bruce Harrison
5705 40th Place
Hyattsville MD 20781
U.S.A.
Phone (301) 277-3467

We look forward to hearing from you. Perhaps your problem will be next month's column topic.

1.52. The Art Of Assembly — Part 52. Cheap And Dirty

By Bruce Harrison

Without further delay, we get right into a new topic. This month, we're re-visiting the subject of pseudo-random numbers. We're also reworking parts of our Video Titler, in response to a request from our friend Dick Bulmer, of Omemee, Ontario. Dick felt that our Titler could use a couple more "wipes". We started playing around with his idea, and added some of our own, ending up with eleven new "wipes" in the program.

One of those is not really a wipe, but a random-block replacement of one picture with the other. To do this in the limited memory available (The Titler program resides entirely in low memory) we had to invent a slightly better method of doing our random numbers. Hence the title "Cheap and Dirty" for today's column, as the random number routine in the Sidebar is not elegant, nor does it supply very high quality sequences of random numbers, but it does well enough for the intended purpose.

1.52.1. The Seeding Process

To get our sequence of pseudo-random numbers started, we need a "seed" number that's unpredictable. In this case, when we enter the process, a "key loop" has been running, and that loop includes the LIMI 2 and LIMI 0 instructions. Thus the counter at >8379 will have been incrementing every sixtieth of a second, and neither we nor the user will be able to predict its state. Being only a one-byte counter, this has only 256 possible states, and so can yield only 256 different sequences of random numbers. That's enough, however, to give the user the impression of complete randomness in the way one picture replaces the other. We've arranged things in this case so that no block in the picture is written twice, and that the process always takes about the same amount of time, regardless of the particular seed number used. How? Well, that would be telling!

1.52.2. Okay, We'll Tell!

We want to take each of the numbers from 0 through 191 once and only once, but in random order. We'll write the new picture into place as 192 blocks of 32 bytes each. We want to do this with minimal expenditure of time and memory. Now look at the Sidebar, and we hope it will all be clear as mud in just a moment. But first, just a word or two about how things are set up in memory when the Titler is running. The program is in low memory. Assuming two pictures have been loaded as Frame 1 and Frame 2, they're in high memory from >A000 through >FFFF. Frame 1's pattern descriptions (black and white data) runs from >A000 through >B7FF. That frame's color data runs from >B800 through >CFFF.

Frame two is in two similar blocks starting at >D000 and running right to the end of memory at >FFFF. Its color part starts at >E800. When one of those is put into VDP RAM so we can see it, its pattern part starts at 0 in VDP, and its color part at >2000.

The first line in the Sidebar uses a subroutine (not shown) to set R1 to point to the pattern part of whichever frame is not currently on-screen. Thus after that subroutine, R1 will contain either >A000 or >D000, as needed. We made a kind of practical choice to take our random blocks in chunks of 32 bytes at a time. That means the whole picture is 192 such blocks, each of which occupies four character-size chunks of the screen.

Now we stash away R1 into R13, so we can change R1 and still get its original value back from R13 when we need it. We will now construct in memory a table of numbers from 0 through 191, each occupying one byte. We put this table starting at label EAUT. That's an area which gets used to store the E/A utilities for our program, but at this point the utilities have already been moved into place in low memory, so we can se this part of memory over again, rather than setting aside a separate block. When the table is finished, he left byte of R3 contains 192, and we want that number, but in the right byte. Since the right byte is still zero, we SWPB to get R3 to contain 192.

Now we have to get a random seed, which we do by simply moving the word at >8378 into >83C0. That number will be somewhere in the range of 0 through 255. Next we use a method borrowed from TI, multiplying the number from >83C0 by 28,645, then adding 31,417 to the low-order word of that product in R5. We move that low order word back to >83C0 for next time, then discard the high order word of the product by clearing R4. At this stage, the register pair R4-R5 contains a "random" number in the range of 0 through 65,535. We don't care what it is, we're going to divide it by the number in R3. For this first pass, R3 contains 192. After division, there will be a remainder in R5 that will always be a number in the range 0 through 191. Gee, that sounds like what we were after, doesn't it? Indeed it is!

Now we take the corresponding number from our table. This first time, that will be whatever number we had in R5 to start with. Let's say it's 95, for example. We move that byte into the left byte of R0 with MOVB @EAUT(R5),R0. To get R0 set to that number as a word, we SRL R0,8. Now R0 equals 95, but we want to point to a block of bytes on a 32-byte boundary in VDP, so we multiply that number in R0 by 32 with SLA R0,5.

Still with us? It's getting tricky now. We get the base of the "Frame" from R13 into R1, then add the "offset" by simply adding R0 to R1. Now before we proceed to write this, let's review where we are. R0 points to a spot in the pattern descriptor part of VDP RAM, on a 32-byte boundary. R1 points to the corresponding place in the frame in memory that's currently not on the screen. R2 still contains 32, so when we BLWP @VMBW, four adjacent characters will get newly defined on the screen. That takes care of the "black and white" for those 32 bytes, so now we have to get the corresponding color information into place to go with that. To do that, we add >2000 to R0, add >1800 to R1, leave R2 alone, and BLWP @VMBW again.

1.52.3. What Have We Done?

At this stage, we have the screen mostly showing one picture, but somewhere in the screen there's a four character block from the new picture. Now hold onto your hat, because here's where it gets even trickier! We're going to remove the number we just picked from the table so that it won't be picked again. We do that in the line MOVB @EAUT-1(R3),@EAUT(R5). The number at EAUT-1(R3) is one that won't be available on the next pass, because we'll DEC R3 before going through the loop again. Thus the number that we took from the table will no longer be there. The position formerly occupied by 95 will contain 191 now. Thus if our random number process should again yield 95, it will go to the 95th place in the table, but 95 won't be in the table any more.

Now the table no longer contains 192 numbers, but only 191 of them (0 through 191 with 95 missing and 191 being in the place of 95), so we have to adjust for that before going for another random number. We thus DECR3, so that it contains 191. Since R3 is not zero, we go back for another random number. This time, when we divide by R3, the remainder in R5 will be a number from 0 through 190, and our random number will be chosen from those 191 numbers still in the table.

Thus we make 192 passes through the loop that starts at label RANDN0. Each time we take a different 32-byte chunk and its corresponding color bytes from the new frame in high memory and write those into VDP, replacing that piece of the picture that was on the screen. Note that on our last pass through the process, R3 contains 1, so the DIV operation will yield no remainder, and our last number will be whatever was left at position EAUT. (That won't necessarily be zero, because zero could have been removed from the table earlier.) When R3 decrements to zero, we're finished. All 192 blocks of the new picture are on-screen, and we can branch back to a place in the program that waits for the user's next keystroke.

Just for the edification of those who are still novices in Assembly, we've put into the Sidebar two small Extended Basic programs that emulate what our Assembly routine does, at least as well as XB can do that. The first runs only the numbers from 0 through 10, so it takes only two seconds to run. The second one does 192 numbers, like the Assembly one does, but that takes about twenty-eight seconds.

To see how fast the Assembly code does this, you'll need to get our disk "Video Titler" from our friend Dr. Good (see last month's column for his address). You'll simply load in two pictures, (samples are on the disk) then view either of them, and press **R** on the keyboard. In less than a second, all 192 random blocks will have been selected and written to the VDP RAM.

1.52.4. Other Changes

The Titler disk has been updated as of 22 December 1994 to include the following additional "wipes":

KEY	*Wipe action*
J	Horizontal wipe from edges to center
C	Vertical wipe from top and bottom to middle
I	Inward spiral from outside to center
O	Outward spiral from center to edges
R	Random sequence of 192 picture pieces
Y	Venetian blind wipe downward
U	Venetian blind wipe upward
1	Corner from upper left to lower right
=	Corner from upper right to lower left
Z	Corner from lower left to upper right
.	Corner from lower right to upper left

Of these, the J and C keys were as requested by Dick Bulmer, to complement the H and V keys which go from center to edges and from middle to top and bottom, respectively. When we started this attempt, we weren't sure whether the J and C key actions would fit in memory, but found those so easy that we went ahead and added the others. We still have some of low memory left, but can't think of any more wipes we'd like to try just now.

The Assembly code in today's Sidebar is of course just a small part of a program, and you'd have to modify it to do your own particular job. In most cases, you'd need a block of memory set aside somewhere to contain your table of numbers. If that table is to contain more than 256 numbers, you'll have to double its size so that each number is a word instead of just a byte, and you'll have to make sure it starts on an even address. There we go again, creating yet another exercise for our serious readers. That should keep you busy 'til next month, when we'll try to provide another pleasant surprise. See you then.

```
* SIDEBAR 52
* CHEAP AND DIRTY RANDOM NUMBERS
*   Code by Bruce Harrison
*   22 December 1994
*   PUBLIC DOMAIN
*
* PART ONE - ASSEMBLY CODE
*
* THIS IS JUST A "SNIPPET",
* NOT A COMPLETE PROGRAM
*
RNDCHG BL   @STFRM       SET FRAME ADDRESS
RNDCH0 LI   R2,32        32 BYTE CHUNKS
       MOV  R1,R13       STASH R1 IN R13
       CLR  R3           START WITH 0 IN R3
       LI   R9,EAUT      POINT AT MEMORY AREA
BLDTBL MOVB R3,*R9+      MOVE LEFT BYTE R3
```

```
       AI   R3,>100      INCREMENT LEFT BYTE
       CI   R9,EAUT+192  AT END OF TABLE?
       JLT  BLDTBL       IF LESS, REPEAT
       SWPB R3           SWAP SO R3=192
       MOV  @>8378,@>83C0 MOV TIMER TO SEED
RANDNO LI   R4,28645     LOAD A BIG NUMBER
       MPY  @>83C0,R4    MULTIPLY BY SEED
       AI   R5,31417     ADD A BIG NUMBER
       MOV  R5,@>83C0    MOV RESULT TO SEED
       CLR  R4           CLEAR HIGH WORD
       DIV  R3,R4        DIVIDE R4-R5 PAIR BY R3
       MOVB @EAUT(R5),R0 TAKE NUMBER FROM TABLE
       SRL  R0,8         RIGHT JUSTIFY
       SLA  R0,5         MULTIPLY BY 32
       MOV  R13,R1       GET R1 BACK
       A    R0,R1        ADD OFFSET FROM R0
       BLWP @VMBW        WRITE 32 BYTES
       AI   R0,>2000     POINT TO COLOR TABLE
       AI   R1,>1800     AND STORED COLOR
       BLWP @VMBW        WRITE COLOR PORTION
       MOVB @EAUT-1(R3),@EAUT(R5) REPLACE NUMBER
       DEC  R3           DECREMENT R3
       JNE  RANDNO       IF NOT ZERO, REPEAT
       B    @PXKEY       BRANCH TO "WAIT FOR KEY"

* PART TWO - AN XB PROGRAM
* THAT EMULATES THE ABOVE RANDOM NUMBER
* ALGORITHM FOR NUMBERS 0-10
* YOU CAN RUN THIS TO SEE THAT THE
* PROCESS REALLY WORKS

10 RANDOMIZE
20 CALL CLEAR :: DIM A(10)
30 FOR I=0 TO 10 :: A(I)=I
40 NEXT I
50 FOR I=10 TO 0 STEP -1
60 X=RND*I
70 PRINT " "&STR$(A(X));
90 A(X)=A(I)
100 NEXT I :: PRINT
110 CALL KEY(0,K,S)
120 IF S<1 THEN 110
130 IF K<>13 THEN 30
*
* PART THREE - ANOTHER XB PROGRAM
* THAT EMULATES THE ABOVE RANDOM NUMBER
* ALGORITHM FOR NUMBERS 0-191
*
10 RANDOMIZE
20 CALL CLEAR :: DIM A(191)
```

```
30 FOR I=0 TO 191 :: A(I)=I
40 NEXT I
50 FOR I=191 TO 0 STEP -1
60 X=RND*I
70 PRINT " "&STR$(A(X));
80 A(X)=A(I)
90 NEXT I
```

1.53. The Art Of Assembly — Part 53. Yes You Can!

By Bruce Harrison

If there's one danger in becoming a "guru" in Assembly, it's this: People are always asking me questions, and expecting that I'll know the answers. Sometimes it's easy to answer, and other times it takes whole days of experimenting with the TI to find the solution. Today we'll start with a "case in point" of a question and its answer.

Our dear friend Dr. Charles Good asked a question about our Compiler. He wanted to know whether a user could make a compiled program work with additional Assembly routines. At first, we thought no, and told him so. That answer turned out to be wrong, but we didn't know that for some months.

1.53.1. Finding Out

Some months after answering the question for Dr. Good, we found a need in our own work to compile a program that included the loading and use of an Assembly routine. The program was a demo for one of our Assembly routines, but we wanted to know how the program would perform when Compiled. It had been many months since we'd used the Compiler, or worked on it, so we weren't sure what it would do with a line like:

```
10 CALL INIT :: CALL LOAD("DSK1.ACCTIME/O")
```

We had a vague memory of having put a provision into the Compiler so it would simply ignore CALL INIT. That was put in because the Compiled program performs a CALL INIT when it starts, and would quit working if another CALL INIT got performed while it was running. Sure enough, the Compiler did indeed ignore the CALL INIT part of line 10. It then included the rest of line 10 with a BL @TOGI instruction, so that the CALL LOAD. . . would be performed by the GPL Interpreter. That looked as if it would work, and indeed it did! When the Compiled program started running, it lit up drive 1, loaded the routine ACCTIME/O, and then went on with its job. The CALL LINKs within the program that used ACCTIME/O also worked just as they had in the original XB program. Neat!

1.53.2. Don't Thank Me!

Thank Harry Wilhelm, who designed the very clever High Memory Loader program. As you know, Harry is the guy I call on when I need help, and he almost always has a ready and correct answer. Harry designed the HML program so that things like my compiled programs could contain Assembly content without tying up any of the low memory space. Thus when that CALL LOAD got executed, the content of the object file got placed into Low Memory, just as if a "normal" XB program were running. Harry also designed his HML so that, if the item needed for a CALL LINK was not found in his lookup list in High Memory, it would be sought in the "normal" DEF table in low memory and executed. Thanks, again, Harry! Your genius never ceases to amaze us.

1.53.3. Not Only That, But. . .

Having proved that this method would indeed work provided that the object file was available on the right disk, we wondered if we could take advantage of another of the features of Harry's HML. That is the ability to load more than one object file into High Memory. In the usual case with the compiler, we load only the compiler's object file, but HML will allow us to add more object files to the list before loading. We decided that this was worth trying, so we removed line 10 from the original program, Compiled and Assembled as usual, and then during the HML operation, we added ACCTIME/O along with the Compiler's object file. Sure enough, HML loaded both files into high memory, and the resulting program worked exactly as we'd hoped. Now we don't have to worry about having ACCTIME/O on the disk, since its contents are "embedded" right inside the compiled program.

1.53.4. There Are Exceptions!

This section could be called "NO YOU CAN'T"! There are some Assembly routines that won't work with compiled programs. One example that immediately comes to mind is one of our own inventions, namely the CALL FILES routine that we wrote to allow CALL FILES to be done from within an XB program. That would fail to work because the XB program gets told by the routine to RUN (line number), and the line number in all likelihood will not be there. It's possible to use the CALL FILES linkage by changing the second parameter in CALL LINK("CALLF". . .) to 32767 before saving the XB program in merge format. We've tried that, and it worked on our demo program. Another clear exception is our boot-tracking routine (TRACK4), which searches for DSK1 in the XB program. That won't work because it has no way of searching through the Assembly part of the compiled program. There may be other Assembly routines that won't work with a Compiled program, but we've no way of knowing what will and won't work unless people try things and let us know.

1.53.5. Another Look At Things

As most of our readers know, from time to time we take another look at old routines that we designed and update them. In recent months, we've been playing various "games" with the concepts of User Interrupt and ways of counting time in 60ths of a second. The idea came to us that we could perhaps make a new and improved version of our screen input routine CRSIN. That was published many moons ago, and we still use it with slight variations in most of our programs. It works well, but when run on a Geneve or a bus-modified TI, the cursor blinks much too fast. Could we use the knowledge we'd gained about interrupts to cure that problem? Yes, we could!

1.53.6. The New ACCEPT Routine

The idea was simple enough. We'd use a USRINT to check the state of the counter at >8379, and switch from cursor to screen character after every 20 60ths of a second. In order to make this work as we wanted, we did some tricks that may appear strange at first sight. For example, we used a BLWP vector out of the USRINT that uses our own workspace for the BLWP. This was possible because the Interrupt servicing happens in another workspace, so there's no need to supply another set of registers for the BLWP that changes the character on screen. In today's Sidebar is a complete program that uses ACCEPT to take input from the screen into a string in memory, then displays that string.

1.53.7. While We're At It . . .

While making this new routine, we added some features that were not in the original. For example, the DATA following the BL @ACCEPT determines the screen position, the location in memory into which the string is reported, and whether or not the field is to be cleared upon entry. We also added an erase feature with **FCTN 3** (or **FCTN 8**) that can clear the field while the routine is running. We kept the ideas of repeat-key action for moving the cursor left and right, with a delay before repeat starts, and used the interrupt count for that too. Thus the delay before repeat starts and the repeat rate itself are independent of the speed of the processor.

The result is a better subroutine. It's simpler than the old one, does not require self-modifying code, and reacts faster to keystrokes. As before, **FCTN 1** and **FCTN 2** perform delete and insert, much the way the TI built-in line editors do, except that the delete key does not repeat. Our students may want to make some changes, and of course that's fine. You might, for example, want to make **FCTN E** or **FCTN X** exit from the routine, by extending the FUNCT series of labels. You might also want **FCTN 8** to exit instead of just replicating what **FCTN 3** does. There you go, another exercise for the serious student. At present, other FCTN key combinations, such as **FCTN 4**, **5**, **6**, or **7** have no effect. Same goes for **FCTN E** and **X**.

1.53.8. How Does It Work?

You think we know! Take a look at the Sidebar, which has lots of annotation, and maybe you could tell us. Just kidding! Here's how it works. The very first thing the routine does is to get its "parameters" from the DATA following the BL. That starts with getting the desired screen position. MOV *R11+,R0 gets the desired position into R0. Since this routine is going to put an edge character just before the first input position, it will check to see if your desired position is the screen origin, and will move one byte to the right so there's room for the edge character ahead of the first character acceptance position. Next, it gets the maximum number of characters allowed into R2, the "clear field" signal into R3, and the address for the output string into R9. The address used as the fourth parameter should have room for the number of characters allowed plus one, for the length byte. Here, we've allowed a BSS of 30 at TEMSTR, but 29 would work, since we've limited the input field to 28 characters. Now the routine clears its insert flag, then stashes away the parameters for starting screen position and number of characters into R7 and R4, for use later. It next points back one spot on the screen and places an edge character there. This is done so we can prevent the cursor from moving outside the field. R0 gets INCed, then R2 added, and another edge character gets placed one spot beyond the last allowed input spot. We set R6 so it points to that last spot in the input field. We'll need that later, too. At CLRSNS, we get back the starting position into R0, then check R3. If R3 is zero, we won't clear the input field. If R3 is anything other than zero, we will clear the field. That's done using a simple loop, which you can easily decipher.

At this point, we're through with all the preliminaries, and ready to actually start accepting keystrokes. The first order of business, then, at label KEYFRC, is to find out what character is already there at the first position in the field. We'll save that character at ALTKEY, so it can alternate with the cursor when the blink is happening. Now we put the cursor's ASCII value (30) into the left byte of R1, and write that to the screen. To make the blinking start, we clear the counter at >8378, then enable our USRINT routine by placing its address at >83C4.

1.53.9. The Magic Part

In the five lines starting at KEYIN, all the magic part happens. Notice that, after scanning the keyboard, we use LIMI 2 and LIMI 0 to briefly allow interrupts during each pass through this five-line loop. Because interrupts are being allowed, the timer at >8379 gets incremented once every 60th of a second, and our USRINT also executes once every 60th to check the state of that counter. When USRINT finds that the counter is equal to or greater than 20, it will invoke the changing code by BLWP @CHVECT. That vector uses our own workspace (WS) and the code starting at CHG1. There, we read the character from the current screen position. If that's a cursor, we replace it with the character from ALTKEY. If it's not a cursor, we put the cursor at that position. Using our own workspace for this vector may seem strange, but it saves our having to set aside another workspace just for this purpose. The only drawback is that while the interrupt is active, we can't use R13, R14, and R15, since those get changed by the BLWP action.

If you'd like, you could set aside another workspace somewhere in the data section like this:

```
CHGWS       BSS  32
```

Then you'd have to change CHVECT to DATA CHGWS,CHG1, and change the start of CHG1 like this:

```
CHG1        MOV  @WS,R0
            BLWP @VSBR
```

The rest could stay unchanged. That would work just as well as what we've shown, and then you could keep R13, R14, and R15 of the workspace at WS available for other things. Of course most of the registers at CHGWS would go unused, but perhaps you could use them in some other vector within your program.

That's how the magic happens. The cursor blinking is carried out in the background, without needing attention from the main part of the subroutine. Just for the fun of it, we've added a part to the main program in the Sidebar so that, after you've "accepted" a string through the routine and then displayed that, we activate the USRINT before the BL @KEYLOO near the end of the main program. Sure enough, while the program waits for a keystroke, the cursor blinks near the bottom of the screen.

1.53.10. The Cautions

Be a bit careful about using this idea. Notice that in some places here we've put in a CLR @>83C4. That's done to de-activate the USRINT while we're doing things like, for example, delaying for repeat-key operation. (We don't want the cursor blinking by itself during that time.) Also, we've taken care to disable USRINT before exiting back to E/A. If we forget that, the cursor will stay there blinking while the PRESS ENTER TO CONTINUE message is on-screen, and even after that.

We've found that having a user interrupt activated can interfere with certain other functions, particularly if we're in the Extended Basic environment. There, for example it messes up the action of ON BREAK NEXT, among other things. In all cases, then, you should make sure that the user interrupt is activated only while needed.

Next month, another surprise topic. We're taking some time off from writing these, so we can get other things done. One could say we're busy servicing an interrupt.

```
* SIDEBAR 53
*
* A STRING INPUT FIELD USING THE USER
* INTERRUPT TO CONTROL CURSOR BLINK.
* A COMPLETE PROGRAM
*   Code by Bruce Harrison
*   21 December 1994
*   PUBLIC DOMAIN
*
       REF  VSBW,VSBR,VMBW,VMBR,KSCAN  REF UTILS
       DEF  START        DEFINE ENTRY
*
* REQUIRED EQUATES
*
STATUS EQU  >837C        GPL STATUS BYTE
KEYADR EQU  >8374        KEY-UNIT
KEYVAL EQU  >8375        KEY VALUE
*
* MAIN CODE SECTION
*
START  LWPI WS           LOAD OUR WORKSPACE
       CLR  @KEYADR      CLEAR KEY-UNIT
RESTR  BL   @ACCEPT      USE ACCEPT SUBROUTINE
       DATA 11*32+2      SCREEN POSITION R12, C3
       DATA 28           FIELD LEN
       DATA 0            0 - DON'T CLEAR  1 - CLEAR FIELD
       DATA TEMSTR       STRING DESTINATION
       LI   R0,13*32+2   ROW 14, COL 3
       LI   R1,TEMSTR    STRING JUST ACCEPTED
       BL   @DISSTR      DISPLAY THAT
       LI   R0,19*32+5   ROW 20, COL 6
       LI   R1,PAK       "PRESS ANY KEY"
       BL   @DISSTR      DISPLAY THAT
       A    R2,R1        NEXT STRING
       LI   R0,21*32+4   ROW 22, COL 5
       BL   @DISSTR      DISPLAY "OR FCTN-8"
       LI   R0,23*32+15  ROW 24, COL 16
       CLR  @>8378       CLEAR TIMER
       MOVB @CURSOR,R1   CURSOR CHAR
       BLWP @VSBW        ON SCREEN
       MOV  @INTLOC,@>83C4 ENABLE USER INTERRUPT
       MOVB @ANYKEY,@ALTKEY ALTERNATE SPACE
       BL   @KEYLOO      USE KEY LOOP
       CLR  @>83C4       STOP USRINT
       MOVB @ANYKEY,R1   SPACE IN R1
       BLWP @VSBW        WRITE THAT
       CI   R8,6         WAS FCTN-8 STRUCK?
```

```
       JEQ  RESTR        IF SO, GO BACK
       LWPI >83E0        GPL WORKSPACE
       B    @>6A         TO GPL INTERPRETER
*
* SUBROUTINES
*
ACCEPT MOV  *R11+,R0     R0 HAS START POSITION
       JNE  GETLEN       IF NOT 0, JUMP
       INC  R0           ELSE POINT AT 1
GETLEN MOV  *R11+,R2     R2 HAS MAX LENGTH
       MOV  *R11+,R3     R3 HAS CLEAR FIELD SIGNAL
       MOV  *R11+,R9     R9 HAS STRING DESTINATION
       CLR  @INSFLG      NOT IN INSERT
       MOV  R0,R7        SAVE START POSITION
       MOV  R2,R4        SAVE LENGTH
       DEC  R0           POINT ONE BACK
       MOVB @EDGE,R1     EDGE CHARACTER
       BLWP @VSBW        WRITE A BYTE
       INC  R0           BACK TO START
       A    R2,R0        ADD LENGTH
       MOV  R0,R6        SAVE THAT POSITION
       DEC  R6           LAST ALLOWED
       BLWP @VSBW        WRITE EDGE CHAR
CLRSNS MOV  R7,R0        BACK TO START
       MOV  R3,R3        CHECK SIGNAL
       JEQ  KEYFRC       IF ZERO, JUMP
       MOV  R4,R2        GET LENGTH BACK IN R2
       MOVB @ANYKEY,R1   SPACE CHAR
CLRFLD BLWP @VSBW        WRITE ONE SPACE
       INC  R0           MOVE AHEAD ONE
       DEC  R2           DEC COUNT
       JNE  CLRFLD       IF NOT 0, RPT
       MOV  R7,R0        GET START BACK
*
* KEYFRC GETS THE CURRENT CHARACTER
* FROM THE SCREEN, FORCES THE CURSOR
* TO THAT POSITION, THEN ACTIVATES THE
* USER INTERRUPT TO BLINK CURSOR
*
KEYFRC BLWP @VSBR        READ BYTE AT R0 POSITION
       MOVB R1,@ALTKEY   PLACE AT ALTKEY
       MOVB @CURSOR,R1   PUT CURSOR IN R1
       BLWP @VSBW        WRITE CURSOR
       CLR  @>8378       CLEAR TIME COUNTER
       MOV  @INTLOC,@>83C4 ENABLE USER INTERRUPT
*
* KEYIN IS THE PART THAT GETS KEYSTROKES
*
KEYIN  BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
```

```
       LIMI 0            STOP THEM
       CB   @STATUS,@ANYKEY KEY STRUCK?
       JNE  KEYIN        IF NOT, REPEAT
*
* FOLLOWING CODE USES THE KEYSTROKE
*
       MOV  @KEYADR,R8   KEY AS WORD IN R8
       MOVB @ALTKEY,R1   OLD CHAR IN R1
       BLWP @VSBW        WRITE TO SCREEN
       CB   @KEYVAL,@ENTERV "ENTER" STRUCK?
       JEQ  KEYEX        IF YES, EXIT
       CB   @KEYVAL,@BACKUP FCTN-S?
       JNE  KEY0         IF NOT, JUMP
*
* FOLLOWING IS CODE THAT HANDLES FCTN-S
* IT MOVES CURSOR ONE SPOT, THEN GOES TO
* RPTKEY, WHICH DELAYS BEFORE ALLOWING REPEAT
*
       DEC  R0           DEC SCRN POSITION
       BLWP @VSBR        READ BYTE
       CB   R1,@EDGE     EDGE CHARACTER?
       JNE  BCKX         IF NOT, JUMP
       INC  R0           ELSE INC POSITION
       JMP  KEYFRC       THEN BACK
BCKX   B    @RPTKEY       AHEAD FOR REPEAT ACTION
KEY0   CB   @KEYVAL,@FWARD FCTN-D?
       JNE  KEY1         IF NOT, JUMP AHEAD
*
* FOLLOWING IS CODE THAT HANDLES FCTN-D
* IT MOVES CURSOR ONE SPOT, THEN GOES TO
* RPTKEY, WHICH DELAYS BEFORE ALLOWING REPEAT
*
       INC  R0           POINT AHEAD
       BLWP @VSBR        READ BYTE
       CB   R1,@EDGE     EDGE CHAR?
       JNE  FWKX         IF NOT, JUMP
       DEC  R0           ELSE POINT BACK
       B    @KEYFRC      THEN BRANCH BACK
FWKX   JMP  RPTKEY       AHEAD FOR REPEAT ACTION
KEY1   CI   R8,32        COMPARE TO SPACE BAR
       JLT  FUNCT        IF LESS, CHECK FOR FUNCT
*
* FOLLOWING HANDLES KEY VALUES 32 AND ABOVE
*
       MOV  @INSFLG,R1   INSERT MODE?
       JEQ  KEY1A        IF NOT, JUMP AHEAD
*
* FOLLOWING HANDLES INSERT IF IN INSERT MODE
*
       C    R0,R6        AT END OF FIELD?
```

```
       JEQ  KEY1A        IF SO, NO INSERT
       MOV  R6,R2        GET LAST POSITION
       S    R0,R2        SUBTRACT CURRENT POSITION
       MOV  R9,R1        USE ASSIGNMENT SPACE
       BLWP @VMBR        PUT BYTES THERE
       INC  R0           POINT AHEAD ONE
       BLWP @VMBW        WRITE THERE
DEC0   DEC  R0           BACK TO OLD POSITION
       JMP  KEY1A        PUT IN THE KEYSTROKE
*
* FOLLOWING HANDLES FUNCTION KEYS WITH VALUES BELOW 32
*
FUNCT  CB   @KEYVAL,@DELKEY DELETE KEY?
       JNE  FUNCT2       IF NOT, JUMP
*
* FOLLOWING HANDLES DELETE WITH FCTN-1
*
       MOV  R0,R3        STASH AWAY R0
       MOV  R6,R2        GET END OF FIELD
       S    R0,R2        SUBTRACT CURRENT POSITION
       JEQ  NULDEL       IF ZERO, JUMP AHEAD
       INC  R0           ELSE POINT AHEAD ONE
       MOV  R9,R1        POINT AT ASSIGNMENT PLACE
       BLWP @VMBR        READ TO THERE
       DEC  R0           POINT BACK ONE
       BLWP @VMBW        WRITE TO THERE
NULDEL MOV  R6,R0        GET END OF FIELD
       MOVB @ANYKEY,R1   SPACE CHAR
       BLWP @VSBW        WRITE A SPACE
       MOV  R3,R0        GET OLD POSITION BACK
       JMP  KEYFRC       JUMP TO GET NEXT KEY
FUNCT2 CB   @KEYVAL,@INSKEY FCTN-2 PRESSED?
       JNE  FUNCT3       IF NOT, JUMP
*
* FOLLOWING SETS INSERT MODE ON FCTN-2
*
       INC  @INSFLG      SET INSERT FLAG
       JMP  KEYFRC       THEN BACK
FUNCT3 CB   @KEYVAL,@ERSKEY FCTN-3 PRESSED?
       JNE  FUNCT9       IF NOT, JUMP
*
* FOLLOWING ERASES FIELD IF FCTN-3 STRUCK
*
ERSFLD MOVB @ANYKEY,R3   SET R3 NON-ZERO
       B    @CLRSNS      BRANCH TO CLEAR FIELD
*
* FCTN-9 EXITS FROM ROUTINE
*
FUNCT9 CI   R8,15        FCTN-9?
       JEQ  KEYEX        IF SO, EXIT ROUTINE
```

```
*
* FCTN-8 CAUSES ERASE OF FIELD
*
       CI   R8,6          FCTN-8?
       JEQ  ERSFLD        IF SO, ERASE
       B    @KEYFRC       ELSE IGNORE KEYSTROKE
*
* FOLLOWING PUTS CURRENT KEYSTROKE ON SCREEN
* THEN MOVES CURSOR TO NEXT SPOT
*
KEY1A  MOVB @KEYVAL,R1    GET KEY VALUE IN R1
       BLWP @VSBW         WRITE THAT
       INC  R0            POINT AHEAD
       BLWP @VSBR         READ A BYTE
       CB   R1,@EDGE      EDGE?
       JNE  KEY1X         IF NOT, OKAY
       DEC  R0            POINT BACK
KEY1X  B    @KEYFRC       THEN BRANCH BACK
*
* KEYEX IS THE EXIT FROM THIS ROUTINE
*
KEYEX  CLR  @>83C4        KILL USER INTERRUPT
       MOV  R4,R2         GET LENGTH
       MOV  R6,R0         AND LAST POSITION
RDBYT  BLWP @VSBR         READ A BYTE
       CB   R1,@ANYKEY    SPACE?
       JNE  RDSTR         IF NOT, JUMP
       DEC  R0            ELSE DEC POSITION
       DEC  R2            AND CHAR COUNT
       JNE  RDBYT         IF NOT ZERO, GO BACK
RDSTR  MOV  R9,R1         GET STRING LOCATION
       MOV  R7,R0         AND START POSITION
       SWPB R2            SWAP BYTES
       MOVB R2,*R1+       PUT LENGTH BYTE AT STRING LOCATION
       JEQ  NULSTR        IF ZERO, JUMP
       SWPB R2            SWAP R2 AGAIN
       BLWP @VMBR         READ STRING CONTENT
NULSTR RT                 RETURN TO CALLER
*
* UPON EXIT, THE ENTRY IS PLACED AS A STRING WHERE ASSIGNED,
* AND REGISTER 8 HAS THE KEYSTROKE THAT CAUSED THE EXIT
*
*
* FOLLOWING IS THE REPEAT-KEY FUNCTION FOR LEFT AND RIGHT
* MOVEMENT OF THE CURSOR
*
RPTKEY BLWP @VSBR         READ CURRENT CHAR
       MOVB R1,@ALTKEY    PLACE AT ALTKEY
       MOVB @CURSOR,R1    GET CURSOR
       BLWP @VSBW         WRITE THAT
```

```
       CLR  @INSFLG      CLEAR INSERT MODE
       CLR  @>8378       CLEAR TIMER
       CLR  @>83C4       DISABLE USRINT
*
* THE LOOP STARTING AT RPT1 DELAYS REPEAT MOTION FOR
* 32/60THS OF A SECOND UNLESS KEY IS RELEASED
*
RPT1   BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTS
       LIMI 0            STOP INTS
       CB   @KEYVAL,@NOKEY NO KEY?
       JEQ  RPTEX        IF SO, EXIT
       CB   @>8379,@ANYKEY COMPARE TO 32
       JLT  RPT1         IF LESS, JUMP
RPT1A  CLR  @>8378       CLEAR TIMER
       MOVB @ALTKEY,R1   GET ALTKEY BACK
       BLWP @VSBW        WRITE
       CB   @KEYVAL,@BACKUP BACKWARD?
       JNE  RPTF         IF NOT, JUMP
       DEC  R0           ELSE BACK ONE
       BLWP @VSBR        READ CHAR
       CB   R1,@EDGE     IS THAT EDGE CHAR?
       JNE  RPTF1        IF NOT, JUMP
       INC  R0           PUT POSITION BACK
       JMP  RPTEX        THEN EXIT
RPTF   INC  R0           AHEAD ONE
RPTF1  BLWP @VSBR        READ CHAR
       CB   R1,@EDGE     EDGE?
       JNE  RPTFA        IF NOT, JUMP
       DEC  R0           BACK ONE
       JMP  RPTEX        THEN EXIT
RPTFA  MOVB R1,@ALTKEY   STASH CURRENT CHAR
       MOVB @CURSOR,R1   CURSOR IN R1
       BLWP @VSBW        WRITE CURSOR
*
* THE LOOP AT RPT2 DELAYS 8/60THS UNLESS KEY IS RELEASED
*
RPT2   BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            INTS ON
       LIMI 0            THEN OFF
       CB   @KEYVAL,@NOKEY NO KEY?
       JEQ  RPTEX        IF SO, EXIT
       CB   @>8379,@BACKUP COMPARE TO 8
       JLT  RPT2         IF LESS, REPEAT
*
* AFTER 8/60THS, CURSOR ADVANCES ANOTHER STEP
*
       JMP  RPT1A        ELSE JUMP BACK
RPTEX  MOVB @ALTKEY,R1   OLD CHAR
       BLWP @VSBW        WRITE THAT
```

```
       B    @KEYFRC      THEN BRANCH BACK
*
* FOLLOWING IS THE "BLINK", DONE WITH USER INTERRUPT
* EVERY 20 60THS, THIS WILL BLWP @CHVECT TO CHANGE
* FROM CURSOR TO CHARACTER OR VICE VERSA
*
USRINT CB   @>8379,@TWENTY TIMER=20?
       JLT  INTEX        IF LESS, EXIT
       BLWP @CHVECT      ELSE CHANGE CHAR
INTEX  RT                RETURN TO INTERRUPT HANDLER
*
* CHVECT CHANGES FROM CURSOR TO CHAR AND VICE VERSA
* EVERY 20/60THS OF A SECOND.  (THAT'S 1/3 SECOND)
*
CHVECT DATA WS,CHG1      OUR OWN WORKSPACE, CHANGE CODE
CHG1   BLWP @VSBR        READ CURRENT BYTE FROM SCREEN
       CB   R1,@CURSOR   IS THAT CURSOR?
       JEQ  CHG2         IF YES, JUMP
       MOVB @CURSOR,R1   ELSE GET CURSOR
       BLWP @VSBW        AND WRITE THAT
       JMP  CHGX         THEN EXIT
CHG2   MOVB @ALTKEY,R1   PUT OLD CHAR IN R1
       BLWP @VSBW        WRITE THAT
CHGX   CLR  @>8378       CLEAR TIMER
       RTWP              THEN RETURN
*
* DISSTR DISPLAYS A STRING ON SCREEN
*
DISSTR MOVB *R1+,R2      GET LENGTH BYTE
       SRL  R2,8         RIGHT JUSTIFY
       JEQ  DISX         IF ZERO, EXIT
       BLWP @VMBW        ELSE WRITE STRING
DISX   RT                RETURN
*
* KEYLOO WAITS FOR A KEYSTROKE, THEN RETURNS
* IN THIS INSTANCE, WE'VE MADE THE CURSOR BLINK
* WHILE KEYLOO IS EXECUTING.
*
KEYLOO BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTS
       LIMI 0            THEN STOP
       CB   @STATUS,@ANYKEY ANY KEY?
       JNE  KEYLOO       IF NOT, REPEAT
       MOV  @KEYADR,R8   KEY AS WORD INTO R8
       RT                THEN RETURN
*
* DATA SECTION
*
WS     BSS  32           OUR WORKSPACE
INTLOC DATA USRINT       USER INTERRUPT ADDRESS
```

```
INSFLG DATA 0            INSERT FLAG
DELKEY BYTE 3            FCTN-1 VALUE
INSKEY BYTE 4            FCTN-2 VALUE
ERSKEY BYTE 7            FCTN-3 VALUE
TEMSTR BSS  30           TEMPORARY STRING
ALTKEY BYTE 0            CURRENT CHARACTER FROM SCREEN
ENTERV BYTE 13           ENTER KEY VALUE
CURSOR BYTE 30           CURSOR CHAR
BACKUP BYTE 8            FCTN-S
FWARD  BYTE 9            FCTN-D
ANYKEY BYTE 32           SPACE OR COMPARISON BYTE
TWENTY BYTE 20           CURSOR BLINK NUMBER
NOKEY  BYTE >FF          NO KEY INDICATION
EDGE   BYTE 31           EDGE CHAR
PAK    BYTE 21
       TEXT 'PRESS ANY KEY TO EXIT'
OR8    BYTE 19
       TEXT 'OR FCTN-8 TO REPEAT'
       END
```

1.54. The Art Of Assembly — Part 54. More Random Numbers

By Bruce Harrison

This will make the third column in this series that deals with Random Numbers. We keep trying new things all the time, though, and sharing what we learn with you, so bear with us, we think this will be worthwhile, especially for those who want to use Random Numbers in Extended Basic. As you no doubt know, making random numbers in XB can take a lot of time. Provided that what you want is random integers in an array variable, the source in today's Sidebar will be a great help.

1.54.1. Without Replacement

What do we mean by "without replacement? Just this: Each number will be included once and only once. Let's say you grab a deck of cards, shuffle them, and then pick a card randomly from the deck. Say that's the three of hearts. If you set that aside and pick another card from the deck, you can be certain it won't be the three of hearts. In the case of the first two routines in today's Sidebar, if you ask for 500 numbers, you can be sure that no number will be repeated. Two months ago, we showed a version of this derived from our Titler program. That one dealt only with single-byte numbers, so it wouldn't do as a general purpose random number routine.

What we set out to do here was to offer a routine that would make up to 500 random numbers without replacement, and would assign them into an XB Array Variable. It had to do this very quickly, and it does. An array of dimension 500 can be filled with random numbers in less than a second. Each number from 1 through 500 will be there, just once. In case you've any doubts, there's a second entry point in this first routine called CHECK, which will check each number against all the rest, just to insure that none are repeated. Checking takes longer than making, so you wouldn't want to leave the CHECK in a finished program, but it's there so you can test with it, to be confident in the results. If CHECK finds a duplicated number in the list, it will stop the XB program with a "BAD VALUE IN XXX" report and the "BOOP" sound, just as if the error were in the XB part of the program. (In hundreds of test runs this hasn't happened to us!)

1.54.2. The CALL LINK

In this case, the CALL LINK may have two, three, or four parameters, depending on what you want to do with the numbers, and on whether or not you're using OPTION BASE 1. The two minimum parameters are the number of numbers you want and the variable name for the array. Let's say, for example, that we want 52 numbers, we've DIMed a variable to 51, and that's called A(). To perform the assignment, we do this:

```
CALL LINK("RANDOM",52,A())
```

When that finishes (in very little time) the array members from A(0) through A(51) will contain the numbers 0 through 51 in random order. These could be used as a shuffled deck of cards, for example, with 0 through 3 being aces, 4 through 7 being deuces, and so on. That could really speed up many card-playing games in XB. Using the 0th element in the array makes best use of memory, and that's why we only had to DIM A(51) instead of (52).

But suppose you have trouble dealing with the value 0, even if only as a "mental block" because people start counting with 1, not zero. You might still want to use the 0th element, but want the numbers in the variable to run from 1 through 52, not 0 through 51. Can do! That's what the third parameter is for in the CALL LINK. You can do this:

```
CALL LINK("RANDOM",52,A(),1)
```

Sure enough, the numbers in your array will range from 1 through 52. If you like, you could put 1001 in that third parameter, and the numbers in A(0) through A(51) will be from 1001 through 1052. You say you're still not happy, and you want to use OPTION BASE 1 like you always do? Yes, Virginia, there is a Santa Claus. You can do this in your XB program:

```
10 OPTION BASE 1
20 DIM A(52)
30 CALL LINK("RANDOM",52,A(),1,1)
```

The fourth parameter is necessary if you've done that OPTION BASE 1 operation. The third parameter may be zero, so the numbers themselves would start at zero, but the fourth parameter has to be 1 for the OPTION BASE 1 case.

1.54.3. The Second Routine

In case you're running short of low memory for Assembly routines, there's a shorter version which eliminates the CHECK capability and also eliminates a block of 500 words in memory. This routine, called RAND3, does exactly the same job as the bigger one in less memory, because it doesn't have the CHECK routine and doesn't need to keep the numbers available for checking. The CALL LINK for this works exactly like the previous case.

1.54.4. And Yet Another

It occurred to us that in some cases you just need a bunch of random numbers in a specific range, and don't care if numbers get repeated. For that case, we've included RAND4. RAND4 can make any number of random numbers you might want, provided only that the array variable has been DIMed large enough to hold that many numbers. This routine needs more parameters, as the range must be specified by lower and upper limits. (In the "without replacement cases, the range was inherently specified.) With RAND4, you're completely free. The CALL LINK might look like this:

```
CALL LINK("RANDOM",B,X,Y,A())
```

Where B is the number of integer random numbers to be created, X is the lower limit of the range, Y is the higher limit of the range, and A() is the array variable into which the numbers will go, starting with A(0) and running through A(B-1). The limits are these: B cannot be larger than the DIM of A() plus 1; X must be less than Y. Either X or Y or both can be negative numbers, as long as X is a smaller (or more negative) number than Y. There's no checking available, but you can easily check the numbers in XB, run a few dozen times, then take out the checking part when you're convinced the routine always works as advertised. Like its companion pieces, this routine works very fast. In our test program (see Sidebar) we've made it assign 1000 random numbers, and that happens in less than two seconds. That test also includes a checking process, which insures us that the limits given have not been exceeded, and shows us where the limiting numbers themselves showed up in the sequence. Having run this test many, many times, we're confident that no number less than X nor more than Y gets generated. Of course now and then neither limit will show up, or else the limits may show up four or five times in any one run, but that's to be expected. The important thing is that the numbers generated are nicely scattered and never exceed the limits.

1.54.5. The Outer Limits

As for any case involving one-word integers, the limiting numbers on limits are -32768 and +32767. It's unlikely you'd use those, but the routine will still work if you do! For the previous cases, where numbers are chosen without replacement, the limits on that third parameter are the same, except that the higher positive limit must be 32767 minus the first parameter, else you'll get screwy results, with the numbers "wrapping around" to negative ones.

1.54.6. That Fifth Parameter

As with the other routines, this one may take an extra parameter. If OPTION BASE 1 is in effect, there must be a fifth parameter, and its value must be at least 1. There are other uses for this parameter, as is also true of the fourth parameter on the RAND2 and RAND3 routines. You can use this fifth (or fourth, as appropriate) parameter to partially fill an array. Let's take an example to show what this means. Suppose you have a case where the array is DIMed at 999, but you want to fill only 500 elements, starting at number 300, with numbers in the range of -250 to 250, inclusive. This can be done as follows:

```
CALL LINK("RANDOM",500,-250,250,A(),300)
```

After this executes, the A() array will have random numbers in it, starting at A(300) and running through A(799). Other members of the array will be unaffected. The same can be done with the earlier routines, but using the fourth parameter instead of the fifth. To be perfectly honest about this, we're not entirely sure why anybody would want to do this, but since the extra parameter was needed in the case of OPTION BASE 1, we decided to make it do extra duty by offsetting the base to which numbers get assigned.

1.54.7. The DEMO Programs

There are four demo programs listed in the Sidebar, and these should be available as programs, along with their respective object files, on the disk version of *MICROpendium*. These are called CARDS, CARDS2, BIGRAND2, and BIGRAND4.

1.54.8. The Finer Points

Before using any of these, be sure to have a RANDOMIZE somewhere in the beginning of your XB program. You'll see there's a "mystery" instruction near the beginning of each routine that says A @>8378,@>83C0. This was put in to insure against a phenomenon that can happen without it. We found in testing that sometimes when the number of numbers asked for was even, (52, for example) the random numbers would fall into a pattern where the first number would always be even. It would be different for each run, but always even, and that meant that our "randomness" was lacking. Adding the number that happens to be at >8378 every time we start the routine avoids that problem, so that on repeated runs, the list will start with odd or even numbers, each about half of the time. The number in >8378 has a limited range, but it keeps changing while XB is running, so it provides an extra measure of randomness when added to the seed number at >83C0.

For those who care about such things, here's how the memory use stacks up. RAND2/O takes up the most, as it must to include the "CHECK" feature, at 2286 bytes. That of course includes 2000 bytes for the two tables of 500 words each. Next in line is RAND3/O, which has no checking, and takes only 1212 bytes, including 1000 for its one table of 500 words. Smallest of the lot is RAND4/O, which doesn't need to keep tables for its numbers. This takes a mere 179 bytes! For most purposes, we'd use RAND3/O or RAND4/O, depending whether we needed the "without replacement" feature. Any of these can be "embedded" into your XB program itself using either Todd Kaplan's ALSAVE or Harry Wilhelm's HML program.

That's all for today. The source code in the Sidebar is well annotated, so you should be able to follow what it's doing without a lengthy explanation. If you don't understand, you can get first-hand help at (301) 277-3467, any time from 9 AM through Midnight Eastern time. Bye for now.

```
* SIDEBAR 54
* FIRST, THREE ROUTINES FOR
* USE WITH XB PROGRAMS
*
* PART ONE - AN ELEGANT VERSION
*
* RAND2/S
* ASSIGNS RANDOM NUMBERS
* WITHOUT REPLACEMENT
* INTO AN XB ARRAY VARIABLE
* CAN ASSIGN UP TO 500 NUMBERS
*    CODE BY:  Bruce Harrison
*    PUBLIC DOMAIN
*    25 December 1994
*
```

```
       DEF  RANDOM,CHECK DEFINE ENTRY POINTS
*
* REQUIRED EQUATES
*
GPLWS  EQU  >83E0        GPL WORKSPACE
FAC    EQU  >834A        F. P. ACCUMULATOR
NUMREF EQU  >200C        NUMERIC REFERENCE
NUMASG EQU  >2008        NUMERIC ASSIGNMENT
XMLLNK EQU  >2018        XML LINK VECTOR
CIF    EQU  >20          CONVERT INTEGER TO F.P.
CFI    EQU  >12B8        CONVERT F.P. TO INTEGER
ERR    EQU  >2034        ERROR REPORT
ARGNUM EQU  >8312        NUMBER OF ARGUMENTS
*
*
RANDOM LWPI WS           LOAD OUR WORKSPACE
       A    @>8378,@>83C0 ADJUST SEED NUMBER
       CLR  R0           CLEAR FOR NON-ARRAY
       CLR  R14          CLEAR START NUMBER
       CLR  R15          CLEAR BASE OPTION
       LI   R1,1         FIRST PARAMETER
       BLWP @NUMREF      GET PARAMETER
       BLWP @XMLLNK      USE XML LINK
       DATA CFI          CONVERT TO INTEGER
       MOV  @FAC,R13     STASH IN REG 13
       MOV  R13,R12      PUT IN R12 ALSO
       SLA  R12,1        DOUBLE R12
       CLR  R3           CLEAR COUNT
       CB   @ARGNUM,@THREE CHECK ARGUMENTS
       JLT  RAND2        LESS THAN 3, JUMP
       LI   R1,3         PARAMETER 3
       BLWP @NUMREF      GET THAT
       BLWP @XMLLNK      USE XML
       DATA CFI          CONVERT TO INTEGER
       MOV  @FAC,R14     PLACE AS START NUMBER
RAND1  CB   @ARGNUM,@FOUR CHECK FOR FOUR
       JLT  RAND2        IF LESS, SKIP
       LI   R1,4         PARAMETER 4
       BLWP @NUMREF      GET THAT
       BLWP @XMLLNK      USE XML
       DATA CFI          TO INTEGER
       MOV  @FAC,R15     STASH AS BASE NUMBER
RAND2  LI   R9,BUFF2     POINT AT BUFF2
       MOV  R9,R10       PLACE IN R10
       A    R12,R10      ADD DOUBLED NUMBER
BRAN0  MOV  R3,*R9+      PLACE A WORD IN TABLE
       INC  R3           INC COUNT
       C    R9,R10       COMPARE POINTER
       JLT  BRAN0        IF LESS, BACK
       MOV  R3,R6        PUT R3 INTO R6
```

```
        SLA  R6,1          DOUBLE THAT
        LI   R10,BUFFER    POINT AT BUFFER
BRAN1   LI   R4,28645      BIG NUMBER IN R4
        MPY  @>83C0,R4     MULT. BY SEED
        AI   R5,31417      ADD BIG NUMBER
        MOV  R5,@>83C0     PUT BACK AT SEED
        CLR  R4            CLEAR HIGH WORD
        DIV  R3,R4         DIVIDE BY R3
        SLA  R5,1          DOUBLE REMAINDER
        MOV  @BUFF2(R5),*R10+ MOVE TABLE TO TABLE
        MOV  @BUFF2-2(R6),@BUFF2(R5) REMOVE USED ONE
        DECT R6            SUBTR. 2 FROM R6
        DEC  R3            SUBTR 1 FROM R3
        JNE  BRAN1         IF NOT ZERO, REPEAT
        LI   R10,BUFFER    POINT BACK AT BUFFER
        CLR  R0            NUMBER 0
        A    R15,R0        ADD BASE OPTION
        LI   R1,2          SECOND PARAMETER
        MOV  R13,R4        NUMBER OF NUMBERS
ASGLP   MOV  *R10+,@FAC    ONE NUMBER TO FAC
        A    R14,@FAC      ADD START NUMBER
        BLWP @XMLLNK       USE XML
        DATA CIF           TO FLOATING POINT
        BLWP @NUMASG       ASSIGN TO PARAMETER
        INC  R0            NEXT ARRAY MEMBER
        DEC  R4            DEC COUNT
        JNE  ASGLP         NOT ZERO, REPEAT
EXIT    LWPI GPLWS         LOAD GPL WS
        B    @>6A          EXIT TO GPL INT.
CHECK   LWPI WS            LOAD OUR WORKSPACE
        CI   R13,3         COMPARE TO 3
        JLT  EXIT          IF LESS, EXIT
        LI   R9,BUFFER     POINT AT BUFFER
        MOV  R13,R4        NUMBER OF NUMBERS IN R4
        MOV  R13,R5        AND R5
        DECT R4            SUBTR 2 FROM R4
        DEC  R5            AND FROM R5
RECMP   MOV  R9,R10        PUT R9 IN R10
        INCT R10           ADD 2
CMP     C    *R9,*R10+     COMPARE
        JEQ  ERROR         IF EQUAL, ERROR
        DEC  R5            DEC COUNT
        JNE  CMP           IF NON-ZERO, REPEAT
        MOV  R4,R5         PUT R4 INTO R5
        INCT R9            ADD 2 TO R9
        DEC  R4            DEC COUNT 2
        JNE  RECMP         IF <>0, NEXT CYCLE
        JMP  EXIT          FINISHED, EXIT
ERROR   LI   R0,>1E00      "BAD VALUE" MESSAGE
        BLWP @ERR          REPORT ERROR
```

```
*
* DATA SECTION
*
WS     BSS  32           OUR WORKSPACE
BUFF2  BSS  1000         SELECTION LIST
BUFFER BSS  1000         OUTPUT LIST
THREE  BYTE 3            JUST 3
FOUR   BYTE 4            JUST 4
       END
*
* PART TWO - A SIMPLER VERSION
*
*
* RAND3/S
* "COMPACT" VERSION
* ASSIGNS RANDOM NUMBERS
* WITHOUT REPLACEMENT
* INTO AN XB ARRAY VARIABLE
* CAN ASSIGN UP TO 500 NUMBERS
* NO "CHECK" FEATURE
*    CODE BY:  Bruce Harrison
*    PUBLIC DOMAIN
*    26 December 1994
*
       DEF  RANDOM DEFINE ENTRY POINT
*
* REQUIRED EQUATES
*
GPLWS  EQU  >83E0        GPL WORKSPACE
FAC    EQU  >834A        F. P. ACCUMULATOR
NUMREF EQU  >200C        NUMERIC REFERENCE
NUMASG EQU  >2008        NUMERIC ASSIGNMENT
XMLLNK EQU  >2018        XML LINK VECTOR
CIF    EQU  >20          CONVERT INTEGER TO F.P.
CFI    EQU  >12B8        CONVERT F.P. TO INTEGER
ARGNUM EQU  >8312        NUMBER OF ARGUMENTS
*
*
RANDOM LWPI WS           LOAD OUR WORKSPACE
       A    @>8378,@>83C0 ADJUST SEED NUMBER
       CLR  R0           CLEAR FOR NON-ARRAY
       CLR  R14          CLEAR START NUMBER
       CLR  R15          CLEAR BASE OPTION
       LI   R1,1         FIRST PARAMETER
       BLWP @NUMREF      GET PARAMETER
       BLWP @XMLLNK      USE XML LINK
       DATA CFI          CONVERT TO INTEGER
       MOV  @FAC,R12     STASH IN REG 12
       SLA  R12,1        DOUBLE R12
       CLR  R3           CLEAR COUNT
```

```
        CB   @ARGNUM,@THREE CHECK ARGUMENTS
        JLT  RAND2          LESS THAN 3, JUMP
        LI   R1,3           PARAMETER 3
        BLWP @NUMREF        GET THAT
        BLWP @XMLLNK        USE XML
        DATA CFI            CONVERT TO INTEGER
        MOV  @FAC,R14       R14 IS START NUMBER
RAND1   CB   @ARGNUM,@FOUR CHECK FOR FOUR
        JLT  RAND2          IF LESS, SKIP
        LI   R1,4           PARAMETER 4
        BLWP @NUMREF        GET THAT
        BLWP @XMLLNK        USE XML
        DATA CFI            TO INTEGER
        A    @FAC,R0        ADD BASE TO R0
RAND2   LI   R9,BUFF2       POINT AT BUFF2
        MOV  R9,R10         PLACE IN R10
        A    R12,R10        ADD DOUBLED NUMBER
BRAN0   MOV  R3,*R9+        PLACE R3 IN TABLE
        INC  R3             INC COUNT
        C    R9,R10         COMPARE POINTER
        JLT  BRAN0          IF LESS, BACK
        MOV  R3,R6          PUT R3 INTO R6
        SLA  R6,1           DOUBLE THAT
        LI   R1,2           PARAMETER 2
BRAN1   LI   R4,28645       BIG NUMBER IN R4
        MPY  @>83C0,R4      MULT. BY SEED
        AI   R5,31417       ADD BIG NUMBER
        MOV  R5,@>83C0      PUT BACK AT SEED
        CLR  R4             CLEAR HIGH WORD
        DIV  R3,R4          DIVIDE BY R3
        SLA  R5,1           DOUBLE REMAINDER
        MOV  @BUFF2(R5),@FAC MOVE WORD TO FAC
        A    R14,@FAC       ADD START NUMBER
        BLWP @XMLLNK        USE XML
        DATA CIF            TO FLOATING POINT
        BLWP @NUMASG        ASSIGN TO PARAMETER
        INC  R0             NEXT ARRAY MEMBER
        MOV  @BUFF2-2(R6),@BUFF2(R5) REMOVE USED ONE
        DECT R6             SUBTR. 2 FROM R6
        DEC  R3             SUBTR 1 FROM R3
        JNE  BRAN1          IF NOT ZERO, REPEAT
EXIT    LWPI GPLWS          LOAD GPL WS
        B    @>6A           EXIT TO GPL INT.

*
* DATA SECTION
*
WS      BSS  32             OUR WORKSPACE
BUFF2   BSS  1000           SELECTION LIST
THREE   BYTE 3              JUST 3
```

```
FOUR   BYTE 4            JUST 4
       END
*
* PART THREE - A "WITH REPLACEMENT"
* RANDOM NUMBER GENERATOR
*
* RAND4/S
* ASSIGNS RANDOM NUMBERS
* WITH REPLACEMENT
* INTO AN XB ARRAY VARIABLE
* CAN ASSIGN ANY NUMBER OF ELEMENTS
* NO "CHECK" FEATURE
*    CODE BY:  Bruce Harrison
*    PUBLIC DOMAIN
*    26 December 1994
*
       DEF  RANDOM DEFINE ENTRY POINT
*
* REQUIRED EQUATES
*
GPLWS  EQU  >83E0        GPL WORKSPACE
FAC    EQU  >834A        F. P. ACCUMULATOR
NUMREF EQU  >200C        NUMERIC REFERENCE
NUMASG EQU  >2008        NUMERIC ASSIGNMENT
XMLLNK EQU  >2018        XML LINK VECTOR
CIF    EQU  >20          CONVERT INTEGER TO F.P.
CFI    EQU  >12B8        CONVERT F.P. TO INTEGER
ARGNUM EQU  >8312        NUMBER OF ARGUMENTS
*
*
RANDOM LWPI WS           LOAD OUR WORKSPACE
       A    @>8378,@>83C0 ADJUST SEED NUMBER
       CLR  R0           CLEAR FOR NON-ARRAY
       LI   R1,1         FIRST PARAMETER
       BLWP @NUMREF      GET PARAMETER
       BLWP @XMLLNK      USE XML LINK
       DATA CFI          CONVERT TO INTEGER
       MOV  @FAC,R3      NUMBER OF NUMBERS IN R3
       INC  R1           2ND PARAMETER
       BLWP @NUMREF      GET PARAMETER
       BLWP @XMLLNK      USE XML LINK
       DATA CFI          CONVERT TO INTEGER
       MOV  @FAC,R15     LOWER LIMIT IN R15
       INC  R1           3RD PARAMETER
       BLWP @NUMREF      GET PARAMETER
       BLWP @XMLLNK      USE XML LINK
       DATA CFI          CONVERT TO INTEGER
       MOV  @FAC,R13     UPPER LIMIT IN R13
       INC  R13          INCREMENT FOR DIVIDES
       S    R15,R13      ADJUST BY LOWER LIMIT
```

```
       CB   @ARGNUM,@FIVE CHECK FOR 5 ARGUMENTS
       JLT  RAND2         LESS THAN 5, JUMP
       INCT R1            FIFTH PARAMETER
       BLWP @NUMREF       GET NUMBER
       BLWP @XMLLNK       USE XML
       DATA CFI           CONVERT TO INTEGER
       A    @FAC,R0       SET BASE NUMBER
RAND2  LI   R1,4          PARAMETER 2
BRAN1  LI   R4,28645      BIG NUMBER IN R4
       MPY  @>83C0,R4     MULT. BY SEED
       AI   R5,31417      ADD BIG NUMBER
       MOV  R5,@>83C0     PUT BACK AT SEED
       CLR  R4            CLEAR HIGH WORD
       DIV  R13,R4        DIVIDE BY R13
       MOV  R5,@FAC       MOVE R5 TO FAC
       A    R15,@FAC      ADD START NUMBER
       BLWP @XMLLNK       USE XML
       DATA CIF           TO FLOATING POINT
       BLWP @NUMASG       ASSIGN TO PARAMETER
       INC  R0            NEXT ARRAY MEMBER
       DEC  R3            SUBTR. 1 FROM R3
       JNE  BRAN1         IF NOT ZERO, REPEAT
EXIT   LWPI GPLWS         LOAD GPL WS
       B    @>6A          EXIT TO GPL INT.

*
* DATA SECTION
*
WS     BSS  32            OUR WORKSPACE
FIVE   BYTE 5             JUST 5
       END
*
* PART FOUR
* AN XB PROGRAM TO DEMO RAND2/O
* INCLUDING THE "CHECK" FEATURE
* PROGRAM CALLED "CARDS"
*
10 CALL INIT :: CALL LOAD("D
SK1.RAND2/O")
20 OPTION BASE 1
30 RANDOMIZE :: CALL CLEAR :
: DIM A(52)
40 PRINT "STARTING ASSIGNMEN
T" :: CALL LINK("RANDOM",52,
A(),1,1):: PRINT "ASSIGNMENT
 DONE"
50 PRINT "CHECKING" :: CALL
LINK("CHECK"):: PRINT "LIST
OKAY":"HERE'S THE LIST"
60 FOR I=1 TO 52 :: PRINT A(
```

```
I);:: NEXT I
*
* PART FIVE
* AN XB PROGRAM TO DEMO RAND3/O
* PROGRAM CALLED "CARDS2"
*
10 CALL INIT :: CALL LOAD("D
SK1.RAND3/O")
20 OPTION BASE 1
30 RANDOMIZE :: CALL CLEAR :
: DIM A(52)
40 PRINT "STARTING ASSIGNMEN
T" :: CALL LINK("RANDOM",52,
A(),1,1):: PRINT "ASSIGNMENT
 DONE"
50 PRINT "HERE'S THE LIST" :
: FOR I=1 TO 52 :: PRINT A(I
);:: NEXT I
*
* PART SIX
* AN XB PROGRAM THAT GETS
* 500 RANDOM NUMBERS
* WITHOUT REPLACEMENT
* PROGRAM CALLED "BIGRAND2"
*
10 CALL INIT :: CALL LOAD("D
SK1.RAND3/O")
20 RANDOMIZE :: CALL CLEAR :
: DIM A(499)
30 PRINT "STARTING ASSIGNMEN
T" :: CALL LINK("RANDOM",500
,A()):: PRINT "ASSIGNMENT DO
NE"
40 PRINT "HERE'S THE LIST" :
: FOR I=0 TO 499 :: PRINT A(
I);:: NEXT I
*
* PART SEVEN
* AN XB PROGRAM THAT GETS
* 1000 "WITH REPLACEMENT"
* RANDOM NUMBERS
* USING RAND4/O
* PROGRAM CALLED "BIGRAND4"
*
10 CALL INIT :: CALL LOAD("D
SK1.RAND4/O")
20 RANDOMIZE :: CALL CLEAR :
: DIM A(999)
30 N=1000 :: X=-250 :: Y=250
 :: PRINT "STARTING ASSIGNME
```

```
NT"
40 CALL LINK("RANDOM",N,X,Y,
A()):: PRINT "ASSIGNMENT DON
E"
50 PRINT "HERE'S THE LIST" :
: FOR I=0 TO N-1 :: PRINT A(
I);:: NEXT I :: PRINT
60 PRINT "CHECKING LIMITS"
70 FOR I=0 TO N-1 :: DISPLAY
 AT(24,1):I
80 IF A(I)=X THEN 90 ELSE IF
 A(I)=Y THEN 100 ELSE IF A(I
)<X OR A(I)>Y THEN 110 ELSE
120
90 PRINT X;"FOUND AT";I :: G
OTO 120
100 PRINT Y;"FOUND AT";I ::
GOTO 120
110 PRINT "ERROR";A(I);"AT";
I
120 NEXT I
```

1.55. The Art Of Assembly — Part 55. We Need Those Digits

By Bruce Harrison

Today's column is for those who've refused to buy a Pentium processor PC because the TI does floating point math more accurately. We're not sure that's true, but the TI does have an uncanny way of getting correct answers to complicated math problems. Last month we provided two very quick ways to make random numbers through Assembly language, but both of those made only integer numbers in the range -32768 through 32767. We realize there are those among our readers who might like to cover a wider span than that, and might even want to have all those many digits available. Today we're filling that need, with two routines that can make the full range of random numbers to as much precision (14 significant digits) as the TI allows. Both of these are routines designed for use with Extended Basic, to provide excellent random numbers in less time than the RND function.

1.55.1. Filling An Array

The first routine, called RANDAR, is designed to fill any or all members of an array variable with floating point random numbers in a user-selected range. Let's start with a simple case in which we want to fill an array with 300 random numbers between 50,000 and 100,000. The typical way to do that in XB is as follows:

```
10 RANDOMIZE :: DIM A(299)
20 FOR I=0 TO 299 :: A(I)=RND*50000+50000 :: NEXT I
```

That will work nicely, and will include all the digits possible, down to several decimal places. Unfortunately, running this loop will take about 25 seconds. That kind of job is what RANDAR is for. Using that routine, we'd accomplish the same thing by:

```
10 RANDOMIZE :: DIM A(299)
20 CALL LINK("RANDAR",A(),300,50000,50000)
```

That will do the same job, filling all members of the array A() with random numbers in the range 50,000 to 100,000, but it will take only about three seconds to do it! As with the RND case, the full range of digits will be preserved in the numbers. So far as we can tell from hours of testing, the randomness of the numbers will be every bit as good as those provided by RND.

1.55.2. What Else Can It Do?

Like the integer routines we provided last month, this can take additional parameters. If OPTION BASE 1 is in effect, you must add one more parameter, and that must be at least 1. You can make that parameter more than one for partial fills. Let's say you want only fifty such numbers in A(), starting at A(100). The CALL LINK would be:

```
CALL LINK("RANDAR",A(),50,50000,50000,100)
```

That will fill A(100) through A(149) with random numbers, leaving all other members of the array alone. This is similar to the partial fill feature we included in last month's column.

It may be that you really don't want those extra digits past the decimal point. Yes, we've covered that for you too. In normal XB, you'd do it by A(I)=INT(RND*50000)+50000. In the RANDAR link, it would be:

```
CALL LINK("RANDAR",A(),300,50000,50000,0,1)
```

Here we've made the fifth parameter 0, so that filling of the array will start with A(0), but added a sixth parameter. The value of that parameter is irrelevant, as the routine just looks to see whether a sixth parameter is present, and does not access the value. Given that sixth parameter, the routine will truncate the numbers to their integer part only. This truncation will work a bit differently from the INT function when negative numbers are being used. In the normal INT function, the result of INT(-1.5) will be -2. With this routine's truncation, the result would be -1. For positive numbers, there will be no difference from an INT function.

1.55.3. How Does It Work?

Okay, now it's time to look at today's Sidebar. The routine starts in the usual fashion, loading our workspace. Like last month's routines, it makes an adjustment to the random number seed, then clears R0 for the non-array parameters. Next it gets the number of arguments into R12, and right-justifies that. If that number is zero, we exit without doing anything. Given that there's at least one argument, the routine proceeds to check for and get other arguments into its own space. Those include the number of numbers to assign, which goes into R13, and the multiplier and addition numbers that go into 8-byte blocks in the data section.

The real action starts at RAND2. Here we set up for the first parameter, put 7 into R7 as a counter, set R6 to 100 for division, then point R10 at FAC+1. The code down to JNE BRAN1 now makes seven random numbers ranging from 0 through 99, and places those into the seven bytes starting at FAC+1. Thus we have a Radix 100 number equivalent to 14 digits of random values. We set the exponent at 100 raised to the -1 power by placing the number 63 in the byte at FAC. Now the number represented in the eight bytes starting at >834A is in the range 0 through 0.99999999999999.

Next, we check to see if there's a third parameter, which is the multiplier. If there is, we move that number's eight bytes into ARG and multiply using an XMLLNK service. In similar fashion, we check for the presence of the fourth parameter, move that to ARG if it's there, and use XML to add it to the multiplied number at FAC. This gets us to label BRAN2.

At BRAN2, we check for the presence of the sixth parameter, and if it's there we truncate the number in FAC to just its integer part. That gets us to BRAN3, where we assign the resulting number in FAC to the current member of the array variable, then INC R0 to point to the next member of the array, and DEC R13. If R13 becomes zero, we're finished. Otherwise, we jump back to RAND2 to set and send the next member of the array.

By doing the whole job in just one CALL LINK, we save lots of time. Compared to doing this kind of random number filling with an Extended Basic FOR-NEXT loop using RND, we get about an 8.3:1 speed advantage.

The other routine, RANDFP, is for assignment of just a single number into an XB variable. Strictly speaking, this isn't really needed, as RANDAR can do the same job by simply setting the number of numbers to be assigned at 1, and using a simple variable or particular member of an array as parameters.

The only real advantages to RANDFP are that it needs one fewer parameters and the Assembly code takes up less memory. Like RANDAR, this performs faster than RND, but it's hard to pin down the exact amount of difference, because using a FOR-NEXT to repeat the operation many times "covers" the action of the routine itself. Generally, if lots of random numbers are needed in the XB program, it would be easier and quicker to make them members of an array, and fill them using RANDAR.

We've put in some minor protection against stupid entries in the CALL LINK. If you supply no parameters to RANDAR, or only one, then it will exit without doing anything. If you set the number of numbers required at a negative number, the routine will just take the absolute value for you and make that many numbers.

Both routines are listed in the Sidebar, so our serious students can study them, make changes, and so on. Also in the Sidebar are two short Extended Basic programs which you can use to show the dramatic difference between using RND and RANDAR. Of course you'd need to assemble RANDAR to test this. We've provided the object files for those who get *MICROpendium* on disk, and the XB programs as FASTAR and SLOWAR.

The routines we've shown in both last month's column and this one are available on a Public Domain disk called RANDOMS. That can be obtained from the Lima Users' Group in the usual fashion. It includes complete instructions and the tools necessary to "embed" the routines into XB programs.

Next month perhaps we'll be off of random numbers, and perhaps not. Toss a coin for yourself to form your opinion.

```
* SIDEBAR 55
* TWO ROUTINES FOR USE UNDER
* TI EXTENDED BASIC
*
* PART ONE - FOR XB ARRAY VARIABLES
*
* RANDAR/S
* ASSIGNS RANDOM NUMBER
* IN F.P FORMAT
* INTO AN XB VARIABLE
*
*  INVOKE BY:
*  CALL LINK("RANDAR",V(),N,M,A,B,1)
*  WHERE:
```

```
*    V() IS ANY ARRAY VARIABLE NAME
*      MUST HAVE DIM (N-1) MINIMUM
*    N IS NUMBER OF RANDOM NUMBERS
*    M IS THE DESIRED MULTIPLIER (RANGE)
*    A IS THE DESIRED ADDITION (START #)
*    B IS THE OPTION BASE FOR ARRAY
*    1 (OR ANY NUMBER) MEANS INTEGERS
* ONLY V() AND N ARE ALWAYS REQUIRED
* OTHER PARAMETERS ARE OPTIONAL
* (SEE TEXT FOR DETAILS)
*
*    CODE BY:  Bruce Harrison
*    PUBLIC DOMAIN
*    27 December 1994
*
       DEF  RANDAR DEFINE ENTRY POINT
*
* REQUIRED EQUATES
*
GPLWS  EQU  >83E0        GPL WORKSPACE
FAC    EQU  >834A        F. P. ACCUMULATOR
FMUL   EQU  >0E88        F. P. MULTIPLY
FADD   EQU  >0D80        F. P. ADDITION
ARG    EQU  >835C        F. P. ARGUMENT
NUMREF EQU  >200C        NUMERIC REFERENCE
NUMASG EQU  >2008        NUMERIC ASSIGNMENT
XMLLNK EQU  >2018        XML LINK VECTOR
CIF    EQU  >20          CONVERT INTEGER TO F.P.
CFI    EQU  >12B8        CONVERT F.P. TO INTEGER
ARGNUM EQU  >8312        NUMBER OF ARGUMENTS
*
*
RANDAR LWPI WS           LOAD OUR WORKSPACE
       A    @>8378,@>83C0 ADJUST SEED NUMBER
       CLR  R0           CLEAR FOR NON-ARRAY
       MOVB @ARGNUM,R12  GET NUMBER OF PARAMETERS
       SRL  R12,8        RT. JUSTIFY
       JEQ  EXIT         IF ZERO, EXIT
       CI   R12,2        2 PARAMETERS?
       JLT  EXIT         IF LESS, EXIT
       LI   R1,2         2ND PARAMETER
       BLWP @NUMREF      GET VALUE
       BLWP @XMLLNK      NUMBER OF NUMBERS
       DATA CFI          MAKE INTEGER
       MOV  @FAC,R13     INTO R13
       JEQ  EXIT         IF ZERO, EXIT
       ABS  R13          ABSOLUTE VALUE
       CI   R12,3        3 PARAMETERS?
       JLT  RAND2        IF LESS, JUMP
       INC  R1           3RD PARAMETER
```

```
        BLWP @NUMREF      GET MULTIPLIER
        LI   R10,HILIM    POINT AT HILIM
        BL   @FRFAC       MOVE 8 BYTES
        CI   R12,4        4 PARAMETERS?
        JLT  RAND2        IF LESS, JUMP
        INC  R1           4TH PARAMETER
        BLWP @NUMREF      GET ADDITION NUMBER
        LI   R10,LOLIM    POINT AT LOLIM
        BL   @FRFAC       MOVE 8 BYTES THERE
        CI   R12,5        5 PARAMETERS?
        JLT  RAND2        IF LESS, SKIP
        INC  R1           5TH PARAMETER
        BLWP @NUMREF      GET BASE
        BLWP @XMLLNK      USE XML
        DATA CFI          TO INTEGER
        A    @FAC,R0      ADD TO R0
RAND2   LI   R1,1         PARAMETER 1
        LI   R7,7         7 BYTES IN F.P. MANTISSA
        LI   R6,100       100 FOR DIVISION
        LI   R10,FAC+1    ONE BYTE PAST >834A
BRAN1   LI   R4,28645     BIG NUMBER IN R4
        MPY  @>83C0,R4    MULT. BY SEED
        AI   R5,31417     ADD BIG NUMBER
        MOV  R5,@>83C0    PUT BACK AT SEED
        CLR  R4           CLEAR HIGH WORD
        DIV  R6,R4        DIVIDE BY 100
        SWPB R5           REMAINDER IN LEFT BYTE
        MOVB R5,*R10+     ONE BYTE TO FAC
        DEC  R7           SUBTRACT 1 FROM R7
        JNE  BRAN1        IF NOT ZERO, REPEAT
        MOVB @SIXTRE,@FAC PUT 63 IN EXPONENT BYTE
        CI   R12,3        3 PARAMETERS?
        JLT  BRAN2        JUMP IF LESS
        LI   R9,HILIM     POINT AT HILIM
        BL   @TOARG       MOVE 8 BYTES TO ARG
        BLWP @XMLLNK      USE XML
        DATA FMUL         MULTIPLY FAC BY ARG
        CI   R12,4        4 PARAMETERS?
        JLT  BRAN2        JUMP IF LESS
        LI   R9,LOLIM     POINT AT LOLIM
        BL   @TOARG       MOVE 8 BYTES TO ARG
        BLWP @XMLLNK      USE XML
        DATA FADD         ADD ARG TO FAC
BRAN2   CI   R12,6        SIX PARAMETERS?
        JLT  BRAN3        JUMP IF LESS
        CLR  R4           MAKE R4=0
        MOV  @FAC,R7      GET THE EXPONENT AND HIGH-ORDER BYTE OF NUMBER INTO R7
        JEQ  BRAN3        IF FAC=0 THEN NUMBER WAS ZERO, SKIP ALL THIS
        SRA  R7,8         SHIFT ARITHMETIC TO PRESERVE SIGN OF EXPONENT
        JGT  SUB62        IF POSITIVE, SKIP AHEAD
```

```
       INV  R7           INVERSION GIVES THE CORRECT EXPONENT VALUE
SUB62  AI   R7,-62       REMOVE BIAS LESS 2
       CI   R7,1         CHECK RESULT AGAINST 1
       JGT  ADDFAC       IF GREATER, JUMP
       CLR  R7           ELSE SET R7=0
ADDFAC AI   R7,FAC       ADD FAC ADDRESS TO R7, SO R7 POINTS TO FIRST DECIMAL
INLOOP CI   R7,FAC+8     CHECK FOR END OF 8 BYTE FP NUMBER
       JGT  BRAN3        IF GREATER THAN, JUMP OUT
       JEQ  BRAN3        IF EQUAL, JUMP OUT
       MOVB R4,*R7+      MOVE ZERO BYTE INTO LOCATION, INCREMENT POINTER (R7)
       JMP  INLOOP       GO BACK FOR NEXT BYTE
BRAN3  BLWP @NUMASG      ASSIGN TO XB VARIABLE
       INC  R0           NEXT ARRAY MEMBER
       DEC  R13          SUBTR. 1 FROM R13
       JNE  RAND2        IF NOT ZERO, REPEAT
EXIT   LWPI GPLWS        LOAD GPL WS
       B    @>6A         EXIT TO GPL INT.
*
* SUBROUTINES
*
TOARG  LI   R10,ARG      ARG IS DESTINATION
       JMP MOVBTS        MOVE BYTES
FRFAC  LI   R9,FAC       FAC IS SOURCE
MOVBTS LI   R4,8         8 BYTES TO MOVE
MOV1   MOVB *R9+,*R10+   MOVE ONE BYTE
       DEC  R4           SUBTR. 1 FROM R4
       JNE  MOV1         IF NOT ZERO, REPEAT
       RT                RETURN
*
* DATA SECTION
*
WS     BSS  32           OUR OWN WORKSPACE
HILIM  BSS  8            MULTIPLIER
LOLIM  BSS  8            ADD-ON NUMBER
SIXTRE BYTE 63           EXPONENT (100 TO THE -1 POWER)
       END


* PART TWO - FOR SIMPLE VARIABLES
*

* RANDFP/S
* ASSIGNS RANDOM NUMBER
* IN F.P FORMAT
* INTO AN XB VARIABLE
*
*
*  INVOKE BY:
*  CALL LINK("RANDFP",X,M,A,1)
*  WHERE:
```

```
*    X IS ANY VARIABLE NAME
*    M IS THE DESIRED MULTIPLIER (RANGE)
*    A IS THE DESIRED ADDITION (START #)
*    1 (OR ANY NUMBER) MEANS INTEGERS
* ONLY X IS ALWAYS REQUIRED
* OTHER PARAMETERS ARE OPTIONAL
* (SEE TEXT FOR DETAILS)
*
*    CODE BY:  Bruce Harrison
*    PUBLIC DOMAIN
*    26 December 1994
*
       DEF  RANDFP DEFINE ENTRY POINT
*
* REQUIRED EQUATES
*
GPLWS  EQU  >83E0        GPL WORKSPACE
FAC    EQU  >834A        F. P. ACCUMULATOR
FMUL   EQU  >0E88        F. P. MULTIPLY
FADD   EQU  >0D80        F. P. ADDITION
ARG    EQU  >835C        F. P. ARGUMENT
NUMREF EQU  >200C        NUMERIC REFERENCE
NUMASG EQU  >2008        NUMERIC ASSIGNMENT
XMLLNK EQU  >2018        XML LINK VECTOR
CIF    EQU  >20          CONVERT INTEGER TO F.P.
CFI    EQU  >12B8        CONVERT F.P. TO INTEGER
ARGNUM EQU  >8312        NUMBER OF ARGUMENTS
*
*
RANDFP LWPI WS           LOAD OUR WORKSPACE
       A    @>8378,@>83C0 ADJUST SEED NUMBER
       CLR  R0           CLEAR FOR NON-ARRAY
       MOVB @ARGNUM,R12  NUMBER OF PARAMETERS
       SRL  R12,8        RT. JUST.
       JEQ  EXIT         EXIT IF ZERO
       CI   R12,2        2 PARAMS?
       JLT  RAND2        JUMP LESS
       LI   R1,2         2ND PARAMETER
       BLWP @NUMREF      GET MULTIPLIER
       LI   R10,ARG      POINT AT ARG
       BL   @FRFAC       MOVE 8 BYTES
       CI   R12,3        3 PARAMS?
       JLT  RAND2        JUMP LESS
       INC  R1           3RD PARAMETER
       BLWP @NUMREF      GET ADDITION
       LI   R10,LOLIM    POINT AT LOLIM
       BL   @FRFAC       MOVE IT THERE
RAND2  LI   R1,1         PARAMETER 1
       LI   R7,7         7 BYTES
       LI   R6,100       100 DIVISOR
```

```
       LI   R10,FAC+1    POINT AT >834B
BRAN1  LI   R4,28645     BIG NUMBER IN R4
       MPY  @>83C0,R4    MULT. BY SEED
       AI   R5,31417     ADD BIG NUMBER
       MOV  R5,@>83C0    PUT BACK AT SEED
       CLR  R4           CLEAR HIGH WORD
       DIV  R6,R4        DIVIDE BY 100
       SWPB R5           REMAINDER IN LEFT BYTE
       MOVB R5,*R10+     ONE BYTE TO FAC
       DEC  R7           SUBTR. 1
       JNE  BRAN1        NOT ZERO, RPT.
       MOVB @SIXTRE,@FAC MOVE EXPONENT IN
       CI   R12,2        TWO PARAMS?
       JLT  BRAN2        JUMP LESS
       BLWP @XMLLNK      USE XML
       DATA FMUL         MULTIPLY FAC BY ARG
       CI   R12,3        3 PARAMS?
       JLT  BRAN2        JUMP LESS
       LI   R9,LOLIM     LOLIM IS SOURCE
       BL   @TOARG       MOVE TO ARG
       BLWP @XMLLNK      USE XML
       DATA FADD         ADD ARG TO FAC
BRAN2  CI   R12,4        FOUR PARAMS?
       JLT  BRAN3        IF LESS, JUMP
       CLR  R4           MAKE R4=0
       MOV  @FAC,R7      GET THE EXPONENT AND HIGH-ORDER BYTE OF NUMBER INTO R7
       JEQ  BRAN3        IF FAC=0 THEN NUMBER WAS ZERO, SKIP ALL THIS
       SRA  R7,8         SHIFT ARITHMETIC TO PRESERVE SIGN OF EXPONENT
       JGT  SUB62        IF POSITIVE, SKIP AHEAD
       INV  R7           INVERSION GIVES THE CORRECT EXPONENT VALUE
SUB62  AI   R7,-62       REMOVE BIAS LESS 2
       CI   R7,1         CHECK RESULT AGAINST 1
       JGT  ADDFAC       IF GREATER, JUMP
       CLR  R7           ELSE SET R7=0
ADDFAC AI   R7,FAC       ADD FAC ADDRESS TO R7, SO R7 POINTS TO FIRST DECIMAL
INLOOP CI   R7,FAC+8     CHECK FOR END OF 8 BYTE FP NUMBER
       JGT  BRAN3        IF GREATER THAN, JUMP OUT
       JEQ  BRAN3        IF EQUAL, JUMP OUT
       MOVB R4,*R7+      MOVE ZERO BYTE INTO LOCATION, INCREMENT POINTER (R7)
       JMP  INLOOP       GO BACK FOR NEXT BYTE
BRAN3  BLWP @NUMASG      ASSIGN TO XB VARIABLE
EXIT   LWPI GPLWS        LOAD GPL WS
       B    @>6A         EXIT TO GPL INT.
*
* SUBROUTINES
*
TOARG  LI   R10,ARG      ARG DESTINATION
       JMP MOVBTS        MOVE BYTES
FRFAC  LI   R9,FAC       FAC IS SOURCE
MOVBTS LI   R4,8         8 BYTES TO MOVE
```

```
MOV1   MOVB *R9+,*R10+   MOVE ONE
       DEC  R4           SUBTR. 1
       JNE  MOV1         IF NOT ZERO, RPT
       RT                RETURN
*
* DATA SECTION
*
WS     BSS  32           OUR OWN WORKSPACE
LOLIM  BSS  8            BOTTOM NUMBER
SIXTRE BYTE 63           EXPONENT (100 TO -1 POWER)
       END

* PART THREE - TWO XB PROGRAMS
* (LISTED IN 28 COLUMNS)
* FIRST USES RANDAR TO ASSIGN
* 300 RANDOM NUMBERS TO A()
*
10 CALL INIT :: CALL LOAD("D
SK1.RANDAR/O")
20 RANDOMIZE :: DIM A(299)
30 B=299 :: DISPLAY AT(24,1)
:"STARTING TAILORED";B+1
40 CALL LINK("RANDAR",A(),B+
1,50000,50000)
50 FOR I=0 TO B :: PRINT A(I
),:: NEXT I
60 CALL KEY(0,K,S):: IF S=0
THEN 60 ELSE IF K=13 THEN EN
D ELSE 30
*
* SECOND PROGRAM USES RND TO ASSIGN
* 300 RANDOM NUMBERS TO A()
*
10 ! RANDOM USING RND
20 RANDOMIZE :: DIM A(299)
30 B=299 :: DISPLAY AT(24,1)
:"STARTING TAILORED";B+1 ::
FOR I=0 TO B
40 A(I)=RND*50000+50000 :: N
EXT I
50 FOR I=0 TO B :: PRINT A(I
),:: NEXT I
60 CALL KEY(0,K,S):: IF S=0
THEN 60 ELSE IF K=13 THEN EN
D ELSE 30
```

1.56. The Art Of Assembly — Part 56. Playing Cards

By Bruce Harrison

Well, another month and more random numbers. This time, however, we're going a couple of steps beyond the numbers themselves, and giving you a shuffled deck of cards on the screen. Besides giving you an extremely fast shuffle routine, we've provided a very quick and easy way to "map" the numbers into suits, complete with the symbols for Spades, Clubs, Hearts, and Diamonds, and the card values from Ace through King. In the sample program, we've even set things up so that the numbers for the red suits will be in red, while those for the black suits will be in black. (Yes, the Heart and Diamond symbols are in red, too)

1.56.1. What's It For?

You decide! What's provided in today's Sidebar is a complete program. It will start with a "PRESS ANY KEY TO BEGIN" prompt, so that we can perform a proper seeding of the random number generator. When you do that, lots of things will happen in just a few milliseconds. You'll see a deck of 52 cards appear on the screen almost instantly. They'll be nicely shuffled, so we can't tell you what order they'll be shown in. Of course these are not a simulation of real cards, but the numbers and symbols in their correct colors on a gray background. There will be eight rows of six cards each, then a last row of four, for a total of 52. At the bottom of the screen will be the legends PRESS ENTER TO EXIT PGM and ANY OTHER KEY TO REPEAT. Try this second option by pressing the **SPACE BAR**. A new "deal" of shuffled cards will appear almost instantly. To see how fast, just start pressing keys randomly as quick as you can. New decks will appear just about as fast as you can press keys. Impressed? We hope so. Of course this complete program isn't a card game per se, but you can use what's here to make one, by taking pieces of this source code and building upon this base.

1.56.2. From The Top

The opening is conventional, simply getting us into our own workspace and putting a prompt on the screen. Instead of just getting a keystroke, however, there's some trickery done here with the Screen Timeout and Vertical Interval timers so that when we finish this "key input" loop, the random number seed at >83C0 has an unpredictable number in the range 0 through 65,535. In some of our other routines, we've used a shortcut method to "seed" our random numbers, but the way shown here gives the best possible results in terms of randomness.

We set up the colors for character sets 0 through 18 to black on gray. Later, we'll set colors for sets 19 through 21 at red on gray for the red card suits. This will leave us with a gray background color and a green border, and as you'll see, the red card suits will have red numbers as well.

Now we set up character definitions. For this, we borrow some of the original TI character definitions, but copy them so that all of the characters we'll be using will be above the "normal" ASCII range. Thus our first character will be at 128. That character will be an A. But first, we grab the character definitions for "2" through "9", and copy those in as definition for characters 129 through 136. Next we take the "A" character definition and place that at 128's definition. In similar fashion, J, Q and K are copied into the definitions for 138-140. Finally, we take a special character definition that makes the number 10 in one 8X8 block, and put that in character 137. This gives us a set of 13 characters starting with 128 and going through 140 that will print as the card numbers A, 2, 3, . . . J, Q, K. In order to allow for red numbers in the red suits, we'll now copy this set from 128-140 into 160-172 character definitions. Finally, we take the "suit" symbols, each of which has four character definitions, and place those in VDP where characters 144 through 159 definitions belong. We finish the preparations by setting the color bytes for character sets 19 through 21 to red on gray.

1.56.3. Build And Shuffle

Now it's time for the real business of the program to start. Starting at label SHUFLE, we build an array of 52 bytes with the values 0 through 51 starting at label TBL1 in the data section. This array will be the source of our "cards". We'll take these numbers one at a time in random order into the array TBL2, which will be our shuffled deck. We've described the process of selection without replacement previously, and this shuffle works exactly that way.

1.56.4. Displaying The Cards

At this point, TBL2 has the 52 numbers from 0 through 51 in random order. We take each number in turn, place that into R2, then right justify in that register, so that R2 has a value of 0 through 51. Now we clear R1, so that the register pair R1-R2 has that number in it. We then divide the R1-R2 pair by 13 (in R6). After this division, R1 contains a quotient that's 0 through 3, and R2 has a remainder that's 0 through 12. Let's take an example or four. Suppose the number in R2 were 0. When divided by 13, this yields a quotient of zero (Spades) and a remainder of 0 (Ace). If it were 15, we'd get quotient 1, remainder 2, which displays as the 3 of Clubs. For 26, we'd get a quotient of 2, with 0 remainder, and that's the Ace of Hearts. For 51, we get a quotient 3 and remainder of 12, which becomes the King of Diamonds. All clear now?

Since there are four possible states for R1, the number there will be the "suit" indicator. R2 has 13 possible values, so that will be used to represent the card numbers Ace through King. (Note that the value 0 is taken as the Ace, and 12 as the King. In some games, Ace can be either below deuce or above King, while in others it's always above the King. How it's valued in a particular game is immaterial to the us in this program.)

We check the value in R1 at this point, to see whether the suit we're dealing with is black or red. The number 0 or 1 in R1 means a black suit, while the number 2 or 3 means a red suit. The suit symbols themselves have been placed in VDP so that the first two are in one character set that's black and the other two are in a character set that's red. We check here to see which is the case, and add an offset of 32 to the card number in R2 if the suit is a red one.

The next order of business in the program is to put the suit symbol on the screen. That process starts at label SHL, where the number in R1 gets multiplied by four to account for the fact that each suit symbol is four characters. The number in R1 now, after SHL R1,2 is 0, 4, 8, or 12. Now we add the offset 144, which is the start of the suit characters, so that R1 contains the number of the first character in one of the suits.

Confession time again. These suit symbol characters were originally used in an old XB program we wrote, and were used with magnified sprites. Thus the first character in the symbol is the upper left quarter, the next is the lower left quarter, the third is the upper right quarter, and you can guess where the fourth one goes. Thus this little section of source code writes a character, moves down one row, writes the next one, then moves up and right for the third, and down one for the fourth.

Next to be written is the card number, which is still there in R2. It may or may not have had 32 added to it, depending on the suit. In either case, we move it to R1 and add the offset 128 to point to one of the sets of card numbers. We SUTRACT 34 from R0 so the card number will be placed to the left of the upper left quarter of the symbol, swap the bytes in R1, perform a BLWP @VSBW, and we've got the card number and suit symbol on the screen. Now we move six spaces to the right, and we're in position to start the next card.

Before writing the next card, though, we first DEC R5, and if that's zero, we've written all 52 cards, so we skip ahead to label KEY. We also DEC R6 and see if that's become zero. If not, we can still put another card on this row. Otherwise we adjust the screen position two rows down from the start of the current row and jump back to NXT0 to start another row of six cards.

The code starting at label KEY just puts the "PRESS ENTER . . . " and "ANY OTHER KEY. . ." legends on screen, then waits for a keypress. If **ENTER** is pressed, we exit, and any other key (except **FCTN =**) will cause a branch back to label SHUFLE, and put a whole new shuffled deck on-screen.

1.56.5. Variations On The Theme

In the part of the source file just before and after label KEY1, you'll see four lines that have been "commented out" with an asterisk. If you erase the asterisks from these lines, the program won't wait forever for you to press a key after displaying a shuffled deck. It will wait precisely four and 1/60th seconds, then will re-shuffle and display again. If you don't like four seconds, change the 4 in the line LI R0,120*4 to some other number, and that's how many seconds the program will wait for a keystroke. The limit is 273 in this case, which will wait about 4 1/2 minutes. Pressing a key during this interval will have the same effect as before, causing a new shuffle immediately. For those who get *MICROpendium* on disk, we've included the file SHUFFLE/O, so you can try this out without having to assemble it.

In this program, we've "borrowed" the standard character definitions for 2 through 9, A, J, Q, and K, then put in our own special character for 10, so that 10 can get printed as just one character. You might want to design your own characters for the numbers and letters, making them a bit narrower so they'll look better with the 10 character. You might also want to make single character suit symbols so they needn't take up so much screen space, and you might want to shift the definitions around so that the numbers go from 0=2 through 12=Ace. In some card games that would make evaluating hands easier than the mapping we've used here, as an Ace at 12 would "beat" a King at 11.

The rest is up to you, dear reader! Using this "baseline" you could make some fascinating games that use ordinary decks of cards. You could add 2 Jokers, make card outlines, and so on. For a real challenge, try making this over into a Pinochle deck with its 48 cards and multiple same suit and values. You've now got your work cut out for you, but with this "leg up" in today's Sidebar, it should be a bit easier. May the luck of the draw go with you always. 'Bye for now.

```
* SIDEBAR 56
* A COMPLETE PROGRAM
*
* SHUFFLE/S
* SHUFFLES 52 CARDS
* AND DISPLAYS THEM
*   Code by Bruce Harrison
*   PUBLIC DOMAIN
*   6 JANUARY 1995
*
       DEF START         ENTRY POINT
       REF VSBW,VMBW,VMBR,VWTR,KSCAN REF UTILS
START  LWPI WS           USE OUR WORKSPACE
       CLR  @>8374       CLEAR KEY-UNIT
*
* NEXT SECTION AWAITS KEYPRESS AND THEN
* PROPERLY SEEDS THE RANDOM NUMBER
*
       MOV  @>8378,R8    CAPTURE VERTICAL COUNT
       ANDI R8,1         JUST LEAST SIGNIFICANT BIT
       LI   R1,PAK       "PRESS ANY KEY..."
       LI   R0,11*32+4   ROW 12, COL 5
       LI   R2,22        22 CHARS
       BLWP @VMBW        WRITE THAT
KEY0   MOV  @>83D6,R10   SCRN TIMEOUT TO R10
       BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTS.
       LIMI 0            STOP INTS.
       CB   @>837C,@ANYKEY KEY PRESSED?
       JNE  KEY0         IF NOT, REPEAT
       MOVB @>8379,R10   VERT TIME TO R10
       A    R8,R10       ADD 1 OR 0 FROM R8
       MOV  R10,@>83C0   R10 TO RAND SEED
       CLR  @>8378       CLEAR VERT TIMER
*
```

```
* NEXT SECTION ERASES THE "PRESS ANY KEY"

       MOVB @ANYKEY,R1   >20 (SPACE) TO R1
CLR0   BLWP @VSBW        WRITE A SPACE
       INC  R0           MOVE ONE TO RIGHT
       DEC  R2           DEC COUNT
       JNE  CLR0         RPT IF NOT 0
*
* NEXT SECTION SETS COLORS FOR CHAR SETS
* 0 THROUGH 18 AT BLACK ON GRAY
*
       LI   R0,>380      POINT AT COLOR TABLE
       LI   R4,19        19 BYTES
       LI   R1,>1E00     BLACK ON GRAY
COLOR  BLWP @VSBW        WRITE ONE
       INC  R0           POINT AHEAD
       DEC  R4           DEC COUNT
       JNE COLOR         RPT IF NOT 0
*
* NEXT SECTION MAKES CARD CHARACTER DEFINITIONS
* USING NUMBERS AND LETTERS WHERE NEEDED
*
       LI   R0,>32*8+>800 START AT DEF FOR "2"
       LI   R1,TBL1       TBL1 IS TEMP STORAGE
       LI   R2,64         8 CHARACTERS WORTH
       BLWP @VMBR         READ THOSE
       LI   R0,129*8+>800 START AT CHAR 129 DEF
       BLWP @VMBW         WRITE 2-9 DEFS THERE
       LI   R0,'A'*8+>800 CHAR DEF FOR "A"
       LI   R2,8         EIGHT BYTES
       BLWP @VMBR        READ CHAR PAT
       LI   R0,128*8+>800 CHAR DEF FOR 128
       BLWP @VMBW        WRITE 'A' CHARPAT THERE
       LI   R0,'J'*8+>800 CHAR DEF FOR "J"
       BLWP @VMBR        READ PATTERN
       LI   R0,138*8+>800 "J" CHAR IN CHR 138
       BLWP @VMBW        PUT "J" PAT THERE
       LI   R0,'Q'*8+>800 "Q" CHAR PAT
       BLWP @VMBR        READ THAT
       LI   R0,139*8+>800 139 CHAR
       BLWP @VMBW        WRITE "Q" PAT THERE
       LI   R0,'K'*8+>800 "K" CHAR PAT
       BLWP @VMBR         READ
       LI   R0,140*8+>800 140 CHAR PAT
       BLWP @VMBW        WRITE "K" PAT THERE
       LI   R0,137*8+>800 137 CHAR
       LI   R1,SPCHAR     SPECIAL CHAR "10"
       BLWP @VMBW         WRITE "10" PAT THERE
*
* NEXT SECTION COPIES 13 CARD NUMBERS INTO
```

```
* AREA FROM CHR 160 - 172 DEFINITIONS
* FOR THE RED CARD CHARACTERS
*
       LI   R0,128*8+>800 128 CHAR PAT
       LI   R2,104        104 BYTES (13 CHARS)
       LI   R1,TBL1       TEMP STORAGE
       BLWP @VMBR         READ 13 CHAR PATS
       LI   R0,160*8+>800 CHAR 160 PAT
       BLWP @VMBW         WRITE 13 CHARS THERE
*
* NEXT SECTION DEFINES CHARS 144 THRU 159 AS
* SUIT SYMBOLS (4 CHARS EACH)
*
       LI   R0,144*8+>800 CHAR 144 PATTERN
       LI   R1,CHR144     SUIT CHARS
       LI   R2,128        16 CHAR PATS
       BLWP @VMBW         WRITE THOSE
*
* NEXT SETS COLORS FOR CHAR SETS 19, 20 & 21 TO RED ON GRAY
*
       LI   R1,>6E00      RED ON GRAY
       LI   R0,>380+19    CHAR SET 19 COLOR BYTE
       BLWP @VSBW         WRITE
       INC  R0            CHAR SET 20
       BLWP @VSBW         WRITE
       INC  R0            CHAR SET 21
       BLWP @VSBW         WRITE
*
* NEXT SECTION BUILDS DECK IN TBL1
* THEN SHUFFLES THAT INTO TBL2
*
SHUFLE CLR  R3            SET R3=0
       LI   R9,TBL1       POINT AT TBL1
       LI   R4,52         52 CARDS
       A    @>8378,@>83C0 ADJUST SEED
BLDTBL MOVB R3,*R9+       LEFT BYTE  R3 TO TABLE
       AI   R3,>100       INC LEFT BYTE R3
       DEC  R4            DEC COUNT
       JNE  BLDTBL        RPT IF NOT 0
       SWPB R3            SWAP BYTES IN R3
*
* R3 NOW CONTAINS 52
*
       LI   R10,TBL2      POINT AT TBL2
RANDNO LI   R4,28645      BIG NUMBER IN R4
       MPY  @>83C0,R4     MULTIPLY BY SEED
       AI   R5,31417      ADD BIG NUMBER
       MOV  R5,@>83C0     RESULT TO SEED
       CLR  R4            CLEAR HI WORD
       DIV  R3,R4         DIVIDE BY R3
```

```
       MOVB @TBL1(R5),*R10+ GET BYTE FROM TBL1 TO TBL2
       MOVB @TBL1-1(R3),@TBL1(R5) REPLACE THE USED ONE
       DEC  R3              DEC COUNT IN R3
       JNE  RANDNO          RPT IF NOT 0
*
* AT THIS POINT TBL2 HAS 52 BYTES RANGING FROM 0 THROUGH 51
* IN RANDOM ORDER, EACH VALUE ONLY ONCE
*
* NEXT SECTION DISPLAYS ALL 52 CARDS ON SCREEN
*
       LI   R0,32+3         ROW 2, COL 4
       MOV  R0,R13          SAVE R0 IN R13
       LI   R9,TBL2         SHUFFLED DECK IN TBL2
       LI   R5,52           52 CARDS TO SHOW
       LI   R6,13           13 IN R6
NXT0   LI   R4,6            6 CARDS PER ROW
NEXT   MOVB *R9+,R2         GET CARD VALUE
       SRL  R2,8            RT. JUST.
       CLR  R1              R1=0
       DIV  R6,R1           DIV R1-R2 BY 13
*
* AT THIS POINT R1 CONTAINS QUOTIENT RANGING FROM 0 THRU 3
* THIS IS THE "SUIT" FOR THIS CARD WITH 0=SPADES, 1=CLUBS
* 2=HEARTS, 3=DIAMONDS
*
* R2 CONTAINS REMAINDER IN THE RANGE 0 THRU 12
* THIS IS THE NUMBER OF THE CARD, WHERE:
* 0=ACE, 1=DEUCE, 2=TREY ... 10=JACK, 11=QUEEN, 12=KING
*
       CI   R1,2            IS R1<2?
       JLT  SHL             IF LESS, JUMP
*
* IF SUIT IS BELOW 2, BLACK NUMBERS
* ELSE MOVE NUMBER TO RED CHARACTERS
*
       AI   R2,32           ELSE ADD OFFSET FOR RED
*
* FOLLOWING PUTS 4 CHARS OF SUIT SYMBOL ON SCREEN
*
SHL    SLA  R1,2            MULTIPLY R1 BY 4
       AI   R1,144          ADD 144
       SWPB R1              SWAP BYTES
       BLWP @VSBW           WRITE ONE BYTE
       AI   R0,32           DOWN ONE ROW
       AI   R1,>100         NEXT CHAR
       BLWP @VSBW           WRITE
       AI   R0,-31          UP ONE, ONE RT.
       AI   R1,>100         NEXT CHAR
       BLWP @VSBW           WRITE
       AI   R0,32           DOWN ONE
```

```
       AI   R1,>100       NEXT CHAR
       BLWP @VSBW         WRITE
*
* FOLLOWING PUTS THE CARD NUMBER ON SCREEN
*
       MOV  R2,R1         GET CARD NUMBER
       AI   R1,128        ADD OFFSET 128
       AI   R0,-34        MOVE TO LEFT AND UP
       SWPB R1            SWAP BYTES
       BLWP @VSBW         WRITE CARD NUMBER
       AI   R0,6          MOVE 6 TO RIGHT
       DEC  R5            DEC R5 COUNT (52 CARDS)
       JEQ  KEY           IF ZERO, WE'RE FINISHED
       DEC  R4            DEC CARDS ON ONE ROW COUNT
       JNE  NEXT          JUMP TO NEXT IF NOT 0
       AI   R13,64        ADD 2 ROWS TO R13
       MOV  R13,R0        PUT THAT IN R0
       JMP  NXT0          THEN ANOTHER ROW
*
* NEXT SECTION PUTS LEGENDS AT BOTTOM, AWAITS KEY PRESS
*
KEY    LI   R0,20*32+4    ROW 21, COL 5
       LI   R1,ENTMSG     ENTER MESSAGE
       LI   R2,23         23 CHARS
       BLWP @VMBW         WRITE
       A    R2,R1         NEXT MSG
       AI   R0,64         TWO ROWS DOWN
       BLWP @VMBW         WRITE THAT
*      LI   R0,120*4      TIME IN 120THS SECOND
*      CLR  @>83D6        CLEAR TIMEOUT COUNT
KEY1   BLWP @KSCAN        SCAN KEYBRD
       LIMI 2             INT ON
       LIMI 0             INT OFF
*      C    @>83D6,R0     CHECK TIME
*      JGT  SHUFLE        IF UP, RE-SHUFFLE
       CB   @>837C,@ANYKEY KEY STRUCK?
       JNE KEY1           RPT IF NOT
       CB   @>8375,@ENTERV "ENTER" STRUCK?
       JEQ  EXIT          EXIT IF SO
       B    @SHUFLE       ELSE NEW SHUFFLE
EXIT   LWPI >83E0         LOAD GPL WORKSPACE
       B    @>6A          GO TO GPL INTERPRETER
*
* DATA SECTION
*
WS     BSS  32           OUR WORKSPACE
*
* CHR144 IS START OF DATA FOR SUIT SYMBOLS
* FOR SPADE, CLUB, HEART, AND DIAMOND
*
```

```
CHR144 DATA >0103,>0707,>0F0F,>1F1F  \
       DATA >3F7F,>7F7D,>3101,>0307   | SPADE SYMBOL
       DATA >80C0,>E0E0,>F0F0,>F8F8   |
       DATA >FCFE,>FEBE,>8C80,>C0E0  /

       DATA >0103,>0707,>0331,>79FF  \
       DATA >FF79,>3101,>0103,>070F   | CLUB SYMBOL
       DATA >80C0,>E0E0,>C08C,>9EFF   |
       DATA >FF9E,>8C80,>80C0,>E0F0  /

       DATA >183C,>7EFF,>FFFF,>FF7F  \
       DATA >3F3F,>1F1F,>0F07,>0301   | HEART SYMBOL
       DATA >183C,>7EFF,>FFFF,>FFFE   |
       DATA >FCFC,>F8F8,>F0E0,>C080  /

       DATA >0103,>070F,>1F3F,>7FFF  \
       DATA >FF7F,>3F1F,>0F07,>0301   | DIAMOND SYMBOL
       DATA >80C0,>E0F0,>F8FC,>FEFF   |
       DATA >FFFE,>FCF8,>F0E0,>C080  /
*
* SPCHAR IS 10 AS ONE 8X8 CHARACTER
*
SPCHAR DATA >0046,>C949,>4949,>4946
TBL1   BSS  52           UNSHUFFLED DECK
TBL2   BSS  52           SHUFFLED DECK
ENTMSG TEXT 'PRESS ENTER TO EXIT PGM'
ANYMSG TEXT 'ANY OTHER KEY TO REPEAT'
PAK    TEXT 'PRESS ANY KEY TO BEGIN'
ENTERV BYTE 13           ENTER KEY VALUE
ANYKEY BYTE >20          HEX 20 TO COMPARE
       END
```

1.57. The Art Of Assembly — Part 57. We Interrupt This Program. . .

By Bruce Harrison

We get letters from readers, and even phone calls now and then. This time Mr. Terry Blovas called to ask about the use of User Interrupts. His goal was to read the CorComp real time clock during a User Interrupt and to place the time on-screen while other programs were running in the "foreground". The reading of the clock would have to be done by a DSRLNK operation, and he was concerned about whether the DSRLNK could be made to operate during a User Interrupt.

1.57.1. It's Another YES, BUT!

The answer is yes, you can operate a DRSLNK operation during a User Interrupt, but there are some unexpected side effects from such an operation. After sending off some quick help to Mr. Blovas, we started doing some experimenting with the combination of DSRLNK and User Interrupt. To give you an appreciation for the strange interaction, we'll start today's column with a simpler case, in which things behave as we expect them to.

The first part of today's Sidebar is a little experiment that uses the User Interrupt to write to VDP RAM. This is just a "nonsense" program called ATEX. It slowly fills the screen with the "@" symbol by writing one per 1/60th second via a User Interrupt. When that's finished, it swaps the bytes in R1, and starts again at the top of the screen, so that space characters get written in a similar fashion, slowly clearing the screen. Just to show it can be done, the interrupt then puts the legend "FINISHED A CYCLE" in the middle of the screen, and starts over again with the "@" symbol.

While all this is happening during the interrupts, the main program is just cycling endlessly through the "Key loop" at label KEYIN. So long as no key is pressed, this will continue all day. If a key is pressed, the main program will exit its loop, clear the word at >83C4, load the GPL workspace, and then go back to the GPL Interpreter. This all works as planned, so the key loop does sense a keypress, and the interrupt goes about its business because the loop includes LIMI 2 and LIMI 0. Mr. Blovas was under the impression that you couldn't write to VDP during an interrupt, so we sent along a copy of ATEX to show him that this could indeed be done, both by VSBW and VMBW. (You can also read from VDP by VSBR and VMBR during an interrupt.) To add some excitement, you can comment out the instruction CLR @>83C4 just after JNE KEYIN. When that's assembled and run, pressing a key will get you out of the program to E/A's PRESS ENTER TO CONTINUE, but the interrupt will continue doing its thing, even after you've pressed **ENTER**, obliterating the E/A Main menu in due course.

1.57.2. But With DSRLNK, . . .

In the second part of the Sidebar is another program, and this one uses DSRLNK during the interrupt. Mr. Blovas was under the impression that DSRLNK was itself an interrupt. We assured him that the TI DSRLNK is not itself an interrupt, and that it should be able to operate during one. Here we've set it up so that on the first pass through the interrupt cycle, the interrupt opens the program's source file for input, then on each successive cycle it reads a record from that file. To save ourselves trouble, we placed the VDP Buffer for this file access at the start of Row 12 in the screen. This way we get to see the records as they come in from the disk without having to put them on-screen ourselves.

We immediately ran into some difficulty. First, the DSRLNK uses parts of the CPU RAM Pad to do its work, and this tended to mess up the operation of the main program. Thus we put in loops at the beginning and end of the interrupt's code to stash away the 256 bytes starting at >8300, then to put those back before leaving the interrupt code. That allowed things to proceed, and sure enough the source file's contents get shown on-screen, one record each 1/60th of a second. There was just one hitch. While this file was open, pressing a key had no effect at all. Only after the file was closed and the interrupt disabled would our main program's BLWP @KSCAN have any effect. The program behaves as if KSCAN were a couple of NOP instructions as long as the Interrupt cycle has a file opened! We know that the main program is still executing, but KSCAN just doesn't work! To re-assure ourselves, we wrote a different version of this program, in which the DSRLNK happens in the main program, not during an interrupt, and in that case KSCAN continues to function as usual while the file is open.

Now before the letters come in, we'll admit that we don't know why this is so. We've nosed around in KSCAN with a dis-assembler, and haven't found any clues as to how KSCAN could "know" that an interrupt had a file opened, nor how that would affect KSCAN's operation. There probably is a reason for this behavior, but TI isn't saying, and we haven't figured out either the why or the how. If any of our readers knows, we'd be very happy to hear from that reader!

1.57.3. It Gets Worse

Up to this point, we're talking about the DSRLNK vector that's built into the E/A module, operating from low memory under E/A Option 3. In anticipation of the need to operate outside the E/A environment, we tried using the Warren/Miller general-purpose GPL/DSR link vectors. This created yet another problem. The Warren/Miller DSRLNK uses its GPLLNK to perform the file operations, and that introduces another complication, in that the GPLLNK branches to >0060 in the Console ROM. When the code starting at >0060 runs, there's a LIMI 2 and LIMI 0 at >0070 thru >0077. This will cause a re-entry into our user interrupt, which of course we can't handle. Thus in the case of the WARREN/MILLER DSRLNK, we have to insert an instruction into our interrupt code itself to shut off the user interrupt before proceeding to the DSRLNK part. That's shown as a commented out line just after JNE PUTPAD in the Sidebar. Doing it this way means that after the interrupt code is finished for one cycle, the loop at GETPAD will put back the previous state of >83C4, thus re-activating the user interrupt for the next cycle.

This is getting pretty muddy, isn't it? Well let's just go on with what happens when the end of file is reached. The code at label CLSF1 executes, first closing the open file, then clearing the "file open" flag, and then there's that mysterious instruction CLR @SAVPAD+>C4. What's That? By clearing the word at SAVPAD+>C4, we allow the code starting at CHEX to put back the RAM Pad contents as before, but when this is done, the word at >83C4 will be cleared, so our user interrupt will no longer be in effect.

1.57.4. Other Things To Try

There are some nifty little experiments you can do with the stuff in today's Sidebar. For example, when the file end is reached in running DSREX, pressing the **SPACE BAR**, or any other key but **ENTER** will cause the file to be re-opened and shown again. If you're really curious, try pressing **FCTN-= (QUIT)** while the file records are flashing by. The file access will stop while you're holding down **FCTN =**, then start up from where it left off when you release either of those keys. If you're quick about it, you can stop and start the file reading several times before the end is reached. Something, however, will "remember" that you pressed **FCTN =**, so when the file ends, you'll go back to the startup title screen or Ramdisk Menu, depending how you're configured. We could even make a "game"of this, seeing who can stop the file the most times before it ends. Perhaps our friend Mickey Cendrowski? But seriously, folks, here's yet another mystery in the inner workings of our favorite computer, and we haven't a clue what's doing this to us.

Our advice to Mr. Blovas was to make sure that he closes that file from which his time information comes before the RTWP ends his interrupt. If he heeds that advice, then he'll most likely be able to achieve his goal, at least while his own programs are running. Of course he'll still have some other problems to overcome. For example, if the clock function is to work on background while other programs are running, he'll have to find a "safe" place for his interrupt code, so that the programs he loads won't overwrite his interrupt code. That could turn out to be a "killer", since people who write programs for the TI usually don't anticipate having to leave room for other things in the expansion memory.

Since we write these things so far in advance, we've sent Mr. Blovas a copy of all this long before it appears here in your *MICROpendium*. Perhaps by the time you see this, he'll have his background clock display working, at least on his own system. We don't have any real-time clock on our own systems, so we can't even test his code for him.

Once again we've demonstrated how far we'll go to help out any of our readers. You, too can take advantage of our nature by sending any plea for help, either through READER-TO-READER or direct to your author. We'll even promise to spell your name correctly! We're available at:

Bruce Harrison
5705 40th Place
Hyattsville MD 20781
U.S.A.
Phone (301) 277-3467

```
*
* SIDEBAR 57
* PLAYING WITH INTERRUPTS
*
*
* ATEX
* STORED AS ATEX/S
* EXPERIMENT WITH INTERRUPTS
* PUBLIC DOMAIN
*   CODE BY: Bruce Harrison
*
       REF VSBW,VMBW,KSCAN REF UTILITIES
       DEF START         DEFINE ENTRY POINT
*
* FIRST SECTION OF CODE JUST SETS THINGS UP
*
START  LWPI WS           LOAD OUR WORKSPACE
       MOV  @INTLOC,@>83C4 ACTIVATE INTERRUPT
       LI   R1,>2040     SPACE IN LEFT BYTE, @ IN RIGHT BYTE
*
* CODE AT SWAP1 GETS REPEATED AFTER SCREEN FINISHED
*
SWAP1  SWPB R1           SWAP SO WE START WITH @ IN LEFT BYTE R1
       CLR  R0           SCREEN ORIGIN
       LI   R2,768       768 TO GO
*
* MAIN PROGRAM CODE HERE JUST WAITS FOR A KEYPRESS,
* BUT KEEPS ALLOWING INTERRUPTS SO THE USER INTERRUPT
* WILL GET SERVICED
*
KEYIN  BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            INTERRUPTS ON
       LIMI 0            INTERRUPTS OFF
       MOV  R2,R2        IS R2 ZERO?
       JEQ  SWAP1        IF SO, BACK TO SWAP1
       CB   @>837C,@ANYKEY KEY PRESSED?
       JNE  KEYIN        IF NOT, REPEAT
       CLR  @>83C4       ELSE CLEAR USER INTERRUPT
       LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         BACK TO GPL INTERPRETER
*
* HERE'S THE INTERRUPT CODE
*
USRINT BLWP @CHVECT      USE CHVECT TO WRITE A CHARACTER
INTEX  RT                THEN RETURN TO INTERRUPT SERVICE ROUTINE
*
* THE VECTOR CHVECT DOES THE SCREEN WRITING
* IT USES THE MAIN CODE'S WORKSPACE
*
CHVECT DATA WS,CHG1      USES OUR OWN WORKSPACE, CODE AT CHG1
```

```
CHG1   BLWP @VSBW        WRITE LEFT BYTE R1 TO SPOT POINTED BY R0
       INC  R0           POINT AT NEXT SPOT
       DEC  R2           DECREMENT COUNT IN R2
       JNE  CHEX         NOT FINISHED
       CB   R1,@ANYKEY   DOING SPACES?
       JNE  CHEX         IF NOT, JUMP
       MOV  R1,R3        STASH R1 FOR NOW
       LI   R0,11*32+5   ROW 12,COL 5
       LI   R1,FINMSG    FINISHED
       LI   R2,17        17 CHARACTERS
       BLWP @VMBW        WRITE THAT
       CLR  R2           THEN CLEAR REG 2
       MOV  R3,R1        GET OLD R1 BACK
CHEX   RTWP              RETURN WITH WORKSPACE POINTER
WS     BSS  32           OUR WORKSPACE
INTLOC DATA USRINT       INTERRUPT'S ADDRESS
FINMSG TEXT 'FINISHED A CYCLE'
ANYKEY BYTE >20          HEX 20 FOR COMPARISON
       END

* PART TWO
* THE DSR LINK PROBLEM
*
* DSREX
* STORED AS DSREX/S
* EXPERIMENT WITH INTERRUPTS
* PUBLIC DOMAIN
*   CODE BY: Bruce Harrison
*
       REF DSRLNK,VSBW,VSBR,VMBW,KSCAN
       DEF START         DEFINE ENTRY POINT
PAB    EQU  >1000        PAB LOCATION IN VDP RAM
BUF    EQU  11*32        BUFFER AT ROW 12, COL 1
PABPNT EQU  >8356        NAME LENGTH POINTER
STATUS EQU  >837C        GPL STATUS BYTE
*
* FIRST SECTION OF CODE JUST SETS THINGS UP
*
START  LWPI WS           LOAD OUR WORKSPACE
SETUI  MOV  @INTLOC,@>83C4 SET USER INTERRUPT
*
* MAIN PROGRAM CODE HERE JUST WAITS FOR A KEYPRESS,
* BUT KEEPS ALLOWING INTERRUPTS SO THE USER INTERRUPT
* WILL GET SERVICED
*
LIM    LIMI 2            INTERRUPTS ON
       LIMI 0            INTERRUPTS OFF
KEYIN  BLWP @KSCAN       SCAN KEYBOARD
       CB   @ANYKEY,@STATUS KEY STRUCK?
       JNE  LIM          IF NOT, BACK TO LIM
```

```
       CB   @>8375,@ENTERV "ENTER" PRESSED?
       JEQ  STPIT         IF SO, STOP
       JMP  SETUI         ELSE RE-SET USER INTERRUPT
STPIT  CLR  @>83C4        CLEAR USER INTERRUPT
       LWPI >83E0         LOAD GPL WORKSPACE
       B    @>6A          BACK TO GPL INTERPRETER
*
* HERE'S THE INTERRUPT CODE
*
USRINT BLWP @DSVECT       USE DSVECT TO OPEN OR READ THE FILE
INTEX  RT                 THEN RETURN TO INTERRUPT SERVICE ROUTINE
*
*
DSVECT DATA WS,DSRACT     USES OUR WORKSPACE, CODE AT DSRACT
DSRACT LI   R9,>8300      POINT AT RAM PAD
       LI   R10,SAVPAD    AND PLACE TO SAVE IT
       LI   R4,256        256 BYTES TO MOVE
PUTPAD MOVB *R9+,*R10+    MOVE A BYTE
       DEC  R4            DECREMENT COUNT
       JNE  PUTPAD        RPT IF NOT ZERO
*      CLR  @>83C4        KILL THE USRINT FOR NOW
       MOV  @F1FLG,R0     IS FILE OPEN?
       JNE  REDREC        IF YES, JUMP
FNOK   LI   R0,PAB        POINT AT PAB
       LI   R1,PABDT      AND PAB DATA
       MOV  @8(R1),R2     GET NAME LENGTH
       AI   R2,10         ADD 10
       BLWP @VMBW         WRITE PAB
       AI   R0,9          ADD NINE
       MOV  R0,@PABPNT    PUT AT POINTER
       BLWP @DSRLNK       USE DSR LINKAGE
       DATA 8             FOR FILE ACCESS
F1OK   INC  @F1FLG        INDICATE FILE OPEN
REDREC LI   R0,PAB        POINT AT PAB
       MOVB @READF,R1     READ OPCODE
       BLWP @VSBW         WRITE THAT
       AI   R0,9          ADD NINE
       MOV  R0,@PABPNT    PUT AT POINTER
       BLWP @DSRLNK       USE DSR LINK
       DATA 8             DATA FOR FILE
       JNE  CLBU          IF NO ERROR, JUMP
       LI   R0,PAB+1      POINT AT PAB PLUS 1
       BLWP @VSBR         READ THE BYTE
       SRL  R1,13         SHIFT RIGHT
       CI   R1,5          "END OF FILE" ERROR?
       JEQ  CLSF1         IF SO, JUMP
       CLR  @SAVPAD+>C4   ELSE CLEAR USER INT
       JMP  CHEX          THEN JUMP
CLSF1  LI   R0,PAB        POINT AT PAB
       MOVB @CLSF,R1      CLOSE OPCODE
```

```
       BLWP @VSBW         WRITE THAT
       AI   R0,9          ADD NINE
       MOV  R0,@PABPNT    TO POINTER
       BLWP @DSRLNK       USE DSR LINK
       DATA 8             CLOSE FILE
       CLR  @F1FLG        CLEAR "OPEN" FLAG
       CLR  @SAVPAD+>C4   CLEAR USER INTERRUPT
       JMP  CHEX          THEN TO EXIT
CLBU   LI   R0,PAB+5      LENGTH OF RECORD
       BLWP @VSBR         READ THAT
       SRL  R1,8          RT. JUST.
       JEQ  CHEX          IF ZERO, JUMP
       LI   R2,80         MAX LENGTH
       S    R1,R2         SUBTRACT ACTUAL
       LI   R0,BUF        POINT AT BUFFER
       A    R1,R0         ADD ACTUAL
       MOVB @ANYKEY,R1    SPACE IN R1
CLRBUF BLWP @VSBW         WRITE A SPACE
       INC  R0            MOVE ONE SPOT
       DEC  R2            DEC COUNT
       JNE  CLRBUF        RPT. IF NOT ZERO
CHEX   LI   R9,SAVPAD     POINT AT SAVED RAMPAD
       LI   R10,>8300     AND AT RAMPAD
       LI   R4,256        256 BYTES
GETPAD MOVB *R9+,*R10+    MOVE ONE
       DEC  R4            DEC COUNT
       JNE  GETPAD        RPT. IF NOT ZERO
       RTWP               RETURN WITH WORKSPACE POINTER
WS     BSS  32            OUR WORKSPACE
SAVPAD BSS  256           SPACE TO SAVE RAMPAD
INTLOC DATA USRINT        INTERRUPT'S ADDRESS
F1FLG  DATA 0             "FILE OPEN" FLAG
PABDT  DATA >0014,BUF,>5000,>0000,>000C
       TEXT 'DSK1.DSREX/S'
ENTERV BYTE 13            ENTER KEY VALUE
READF  BYTE 2             READ OPCODE
CLSF   BYTE 1             CLOSE OPCODE
ANYKEY BYTE >20           HEX 20 FOR COMPARISON
       END
```

1.58. The Art Of Assembly — Part 58. The Limits Of Randomness

By Bruce Harrison

This month we're taking yet another look at the subject of Random Numbers. Yes, it seems we're doing a lot of that, but we think it's important to explore such topics thoroughly, even if some of you might accuse us of "beating a dead horse". The subject today is some limitations we've found in our little Random Number algorithm that we've been using for some time. We found these limits mainly by doing some experiments with "nonsense" programs, three of which are provided in today's Sidebar. These programs are not going to cause any big stir in the TI Community, but will serve to illustrate our lesson.

We started this experimenting with a simple idea. Let's make a program that randomly puts the "@" character on screen until the screen is filled with them, then erase them in random fashion until the screen is cleared. That worked pretty much as expected, except that we noticed a tendency for certain columns to always get filled early in the process. To see better what was happening, we then changed our program so that instead of the "@", we'd use a randomly selected character in the range of 33 through 126 for each randomly chosen screen position. This makes a nice "mess" on the screen. We noticed, though, that on any particular run, we'd get only the odd numbered characters or only the even numbered ones. We could let that run continue as long as we wanted, and still see only odd or only even characters. Why? Starting a new run would change sometimes from odd to even, but we'd never see both odd and even in the same run!

1.58.1. The Even Divisor Problem

To examine this a bit more scientifically, we devised a different test program, (Part 1 of the Sidebar) which just puts the random numbers themselves on the screen until we've filled it to the bottom, then clears the screen and starts over at the top. We put a delay into this program between each number's selection, so we'd have some time to look at the numbers before they'd disappear. What we found was that if we used an even number in R3 as the divisor, then the algorithm would alternate between odd and even numbers. Let's say, for example, we've used 100. The numbers produced would be "randomly" scattered in values between 0 and 99, but they would alternate between odd and even numbers no matter how long the run continued. If we changed the divisor to an odd number, (99, for example) the problem went away, and our random numbers would scatter nicely between 0 and 98, with no apparent pattern of odd and even numbers. Sometimes there would be three or four odds before an even, and vice versa, but there was no discernible pattern.

1.58.2. It Gets Worse

If the divisor is a small number, this gets worse. For example, using 2 simply makes alternating 1 and 0 values, since those are the only non-negative numbers less than 2. This, folks, is not random! Using four also creates a repeating non-random pattern that goes 3,0,1,2 over and over. This may start on any of those numbers, but the repeating is the same as long as it's left to run. Five, however, seems to be okay, in that the numbers 0 through 4 seem to come up in random order.

1.58.3. Can We Fix This?

Fortunately, there are solutions to this problem, and the Sidebar shows a couple of them. In the source for the RANDEX program, you'll notice right after the LWPI WS a LI R3,4, which sets the range for random numbers to 0 through 3, but will create that repeating pattern. Just after that is a "commented out" line that says LI R3,5. A bit further down, just after label BUILD1, you'll see two commented out lines that say AI R5,-1 and JLT BUILD1. These lines are a solution to the problem! If you change the * to a space in each of those three lines, then assemble and run the program, you'll get the numbers from 0 through 3 that you were after, but they'll be random like they should be, not in a fixed pattern. What we've done is to use an odd number in R3, which gives us a number in R5 after BUILD1 that ranges from 0 through 4. We subtract 1 from that, so it ranges from -1 through 3, then if the result is less than zero, we simply repeat the BL @RANDNO at BUILD1. This will mean that, on average, one out of five numbers generated will be rejected, but that will not make any noticeable change in the speed of generating the correct numbers, and the sequence will be random, which is the desired result. This method can be applied for any situation in which the number in R3 would be even. You can use the next odd number, then subtract 1 from R5 after the BL @RANDNO, and repeat the BL if that's below zero.

For example, if the desired range is 0 through 99, you could do something like this:

```
            LI   R3,101
GETRND      BL   @RANDNO
            AI   R5,-1
            JLT  GETRND
```

That will yield nicely scattered numbers between 0 and 99 inclusive, but without the alternating odd-even problem. Of course if you don't care whether odd and even numbers alternate, you could just LI R3,100 and omit the AI R5,-1 and JLT lines.

1.58.4. Tossing A Fair Coin

We said earlier that the situation for 1 and 0 was particularly bad, in that loading R3 with 2 and then doing the BL @RANDNO would yield just alternating ones and zeros. True, but that too can be easily solved. Just do this:

```
            LI   R3,5
            BL   @RANDNO
            ANDI R5,1
```

This will generate a random sequence of 1's and 0's without any noticeable pattern. There will of course be cases where six or so 1's appear in a row, and vice versa, but there will over time be as many ones as zeros, at least so far as human perception can tell. Any odd number greater than 5 can be used where we've shown LI R3,5. The result will be about the same. The computer will effectively toss a coin for you.

1.58.5. More Than One Way. . .

That usually ends "to skin a cat", but since we have cats here in the house, we dare not say that. Our cats might come up with the idea that there's more than one way to skin a human. We're talking in this case about the second part of the Sidebar, a nonsense program called "RANDSC". This is the program that randomly scatters the characters from 33 through 126 about the screen at randomly chosen places. You'll notice that a few lines past label CKR1 there are two successive lines that BL @RANDNO. In the original program there was just one. Altogether, there are three BL's to RANDNO in the main loop. Originally, there were two, which meant that only the odd or even characters would show up in any given run. By adding a "dummy" BL line, we make the alternation case again, but we really don't care here because it's not important for the program's purpose.

1.58.6. Why The Alternation?

It's built into the process by which the random number sequence is being generated. To prove this, we modified the program in PART 1 into the program in Part 3, called RAWEX. This one doesn't bother with a divisor, but just displays the "raw" random number that's in R5. These numbers will range from -32768 through 32767. Sure enough, when RAWEX runs, the numbers it generates will be randomly scattered over the range, but will alternate between odd and even, no matter how long the run. When we start a new run, the sequence may start with either an odd or an even number, depending on the starting seed number we got when we "PRESS A KEY TO START", but they'll alternate between odd and even or vice versa from that point on. Thus we have to do something about it if this alternation is not to be seen as a pattern.

1.58.7. Things They Never Taught

Having been through many years of school, from first grade through a Master's degree, including the BSEE work, which included lots of math, you'd expect your author to have a real grasp of mathematical "facts", but in the work leading to this article, we've uncovered a math fact that they never taught us.

t goes like this: If you divide a number by an even number, the remainder will take its even or odd quality from the number you started with. If you divide by an odd number, the remainder may be odd or even, not necessarily following the odd or even quality of the original number. Thus when we use an even divisor in R3, the alternation between odd and even in the "raw" number will show up in the remainders. If an odd divisor is used, that alternation vanishes. Maybe that's not important except in this narrow field of work, but there it is anyway.

1.58.8. A Free Extra "Goodie"

In Part one of the Sidebar, we've put in a new version of an old subroutine. This is called SHWINT, and uses an undocumented GPLLNK service to convert an integer at >835E to a string for display on the screen. The GPLLNK service (DATA >2F7C) treats the integer at >835E as an unsigned number, so the string will range from 0 through 65535. Here, however, we've added a few lines to the beginning so that it will recognize the "sign" bit, and display the negative numbers with a "-" sign. Thus the positive numbers will show up as 0 through 32767, while the negative ones will show up as -1 through -32768.

This will work as shown so long as we're in the E/A Option 3 mode. If you're going to use it in any other way, for example in an Option 5 program, you should use the Warren/Miller GPLLNK routine in the program instead of the REF GPLLNK that we've used here. In this particular program, the numbers to be displayed are all positive, but we left in the "negative handling" part just to provide our readers with another useful subroutine.

There's one other thing you can do sometimes to improve the randomness of your numbers. In many programs, there's a key loop somewhere to wait for the user's input. If that key loop includes LIMI 2 and LIMI 0, as ours always do, you can put in an instruction like A @>8378,@>83C0 right after one of those BL @KEYLOO instruction, and that will adjust the seeding of the random number process each time that user input gets taken. The same can be done after a string or number input routine that uses LIMI 2 and LIMI 0. Doing this will throw in another layer of unpredictability to your random number sequence, as nobody can know in advance what number will be at >8378 in such cases.

We're making no promises about next month's column, except that it won't concern Random Numbers. Even your author can get tired of beating a dead horse.

```
* SIDEBAR 58
* THREE PROGRAMS
* EXPERIMENTS IN RANDOMNESS
*
* PART ONE
*
* RANDEX/S
* RANDOM NUMBERS
*
*  PUBLIC DOMAIN
*  Code by Bruce Harrison
*  10 FEB 1995
*
       REF  VSBW,VMBW,KSCAN,GPLLNK
       DEF  START
*
STATUS EQU  >837C
*
START  LWPI WS           LOAD OUR WORKSPACE
       LI   R3,4         RANGE 0-3
*      LI   R3,5         RANGE 0-4
```

```
       LI   R0,11*32+5   POINT ROW 12, COL 6
       LI   R1,PAK       "PRESS A KEY"
       LI   R2,20        20 CHARS
       BLWP @VMBW        WRITE THAT
SEED   MOVB @>83D7,R10   LOW BYTE TIMEOUT COUNTER
       BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            STOP THEM
       CB   @>837C,@ANYKEY KEY PRESSED?
       JNE  SEED         IF NOT, REPEAT
       A    @>8378,R10   ADD VERTICAL INT TIMER BYTE
       MOV  R10,@>83C0   PLACE R10 AT SEED
CLEAR  LI   R1,>2000     SPACE IN L.B. R1
       LI   R2,768       768 CHARS
       CLR  R0           SCREEN ORIGIN
CLRLP  BLWP @VSBW        WRITE ONE
       INC  R0           NEXT SPOT
       DEC  R2           DEC COUNT
       JNE  CLRLP        RPT. IF NOT ZERO
       CLR  R0           SCREEN ORIGIN
BUILD  CI   R0,>2FE      COMPARE TO 2 BYTES BEFORE END
       JLT  BUILD1       IF LESS, PROCEED
       JMP  CLEAR        ELSE RE-CLEAR SCREEN
BUILD1 BL   @RANDNO      GET A RANDOM NUMBER
*      AI   R5,-1        SUBTRACT 1
*      JLT  BUILD1       RPT IF BELOW 0
       MOV  R5,@>835E    PLACE R5 AT >835E
       BL   @SHWINT      DISPLAY ON-SCREEN
       A    R2,R0        ADD LENGTH
       INC  R0           SKIP A SPACE
       MOV  R0,R1        PUT SCREEN ADDR IN R1
       ANDI R1,>001F     MASK OFF TO >1F
       CI   R1,>1E       NEAR END OF ROW?
       JLT  SCN          IF LESS, OKAY
       ANDI R0,>FFE0     MASK TO START OF ROW
       AI   R0,>20       MOVE DOWN ONE ROW
SCN    BLWP @KSCAN       SCAN KEYBOARD
       CB   @ANYKEY,@>837C KEY PRESSED?
       JEQ  EXIT         IF SO, EXIT
STDLY  CLR  @>83D6       CLEAR TIMEOUT COUNTER
       LI   R4,12        12 = 6/60THS = 1/10 SECOND DELAY
DLY    LIMI 2            ALLOW INTS
       LIMI 0            STOP INTS
       C    @>83D6,R4    COMPARE TIMEOUT TO 1/10
       JLT  DLY          IF LESS, REPEAT
       JMP  BUILD        ELSE JUMP BACK
EXIT   LWPI >83E0        LOAD GPL WS
       B    @>6A         EXIT TO GPL INT.
RANDNO LI   R4,28645     BIG NUMBER IN R4
       MPY  @>83C0,R4    MULT. BY SEED
```

```
       AI   R5,31417     ADD BIG NUMBER
       MOV  R5,@>83C0    PUT BACK AT SEED
       CLR  R4           CLEAR HIGH WORD
       DIV  R3,R4        DIVIDE BY R3
       RT
*
* PUT INTEGER AT >835E, THEN BL HERE
* WILL DISPLAY AT CURRENT R0 POSITION
*
SHWINT MOV  @>835E,R2    GET NUMBER
       JEQ  SHWIN0       IF ZERO, JUMP
       JGT  SHWIN0       IF POSITIVE, JUMP
       NEG  R2           ELSE TAKE 2'S COMPLEMENT
       MOV  R2,@>835E    PLACE AT >835E
       LI   R1,>2D00     "-" IN L.B. R1
       BLWP @VSBW        DISPLAY -
       INC  R0           POINT TO NEXT SPOT
SHWIN0 CLR  @STATUS      CLEAR GPL STATUS BYTE
       BLWP @GPLLNK      USE GPL LINK
       DATA >2F7C        CONVERT INTEGER TO STRING
       MOVB @>8361,R2    GET LENGTH
       SRL  R2,8         RIGHT JUST.
       MOVB @>8367,R1    GET L.B. ADDRESS
       SRL  R1,8         RIGHT JUST.
       AI   R1,>8300     ADD HIGH BYTE >83
       BLWP @VMBW        DISPLAY THE STRING
       RT                THEN RETURN
*
* DATA SECTION
*
WS     BSS  32           OUR OWN WORKSPACE
PAK    TEXT 'PRESS A KEY TO START'
ANYKEY BYTE >20          HEX 20 FOR COMPARISON
       END
*
* PART TWO
*
* RANDSC/S
* RANDOM SCREEN
*
*  PUBLIC DOMAIN
*  Code by Bruce Harrison
*  10 FEB 1995
*
       REF  VSBW,VMBW,KSCAN
       DEF  START
*
*
*
START  LWPI WS           LOAD OUR WORKSPACE
```

```
        LI   R0,11*32+5   ROW 12, COL 6
        LI   R1,PAK       "PRESS A KEY"
        LI   R2,20        20 CHARS
        BLWP @VMBW        WRITE
SEED    MOVB @>83D7,R10   L.B. SCRN TIMEOUT TO H.B. R10
        BLWP @KSCAN       SCAN KEYBOARD
        LIMI 2            INTS ON
        LIMI 0            INTS OFF
        CB   @>837C,@ANYKEY KEY PRESSED?
        JNE  SEED         RPT IF NOT
        A    @>8378,R10   ADD VERT TIMER TO R10
        MOV  R10,@>83C0   PLACE AT SEED
        LI   R1,>2040     SPACE AND "@" IN R1
BUILD   CLR  R3           REG 3 = 0
        SWPB R1           SWAP BYTES IN R1
        LI   R9,BUFF2     POINT AT BUFFER 2
BRAN0   MOV  R3,*R9+      PLACE A WORD IN TABLE
        INC  R3           INC COUNT
        CI   R3,>300      COMPARE POINTER
        JLT  BRAN0        IF LESS, BACK
        MOV  R3,R6        PUT R3 INTO R6
        SLA  R6,1         DOUBLE THAT
BRAN1   BL   @RANDNO      GET A RANDOM NUMBER
        SLA  R5,1         DOUBLE REMAINDER
        MOV  @BUFF2(R5),R0   MOVE WORD FROM TABLE TO FAC
        BLWP @VSBW        WRITE A CHARACTER
        BLWP @KSCAN       SCAN KEYBOARD
        LIMI 2            INTS ON
        LIMI 0            INTS OFF
        CB   @ANYKEY,@>837C KEY?
        JNE  CKR1         JUMP IF NOT
        JMP  EXIT         ELSE EXIT
CKR1    CB   R1,@ANYKEY   SPACE IN L.B. R1?
        JEQ  DEC6         IF SO, JUMP
        MOV  R3,R9        SAVE R3
        MOV  R5,R10       SAVE CURRENT R5
        LI   R3,127-33    RANGE = 94
        BL   @RANDNO      GET RANDOM NUMBER
        BL   @RANDNO      TWICE
        AI   R5,33        ADD BOTTOM TO R5
        SWPB R5           SWAP
        MOVB R5,R1        PLACE BYTE IN L.B. R1
        MOV  R9,R3        GET OLD R3 BACK
        MOV  R10,R5       AND OLD R5 BACK
DEC6    DECT R6           SUBTR. 2 FROM R6
        MOV  @BUFF2(R6),@BUFF2(R5) REPLACE WORD JUST TAKEN
        DEC  R3           SUBTR 1 FROM R3
        JNE  BRAN1        IF NOT ZERO, REPEAT
        CB   R1,@ANYKEY   SPACE IN R1?
        JNE  STDLY        IF NOT, JUMP
```

```
SHWMSG MOV  R1,R3        STASH R1 IN R3
       LI   R2,16        16 CHARS
       LI   R1,FINMSG    "FINISHED ..."
       LI   R0,11*32+8   ROW 12, COL 9
       BLWP @VMBW        WRITE MESSAGE
       MOV  R3,R1        GET OLD R1 BACK
STDLY  CLR  @>83D6       CLEAR TIMEOUT
       LI   R4,240       TWO SECONDS DELAY
DLY    LIMI 2            INTS ON
       LIMI 0            INTS OFF
       C    @>83D6,R4    CHECK TIME
       JLT  DLY          IF LESS, REPEAT
       JMP  BUILD        ELSE TO BUILD
EXIT   LWPI >83E0        LOAD GPL WS
       B    @>6A         EXIT TO GPL INT.
RANDNO LI   R4,28645     BIG NUMBER IN R4
       MPY  @>83C0,R4    MULT. BY SEED
       AI   R5,31417     ADD BIG NUMBER
       MOV  R5,@>83C0    PUT BACK AT SEED
       CLR  R4           CLEAR HIGH WORD
       DIV  R3,R4        DIVIDE BY R3
       RT                THEN RETURN
*
* DATA SECTION
*
WS     BSS  32           OUR OWN WORKSPACE
BUFF2  BSS  768*2        SELECTION LIST
FINMSG TEXT 'FINISHED A CYCLE'
PAK    TEXT 'PRESS A KEY TO START'
ANYKEY BYTE >20          HEX 20 FOR COMPARISON
       END
*
* PART THREE
*
* RAWEX/S
* RAW RANDOM NUMBERS
* ILLUSTRATES ODD/EVEN ALTERNATION
*
*  PUBLIC DOMAIN
*  Code by Bruce Harrison
*  10 FEB 1995
*
       REF  VSBW,VMBW,KSCAN,GPLLNK
       DEF  START
*
STATUS EQU  >837C
*
START  LWPI WS           LOAD OUR WORKSPACE
       LI   R0,11*32+5   POINT ROW 12, COL 6
       LI   R1,PAK       "PRESS A KEY"
```

```
       LI   R2,20         20 CHARS
       BLWP @VMBW         WRITE THAT
SEED   MOVB @>83D7,R10    LOW BYTE TIMEOUT COUNTER
       BLWP @KSCAN        SCAN KEYBOARD
       LIMI 2             ALLOW INTERRUPTS
       LIMI 0             STOP THEM
       CB   @>837C,@ANYKEY KEY PRESSED?
       JNE  SEED          IF NOT, REPEAT
       A    @>8378,R10    ADD VERTICAL INT TIMER BYTE
       MOV  R10,@>83C0    PLACE R10 AT SEED
CLEAR  LI   R1,>2000      SPACE IN L.B. R1
       LI   R2,768        768 CHARS
       CLR  R0            SCREEN ORIGIN
CLRLP  BLWP @VSBW         WRITE ONE
       INC  R0            NEXT SPOT
       DEC  R2            DEC COUNT
       JNE  CLRLP         RPT. IF NOT ZERO
       CLR  R0            SCREEN ORIGIN
BUILD  CI   R0,>2FA       COMPARE TO 6 BYTES BEFORE END
       JLT  BUILD1        IF LESS, PROCEED
       JMP  CLEAR         ELSE RE-CLEAR SCREEN
BUILD1 BL   @RANDNO       GET A RANDOM NUMBER
       A    R2,R0         ADD LENGTH
       INC  R0            SKIP A SPACE
       MOV  R0,R1         PUT SCREEN ADDR IN R1
       ANDI R1,>001F      MASK OFF TO >1F
       CI   R1,>1A        NEAR END OF ROW?
       JLT  SCN           IF LESS, OKAY
       ANDI R0,>FFE0      MASK TO START OF ROW
       AI   R0,>20        MOVE DOWN ONE ROW
SCN    BLWP @KSCAN        SCAN KEYBOARD
       CB   @ANYKEY,@>837C KEY PRESSED?
       JEQ  EXIT          IF SO, EXIT
STDLY  CLR  @>83D6        CLEAR TIMEOUT COUNTER
       LI   R4,28         14/60THS DELAY
DLY    LIMI 2             ALLOW INTS
       LIMI 0             STOP INTS
       C    @>83D6,R4     COMPARE TIMEOUT TO 1/10
       JLT  DLY           IF LESS, REPEAT
       JMP  BUILD         ELSE JUMP BACK
EXIT   LWPI >83E0         LOAD GPL WS
       B    @>6A          EXIT TO GPL INT.
RANDNO LI   R4,28645      BIG NUMBER IN R4
       MPY  @>83C0,R4     MULT. BY SEED
       AI   R5,31417      ADD BIG NUMBER
       MOV  R5,@>83C0     PUT BACK AT SEED
SHWINT MOV  R5,@>835E     GET NUMBER
       JEQ  SHWIN0        IF ZERO, JUMP
       JGT  SHWIN0        IF POSITIVE, JUMP
       NEG  R5            ELSE TAKE 2'S COMPLEMENT
```

```
       MOV  R5,@>835E    PLACE AT >835E
       LI   R1,>2D00     "-" IN L.B. R1
       BLWP @VSBW        DISPLAY -
       INC  R0           POINT TO NEXT SPOT
SHWIN0 CLR  @STATUS      CLEAR GPL STATUS BYTE
       BLWP @GPLLNK      USE GPL LINK
       DATA >2F7C        CONVERT INTEGER TO STRING
       MOVB @>8361,R2    GET LENGTH
       SRL  R2,8         RIGHT JUST.
       MOVB @>8367,R1    GET L.B. ADDRESS
       SRL  R1,8         RIGHT JUST.
       AI   R1,>8300     ADD HIGH BYTE >83
       BLWP @VMBW        DISPLAY THE STRING
       RT                THEN RETURN

*
* DATA SECTION
*
WS     BSS  32           OUR OWN WORKSPACE
PAK    TEXT 'PRESS A KEY TO START'
ANYKEY BYTE >20          HEX 20 FOR COMPARISON
       END
```

1.59. The Art Of Assembly — Part 59. Six And A Quarter Cents

By Bruce Harrison

The subtitle of today's column answers the question "If a Quarter is two bits, what's half a bit?" That's just our silly way of telling you that today's column is on a new and different topic, the so-called Half-Bitmap Mode. We've actually had letters from readers asking about this topic, and until March of 1995 we've never been able to answer those questions. Back in January of '94 we got some tentative information about this strange mode from Harry Wilhelm, but it wasn't until Mr. Oscar A. Ros, of Sylmar CA, asked in the Reader to Reader column (Feb 95 *MICROpendium*) that we decided to try an experiment with it.

To start with, the term "Half-Bitmap" is misleading. Yes there are computer scientist types who have some concept of a half of a bit, but that has nothing at all to do with this special VDP RAM operating mode on the TI. Harry Wilhelm suggests that this mode should be called "Enhanced Graphics Mode", and we think that's a much better name for what it allows us to do. From here on in this column, we'll use Harry's term.

In the normal Graphics mode one can define a total of 256 characters with eight byte patterns in the Pattern Table of VDP RAM. Color definitions, however, use only one byte per set of eight characters, so we don't have the freedom to "color" a specific character to a particular foreground and background choice without changing seven other characters' colors. In Enhanced Graphic Mode, we can change the color scheme for just a single character, and can even use more than one color scheme within a character. That's so because in Enhanced Graphics Mode each defined character has eight bytes available for its color scheme, as well as the eight bytes to define the character.

1.59.1. The Sidebar Program

Today's Sidebar has a complete program that illustrates the use of Enhanced Graphics Mode. Parts of this are modified from the source code for our Drawing program, which operates in full Bitmap Mode. By making some small changes in the SETHB subroutine from that source, we were able to explore the capabilities of Enhanced Graphics Mode. The results are interesting, to say the least. We can do some things that could not be done in the normal graphics mode. For example, we can even change the color scheme of an individual character into a rainbow pattern where each row of the character is in a different combination of colors.

The Sidebar is set up so that the section that sets up the Enhanced Graphics mode is a subroutine. That should make it easier for you to excerpt that part for use in your own programs. Like the full-blown Bitmap Mode, the Enhanced Graphics mode needs to have the color table, the pattern table, and the screen image table re-located from their normal places in the VDP Memory. We've put the Pattern Table at 0000, the Color Table at >2000, and the Screen Image Table at >1800. Those are the same locations we used in the full Bitmap Mode, but the lengths of the pattern and color tables is shorter by >1000 bytes each. The Pattern table contains 256 8-byte character patterns, and the Color table contains 256 8-byte color definitions, one for each character. Thus in our present case the pattern table runs from >0000 through >07FF, the screen image table runs from >1800 through >1AFF, and the color table runs from >2000 through >27FF. This leaves some significant blocks of VDP RAM available for other purposes. You might, for example, use the space from >0800 through >17FF for PABs and Buffers to handle file operations, and might make other use of the spaces from >1B00 through >1FFF and >2800 through >37FF.

The way Enhanced Graphics is set up is different from the full Bitmap case in only two places in the subroutine SETHB. The value written to VDP Register 4 is >00 instead of >03, and the value written to VDP Register 3 is >9F instead of >FF. We made one other change in this subroutine, adding a VWTR to Register 1 with value >A0. That causes the screen to blank, so that we won't see all kinds of garbage on the screen until everything has been set. Unblanking is done in the main program, after the colors and patterns have been put in place and the screen image table has been filled with spaces. We "unblank" the screen by writing >E0 to VDP Register 1.

You'll recall that in the full Bitmap Mode, we filled the Screen Image table with three sets of the characters from 0 through 255. Here we don't do that, but simply write 768 spaces into the screen image table to provide a cleared screen. We use the pattern table in the "normal" fashion, placing eight bytes of definition for each character. The color table gets eight bytes for the color corresponding to each character definition. We then write on the screen in "conventional" mode, simply placing one byte at the row and column position where we want the characters to appear. (We do have to add the >1800 offset, of course, to write to the screen.) This differs sharply from the full Bitmap case, where we wrote individual bits into the pattern table to place pixels on the screen.

For our demo purposes, we just used the character definitions that are provided when the program starts up, so we have definitions for the characters from 33 through 126. We replicate these into patterns for characters 161 through 254, so that we can show two sets of characters in the initial screen display.

1.59.2. Walking Through It

The initial part of the program just does the same steps that we did in the opening of some other Bitmap programs. We stash away the original color table and character definitions, plus the six bytes from the address pointed to by >8370. In this particular program, those things get put back in place just before we exit. In a normal start from E/A Option 3, the character definitions are available only up through character 126. Here, we take those characters twice into our storage at CHRTBL, so that characters from 161 through 254 can be used in our Enhanced Graphics experiment. That takes us down to the line that says BL @SETHB.

The subroutine SETHB starts by writing >A0 to VDP Register 1. This makes the screen blank out. That's followed by a whole series of VWTR operations to set up the other VDP registers. In this program we won't be using sprites, so we've set both the sprite attribute table and the sprite definition table to >3800, just to get them out of our way. Writing >02 to VDP register 0 puts us into the Enhanced Graphics mode. To be safe, we clear the byte at >837A and place a "delete" value in the first byte of the sprite attribute list so that no sprites will appear.

On return from SETHB, our screen is still blank while we put some things into place in VDP RAM. First we place the 256 character definitions into the Pattern Descriptor Table. Next, we point at the Screen Image Table (SCRORG), put the space character in R1's left byte, and write 768 spaces to clear the screen.

The color table gets set up in two halves. The first half is filled with Blue on White, and the second half with White on Blue. We did that in this case just to make it easy to see on the screen which were the characters from 33 through 126 and which were the "second set", from 161 through 254. After both halves of the color table are filled, we unblank the screen, which at this point is cleared to all white. Now we write the two sets of characters onto the screen, twice each. At this point the screen shows the cycle from ! through ~ repeated four times in alternating color schemes.

Now to prove we can do what we said, the program changes the color schemes for certain characters. The ! and 0 of the first set get changed to White on Green. The 9 of the first set gets changed to a "Rainbow" color scheme that goes from White on Dark Red to White on Dark Blue from top to bottom of the character. Then just to show we can do this to the second character set, we change the z (lower case) of the second set to Magenta on White.

Now we do something a bit more tricky. We put the legend "HALF-BITMAP MODE" at the bottom of the screen in rainbow colors. We do this by "borrowing" the space in the character table and color table that serves for characters 0 through 18. Since our display doesn't need those characters, we copy the patterns for the 19 characters in the legend at HBSTR into the Pattern Table starting at 0000, and put the "RBOW" color scheme into the color table space corresponding. Once the patterns and colors are in those places, we set R0 for Row 23, Col 7 in the line LI R0,22*32+6+SCRORG, clear R1 so we start with character 0, then write characters 0 through 18 to the screen. Thus we have a legend at the bottom of the screen in "rainbow" color scheme.

After that's all done, the program enters a key loop so you'll be able to see what it's done to the screen. Pressing a key will cause the program to re-set everything back to the normal Graphics Mode, clear the graphics screen, and then exit to E/A's PRESS ENTER TO CONTINUE prompt.

This program, like many of our Sidebar programs, doesn't do anything useful. It should, however, serve as a starting point for your own work. You could change the SETHB subroutine to that it includes the screen clearing process, which we put in the main part of our program. You could define your own graphics characters instead of just "borrowing" TI's character definitions as we did. Harry Wilhelm is of the opinion that with some care one could even get automatic sprite motion to work in this mode, provided you give up the characters from 240 onwards, so that the motion table can be placed in the space from >0780 through >07FF. Harry's probably right as usual, but we haven't tried that yet. (Maybe for next month's column?) In any case our deep thanks go to Harry once again for providing us those little tidbits of information that made this month's column possible. Our thanks also go out to Mr. Oscar A. Ros for sparking our interest in the "Half-Bitmap Mode".

Next month's topic is as yet undecided, but we're leaning toward more experimenting with Enhanced Graphics Mode, unless some other vital topic presents itself.

```
* SIDEBAR 59
* A COMPLETE PROGRAM
* TO DEMO THE ENHANCED GRAPHICS MODE
* (A.K.A. HALF-BITMAP MODE)
*
* 27 MAR 1995
* CODE BY BRUCE HARRISON
* PUBLIC DOMAIN
       DEF  START         DEFINE ENTRY
       REF  VMBW,VWTR,VSBW,VMBR,KSCAN REF UTILS
STATUS EQU  >837C         GPL STATUS BYTE
SCRORG EQU  >1800         SCREEN TABLE
START  LWPI WS            LOAD OUR WS
       LI   R0,>380       POINT AT COLOR TABLE
       LI   R1,SAVCLR     AND AT STORAGE SPACE
       LI   R2,32         32 BYTES TO GET
       BLWP @VMBR         READ COLOR TABLE
       MOV  @>8370,R0     GET VDP ADDR FROM >8370
       LI   R1,ANYKEY+1   POINT AT STORAGE BUFFER
       LI   R2,6          SIX BYTES TO READ
       BLWP @VMBR         READ THOSE
*
* THE FOLLOWING TAKES THE CHAR DEFS FOR CHARS
* 0 THROUGH 127 FROM EXISTING DEF TABLE, THEN
* PUTS THEM TWICE INTO OUR STORAGE AT CHRTBL
* THIS MAKES CHARS 128 THRU 255 THE SAME AS
* CHARS 0 THROUGH 127
*
       LI   R0,>800       POINT AT CHARACTER TABLE
       LI   R1,CHRTBL     AND AT BUFFER STORAGE
       LI   R2,128*8      128 CHARACTER DEFINITIONS
       BLWP @VMBR         STASH CHARACTER DEFS
       A    R2,R1         MOVE TO SECOND HALF CHRTBL
       BLWP @VMBR         DUPLICATE CHARS IN SECOND HALF
*
```

```
* SETHB SETS THE VDP FOR ENHANCED GRAPHICS MODE
*
       BL   @SETHB       SET TO HALF-BITMAP
*
* THE FOLLOWING PUTS THE CHARACTER DEFINITIONS
* INTO THE VDP RAM FROM 0 THRU >800 AS TWO DUPLICATES
* OF THE CHARS 0 THRU 127
*
       CLR  R0           START OF CHAR DEF TABLE
       LI   R1,CHRTBL    THE SAVED CHARACTERS
       LI   R2,256*8     ALL 256 DEFINITIONS
       BLWP @VMBW        WRITE THOSE
*
* NEXT CLEARS THE EGM SCREEN IMAGE TABLE
*
       LI   R0,SCRORG    POINT AT SCREEN ORIGIN
       LI   R1,>2000     SPACE IN LB R1
       LI   R2,768       768 CHARS IN SCREEN
CLRLP  BLWP @VSBW        WRITE A SPACE
       INC  R0           NEXT SPOT
       DEC  R2           DEC COUNT
       JNE  CLRLP        REPEAT IF NOT ZERO
*
* THIS SECTION SETS THE COLORS FOR CHARS 0 THRU 127
* TO BLUE ON WHITE, AND THOSE FOR CHARS 128 THRU 255
* TO THE OPPOSITE COLORS
*
       LI   R0,>2000     COLOR TABLE
       LI   R1,>4F00     BLUE ON WHITE
       LI   R2,>400      HALF OF COLOR TABLE
COLSET BLWP @VSBW        WRITE A BYTE
       INC  R0           NEXT POSITION
       DEC  R2           DEC COUNT
       JNE  COLSET       REPEAT IF NOT ZERO
       LI   R0,>2400     2ND HALF OF COLOR TABLE
       LI   R1,>F400     WHITE ON BLUE
       LI   R2,>400      HALF OF TABLE
COLSE2 BLWP @VSBW        WRITE
       INC  R0           NEXT BYTE
       DEC  R2           DEC COUNT
       JNE  COLSE2       LOOP IF NOT ZERO
*
* AFTER ALL THAT'S DONE, WE UNBLANK THE SCREEN
* AT THIS POINT THE SCREEN IS WHITE WITH GREEN BORDER
*
       LI   R0,>1E0      UNBLANK SCREEN
       BLWP @VWTR        BY VWTR
*
* THE NEXT PART PUTS TWO REPEATS OF THE TWO SETS OF
* CHARACTERS (33 - 126 AND 161 - 254) ON SCREEN
```

```
* THE LOWER SET IS IN BLUE ON WHITE, THE UPPER IS
* IN WHITE ON BLUE
*
       LI   R0,SCRORG    POINT AT >1800
       LI   R4,2         TWO CYCLES
WRTOM  LI   R1,>2100     START WITH CHAR 33 (!)
       LI   R2,127-33    ALL DEFINED CHARS THRU 126 (~)
WRTCH  BLWP @VSBW        WRITE ONE
       AI   R1,>100      NEXT CHAR
       INC  R0           NEXT SCREEN LOCATION
       DEC  R2           DEC COUNT
       JNE  WRTCH        RPT TIL ZERO
       LI   R1,>A100     SET FOR SECOND SET'S (!)
       LI   R2,127-33    SAME COUNT
       INCT R0           MOVE TO START OF ROW
       AI   R0,64        SKIP TWO ROWS
WRTCH2 BLWP @VSBW        WRITE ONE
       AI   R1,>100      NEXT CHAR
       INC  R0           NEXT SCREEN LOCATION
       DEC  R2           DEC COUNT
       JNE  WRTCH2       RPT TIL ZERO
       AI   R0,66        ADD 66 TO R0
       DEC  R4           DEC R4 COUNT
       JNE  WRTOM        ANOTHER IF NOT ZERO
*
* FOLLOWING SELECTIVELY CHANGES COLORS FOR SOME OF THE
* CHARACTERS TO ILLUSTRATE COLORING SINGLE CHARACTERS
*
       LI   R0,33*8+>2000  COLOR FOR FIRST !
       LI   R1,NEWCOL    NEW COLOR SCHEME (WHITE ON GREEN)
       LI   R2,8         EIGHT BYTES
       BLWP @VMBW        RE-COLOR THE !
       LI   R0,48*8+>2000 POINT AT ZERO CHARACTER'S COLOR
       BLWP @VMBW        RE-COLOR THE 0
       LI   R0,57*8+>2000 POINT AT NINE CHAR'S COLOR
       LI   R1,RBOW      RAINBOW
       BLWP @VMBW        RE-COLOR THE 9
       LI   R0,122+128*8+>2000 SECOND SET'S L.C. Z COLOR
       LI   R1,MAG       MAGENTA
       BLWP @VMBW        COLOR THAT
*
* FOLLOWING SECTION PUTS THE LEGEND 'HALF-BITMAP MODE'
* ON THE SCREEN IN RAINBOW COLOR SCHEME
*
       LI   R3,HBSTR     TITLE STRING
       LI   R4,18        18 CHARACTERS
       LI   R2,8         8 BYTES PER DEF
       CLR  R13          START AT 0
LEGLP  MOVB *R3+,R1      GET A CHARACTER FROM STRING
       SRL  R1,8         RT. JUSTIFY
```

```
       SLA  R1,3          MULT. BY 8
       AI   R1,CHRTBL     ADD TABLE OFFSET
       MOV  R13,R0        GET R13 INTO R0
       BLWP @VMBW         WRITE CHAR DEF
       LI   R1,RBOW       RAINBOW COLOR
       AI   R0,>2000      COLOR TBL OFFSET
       BLWP @VMBW         WRITE COLORS
       A    R2,R13        ADD 8 TO R13
       DEC  R4            DEC CHAR COUNT
       JNE  LEGLP         RPT IF NOT 0
       LI   R0,22*32+6+SCRORG ROW 23, COL 7
       CLR  R1            ZERO IN R1
       LI   R2,19         19 CHARACTERS
LEGWRT BLWP @VSBW         WRITE A CHAR
       AI   R1,>100       INC LB R1
       INC  R0            NEXT SPOT
       DEC  R2            DEC CNT
       JNE  LEGWRT        BACK IF NOT 0
*
* LAST WAITS FOR A KEYPRESS, THEN RE-SETS TO GRAPHICS
* MODE AND EXITS TO GPL INTERPRETER
*
KEY    BLWP @KSCAN        SCAN KEYBOARD
       LIMI 2             ALLOW INTS
       LIMI 0             STOP INTS
       CB   @ANYKEY,@STATUS  KEY PRESSED?
       JNE  KEY           IF NOT, RE-SCAN
       BL   @SETGM        RESET TO GRAPHICS MODE
       LI   R0,>1E0       UNBLANK SCREEN
       BLWP @VWTR         BY VWTR
       LWPI >83E0         GPL WORKSPACE
       B    @>6A          TO GPL INTERPRETER
*
* SUBROUTINES
*
* FOLLOWING SETS FOR HALF BITMAP MODE
SETHB  LI   R0,>01A0      BLANK SCREEN
       BLWP @VWTR         BY VWTR
       LI   R0,>206       SET TO WRITE VDP REGISTER 2
       BLWP @VWTR         SIT TO >1800  (SCREEN IMAGE TABLE)
       LI   R0,>400       SET TO WRITE TO VDP REG. 4
       BLWP @VWTR         PATTERN TABLE - SPECIAL VALUE
       LI   R0,>39F       SET TO WRITE TO VDP REG 3
       BLWP @VWTR         COLOR TABLE -SPECIAL VALUE
       LI   R0,>607       SET TO WRITE VDP REG 6
       BLWP @VWTR         Sprite descriptor table to >3800
       LI   R0,>570       SET TO WRITE VDP REG 7
       BLWP @VWTR         Sprite attribute list to >3800
       LI   R0,2          SET R0 TO WRITE 2 TO VDP REGISTER ZERO
       BLWP @VWTR         SET TO M3 MODE (BITMAP)
```

```
       CLR  R1           CLEAR R1
       MOVB R1,@>837A    NO SPRITES IN MOTION
       LI   R1,>D000     SPRITE DELETE
       LI   R0,>3800     AT DESCRIPTOR TABLE
       BLWP @VSBW        WRITE THAT
       RT                RETURN
*
* FOLLOWING SETS COMPUTER BACK TO GRAPHICS MODE
*
SETGM  LI   R0,>1A0      SET TO WRITE VDP REG 1 (BLANK SCREEN)
       BLWP @VWTR        WRITE
       LI   R0,>200      SET TO WRITE VDP REG 2
       BLWP @VWTR        WRITE
       LI   R0,>401      SET TO WRITE VDP REG 4
       BLWP @VWTR        WRITE
       LI   R0,>30E      VDP REG 3
       BLWP @VWTR        WRITE
       LI   R0,>600      VDP REG 6
       BLWP @VWTR        WRITE
       LI   R0,>506      VDP REG 5
       BLWP @VWTR        WRITE
       LI   R0,>380      POINT AT COLOR TABLE
       LI   R1,SAVCLR    AND AT SAVED COLOR DATA
       LI   R2,32        32 BYTES
       BLWP @VMBW        WRITE THE COLOR TABLE BACK
       LI   R0,>800      POINT AT GRAPHICS CHAR TABLE
       LI   R1,CHRTBL    AND AT STORED CHARACTER DATA
       LI   R2,256*8     256 CHARACTERS
       BLWP @VMBW        WRITE CHARACTER DEFS BACK
       LI   R1,>2000     SPACE IN LB R1
       LI   R2,768       768 CHARS
       CLR  R0           SCREEN ORIGIN
CLRGM  BLWP @VSBW        WRITE ONE
       INC  R0           NEXT SPOT
       DEC  R2           DEC COUNT
       JNE  CLRGM        NOT 0, RPT
       MOV  @>8370,R0    ADDRESS FOR FILE STUFF
       LI   R1,ANYKEY+1  SAVED DATA
       LI   R2,6         SIX BYTES
       BLWP @VMBW        WRITE
       CLR  R0           PREP TO WRITE VDP REG 0
       BLWP @VWTR        WRITE THAT TO REMOVE BITMAP
       RT                RETURN
*
* DATA SECTION
*
CHRTBL BSS  8*256        CHAR DEF STORAGE
SAVCLR BSS  32           COLOR TBL STORAGE
WS     BSS  32           OUR WORKSPACE
NEWCOL DATA >FCFC,>FCFC,>FCFC,>FCFC
```

```
MAG    DATA >DFDF,>DFDF,>DFDF,>DFDF
RBOW   DATA >F6F8,>F9F3,>F2FC,>F5F4
HBSTR  TEXT ' HALF-BITMAP MODE '
ANYKEY BYTE 32
       BSS  6
       END
```

1.60. The Art Of Assembly — Part 60. A Moving Experience

By Bruce Harrison

This month we mark yet another year's worth of columns with what we consider a genuine "breakthrough" in Assembly programs. We've overcome another of those "can't be done" things from the Editor/Assembler manual. The manual says we can use Sprites in Bitmap Mode, but not their automatic motion. Well, we've done that!

1.60.1. The First Steps

We started this effort right where last month's column left off, in the Enhanced Graphics Mode (a.k.a. Half-Bitmap). At first, we tried using TI's built-in Automatic Sprite motion, but this soon proved troublesome, because the routine in the console ROM which provides that service is hard-coded to look for the Sprite Attribute Table at >0300 and the Sprite Motion Table at >0780 in VDP RAM. Both of those locations are within the area we were using for our Pattern Descriptor Table. We could get automatic motion for our sprites, but only by giving up the use of 32 character definitions, 16 of them at >0300, and another 16 at >0780. That, we felt, was too great a price to pay, since many programmers would want to be able to use all 256 character definitions at will. There just had to be another way.

1.60.2. The Better Way

Our solution was to ignore the ROM's Sprite Motion service, and instead provide our own through a User Interrupt. To do that, we dis-assembled TI's ROM code, then re-wrote it so that it could use the Sprite Attribute Table and Sprite Motion Table that we'd assigned in our own program. This way, we're able to put the Attribute and Motion Tables at any convenient location in VDP RAM. In the Half-Bitmap case, we set the Pattern Table for sprites at >0800. When we start up, the standard character definitions are already in place at >800, but we will re-define some of them for demo purposes.

To make sure that the ROM sprite motion is disabled, we set the byte at >837A to zero before putting our sprites in place. We use our own memory location (at label SPMOT) to indicate how many sprites are to be in motion. After writing the data for our four sprites into VDP RAM through subroutine SPRITE, we activate the motion by placing the address of USRINT at >83C4. After that, we enter a "key loop" that includes LIMI 2 and LIMI 0, and the user interrupt routine takes care of the desired motion for our sprites.

The Interrupt routine (at label USRINT) performs a BLWP to the vector at SPRINT. The code section starts at label SPRVEL, and there is a separate workspace at label INTWS just for that code. This saved us some memory by allowing R1, R2 and R3 of that WS to be pre-loaded with addresses. In this particular case, we've set the Sprite Attribute Table at >1000 and the Sprite Motion Table at >1080. Other locations in VDP can be used for those, but these locations are handy. Doing it this way leaves all of VDP RAM from >2800 onward available for other things.

1.60.3. In The Sidebar

Once again the Sidebar is a complete program. It's very similar in the early parts to last month's Sidebar, but there are two added EQUates for the ATTLST (Sprite Attribute Table) and the MOTBL (Sprite Motion Table) Otherwise it's pretty much the same down to six lines before label CLRLP. There we put a solid block character definition into the 0th and 255th spots in the Sprite Definition space. These will be used to show that all 256 Sprite Character Patterns are available for our use. A little later we display another string on the screen that says "With Sprites in Motion". Just to illustrate that it can be done, we've put the sprites in magnified form by setting R0 to >01E1 and then doing a VWTR. This way each sprite will use just one character, but will be twice the height and width of the normal screen characters.

Now the program uses the subroutine SPRITE four times. First it places a sprite at dot-row 148 and dot-column 120 that's a red "A" moving upwards at speed 10 and to the right at speed 20. Next it sets Sprite 1 at the same position, but this time it's a yellow "C" moving upwards at speed 10 and to the right at speed 40. Sprites 2 and 3 are solid blocks using the characters 0 and 255,and these move downward at speed 20. Number 2 is green, and 3 is magenta. As soon as the data for these Sprites has been sent to VDP by the subroutine, the main program enables the User Interrupt by placing the address from INTLOC at >83C4. From here on, we just sit at the key loop and wait for a keypress. While waiting, you get to see these four sprites moving around the screen just as if we were in regular Graphics mode. Of course you can tell from the individually colored characters on the screen that we're actually in Half-Bitmap, or Enhanced Graphics mode. Given a keypress, we re-set everything to "normal", kill the User Interrupt by CLR @>83C4, then exit to the GPL Interpreter. That puts us at the "PRESS ENTER TO CONTINUE" prompt. For those of you who get *MICROpendium* on disk, we've included the Object file SIDE60/O, so you can try this out immediately.

1.60.4. That User Interrupt

As we said, the code in our User Interrupt, starting at label SPRVEL, was derived partly from disaasembly of the ROM code starting at >095C and going through >09E8. This required extensive modification to get out the Hard Coded addresses and set up to use the Attribute List and Motion Table where we'd put them. It's written so that as long as the Equates are correct, and the correct value has been sent to VDP register 5, the motion of sprites will proceed without trouble. The main program has to be executing LIMI 2 and LIMI 0 for the sprites to remain in motion, but that's the same in normal Graphics mode.

We start by pointing R9 and R10 at the Motion Table and Attribute List, respectively. Next we get the number of sprites that are in motion into R12. If that's zero, we skip the rest of the subroutine. In this particular case, the number at SPMOT is 4, so sprites 0, 1, 2, and 3 are in motion. The number in motion is right-justified in R12. Now the address from R9 is placed in R8, then is set into place as the VDP address in the Motion Table. The routine reads the Y velocity into R4 and the X velocity into R6. It reads the first auxiliary byte into R5 and the second into R7. Each of these four registers then gets shifted right by four bits, effectively dividing the numbers by 16. It adds R4 to R5 and R6 to R7.

Now the routine shifts over to the Attribute Table, and reads the Y position of the sprite into R4 and the X position into R6. R5, which contains the sum of the Y velocity and Y auxiliary value, each divided by sixteen, is added to the position in R4. The vertical is checked by comparing R4 to certain limit values, and its value is adjusted if necessary. This allows a sprite to wrap around correctly from top to bottom and vice versa.

Now the routine writes the new position bytes to the attribute table, and the new Auxiliary values to the Motion Table. That finishes its work for one sprite. Now we use a trick code to add four to both R9 and R10, so that these point to the locations in the tables for the next sprite. The trick is written C *R9+,*R9+. This just causes R9 to be incremented by two twice, and the results of the comparison itself are ignored. Finally, we DECrement R12, and go back to service the next sprite if it's not zero. When R12 becomes zero, we Return with Workspace Pointer by RTWP. That sends us back to the second instruction at USRINT, which returns to the interrupt handler.

1.60.5. Words Of Caution

If your program accesses files during execution, you'll need to be very careful about the placement of the various tables in VDP RAM. You might even need to re-load them after file accesses, depending where your PABs and Buffers are placed. Also, before using a file, you'll need to put back the six bytes we stashed at ANYKEY+1 into the location pointed to by >8370, or else file access won't happen. In earlier columns, we've covered how to execute a CALL FILES(1) from Assembly, and you may want to do that. That will mean your "file" space reservation will move to >3BE4. (>8370 will contain >3EB3)

The process as we've shown it here allows for the use of up to 32 sprites, each having a choice of any of 256 character patterns. In our own work, we've never found it necessary to use that many of either sprites or character patterns, but you should have the maximum capability available, so we've arranged it that way for you.

Today's column is being kept short, because the Sidebar is so long. This new capability may be the breakthrough you've all been waiting for. Next month, we'll do moving sprites in Full Bitmap Mode, again with a complete program for you to experiment with. See you then.

```
* SIDEBAR 60
* A COMPLETE PROGRAM
* DEMOS ENHANCED GRAPHICS MODE
* WITH AUTOMATIC SPRITE MOTION
* 01 APR 1995
* MODIFIED VERSION FOR SPRITES
* PUBLIC DOMAIN
*    CODE BY Bruce Harrison
*
       DEF  START
       REF  VMBW,VWTR,VSBW,VMBR
       REF  KSCAN
STATUS EQU  >837C
SCRORG EQU  >1800        HALF BIT SCREEN
```

```
MOTBL  EQU  >1080         SPRITE MOTION TABLE
ATTLST EQU  >1000         SPRITE ATTRIBUTE TABLE
START  LWPI WS            LOAD OUR WORKSPACE
       LI   R0,>380       POINT AT COLOR TABLE
       LI   R1,SAVCLR     AND AT STORAGE SPACE
       LI   R2,32         32 BYTES TO GET
       BLWP @VMBR         READ COLOR TABLE INTO STORAGE
       MOV  @>8370,R0     GET VDP ADDR FROM >8370
       LI   R1,ANYKEY+1   POINT AT STORAGE BUFFER
       LI   R2,6          SIX BYTES TO READ
       BLWP @VMBR         READ THOSE INTO BUFFER
       LI   R0,>800       POINT AT CHARACTER TABLE
       LI   R1,CHRTBL     AND AT BUFFER STORAGE
       LI   R2,128*8      128 CHARACTER DEFINITIONS
       BLWP @VMBR         STASH CHARACTER DEFS
       A    R2,R1         MOVE TO SECOND HALF CHRTBL
       BLWP @VMBR         DUPLICATE CHARS IN SECOND HALF
       BL   @SETHB        SET TO HALF-BITMAP
       CLR  R0            START OF CHAR DEF TABLE
       LI   R1,CHRTBL     THE SAVED CHARACTERS
       LI   R2,256*8      ALL 256 DEFINITIONS
       BLWP @VMBW         WRITE THOSE
       LI   R0,>800       SPRITE PATTERN TABLE
       LI   R1,BLOCK      SOLID BLOCK PATTERN
       LI   R2,8          EIGHT BYTES
       BLWP @VMBW         WRITE THAT TO SPRITE CHAR 0
       LI   R0,255*8+>800 LAST PLACE IN SPRITE PAT. TBL.
       BLWP @VMBW         WRITE BLOCK TO SPRITE CHAR 255
       LI   R0,SCRORG     POINT AT SCREEN ORIGIN
       LI   R1,>2000      SPACE IN LB R1
       LI   R2,768        768 CHARS IN SCREEN
CLRLP  BLWP @VSBW         WRITE A SPACE
       INC  R0            NEXT SPOT
       DEC  R2            DEC COUNT
       JNE  CLRLP         REPEAT IF NOT ZERO
       LI   R0,>2000      COLOR TABLE
       LI   R1,>4F00      BLUE ON WHITE
       LI   R2,>400       HALF OF COLOR TABLE
COLSET BLWP @VSBW         WRITE A BYTE
       INC  R0            NEXT POSITION
       DEC  R2            DEC COUNT
       JNE  COLSET        REPEAT IF NOT ZERO
       LI   R1,>F400      WHITE ON BLUE
       LI   R2,>400       2ND HALF OF TABLE
COLSE2 BLWP @VSBW         WRITE
       INC  R0            NEXT BYTE
       DEC  R2            DEC COUNT
       JNE  COLSE2        LOOP IF NOT ZERO
       LI   R0,>1E0       UNBLANK SCREEN
       BLWP @VWTR         BY VWTR
```

```
       LI   R0,SCRORG    POINT AT >1800
       LI   R4,2         TWO CYCLES
WRTOM  LI   R1,>2100     START WITH CHAR 33 (!)
       LI   R2,127-33    ALL DEFINED CHARS THRU 126 (~)
WRTCH  BLWP @VSBW        WRITE ONE
       AI   R1,>100      NEXT CHAR
       INC  R0           NEXT SCREEN LOCATION
       DEC  R2           DEC COUNT
       JNE  WRTCH        RPT TIL ZERO
       LI   R1,>A100     SET FOR SECOND SET'S (!)
       LI   R2,127-33    SAME NUMBER
       AI   R0,66        DOWN 2, OVER 2
WRTCH2 BLWP @VSBW        WRITE ONE
       AI   R1,>100      NEXT CHAR
       INC  R0           NEXT SCREEN LOCATION
       DEC  R2           DEC COUNT
       JNE  WRTCH2       RPT TIL ZERO
       AI   R0,66        DOWN 2, OVER 2
       DEC  R4           DEC R4 COUNT
       JNE  WRTOM        ANOTHER IF NOT ZERO
       LI   R0,33*8+>2000  COLOR FOR FIRST !
       LI   R1,NEWCOL    NEW COLOR SCHEME (WHITE ON GREEN)
       LI   R2,8         EIGHT BYTES
       BLWP @VMBW        RE-COLOR THE !
       LI   R0,48*8+>2000 POINT AT ZERO CHARACTER'S COLOR
       BLWP @VMBW        RE-COLOR THE 0
       LI   R0,57*8+>2000 POINT AT NINE CHAR'S COLOR
       LI   R1,RBOW      RAINBOW
       BLWP @VMBW        RE-COLOR THE 9
       LI   R0,122+128*8+>2000 SECOND SET'S L.C. Z COLOR
       LI   R1,MAG       MAGENTA ON WHITE
       BLWP @VMBW        COLOR THAT
       LI   R3,HBSTR     TITLE STRING
       LI   R4,18        18 CHARACTERS
       LI   R2,8         8 BYTES PER DEF
       CLR  R13          START AT 0
LEGLP  MOVB *R3+,R1      GET A CHARACTER FROM STRING
       SRL  R1,8         RT. JUSTIFY
       SLA  R1,3         MULT. BY 8
       AI   R1,CHRTBL    ADD TABLE OFFSET
       MOV  R13,R0       GET R13 INTO R0
       BLWP @VMBW        WRITE CHAR DEF
       LI   R1,RBOW      RAINBOW COLOR
       AI   R0,>2000     COLOR TBL OFFSET
       BLWP @VMBW        WRITE COLORS
       A    R2,R13       ADD 8 TO R13
       DEC  R4           DEC CHAR COUNT
       JNE  LEGLP        RPT IF NOT 0
       LI   R0,20*32+6+SCRORG ROW 21, COL 7
       CLR  R1           ZERO IN R1
```

```
       LI   R2,19         19 CHARACTERS
LEGWRT BLWP @VSBW         WRITE A CHAR
       AI   R1,>100       INC HIGH BYTE R1
       INC  R0            NEXT SPOT
       DEC  R2            DEC CNT
       JNE  LEGWRT        BACK IF NOT 0
       LI   R0,22*32+5+SCRORG ROW 23, COL 6
       LI   R1,SIMSTR     "SPRITES IN MOTION"
       LI   R2,22         22 CHARS
       BLWP @VMBW         WRITE THAT
       LI   R0,>01E1      ENLARGED SPRITES
       BLWP @VWTR         TO VDP REG 1
       LI   R1,>0400      FOUR SPRITES MOVING
       MOVB R1,@SPMOT     AT SPMOTION
       BL   @SPRITE       SET SPRITE
       DATA 0             NUMBER ZERO
       BYTE 148,120,65,6,-10,20  PARAMETERS (A RED "A")
       BL   @SPRITE       SET SPRITE
       DATA 1             NUMBER ONE
       BYTE 148,120,67,10,-10,-40 PARAMETERS (A YELLOW "C")
       BL   @SPRITE       SET SPRITE
       DATA 2             NUMBER TWO
       BYTE 148,120,0,3,20,20  PARAMETERS (GREEN BLOCK)
       BL   @SPRITE       SET SPRITE
       DATA 3             NUMBER THREE
       BYTE 148,120,255,13,20,-40 PARAMETERS (MAGENTA BLOCK)
       MOV  @INTLOC,@>83C4 ENABLE USER INTERRUPT
KEY    BLWP @KSCAN        SCAN KEYBOARD
       LIMI 2             ALLOW INTS
       LIMI 0             STOP INTS
       CB   @ANYKEY,@STATUS  KEY PRESSED?
       JNE  KEY           IF NOT, RE-SCAN
       BL   @SETGM        RESET TO GRAPHICS MODE
       LI   R0,>1E0       UNBLANK SCREEN
       BLWP @VWTR         BY VWTR
       CLR  @>83C4        DISABLE USER INTERRUPT
       LWPI >83E0         GPL WORKSPACE
       B    @>6A          TO GPL INTERPRETER
*
* SETUP FOR HALF-BITMAP
* SUBROUTINES FOR HANDLING BITMAP
* OPERATIONS AND TRANSITIONS
*
* FOLLOWING SETS FOR HALF-BITMAP MODE
SETHB  LI   R0,>01A0      BLANK SCREEN
       BLWP @VWTR         BY VWTR
       LI   R0,>206       SET TO WRITE VDP REGISTER 2
       BLWP @VWTR         SIT TO >1800  (SCREEN IMAGE TABLE)
       LI   R0,>400       SET TO WRITE TO VDP REG. 4
       BLWP @VWTR         PATTERN TABLE - SPECIAL VALUE
```

```
       LI   R0,>39F      SET TO WRITE TO VDP REG 3
       BLWP @VWTR        COLOR TABLE -SPECIAL VALUE
       LI   R0,>601      SET TO WRITE VDP REG 6
       BLWP @VWTR        Sprite descriptor table to >0800
       LI   R0,>520      SET TO WRITE VDP REG 5
       BLWP @VWTR        Sprite attribute table to >1000
       LI   R0,2         SET R0 TO WRITE 2 TO VDP REGISTER ZERO
       BLWP @VWTR        SET TO M3 MODE (BITMAP)
       CLR  R1           CLEAR R1
       MOVB R1,@>837A    NO SPRITES IN MOTION
       LI   R1,>D000     SPRITE DELETE
       LI   R0,>1000     AT ATTRIBUTE TABLE
       BLWP @VSBW        WRITE THAT
       RT                RETURN
*
* FOLLOWING SETS COMPUTER BACK TO NORMAL GRAPHICS MODE
*
SETGM  LI   R0,>1A0      SET TO WRITE VDP REG 1 (BLANK SCREEN)
       BLWP @VWTR        WRITE
       LI   R0,>200      SET TO WRITE VDP REG 2
       BLWP @VWTR        WRITE
       LI   R0,>401      SET TO WRITE VDP REG 4
       BLWP @VWTR        WRITE
       LI   R0,>30E      VDP REG 3
       BLWP @VWTR        WRITE
       LI   R0,>600      VDP REG 6
       BLWP @VWTR        WRITE
       LI   R0,>506      VDP REG 5
       BLWP @VWTR        WRITE
       LI   R0,>380      POINT AT COLOR TABLE
       LI   R1,SAVCLR    AND AT SAVED COLOR DATA
       LI   R2,32        32 BYTES
       BLWP @VMBW        WRITE THE COLOR TABLE BACK
       LI   R0,>800      POINT AT GRAPHICS CHAR TABLE
       LI   R1,CHRTBL    AND AT STORED CHARACTER DATA
       LI   R2,256*8     256 CHARACTERS
       BLWP @VMBW        WRITE CHARACTER DEFS BACK
       LI   R1,>2000     SPACE IN LB R1
       LI   R2,768       768 CHARS
       CLR  R0           SCREEN ORIGIN
CLRGM  BLWP @VSBW        WRITE ONE
       INC  R0           NEXT SPOT
       DEC  R2           DEC COUNT
       JNE  CLRGM        NOT 0, RPT
       MOV  @>8370,R0    ADDRESS FOR FILE STUFF
       LI   R1,ANYKEY+1  SAVED DATA
       LI   R2,6         SIX BYTES
       BLWP @VMBW        WRITE
       CLR  R0           PREP TO WRITE VDP REG 0
       BLWP @VWTR        WRITE THAT TO REMOVE BITMAP
```

```
       RT                RETURN
SPRITE LI   R0,ATTLST    POINT AT ATTR. TABLE
       MOV  *R11+,R3     GET SPRITE #
       SLA  R3,2         MULTIPLY BY 4
       A    R3,R0        ADD TO R0
       LI   R2,4         4 BYTES DATA
SPLP   MOVB *R11+,R1     GET A BYTE
       BLWP @VSBW        WRITE THAT
       INC  R0           NEXT BYTE
       DEC  R2           DEC COUNT
       JNE  SPLP         RPT IF NOT 0
       LI   R1,>D000     "DELETE" BYTE
       BLWP @VSBW        WRITE THAT
       LI   R0,MOTBL     POINT AT MOTION TABLE
       A    R3,R0        ADD NUMBER X 4
       MOVB *R11+,R1     GET Y VELOCITY
       BLWP @VSBW        WRITE THAT
       INC  R0           NEXT BYTE
       MOVB *R11+,R1     GET X VELOCITY
       BLWP @VSBW        WRITE THAT
       CLR  R1           ZERO IN R1
       INC  R0           FIRST AUX BYTE
       BLWP @VSBW        WRITE 0 TO 1ST AUX BYTE
       INC  R0           SECOND AUX BYTE
       BLWP @VSBW        WRITE 0 TO SECOND AUX BYTE
       RT                RETURN
USRINT BLWP @SPRINT      USE BLWP TO VECTOR
       RT                THEN RETURN
SPRINT DATA INTWS,SPRVEL WORKSPACE AND CODE
SPRVEL LI   R9,MOTBL     POINT AT MOTION TBL
       LI   R10,ATTLST   AND ATTR. TABLE
       MOVB @SPMOT,R12   HOW MANY IN MOTION?
       JEQ  INTEX        IF ZERO, EXIT
       SRL  R12,8        RIGHT-JUSTIFY
STR8   MOV  R9,R8        MOTION TABLE
       MOVB @INTWS+17,*R1 LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1       HIGH BYTE R8 TO VDP ADDR
       CLR  R4           CLEAR REG 4
       MOVB *R2,R4       GET Y VELOCITY FROM TABLE
       CLR  R6           CLEAR REG 6
       MOVB *R2,R6       GET X VELOCITY FROM TABLE
       SRA  R4,4         SHIFT RIGHT 4 BITS WITH SIGN
       MOVB *R2,R5       GET AUX DATA BYTE
       SRA  R5,4         SHIFT RIGHT WITH SIGN
       A    R4,R5        ADD R4 TO R5
       MOVB *R2,R7       GET 2ND AUX BYTE
       SRA  R6,4         SHIFT RIGHT 4 WITH SIGN
       SRA  R7,4         SAME FOR R7
       A    R6,R7        ADD R6 TO R7
       MOV  R10,R8       ATTRIBUTE ADDR TO R8
```

```
       MOVB @INTWS+17,*R1  LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1         HIGH BYTE R8 TO VDP
       CLR  R4             CLEAR REG 4
       MOVB *R2,R4         Y POSITION TO R4
       A    R5,R4          ADD R5 TO R4
       CI   R4,>C0FF       COMPARE TO LIMIT
       JLE  AC             JUMP IF LOW OR EQUAL
       CI   R4,>E000       COMPARE TO >E000
       JH   AC             JUMP IF HIGH
       MOV  R5,R5          MOVE R5 TO ITSELF
       JGT  AD             IF POSITIVE, JUMP
       AI   R4,>C000       ADD >C000 TO R4
AD     AI   R4,>2000       ADD >2000 TO R4
AC     CLR  R6             CLEAR REG 6
       MOVB *R2,R6         X POSITION TO R6
       A    R7,R6          ADD R7
       ORI  R8,>4000       SET >4000 BIT IN R8 (WRITING)
       MOVB @INTWS+17,*R1 LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1         HIGH BYTE R8 TO VDP ADDR
       MOVB R4,*R3         WRITE NEW Y POSITION
       MOVB R6,*R3         WRITE NEW X POSITION
       MOV  R9,R8          GET MOTION ADDR
       INCT R8             ADD TWO FOR AUX BYTE
       ORI  R8,>4000       SET FOR WRITE
       MOVB @INTWS+17,*R1 LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1         HIGH BYTE R8
       SWPB R5             SWAP R5
       SRL  R5,4           SHIFT R5 LEFT 4 BITS
       MOVB R5,*R3         WRITE NEW AUX BYTE #1
       SWPB R7             SWAP R7
       SRL  R7,4           SHIFT LEFT 4
       MOVB R7,*R3         WRITE NEW AUX BYTE #2
       C    *R9+,*R9+      ADD 4 TO R9
       C    *R10+,*R10+    ADD 4 TO R10
       DEC  R12            DEC COUNT IN R12
       JGT  STR8           IF NOT ZERO, DO ANOTHER SPRITE
INTEX  RTWP                RETURN WITH WORKSPACE POINTER
*
* DATA SECTION
*
SPMOT  BYTE 0              NUMBER OF SPRITES IN MOTION
INTLOC DATA USRINT         ADDRESS OF USER INTERRUPT
CHRTBL BSS  8*256          CHAR DEF STORAGE
SAVCLR BSS  32             COLOR TBL STORAGE
WS     BSS  32             OUR WORKSPACE
INTWS  BSS  2              USER INTERRUPT WORKSPACE R0
       DATA >8C02,>8800,>8C00  PRE-LOADED R1, R2 AND R3
       BSS  24             R4 THRU R15
NEWCOL DATA >FCFC,>FCFC,>FCFC,>FCFC WHITE ON GREEN
MAG    DATA >DFDF,>DFDF,>DFDF,>DFDF MAGENTA ON WHITE
```

```
RBOW   DATA >F6F8,>F9F3,>F2FC,>F5F4 "RAINBOW" COLORS
BLOCK  DATA >FFFF,>FFFF,>FFFF,>FFFF BLOCK CHARACTER
HBSTR  TEXT ' HALF-BITMAP MODE ' MESSAGE 1
SIMSTR TEXT 'With Sprites in Motion' MESSAGE 2
ANYKEY BYTE 32           BYTE FOR COMPARISON
       BSS  6            STORAGE FOR FILE BYTES
       END
```

1.61. The Art Of Assembly — Part 61. Full Bitmap Motion

By Bruce Harrison

As promised, we're going all the way this month. We are offering source code that gives you just about everything you could ask for in the Full Bitmap Mode. How about 32 sprites with their own set of 256 character patterns, and automatic sprite motion! In the demo, there are only four sprites used, but you can extend them to 32, and everything will still work. You can even use the magnified sprites, either the single character type or the four-character type. For those who get *MICROpendium* on disk, there's not only the source file as SIDEBAR61, but the object file SIDE61/O, so you can try out this program right away. The program name is the ever-popular START.

Last month, in the Enhanced Graphics Mode (a.k.a. Half-Bitmap) we tucked away the Sprite Pattern Table, Sprite Attribute Table, and Sprite Motion table in the lower parts of VDP RAM. In the Full Bitmap case, we still managed to tuck away the Attribute and Motion tables in the space between the Screen Image Table and the Color Table. The Motion Table is at >1B00, and the Attribute Table is at >1B80. The Sprite Pattern Table had to be placed at >3800, and it occupies all the space from there to the end of VDP RAM. We copied the standard character set into that space, then put a special "block" character definition into the patterns for characters 0 and 255

To make this demo, we borrowed some source code from Part 42 of this series, so that a nifty little single-pixel spiral gets drawn on the screen, then the legend "Bitmap with Moving Sprites" is placed at the bottom. Now the main event begins, with four sprites appearing nearly dead-center of the screen, then going into "automatic" motion. As in last month's case, their motion is controlled by the same User Interrupt code. This month's source code includes the PLOT, CHAR, and BITSTR routines that first appeared in Part 42. Again, as in last month's program, we've magnified the sprites.

There are some differences in the subroutines we borrowed from Part 42. The SETBM routine, for example, starts by blanking the screen, so that all the setting-up operations are invisible to the user. SETGM, which resets to the Graphics mode, also blanks out the screen, and clears the graphics screen while it's blanked.

The subroutines PLOT, CHAR, and BITSTR are unchanged from Part 42, and the subroutines SPRITE and SPRINT are identical to those in last month's Sidebar. The SPIRAL routine is also identical to the one in Sidebar 42. We've put the data items that are needed by the user interrupt into a section by themselves, just to make that data easier to recognize.

1.61.1. Today's Sidebar

Yes, it's another complete but "nonsense" program. As in previous examples, we've stashed away the stuff from the graphics mode, including the color table and pattern table, and those six bytes from the location pointed to by >8370. Next we write the character definitions into the Sprite Pattern Table at >3800, and then supplement those by putting in a solid block pattern for characters 0 and 255.

Now we use SETBM to put us into Bitmap Mode. This is similar in some respects to the SETHB subroutine in last month's Sidebar, except that this clears the Bitmap screen to all white with a pre-set black foreground color. We changed this from what was in Part 42 in two ways. First, we added the screen blanking in the first two instructions, so the transition would be invisible. Second, we "folded in" the subroutine CPDT, which used to be a separate one. By the way, if you like different starting colors, change the LI statement just before label CT in the SETBM subroutine to any combination you like.

Upon return from SETBM, our main program writes >E1 to VDP Register 1. That unblanks the screen and also sets up for magnified sprites. We put a copy of that byte >E1 at >83D4, which isn't really necessary in this program, but should be done in your own programs so that a keypress won't destroy what you've done to VDP Register 1. This level of magnification means the sprites use only one character, but they're twice the normal height and width of a screen character. The other levels of magnify will work just as well, but you can try that out for yourself by putting in your own four-character sprite definitions.

Now the program uses the PLOT subroutine in a series of loops to create a single-pixel spiral starting near the upper left corner of the screen and working its way to a spot near the center of the screen. When that finishes, we use the BITSTR subroutine to put the legend "Bitmap with Moving Sprites" at the bottom of the screen in white on green.

1.61.2. The Main Event

Now we set our own memory location to allow four sprites in motion, then use the SPRITE subroutine four times to put a red "A", a yellow "B", a green block and a magenta block on the screen as sprites. These start out near the middle of the screen, then move outward in four directions. The B and the Magenta block move to the left, while the A and the Green block move to the right. The blocks move downward and the letters upward. To start the motion itself, we put the address of our user interrupt at >83C4. Our boy Jean-Guy says this stuff on the screen could hypnotize him. Maybe so. After a while, the four sprites will all converge back at their starting point, so you can observe that their motion is accurate.

Now the program just does a BL @KEY, so nothing happens until you press a key. When you do, the program first clears the word at >83C4 to disable the user interrupt, then uses SETGM to put the VDP back to the normal graphics mode. SETGM blanks the screen first, so you won't see any glitches during the transition. It also includes a loop to clear the graphics screen before it unblanks the screen. Finally, the main program puts back those six bytes at the location pointed to by >8370, so that file operations will work when you're back in the Editor/Assembler's control.

That's it! The subroutines here should prove useful to any reader who's trying to make programs that involve Bitmap Mode. If your program needs to have disk access, you'll probably have to go back to the Graphics mode before doing that, as there really isn't anyplace to do file operations in VDP RAM without overwriting at least part of your sprite descriptor table, even if you've done a CALL FILES(1) in the beginning. Of course there is another way, if you can live with having only 128 sprite character definitions. You could place the Sprite Descriptor Table at >1800, but put actual definitions in starting at >1C00. Your sprite character numbers would then start at number 128 and run through 255. Combining that with a CALL FILES(1) would allow you to perform some file operations even while in Bitmap Mode. Be warned, though, that your sprites will stop moving while files are read or written, because the DSR does not allow interrupts. For help on doing CALL FILES, see SIDEBAR44, starting at the bottom of page 7 in the February 1995 issue of *MICROpendium*.

Next month we'll have another set of routines for you. Among other things, we'll show a quick and easy way to reverse single characters or four-character sprite definitions from left to right and vice versa. See you then.

```
* SIDEBAR 61
* A COMPLETE PROGRAM
* DEMONSTRATES FULL BITMAP MODE
* WITH AUTOMATIC SPRITE MOTION
* 02 APR 1995
* PUBLIC DOMAIN
*   CODE BY
*   Bruce Harrison
*
       DEF  START        DEFINE ENTRY POINT
       REF  VWTR,KSCAN,VMBW,VMBR,VSBW,VSBR
MOTBL  EQU  >1B00        SPRITE MOTION TABLE
ATTLST EQU  >1B80        SPRITE ATTRIBUTE TABLE
START  LWPI WS           LOAD OUR WORKSPACE
       LI   R0,>380      POINT AT COLOR TABLE
       LI   R1,SAVCLR    AND AT STORAGE SPACE
       LI   R2,32        32 BYTES TO GET
       BLWP @VMBR        READ COLOR TABLE INTO STORAGE
       MOV  @>8370,R0    GET VDP ADDR FROM >8370
       LI   R1,ANYKEY+1  POINT AT STORAGE BUFFER
       LI   R2,6         SIX BYTES TO READ
       BLWP @VMBR        READ THOSE INTO BUFFER
       LI   R0,>800      POINT AT CHARACTER TABLE
       LI   R1,CHRTBL    AND AT BUFFER STORAGE
       LI   R2,256*8     256 CHARACTER DEFINITIONS
       BLWP @VMBR        STASH CHARACTER DEFS
       LI   R0,>3800     SPRITE PATTERN TABLE
       BLWP @VMBW        WRITE CHAR DEFS THERE
       LI   R1,BLOCK     SOLID BLOCK PATTERN
       LI   R2,8         EIGHT BYTES
       BLWP @VMBW        SPRITE CHAR 0
       LI   R0,255*8+>3800 LAST SPRITE CHAR
```

```
       BLWP @VMBW        A SOLID BLOCK
       BL   @SETBM       GO TO BITMAP
MAGNI  LI   R0,>01E1     UNBLANK AND ENLARGE SPRITES
       BLWP @VWTR        WRITE VDP REG 1
       MOVB @MAGNI+3,@>38D4 PUT >E1 AT >83D4
SPIRAL LI   R12,20       STARTING DOT-COL 20
       LI   R13,10       STARTING DOT-ROW 10
       LI   R14,180      STOP ROW 180
       LI   R15,240      STOP COLUMN 240
       MOV  R13,R8       PUT START ROW IN R8
LINE   MOV  R12,R7       STARTING COL IN R7
       AI   R12,10       ADD 10 TO START COL
       CLR  R9           COLORS BLACK ON WHITE
LOOP1  BL   @PLOT        DRAW ONE PIXEL
       INC  R8           MOVE DOWN ONE ROW
       C    R8,R14       COMPARE TO LIMIT
       JL   LOOP1        IF LOW, REPEAT
       LI   R9,>6000     COLOR DARK RED
LOOP2  BL   @PLOT        DRAW ONE PIXEL
       INC  R7           INC COLUMN
       C    R7,R15       COMPARE TO LIMIT
       JL   LOOP2        IF LOW, REPEAT
       CLR  R9           COLOR BLACK ON WHITE
LOOP3  BL   @PLOT        DRAW ONE PIXEL
       DEC  R8           DEC ROW
       C    R8,R13       COMPARE TO TOP LIMIT
       JH   LOOP3        IF HIGH, REPEAT
       MOV  R13,R8       PUT LIMIT IN R8
       AI   R13,10       ADD 10 TO TOP LIMIT
       LI   R9,>4000     COLOR DARK BLUE
LOOP4  BL   @PLOT        DRAW ONE PIXEL
       DEC  R7           DEC COL
       C    R7,R12       COMPARE TO LEFT LIMIT
       JH   LOOP4        IF HIGH, REPEAT
       AI   R14,-10      SUBTRACT 10 FROM LEFT LIMIT
       AI   R15,-10      AND FROM STOP COLUMN
       C    R13,R14      COMPARE TOP AND BOTTOM LIMITS
       JLT  LINE         IF LESS, BACK TO START
       LI   R8,24        ROW 24
       LI   R7,4         COL 4
       LI   R9,>FC00     COLOR WHITE ON DARK GREEN
       LI   R12,SPISTR   MESSAGE STRING
       BL   @BITSTR      DISPLAY THAT
       LI   R1,>0400     FOUR SPRITES MOVING
       MOVB R1,@SPMOT    AT SPMOTION
       BL   @SPRITE      SET SPRITE
       DATA 0            NUMBER ZERO
       BYTE 88,120,'A',6,-10,20  PARAMETERS (A RED "A")
       BL   @SPRITE      SET SPRITE
       DATA 1            NUMBER ONE
```

```
       BYTE 88,120,'B',10,-10,-40 PARAMETERS (A YELLOW "B")
       BL   @SPRITE      SET SPRITE
       DATA 2            NUMBER TWO
       BYTE 88,120,0,3,20,20  PARAMETERS (GREEN BLOCK)
       BL   @SPRITE      SET SPRITE
       DATA 3            NUMBER THREE
       BYTE 88,120,255,13,20,-40 PARAMETERS (MAGENTA BLOCK)
       MOV  @INTLOC,@>83C4 ENABLE USER INTERRUPT
       BL   @KEY         WAIT FOR KEYSTROKE
       CLR  @>83C4       DISABLE USER INTERRUPT
       BL   @SETGM       SET GRAPHICS MODE
EXIT   MOV  @>8370,R0    GET BACK >8370 ADDRESS
       LI   R1,ANYKEY+1  POINT AT BUFFER STORAGE
       LI   R2,6         SIX BYTES
       BLWP @VMBW        WRITE THOSE BACK TO VDP
       LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         RETURN TO GPL INTERPRETER
*
* SUBROUTINES FOR HANDLING BITMAP
* OPERATIONS AND TRANSITIONS
*
* FOLLOWING SECTION SETS COMPUTER INTO BITMAP MODE
*
SETBM  LI   R0,>1A0      SET FOR BLANK SCREEN
       BLWP @VWTR        WRITE TO VDP REG 1
       LI   R0,>206      SET TO WRITE VDP REGISTER 2
       BLWP @VWTR        SIT TO >1800  (SCREEN IMAGE TABLE)
       LI   R0,>403      SET TO WRITE TO VDP REG. 4
       BLWP @VWTR        PDT TO >0000  (PATTERN DESCRIPTOR TABLE)
       LI   R0,>3FF      SET TO WRITE TO VDP REG 3
       BLWP @VWTR        CT TO >2000   (COLOR TABLE)
       LI   R0,>607      SET TO WRITE VDP REG 6
       BLWP @VWTR        Sprite descriptor table to >3800
       LI   R0,>537      SET TO WRITE VDP REG 5
       BLWP @VWTR        Sprite attribute list to >1B80
       LI   R0,>58       INITIALIZE SCREEN IMAGE TABLE (SIT) (AT >1800)
       MOVB R0,@>8C02    WRITE LOW BYTE VDP ADDRESS
       SWPB R0           SWAP R0
       MOVB R0,@>8C02    WRITE HIGH BYTE VDP ADDRESS
       LI   R0,3         THREE TABLES OF 256 BYTES EACH
       CLR  R1           START WITH ZERO
SIT    MOVB R1,@>8C00    WRITE TO VDP (SELF-INCREMENTING)
       AI   R1,>100      ADD 1 TO HIGH BYTE R1
       JNE  SIT          IF NOT ZERO, REPEAT
       DEC  R0           ELSE DEC COUNT
       JNE  SIT          IF NOT ZERO, REPEAT
       LI   R0,>60       INIT COLOR TABLE  (CT) AT >2000
       MOVB R0,@>8C02    WRITE LOW BYTE OF ADDRESS
       SWPB R0           SWAP R0
       MOVB R0,@>8C02    WRITE HIGH BYTE OF ADDRESS
```

```
       LI   R0,>1800     >1800 BYTES TO WRITE
       LI   R1,>1F00     COLORS ALL BLACK ON WHITE
CT     MOVB R1,@>8C00    WRITE ONE BYTE
       DEC  R0           DEC COUNT
       JNE  CT           IF NOT ZERO, REPEAT
       LI   R0,>40       CLEAR PATTERN DESCRIPTOR TABLE  (PDT) AT >0000
       MOVB R0,@>8C02    WRITE LOW BYTE ADDR
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH BYTE ADDRESS
       LI   R0,>1800     >1800 BYTES TO WRITE
       CLR  R1           ALL ZEROS
PDT    MOVB R1,@>8C00    WRITE ONE
       DEC  R0           DEC COUNT
       JNE  PDT          IF NOT ZERO, REPEAT
       LI   R0,2         SET R0 TO WRITE 2 TO VDP REGISTER ZERO
       BLWP @VWTR        SET TO M3 MODE (BITMAP)
       CLR  R1           ZERO IN R1
       MOVB R1,@>837A    NO SPRITES IN MOTION
       LI   R1,>D000     "DELETE" Y-POSITION
       LI   R0,>1B80     TO SPRITE 0 IN ATTR. TABLE
       BLWP @VSBW        WRITE THAT
       RT                RETURN
*
* FOLLOWING SETS COMPUTER BACK TO NORMAL GRAPHICS MODE
*
SETGM  LI   R0,>01A0     SET TO WRITE VDP REG 1
       BLWP @VWTR        WRITE
       LI   R0,>200      SET TO WRITE VDP REG 2
       BLWP @VWTR        WRITE
       LI   R0,>401      SET TO WRITE VDP REG 4
       BLWP @VWTR        WRITE
       LI   R0,>30E      VDP REG 3
       BLWP @VWTR        WRITE
       LI   R0,>600      VDP REG 6
       BLWP @VWTR        WRITE
       LI   R0,>506      VDP REG 5
       BLWP @VWTR        WRITE
       LI   R0,>380      POINT AT COLOR TABLE
       LI   R1,SAVCLR    AND AT SAVED COLOR DATA
       LI   R2,32        32 BYTES
       BLWP @VMBW        WRITE THE COLOR TABLE BACK
       LI   R0,>800      POINT AT GRAPHICS CHAR TABLE
       LI   R1,CHRTBL    AND AT STORED CHARACTER DATA
       LI   R2,256*8     256 CHARACTERS
       BLWP @VMBW        WRITE CHARACTER DEFS BACK
       CLR  R0           PREP TO WRITE VDP REG 0
       BLWP @VWTR        WRITE THAT TO REMOVE BITMAP
       LI   R1,>2000     SPACE IN HIGH BYTE R1
       LI   R2,768       768 SCREEN POSITIONS
CLRGM  BLWP @VSBW        WRITE ONE
```

```
       INC  R0           NEXT POSITION
       DEC  R2           DEC COUNT
       JNE  CLRGM        RPT IF NOT ZERO
       LI   R0,>01E0     GRAPHICS MODE
       BLWP @VWTR        WRITE TO VDP REG 1
       RT                RETURN
*
* FOLLOWING WRITES ONE PIXEL TO SCREEN AT LOCATION POINTED BY
* R8 (DOT ROW) AND R7 (DOT COLUMN)
*
PLOT   MOV  R7,R3        MOVE DOT COLUMN TO R3
       MOV  R8,R4        AND DOT ROW TO R4
       MOV  R4,R5        DOT ROW ALSO IN R5
       ANDI R5,7         R5 HAS DOT ROW MODULO 8
       SZC  R5,R4        SO DOES R4
       SLA  R4,5         MULTIPLY R4 BY 32
       A    R5,R4        ADD R5, SO R4 HAS DR MOD. 8 * 32 + DR MOD 8
       MOV  R3,R0        MOVE DOT COL TO R0
       ANDI R0,>FFF8     R0 HAS DC - DC MOD 8
       S    R0,R3        R3 HAS DC MOD 8
       A    R4,R0        ADD R4
       SWPB R0           SWAP BYTES
       MOVB R0,@>8C02    WRITE LOW ADDRESS BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH ADDRESS BYTE
       NOP               WASTE TIME
       MOVB @>8800,R1    READ THE BYTE
       SOCB @M(R3),R1    OVERLAY MASK FROM TABLE M
       ORI  R0,>4000     SET THE 4000 BIT IN R0
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE LOW BYTE OF ADDRESS
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH BYTE OF ADDRESS
       NOP               WASTE TIME
       MOVB R1,@>8C00    WRITE MODIFIED BYTE BACK TO VDP
       MOV  R9,R9        IS COLOR TO BE SET?
       JEQ  PLOTX        IF NOT, JUMP AHEAD
       ANDI R0,>3FFF     STRIP OFF "4" FROM R0
       AI   R0,>2000     ADD >2000 TO POINT AT COLOR TABLE ENTRY
       BLWP @VSBR        READ THAT BYTE INTO R1
       MOVB R1,R2        MOVE THE BYTE TO R2
       ANDI R2,>F000     STRIP ALL BUT LEFT NYBBLE
       CB   R2,R9        COMPARE TO LEFT BYTE R9
       JEQ  PLOTX        IF EQUAL, COLOR ALREADY SET
       ANDI R1,>0F00     ELSE STRIP OFF LEFT NYBBLE R1
       AB   R9,R1        REPLACE WITH LEFT NYBBLE R9
       BLWP @VSBW        THEN WRITE COLOR BYTE BACK
PLOTX  RT                RETURN
BITSTR MOV  R11,R15      STASH R11
       MOV  R7,R13       SAVE COLUMN IN R13
```

```
       MOVB *R12+,R4     GET STRING LENGTH BYTE IN R4
       JEQ  BITSX        IF ZERO, SKIP PROCESS
       SRL  R4,8         RIGHT JUSTIFY
BITST0 MOVB *R12+,R6     MOVE ONE BYTE OF STRING TO R6
       SRL  R6,8         RIGHT JUSTIFY
       BL   @CHAR        DISPLAY THAT CHARACTER
       INC  R7           INC COLUMN
       DEC  R4           DEC LENGTH COUNT
       JNE  BITST0       IF NOT ZERO, REPEAT
       MOV  R13,R7       PUT COLUMN BACK IN R7
BITSX  B    *R15         ELSE RETURN
CHAR   MOV  R8,R0        PUT ROW COUNT IN R0
       DEC  R0           DEC TO ZERO-BASE NUMBER
       LI   R2,8         PUT 8 IN R2
       SLA  R0,5         MULTIPLY R0 BY 32
       A    R7,R0        ADD COLUMN
       DEC  R0           DEC TO ZERO-BASE COLUMN
       SLA  R0,3         MULTIPLY R0 BY 8
       MOV  R6,R1        PUT CHARACTER FROM R6 INTO R1
       SLA  R1,3         MULTIPLY BY 8
       AI   R1,CHRTBL    ADD START OF STORED CHARACTER DEFINITIONS
       BLWP @VMBW        WRITE 8 BYTES TO VDP RAM
       MOV  R9,R9        CHECK FOR COLOR CHANGE
       JEQ  CHARX        IF NONE, SKIP AHEAD
       AI   R0,>2000     ELSE ADD COLOR TABLE OFFSET
       MOVB R9,R1        MOVE COLOR BYTE TO R1
CHCL   BLWP @VSBW        WRITE ONE BYTE
       INC  R0           POINT AT NEXT LOCATION
       DEC  R2           DEC COUNT IN R2
       JNE  CHCL         IF NOT ZERO, REPEAT
CHARX  RT                RETURN
KEY    BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            THEN TURN THEM OFF
       CB   @ANYKEY,@>837C KEY STRUCK?
       JNE  KEY          IF NOT, SCAN AGAIN
       RT                ELSE RETURN
SPRITE LI   R0,ATTLST    POINT AT ATTR. TABLE
       MOV  *R11+,R3     GET SPRITE #
       SLA  R3,2         MULT. BY 4
       A    R3,R0        ADD TO ADDRESS
       LI   R2,4         FOUR ATTR BYTES
SPLP   MOVB *R11+,R1     GET ATTRIBUTE
       BLWP @VSBW        WRITE TO TABLE
       INC  R0           NEXT SPOT
       DEC  R2           DEC COUNT
       JNE  SPLP         RPT IF NOT ZERO
       LI   R1,>D000     "DELETE" NEXT SPRITE
       BLWP @VSBW        WRITE THAT
       LI   R0,MOTBL     MOTION TABLE
```

```
       A    R3,R0         ADD OFFSET
       MOVB *R11+,R1      GET Y VELOCITY
       BLWP @VSBW         WRITE THAT
       INC  R0            NEXT SPOT
       MOVB *R11+,R1      GET X VELOCITY
       BLWP @VSBW         WRITE THAT
       CLR  R1            ZERO R1
       INC  R0            IST AUX BYTE
       BLWP @VSBW         WRITE A ZERO
       INC  R0            2ND AUX BYTE
       BLWP @VSBW         ZERO THERE TOO
       RT                 RETURN
USRINT BLWP @SPRINT       USE BLWP TO VECTOR
       RT                 THEN RETURN
SPRINT DATA INTWS,SPRVEL  WORKSPACE AND CODE
SPRVEL LI   R9,MOTBL      POINT AT MOTION TBL
       LI   R10,ATTLST    AND ATTR. TABLE
       MOVB @SPMOT,R12    HOW MANY IN MOTION?
       JEQ  INTEX         IF ZERO, EXIT
       SRL  R12,8         RIGHT-JUSTIFY
STR8   MOV  R9,R8         MOTION TABLE
       MOVB @INTWS+17,*R1 LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1        HIGH BYTE R8 TO VDP ADDR
       CLR  R4            CLEAR REG 4
       MOVB *R2,R4        GET Y VELOCITY FROM TABLE
       CLR  R6            CLEAR REG 6
       MOVB *R2,R6        GET X VELOCITY FROM TABLE
       SRA  R4,4          SHIFT RIGHT 4 BITS WITH SIGN
       MOVB *R2,R5        GET AUX DATA BYTE
       SRA  R5,4          SHIFT RIGHT WITH SIGN
       A    R4,R5         ADD R4 TO R5
       MOVB *R2,R7        GET 2ND AUX BYTE
       SRA  R6,4          SHIFT RIGHT 4 WITH SIGN
       SRA  R7,4          SAME FOR R7
       A    R6,R7         ADD R6 TO R7
       MOV  R10,R8        ATTRIBUTE ADDR TO R8
       MOVB @INTWS+17,*R1  LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1        HIGH BYTE R8 TO VDP
       CLR  R4            CLEAR REG 4
       MOVB *R2,R4        Y POSITION TO R4
       A    R5,R4         ADD R5 TO R4
       CI   R4,>C0FF      COMPARE TO LIMIT
       JLE  AC            JUMP IF LOW OR EQUAL
       CI   R4,>E000      COMPARE TO >E000
       JH   AC            JUMP IF HIGH
       MOV  R5,R5         MOVE R5 TO ITSELF
       JGT  AD            IF POSITIVE, JUMP
       AI   R4,>C000      ADD >C000 TO R4
AD     AI   R4,>2000      ADD >2000 TO R4
AC     CLR  R6            CLEAR REG 6
```

```
       MOVB *R2,R6       X POSITION TO R6
       A    R7,R6        ADD R7
       ORI  R8,>4000     SET >4000 BIT IN R8 (WRITING)
       MOVB @INTWS+17,*R1 LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1       HIGH BYTE R8 TO VDP ADDR
       MOVB R4,*R3       WRITE NEW Y POSITION
       MOVB R6,*R3       WRITE NEW X POSITION
       MOV  R9,R8        GET MOTION ADDR
       INCT R8           ADD TWO FOR AUX BYTE
       ORI  R8,>4000     SET FOR WRITE
       MOVB @INTWS+17,*R1 LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1       HIGH BYTE R8
       SWPB R5           SWAP R5
       SRL  R5,4         SHIFT R5 LEFT 4 BITS
       MOVB R5,*R3       WRITE NEW AUX BYTE #1
       SWPB R7           SWAP R7
       SRL  R7,4         SHIFT LEFT 4
       MOVB R7,*R3       WRITE NEW AUX BYTE #2
       C    *R9+,*R9+    ADD 4 TO R9
       C    *R10+,*R10+  ADD 4 TO R10
       DEC  R12          DEC COUNT IN R12
       JGT  STR8         IF NOT ZERO, DO ANOTHER SPRITE
INTEX  RTWP              RETURN WITH WORKSPACE POINTER
*
* DATA SECTION FOR INTERRUPT
*
SPMOT  BYTE 0            SPRITES IN MOTION
INTLOC DATA USRINT       USER INTERRUPT ADDRESS
INTWS  BSS  2            REG 0
       DATA >8C02,>8800,>8C00  PRELOADED R1, R2 AND R3
       BSS  24           REGS 4 THRU 15
*
*
* DATA SECTION
*
WS     BSS  >20          OUR WORKSPACE
M      DATA >8040,>2010,>0804,>0201  MASK DATA
BLOCK  DATA >FFFF,>FFFF,>FFFF,>FFFF SOLID BLOCK PATTERN
SPISTR BYTE 26
       TEXT 'Bitmap with Moving Sprites'
ANYKEY BYTE >20          COMPARISON BYTE FOR KEYSTROKE
       BSS  6            STORAGE FOR DSR DATA FROM VDP RAM
SAVCLR BSS  32           STORAGE FOR GRAPHICS COLOR TABLE
CHRTBL BSS  256*8        STORAGE FOR GRAPHICS CHARACTER DEFINITIONS
       END
```

1.62. The Art Of Assembly — Part 62. Book Wrong!

By Bruce Harrison

Today's article is about sprite operations, but we're going to start with a little anecdote from your author's undergraduate days at Penn State. How long ago was that? Here's a hint: Joe Paterno was an Assistant coach for the football team, under Head Coach Rip Engle! (Early 1960's) Our college roommate was a young man named Frank Ruhman. Frank was studying Mechanical Engineering, and always placed great faith in the accuracy of his textbooks. One time Frank's faith in textbooks was sorely tested when he got back a mid-term exam in which one of his solutions was marked wrong. He checked his solution carefully, and found that his method exactly matched what the textbook said, and that his math was all done correctly. He went to the instructor, a young man from India on some kind of teaching fellowship, and pointed out that his solution on the mid-term exactly matched the book. To that, the instructor gave a two-word reply: "Book wrong!"

1.62.1. How The Book Is WRONG

There have been a few times before in this column where we've said that about the TI Editor/Assembler manual, and here's another such case. The book says that if you place the value >D0 in the Y-position byte for any sprite, that will delete this sprite and all higher numbered sprites in use. To quote from Porgy and Bess, that "Ain't Necessarily So". If all your sprites are stationary, it's so. If, however, a higher numbered sprite is in motion, that sprite will remain visible and in motion after deleting the lower numbered one. If you really want all higher numbered ones to disappear, you'll also have to change the byte at >837A to a number equal to or lower than the sprite you've deleted. Let's for example say that you had three sprites in motion, numbered 0, 1, and 2. If you do the >D0 delete to number 1 without changing the value at >837A, you'll see that sprites 0 and 2 will stay on-screen and continue their motion.

If sprites are magnified, even the deleted one may still be visible in part at the top or bottom of the screen unless you've changed its color to transparent. For the benefit of our readers, we've put some source code in today's Sidebar that will do all things necessary to really eliminate that sprite and all that follow it in the number sequence. The routine is called DELSPR, and gets used via a BLWP operation.

1.62.2. Sprite Subroutines

Today's Sidebar is a whole series of subroutines for dealing with sprites. It includes one to "turn on" sprites, to delete them, and to change their patterns or motions, etc. We tried to include all of the CALLs from Extended Basic that work for sprites. These are all designed as BLWP vectors, so that, with just one exception they won't affect your own registers. That one exception is the POSIT routine, which affects the calling program's Register 1. On return from POSIT, your R1 will contain the Y and X positions of the sprite, in its left and right bytes, respectively. Here's the list of routines that you can BLWP to:

SPRITE	— puts a sprite on-screen
MOTION	— changes the y and x velocities
PATTRN	— changes the character pattern
LOCATE	— changes the y and x positions
POSIT	— reports the y and x positions in R1
COLOR	— changes the color of a sprite
DELSPR	— deletes a sprite and all following
MAGNIF	— magnifies all sprites
COINC	— determines coincidence
REVMO	— reverses motion
REV1	— reverses a single character left-right
REV4	— reverses four characters left-right

The last three of these are not included in the sprite services of Extended Basic. REVMO simply reverses the motion of a Sprite in both vertical and horizontal axes. If a motion byte is 0, that is unaffected by REVMO. REV1 reverses a single character definition from left to right. REV4 does that for a four-character sprite definition.

All of these are set up for the "normal" sprite operations in Graphics mode. Most of them will also work for the Bitmap cases, provided only that the ATTLST and MOTBL equates are set correctly for those table locations in your own program's code, but DELSPR will need to be modified after the line that says JNE DELLOP. The code from there to DELEX should be replaced with the following:

```
            MOV  @SPMOT,R1
            JEQ  DELEX
            C    R1,R4
            JLT  DELEX
            MOV  R4,@SPMOT
DELEX       RTWP
```

That will perform the same function for you as the code starting with MOVB @>837A,R1. This will make the deletion of sprites work just the way the book says, in spite of the book being wrong about how to delete sprites. Sprites should therefore be added in numerical order and deleted in inverse numerical order. To delete all sprites, just perform a BLWP @DELSPR with DATA 0.

Each of these BLWP routines requires at least one DATA line following it. In most cases, that data item must be the number of a sprite. Sprite numbers in Assembly run from 0 through 31. The exception is the case of MAGNIF. In that case the DATA following the BLWP instruction must be a number from 0 through 3, and this affects all sprites in use. The meaning of 0 through 3 is exactly like the meaning of the numbers 1 through 4 used in CALL MAGNIFY in Extended Basic. (The pictures in the XB manual are incorrect. Book Wrong there, too.)

0 means single character, normal size
1 means single character, double size
2 means four-character, normal size
3 means four-character, double size.

In the four-character cases, the sprite will consist of four character definitions, where the first one will be a character number that's evenly divisible by 4. That first pattern will form the upper left part of the sprite. the second will form the lower left quarter, the third the upper right quarter, and the fourth the lower right quarter. (This is exactly the same as in the Extended Basic four-character cases.) If the character value given in the setup of the sprite is not evenly divisible by four, then the upper left quarter will be the next lower character that is evenly divisible by 4. For example, if you've called for the sprite character to be A, B, or C, you'll get exactly the same result, with the @ symbol in the upper left quarter, the A in the lower left, B in the upper right, and C in the lower right. That's so because the @ symbol's ASCII code of 64 (>40) is the next lower number evenly divisible by 4 below the ASCII values 65, 66, and 67 for A, B, or C.

Two other exceptions to the use of a Sprite number as the DATA word are the character reversal routines REV1 and REV4. These each need one data words, that being the ASCII value of the character to be reversed. Thus to reverse the four character definition starting at character 128, you would do this:

```
BLWP @REV4
DATA 128
```

The subroutine will work just like the sprite itself, in that it will insure that the character number is divisible by four. If the DATA above is 129, 130, or 131, the subroutine will go to the address for character 128.

The subroutine has been set up on the assumption that the Sprite Pattern table has been set to coincide with the normal Character table for E/A operations. We do that in many programs by LI R0,>601 and then BLWP @VWTR. If your sprite pattern table is not set up that way, you'll need to change that AI R0,>800 just after label REV in the Sidebar.

In both cases, the Assembler will calculate the address in VDP RAM of the character pattern's address, and will put the appropriate hex value in the object file.

1.62.3. Using The Sidebar

Today's Sidebar has been set up so it can be used in several ways. You can simply insert this into your own program as is, taking care that no labels are duplicated. If you like, you can do it "module" fashion by assembling the Sidebar, then inserting REFs in your own code so the routines will be treated as external to your own program. After assembling your own code, you'd load your own object file, then load the object file created by assembling the Sidebar. That way, the linking loader will resolve all references for you, and the program in memory will have both your program and these subroutines. The third way to use this is to just put a copy directive into your own source file, as in COPY "DSK1.SIDEBAR62". Of course then you have to make sure that the Sidebar is in the disk drive called for in the copy directive when you're assembling your program.

Each of these routines in the Sidebar is preceded by a "commented out" section that explains how to invoke that routine from the main code. Except for POSIT, none of these will affect any of your own workspace registers. COINC will affect the status register, so that on return, you can use a JEQ for no coincidence or a JNE for coincidence. That would look something like this:

```
BLWP @COINC
DATA 1,3
JNE  C13
(code to handle non-coincidence case)
...
C13 (code to handle coincidence case)
...
```

That's about it. We hope this makes it easier to handle all those sprite operations. Next month's topic is undecided, but we hope you'll "tune in" for whatever it is.

```
* SIDEBAR 62
*
* HANDY SUBROUTINES FOR SPRITE
* OPERATIONS IN ASSEMBLY
* USING THE NORMAL GRAPHICS MODE
* CODE BY Bruce Harrison
* PUBLIC DOMAIN
* 14 MAY 1995
*
* FOLLOWING ARE THE DEFAULT EQUATES
* FOR GRAPHICS MODE - CHANGE THEM FOR
* USE WITH THE BITMAP MODES
*
ATTLST EQU  >300         SPRITE ATTRIBUTE TABLE
MOTBL  EQU  >780         SPRITE MOTION TABLE
*
* INSURE THE FOLLOWING REFS ARE IN YOUR PROGRAM
* UNLESS YOU USE THE "OBJECT MODULE" APPROACH
*
       REF  VSBW,VSBR,VWTR,VMBR
```

```
*
* THE FOLLOWING CAN BE USED AS REFS IN YOUR
* PROGRAM IF YOU USE THIS AS A SEPARATE MODULE
*
       DEF  SPRITE,MOTION,POSIT,LOCATE,PATTRN
       DEF  COLOR,MAGNIF,REVMO,COINC,DELSPR
       DEF  REV1,REV4
*
* SUBROUTINES FOR SPRITES
* EXCEPT FOR POSIT, NONE OF THESE
* WILL CHANGE ANYTHING IN THE CALLER'S
* WORKSPACE REGISTERS
*
* INVOKE SPRITE BY:
*      BLWP @SPRITE
*      DATA SPRITE #   (0 THRU 31)
*      BYTE YPOS,XPOS  Y POSITION, X POSITION
*      BYTE CHAR,COLOR CHARACTER VALUE, COLOR
*      BYTE YVEL,XVEL  Y VELOCITY, X VELOCITY
*
SPRITE DATA SPWS,SPRCOD
SPRCOD LI   R0,ATTLST    POINT AT ATTR. TABLE
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULT. BY 4
       A    R3,R0        ADD TO ADDRESS
       LI   R2,4         FOUR ATTR BYTES
SPLP   MOVB *R14+,R1     GET ATTRIBUTE
       BLWP @VSBW        WRITE TO TABLE
       INC  R0           NEXT SPOT
       DEC  R2           DEC COUNT
       JNE  SPLP         RPT IF NOT ZERO
       LI   R1,>D000     "DELETE" NEXT SPRITE
       BLWP @VSBW        WRITE THAT
       LI   R0,MOTBL     MOTION TABLE
       A    R3,R0        ADD OFFSET
       MOVB *R14+,R1     GET Y VELOCITY
       MOV  R1,R4        STASH IN R4
       BLWP @VSBW        WRITE THAT
       INC  R0           NEXT SPOT
       MOVB *R14+,R1     GET X VELOCITY
       MOV  R1,R5        STASH IN R5
       BLWP @VSBW        WRITE THAT
CLR1   CLR  R1           ZERO R1
       INC  R0           IST AUX BYTE
       BLWP @VSBW        WRITE A ZERO
       INC  R0           2ND AUX BYTE
       BLWP @VSBW        ZERO THERE TOO
       MOV  R4,R4        CHECK FOR R4=0
       JNE  SETMO        IF <>0, JUMP
       MOV  R5,R5        CHECK FOR R5=0
```

```
       JEQ  SPREX        IF 0, EXIT
SETMO  SRL  R3,2         DIVIDE R3 BY 4
       INC  R3           ADD ONE
       SWPB R3           SWAP R3
       CB   R3,@>837A    CHECK AGAINST SPRITES IN MOTION
       JLT  SPREX        IF LESS, EXIT
       MOVB R3,@>837A    ELSE SET THIS NUMBER IN MOTION
SPREX  RTWP              RETURN
*
* INVOKE MOTION BY:
*      BLWP @MOTION
*      DATA SPRITE #
*      BYTE YVEL,XVEL  Y VELOCITY,X VELOCITY
*

MOTION DATA SPWS,MOTCOD

MOTCOD LI   R0,MOTBL     POINT AT MOTION TABLE
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULTIPLY BY 4
       A    R3,R0        ADD OFFSET
       CLR  R4           CLEAR R4
       CLR  R5           AND R5
       MOV  *R14+,R1     GET DESIRED VELOCITIES
       MOVB R1,R4        Y VEL TO R4
       BLWP @VSBW        WRITE Y VEL
       SWPB R1           SWAP
       INC  R0           NEXT ADDR
       MOVB R1,R5        XVEL TO R5
       BLWP @VSBW        WRITE X VEL
       JMP  CLR1         JUMP TO END OF SPRITE
*
* INVOKE POSIT BY:
*      BLWP @POSIT
*      DATA SPRITE #
*
* ON RETURN, YOUR R1 WILL CONTAIN
* Y POSITION IN LEFT BYTE
* X POSITION IN RIGHT BYTE
*
POSIT  DATA SPWS,POSCOD

POSCOD LI   R0,ATTLST    ATTRIB TABLE
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULT. BY 4
       A    R3,R0        ADD TO R0
       BLWP @VSBR        READ Y POS
       SWPB R1           SWAP
       INC  R0           NEXT ADDR
       BLWP @VSBR        READ X POS
```

```
       SWPB R1           SWAP
       MOV  R1,@2(R13)   PUT R1 INTO CALLER'S R1
       RTWP              RETURN
*
* INVOKE LOCATE BY:
*      BLWP @LOCATE
*      DATA SPRITE #
*      BYTE YPOS,XPOS  Y POSITION, X POSITION
*
LOCATE DATA SPWS,LOCCOD

LOCCOD LI   R0,ATTLST    ATTRIB TABLE
       MOV  *R14+,R3     GET SPRITE NUM
       SLA  R3,2         MULT. BY 4
       A    R3,R0        ADD OFFSET
       MOV  *R14+,R1     GET POSITIONS
       BLWP @VSBW        WRITE Y POS
       SWPB R1           SWAP
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE X POS
       LI   R0,MOTBL+2   2 PAST MOTION TABLE
       A    R3,R0        ADD OFFSET
       CLR  R1           0 IN R1
       BLWP @VSBW        WRITE AUX BYTE 1
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE AUX BYTE 2
       RTWP              RETURN
*
* INVOKE PATTRN BY:
*      BLWP @PATTRN
*      DATA SPRITE #
*      DATA CHAR       CHARACTER ASCII
*
PATTRN DATA SPWS,PATCOD

PATCOD LI   R0,ATTLST+2  OFFSET TO CHAR
       MOV  *R14+,R3     GET SPRITE NUM
       SLA  R3,2         MULT BY 4
       A    R3,R0        ADD TO ADDR
       MOV  *R14+,R1     DESIRED CHAR
       SWPB R1           SWAP
       BLWP @VSBW        WRITE
       RTWP              RETURN
*
* INVOKE COLOR BY:
*      BLWP @COLOR
*      DATA SPRITE #     (0-31)
*      DATA COLOR        (0-15)
*
COLOR  DATA SPWS,COLCOD
```

```
COLCOD LI   R0,ATTLST+3  OFFSET TO COLOR
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULT. BY 4
       A    R3,R0        ADD TO ADDR
       MOV  *R14+,R1     GET DESIRED COLOR
       ANDI R1,>000F     INSURE RANGE 0->F
       SWPB R1           SWAP
       BLWP @VSBW        WRITE
       RTWP              RETURN
*
* INVOKE MAGNIF BY:
*      BLWP @MAGNIF
*      DATA MAGLEV     (0 THRU 3)
*
* WHERE:
* 0 MEANS UNMAGNIFIED
* 1 MEANS DOUBLE SIZE, SINGLE CHARACTER
* 2 MEANS NORMAL SIZE, FOUR CHARACTERS
* 3 MEANS DOUBLE SIZE, FOUR CHARACTERS
*
MAGNIF DATA SPWS,MAGCOD

MAGCOD MOV  *R14+,R0     DESIRED MAG
       ANDI R0,0003      INSURE 0-3
       ORI  R0,>01E0     ADD >1E0
       BLWP @VWTR        WRITE TO VDP REG 1
       SWPB R0           SWAP
       MOVB R0,@>83D4    PUT BYTE AT >83D4
       RTWP              RETURN
*
* REVMO REVERSES THE MOTION OF SPRITE
* BOTH HORIZONTAL AND VERTICAL
* MOTIONS WILL REVERSE
*
* INVOKE REVMO BY
*      BLWP @REVMO
*      DATA SPRITE #
*
REVMO  DATA SPWS,REVMC

REVMC  LI   R0,MOTBL     MOTION TABLE
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULT BY 4
       A    R3,R0        ADD OFFSET
       BLWP @VSBR        READ Y MOTION
       SRA  R1,8         RIGHT JUST
       NEG  R1           MUL BY -1
       SWPB R1           SWAP
       BLWP @VSBW        WRITE
       INC  R0           NEXT ADDR
```

```
       BLWP @VSBR        READ X MOTION
       SRA  R1,8         RIGHT JUST
       NEG  R1           MUL BY -1
       SWPB R1           SWAP
       BLWP @VSBW        WRITE
       INC  R0           NEXT ADDR
       CLR  R1           0 IN R1
       BLWP @VSBW        WRITE AUX BYTE 1
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE AUX BYTE 2
       RTWP              RETURN
*
* INVOKE COINC BY:
*      BLWP @COINC
*      DATA SPR1,SPR2  SPRITE NUMBERS
*
* ON RETURN, STATUS WILL BE EQ IF NO COINC
* OR NE IF COINC FOUND
*
COINC  DATA SPWS,COINCD

COINCD CLR  R8           CLEAR A REGISTER
       MOV  *R14+,R3     GET 1ST SPRITE #
       BL   @GP0         GET SPRITE POS
       MOV  R4,R6        Y POS TO R6
       MOV  R5,R7        X POS TO R7
       MOV  *R14+,R3     GET 2ND SPRITE #
       BL   @GP0         GET POSITION
       AI   R4,10        ADD TOLERANCE
       AI   R5,10        TO BOTH
       C    R6,R4        COMPARE Y POSITIONS
       JGT  COINX        IF GREATER, EXIT
TST5   C    R7,R5        COMPARE X POSITIONS
       JGT  COINX        IF GREATER, EXIT
TST42  AI   R4,-20       SUBTRACT 2X TOLERANCE
       AI   R5,-20       FROM BOTH
       C    R6,R4        COMPARE Y POSITIONS
       JLT  COINX        IF LESS, EXIT
       C    R7,R5        COMPARE X POSITIONS
       JLT  COINX        IF LESS, EXIT
       INC  R8           ELSE INC R8
COINX  MOV  R8,R8        CHECK R8 FOR ZERO
       STST R15          STATUS REG TO R15
       RTWP              THEN RETURN

GP0    SLA  R3,2         MULT. BY 4
       LI   R0,ATTLST    LOAD ATTLST ADDR
       A    R3,R0        ADD OFFSET
       BLWP @VSBR        READ Y POS.
       MOVB R1,R4        MOVE TO R4
```

```
       SRL  R4,8          RT. JUST.
       INC  R0            NEXT ADDR
       BLWP @VSBR         READ X POS.
       MOVB R1,R5         PUT IN R5
       SRL  R5,8          RT. JUST.
       RT                 RETURN
*
* INVOKE DELSPR BY:
*      BLWP @DELSPR
*      DATA SPRITE #
*
* NOTE:
*        DELSPR WILL DELETE THE
*        SPRITE SPECIFIED AND
*        ALL HIGHER NUMBERED ONES
*

DELSPR DATA SPWS,DELCOD

DELCOD LI   R0,ATTLST     ATTRIB TABLE
       MOV  *R14+,R3      GET SPRITE #
       MOV  R3,R4         STASH IN R4
       SLA  R3,2          MUL BY 4
       A    R3,R0         ADD OFFSET
       LI   R1,>D000      "DEL" CODE
       BLWP @VSBW         WRITE THAT
       CLR  R1            ZERO IN R1
       AI   R0,3          POINT AT COLOR BYTE
       BLWP @VSBW         TRANSPARENT COLOR
       LI   R0,MOTBL      MOTION TABLE
       A    R3,R0         ADD OFFSET
       LI   R5,4          FOUR BYTES
DELLOP BLWP @VSBW         WRITE ONE 0
       INC  R0            NEXT ADDR
       DEC  R5            DEC COUNT
       JNE  DELLOP        RPT IF NOT 0
       MOVB @>837A,R1     GET SPRITES IN MOTION
       JEQ  DELEX         EXIT IF 0
       SWPB R4            SWAP R4 (SPRITE #)
       CB   R1,R4         COMPARE
       JLT  DELEX         IF R1<R4, EXIT
       MOVB R4,@>837A     ELSE NEW NUM TO >837A
DELEX  RTWP               RETURN
*
* REV1 WILL REVERSE THE PATTERN OF A CHARACTER
* REV4 WILL REVERSE THE PATTERNS OF A 4-CHARACTER SET
*
* INVOKE REV1 OR REV4 BY:
*      BLWP @REV1      (OR REV4)
*      DATA CHAR       CHARACTER ASCII VALUE
```

```
*
REV1   DATA SPWS,REV1CD
REV4   DATA SPWS,REV4CD

REV1CD LI   R2,8         EIGHT BYTES
       MOV  *R14+,R0     GET CHARACTER NUMBER
       JMP  REV          THEN JUMP AHEAD
REV4CD LI   R2,32        32 BYTES
       MOV  *R14+,R0     CHARACTER ASCII
       ANDI R0,>00FC     INSURE DIVISIBLE BY 4
REV    SLA  R0,3         MULTIPLY BY 8
       AI   R0,>800      ADD OFFSET TO CHAR TBL
*****************
*
* NOTE: THE >800 ABOVE ASSUMES YOU'VE SET THE
* SPRITE PATTERN TABLE TO THAT VALUE.
* IF YOU'VE LEFT THE DEFAULT 0 FOR VDP REG 6,
* THEN OMIT THAT LINE.  IF YOU'VE PUT THE SPRITE
* PATTERN TABLE SOMEWHERE ELSE BY WRITING TO
* VDP REG 6, THEN PUT THE APPROPRIATE VALUE
* IN PLACE OF >800.
*
******************
       LI   R1,CPBUFF    POINT AT BUFFER
       BLWP @VMBR        READ THE BYTES
       CI   R2,32        COMPARE R2 TO 32
       JLT  MOV19        JUMP IF LESS
       AI   R0,16        POINT R0 AT THIRD CHAR DEF
MOV19  MOV  R1,R9        POINT R9 AT BUFFER
       MOV  R2,R5        COPY R2 VALUE INTO R5
LD4    LI   R4,8         EIGHT BITS PER BYTE
       CLR  R1           R1=0
       MOVB *R9+,R3      GET ONE BYTE FROM BUFFER AND INC R9
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            STOP INTERRUPTS
SHFT1  SRL  R1,1         SHIFT R1 TO RIGHT 1 BIT
       SLA  R3,1         SHIFT R3 TO LEFT ONE BIT
       JNC  DEC4         IF NO CARRY, JUMP AHEAD
       ORI  R1,>8000     ELSE SET MSB OF R1
DEC4   DEC  R4           DEC BIT COUNT
       JNE  SHFT1        IF NOT ZERO, REPEAT
       BLWP @VSBW        WRITE ONE BYTE
       INC  R0           POINT AHEAD ONE
       DEC  R5           DEC BYTE COUNT
       JEQ  DONE         IF ZERO, EXIT SUBROUTINE
       CI   R5,16        COMPARE R5 TO 16
       JNE  LD4          IF NOT EQUAL, JUMP BACK
       S    R2,R0        IF R5=16, SUBTRACT R2 FROM R0
       JMP  LD4          THEN JUMP BACK
DONE   RTWP              RETURN
```

```
*
* DATA SECTION
*
SPWS   BSS  32           SPRITE WORKSPACE
CPBUFF BSS  32           CHAR PATTERN BUFFER
*
* TO ASSEMBLE THIS AS AN OBJECT MODULE,
* DELETE THE "*" FROM THE LINE BELOW
*       END
```

1.63. The Art Of Assembly — Part 63. The C Connection

By Bruce Harrison

And now for something completely different. . . With apologies to all Monty Python fans, our topic for today is all new. We're going into the mysterious world of C programming, and making some things happen that C normally won't allow. First, our thanks to Clint Pulley, for designing C99, and to Vern Jensen, for inspiring us to create some new libraries for use by the C programmers in our midst. Thanks also to Tom Shorock. whose disk 'O Say Can You C' provided some early insight into our experimental C programming efforts, and to Charles Kirkwood, who also provided some examples of C programming. No, your Assembly author is not turning into a C programmer, but if one is going to make libraries of routines to be called from C, one must have at least a little knowledge of that language in order to test the routines.

This all started with a letter from Vern Jensen. He's doing C programs, and some of what he's trying to do is outside the normal realm of C. For example, he wanted to do some tricks with scrolling that simply can't be done outside of the Enhanced Graphics (a.k.a. Half-Bitmap) Mode. For those who don't C, the normal mode for C programs is TEXT mode, with white on dark blue as its color scheme. That works nicely for many purposes, but for games of any kind, one needs the Graphics modes. In Vern's case, we were sure from the outset that he'd need at least the Enhanced Graphics mode, and he'd also need some "tools" for operating in that mode, including Sprite capability.

1.63.1. Crossing The Bridge

As our readers will remember, we just recently crossed the bridge to Enhanced Graphics with subroutines for use in Assembly programs. Those get invoked by BLWP vectors, which is very handy indeed for Assembly work, but isn't at all useful with C. Thus we took our original HBSUB/S file, printed out Clint Pulley's Manual for C99, and set to work. In the originals, the parameters were passed as DATA/BYTE statements after the BLWP instruction, so our getting of parameters involved moving words or bytes from the location pointed to by R14. In the C case, R14 is a stack pointer, and our parameters get placed on C's stack before our routine is called. Thus we use R14 as a base to get our parameters. For example, if there's just one parameter, we MOV @2(R14),R0 to get our parameter. If there are more, we pull them from the stack with MOV @4(R14),R1 etc. That turned out to be easy to handle, provided we remember that the parameters are on the stack in inverse order, so the last one is at 2 bytes above R14, the next to last at 4 bytes above R14, and so on. In our original Assembly BLWP routines, we return to the main program by an RTWP instruction. In the case of C, we return to the C program by B *R13, since R13 is where C puts the return address when it branches to our code. In our own code, we can use Registers 0 through 7 with complete freedom, since C uses them only for temporary storage. If we have a value to pass back to a C variable, that goes into R8, and C places it in the variable for us upon return. Otherwise, we leave R8 through R15 alone! C has those registers "spoken for". Since most of our routines already were using the lower numbered registers, the limitation of R0 through R7 was no problem.

1.63.2. The C Side Of The Bridge

Each of our routines gets called from C by just a simple statement. Let's say, for example, that we wanted to put a string of characters on the Half-Bitmap screen using our routine HBSTR. We do it from C like this:

```
hbstr(row,col,"This goes on the screen at Row, Col");
```

Note that the lower case is used, and that there's a semicolon at the end of the statement. Row and Col may be either numbers or variables. The string is passed to us as an address, and is actually stored by C as an ASCIZ string. "What's THAT?", you ask. It means that the characters in the string's content start at the address passed us by C, and the end of the string is marked by a byte of 0. Thus our HBSTR routine has to check each character before putting it on-screen, and if it finds a 0, then the string is finished. There's one more trick up C's sleeve in these ASCIZ strings. If the quoted string contains a \n, C compiles that to character 10 (>A), and this indicates that we're to start a new line on the screen. In our routine, then, whenever a character 10 is found, we AI R0,32 and then ANDI R0,>FFE0, which puts our writing address at the start of the next Half-Bitmap screen row.

1.63.3. The Extra Goodies

In our usual fashion, we added a whole library of routines to put Sprites on the Half-Bitmap screen, complete with automatic motion, and fleshed out the "normal" stuff with a scroll screen routine, a clear screen routine, and so forth. There's a third library for taking inputs from the Half-Bitmap screen. The main Half-Bitmap library also includes routines to take you smoothly back to Text mode, or into the normal Graphics mode from Enhanced Graphics, so you can mode-switch gracefully within a C program. As time goes on, we're always adding new things to the libraries. For example, we added a "rainbow" capability to take maximum advantage of Enhanced Graphics mode. Thus you can colorize one or more characters with eight different color schemes, one for each row of the character. There's a simpler routine called HCOLOR, which uses only a single foreground and background color, but which can color a number of successive characters on just one call.

1.63.4. Different Conventions

In our normal Assembly work, everything is "zero-based". Color codes run from 0 through 15, rows from 0 through 23, columns from 0 through 31, and so on. In C parlance, colors run from 1 through 16, rows from 1 through 24, and columns from 1 through 32, etc. The routines that we designed for use with C follow the C conventions in these matters, with one exception. The numbers for Sprites run from 0 through 31. Thus, except for this case, the average C programmer should feel quite at home with our routines. Of course he or she will need to get used to dealing with the idea of having the Enhanced Graphics mode itself to deal with, plus the burden of dealing with sprites and such, all of which are alien to the run of the mill C program. As we did for Assembly programmers, we've given the C programmer working in Enhanced Graphics mode a complete set of character definitions for the sprites, separate from the character definitions for ordinary screen printing. There are routines for doing a "CALL CHAR" type operation for both the normal and sprite characters. These use the conventional hex character string just as you'd use in Basic, except that each call can have no more than 16 hex characters, thus one can define only one character per call.

1.63.5. The Choices Are Yours

There are three libraries available. They're called CHBSUB/O, CHBINP/O, and CHBSPR/O. The second and third ones cannot be used alone, but must be used along with CHBSUB/O. If your Enhanced Graphics work doesn't need sprites or inputs from the screen, then you can use just CHBSUB/O without the second or third ones. Please note that these subroutine libraries are designed for use only with C programs, and can't be used from ordinary Assembly programs. That's so because they depend entirely on the C method of parameter passing and the C method of executing a return to the main program. Fortunately for us, Clint Pulley made the whole process very simple by providing for stacking parameters and returning values to the C program.

1.63.6. For Whatever They're Worth

Here are our opinions about C99. Clint Pulley did a fantastic job of designing a "system" to implement this language on the TI-99/4A. Ordinarily, we'd consider this kind of thing impossible, but Clint did it and made a very easy and natural interface to the Assembly that's the machine's natural language. His one-pass compiler is easy, quick, and user-friendly. Our only gripe, and this is true of any compiled language, is that it takes a lot of memory space to do even a little program. We place the same gripe against our own Extended Basic Compiler, where small programs become quite large when compiled.

We have found two areas of incompatibility between Clint's C "system" and our own Enhanced TI. These involve our use of Horizon Ramdisks and Horizon's P-Gram card. The C compiler itself will not run when any of our three Horizon Ramdisks is turned on. It loads, but then lights up one of the Ramdisk activity lights and locks the computer up. We talked to Clint about this, and while he's aware of it, he's got no solution.

The second area of incompatibility is a bit more serious, as it involves both the Compiler and Option 5 programs generated from C source. Upon startup, the C derived Option 5 programs check for the presence of the E/A utilities in low memory. If those are already present, all is well. If the E/A utilities are not loaded, the C support tries to load them directly from GROM, and there's where the problem exhibits itself. In the "normal" case where an E/A module is present, the utilities are there in the >7000 block of GROM, and the C program finds and loads them perfectly well. If, however, one has a P-Gram setup like ours, Extended Basic is in the GROM space up through the >D000 block, and E/A is in the >E000 and >F000 blocks. Thus if we load up a C derived Option 5 program in our normal configuration, we get a system lockup as soon as it tries to load the utilities.

For our own purposes, we use a version of our own "sandwich" for converting the C object file(s) into Option 5, and since the "sandwich" loads the E/A utilities from memory using the SLAST memory space, everything works okay even with our P-Gram setup. Of course this latter problem can also be solved by simply putting our E/A cartridge in the GROM port, and then our P-Gram effectively removes itself from the system.

As we almost always do, we've made the disk containing these C utilities for operation in Enhanced Graphics (or Half-Bitmap) available as Public Domain software through the Lima Users' Group library. As always, the disk contains demos that show off most of the features, a set of instructions for using the libraries, and such. This disk will not be of any use to those who are not into C programming. Advance copies have gone to some of our friends who are programming in C. (You know who you are.)

Here's a list of the routines contained in these libraries, with a short description of what each does. We'll start with the CHBSUB/O file:

`sethb()`
: Sets computer to Half-Bitmap

`setcc(fc,bc)`
: Sets colors for all Half-Bitmap screen characters to fc foreground and bc background.

`setgm()`
: Sets computer to Graphics mode

`settm()`
: Sets back to Text Mode (C's default)

`hbstr(row,col,"Quoted Sring")`
: Displays the stuff between quotes at row, col. (Note that col must be 32 or less)

`hbchr(row,col,ascii)`
: Displays a single char at row, col

`hbcls()`
Clears the half-bit screen

`hbclf(row,col,rpts)`
Clears a part of the screen starting at row, col, and extending rpts locations horizontally

`hcolor(char,foregrnd,backgnd,repeats)`
Sets color scheme for one or more characters

`hrbow(char,repeats,fc,bc,fc,bc...)`
Sets a multi-color scheme for one or more characters

`calchr(char,"pattern string")`
Sets the definition of one character to the contents of the string. (like CALL CHAR)

`hbscrl()`
Scrolls the half-bit screen up one row

`hbvchr(row,col,char,repeats)`
Similar to Basic VCHAR

`hbhchr(row,col,char,repeats)`
Similar to Basic HCHAR

`hbdint(row,col,integer)`
Displays an integer numeric value or variable on the screen at row,col

Next, the contents of CHBINP/O:

`x=accnum(row,col,clrsig)`
Accepts an integer number from the screen at row, col. The clrsig indicates whether or not to clear the field before taking input.

`accstr(row,col,maxlen,clrsig,buffer)`
Accepts a string of maxlen at row, col, places that in buffer. As above, clrsig indicates clearing of the field, or not.

`k=hbgkf(row,col)`
Accepts a single keystroke with flashing cursor, then echoes the key to the screen at row, col.

`k=hbgk(row,col)`
Same as above, but no cursor.

`hbclf(row,col,rpts)`
Clears a part of the screen starting at row, col, and extending rpts characters.

Now the contents of CHBSPR/O, for sprites:

`sprite(#,ypos,xpos,char,color,yvel,xvel)`
Puts a sprite on the screen at ypos, xpos, with Char character, Color color, and velocities Yvel, Xvel.

`motion(#,yvel,xvel)`
Changes motion of sprite # to yvel,xvel

`y=yposit(#)`
Sets y to the sprite #'s y position

`x=xposit(#)`
Sets x to the sprite #'s x position

`locspr(#,ypos,xpos)`
Sets location of sprite # to ypos, xpos

`pattrn(#,char)`
Sets character value sprite # to char

`color(#,color)`
Sets color of Sprite # to color

`magnif(maglev)`
Magnifies all sprites to maglev (1-4)

`revmo(#)`
Reverses motion of sprite #

`k=coinc(a,b)`
k will be one if sprites a and b are in coincidence, 0 if they're not

`delspr(#)`
Deletes sprite # and all higher numbers

`rev1(char)`
Reverses a sprite character pattern

`rev4(char)`
Reverses a 4-character sprite pattern

`sprchr(char,"pattern string")`
Defines pattern for a sprite character to contents of string (similar to CALL CHAR)

That's quite a lot, isn't it. Where we've used # in the sprite cases, that would be a number from 0-31, corresponding to the full range of sprites available from Assembly code. The levels for magnif are 1-4 to correspond with XB numbering, where 1 is normal size, 2 is double size, and 3 and 4 are 4-character sprites. Positions are in dot-row and dot-column coordinates for sprites. For other things, Row and Col numbers are in the graphics mode ranges 1-24 and 1-32, respectively. (These correspond to the HCHAR, VCHAR, GCHAR columns in Basic/XB parlance.)

C99 programmers should note that all variables with the exception of the string variables used in ACCSTR and HBSTR must be of the INT type. The string variables used with ACCSTR and HBSTR must be a CHAR type array. HBSTR will actually work with either a CHAR array or a quoted string, as C99 passes us the starting address in either case. ACCSTR puts the string into the named array in ASCIZ format, as required by C99.

There's no Sidebar this month, as the source code for these libraries would take up half an issue by themselves. We invite others besides Vern Jensen to ask for help with things related to interfacing between C and Assembly. Perhaps we'll turn out a special routine just for you! Next month we'll try to surprise you again with an off-the-wall topic like this.

```
* SIDEBAR 63
*
* HANDY SUBROUTINES
* FOR SPRITE OPERATIONS
* WITH BITMAP MODES
* CODE BY Bruce Harrison
* PUBLIC DOMAIN
* 13 APR 1995
*
* EQUATES FOR FULL BITMAP
*
ATTLST EQU  >1B80
MOTBL  EQU  >1B00
*
* FOR HALF-BITMAP MODE, CHANGE
* THE TWO EQUATES ABOVE TO
*ATTLST EQU  >1000
*MOTBL  EQU  >1080
*
* REQUIRED UTILITY REFERENCES
*
       REF  VSBW,VSBR,VWTR,VMBR
*
* DEFINED LABELS FOR EXTERNAL
* REFERENCE IN YOUR MAIN PROGRAM
*
       DEF  SPRITE,MOTION,POSIT,LOCATE,PATTRN
       DEF  COLOR,MAGNIF,REVMO,COINC,DELSPR
       DEF  REV1,REV4,CPBUFF,SPMOT,INTLOC
*
```

```
* SUBROUTINES FOR SPRITES
* EXCEPT FOR POSIT, NONE OF THESE
* WILL CHANGE ANYTHING IN THE CALLER'S
* WORKSPACE REGISTERS
*
* INVOKE SPRITE BY:
*      BLWP @SPRITE
*      DATA SPRITE #   (0 THRU 31)
*      BYTE YPOS,XPOS  Y POSITION, X POSITION
*      BYTE CHAR,COLOR CHARACTER VALUE, COLOR
*      BYTE YVEL,XVEL  Y VELOCITY, X VELOCITY
*
SPRITE DATA SPWS,SPRCOD
SPRCOD LI   R0,ATTLST    POINT AT ATTR. TABLE
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULT. BY 4
       A    R3,R0        ADD TO ADDRESS
       LI   R2,4         FOUR ATTR BYTES
SPLP   MOVB *R14+,R1     GET ATTRIBUTE
       BLWP @VSBW        WRITE TO TABLE
       INC  R0           NEXT SPOT
       DEC  R2           DEC COUNT
       JNE  SPLP         RPT IF NOT ZERO
       LI   R1,>D000     "DELETE" NEXT SPRITE
       BLWP @VSBW        WRITE THAT
       LI   R0,MOTBL     MOTION TABLE
       A    R3,R0        ADD OFFSET
       MOVB *R14+,R1     GET Y VELOCITY
       BLWP @VSBW        WRITE THAT
       INC  R0           NEXT SPOT
       MOVB *R14+,R1     GET X VELOCITY
       BLWP @VSBW        WRITE THAT
       CLR  R1           ZERO R1
       INC  R0           IST AUX BYTE
       BLWP @VSBW        WRITE A ZERO
       INC  R0           2ND AUX BYTE
       BLWP @VSBW        ZERO THERE TOO
       RTWP              RETURN
*
* INVOKE MOTION BY:
*      BLWP @MOTION
*      DATA SPRITE #
*      BYTE YVEL,XVEL  Y VELOCITY,X VELOCITY
*

MOTION DATA SPWS,MOTCOD

MOTCOD LI   R0,MOTBL     POINT AT MOTION TABLE
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULTIPLY BY 4
```

```
       A    R3,R0        ADD OFFSET
       MOV  *R14+,R1     GET DESIRED VELOCITIES
       BLWP @VSBW        WRITE Y VEL
       SWPB R1           SWAP
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE X VEL
       INC  R0           NEXT ADR
       CLR  R1           0 IN R1
       BLWP @VSBW        WRITE AUX BYTE 1
       INC  R0           NEXT ADR
       BLWP @VSBW        WRITE AUX BYTE 2
       RTWP              RETURN
*
* INVOKE POSIT BY:
*      BLWP @POSIT
*      DATA SPRITE #
*
* ON RETURN, YOUR R1 WILL CONTAIN
* Y POSITION IN LEFT BYTE
* X POSITION IN RIGHT BYTE
*
POSIT  DATA SPWS,POSCOD

POSCOD LI   R0,ATTLST    ATTRIB TABLE
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULT. BY 4
       A    R3,R0        ADD TO R0
       BLWP @VSBR        READ Y POS
       SWPB R1           SWAP
       INC  R0           NEXT ADDR
       BLWP @VSBR        READ X POS
       SWPB R1           SWAP
       MOV  R1,@2(R13)   PUT R1 INTO CALLER'S R1
       RTWP              RETURN
*
* INVOKE LOCATE BY:
*      BLWP @LOCATE
*      DATA SPRITE #
*      BYTE YPOS,XPOS  Y POSITION, X POSITION
*
LOCATE DATA SPWS,LOCCOD

LOCCOD LI   R0,ATTLST    ATTRIB TABLE
       MOV  *R14+,R3     GET SPRITE NUM
       SLA  R3,2         MULT. BY 4
       A    R3,R0        ADD OFFSET
       MOV  *R14+,R1     GET POSITIONS
       BLWP @VSBW        WRITE Y POS
       SWPB R1           SWAP
       INC  R0           NEXT ADDR
```

```
       BLWP @VSBW        WRITE X POS
       LI   R0,MOTBL+2   2 PAST MOTION TABLE
       A    R3,R0        ADD OFFSET
       CLR  R1           0 IN R1
       BLWP @VSBW        WRITE AUX BYTE 1
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE AUX BYTE 2
       RTWP              RETURN
*
* INVOKE PATTRN BY:
*      BLWP @PATTRN
*      DATA SPRITE #
*      DATA CHAR       CHARACTER ASCII
*
PATTRN DATA SPWS,PATCOD

PATCOD LI   R0,ATTLST+2  OFFSET TO CHAR
       MOV  *R14+,R3     GET SPRITE NUM
       SLA  R3,2         MULT BY 4
       A    R3,R0        ADD TO ADDR
       MOV  *R14+,R1     DESIRED CHAR
       SWPB R1           SWAP
       BLWP @VSBW        WRITE
       RTWP              RETURN
COLOR  DATA SPWS,COLCOD

COLCOD LI   R0,ATTLST+3  OFFSET TO COLOR
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULT. BY 4
       A    R3,R0        ADD TO ADDR
       MOV  *R14+,R1     GET DESIRED COLOR
       ANDI R1,>000F     INSURE RANGE 0->F
       SWPB R1           SWAP
       BLWP @VSBW        WRITE
       RTWP              RETURN
*
* INVOKE MAGNIF BY:
*      BLWP @MAGNIF
*      DATA MAGLEV     (0 THRU 3)
*
* WHERE:
* 0 MEANS UNMAGNIFIED
* 1 MEANS DOUBLE SIZE, SINGLE CHARACTER
* 2 MEANS NORMAL SIZE, FOUR CHARACTERS
* 3 MEANS DOUBLE SIZE, FOUR CHARACTERS
*
MAGNIF DATA SPWS,MAGCOD

MAGCOD MOV  *R14+,R0     DESIRED MAG
       ANDI R0,0003      INSURE 0-3
```

```
       ORI  R0,>01E0     ADD >1E0
       BLWP @VWTR        WRITE TO VDP REG 1
       SWPB R0           SWAP
       MOVB R0,@>83D4    PUT BYTE AT >83D4
       RTWP              RETURN
*
* REVMO REVERSES THE MOTION OF SPRITE
* BOTH HORIZONTAL AND VERTICAL
* MOTIONS WILL REVERSE
*
* INVOKE REVMO BY
*      BLWP @REVMO
*      DATA SPRITE #
*
REVMO  DATA SPWS,REVMC

REVMC  LI   R0,MOTBL     MOTION TABLE
       MOV  *R14+,R3     GET SPRITE #
       SLA  R3,2         MULT BY 4
       A    R3,R0        ADD OFFSET
       BLWP @VSBR        READ Y MOTION
       SRA  R1,8         RIGHT JUST
       NEG  R1           MUL BY -1
       SWPB R1           SWAP
       BLWP @VSBW        WRITE
       INC  R0           NEXT ADDR
       BLWP @VSBR        READ X MOTION
       SRA  R1,8         RIGHT JUST
       NEG  R1           MUL BY -1
       SWPB R1           SWAP
       BLWP @VSBW        WRITE
       INC  R0           NEXT ADDR
       CLR  R1           0 IN R1
       BLWP @VSBW        WRITE AUX BYTE 1
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE AUX BYTE 2
       RTWP              RETURN
*
* INVOKE COINC BY:
*      BLWP @COINC
*      DATA SPR1,SPR2  SPRITE NUMBERS
*
* ON RETURN, STATUS WILL BE EQ IF NO COINC
* OR NE IF COINC FOUND
*
COINC  DATA SPWS,COINCD

COINCD CLR  R8           CLEAR A REGISTER
       MOV  *R14+,R3     GET 1ST SPRITE #
       BL   @GP0         GET SPRITE POS
```

```
       MOV  R4,R6        Y POS TO R6
       MOV  R5,R7        X POS TO R7
       MOV  *R14+,R3     GET 2ND SPRITE #
       BL   @GP0         GET POSITION
       AI   R4,10        ADD TOLERANCE
       AI   R5,10        TO BOTH
       C    R6,R4        COMPARE Y POSITIONS
       JGT  COINX        IF GREATER, EXIT
TST5   C    R7,R5        COMPARE X POSITIONS
       JGT  COINX        IF GREATER, EXIT
TST42  AI   R4,-20       SUBTRACT TWICE TOLERANCE
       AI   R5,-20       FROM BOTH
       C    R6,R4        COMPARE Y POSITIONS
       JLT  COINX        IF LESS, EXIT
       C    R7,R5        COMPARE X POSITIONS
       JLT  COINX        IF LESS, EXIT
       INC  R8           ELSE INC R8
COINX  MOV  R8,R8        CHECK R8 FOR ZERO
       STST R15          STATUS REG TO R15
       RTWP              THEN RETURN

GP0    SLA  R3,2         MULT. BY 4
       LI   R0,ATTLST    LOAD ATTLST ADDR
       A    R3,R0        ADD OFFSET
       BLWP @VSBR        READ Y POS.
       MOVB R1,R4        MOVE TO R4
       SRL  R4,8         RT. JUST.
       INC  R0           NEXT ADDR
       BLWP @VSBR        READ X POS.
       MOVB R1,R5        PUT IN R5
       SRL  R5,8         RT. JUST.
       RT                RETURN
*
* INVOKE DELSPR BY:
*      BLWP @DELSPR
*      DATA SPRITE #
*
* NOTE:
*        DELSPR WILL DELETE THE
*        SPRITE SPECIFIED AND
*        ALL HIGHER NUMBERED ONES
*

DELSPR DATA SPWS,DELCOD

DELCOD LI   R0,ATTLST    ATTRIB TABLE
       MOV  *R14+,R3     GET SPRITE #
       MOV  R3,R4        STASH IN R4
       SLA  R3,2         MUL BY 4
       A    R3,R0        ADD OFFSET
```

```
       LI   R1,>D000     "DEL" CODE
       BLWP @VSBW        WRITE THAT
       CLR  R1           ZERO IN R1
       AI   R0,3         POINT AT COLOR BYTE
       BLWP @VSBW        TRANSPARENT COLOR
       LI   R0,MOTBL     MOTION TABLE
       A    R3,R0        ADD OFFSET
       LI   R5,4         FOUR BYTES
DELLOP BLWP @VSBW        WRITE ONE 0
       INC  R0           NEXT ADDR
       DEC  R5           DEC COUNT
       JNE  DELLOP       RPT IF NOT 0
       MOV  @SPMOT,R1    GET SPRITES IN MOTION
       JEQ  DELEX        EXIT IF 0
       SWPB R4           SWAP R4 (SPRITE #)
       CB   R1,R4        COMPARE
       JLT  DELEX        IF R1<R4, EXIT
       MOV  R4,@SPMOT    ELSE NEW NUM TO >837A
DELEX  RTWP              RETURN
*
* REV1 WILL REVERSE THE PATTERN OF A CHARACTER
* REV4 WILL REVERSE THE PATTERNS OF A 4-CHARACTER SET
*
* INVOKE REV1 OR REV4 BY:
*      BLWP @REV1 (OR REV4)
*      DATA VDPADR     ADDRESS OF CHAR PATTERN IN VDP RAM
*      DATA CPBUFF     BUFFER ADDRESS (32 BYTES IN MEMORY)
*
REV1   DATA SPWS,REV1CD
REV4   DATA SPWS,REV4CD

REV1CD LI   R2,8         EIGHT BYTES
       JMP  REV          THEN JUMP AHEAD
REV4CD LI   R2,32        32 BYTES
REV    MOV  *R14+,R0     POINT AT CHARACTER DEF
       MOV  *R14+,R1     POINT AT BUFFER
       BLWP @VMBR        READ THE BYTES
       CI   R2,32        COMPARE R2 TO 32
       JLT  MOV19        JUMP IF LESS
       AI   R0,16        POINT R0 AT THIRD CHAR DEF
MOV19  MOV  R1,R9        POINT R9 AT BUFFER
       MOV  R2,R5        COPY R2 VALUE INTO R5
LD4    LI   R4,8         EIGHT BITS PER BYTE
       CLR  R1           R1=0
       MOVB *R9+,R3      MOVE ONE BYTE TO R3 AND INC R9
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            STOP INTERRUPTS
SHFT1  SRL  R1,1         SHIFT R1 TO RIGHT 1 BIT
       SLA  R3,1         SHIFT R3 TO LEFT ONE BIT
       JNC  DEC4         IF NO CARRY, JUMP AHEAD
```

```
        ORI  R1,>8000     ELSE SET MSB OF R1
DEC4    DEC  R4           DEC BIT COUNT
        JNE  SHFT1        IF NOT ZERO, REPEAT
        BLWP @VSBW        WRITE ONE BYTE
        INC  R0           POINT AHEAD ONE
        DEC  R5           DEC BYTE COUNT
        JEQ  DONE         IF ZERO, EXIT SUBROUTINE
        CI   R5,16        COMPARE R5 TO 16
        JNE  LD4          IF NOT EQUAL, JUMP BACK
        S    R2,R0        IF R5=16, SUBTRACT R2 FROM R0
        JMP  LD4          THEN JUMP BACK
DONE    RTWP              RETURN
USRINT BLWP @SPRINT       USE BLWP TO VECTOR
        RT                THEN RETURN
SPRINT DATA INTWS,SPRVEL WORKSPACE AND CODE
SPRVEL LI   R9,MOTBL      POINT AT MOTION TBL
        LI   R10,ATTLST   AND ATTR. TABLE
        MOV  @SPMOT,R12   HOW MANY IN MOTION?
        JEQ  INTEX        IF ZERO, EXIT
STR8    MOV  R9,R8        MOTION TABLE
        MOVB @INTWS+17,*R1 LOW BYTE R8 TO VDP ADDR
        MOVB R8,*R1       HIGH BYTE R8 TO VDP ADDR
        CLR  R4           CLEAR REG 4
        MOVB *R2,R4       GET Y VELOCITY FROM TABLE
        CLR  R6           CLEAR REG 6
        MOVB *R2,R6       GET X VELOCITY FROM TABLE
        SRA  R4,4         SHIFT RIGHT 4 BITS WITH SIGN
        MOVB *R2,R5       GET AUX DATA BYTE
        SRA  R5,4         SHIFT RIGHT WITH SIGN
        A    R4,R5        ADD R4 TO R5
        MOVB *R2,R7       GET 2ND AUX BYTE
        SRA  R6,4         SHIFT RIGHT 4 WITH SIGN
        SRA  R7,4         SAME FOR R7
        A    R6,R7        ADD R6 TO R7
        MOV  R10,R8       ATTRIBUTE ADDR TO R8
        MOVB @INTWS+17,*R1  LOW BYTE R8 TO VDP ADDR
        MOVB R8,*R1       HIGH BYTE R8 TO VDP
        CLR  R4           CLEAR REG 4
        MOVB *R2,R4       Y POSITION TO R4
        A    R5,R4        ADD R5 TO R4
        CI   R4,>C0FF     COMPARE TO LIMIT
        JLE  AC           JUMP IF LOW OR EQUAL
        CI   R4,>E000     COMPARE TO >E000
        JH   AC           JUMP IF HIGH
        MOV  R5,R5        MOVE R5 TO ITSELF
        JGT  AD           IF POSITIVE, JUMP
        AI   R4,>C000     ADD >C000 TO R4
AD      AI   R4,>2000     ADD >2000 TO R4
AC      CLR  R6           CLEAR REG 6
        MOVB *R2,R6       X POSITION TO R6
```

```
       A    R7,R6        ADD R7
       ORI  R8,>4000     SET >4000 BIT IN R8 (WRITING)
       MOVB @INTWS+17,*R1 LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1       HIGH BYTE R8 TO VDP ADDR
       MOVB R4,*R3       WRITE NEW Y POSITION
       MOVB R6,*R3       WRITE NEW X POSITION
       MOV  R9,R8        GET MOTION ADDR
       INCT R8           ADD TWO FOR AUX BYTE
       ORI  R8,>4000     SET FOR WRITE
       MOVB @INTWS+17,*R1 LOW BYTE R8 TO VDP ADDR
       MOVB R8,*R1       HIGH BYTE R8
       SWPB R5           SWAP R5
       SRL  R5,4         SHIFT R5 LEFT 4 BITS
       MOVB R5,*R3       WRITE NEW AUX BYTE #1
       SWPB R7           SWAP R7
       SRL  R7,4         SHIFT LEFT 4
       MOVB R7,*R3       WRITE NEW AUX BYTE #2
       C    *R9+,*R9+    ADD 4 TO R9
       C    *R10+,*R10+  ADD 4 TO R10
       DEC  R12          DEC COUNT IN R12
       JGT  STR8         IF NOT ZERO, DO ANOTHER SPRITE
INTEX  RTWP              RETURN WITH WORKSPACE POINTER
*
* DATA SECTION FOR INTERRUPT
*
INTLOC DATA USRINT       USER INTERRUPT ADDRESS
INTWS  BSS  2            REG 0
       DATA >8C02,>8800,>8C00  PRELOADED R1, R2 AND R3
       BSS  24           REGS 4 THRU 15
SPMOT  DATA 0
SPWS   BSS  32           SPRITE WORKSPACE
CPBUFF BSS  32           CHAR PATTERN BUFFER
*
* TO ASSEMBLE AS AN OBJECT MODULE,
* DELETE THE "*" FROM THE LINE BELOW
*       END
```

1.64. The Art Of Assembly — Part 64. Thank You Notes

By Bruce Harrison

Once again we're passing out thanks. Today's first person for thanks is Tony McGovern. Once again Tony's talents have made our lives easier than we have a right to expect. The main reason for our thanks this time is of course the Funnelweb system of programs. This story starts with last month's problem about using the C99 compiler while our Ramdisks were turned on. It turned out that there's an exception. In our setup, there are three Horizon Ramdisk cards. The first one is at CRU address >1000, and that has only DSK3. The second is at >1200, and has drives DSK4, 5, and 8. The third one is at >1400, and has the drives DSK6 and DSK7 on it. Through a series of carefully controlled experiments, (accidentally left 6 and 7 card turned on) we found that the C99 compiler would work with that Ramdisk turned on. Apparently the problem with C99 is related to CRU addresses. What was important, though, was that we keep Funnelweb on Drive 7 at all times. Thus we could gain access to editing capability without having to have E/A's EDIT1 on a disk in Drive 1, and could also use Drive 6 for our C99 source files and such. Funnelweb had once more saved our bacon, but that's not all.

1.64.1. What Is This Script Stuff?

There are a lot of features packed into Funnelweb, many of which we've never tried. Mainly that's because we're too busy creating new stuff to properly examine the old stuff. We'd used some of Tony's loaders to check out our programs, but there was one on that list called Script Loader that was a mystery. In our recent work, we've been using pre-assembled object modules with little "demo" programs, both from C99 and pure Assembly. This means that for each test run we've got several object files to load before we can run anything. On countless occasions, we've got partway through the loading and couldn't remember whether the one just loaded was the demo program or the support files. E/A has that nasty habit of blanking the input field after a file loads, so if you're distracted by something, it's easy to lose track, and then you'll surely get some kind of stupid error report from E/A. When we get lost like that, we just **FCTN 9** back to the Main Menu of E/A and start over.

Tony's option 4 loader leaves the previous file name on the screen (scrolled up some) so you can much more easily track which files have already been loaded. But there was still that option 5 on Tony's Loader menu, called SCRIPT LOADER. Finally, we decided to try that out. After some fumbling about, we found that our old floppy copy of Funnelweb had a sample file on it called SCRIPT. By using that as a starting point, we were able to put together a couple of script files for our projects. Boy did that help! By having a couple of those on Drive 6, right along with the object files, we could get into Funnelweb, then use that script loader option, and very quickly load just the right files for our test. Tony even thoughtfully provided for an auto start capability so that we can enter at any defined label after loading all necessary files. WOW! Never again will we forget what we're doing part way through a load sequence, at least not when we can quickly make up a little Script file. Tony even designed his sample so that sufficient instructions for using it are built right in!

1.64.2. Art's Artful Assembler

We've said thanks to Art Green before for his wonderful RAGASM assembler. That was for the fact that his error reporting approach is so much better than the TI Assembler gives. Now we're thanking Art again for the same product, but this time for how smart the Assembler itself is. We keep that assembler handy at all times on Ramdisk, including a copy on our Drive 7. The copies of RAGASM that we use have all been through an install process so that the "R" option is there by default, so we don't have to answer the OPTIONS prompt at all. That in itself is a neat feature! You can install RAGASM tailored to your particular needs.

Recently, as our readers know, we've been conducting some experiments using Clint Pulley's C99 compiler. That program produces Assembly source code that's without the "R" for the registers. Again by carefully controlled experiment (namely just forgetting about it) we discovered that Art Green's RAGASM can handle non-"R" files just the same as "R" files, even when the "R" option is set. In at least one case, we embedded some original assembly code of our own within the C99 source, so the compiled file contained a mixture of "R" and non-"R" register references. RAGASM handled all that without a whimper! The TI Assembler can work this way too, but having this feature available in our favorite version of an Assembler is great. Thank you, Art!

1.64.3. Still More Thanks

To Clint Pulley, for making a truly marvelous C Compiler on our little machine. In the process of trying to make little test programs for our C99 support libraries, we got a number of books on C from the local library. Reading those provided a whole load of ideas that we wanted to try out, but we couldn't be sure whether these slightly advanced tricks would work, so we just tried them out. They WORKED!

For example, in the C version of a FOR-NEXT loop, one can use more than one argument in the FOR part, thus setting initial values to other variables besides the one controlling the loop. In fact, the TO part can even use a variable that's not part of the FOR. Though the TO part can only have one test variable, the STEP part can include more than one variable as well. That looked like something that would be handy in one of our test programs, so we tried it this way:

```
for(row=8,col=12,chr='0';chr<':';++col,++chr)
hbchr(row,col,chr);
```

THAT WORKED! Notice that there's no NEXT required. The semicolon at the end of the second line marks the end of the loop, which keeps repeating so long as the variable chr is less than ':' (58). Notice also, as is the case in Assembly, that we can load chr with the number 48 by putting a zero between single quote marks, and could make the limit 58 in a similar fashion using a colon between single quotes. In the last part of the FOR statement, (STEP) we have made both col and chr increment by placing both variables there preceded by a double plus sign. In case you're curious about such things, this loop puts the numbers from 0 through 9 on the screen at row 12, starting at column 8. The same thing could be done in Extended Basic like this:

```
ROW=8:: COL=12:: FOR CHR=48 TO 57 :: CALL HCHAR(ROW,COL,CHR)
:: COL=COL+1 :: NEXT CHR
```

Clint has built lots of nice features into his C99, including the ability to directly embed Assembly source, which is a great feature for us "Assembly guys". That's a feature that the PC versions of C lack. And so we thank Clint Pulley for incorporating such nifty tricks into C99.

1.64.4. For My Next Trick

No, I won't be leaping higher than Michael Jordan. In today's Sidebar, however, is the source for an object module that takes string, numeric, and single-key inputs from the screen while in Half-Bitmap mode. This has been designed so that the string and numeric input parts roughly equal an ACCEPT AT as used in Extended Basic. Thus the function keys have the same actions as you'd expect. **FCTN 1** deletes the character under the cursor, moving the rest of the field to the left by one, and putting a space at the last character. **FCTN 2** puts you into Insert mode, so that each key typed moves all text following it rightward by one character, then appears. **FCTN 3** clears the input field. **FCTN S** moves the cursor left, and cancels insert. **FCTN D** moves the cursor to the right and cancels insert. Both **FCTN S** and **FCTN D** have a pause-then-repeat action, so you can quickly move many places. **ENTER** or **FCTN E**, **FCTN X**, **FCTN 8** or **FCTN 9** will exit the routine.

There are two numeric input routines, which share much of the same source code, but take numbers instead of strings. One of these takes an integer in the range -32768 through 32767. The other takes a floating point number, which can span the whole range of TI floating point numbers. The integer case gives you an input field of only six spaces, so it has room for -32768. The floating point input gives a 15 space field, which should be large enough. In either case, the result is left for you at >834A, the floating point accumulator. If it's an integer, it will be just the word at that location. If it's floating point, it will be eight bytes starting at >834A. In both cases, scientific notation is accepted, and in both cases leaving the field blank will report 0. In the integer case, if you enter something with a decimal, that will be rounded to the nearest integer. (e.g. 4.4 will become 4, 4.5 will become 5) No, that wasn't our magic, but TI's, in the Convert Floating Point to Integer routine that we use.

For string inputs, you specify the number of spaces in the field, up to a maximum of 80. Input in all cases is taken at the row and column you specify, where row runs from 1 through 24 and column from 1 through 32. All three routines check the validity of row and column, and the validity of the endpoint after the field length is added. If either the start or end point is out of range, the routine will exit back to your code without doing anything. In other words, be careful what you specify for row, column, and max length.

In addition to these three multi-character input routines, there are two single-key inputs, one called HBGK, and the other HBGKF. Each requires that row and column get specified. HBGK just awaits a keystroke, then echoes that key at row, col. HBGKF is a bit fancier, in that it puts a flashing cursor at row, column, and can accept a default response if you've put a character on the screen at row, col. Let's say for example that you've put a Y at 12,16, and then used HBGKF at that spot. Pressing **ENTER** will accept the Y as the input. Pressing **Y** will do the same, but any other key will be accepted "as is" and echoed to the screen at 12, 16. The key pressed, or the default response if **ENTER** was pressed, will be placed in R1 of your workspace as a word. The makes it easier for you to check what the response was. For example you could CI R1,'N' to see whether **N** was pressed.

1.64.5. The Public Domain Disk

As usual with things like this, we've put together a Public Domain disk with the stuff in today's Sidebar, plus a collection of other routines for dealing with Half-Bitmap mode. These include capabilities for using sprites, for displaying strings or numbers on the Half-Bitmap screen, and so forth. The disk is called ADVHB, and is available through the Lima Users Group. All the source code is included, with full annotation, so you can see how it works. Instructions are also provided, and several "demo" programs that let you see these routines at work without too much effort.

1.64.6. The Little Details

You may notice something peculiar in the Sidebar, in that we've included instructions that save the User Interrupt from >83C4 before our cursor starts and restored it afterward. This was done so that these routines would be compatible with the routines that provide sprites on the Half-Bitmap screen. If there are sprites in motion when you enter one of these flashing cursor input routines, they will simply stop in their tracks while input is accepted, but will resume their motion when you exit the input routine. We thought this would be more desirable than just making you re-set all the sprites after the input routines. The simple key input routine HBGK does not do this, so your sprites will remain in motion while that routine is awaiting a keystroke.

Since all the source code is provided, you can modify some of this to suit your own needs, or just take excerpts for use in your own program. Note that none of this is useful unless you're in the Half-Bitmap mode already. The Public Domain disk ADVHB includes the routines to get you smoothly into and out of that mode.

We hope that some of you will find this useful. Next month we'll nose around some other dark corner of the TI, and write what we find.

```
* SIDEBAR64
* A.K.A. HBINP/S
*
*   FOR INPUTS TO ASSEMBLY PROGRAM
*   USING HALF-BITMAP SCREEN
*   Code by Bruce Harrison
*   26 MAY 1995
*   PUBLIC DOMAIN
*
       REF  VSBW,VSBR,VMBW,VMBR,KSCAN  REF UTILS
       REF  XMLLNK
       DEF  ACCNUM,ACCSTR,HBGK,HBGKF,ACCFP
*
* REQUIRED EQUATES
*
STATUS EQU  >837C        GPL STATUS BYTE
KEYADR EQU  >8374        KEY-UNIT
KEYVAL EQU  >8375        KEY VALUE
FAC    EQU  >834A        FLOATING POINT ACCUMULATOR
FAC12  EQU  >8356        FAC + 12
CSN    EQU  >1000        CONVERT STRING TO NUMBER
CFI    EQU  >1200        CONVERT FLOATING TO INTEGER
*
* ACCNUM accepts an integer input
* INVOKE ACCUM by:
*      BLWP @ACCNUM
*      BYTE ROW,COL
*      DATA CLRSIG
* ON EXIT, INTEGER RESULT IS AT >834A
*
*
ACCNUM DATA WS,ACNCOD

ACNCOD LI   R2,6         LENGTH 6
       CLR  R5           R5=0
       CLR  R12          R12=0
       BL   @GRC         R0 HAS SCREEN ADDRESS
       MOV  @>83C4,R9    SAVE EXISTING USER INTERRUPT
       MOV  *R14+,R3     R3 HAS CLEAR FIELD FLAG
       JMP  ACCST2       JUMP AHEAD
*
* ACCFP  accepts a floating point number input
* INVOKE ACCFP by:
*      BLWP @ACCFP
*      BYTE ROW,COL
*      DATA CLRSIG
* ON EXIT, FLOATING POINT NUMBER IS AT >834A
* (EIGHT BYTES IN RADIX 100 FORMAT)
*
ACCFP  DATA WS,ACFCOD
```

```
ACFCOD LI   R2,15        LENGTH 15
       CLR  R5           R5=0
       LI   R12,1        R12=1
       BL   @GRC         R0 HAS SCREEN ADDRESS
       MOV  @>83C4,R9    SAVE EXISTING USER INTERRUPT
       MOV  *R14+,R3     R3 HAS CLEAR FIELD FLAG
       JMP  ACCST2       JUMP AHEAD
*
* ACCSTR ACCEPTS A STRING INPUT
* INVOKE ACCSTR BY:
*      BLWP @ACCSTR
*      BYTE ROW,COL
*      BYTE MAXLEN,CLRSIG
*      DATA BUFFER ADDRESS
*
* ON EXIT, STRING IS AT BUFFER ADDRESS AS LENGTH,CONTENT
*
ACCSTR DATA WS,ACSCOD


ACSCOD BL   @GRC         R0 HAS SCREEN ADDRESS
       MOV  @>83C4,R9    SAVE INTERRUPT STATE
       MOVB *R14+,R2     R2 HAS MAX LEN
       SRL  R2,8         RT. JUST LEN
       MOVB *R14+,R3     R3 HAS CLEAR FLAG
       SRL  R3,8         RT. JUST CLRSIG
       MOV  *R14+,R5     R5 HAS ADDRESS OF BUFFER
       MOV  R2,R2        CHECK FOR ZERO LENGTH
       JEQ  EMEX         EXIT IF SO
       CI   R2,80        CHECK MAX ALLOWED
       JGT  EMEX         IF GREATER, EXIT
ACCST2 CLR  @INSFLG      NOT IN INSERT
       CLR  @KEYADR      KEY-UNIT 0
       MOV  R10,R10      CHECK POSITION ERROR
       JNE  EMEX         IF ERROR, EXIT
       MOV  R0,R7        SAVE START POSITION
       MOV  R2,R4        SAVE LENGTH
       A    R2,R0        ADD LENGTH
       CI   R0,>1B00     CHECK END OF SCREEN
       JLT  SAVEND       IF LESS, OKAY
EMEX   CLR  R8           EMERGENCY EXIT
       MOV  R5,R5        CHECK R5 FOR 0
       JEQ  EMEXX        IF SO, SKIP
       MOVB R8,*R5       ELSE NULL THE STRING
EMEXX  MOV  R9,@>83C4    PUT BACK OLD USER INTERRUPT
       RTWP              RETURN TO CALLER
SAVEND MOV  R0,R6        SAVE THAT POSITION
       DEC  R6           LAST ALLOWED
CLRSNS MOV  R7,R0        BACK TO START
       MOV  R3,R3        CHECK SIGNAL
```

```
       JEQ  KEYFRC       IF ZERO, JUMP
       MOV  R4,R2        GET LENGTH BACK IN R2
       MOVB @ANYKEY,R1   SPACE CHAR
CLRFLD BLWP @VSBW        WRITE ONE SPACE
       INC  R0           MOVE AHEAD ONE
       DEC  R2           DEC COUNT
       JNE  CLRFLD       IF NOT 0, RPT
       MOV  R7,R0        GET START BACK
*
* KEYFRC GETS THE CURRENT CHARACTER
* FROM THE SCREEN, FORCES THE CURSOR
* TO THAT POSITION, THEN ACTIVATES THE
* USER INTERRUPT TO BLINK CURSOR
*
KEYFRC BLWP @VSBR        READ BYTE AT R0 POSITION
       MOVB R1,@ALTKEY   PLACE AT ALTKEY
       MOVB @CURSOR,R1   PUT CURSOR IN R1
       BLWP @VSBW        WRITE CURSOR
       CLR  @>8378       CLEAR TIME COUNTER
       MOV  @INTLOC,@>83C4 ENABLE USER INTERRUPT
*
* KEYIN IS THE PART THAT GETS KEYSTROKES
*
KEYIN  BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            STOP THEM
       CB   @STATUS,@ANYKEY KEY STRUCK?
       JNE  KEYIN        IF NOT, REPEAT
*
* FOLLOWING CODE USES THE KEYSTROKE
*
       MOV  @KEYADR,R8   KEY AS WORD IN R8
       MOVB @ALTKEY,R1   OLD CHAR IN R1
       BLWP @VSBW        WRITE TO SCREEN
       CB   @KEYVAL,@ENTERV "ENTER" STRUCK?
       JEQ  KEYEX        IF YES, EXIT
       CB   @KEYVAL,@BACKUP FCTN-S?
       JNE  KEY0         IF NOT, JUMP
*
* FOLLOWING IS CODE THAT HANDLES FCTN-S
* IT MOVES CURSOR ONE SPOT, THEN GOES TO
* RPTKEY, WHICH DELAYS BEFORE ALLOWING REPEAT
*
       DEC  R0           BACK ONE
       C    R0,R7        CHECK FOR FIRST POS
       JGT  BCKX         IF GREATER, JUMP
       MOV  R7,R0        R7 TO R0
       B    @KEYFRC      BACK TO MAIN ENTRY
BCKX   B    @RPTKEY      AHEAD FOR REPEAT ACTION
```

```
KEY0   CB   @KEYVAL,@FWARD FCTN-D?
       JNE  KEY1          IF NOT, JUMP AHEAD
*
* FOLLOWING IS CODE THAT HANDLES FCTN-D
* IT MOVES CURSOR ONE SPOT, THEN GOES TO
* RPTKEY, WHICH DELAYS BEFORE ALLOWING REPEAT
*
       INC  R0            POINT AHEAD
       C    R0,R6         LAST SPOT?
       JLT  FWKX          IF LESS, JUMP
       MOV  R6,R0         ELSE POINT BACK
       B    @KEYFRC       THEN BRANCH BACK
FWKX   B    @RPTKEY       AHEAD FOR REPEAT ACTION
KEY1   CI   R8,32          COMPARE TO SPACE BAR
       JLT  FUNCT         IF LESS, CHECK FOR FUNCT
*
* FOLLOWING HANDLES KEY VALUES 32 AND ABOVE
*
       MOV  @INSFLG,R1    INSERT MODE?
       JEQ  KEY1A         IF NOT, JUMP AHEAD
*
* FOLLOWING HANDLES INSERT IF IN INSERT MODE
*
       C    R0,R6         AT END OF FIELD?
       JEQ  KEY1A         IF SO, NO INSERT
       MOV  R6,R2         GET LAST POSITION
       S    R0,R2         SUBTRACT CURRENT POSITION
       LI   R1,TEMSTR     TEMP STORAGE SPACE
       BLWP @VMBR         PUT BYTES THERE
       INC  R0            POINT AHEAD ONE
       BLWP @VMBW         WRITE THERE
DEC0   DEC  R0            BACK TO OLD POSITION
       JMP  KEY1A         PUT IN THE KEYSTROKE
*
* FOLLOWING HANDLES FUNCTION KEYS WITH VALUES BELOW 32
*
FUNCT  CB   @KEYVAL,@DELKEY DELETE KEY?
       JNE  FUNCT2        IF NOT, JUMP
*
* FOLLOWING HANDLES DELETE WITH FCTN-1
*
       MOV  R0,R3         STASH AWAY R0
       MOV  R6,R2         GET END OF FIELD
       S    R0,R2         SUBTRACT CURRENT POSITION
       JEQ  NULDEL        IF ZERO, JUMP AHEAD
       INC  R0            ELSE POINT AHEAD ONE
       LI   R1,TEMSTR     TEMPORARY STORAGE
       BLWP @VMBR         READ TO THERE
       DEC  R0            POINT BACK ONE
       BLWP @VMBW         WRITE TO THERE
```

```
NULDEL MOV  R6,R0        GET END OF FIELD
       MOVB @ANYKEY,R1   SPACE CHAR
       BLWP @VSBW        WRITE A SPACE
       MOV  R3,R0        GET OLD POSITION BACK
       JMP  KEYFRC       JUMP TO GET NEXT KEY
FUNCT2 CB   @KEYVAL,@INSKEY FCTN-2 PRESSED?
       JNE  FUNCT3       IF NOT, JUMP
*
* FOLLOWING SETS INSERT MODE ON FCTN-2
*
       INC  @INSFLG      SET INSERT FLAG
       JMP  KEYFRC       THEN BACK
FUNCT3 CB   @KEYVAL,@ERSKEY FCTN-3 PRESSED?
       JNE  FUNCTX       IF NOT, JUMP
*
* FOLLOWING ERASES FIELD IF FCTN-3 STRUCK
*
ERSFLD MOVB @ANYKEY,R3   SET R3 NON-ZERO
       B    @CLRSNS      BRANCH TO CLEAR FIELD
*
* FCTN-X OR FCTN-E
* OR FCTN-8 OR FCTN-9
* ALL EXIT PROGRAM
*
FUNCTX CI   R8,10        FCTN-X?
       JEQ  KEYEX        IF SO EXIT
       CI   R8,11        FCTN-E?
       JEQ  KEYEX        IF SO, EXIT
       CI   R8,15        FCTN-9?
       JEQ  KEYEX        IF SO, EXIT ROUTINE
       CI   R8,6         FCTN-8?
       JEQ  KEYEX        IF SO, EXIT
       B    @KEYFRC      ELSE IGNORE KEYSTROKE
*
* FOLLOWING PUTS CURRENT KEYSTROKE ON SCREEN
* THEN MOVES CURSOR TO NEXT SPOT
*
KEY1A  MOVB @KEYVAL,R1   GET KEY VALUE IN R1
       BLWP @VSBW        WRITE THAT
       INC  R0           POINT AHEAD
       C    R0,R6
       JLT  KEY1X
       MOV  R6,R0
KEY1X  B    @KEYFRC      THEN BRANCH BACK
*
* KEYEX IS THE EXIT FROM THIS ROUTINE
*
KEYEX  MOV  R9,@>83C4    PUT PREVIOUS USER INTERRUPT BACK
       MOV  R5,R5        NUMBER INPUT?
       JEQ  NUMOUT       IF SO, JUMP
```

```
       MOV  R4,R2        GET LENGTH
       MOV  R6,R0        AND LAST POSITION
RDBYT  BLWP @VSBR        READ A BYTE
       CB   R1,@ANYKEY   SPACE?
       JNE  RDSTR        IF NOT, JUMP
       DEC  R0           ELSE DEC POSITION
       DEC  R2           AND CHAR COUNT
       JNE  RDBYT        IF NOT ZERO, GO BACK
RDSTR  MOV  R5,R1        GET STRING LOCATION
       MOV  R7,R0        AND START POSITION
       MOVB @WS+5,*R1+   PUT LENGTH AT STRING ADDRESS
       JEQ  KEYXX        IF NULL, EXIT
       BLWP @VMBR        READ STRING FROM SCREEN
KEYXX  RTWP              RETURN TO CALLER
NUMOUT MOV  R7,@FAC12    START POSITION TO >8356
       BLWP @XMLLNK      USE XML VECTOR
       DATA CSN          CONVERT STRING TO NUMBER
       MOV  R12,R12      FLOATING POINT?
       JNE  FPOUT        JUMP IF SO
       BLWP @XMLLNK      USE XML VECTOR
       DATA CFI          CONVERT FLOAT TO INT
FPOUT  MOV  @FAC,R1      GET NUMBER
       JNE  NUMOX        IF NOT ZERO, OKAY
       MOV  R7,R0        GET START POINT
       MOV  R4,R2        AND LENGTH
       LI   R1,SIXBLN    SIX SPACES
       BLWP @VMBW        WRITE THOSE
       LI   R1,>3000     '0' IN LEFT BYTE R1
       BLWP @VSBW        ECHO 0 TO SCREEN
NUMOX  RTWP              RETURN TO CALLER
*
*
* FOLLOWING IS THE REPEAT-KEY FUNCTION FOR LEFT AND RIGHT
* MOVEMENT OF THE CURSOR
*
RPTKEY BLWP @VSBR        READ CURRENT CHAR
       MOVB R1,@ALTKEY   PLACE AT ALTKEY
       MOVB @CURSOR,R1   GET CURSOR
       BLWP @VSBW        WRITE THAT
       CLR  @INSFLG      CLEAR INSERT MODE
       CLR  @>8378       CLEAR TIMER
       CLR  @>83C4       DISABLE USRINT
*
* THE LOOP STARTING AT RPT1 DELAYS REPEAT MOTION FOR
* 32/60THS OF A SECOND UNLESS KEY IS RELEASED
*
RPT1   BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTS
       LIMI 0            STOP INTS
       CB   @KEYVAL,@NOKEY NO KEY?
```

```
       JEQ  RPTEX        IF SO, EXIT
       CB   @>8379,@ANYKEY COMPARE TO 32
       JLT  RPT1         IF LESS, JUMP
RPT1A  CLR  @>8378       CLEAR TIMER
       MOVB @ALTKEY,R1   GET ALTKEY BACK
       BLWP @VSBW        WRITE
       CB   @KEYVAL,@BACKUP BACKWARD?
       JNE  RPTF         IF NOT, JUMP
       DEC  R0           ELSE BACK ONE
       C    R0,R7        AT START OF FIELD?
       JGT  RPTFA        IF GREATER, OKAY
       JEQ  RPTFA        IF EQUAL, OKAY
       INC  R0           PUT POSITION BACK
       JMP  RPTEX        THEN EXIT
RPTF   INC  R0           AHEAD ONE
RPTF1  C    R0,R6        AT END OF FIELD?
       JLT  RPTFA        IF LESS, OKAY
       JEQ  RPTFA        IF EQUAL, OKAY
       DEC  R0           BACK ONE
       JMP  RPTEX        THEN EXIT
RPTFA  BLWP @VSBR        READ BYTE AT CURRENT POSITION
       MOVB R1,@ALTKEY   STASH CURRENT CHAR
       MOVB @CURSOR,R1   CURSOR IN R1
       BLWP @VSBW        WRITE CURSOR
*
* THE LOOP AT RPT2 DELAYS 8/60THS UNLESS KEY IS RELEASED
*
RPT2   BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            INTS ON
       LIMI 0            THEN OFF
       CB   @KEYVAL,@NOKEY NO KEY?
       JEQ  RPTEX        IF SO, EXIT
       CB   @>8379,@BACKUP COMPARE TO 8
       JLT  RPT2         IF LESS, REPEAT
*
* AFTER 8/60THS, CURSOR ADVANCES ANOTHER STEP
*
       JMP  RPT1A        ELSE JUMP BACK
RPTEX  MOVB @ALTKEY,R1   OLD CHAR
       BLWP @VSBW        WRITE THAT
       B    @KEYFRC      THEN BRANCH BACK
*
* FOLLOWING IS THE "BLINK", DONE WITH USER INTERRUPT
* EVERY 20 60THS, THIS WILL BLWP @CHVECT TO CHANGE
* FROM CURSOR TO CHARACTER OR VICE VERSA
*
USRINT CB   @>8379,@TWENTY TIMER=20?
       JLT  INTEX        IF LESS, EXIT
       BLWP @CHVECT      ELSE CHANGE CHAR
INTEX  RT                RETURN TO INTERRUPT HANDLER
```

```
*
* CHVECT CHANGES FROM CURSOR TO CHAR AND VICE VERSA
* EVERY 20/60THS OF A SECOND.  (THAT'S 1/3 SECOND)
*
CHVECT DATA INTWS,CHG0
CHG0   MOV  @WS,R0       GET MAIN R0 INTO THIS R0
CHG1   BLWP @VSBR        READ CURRENT BYTE FROM SCREEN
       CB   R1,@CURSOR   IS THAT CURSOR?
       JEQ  CHG2         IF YES, JUMP
       MOVB @CURSOR,R1   ELSE GET CURSOR
       BLWP @VSBW        AND WRITE THAT
       JMP  CHGX         THEN EXIT
CHG2   MOVB @ALTKEY,R1   PUT OLD CHAR IN R1
       BLWP @VSBW        WRITE THAT
CHGX   CLR  @>8378       CLEAR TIMER
       RTWP              THEN RETURN
*
* HBGKF takes a single keystroke
* AT ROW,COL
* WITH FLASHING CURSOR

* and flashes the cursor
* INVOKE HBGKF by:
*      BLWP @HBGKF
*      BYTE ROW,COL
* ON EXIT, KEYSTROKE IS IN USERS R1
* AND IS ECHOED AT ROW, COL
HBGKF  DATA WS,HBF
HBGK   DATA WS,HBG
HBF    BL   @GRC         SET R0 TO SCREEN ADDRESS
       MOV  R10,R10      NO ERROR?
       JNE  EMERX2       IF ERROR, JUMP
       BLWP @VSBR        READ PRESENT CHAR
       MOVB R1,@ALTKEY   PLACE AT ALTKEY
       MOVB @CURSOR,R1   CURSOR
       BLWP @VSBW        WRITE THAT
       MOV  @>83C4,R9    SAVE USER INTERRUPT
       MOV  @INTLOC,@>83C4  ENABLE USER INT
       CLR  @>8378       CLEAR VDP TIMER
       BL   @KEYLOO      WAIT FOR KEYSTROKE
       CB   @KEYVAL,@ENTERV ENTER PRESSED?
       JNE  HBFX         IF NOT, JUMP
       MOVB @ALTKEY,@KEYVAL PUT PREVIOUS CHAR IN KEYVAL
       MOV  @KEYADR,R8   KEY AS WORD IN R8
HBFX   MOV  R9,@>83C4    PUT BACK OLD USER INT
       JMP  HBGKX        JUMP AHEAD
*
* HBGK accepts a single keystroke
* AND ECHOES AT ROW, COL
* WITHOUT CURSOR
```

```
* INVOKE HBGK by:
*       BLWP @HBGK
*       BYTE ROW,COL
* ON EXIT, KEY VALUE IS IN CALLER'S R1
*
HBG     BL   @GRC          SCREEN LOCATION TO R0
        MOV  R10,R10       NO ERROR?
        JNE  EMERX2        IF ERROR, JUMP
        BL   @KEYLOO       GET KEYSTROKE
HBGKX   MOVB @KEYVAL,R1    KEY VALUE IN R1
        BLWP @VSBW         ECHO TO SCREEN
        MOV  @KEYADR,@2(R13)   PUT IN CALLER'S R1
        RTWP               RETURN TO CALLER

EMERX2  CLR  @2(R13)       CLEAR CALLER'S R1
        RTWP               RETURN
*
* KEYLOO WAITS FOR A KEYSTROKE, THEN RETURNS
* THE VALUE OF THE KEY STRUCK GOES INTO REG 8
*
KEYLOO  CLR  @KEYADR       KEY-UNIT 0
KEYLO1  BLWP @KSCAN        SCAN KEYBOARD
        LIMI 2             ALLOW INTS
        LIMI 0             THEN STOP
        CB   @STATUS,@ANYKEY ANY KEY?
        JNE  KEYLO1        IF NOT, REPEAT
        MOV  @KEYADR,R8    KEY AS WORD INTO R8
        RT                 THEN RETURN
* GRC IS INTERNAL SUBROUTINE THAT GETS
* THE ROW, COL FROM CALLER'S DATA
* AND TRANSLATES THAT TO SCREEN ADDRESS IN R0
*
GRC     CLR  R10           CLEAR ERROR FLAG
        MOVB *R14+,R0      ROW IN R0
        MOVB *R14+,R1      COL IN R1
        SRL  R0,8          RT. JUST ROW
        SRL  R1,8          RT. JUST COL
        DEC  R0            ZERO BASE ROW
        DEC  R1            ZERO BASE COL
        SLA  R0,5          ROW * 32 IN R0
        A    R1,R0         ADD COL TO R0
        JLT  GRCERR        IF < 0, ERR
        CI   R0,>2FF       CHECK SCREEN LIMIT
        JGT  GRCERR        IF >, ERR
        AI   R0,>1800      ADD SCREEN OFFSET
        RT                 RETURN
GRCERR  INV  R10           SET ERROR FLAG
        RT                 RETURN
*
* DATA SECTION
```

```
*
WS     BSS  32           OUR WORKSPACE
INTWS  BSS  32
INTLOC DATA USRINT       USER INTERRUPT ADDRESS
NUMFLG DATA 0            NUMBER INPUT FLAG
INSFLG DATA 0            INSERT FLAG
DELKEY BYTE 3            FCTN-1 VALUE
INSKEY BYTE 4            FCTN-2 VALUE
ERSKEY BYTE 7            FCTN-3 VALUE
TEMSTR BSS  80           TEMPORARY STRING
ALTKEY BYTE 0            CURRENT CHARACTER FROM SCREEN
ENTERV BYTE 13           ENTER KEY VALUE
CURSOR BYTE 30           CURSOR CHAR
BACKUP BYTE 8            FCTN-S
FWARD  BYTE 9            FCTN-D
ANYKEY BYTE 32           SPACE OR COMPARISON BYTE
TWENTY BYTE 20           CURSOR BLINK NUMBER
NOKEY  BYTE >FF          NO KEY INDICATION
EDGE   BYTE 31           EDGE CHAR
SIXBLN TEXT '               '
       END
```

1.65. The Art Of Assembly — Part 65. 3X5=1X1

By Bruce Harrison

Talk about new math! The non-equation in today's subtitle is not a joke, but a magic formula that we actually used in two of our Assembly programs. The formula applies for the case of printing pictures from the TI screen (Bitmap wise) on a dot matrix printer. It all started back when we created our own drawing program. We wanted to have the printing function keep the proportions (or aspect ratio) the same on paper as it appears on the screen. Thus the image area had to be in exactly the same ratio height-to-width as the screen image area. Doing this means that circles on the screen are circles on the paper, and squares are squares, etc.

1.65.1. A Common Denominator

All of our dot matrix printers are of the 9-pin variety, so keep in mind that this discussion refers to those, and not the more modern 24-pin type. The printhead on such printers has nine pins arranged vertically, spaced 1/72nd of an inch apart. In their normal "graphics" mode, these printers make dots at 1/60th of an inch spacing across the paper. What was needed was some way to make the horizontal and vertical dimensions of each bit in the screen image as a square area on the paper. Of course this had to be done in a way to allow 192 by 256 pixels to fit on the 8 1/2 by 11 inch paper, too. The answer was to use the printer's double density graphics mode, in which it puts the dots at 120 per inch across.

1.65.2. The Solution

For each bit in the image, we fire three pins at the top of the printhead, then replicate that five times horizontally. Thus one pixel from the screen is 3/72nds in height and 5/120ths wide. Now get out your calculator and divide 3 by 72. Write that down, then divide 5 by 120. Eureka! Since these fractions are identical in value, our pixel will appear square on the paper. Now of course we have to see if that will fit on the paper. If each pixel in the height of the screen (width of the paper) occupies 5/120ths, and there are 192 such, that equals exactly 8 inches, so the height dimension is okay. Now along the length of the sheet, we need 256 pixels. Thus we get out the calculator again and multiply 3/72nds by 256. That comes out to 10.66666. . . (10 2/3) inches, so we're safe on the 11 inch paper length as well.

1.65.3. Why Do This?

In case anybody missed it, we don't own TI-Artist. What we were seeking in our most recent experiment was a way to print TI-Artist picture files without having that program. We had another incentive to do this. Our friend Charles Kirkwood, Jr. wrote complaining that he couldn't get his own TI-Artist pictures to print correctly. He sent along some examples, so we started experimenting. Our own Drawing program can print its own pictures, but it can't handle all of the picture area of the TI-Artist pictures. (It leaves off 24 dot-rows.) What we wanted to make for our own use and for our friend Mr. Kirkwood was a very simple program that would both show the pictures on screen and send them to the printer.

1.65.4. Borrowed Source Code

As we often do, we borrowed from ourselves. We took some source code from our "picture show" program, to catalog a disk and produce a listing of the TI-Artist picture files that are on the selected disk. Now when the user selects one from that list, the program takes that picture from that disk and displays it on the screen. To complete the program, then, we took the section of our Drawing program that sends the pixels to the printer, modified it to use the entire TI-Artist picture, and added that to the selection source code from our Picture Show. We added one small thing in the beginning, so that when the program starts up, it will prompt for the printer's file name, with a default entry of PIO.CR. This way, most users will just have to press **ENTER** at the prompt, but for those whose printer is connected via RS-232, there's a way to "fix" that for the duration of the run. Next there's a prompt for a drive number, and that will accept a single keystroke in the ranges 1-9 or A-Z. Given that, the program catalogs the disk in the designated drive. If there's no such drive, or no disk in that drive, an error gets reported. If the catalog contains no TI-Artist Picture files, that will be reported. Given a disk that contains one or more picture files, the screen will show a list of the picture files on that disk, with a selection marker. The user then moves that marker to any file name and presses **ENTER**.

The program accesses the selected drive and puts the selected picture on screen. The program now waits for a keystroke from the user. Pressing **P** or **p** on the keyboard will start the printing process, any other key will simply put back the list of files, with the marker where it was. We figured that P for Print was an easy enough mnemonic connection for most anybody. During printing, the picture just remains on the screen. When printing finishes, the list comes back.

1.65.5. A Rave Review

When we sent an advance copy of this new product to Charles Kirkwood, we got a very enthusiastic response, and some examples of how TI-Artist printed his pictures. The pictures in question were all on a "railroad" theme, including a very nice rendition of an old fashioned steam engine. Steam engines have large wheels on them, and those are supposed to be round! Charles made them so they are nicely round on the screen, but on the TI-Artist printouts they came out as ovals. On printouts made with our new program, the wheels are round, as they should be.

We have in our collection a couple of Ken Gilliland's "Disk Of. . ." products. These contain very nice TI-Artist pictures, including even a self-portrait of Ken. We tried printing these with our new program, and the results were just what we'd hoped for, in that the printed image nearly fills the sheet of paper, and looks exactly like the screen image.

1.65.6. But Then Again . . .

Our little program is not a panacea for the TI-Artist users of the world. We have a collection of very colorful pictures from the Lima library, but none of them will print out as anything but "garbage" in black and white. Seems that the folks who created these took advantage of being able to manipulate the foreground and background colors to make their pictures very pretty on the screen, but this means that the pattern part is not a black-and-white rendering of the image content. Thus we recommend that our new product should only be used with the black and white type of picture, of the sort found in Notung's "Disk of. . ." series.

The other question that plagues users of dot matrix printers is of course the business of control codes. We used very simple control codes that are common to most models of the Epson and Star Micronics brands, and also most 9-pin Panasonic printers. Thus the program works correctly on our Star NX-1000, on our Gemini-10X, and on Charles' Epson RX-80. It will probably work just as well on the Epson FX series, and on many Panasonic models.

We have as usual released the program as Public Domain software, on a SS/SD disk called TIAPRINT. That's available through the Lima users' group. The disk contains instructions, an XB program to print the instructions, an XB LOAD program to run the Assembly program from XB, and two sample pictures kindly donated by Charles Kirkwood, Jr. We hope you'll all get some pleasure from this.

1.65.7. Today's Sidebar

In the Sidebar is the "printing" part of the source code, with its subroutines and data sections, so you who are interested can follow just how we did this 3X5=1X1 trick. If you're using a 24-pin printer, you'll probably see that the same control sequences we've used are available on your printer, but that they perform differently. On the 9-pin models, for example, the ESC A n sets up for an n/72nds line feed, and when followed by >A, the paper advances by 3 72nds if n was 3.

For those with 24 pin printers, though, the ESC "A" sequence would result in a line feed measured in 60ths of an inch, not 72nds. Fortunately, our friend Harley Ryan sent us an extra copy of the manual for his Panasonic KX-P2123 24 pin model. It turns out that on 24 pin printers you can use ESC "+" to set line feed amounts in 360ths of an inch. A little simple math shows that 360 is exactly 5 times 72. Thus by multiplying our 3 by 5, we were able to create a version of the program that makes line feeds of 15/360ths, which reduces to 3/72nds, thus keeping our "square pixel" idea alive for the 24 pin printer. We sent copies of this new version out to Harley, and his test showed that it worked exactly as planned. It's also been tested on a 24 pin Epson by Earl Raguse, and worked perfectly there, too.

Thus there are new versions of our Drawing Program (DRAW24) and our TI-Artist printing program (TIAPRN24) which have been released through the Lima Users' Group for use by anyone having the 24 pin problem.

Next month, we'll discuss the addition of circles to our Drawing program, by use of another Bresenham algorithm.

```
* SIDEBAR 65
* PRINTING PORTION - JUST A FRAGMENT
* (NOT ALL SUBROUTINES ARE SHOWN)
* CODE BY Bruce Harrison
* PRINTER OUTPUT
* 15 JUN 1995
PRNBUF EQU  >3800         BUFFER IN VDP RAM
PRNOUT LI   R1,PPABDT     PRINTER PAB DATA
       BL   @FILOP        OPEN THE FILE
       JNE  PRNSET        IF SUCCESS, JUMP
       BL   @CLOSE        ELSE CLOSE
       BL   @BLNK         BLANK SCREEN
       BL   @SETGM        BACK TO GRAPHICS
       BL   @CLS          CLEAR SCREEN
       BL   @UNBLNK       UNBLANK
       LI   R1,PNAMSG     PRINTER NOT AVAILABLE
       BL   @ERRRPT       REPORT ERROR
       B    @SAVE0        THEN BACK TO MAIN PROGRAM
PRNSET LI   R5,32         32 COLUMNS OF PICTURE
       LI   R12,>1707     START AT LOWER RIGHT CORNER
PRNOTL MOV  R12,R3        PUT ADDRESS IN R3
       LI   R4,24         24 ROWS
       CLR  R14           R14=0
       LI   R10,TEMSTR    TEMPORARY STRING
PRNMIL LI   R6,8          8 BYTES
PRNINL MOVB @PIXBUF(R3),*R10+ GET A BYTE
       JEQ  PDEC3         IF ZERO, JUMP
       INC  R14           ELSE INCREMENT R14
PDEC3  DEC  R3            BACK ONE BYTE
       DEC  R6            DONE 8?
       JNE  PRNINL        IF NOT, JUMP
       AI   R3,-248       NEXT CHAR IN COLUMN
       DEC  R4            DONE 24?
       JNE  PRNMIL        IF NOT, REPEAT
       MOV  R14,R14       CHECK R14
       JNE  PRLD8         IF NOT ZERO, JUMP
       LI   R0,PRNBUF     LOAD PRINT BUFFER
       LI   R1,SPLF       ADVANCE 24/72NDS
       BL   @DISSTR       PLACE STRING
       BL   @PRNSND       SEND TO PRINTER
       JMP  PRDEC5        THEN JUMP AHEAD
PRLD8  LI   R13,8         8 BITS/BYTE
PLRTS  LI   R4,4          FOUR GROUPS
```

```
        LI   R0,PRNBUF    POINT AT BUFFER
        LI   R1,BGRSTR    BIT GRAPHICS CONTROL STRING
        BL   @DISSTR      PLACE IN BUFFER
        BL   @PRNSND      SEND TO PRINTER
        LI   R10,TEMSTR   POINT AT BYTES FOR 1 COLUMN
PRLBUF  LI   R0,PRNBUF    PRINT BUFFER
        LI   R2,48        GROUP BY 48
PRMV1   MOVB *R10+,R1     GET A BYTE
        ANDI R1,>8000     MASK ONLY MSB
        SRA  R1,2         REPLICATE IN THREE MSBS
        BLWP @VSBW        WRITE THAT
        INC  R0           NEXT ADDR
        BLWP @VSBW        WRITE AGAIN
        INC  R0           NEXT ADDR
        BLWP @VSBW        WRITE AGAIN
        INC  R0           NEXT ADDR
        BLWP @VSBW        WRITE AGAIN
        INC  R0           NEXT ADDR
        BLWP @VSBW        WRITE 5TH TIME
        INC  R0           NEXT ADDR
        DEC  R2           DONE 48?
        JNE  PRMV1        IF NOT, REPEAT
        LI   R2,240       240 BYTES IN BUFFER
        BL   @PRNSND      SEND THOSE TO PRINTER
        DEC  R4           DONE 4 GROUPS?
        JNE  PRLBUF       IF NOT, ANOTHER GROUP
PRCRLF  LI   R0,PRNBUF    PRINT BUFFER
        LI   R1,CRLSTR    CR/LF (3/72NDS)
        BL   @DISSTR      PLACE IN BUFFER
        BL   @PRNSND      SEND
        LI   R10,TEMSTR   COLUMN STORED
        LI   R2,192       192 BYTES
PRNSHL  MOVB *R10,R1      GET ONE
        SLA  R1,1         SHIFT LEFT BY ONE
        MOVB R1,*R10+     PUT BACK AND INC POINTER
        DEC  R2           DONE 192?
        JNE  PRNSHL       IF NOT, REPEAT
        DEC  R13          USED ALL 8 BITS?
        JNE  PLRTS        IF NOT, REPEAT FOR NEXT BIT
PRDEC5  AI   R12,8        NEXT COLUMN
        DEC  R5           DONE 32?
        JNE  PRNOTL       IF NOT, CONTINUE
        LI   R0,PRNBUF    POINT AT PRINT BUFFER
        LI   R1,FFSTR     FORM FEED
        BL   @DISSTR      PUT IN BUFFER
        BL   @PRNSND      PICTURE FINISHED
PRNCLS  MOV  @PABLOC,R0   PAB LOCATION
        MOVB @ONE,R1      CLOSE OPCODE
        BLWP @VSBW        WRITE THAT
        BL   @FILOP3      CLOSE THE FILE
```

```
       B    @SAVE0       BACK TO MAIN PROGRAM
*
PRNSND MOVB @DELKEY,R1   WRITE OPCODE
       MOV  @PABLOC,R0   GET PAB LOCATION
       BLWP @VSBW        WRITE ONE BYTE
       AI   R0,5         ADD FIVE
       SWPB R2           LENGTH TO LEFT BYTE
       MOVB R2,R1        PUT IN R1
       BLWP @VSBW        WRITE LENGTH
       AI   R0,4         ADD 4
       MOV  R0,@>8356    AT DSR POINTER
       BLWP @DSRLNK      USE DSR LINK
       DATA 8            WRITE TO FILE
       JNE  PRNOK        IF NOT "EQUAL", NO ERROR
       BL   @BLNK        ELSE BLANK SCREEN
       BL   @SETGM       GO TO GRAPHICS MODE
       BL   @CLS         CLEAR THE SCREEN
       BL   @UNBLNK      UNBLANK
       LI   R1,PTMSTR    PRINT TERMINATED
       LI   R0,21*32+3   ROW 22, COL 4
       LI   R15,PRNCLS   LOAD RETURN ADDR
       JMP  ERRRP0       THEN JUMP TO ERROR TRAP
PRNOK  RT                RETURN
*
ERRRPT MOV  R11,R15      SAVE R11 IN R15
       LI   R0,21*32+4   ROW 22, COL 5
ERRRP0 BL   @DISSTR      DISPLAY STRING
       LI   R0,23*32+4   ROW 24, COL 5
       LI   R1,PAK       "PRESS ANY KEY"
       BL   @DISSTR      DISPLAY THAT
       BLWP @KSCAN       CHECK KEYBOARD
       BL   @KEY         THEN WAIT FOR KEYPRESS
       LI   R0,21*32     ROW 22, COL 1
       LI   R4,32*3      3 ROWS
       BL   @CLRFLD      CLEAR THE ERROR MESSAGE
       B    *R15         THEN RETURN
FILOP  MOV  @8(R1),R2    LENGTH INTO R2
       AI   R2,10        10 FIXED DATA
       MOV  @>8370,R0    HIGH VDP ADDR
       S    R2,R0        SUBTRACT LENGTH
       MOV  R0,@PABLOC   SAVE PAB VDP LOCATION
FILOP2 BLWP @VMBW        WRITE TO VDP
FILOP3 AI   R0,9         ADD 9 TO R0
       MOV  R0,@PABPNT   PUT AT >8356
       BLWP @DSRLNK      USE DSR LINK
       DATA 8            FOR FILE ACCESS
       RT
*
* DATA SECTION
*
```

```
TEMSTR BSS  256
PNAMSG BYTE 21
       TEXT 'PRINTER NOT AVAILABLE'
PRNNAM BYTE 17
       TEXT 'PRINTER FILE NAME'
PTMSTR BYTE 19
       TEXT 'PRINTING TERMINATED'
FFSTR  BYTE 3,12,27,'@'
SAVBYT BYTE 6
LODBYT BYTE 5
HEX80  BYTE >80
PPABDT DATA >0012,PRNBUF,>FE00,0,>0006
       TEXT 'PIO.CR                      '
BGRSTR BYTE 4,27,'L',192,3
CRLSTR BYTE 5,13,27,'A',3,10
SPLF   BYTE 5,13,27,'A',24,10
*
* ALTERNATE DATA VERSIONS FOR
* 24 PIN PRINTERS
* THE FOLLOWING TWO LINES
* ARE USED IN PLACE OF THEIR
* COUNTERPARTS FOR 24 PIN PROGRAMS
* DRAW24 AND TIAPRN24
*
CRLSTR BYTE 5,13,27,'+',15,10
SPLF   BYTE 5,13,27,'+',120,10
```

1.66. The Art Of Assembly — Part 66. Running in Circles

By Bruce Harrison

Some time back, we wrote about drawing a straight line on the computer screen, and presented a method based on an algorithm developed by a man named Bresenham. That algorithm uses only very simple integer math to draw an optimal straight line on our screen. That article caused some reaction among our readers about who this fellow Bresenham was and how his method actually worked. We received correspondence from John H. Bull and Phil Van Nordstrand, both of whom started "searching" for the mysterious Bresenham. Phil Van Nordstrand found a reference in Dr. Dobbs' Journal about a circle drawing algorithm by the very same Bresenham. Now that really made us curious. Could this genius Bresenham have made a "simple math" way of generating circles as well? How could he avoid using sines or cosines or square roots and still generate a circle? Yes, he could, and in a way that's even more mysterious than the straight line.

Phil Van Nordstrand was able to find one of the books referenced in the Dr. Dobbs source, in the Houston Public Library. He sent along copies of a few pages, which had the algorithm as implemented in Pascal code. After studying this for a few minutes, it became evident that this would be easy to translate from Pascal to TI Assembly, and that it would execute fairly fast. The algorithm requires that some numbers be multiplied, but always by powers of two, and thus simple SLA instructions would accomplish the needed multiplications. Other than that all the math is simple integer comparison, addition, and subtraction.

1.66.1. The Algorithm

We enter the algorithm with three numbers. These are the X and Y coordinates of the circle's center and its radius. The algorithm itself calculates points to be plotted only for one-eighth of a circle, so we have to devise our own method of replicating the points for the rest. The algorithm uses a single parameter which is initially derived from the radius by this formula:

```
P=3-2*R
```

For the moment, we'll ignore the center coordinates. The first point to be plotted is at the top of the circle, so X is zero and Y is equal to R. (In our implementation, we add the center coordinates for each point as it gets plotted.) We now plot the point at the top of the circle. After each plotted point, we examine the parameter P, and adjust its value in one of two ways.

If P<0, we don't change Y, but adjust the value of P by this formula:

```
P=P+4*X+6
```

If P>=0, we calculate the new value of P by the formula:

```
P=P+4*(X-Y)+10
```

and then subtract one from Y. Note the formula uses the values of X and Y from the previous point, so Y is adjusted after calculating the new value of P.

In either case, we increment X after calculating the new value of P. We continue doing this until X is greater than Y, which indicates that we've finished 1/8th of a circle. Notice that there are multiplications involved in the derivation and adjustment of P, but these are by 2 or 4 in all cases, so that in our Assembly version we can accomplish the multiply by shifting a register left by one or two bits.

1.66.2. Our Assembly Implementation

In our implementation, which is in the Sidebar at label BCIRC, we've stashed away the center's coordinates, so the algorithm only needs to deal in "deltas" for X and Y. At the outset, we establish four such deltas, called DELXP, DELXM, DELYP and DELYM. The DELX's are set to zero, and the DELYP and DELYM are set to plus and minus the value of the Radius, respectively. For each pass through the algorithm, then, we plot four points, two in the top quarter circle and two in the bottom. Thus our circle grows from top and bottom centers both left and right, so that when we reach the 1/8th circle limit, we've got a half circle, one quarter at the top and one quarter at the bottom. To complete the circle, we repeat the entire process (at label HALF2) with the roles of DELX and DELY interchanged, so the quarter circles at either side get drawn. All of this is being done in Bitmap Mode, so the circle is a single-pixel thickness. Each point gets plotted by the PLOT subroutine, which dates back to our first column on Bitmap operation (Part 42).

As with the Bresenham Line algorithm, we can see that this works, and it makes optimally round circles on our screen, but we don't know why it works. If we knew where to reach him, we'd ask Bresenham, but probably wouldn't understand his answer. John Bull was able to discover that Bresenham was a mathematician who worked for IBM. He actually found the original papers containing derivations and proofs of the algorithm's effectiveness. As John and I suspected, these are pretty heavy going.

1.66.3. Today's Sidebar

Yes, it's a complete program in E/A source code, with as much annotation as we could stand to do. This program uses modified versions of our old standby SETBM and SETGM subroutines, the old PLOT subroutine, and of course the BCIRC subroutine, which uses PLOT to place the pixels on the circles. The action starts with a circle at the center of the screen with radius 10, then keeps adding ten to the radius until the screen is filled with concentric circles. Note that the outer ones would run off the screen edges, but we've put in a limit check before actually plotting each point, so the outer circles go up to but not past the screen edges. In some cases you might want your circles to "wrap around" from edge to edge of the screen, but we'll leave the method for doing so to the serious student of Assembly. (Hint: this is easier than you might think.)

Before the mail comes, we'll confess that the code in today's Sidebar is nowhere near optimized in any respect. It's the result of a quick and easy attempt to test and apply the algorithm, so it may even appear crude and wasteful of memory. Nonetheless, it shows that this works, and quite well. The algorithm itself is crash-proof. You can, for example, start with radius=0, and the screen will just get a single pixel plotted at the center. For those of you who want to try this, just go into the Sidebar and find label CIRCLE. Make that label say either CLR R0 or LI R0,0, then assemble the result. When run, you'll see a single dot at the center of the family of concentric circles. That's the circle of radius 0. Radius 1 will produce a single pixel open space surrounded by four black pixels in a "diamond" pattern.

The Sidebar program doesn't do much. It puts the computer into Bitmap Mode, then creates a series of concentric circles starting with radius 10 and growing by 10 on each iteration. Only the first 9 will fit completely on the screen. The rest are shown partially only so far as their pixels fit on the screen. The code at label CPLM makes sure that we don't try putting any points off the edges of the screen.

For the benefit of those who get *MICROpendium* on disk, we've included the object file SIDE66/O along with this submission. As in the case with Bresenham's line drawing algorithm, we've also done a simple Extended Basic implementation of the circle, as shown in the Sidebar in 28 column listing. This XB program should also be on your *MICROpendium* disk as CIRCXB. This will give you a chance to play around with the algorithm without having to use Assembly code. Since this XB version goes slowly and puts a very coarse representation on the screen with cursor characters, you'll be able to see more clearly what's happening as the circle gets generated.

The circle algorithm, pretty much as shown here, has been incorporated into our Drawing program, in both the 9-pin and 24-pin versions. As we mentioned last month, those programs allow circles from the screen to be printed as circles on paper. If you're using an old version of our Drawing program, now might be a good time to order an updated copy from our friends in Lima, Ohio.

That's it for this time. Next time we'll be discussing the topic of "sound lists", and some new ideas and programs to make that concept easier to grasp and use. See you then.

```
* SIDEBAR 66
* BITMAP CIRCLES
* CODE BY Bruce Harrison
* 20 JUL 1995
* PUBLIC DOMAIN
*
* A COMPLETE PROGRAM THAT PUTS THE COMPUTER
* INTO BITMAP MODE AND DRAWS A SERIES OF
* CONCENTRIC CIRCLES USING BRESENHAM'S ALGORITHM
*
       DEF  START         DEFINE ENTRY POINT
       REF  VWTR,KSCAN,VMBW,VMBR,VSBW,VSBR
START  LWPI WS            LOAD OUR WORKSPACE
       LI   R0,>380       POINT AT COLOR TABLE
       LI   R1,SAVCLR     AND AT STORAGE SPACE
       LI   R2,32         32 BYTES TO GET
       BLWP @VMBR         READ COLOR TABLE INTO STORAGE
       MOV  @>8370,R0     GET VDP ADDR FROM >8370
       LI   R1,ANYKEY+1   POINT AT STORAGE BUFFER
       LI   R2,6          SIX BYTES TO READ
       BLWP @VMBR         READ THOSE INTO BUFFER
       LI   R0,>800       POINT AT CHARACTER TABLE
       LI   R1,CHRTBL     AND AT BUFFER STORAGE
       LI   R2,256*8      256 CHARACTER DEFINITIONS
       BLWP @VMBR         STASH CHARACTER DEFS
       BL   @SETBM        BITMAP MODE
CIRCLE LI   R0,10         RADIUS 10
       LI   R8,95         CENTER DOT-ROW
       LI   R7,127        CENTER DOT-COLUMN
       CLR  R9            BLACK ON WHITE
       MOV  R7,@SAVR7     STASH CENTER COLUMN
       MOV  R8,@SAVR8     STASH CENTER ROW
CIRCLP MOV  R0,@SAVR0     STASH RADIUS
       BL   @BCIRC        DRAW A CIRCLE
       MOV  @SAVR0,R0     GET RADIUS BACK
       MOV  @SAVR7,R7     GET CENTER COL
       MOV  @SAVR8,R8     GET CENTER ROW
       AI   R0,10         ADD 10 TO RADIUS
       CI   R0,160        COMPARE TO 160
       JLT  CIRCLP        IF LESS, REPEAT
KEYLOO BLWP @KSCAN        SCAN KEYBOARD
       LIMI 2             INTS ON
       LIMI 0             INTS OFF
       CB   @ANYKEY,@>837C KEY PRESSED?
       JNE  KEYLOO        IF NOT, REPEAT
       BL   @SETGM        SET GRAPHICS MODE
EXIT   MOV  @>8370,R0     GET BACK >8370 ADDRESS
       LI   R1,ANYKEY+1   POINT AT BUFFER STORAGE
       LI   R2,6          SIX BYTES
       BLWP @VMBW         WRITE THOSE BACK TO VDP
```

```
       LWPI >83E0        LOAD GPL WORKSPACE
       B    @>6A         RETURN TO GPL INTERPRETER
*
*
* SUBROUTINES FOR HANDLING BITMAP
* OPERATIONS AND TRANSITIONS
*
* FOLLOWING SECTION SETS COMPUTER INTO BITMAP MODE
*
SETBM  LI   R0,>1A0      SET TO BLANK
       BLWP @VWTR        BLANK OUT SCREEN
       LI   R0,>206      SET TO WRITE VDP REGISTER 2
       BLWP @VWTR        SIT TO >1800  (SCREEN IMAGE TABLE)
       LI   R0,>403      SET TO WRITE TO VDP REG. 4
       BLWP @VWTR        PDT TO >0000  (PATTERN DESCRIPTOR TABLE)
       LI   R0,>3FF      SET TO WRITE TO VDP REG 3
       BLWP @VWTR        CT TO >2000   (COLOR TABLE)
       LI   R0,>607      SET TO WRITE VDP REG 6
       BLWP @VWTR        Sprite descriptor table to >3800
       LI   R0,>570      SET TO WRITE VDP REG 7
       BLWP @VWTR        Sprite attribute list to >3800
       LI   R0,>58       INITIALIZE SCREEN IMAGE TABLE (SIT) (AT >1800)
       MOVB R0,@>8C02    WRITE LOW BYTE VDP ADDRESS
       SWPB R0           SWAP R0
       MOVB R0,@>8C02    WRITE HIGH BYTE VDP ADDRESS
       LI   R0,3         THREE TABLES OF 256 BYTES EACH
       CLR  R1           START WITH ZERO
SIT    MOVB R1,@>8C00    WRITE TO VDP (SELF-INCREMENTING)
       AI   R1,>100      ADD 1 TO HIGH BYTE R1
       JNE  SIT          IF NOT ZERO, REPEAT
       DEC  R0           ELSE DEC COUNT
       JNE  SIT          IF NOT ZERO, REPEAT
       LI   R0,>60       INIT COLOR TABLE  (CT) AT >2000
       MOVB R0,@>8C02    WRITE LOW BYTE OF ADDRESS
       SWPB R0           SWAP R0
       MOVB R0,@>8C02    WRITE HIGH BYTE OF ADDRESS
       LI   R0,>1800     >1800 BYTES TO WRITE
       LI   R1,>1F00     COLORS ALL BLACK ON WHITE
CT     MOVB R1,@>8C00    WRITE ONE BYTE
       DEC  R0           DEC COUNT
       JNE  CT           IF NOT ZERO, REPEAT
CPDT   LI   R0,>40       CLEAR PATTERN DESCRIPTOR TABLE  (PDT) AT >0000
       MOVB R0,@>8C02    WRITE LOW BYTE ADDR
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH BYTE ADDRESS
       LI   R0,>1800     >1800 BYTES TO WRITE
       CLR  R1           ALL ZEROS
PDT    MOVB R1,@>8C00    WRITE ONE
       DEC  R0           DEC COUNT
       JNE  PDT          IF NOT ZERO, REPEAT
```

```
       LI   R0,2         SET R0 TO WRITE 2 TO VDP REGISTER ZERO
       BLWP @VWTR        SET TO M3 MODE (BITMAP)
       LI   R0,>1E0      UNBLANK
       BLWP @VWTR        WRITE THAT
       RT
*
* FOLLOWING SETS COMPUTER BACK TO NORMAL GRAPHICS MODE
*
SETGM  LI   R0,>1A0      SET TO WRITE VDP REG 1 (BLANK SCREEN)
       BLWP @VWTR        WRITE
       LI   R0,>200      SET TO WRITE VDP REG 2
       BLWP @VWTR        WRITE
       LI   R0,>401      SET TO WRITE VDP REG 4
       BLWP @VWTR        WRITE
       LI   R0,>30E      VDP REG 3
       BLWP @VWTR        WRITE
       LI   R0,>600      VDP REG 6
       BLWP @VWTR        WRITE
       LI   R0,>506      VDP REG 5
       BLWP @VWTR        WRITE
       LI   R0,>380      POINT AT COLOR TABLE
       LI   R1,SAVCLR    AND AT SAVED COLOR DATA
       LI   R2,32        32 BYTES
       BLWP @VMBW        WRITE THE COLOR TABLE BACK
       LI   R0,>800      POINT AT GRAPHICS CHAR TABLE
       LI   R1,CHRTBL    AND AT STORED CHARACTER DATA
       LI   R2,256*8     256 CHARACTERS
       BLWP @VMBW        WRITE CHARACTER DEFS BACK
       LI   R2,768       768 BYTES
       LI   R1,>2000     SPACE CHAR
       CLR  R0           ZERO IN R0
       BLWP @VWTR        CANCEL BITMAP
       MOVB R0,@>837A    NO SPRITE MOTION
CLSLOP BLWP @VSBW        WRITE A SPACE
       INC  R0           NEXT ADDR
       DEC  R2           DEC COUNT
       JNE  CLSLOP       RPT IF NOT 0
       LI   R1,>D000     "DELETE" SPRITE #0
       BLWP @VSBW        BY VDP WRITE
       LI   R0,>1E0      GRAPHICS MODE
       BLWP @VWTR        UNBLANK SCREEN
       RT                RETURN
*
* Bresenham's Circle Algorithm
* in TI Assembly Language
* on entry, R8=Y POSITION OF CENTER
*           R7=X POSITION OF CENTER
*           R0=RADIUS
* WITH THANKS TO PHIL VAN NORDSTRAND
*
```

```
BCIRC  MOV  R11,R13      SAVE RETURN ADDR
       MOV  R8,@CY       SAVE CENTER Y
       MOV  R7,@CX       SAVE CENTER X
       MOV  R0,@RADIUS   SAVE RADIUS
       MOV  R0,@DELYP    MAKE INITIAL TOP DELY=RADIUS
       NEG  R0           R0=-R0
       MOV  R0,@DELYM    MAKE INITIAL BOTTOM DELY=-RADIUS
       CLR  @DELXP       INITIAL DELXPLUS = 0
       CLR  @DELXM       INITIAL DELXMINUS = 0
       SLA  R0,1         DOUBLE R0 (R0=-2*RADIUS)
       AI   R0,3         ADD 3
       MOV  R0,@PARAM    INITIAL PARAM = 3 - 2*RADIUS
*
* FIRST LOOP DOES QUARTER CIRCLES AT THE
* TOP AND BOTTOM
*
PLACE  C    @DELXP,@DELYP CHECK DELTA X VS DELTA Y
       JGT  HALF2         IF GREATER, WE'RE DONE
       MOV  @DELYP,R8     GET TOP DELTA Y
       A    @CY,R8        ADD CENTER Y COORDINATE
       MOV  @DELXP,R7     GET POS DELTA X
       A    @CX,R7        ADD CENTER X COORDINATE
       BL   @CPLM         PLOT ONE POINT
       MOV  @DELXM,R7     GET NEGATIVE DELTA X
       A    @CX,R7        ADD CENTER X
       BL   @CPLM         PLOT LEFT POINT
       MOV  @DELYM,R8     GET BOTTOM DELTA Y
       A    @CY,R8        ADD CENTER Y
       BL   @CPLM         PLOT BOTTOM LEFT
       MOV  @DELXP,R7     GET POS DELTA X
       A    @CX,R7        ADD CENTER X
       BL   @CPLM         PLOT BOTTOM RIGHT
       MOV  @PARAM,R0     GET CURRENT PARAMETER
       JLT  ADJP          IF LESS THAN ZERO, JUMP
       MOV  @DELXP,R3     GET POS DELTA X
       S    @DELYP,R3     SUBTRACT POS DELTA Y
       SLA  R3,2          MULTIPLY RESULT BY 4
       A    R3,R0         ADD TO PREVIOUS PARAM
       AI   R0,10         THEN ADD 10
       MOV  R0,@PARAM     REPLACE PARAMETER WITH P+4(DELX-DELY)+10
       DEC  @DELYP        DEC POS DELY
       INC  @DELYM        INC NEG DELY
       JMP  ADDX          THEN JUMP
*
* CASE FOR PARAM <0
*
ADJP   MOV  @DELXP,R3     GET POS DELX
       SLA  R3,2          MULTIPLY BY 4
       A    R3,R0         ADD TO PARAM
       AI   R0,6          ADD 6 TO RESULT
```

```
       MOV  R0,@PARAM    REPLACE PARAM WITH P+4*DELX+6
ADDX   INC  @DELXP       INC POS DELX
       DEC  @DELXM       DEC NEG DELX
       JMP  PLACE        DO ANOTHER SET OF POINTS
*
* HALF2 DOES THE QUARTER CIRCLES ON THE SIDES
* BY REPEATING THE ALGORITHM WITH X AND Y INTERCHANGED
*

HALF2  MOV  @RADIUS,R0   GET BACK RADIUS
       MOV  R0,@DELXP    MAKE INITIAL POS DELX=RADIUS
       NEG  R0           R0=-R0
       MOV  R0,@DELXM    MAKE INITIAL NEG DELX=-RADIUS
       CLR  @DELYP       INITIAL DELYPLUS = 0
       CLR  @DELYM       INITIAL DELYMINUS = 0
       SLA  R0,1         DOUBLE R0 (R0=-2*RADIUS)
       AI   R0,3         ADD 3
       MOV  R0,@PARAM    INITIAL PARAM = 3 - 2*RADIUS
PLACE2 C    @DELYP,@DELXP CHECK DELTA Y VS DELTA X
       JGT  CIRCX        IF GREATER, WE'RE DONE
       MOV  @DELYP,R8    GET POS DELTA Y
       A    @CY,R8       ADD CENTER Y COORDINATE
       MOV  @DELXP,R7    GET POS DELTA X
       A    @CX,R7       ADD CENTER X COORDINATE
       BL   @CPLM        PLOT UPPER RIGHT
       MOV  @DELYM,R8    GET NEGATIVE DELTA Y
       A    @CY,R8       ADD CENTER Y
       BL   @CPLM        PLOT LOWER RIGHT
       MOV  @DELXM,R7    GET LEFT DELTA X
       A    @CX,R7       ADD CENTER X
       BL   @CPLM        PLOT LOWER LEFT
       MOV  @DELYP,R8    GET POS DELTA Y
       A    @CY,R8       ADD CENTER Y
       BL   @CPLM        PLOT UPPER LEFT
       MOV  @PARAM,R0    GET CURRENT PARAMETER
       JLT  ADJP2        IF LESS THAN ZERO, JUMP
       MOV  @DELYP,R3    GET POS DELTA Y
       S    @DELXP,R3    SUBTRACT POS DELTA X
       SLA  R3,2         MULTIPLY RESULT BY 4
       A    R3,R0        ADD TO PREVIOUS PARAM
       AI   R0,10        THEN ADD 10
       MOV  R0,@PARAM    REPLACE PARAMETER WITH P+4(DELY-DELX)+10
       DEC  @DELXP       DEC POS DELX
       INC  @DELXM       INC NEG DELX
       JMP  ADDY         THEN JUMP
*
* CASE FOR PARAM <0
*
ADJP2  MOV  @DELYP,R3    GET POS DELY
       SLA  R3,2         MULTIPLY BY 4
```

```
       A    R3,R0        ADD TO PARAM
       AI   R0,6         ADD 6 TO RESULT
       MOV  R0,@PARAM    REPLACE PARAM WITH P+4*DELY+6
ADDY   INC  @DELYP       INC POS DELY
       DEC  @DELYM       DEC NEG DELY
       JMP  PLACE2       DO ANOTHER SET OF POINTS
CIRCX  B    *R13         RETURN TO CALLER
*
* FOLLOWING CHECKS SCREEN LIMITS BEFORE
* ALLOWING A POINT TO BE PLOTTED ON SCREEN
*
CPLM   MOV  R7,R7        CHECK COL FOR <0
       JLT  NPLEX        IF <0, JUMP
       MOV  R8,R8        CHECK ROW FOR <0
       JLT  NPLEX        IF <0, JUMP
       CI   R7,255       CHECK UPPER COL LIMIT
       JGT  NPLEX        IF GREATER, JUMP
       CI   R8,191       CHECK UPPER ROW LIMIT
       JGT  NPLEX        IF GREATER, JUMP
       JMP  PLOT         ELSE PLOT THE POINT
NPLEX  RT                RETURN
*
* FOLLOWING WRITES ONE PIXEL TO SCREEN AT LOCATION POINTED BY
* R8 (DOT ROW) AND R7 (DOT COLUMN)
*
PLOT   MOV  R7,R3        MOVE DOT COLUMN TO R3
       MOV  R8,R4        AND DOT ROW TO R4
       MOV  R4,R5        DOT ROW ALSO IN R5
       ANDI R5,7         R5 HAS DOT ROW MODULO 8
       SZC  R5,R4        SO DOES R4
       SLA  R4,5         MULTIPLY R4 BY 32
       A    R5,R4        ADD R5, SO R4 HAS DR MOD. 8 * 32 + DR MOD 8
       MOV  R3,R0        MOVE DOT COL TO R0
       ANDI R0,>FFF8     R0 HAS DC - DC MOD 8
       S    R0,R3        R3 HAS DC MOD 8
       A    R4,R0        ADD R4
       SWPB R0           SWAP BYTES
       MOVB R0,@>8C02    WRITE LOW ADDRESS BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH ADDRESS BYTE
       NOP               WASTE TIME
       MOVB @>8800,R1    READ THE BYTE
       SOCB @M(R3),R1    OVERLAY MASK FROM TABLE M
       ORI  R0,>4000     SET THE 4000 BIT IN R0
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE LOW BYTE OF ADDRESS
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH BYTE OF ADDRESS
       NOP               WASTE TIME
       MOVB R1,@>8C00    WRITE MODIFIED BYTE BACK TO VDP
```

```
       MOV  R9,R9        IS COLOR TO BE SET?
       JEQ  PLOTX        IF NOT, JUMP AHEAD
       ANDI R0,>3FFF     STRIP OFF "4" FROM R0
       AI   R0,>2000     ADD >2000 TO POINT AT COLOR TABLE ENTRY
       BLWP @VSBR        READ THAT BYTE INTO R1
       MOVB R1,R2        MOVE THE BYTE TO R2
       ANDI R2,>F000     STRIP ALL BUT LEFT NYBBLE
       CB   R2,R9        COMPARE TO LEFT BYTE R9
       JEQ  PLOTX        IF EQUAL, COLOR ALREADY SET
       ANDI R1,>0F00     ELSE STRIP OFF LEFT NYBBLE R1
       AB   R9,R1        REPLACE WITH LEFT NYBBLE R9
       BLWP @VSBW        THEN WRITE COLOR BYTE BACK
PLOTX  RT                RETURN
*
*
* DATA SECTION
*
WS     BSS  >20          OUR WORKSPACE
M      DATA >8040,>2010,>0804,>0201  MASK DATA
CX     DATA 0            CENTER X POSITION
CY     DATA 0            CENTER Y POSITION
RADIUS DATA 0            CIRCLE RADIUS
DELXP  DATA 0            POSITIVE DELTA X
DELYP  DATA 0            POSITIVE DELTA Y
DELYM  DATA 0            NEGATIVE DELTA Y
DELXM  DATA 0            NEGATIVE DELTA X
PARAM  DATA 0            PARAMETER
SAVR0  DATA 0            STORAGE FOR R0
SAVR7  DATA 0            STORAGE FOR R7
SAVR8  DATA 0            STORAGE FOR R8
ANYKEY BYTE >20          COMPARISON BYTE FOR KEYSTROKE
       BSS  6            STORAGE FOR DSR DATA FROM VDP RAM
SAVCLR BSS  32           STORAGE FOR GRAPHICS COLOR TABLE
CHRTBL BSS  256*8        STORAGE FOR GRAPHICS CHARACTER DEFINITIONS
       END

* FOLLOWING IS AN XB PROGRAM TO
* ILLUSTRATE THE CIRCLE ALGORITHM
* (LISTED IN 28 COLUMNS)

10 CALL CLEAR :: INPUT "RADI
US ":RADIUS :: RADIUS=INT(RA
DIUS):: IF RADIUS<0 OR RADIU
S>11 THEN 10
20 CALL CLEAR :: CX=14 :: CY
=12
30 P=3-2*RADIUS :: DELYP=RAD
IUS :: DELYM=-RADIUS :: DELX
P,DELXM=0
40 IF DELXP>DELYP THEN 100
```

```
50 DISPLAY AT(CY+DELYP,CX+DE
LXP):CHR$(30);:: DISPLAY AT(
CY+DELYP,CX+DELXM):CHR$(30);
60 DISPLAY AT(CY+DELYM,CX+DE
LXP):CHR$(30);:: DISPLAY AT(
CY+DELYM,CX+DELXM):CHR$(30);
70 IF P<0 THEN P=P+(4*DELXP)
+6 :: GOTO 90
80 P=P+(4*(DELXP-DELYP))+10
:: DELYP=DELYP-1 :: DELYM=DE
LYM+1
90 DELXP=DELXP+1 :: DELXM=DE
LXM-1 :: GOTO 40
100 P=3-2*RADIUS :: DELXP=RA
DIUS :: DELXM=-RADIUS :: DEL
YP,DELYM=0
110 IF DELYP>DELXP THEN 170
120 DISPLAY AT(CY+DELYP,CX+D
ELXP):CHR$(30);:: DISPLAY AT
(CY+DELYP,CX+DELXM):CHR$(30)
;
130 DISPLAY AT(CY+DELYM,CX+D
ELXP):CHR$(30);:: DISPLAY AT
(CY+DELYM,CX+DELXM):CHR$(30)
;
140 IF P<0 THEN P=P+(4*DELYP
)+6 :: GOTO 160
150 P=P+(4*(DELYP-DELXP))+10
:: DELXP=DELXP-1 :: DELXM=DE
LXM+1
160 DELYP=DELYP+1 :: DELYM=D
ELYM-1 :: GOTO 110
170 DISPLAY AT(24,7):"PRESS
R TO REPEAT";
180 CALL KEY(0,K,S):: IF S<1
 THEN 180 ELSE IF K=82 OR K=
114 THEN 10
```

1.67. The Art Of Assembly — Part 67. Sounds Good To Me!

By Bruce Harrison

It's been a while since we've done anything about the TI sound chip. Once again, however, we've been inspired by other programmers. Vern Jensen asked about whether the sound list method that's available to Assembly programmers could be made to work from C99 programs. Yes, it can, and in creating some routines for use with C99, we also added new tricks for the Assembly programmer, and even an easier method for preparing the sound lists to use with these routines.

1.67.1. Continuous Background

The first thing we wanted to do was to make it possible to have a "theme song" play continuously in the computer's interrupt cycle, while allowing the main program to continue "doing its thing". Since no sound list can last forever, we set up a User Interrupt to sense when the tune is finished and start it over again. All that the main program needs to do is cycle through LIMI 2 and LIMI 0 now and then, so that sound processing can be done through the code in ROM and the User Interrupt.

1.67.2. We Interrupt This Interrupt

Okay, once we had the theme playing all the time and the program able to do other stuff as the theme played on, the thought came to us: "What about sound effects?" The idea then was to have an effect sound list interrupt the theme for a short time, then have the theme pick up where it left off. This idea was aimed at games, where some event in the game may need to start a sound effect and perhaps visual effects as well, but then the theme should pick up at its next note. This led to another User Interrupt that would "time out" the sound effect and then turn control back to the theme's User Interrupt with the theme then resuming at whatever was the next note in its list.

1.67.3. Two Other Things

There are two other ways to run the sound list. One should be able to just run a sound list one time through, then have it stop, but have the main program continue unhindered. Finally, one should have a way to let the program be delayed until a sound list finishes. We've put those two capabilities in as well.

1.67.4. The Bouncing Ball

No, we didn't really make a ball bounce, but we provided a "pacing" capability so that, while a sound list is playing, we can let the main program check on its progress. Thus we can have the main program perform actions that are "paced" by the notes in the music, yet still be able to perform other actions, like keyboard scans, during the music. While we were at that, we added a "quick check" so that if the music is on a one-time playing, the main program can determine whether it's finished that play or not.

1.67.5. Loading The Lists

To make the placing of sound lists into VDP RAM quick and easy, we created a routine that will load up to ten sound lists at one whack, and had this routine stash away the VDP locations of each in a lookup table. This way, the sound lists can be activated selectively whenever needed. The lists themselves can be either within the main program's object file or in separate object files with REFs and DEFs linking them up at load time. The calling of this routine is different for C99 and Assembly, even though the internal process is similar. Let's say we want to load three lists called CHIME, CRASH, and VIV. From C99, we'd do this:

```
setsou(0,chime,crash,viv);
```

From Assembly, that would be:

```
BLWP @SETSOU
DATA 3,CHIME,CRASH,VIV
```

In either case, the lists would be loaded and referenced in the order given, so that activation would be by number, with 1 being CHIME, 2 being CRASH, and 3 being VIV. In the C99 case, the routine works backwards through the C99 stack until it finds that 0, which is how it "knows" how many lists to load. In the Assembly case, it just takes the first DATA after the BLWP as the number of lists to load. In either case, it's important that each list itself ends with a "note" of zero duration. Our usual practice is to make the last note in a list be a "silence" for all four generators. Thus the last line in our source list would look like this:

```
BYTE 4,>9F,>BF,>DF,>FF,0
```

That zero duration is important anyway because it signals the Interrupt Handler in ROM when a sound list ends. Our loading routine uses that 0 duration to determine the size of the list, so it "knows" how many bytes to dump into VDP RAM. The loading routine does a very quick scanning of each list, stopping when it finds 0 duration, then dumps the list into VDP by a VMBW operation. The lists are actually loaded backwards in sequence, with the last byte of the first list being at the address pointed to by >8370 when the routine is called. Each successive list is then loaded at lower addresses, until all the lists are in place. As it loads each list, the routine updates the word at >8370, and puts each list's VDP address into a lookup table in its own memory space. Later, the activation routine uses that lookup table to tell the sound processor where the selected list starts.

1.67.6. Activating Sound Lists

When the lists get loaded, no sound starts. The actual playing of the lists is done by activating them. Activation requires two parameters, the first being the number of the particular list from the loading process, and the second being the type of play for that list. The number is just 1 through n, where 1 is the first one named in the loading process and n the last. The activation checks the number against the number actually loaded, and will do nothing if you're asking to play number 4 when you've only loaded 3. It also will reject 0 or any negative number.

We mentioned earlier that there are four ways to use a list. Here are the four "types" of play:

Type 1 — Play continuously with repeats when done.
Type 2 — Play once through on background, then stop.
Type 3 — Interrupt playing of a type 1 for one play.
Type 4 — Play once but delay main program.

In any of these cases, the call to the activation routine gets two parameters. From Assembly:

```
BLWP @STSOU
DATA NUMBER,TYPE
```

from C99:

```
stsou(number,type);
```

Let's suppose we're using number 3 sound list as a "theme song" that will run all the time in a game program. We'd start that in the beginning of the game by BLWP @STSOU with DATA 3,1. It would start playing, repeating itself when it finished, and so continuing "forever" in the background. Let's then say we reached a point in the game where some event (like a sprite coincidence) occurred, and a crash sound effect was needed. We'd just BLWP @STSOU with DATA 2,3. The crash would stop the theme, play itself in the background, then the theme would resume at its next note.

One very important "service" is provided in the routines, called ENDSND. This not only stops any sounds that were playing, but restores the original state of >8370 and in effect removes the lists loaded into VDP RAM. It's important that your programs use BLWP @ENDSND before exiting.

There's also an emergency stop, which will stop any sound that's playing, but won't affect the lists like ENDSND. This is called STPSND, and is invoked by just BLWP @STPSND, with no parameters. This stops the current sound, but does not affect the lists in VDP RAM, nor does it restore the original value to >8370.

1.67.7. Making Sound Lists

This has been a terrible chore. If you're doing anything that involves musical notes instead of just noises, it's a bear! Not only do you have to have some knowledge of music, but you have to translate music into hex code. Many of us learned in Grade School that the staff with a G clef could be read by remembering two simple mnemonics: The lines starting at the bottom are E, G, B, D, and F. These can be remembered as the initials of Every Good Boy Does Fine. The spaces are F, A, C, and E. These can be remembered as spelling the word FACE. (Please don't ask about leger lines.) Given this, your author can plow through a piece of music, reading one note at a time. The hex part has to be looked up in that great book, the E/A Manual, also one note at a time. Now let's take a trip down memory lane. About ten years ago, your author devised a method whereby his partner Dolores P. Werths, who's a musician, could translate music from the printed staves into a quasi-musical notation in source files, and then by using auxiliary files of EQUates, the Assembler would generate the hex. That technique, in modified form, has been re-invented for the rest of you, so that you can write your Sound List source files in musical terms, then have the Assembler convert your musical instructions into hex codes for the sound chip. Just for example, suppose you wanted Generator 1 to play a C of the fourth octave, at volume level 3, for a duration of one quarter note. You'd put this into the source file:

```
BYTE 3,G1+C4A,C4B,G1V+3,Q
```

Notice that there are two pieces of the note C4, a part called C4A, which gets added to the generator number (G1), and a second part, which is a byte of its own as C4B. To indicate a volume, you tack a V onto G1, to get G1V, then add the level desired, in this case 3. The duration is just a single letter Q for Quarter note. The 3 after the word BYTE is just to tell the ROM software that what follows are three bytes to be sent to the sound generator. That is, G1+C4A, one byte, C4B, a second, and G1V+3, a third. The Q is one byte, too, but that byte doesn't go to the sound generator. The ROM software uses that byte to count the duration of this note in 60ths of a second. To get the Assembler to translate what you see above into hex, we use two "COPY" files that are nothing but EQUates. The note frequency and volume equivalents are in NOTES, and the durations in TEMPO.

1.67.8. How Do I Get This Stuff?

Easy as a $2.00 apple pie. (Where in this world could you get an apple pie for $2.00?) There are three disks in the set, of which you'll only need two. For either C99 or Assembly programmers, you get the disk LISTMAKER. Then you add to that either the CSOUND disk or the ASMSOUND disk as appropriate to your choice of programming language. You send $1.00 per disk to the Lima Users' Group, c/o Dr. Charles Good, P.O. Box 647, Venedocia, OH, 45894. Each of those disks contains complete instructions, demo programs written in the language of your choice, all the source files, object modules, and even some ready-made sound lists that you may find useful and amusing. We guarantee these will last longer than a $2.00 apple pie, too.

1.67.9. Today's Sidebar

In the Sidebar is the source code for the Assembly version of the sound list loading and activation subroutines. We've annotated this fairly completely, so at least our Assembly programmers should be able to follow it. If you get *MICROpendium* on disk, I've included in the mailing to John and Laura the two demo programs DEMO and DEMO2 in E/A Option 5 form, and the Sidebar's source code as an object module called SIDE67/O, for inclusion on the MICRO - On - Disk. The code as shown in the Sidebar is complete, and will Assemble into an object module, ready for use. The programs DEMO and DEMO2 are the demos from our ASMSOUND disk. There are similar demo programs on the CSOUND disk, with the names SOUND and PACED.

Next month we're going back into the Bitmap, but we'll be dealing with TI-Artist Instances.

```
*  SIDEBAR67
*
*  A COMPLETE MODULE
*
*  PUBLIC DOMAIN SOFTWARE
*  SOUND LIST PROCESSING
*  USER MAY LOAD UP TO 10 SOUND LISTS
*  AND SELECTIVELY ACTIVATE THEM
*
*  by Bruce Harrison
*  01 SEP 1995
       DEF  SETSOU,STSOU,ENDSND,STPSND
       DEF  PACEIT,FINEPC,SETPAC,TESTSO
       REF  VMBW,VSBR
*
* SETSOU PLACES THE SOUND LISTS IN VDP RAM
* FOR ACTIVATION WHEN DESIRED
* INVOKE SETSOU BY:
* BLWP @SETSOU
* DATA (NUMBER OF LISTS TO FOLLOW)
* DATA LIST1,LIST2,LIST3,...LIST 10
*
SETSOU DATA SWS,SETSO
SETSO  MOV  *R14+,R8     NUMBER OF LISTS TO R8
       MOV  R8,@NOLST    PLACE AT NOLST
       MOV  @>8370,@OLD70 SAVE ADDRESS FROM >8370
       LI   R7,LISTBL    POINT AT TABLE OF ADDRESSES
STSOU0 MOV  *R14+,R4     GET NEXT CPU LIST ADDRESS
       MOV  R4,R1        ADDRESS IN R1
       CLR  R2           R2=0
SETSO2 MOVB *R4+,R6      LENGTH BYTE IN R6
       SRL  R6,8         MAKE A WORD
       A    R6,R2        ADD LENGTH TO R2
       INCT R2           ADD TWO FOR DURATION & LENGTH BYTES
       A    R6,R4        ADD LENGTH TO ADDRESS
       MOVB *R4+,R5      GET DURATION BYTE
```

```
       JNE  SETSO2       IF NOT ZERO, REPEAT
* IF THE DURATION BYTE IS NOT 0, WE'RE
* NOT AT THE END OF THE SOUND LIST, SO WE
* KEEP ON GOING.  R4 POINTS TO LENGTH BYTE OF NEXT GROUP
       MOV  @>8370,R0    LAST AVAIL VDP ADDR TO R0
       S    R2,R0        SUBTRACT LENGTH OF SOUND LIST
       BLWP @VMBW        WRITE LIST TO VDP
       MOV  R0,*R7+      PUT VDP ADDRESS IN TABLE
       DEC  R0           ONE BYTE BACK
       MOV  R0,@>8370    NEW LAST AVAILABLE TO >8370
       DEC  R8           DEC COUNT OF LISTS
       JGT  STSOU0       IF > ZERO, DO ANOTHER
SETEX  RTWP              ELSE RETURN TO CALLER
*
* STSOU  STARTS A SOUND THAT WILL CONTINUE OR NOT
*  INVOKE BY :
*  BLWP @STSOU
*  DATA LIST#,TYPE
*  list # = number of list (1-10)
*  type = 1,2,3, or 4
*             1 means continue indefinitely, with repeat
*             2 means play once, then stop sound
*             3 means play and then continue previous list
*             4 means play and suspend other action until list finishes
STSOU  DATA SWS,STSO
STSO   MOV  *R14+,R4     LIST NUMBER TO R4
       MOV  *R14+,R7     TYPE OF PLAY TO R7
       CI   R7,1         COMPARE TYPE TO 1
       JLT  SETEX        IF LESS, EXIT
       CI   R7,4         COMPARE TO 4
       JGT  SETEX        IF >, EXIT
       C    R4,@NOLST    CHECK NUMBER OF LISTS LOADED
       JGT  SETEX        IF R4 >, EXIT
       DEC  R4           ZERO BASE
       JLT  SETEX        IF <0, EXIT
       SLA  R4,1         DOUBLE NUMBER IN R4
       LIMI 0            STOP INTERRUPTS
       CI   R7,3         TYPE 3?
       JNE  NOT3         JUMP IF NOT
       MOV  @>83CC,@SAVCON ELSE SAVE PRESENT PLACE
NOT3   MOV  @LISTBL(R4),R0 GET THE LIST'S ADDRESS
       MOV  R0,@>83CC    MOVE ADDRESS TO >83CC
       SOCB @H01,@>83FD  SET VDP FLAG
       MOVB @H01,@>83CE  TRIGGER SOUND PROCESSING
       CI   R7,4         TYPE 4?
       JEQ  TYPE4        IF SO, JUMP
       CI   R7,1         TYPE 1?
       JEQ  STCNT        IF SO, JUMP
       CI   R7,3         TYPE 3?
       JEQ  SET3         IF SO, JUMP
```

```
CLRALL CLR  @>83C4       NO USER INTERRUPT
       CLR  @SAVCON      CLEAR SAVCON
       CLR  @RESADR      CLEAR RESET ADDRESS
       JMP  SETEX        THEN EXIT
SET3   MOV  @CK2ADR,@>83C4 IF TYPE 3, DIFFERENT INTERRUPT
       JMP  SETEX        THEN EXIT
STCNT  MOV  @CKADR,@>83C4  PUT INTERRUPT'S ADDRESS IN PLACE
       MOV  R0,@RESADR   SET FOR REPEATS
       JMP  SETEX        RETURN TO C99
TYPE4  CLR  @>83C4       CLEAR USER INTERRUPT
       CLR  @RESADR      CLEAR RESET ADDR
       CLR  @SAVCON      CLEAR CONTINUE ADDRESS
T4R    LIMI 2            ALLOW INTS
       LIMI 0            STOP THEM
       MOVB @>83CE,R1    CHECK COUNT BYTE
       JNE  T4R          IF NOT 0, REPEAT
       JMP  SETEX
*
* ENDSND MUST BE USED JUST BEFORE
* EXITING YOUR PROGRAM
* INVOKE ENDSND BY:
*   BLWP @ENDSND
*   stops all sound generation
*   (unconditional)
*   cancels all lists loaded
*
ENDSND DATA SWS,ENDSN
ENDSN  LI   R1,SNDOFF    SILENCE SOUND STRING
       MOVB *R1+,R2      LENGTH BYTE TO R4
       SRL  R2,8         RIGHT JUSTIFY
MOVLP  MOVB *R1+,@>8400  ONE BYTE TO SOUND CHIP
       DEC  R2           DEC COUNT
       JNE  MOVLP        IF NOT ZERO, REPEAT
       MOVB @ZERO,@>83CE CLEAR THE SOUND COUNT
       SZCB @HFE,@>83FD  AND THE SOUND FLAG
       MOV  @OLD70,@>8370 OLD ADDRESS BACK IN >8370
       CLR  @NOLST       CLEAR NUMBER OF LISTS
       JMP  CLRALL       THEN CLEAN UP
*
* STPSND SIMPLY STOPS ANY SOUND LIST INSTANTLY
* INVOKE BY:
*      BLWP @STPSND
*
STPSND DATA SWS,STOP
STOP   LI   R1,SNDOFF+1  SILENCE BYTES
       LI   R2,4         FOUR TO SEND
SILEN  MOVB *R1+,@>8400  SILENCE ONE
       DEC  R2           DEC COUNT
       JNE  SILEN        RPT IF NOT 0
       MOVB @ZERO,@>83CE ZERO COUNT
```

```
       SZCB @HFE,@>83FD  RESET VDP FLAG BIT
       CLR  @>83C4       CLEAR USER INTERRUPT
       RTWP              RETURN
*
* CKS2 AND CKSND ARE USER INTERRUPTS
* FOR HANDLING CONTINUING SOUND LISTS
*
CKS2   MOVB @>83CE,@>83CE SOUND STILL PLAYING?
       JNE  QRET          IF SO, EXIT
       MOV  @RESADR,@RESADR ANY PREVIOUS LIST IN PLAY?
       JEQ  QRET          IF NOT, EXIT
       MOV  @SAVCON,@>83CC OLD LIST POSITION BACK
       MOV  @CKADR,@>83C4 SET OTHER USER INTERRUPT
       JMP  REST2         THEN JUMP AHEAD
CKSND  MOVB @>83CE,@>83CE  CHECK THE SOUND PROCESSING BYTE
       JNE  QRET         IF NOT ZERO, EXIT
RESET  MOV  @RESADR,@>83CC   ELSE RE-SET VDP ADDRESS
       JEQ  QRET         IF NO RESET, STOP
REST2  SOCB @H01,@>83FD  SET VDP FLAG
       MOVB @H01,@>83CE  TRIGGER SOUND PROCESSING
QRET   RT                RETURN TO INTERRUPT HANDLER
* PACEIT
* PACES BY A RUNNING SOUND LIST
* CAN BE USED TO TIME NOTE BY NOTE
* OR BY 1/60 SECOND INTERVALS
* PACEIT MEASURES BY NOTES
* FINEPC MEASURES BY 1/60THS
* SETPAC INITIALIZES STATES
*
* INVOKE PACEIT BY:
* BLWP @PACEIT
* on exit STATUS EQ IF SAME NOTE
*         STATUS NE IF NEW NOTE
*
PACEIT DATA SWS,PACE
PACE   MOVB @>83CE,R1    ANYTHING IN >83CE
       JEQ  PACEX        IF ZERO, NO SOUNDS PLAYING, EXIT
       C    @>83CC,@OLDCC COMPARE POINTER TO OLD VALUE
       JEQ  PACEX        IF EQUAL, SAME NOTE STILL PLAYING
       MOV  @>83CC,@OLDCC ELSE NEW NOTE STARTED
PACEX  STST R15          STATUS REGISTER TO R15
       RTWP              THEN RETURN
* FINEPC REPORTS WHETHER A 1/60 HAS PASSED
* SINCE LAST CALL
* INVOKE FINEPC BY:
* BLWP @FINEPC
* on exit STATUS EQ IF LESS THAN 1/60th ELASPSED
*         STATUS NE IF 1/60TH HAS PASSED
*
FINEPC DATA SWS,FINP
```

```
FINP   CB   @>83CE,@OLDCE SAME 1/60TH INTERVAL?
       JEQ  PACEX        IF SO, EXIT
       MOVB @>83CE,@OLDCE ELSE REPLACE SAVED VALUE
       JMP  PACEX
* SETPAC STARTS THE PACING PROCESS
* INVOKE BY:
* BLWP @SETPAC
* DO THIS IMMEDIATELY AFTER STARTING SOUND LIST
*
SETPAC DATA SWS,SETPC
SETPC  MOV  @>83CC,@OLDCC GET EXISTING SOUND POINTER
       MOVB @>83CE,@OLDCC AND 60THS COUNT
       JMP  PACEX         THEN EXIT
*
*  TESTSO REPORTS WHETHER A SOUND LIST IS
*  CURRENTLY RUNNING
*  INVOKE BY:
*  BLWP @TESTSO
*  on return, STATUS EQ IF NO SOUND LIST RUNNING
*             STATUS NE IF SOUND LIST RUNNING
*
TESTSO DATA SWS,TESTS
TESTS  MOVB @>83CE,R1    MOVE THE COUNT BYTE
       JMP  PACEX        THEN JUMP
*
* DATA SECTION
*
OLDCE  BYTE 0            STORAGE FOR 60THS COUNT
OLDCC  DATA 0            STORAGE FOR POINTER
SWS    BSS  32           SOUND WORKSPACE
CK2ADR DATA CKS2         ADDRESS OF SPECIAL INTERRUPT
CKADR  DATA CKSND        ADDRESS OF OUR INTERRUPT
ZERO   BYTE 0            ZERO BYTE
H01    BYTE >01          BYTE OF 1
HFE    BYTE >FE          BYTE OF -2
NOLST  DATA 0            NUMBER OF LISTS LOADED
RESADR DATA 0            ADDR FOR CONTINUOUS SOUND PLAY
SAVCON DATA 0            CONTINUE SOUND ADDR
LISTBL DATA 0,0,0,0,0,0,0,0,0,0  TABLE OF SOUND LIST ADDRESSES
SNDOFF BYTE >04,>9F,>BF,>DF,>FF
OLD70  DATA 0
       END
```

1.68. The Art Of Assembly — Part 68. The Ins And Outs Of Instances

By Bruce Harrison

Many moons ago, we generated some programs to deal with the problems of drawing Bitmap images and printing them. These products include our own Drawing Program, and our TIAPRINT, for printing TI-Artist picture files. Both of those are now available in versions for both 9-pin and 24-pin printers. As happens, we had some correspondence from our good friend Charles Kirkwood, Jr. He ran into the need to deal with printing TI-Artist Instances. At first, our reaction was not very positive. True, we'd made a provision in our Drawing Program to bring in TI-Artist Instance files and incorporate them into drawings, but otherwise hadn't done anything with them. What Charles was seeking was a way to use Instances as a "letterhead" for his correspondence. That seemed a reasonable thing to do, so when some free time was available, we sat down at our faithful TI and started writing source code.

1.68.1. The Formatter Problem

Others in our "community" have devised methods for printing Instances as part of a TI-Writer document, through the Formatter. This creates problems we don't need, mainly because the Formatter uses "printable" characters as control codes. Thus if the binary content of a byte in the Instance happens to equal the "@" symbol, the formatter will translate that into an escape sequence intended to put the printer into double strike mode. That's definitely not a good thing to have happen in the middle of a picture. Also, the Formatter cannot gracefully handle strings of characters longer than 80, and we thought that this too would be a "killing" limitation.

1.68.2. The Standalone Solution

Our idea, then, was to keep the Formatter "out of the loop" when printing an Instance as a letterhead. We devised a method that takes an Instance file, automatically centers it on the paper, re-maps the bits into the correct bit graphics form, and prints it as single density graphics. Using this simplest approach meant that, among other things, our program would be compatible with a broad range of 9-pin Dot Matrix printers. That includes most Epson, Star Micronics, and Panasonic models in use by our TI "community". Since our standalone program opens the file to the printer as a D/V 254 file, and with the .CR option, we have no problem with sending the bytes we need without suffering any unwanted carriage returns or line feeds. Thus one can take any instance file and send it to the printer without fear.

1.68.3. The Drawbacks

There is one major drawback. After printing the Instance, you must turn off the printer and roll the paper back to the start of the sheet so that whatever is printing your text won't get confused about where pages begin and end. Also, of course, the text file must begin with some number of blank lines or carriage returns so that the letterhead won't get overprinted by the text. We felt, though, that these two things would be fairly easy to do, so we've gone ahead and offered the product called PRNINST to the TI "public" as Public Domain software. The disk also includes versions for double density bit graphics and for 24-pin printers, as PRNINST2 and PRNINST24.

1.68.4. But What About Editing?

We've never invested in a copy of TI-Artist. Most of the time we don't need it anyway, so why bother? Then we started playing around with Instances. Back some time ago, Dr. Charles Good sent us about six disks full of very nice Instances from Lima's Public Domain library. We started printing these out as part of the testing for our PRNINST product. In doing so, we discovered that some of these had stray "artifacts" in them, and that they needed some editing to correct these small flaws. Without TI-Artist, we had no way of visually editing these Instance files. In some cases, the flaws were in obvious enough places that we could edit them using the E/A Editor, but those were the exceptions. What we needed was some method to see what we were doing to an instance directly on the screen, then save it back to disk in Instance (D/V 80) format.

1.68.5. Back To The Drawing Board

Or, rather, the source code from our Drawing program. We figured that with a few changes here and there, we could adapt the Drawing program into an editor for Instance files. It took a few days and nights of hard work, but we were able to produce an easy to use editor that would take care of instances so long as they were 22 characters or less in height, and 32 characters or less in width. Since most instances are in that category, we feel our effort was worthwhile. Besides allowing the direct input of Instance files, we kept the ability to load drawings in either our own format or in the TI-Artist Picture file format. That way, we could extract some neat part of an existing picture into an Instance that could then be used by itself or incorporated into some other picture.

In testing, we of course found many small bugs, and have corrected them, so the INSTED product was made available through the usual channels. This program, like the Drawing program that spawned it, is very versatile. One can, for example, load in font files, either of the CHARA1 type or of the TI-Artist type, and use those to type things on the instances. We mentioned the ability to bring in pictures, and that turned out to be very useful. Some of Charles Kirkwood's pictures were converted "in toto" from picture files into instances. We can also use this program to make new instances from scratch.

1.68.6. The Saving Process

Saving Bitmapped imagery as instances is a non-trivial matter, as you might guess. In the first place, we don't want the saved Instance to become 22 rows high by 32 columns wide. Thus we need to scan the contents of the Bitmap pattern table to determine the required height and width of the instance. To do this quickly, we dump the entire pattern table, starting from Row 2 of the screen, and extending to row 23, into CPU memory. That's actually done when you come out of the editing mode, so that we can put the program's menu on the screen without losing the picture. Thus while we're looking at the menu, the pattern table is stashed away as >1600 bytes in the high memory. The save process scans through that memory, looking first from what would be the top of the picture, until it finds a byte with non-zero content. The row where a non-zero byte is found gets stashed away as the start row for saving. Now the program scans again, this time starting from the bottom of the stashed pattern table, working upwards until it finds a non-zero byte. This tells us the height of the instance.

This process of scanning is then repeated going sideways to find the leftmost column of image content, then its width. Now we have all the needed parameters of the image content to start actually saving the Instance file.

1.68.7. Structure Of An Instance

Except for the first record in the file, which contains the width and height of the instance in character rows and columns, the content of the file is in groups of eight numbers, so that each record contains the numbers for one character definition. These numbers are written into the file in human-readable form, so that the file is editable (in theory) with the E/A editor as with any D/V 80 file. Thus a record in the file is eight numbers, separated by commas. Such a record looks like this:

```
0,0,126,67,0,14,253,85
```

Since each number represents a byte, the range of these numbers goes from 0 through 255. Knowing this, we're ready to save our Instance to disk.

The first order of business is to write that first record, which contains the width and height of the instance to follow. To do that, we take the width first, convert it from a number to a string, and write that string into a buffer in VDP. For this, we use an "undocumented" GPLLNK feature, given to us by Merle Vogt. Since these are all single byte numbers, we clear the byte at >835E and put the byte to be converted at >835F. We use the Warren/Miller GPLLNK routine, followed by the DATA >2F7C. That undocumented feature provides us with the number in the form of a string, but without the leading space that's usually reserved for a plus sign in the "normal" convert number to string routine. That's a fortunate circumstance, since that leading space is neither desired nor required.

After the width, we write a comma to the VDP Buffer, then the string form of the height. Now we're ready to send the first (actually 0th) record to the file. Somewhere in here, of course, we had to open the file as D/V 80, for output. The first record gets written. Meanwhile, we've stashed away the product of height and width, as that tells us how many total records (after the 0th one) we need to write to this file.

Next, we go back into the stashed pattern table, and take the bytes starting at the top row, left column. These get translated to strings and then sent out in groups of eight, separated by commas as shown above. We continue this for WIDTH repeats, then move down one row from our start point and do another set of WIDTH character definitions. When we've done HEIGHT times WIDTH records in this fashion, we're done, so we close the file and return to the main menu. All of this takes time, particularly if we're writing to a floppy disk. Thus we give the user a "PLEASE WAIT PATIENTLY" message on the screen while saving the Instance.

1.68.8. A Small Caution

For reasons we've not been able to determine, some of the Instances we've gotten from Charles Good come with one or more blank rows at their beginning and/or ending. We don't know why they were made that way, as extra rows of all zeros add nothing to the instance, but just take up space on the disk. If you edit such an instance with our program, those blank rows will be eliminated from the saved instance.

1.68.9. Today's Sidebar

The Sidebar source code today is not a complete program, but just a section of the code from the double density bit graphics printing program PRNINST2. This section is the vital ingredient in converting an eight byte character pattern into sixteen bytes of Bit Graphics data for the printer. We thought this section, incomplete as it is, might prove helpful to those trying to work with this kind of transition. This works by shifting registers, and inserting a bit into the output byte only if there's a "carry" on the input byte. In other words, if the most significant bit of the input byte was a 1 before the shift, then a 1 gets placed in the least significant bit of the byte being prepared for output. It takes eight passes through all 8 bytes of one character pattern to prepare one 16 byte section of the output to the printer. Each such output section makes 16/120ths (or 8/60ths) of an inch character on the paper. We should point out that on the 9-pin printers the aspect ratio is incorrect, so circles on your instance will be slightly "ovaled" on the paper. They'll be a bit wider than their height. On 24-pin printers, the aspect ratio will be correct, since the special line feeds are in 60ths of an inch instead of in 72nds.

Just recently, we purchased a Canon Bubble Jet printer, and the PRNINST stuff works beautifully with that machine. Oddly, it makes no difference whether we use the single or double density versions, as the print quality is determined by the mode of the printer. In its HQ mode, either the single or double density graphics look superb, while in the printer's HS mode, the graphics look "thinned". For normal 9 or 24 pin printers, we'd recommend using the double density graphics (PRNINST2 or PRNINST24), as these produce much better looking renditions of the Instance.

Next month, among other things, we'll discuss some things outside the realm of Assembly programming. See you then.

```
* SIDEBAR 68
* RE-MAPPING FRAGMENT
* WE ENTER WHERE A RECORD HAS JUST BEEN READ
* THE STRINGS IN THE RECORD MUST BE CONVERTED TO
* NUMBERS, WHICH ARE STASHED IN INPBUF
*
RNREC1 LI   R0,INPPAB+5  LENGTH OF RECORD
       BLWP @VSBR        READ THAT
       MOVB R1,R2        MOVE TO R2
       SRL  R2,8         RT. JUST
       CLR  R1           R1=0
       LI   R9,INPBUF    POINT TO CPU INPUT BUFFER
       LI   R0,IBUFF     AND TO VDP INPUT BUFFER
       A    R2,R0        ADD LENGTH TO ADDR
       MOV  R0,R7        COPY ADDR TO R7
       BLWP @VSBW        WRITE A 0 AFTER RECORD
       S    R2,R0        BACK TO START OF RECORD
GNNUM  BL   @CONVN       GET A NUMBER
*
* SUBROUTINE CONVN USES XMLLNK TO CONVERT THE STRING IN
* THE VDP RAM BUFFER INTO A NUMBER AT FAC, THEN USES
* XMLLNK TO CONVERT FLOATING POINT TO AN
* INTEGER AT FAC.  WE TAKE ONLY THE LOW ORDER BYTE
*
       MOVB @FAC+1,*R9+  PUT INTO CPU INPUT BUFFER
SKCOM1 BLWP @VSBR        READ A BYTE
       INC  R0           POINT AHEAD ONE
       CB   R1,@COMMA    IS THAT A COMMA?
       JEQ  GNNUM        IF SO, JUMP BACK
       C    R0,R7        CHECK END OF RECORD
       JLT  SKCOM1       IF LESS, REPEAT
*
* WHEN WE GET HERE, INPBUF WILL CONTAIN THE
* EIGHT BYTES REPRESENTING THE NUMBER STRINGS
* IN ONE INPUT RECORD
*
* CORE PROCESSING SECTION
* RE-MAPPING OF ONE CHARACTER DEFINITION
* FOR BIT GRAPHICS PRINTING
* THIS TAKES THE 8 BYTES THAT DEFINE
* A CHARACTER AND RE-MAPS THEM INTO
* 8 PAIRS OF BIT-GRAPHICS BYTES FOR THE PRINTER
* ON EACH PASS THROUGH INNER LOOP, THIS
* TAKES THE MSB OF EACH OF THE 8 BYTES
* IN INPBUF, PUTS THAT INTO R1, WHICH GETS
* SHIFTED LEFT BY ONE BIT ON EACH PASS
* THE INPUT BYTE ALSO GETS SHIFTED LEFT ONE BYTE,
* THEN PUT BACK IN INPBUF.
* WHEN INNER FINISHES, R1 HAS ONE BIT FROM EACH INPUT BYTE,
* EACH INPUT BYTE IS SHIFTED LEFT BY ONE BIT.
```

```
* REPEATING ALL THIS 8 TIMES
* CREATES 16 BYTES IN OUTBUF
* WHICH REPRESENT CONTENT OF THE CHARACTER
* RE-MAPPED INTO PRINTER BIT GRAPHICS FORMAT
* SUITABLE FOR DOUBLE DENSITY GRAPHICS MODE
*
       LI   R4,8          8 BYTES TO DO
       LI   R10,OUTBUF    R10 TO OUTPUT CPU BUFFER
MAPLOP LI   R5,8          8 BITS TO MAP
       LI   R9,INPBUF     R9 POINTS TO INPUT
       CLR  R1            R1=0
INNER  MOVB *R9,R3        GET AN INPUT BYTE
       SLA  R1,1          SHIFT R1 LEFT 1 BIT
       SLA  R3,1          SHIFT INPUT BYTE LEFT 1 BIT
       JNC  NOCAR         IF NO CARRY, LEFT BIT WAS 0
       ORI  R1,1          IF A ONE, PLACE 1 IN LOW BIT OF R1
NOCAR  MOVB R3,*R9+       PUT BACK INPUT BYTE, INC POINTER
* AT THIS POINT THE INPUT BYTE, SHIFTED LEFT BY ONE BIT,
* HAS BEEN PUT BACK INTO THE INPUT BUFFER LOCATION
* AND R9 POINTS AT THE NEXT BYTE OF INPBUF
       DEC  R5            DEC BIT COUNT
       JNE  INNER         IF NOT DONE, REPEAT
       SWPB R1            SWAP SO RESULT IS IN LEFT BYTE R1
       MOVB R1,*R10+      PLACE IN OUTPUT
       MOVB R1,*R10+      REPLICATE IN NEXT BYTE
       DEC  R4            DEC BYTE COUNT
       JNE  MAPLOP        IF NOT DONE, REPEAT
       LI   R0,OBUFF      POINT AT OUTPUT VDP BUFFER
       LI   R1,OUTBUF     AND CPU OUTPUT BUFFER
       LI   R2,16         16 BYTES (ONE CHARACTER)
       BLWP @VMBW         WRITE TO BUFFER
       BL   @WRTREC       SEND 16 BYTES TO PRINTER
*
* THIS PROCESS CONTINUES FOR ALL THE RECORDS
* COMPRISING ONE ROW OF THE INSTANCE, THEN IF MORE
* RECORDS REMAIN, IT GOES BACK TO AN EARLIER POINT,
* SENDS THE CONTROL CODES AND MARGIN BYTES, THEN
* STARTS SENDING THE NEXT ROW OF THE INSTANCE.
```

1.69. The Art Of Assembly — Part 69. Click On Icon!

By Bruce Harrison

It's a different world we live in today from the world we lived in when first we bought our TI-99/4A back in 1983. Not only are there no more TI's on the market, but the PCs now available dwarf the capability available only a few years back.

1.69.1. A Rude Awakening

We got a "wakeup call" recently, when we ordered and received our new Canon BJ-200e Printer. This was actually not a new machine, but a "factory refurbished" one, costing only $180.00. The machine is great! It makes our printed output look like "laser quality". It was the printer's manual that really opened our eyes. In the past, printer manuals always contained a section explaining all of the escape sequences that could be used with that machine, and usually also contained one or more test or sample programs in GW Basic, which one could run from any "garden variety" PC. Those days are GONE! The BJ-200 was supplied with three 3 1/2 inch disks of the High Density type, containing drivers and fonts. In the manual, we were told how to load these drivers and fonts using Windows 3.1 or higher. NO ESCAPE CODES! NO SAMPLE PROGRAMS! The folks at Canon not only assume that you have a PC, but that it's a modern one, capable of running Windows. That of course also means the presence of one or more HD 3 1/2 inch drives. Among our eight computers, three are of the PC type, but none is capable of using either Windows or the High Density disks. Thus the supplied drivers and fonts were of utterly no use in our house.

1.69.2. The Help Line

Fortunately, Canon has a user help line at an 800 number. We were a bit scared to call, because the manual said that for those who wanted the escape codes, Canon would FAX them in response to a call. We imagined one of those "press 1 for . . ." recordings coming through our phone, and that we'd then be asked to key in our FAX number on the phone's touch tone keys. No, we don't have a FAX either, but Canon assumes that right along with our 198 Megahertz Pentium PC with 4096 Megabytes of memory and 160,000 Megabyte hard drive, we've also got a FAX, or a FAX Receiving capability built into our PC.

Luckily for us, Canon is not quite that modern. Our phone call was answered, after a slight delay, by a human operator. She heard our plea, then placed us on hold for a while, until a technician was available. After about twenty minutes of elevator music, interrupted now and then by recordings that expressed Canon's regret about our having to hold, we got to talk to a real person. This lady was very polite and helpful. Not only did she understand that some people still write their own "drivers", she also quickly agreed to send us the complete escape code summary by mail. It took about a week to get here from California, but it did arrive safely, and contained what we wanted to know, including some information about limitations on the emulation of an Epson. This was, however, only a brief summary, and did not explain in any detail how the escape sequences are used. For example, the ESC "k" n sequence, which selects an LQ font, was given as just that, with no indication of what values of n correspond to what LQ fonts. The same was true for the ESC q n sequence, etc.

So once again we owe thanks to Harley Ryan. Harley sent us a spare copy of his Panasonic KX-P2123 manual. That printer, as it turns out, emulates an Epson in a fashion very similar to the Epson emulation mode on our Canon BJ-200e, and its manual gives reasonably complete descriptions of the use of all its escape sequences. With very few exceptions the sequences as given in the Panasonic manual worked perfectly on the Canon. Even more exotic things, like downloading fonts, worked perfectly on the Canon using the methods shown in the Panasonic manual. Using that information, we were able to download some of the old Jim Peterson screen fonts from our TI into the BJ-200e, and thus prove that this too could be done. We've downloaded sets of fonts from character 33 through 126 with no problem. We were able, again with the help of Harley's manual, to adapt our own Word Processor so that we can use all of the resident fonts in the BJ-200. (There are eight very nice fonts available, seven of which are of the Letter Quality kind.) Even the "shadow" and "outline" modes mentioned in the Panasonic manual worked perfectly on the Canon, with exactly the same ESC "q" sequences.

The only sequences which don't work on the Canon are the ESC "a" for auto-centering and the ESC "C" for setting page length. The Canon Escape Sequences Summary mentions the non-availability of the ESC "a", but claims that the ESC "C" sequences will work. They DON'T! All in all, however, we're delighted with the quality of the printing done by the Canon BJ-200e, and would recommend it to anyone who's in the market for a new printer. Besides providing fast and excellent quality printing of both text and graphics, it is QUIET! There's a barely audible "whoosh" sound while printing. This is a pleasant change from the loud "brapp, brapp, brapp" sound of the impact dot matrix printers we've used.

1.69.3. CoCo Revisited

Some time ago, we mentioned our initial reactions to the Radio Shack Color Computer that we picked up at a "thrift" store. Since then, we also picked up the necessary parts to add disk drive capability, and also found a complete Color Computer III model. As is our normal desire, we wanted to be able to program the CoCo in Assembly language, so we shelled out some money to Radio Shack for the OS-9 Operating system. In the TI world, we keep running across things that aren't compatible with one another. Usually these involve Myarc products, such as the Hard and Floppy Disk Controller (HFDC). In the case of our CoCo II, we didn't need a "third party" like Myarc to create the lack of compatibility — Radio Shack did that all by itself. Soon after cracking the books on our new OS-9, we discovered that disks initialized through the normal "basic" built into the CoCo were completely incompatible with those initialized by OS-9, and vice versa. OS-9 organizes its disks in a UNIX-like fashion, (similar to a PC DOS disk) while CoCo Basic uses its own peculiar mid-disk method, where the catalog information is on track 17. Thus anything done with OS-9 can't be used with Basic, nor can any disk used with Basic be used under OS-9. There is another product offered by Radio Shack called BASIC09, which includes a compiler, so that one can "cross the bridge" in some fashion, but still one can't "import" an old Basic program into BASIC09 either, since the disk formats are incompatible.

Undaunted by this, we studied some books on OS-9 Assembly programming, then created some simple "starter" programs on our CoCo II that would run under OS-9. These worked, and quite nicely, but still we were just beginning to learn the CoCo's rather peculiar Assembly language. It was like starting to learn Assembly all over again. The Motorola 6809 chip's instructions are similar in some ways to the Intel 8088 that's used in our PCs, and in some ways similar to the Z80 chip that we'd had some small experience with.

It was at about this time, when we were just getting started with Assembly on the CoCo II, that we found and purchased the CoCo III. This came from the same "thrift" store, but as a complete package deal, including a very nice RGB Monitor, two printers (one Dot Matrix and one Daisy Wheel), plus a ton of software and manuals, all for only $150.00. Everything was in perfect working order! The RGB monitor uses a special interface cable (supplied) to plug into a connector on the bottom of the CoCo III. This can't be used with the CoCo II, however, which has only RF output. Like the CoCo II, the III has no parallel port, so the printers have to be compatible with the strange 4-pin RS-232 port of the CoCo. The CoCo III offers additional features in its Basic, so that one may use 256 colors 16 at a time, and one may use a 40 character screen mode. Of course this means that programs written in CoCo III Basic won't always work on the CoCo II.

1.69.4. The Really Big Shock

One of the things we wanted to do soon after getting our CoCo III was to try out those little Assembly things we'd done under OS-9. We dutifully hooked up the CoCo III to its disk drive, put in the disk with OS-9 and some of our experiments, then tried to "launch" OS-9. The disk drive whirred for a bit, but then NOTHING! The CoCo III just plain locked up! A little consultation with our friendly Radio Shack manager indicated that indeed the OS-9 that we'd shelled out $70.00 for was for use only with CoCo II, and that the different version of OS-9 for the CoCo III would cost us another $100.00! The BASIC09 for the CoCo III was also available at a similar price. Talk about "Action Figures Sold Separately"! We'd need to lay out another $200.00 to get any real value out of our CoCo III, then learn another version of Basic and Assembly, and so on!

Since then, our CoCo III and CoCo II have been placed "in storage", probably forever. Nice machines, with very advanced features for their time, but now merely relics that clutter up our computer room. Meanwhile Tandy has stopped making computers altogether. The PCs offered at Radio Shack now are all made by Packard Bell, AST, and even IBM, but not Tandy. Our Tandy 1400FD laptop is still in use, being used to create this article. Most of the time, however, we're pecking away at the TI console, finding new and different things that people thought were impossible to do on that "orphan". Still, we haven't thrown away the CoCos. Maybe some day Radio Shack will cut the price on the OS-9 and BASIC09 for the CoCo III, and then . . .

1.69.5. Let This Be A Lesson

One day, perhaps we'll invest in a new Pentium PC, Windows 95, and all that other stuff, but no more orphans. Of course by the time we get around to buying a new PC, we expect it too will be made obsolete soon after we buy it. As we write this, it's impossible to buy any software that will work on our three "dinosaur" PCs, since none of them has windows or HD disk capability. In a few years, we predict, the present high end PCs will face a similar fate, with the new Holographic Pill replacing disk drives completely, and with millions of Megabytes required to run Windows 2000. No matter, so long as we still have our old DOS 3.xx PCs running the old software, we'll still have all the computer power we need, and then some.

1.69.6. In The Meantime . . .

We'll still use our TI to answer correspondence and to serve as an outlet for our creativity. With the new printer, our output will look just as good as anything we could do with even the most modern PC, at a small fraction of the cost. Next month, maybe we'll start working on something new, like a "mouse" oriented GUI system for the TI. (What ever happened to Joe Ross and his C-Shell?)

1.70. The Art Of Assembly — Part 70. The Bathtub Curve

By Bruce Harrison

Today we're starting with a subject that's involved more with probability and statistics than with programming. This should be of interest to the more general segment of the readers of *MICROpendium* than to the Assembly "buffs". To make it even more eye-catching, we've included a picture, shown as Figure 1.

1.70.1. The Picture Explained

No, we didn't make the picture in Figure 1 using some exotic equation with our Plotter program. There probably is an equation for this curve, but we simply "fudged" the curve using our Drawing program. This curve is normally used in connection with the life cycle of some product or other, and depicts the frequency of failures for the product over time. Notice that neither time nor failure rate is "quantified" on the graph. The time span may be just a few years or a whole lifetime, but the general shape of the curve is what's important. The first part of a life cycle shows that failure rates are initially very high while the "product" is in development. Failures come down during this early period, as the product gets improved upon. Thus the early part of the curve is referred to as a learning curve. After this learning part, the failure rate levels off into what's called the random failures portion, with a relatively constant low failure rate. During this flat part of the curve, things fail at random intervals, but not too many failures occur. As you can see, the biggest part of the curve in terms of time is this random failure part. Finally, though, time "catches up" with our product, and failure rate increases rapidly in what's called the "wearout" phase of the product's life cycle.

This may well be what awaits us in the near future with our beloved TI-99/4A. We've been enjoying that flat part of the curve for more than ten years now, but we are about to be slapped in the face by the other end of the bathtub. That's the thinking which has inspired our friend Mike Wright and others to create a full-blown emulation of the TI-99/4A on modern PC computers. In theory, at least, we can transfer our beloved computer into a product that's in the early part of the random failure phase, and thus can continue to enjoy the TI's unique capabilities while the machines themselves are dying out very rapidly.

1.70.2. Other Applications

The Bathtub Curve can also apply to other things, like for example the life cycle of a programmer. When first we start programming a computer, we suffer frequent failures of one kind or another during the learning period. As we learn, our failures "level off", so there are little mistakes or bugs here and there, but these become infrequent and easily corrected. Sooner or later, however, we reach the wearout phase. In humans, this may affect either the mind or the body. For most of us, including your author, it seems the body starts to fall apart first. Thus the fingers don't always hit the right keys, the eyes get more myopic, and the energy required even to type simple programs gets harder to muster. Thus while the mind is still "productive" of sound ideas, the body won't allow them to be realized as programs. In other words, that steep back end of the bathtub is catching up with us.

1.70.3. Colorful Filling

Today's Sidebar has nothing to do with bathtubs, but deals with a question posed many moons ago by an anonymous reader. We answered a letter from that reader a long time ago, but the fact that he was concerned enough to write us triggered our putting some source code in today's column. The question was first put to us through Laura Burns as "how do you fill an area with color?" That was later expanded in a letter from the reader to your author as "how do we fill a bounded area in a bitmap screen with some selected color?" Those who've used our Drawing program know that this capability is included in that program, so the source code for today's Sidebar was readily available from that disk. The code in today's Sidebar is just a fragment, so it can't be assembled or run "as is". We sent along a copy of the Drawing program, complete with all its source code, to our reader. Here, though, we're going to lead you through the filling process in some detail.

For openers, the computer must be operating in its Bitmap Mode, and there must be an enclosed area drawn in pixels on the screen. The area can be of nearly any shape, but let's for purposes of discussion assume it's a square. The first thing we have to do in this fragment is to determine the position of our drawing cursor. The cursor in this case is a sprite in the shape of a "+" sign. The sprite attribute table has been set to >3800, so as not to interfere with the Bitmap screen. Thus to get the position coordinates for sprite #0 (our cursor), we set R0 to >3800. Reading the byte at >3800 gives us the y position of the sprite's upper left corner in dot-rows. We offset this number by 5 so that we've got the position of the center of the "+". We read the next byte from VDP RAM to get the x position, then offset that by 3 to get to the center of the "+". Thus we have in R8 the dot row position of the center of our cursor and in R7 the dot column position of the center of our cursor.

The first thing that the routine does after getting its position information is to start looking on the current row for a "filled" pixel. That is, it's examining the pattern table in VDP RAM, moving left until it finds a pixel that's "on". It does that by using the subroutine PLOTCK. That subroutine looks at the pixel pointed to by R8 and R7, and returns with status EQ if the pixel is on. If the pixel at R8, R7 coordinates is off, status will return with an NE indication. Thus the instruction just following SKLFT will cause the routine to stop moving left as soon as it finds a turned on pixel. At that point (label NMSL) we save the dot-column position from R7 into R13, then move the original starting dot-column position back into R7 and start moving to the right. The loop starting at SKRT moves R7 to the right until a filled pixel is found, then exits its loop to label NMSR. At that point, we save the right side position into R14. The section of code just after NMSR sets R7 to the center of the current row of the enclosed area, then moves up one row. If that takes it beyond our picture area, we have reached the top of our enclosed figure. If the pixel at the center of the next upward row is on, that also indicates that we've reached the top of the enclosed area. In our drawing program, dot-rows from 0 through 7 are used to display information, and are not part of the picture area, hence the checking of R8 against 8, and the JLT GOLFT0. If the pixel we checked in the new row is not turned on, then we start over scanning this new row at label STLFT.

At label GOLFT0, we start scanning downward, moving left and right, and filling the pixels with whatever color is currently set for drawing. This section of code first finds the left boundary on each row, then turns on and colors the pixels moving right until it reaches the right boundary of that row of the figure. It also center-seeks on each row, thus insuring that when we move downward, we're in the center of the row just above. This process continues until we've reached either the bottom of the enclosed area or the bottom of the picture area at row 176. When the whole area is filled, we jump to the code at FILLX, re-set conditions for the cursor, and then go back to scanning the keyboard and joystick for user input.

Along the way during our left-right scanning, we also check to see when we've reached the left and right boundaries of the picture area. Thus a "closed" area may actually be open on one side, bounded by the edge of the screen, and the fill process will still work correctly. Similarly, the closed area can be bounded by the top or bottom of the picture area.

In your own programming, you might want to use the whole screen for your picture area, instead of having our top and bottom limits. In such a case in the section following NMSR, after DEC R8, you'd simply omit the CI R8,8 line, so your "seek top" would continue through dot-row 0. In the section of code after NMRT, following the INC R8, you'd want to CI R8,191 instead of the CI R8,175 that's there. This would allow your filling to go all the way to the bottom row of the screen.

1.70.4. Nobody's Perfect

This method works very nicely for most shapes, but on occasion gets fooled. Circles, rectangles, and squares get filled perfectly. Triangles can be tricky, and sometimes they will wind up with a narrow corner unfilled. In the Drawing program, we've provided a way to fill those tiny gaps by just putting the program into "PEN DOWN" condition and moving the cursor through the unfilled area.

1.70.5. Some Cautions

This method won't work if you try filling horizontally adjacent areas with different colors. That's not because of the program per se, but because of the way the color bytes themselves are organized in the VDP memory. One byte of color affects an area of eight pixels in the horizontal direction. Thus when you fill another area that's within the eight pixel "zone" of an already filled area, the new color will bleed into the already filled area. Presumably, you could correct for this by using a background coloring in the overlapped area, but that's beyond the scope of our current discussion.

1.70.6. The Subroutines

You'll notice that we've shown the entire PLOT subroutine, but that the main routine doesn't use all of that subroutine. Instead, it uses only the part starting at label PLOTF. That's done because the necessary work of that first part of PLOT has already been performed by the PLOTCK routine, so we save some time by jumping in at PLOTF to actually put a pixel on the screen and color it. These subroutines are not entirely of our own design, but were derived from material given us by John C. Johnson of Cedar Rapids, Iowa. We're not completely sure that the annotations in these are correct, but we know that the subroutines work as intended, and that's all that matters. We hope this stuff will be helpful to any of our readers who are struggling with Bitmap Mode.

Next month, we're taking the plunge into the deep dark water of floating point math. We hope you'll be there to float along with us.

```
* SIDEBAR 70
* A FRAGMENT FOR FILLING A CLOSED AREA
* WITH COLOR IN BITMAP MODE
* Code by Bruce Harrison
* PUBLIC DOMAIN
* 9 MAR 1995
*
FILL   LI   R0,>3800     POINT AT SPRITE 0
       BLWP @VSBR        READ Y-POSITION
       MOVB R1,R8        PLACE IN R8
       MOV  R8,@OLDROW   STASH IN MEMORY
       SRL  R8,8         RIGHT JUSTIFY
       AI   R8,5         ADD 5 DOT ROWS
       INC  R0           NEXT VDP ADDRESS
       BLWP @VSBR        READ X-POSITION
       MOVB R1,R7        MOVE TO R7
       MOV  R7,@OLDCOL   STASH IN MEMORY
       SRL  R7,8         RIGHT JUSTIFY
       AI   R7,3         ADD 3 DOT COLUMNS
STLFT  MOV  R7,@ENDOC    STASH R7
SKLFT  BL   @PLOTCK      CHECK PIXEL AT R8,R7 POSITION
       JEQ  NMSL         IF SET, JUMP
       DEC  R7           ELSE MOVE LEFT ONE DOT COLUMN
       JLT  NMSL         IF PAST LEFT EDGE, JUMP
       JMP  SKLFT        ELSE REPEAT MOVING LEFT
NMSL   MOV  R7,R13       STASH AWAY COL
       MOV  @ENDOC,R7    PUT ORIGINAL COLUMN BACK
SKRT   BL   @PLOTCK      CHECK PIXEL
       JEQ  NMSR         IF SET, JUMP
       INC  R7           MOVE RIGHT ONE COLUMN
       CI   R7,255       CHECK FOR RIGHT EDGE
       JGT  NMSR         IF GREATER, JUMP
       JMP  SKRT         ELSE CONTINUE MOVING RIGHT
NMSR   MOV  R7,R14       STASH AWAY R7
       S    R13,R14      GET DIFFERENCE IN COLUMNS
```

```
       CI   R14,2        CHECK FOR 2
       JLT  GOLFT0       IF LESS, JUMP
       SRL  R14,1        TAKE HALF OF RIGHT COL
       A    R14,R13      ADD TO LEFT COL
       MOV  R13,R7       PUT CENTER POSITION IN R7
       DEC  R8           MOVE UP ONE ROW
       CI   R8,8         COMPARE TO TOP OF PICTURE AREA
       JLT  GOLFT0       IF LESS, JUMP
       BL   @PLOTCK      CHECK PIXEL
       JNE  STLFT        IF NOT SET, JUMP
*
* WHEN WE REACH HERE, WE'VE FOUND THE TOP CENTER OF THE AREA
*
GOLFT0 INC  R8           DOWN ONE ROW
       MOV  @ENDOC,R7    GET OLD COLUMN
GOLFT  BL   @PLOTCK      CHECK PIXEL
       JEQ  GORT0        IF SET, JUMP
       DEC  R7           ELSE LEFT ONE COLUMN
       JLT  GORT0        IF <0, JUMP
       JMP  GOLFT        ELSE KEEP MOVING LEFT
GORT0  MOV  R7,R13       SAVE R7
GORT   INC  R7           RIGHT ONE COLUMN
       CI   R7,255       CHECK FOR EDGE
       JGT  NMRT         IF GREATER, JUMP
       BL   @PLOTCK      CHECK PIXEL
       JEQ  NMRT         IF SET, JUMP
       MOVB @LINCLR,R9   PUT LINE COLOR BYTE IN R9
       BL   @PLOTF       PLOT AND COLOR ONE PIXEL
       JMP  GORT         THEN REPEAT
NMRT   MOV  R7,R14       STASH R7
       S    R13,R14      TAKE DIFFERENCE
       CI   R14,2        CHECK FOR 2
       JLT  FILLX        IF LESS, EXIT
       SRL  R14,1        HALVE R14
       A    R14,R13      ADD TO R13
       MOV  R13,R7       CENTER COLUMN
       INC  R8           NEXT ROW
       CI   R8,175       CHECK BOTTOM OF PICTURE AREA
       JGT  FILLX        IF GREATER, JUMP
       BL   @PLOTCK      ELSE CHECK PIXEL
       JNE  GOLFT        IF NOT SET, REPEAT FOR THIS ROW
FILLX  CLR  R4           CLEAR REG 4
       CLR  R10          CLEAR REG 10
       MOV  @OLDCOL,R7   OLD COLUMN BACK
       MOV  @OLDROW,R8   OLD ROW BACK
       BL   @NODRW       RESET THE SPRITE
       B    @KJSCAN      BACK TO SCANNING MODE
*
* SUBROUTINES
*
```

```
*
* FOLLOWING WRITES ONE PIXEL TO SCREEN AT LOCATION POINTED BY
* R8 (DOT ROW) AND R7 (DOT COLUMN)
*
PLOT   MOV  R7,R3        MOVE DOT COLUMN TO R3
       MOV  R8,R4        AND DOT ROW TO R4
       MOV  R4,R5        DOT ROW ALSO IN R5
       ANDI R5,7         R5 HAS DOT ROW MODULO 8
       SZC  R5,R4        SO DOES R4
       SLA  R4,5         MULTIPLY R4 BY 32
       A    R5,R4        ADD R5, SO R4 HAS DR MOD. 8 * 32 + DR MOD 8
       MOV  R3,R0        MOVE DOT COL TO R0
       ANDI R0,>FFF8     R0 HAS DC - DC MOD 8
       S    R0,R3        R3 HAS DC MOD 8
       A    R4,R0        ADD R4
       SWPB R0           SWAP BYTES
       MOVB R0,@>8C02    WRITE LOW ADDRESS BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH ADDRESS BYTE
       NOP               WASTE TIME
       MOVB @>8800,R1    READ THE BYTE
PLOTF  SOCB @M(R3),R1    OVERLAY MASK FROM TABLE M
PLOTF0 ORI  R0,>4000     SET THE 4000 BIT IN R0
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE LOW BYTE OF ADDRESS
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH BYTE OF ADDRESS
       NOP               WASTE TIME
       MOVB R1,@>8C00    WRITE MODIFIED BYTE BACK TO VDP
       MOV  R9,R9        IS COLOR TO BE SET?
       JEQ  PLOTX        IF NOT, JUMP AHEAD
       ANDI R0,>3FFF     STRIP OFF "4" FROM R0
       AI   R0,>2000     ADD >2000 TO POINT AT COLOR TABLE ENTRY
       BLWP @VSBR        READ THAT BYTE INTO R1
       MOVB R1,R2        MOVE THE BYTE TO R2
       ANDI R2,>F000     STRIP ALL BUT LEFT NYBBLE
       CB   R2,R9        COMPARE TO LEFT BYTE R9
       JEQ  PLOTX        IF EQUAL, COLOR ALREADY SET
       ANDI R1,>0F00     ELSE STRIP OFF LEFT NYBBLE R1
       AB   R9,R1        REPLACE WITH LEFT NYBBLE R9
       BLWP @VSBW        THEN WRITE COLOR BYTE BACK
PLOTX  RT                RETURN
PLOTCK MOV  R7,R3        MOVE DOT COLUMN TO R3
       MOV  R8,R4        AND DOT ROW TO R4
       MOV  R4,R5        DOT ROW ALSO IN R5
       ANDI R5,7         R5 HAS DOT ROW MODULO 8
       SZC  R5,R4        SO DOES R4
       SLA  R4,5         MULTIPLY R4 BY 32
       A    R5,R4        ADD R5, SO R4 HAS DR MOD. 8 * 32 + DR MOD 8
       MOV  R3,R0        MOVE DOT COL TO R0
```

```
       ANDI R0,>FFF8     R0 HAS DC - DC MOD 8
       S    R0,R3        R3 HAS DC MOD 8
       A    R4,R0        ADD R4
       SWPB R0           SWAP BYTES
       MOVB R0,@>8C02    WRITE LOW ADDRESS BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    WRITE HIGH ADDRESS BYTE
       CLR  R1           CLEAR REGISTER 1
       CLR  R2           CLEAR REGISTER 2
       MOVB @>8800,R1    READ THE BYTE
       MOVB @M(R3),R2    GET MASK BIT INTO R2
       COC  R2,R1        SEE IF R1 HAS A ONE AT MASK BIT
       RT
*
* DATA SECTION
*
M      DATA >8040,>2010,>0804,>0201  MASK DATA
LINCLR BYTE >10          LINE DRAWING COLOR
OLDROW DATA 0            OLD DOT-ROW
OLDCOL DATA 0            OLD DOT-COLUMN
ENDOC  DATA 0            STORAGE WORD
```

1.71. The Art Of Assembly — Part 71. Floating Points

By Bruce Harrison

Last month we led off with a bathtub, and this month we're floating, but not on the water in a bathtub. Back in October of 1995, we received a letter from Mr. Greg Knightes, of Coral Springs, Florida. He had noticed that the subject of Floating Point math operations had been sorely lacking in our columns. A quick check showed that he was right. While we'd mentioned the subject now and then, there was no full discussion about using the floating point operations in any of our columns. As is our usual practice, we answered Mr. Knightes' letter in a couple of days, and included a disk for him with annotated source code to illustrate the use of floating point math. That "demo" program forms the heart of today's Sidebar.

1.71.1. Getting the Numbers In

The first thing that we had to address is how to input floating point numbers through Assembly routines. There are probably many ways to do this, but the easiest way is to use a fairly simple method that's provided by an internal ROM routine available to us through XMLLNK. That routine, which we call CSN, for Convert String to Number, is very powerful. To use the routine, we first employ any of our "string input" routines, such as the CRSIN routine that we developed long ago, or the ACCEPT routine that we showed in Part 53 of this series. This simply allows the user to type on the screen in a manner akin to an ACCEPT AT operation in Extended Basic. Our ACCEPT routine includes insert and delete character capabilities, field clearing with **FCTN 3**, and so on. The key to using it for floating point numbers is that when it exits, Register 0 of our workspace points to the VDP address of the first character entered by the user. If what's typed there is a number, we can take that number by simply putting the address from R0 into the RAM Pad at >8356, then invoking the CSN routine through XMLLNK. The CSN routine reads the number from the screen image in VDP RAM, and creates from that a floating point number in the eight byte area starting at >834A. That area of RAM Pad is called the Floating Point Accumulator, or FAC, for short.

1.71.2. The Rules Of Input

The number must start at the beginning of the input field with either a numeric character (0-9), a minus sign, a decimal point, or a plus sign. If none of those is found at the beginning of the field, the result reported to FAC will be zero. To put that another way, the numeric entry must be left justified in the entry field. If the field is blank or has leading spaces, the result will be zero at FAC. Operators plus or minus will be accepted as the first character, but not later in the entry field, except as part of an exponent. The multiply, raise to, and divide operators will not be accepted anywhere in the entry. In other words, the entry must be purely numbers. The one exception to this rule is the E for exponent operation, as in scientific number notation. If for example the field contains 2E3, the E3 will be correctly interpreted as meaning that the two gets multiplied by 10 raised to the 3rd power, so the number reported to FAC will be 2000 in floating point notation. The E must be upper case, and may be followed by a + or - to indicate the sign of the exponent. In the version of ACCEPT shown in today's Sidebar, we've included code that will make any alpha character into upper case, so your E for exponent may be typed as lower case, but will appear in upper case on the screen.

1.71.3. Differences From XB INPUT

If we use the Basic or Extended Basic INPUT routine for a numeric value, as in INPUT N, the operation is different from what happens in our Assembly case. To start with, our number may be preceded by leading spaces in Basic or XB, but the number will still be recognized and reported correctly to the numeric variable in floating point notation. If the input field in Basic or XB is left blank, a WARNING will be issued, and the value of the variable will not be affected. In our Assembly case, a blank field will simply be accepted as zero. Of course you could put in some Assembly code to "strip off" any leading spaces in your input field before using the CSN routine, and thus make your input behave like the Basic INPUT in that respect. The floating point number generated by CSN will be correct to 14 significant digits, with the last digit rounded if need be. The number is placed at FAC as eight bytes in radix 100 notation. The first of those eight bytes is the power of 100 by which the remaining seven bytes are multiplied. Each of the remaining seven bytes is equal to two significant digits, ranging from 0 through 99 (decimal) in value. The power of 100 in the first byte is offset by >40, so that both positive and negative powers of 100 can be handled. In other words, the first byte being >40 means the number is multiplied by 100 to the 0th power, >41 means the number is multiplied by 100 to the 1st power, >3F means the number is multiplied by 100 to the -1 power, etc. This way, the powers of 100 can range from ->40 through >7F. The most significant bit (>80) in this first byte is used for the sign of the number, so if the MSB is set, the number itself is negative.

This way of doing floating point numbers is radically different from the way floating point numbers are handled by most other computers. On PC computers, for example, the floating point numbers are handled in only four bytes in binary notation. The method used in the TI yields much more accurate numbers than the PC method. Of course it takes twice as many bytes to store a number, but having 14 digit accuracy can come in handy. Your TI does more accurate math than the PC!

1.71.4. Today's Sidebar

It's very long, but is a complete program that demonstrates the use of floating point numbers on your TI. The program accepts two numbers from the user, stashes them away in memory, then performs various math operations on them. Before each operation, it copies the numbers back into the RAM Pad memory at locations called FAC and ARG. FAC is the eight bytes starting at >834A, and ARG is the eight bytes starting at >835C. After the numbers are in these two places, we can perform a wide variety of operations on them. For example, we can use XMLLNK to add, subtract, multiply, divide, or compare the numbers. For the four main operations, the result of the operation is a number at FAC, and the number at ARG is meaningless after the operation. For the compare operation, both FAC and ARG numbers remain intact after the compare, and the result is indicated by the state of the GPL Status byte at >837C. Please note that the computer's status register is not set by this compare, so we have to examine the byte at >837C to figure out the result of the comparison. If, for example, the numbers are equal, the byte at >837C will equal >20. If they're not equal, then we have to isolate the bits of that byte to determine whether the number at ARG is greater than or less than the number at FAC. In our example, we've put the status byte into R3, then masked R3 with >4000, which leaves just the "greater than" bit in R3. If the "greater than" bit was zero, then the number at ARG was less than the number at FAC.

1.71.5. Nasty Little Details

If you examine the Sidebar closely, you'll see some odd little things done, which we'd better explain now. Among the EQUates, you'll see one called VSTACK, set to >1000. That's used in the code soon after label START to put the number >1000 into the word at >836E. If we were doing only the math operations that use XMLLNK, we would not have to set a value in >836E. We set this number because later in our program, we perform a SIN function using GPLLNK. That function, and presumably others that are used through GPLLNK, uses a "value stack" in VDP RAM to do its calculations. If we don't initialize the Value Stack Pointer at >836E, the SIN operation will mess up part of our display screen by writing stuff into the screen image portion of VDP RAM.

1.71.6. Operation Of The Program

The Sidebar is a complete program which illustrates many things. First, it prompts for and accepts two floating point numbers. These get placed into memory as eight bytes each at labels NUM1 and NUM2. We've made a little subroutine called MOVNUM to make it easier to move eight-byte numbers from one place to another. For example, when we wish to move NUM1 into the eight bytes at ARG, we simply LI R9,NUM1, LI R10,ARG, then BL @MOVNUM.

You'll notice that we do this each time before an operation, and that except for the first case, we have to put NUM2 into FAC through a MOVNUM operation. In that first case, we didn't have to move NUM2 into FAC because it's still there from the previous Convert String to Number operation.

For the add and multiply operations, it doesn't matter which number is at FAC and which at ARG. For the subtract and divide operations, however, it's important to remember that the number at ARG gets the number at FAC subtracted from it, and the number at ARG gets divided by the number at FAC. Similarly in the compare operation, the indication for greater than means that the number at ARG is greater than the number at FAC. In all floating point math operations except compare, the result of the operation is reported at FAC. Thus when we invoke our subroutine at DISNUM, the result number at FAC is what gets converted to a string for display. The subroutine DISNUM puts the number from FAC on the screen at whatever location was set in R0 before calling DISNUM. For positive numbers, the string displayed by DISNUM will have a leading space, while negative numbers will have a minus in that first string character.

We're going to stop at this point because the Sidebar this month is very long. Keep this month's issue handy, because next month we're going to continue this discussion with more detailed examination of today's Sidebar. See you then.

```
* SIDEBAR 71
* A COMPLETE PROGRAM
*
* DEMO OF FLOATING POINT OPERATIONS
* PUBLIC DOMAIN
* 10/31/95
* BY Bruce Harrison
*
       REF  VSBW,VSBR,VMBW,VMBR,KSCAN,XMLLNK  REF UTILS
       DEF  START        DEFINE ENTRY
*
* REQUIRED EQUATES
*
STATUS EQU  >837C        GPL STATUS BYTE
KEYADR EQU  >8374        KEY-UNIT
KEYVAL EQU  >8375        KEY VALUE
FAC    EQU  >834A        F.P. ACCUMULATOR (8 BYTES)
FAC11  EQU  >8355        F.P. ACCUM +11
FAC12  EQU  >8356        F.P. ACCUM +12
ARG    EQU  >835C        F.P. ARGUMENT (8 BYTES)
CNS    EQU  >0014        CONVERT F.P. TO STRING W/GPLLNK
CSN    EQU  >1000        CONV. STRING (IN VDP) TO F.P. W/XMLLNK
FADD   EQU  >0600        ADD F.P.NUMBERS W/XMLLNK
FSUB   EQU  >0700        SUBTRACT FAC FROM ARG F.P. W/XMLLNK
FMUL   EQU  >0800        MULTIPLY F.P. NUMBERS W/XMLLNK
FDIV   EQU  >0900        DIVIDE ARG BY FAC F.P. W/XMLLNK
FCOM   EQU  >0A00        COMPARE ARG TO FAC F.P. W/XMLLNK
SINE   EQU  >002E        SINE OF FAC F.P. W/GPLLNK
VSTACK EQU  >1000        OUR VDP STACK
GPLWS  EQU  >83E0        GPL WORKSPACE
GR4    EQU  GPLWS+8      GPL REG 4
GR6    EQU  GPLWS+12     GPL REG 6
STKPNT EQU  >8373        STACK POINTER
LDGADD EQU  >60          LOAD GPL ADDRESS
XTAB27 EQU  >200E        XTABLE 27
GETSTK EQU  >166C        GET STACK
*
* MAIN CODE SECTION
*
START  LWPI WS           LOAD OUR WORKSPACE
       CLR  @KEYADR      CLEAR KEY-UNIT
       LI   R0,VSTACK    VALUE STACK ADDRESS
       MOV  R0,@>836E    SET VALUE STACK POINTER
RESTR  LI   R0,3         ROW 1, COL 4
       LI   R1,N1STR     'ENTER FIRST NUMBER'
       BL   @DISSTR      DISPLAY
       BL   @ACCEPT      USE ACCEPT SUBROUTINE
       DATA 32+2         SCREEN POSITION R2, C3
       DATA 28           FIELD LEN
       DATA 1            0 - DON'T, 1 - CLEAR FIELD
```

```
        DATA TEMSTR       STRING DESTINATION
        MOV  R0,@FAC12    VDP ADDRESS TO FAC12
        BLWP @XMLLNK      USE XML LINKAGE
        DATA CSN          CONVERT STRING FROM VDP TO NUMBER
        LI   R9,FAC       POINT AT FLOATING POINT ACCUMULATOR
        LI   R10,NUM1     MEM ADDRESS FOR 1ST NUMBER
        BL   @MOVNUM      PLACE THE NUMBER AT NUM1
        LI   R0,3*32+3    ROW 4, COL 4
        LI   R1,N2STR     'ENTER SECOND NUMBER'
        BL   @DISSTR      DISPLAY
        BL   @ACCEPT      USE ACCEPT SUBROUTINE
        DATA 4*32+2       SCREEN POSITION R5, C3
        DATA 28           FIELD LEN
        DATA 1            0 - DON'T, 1 - CLEAR FIELD
        DATA TEMSTR       STRING DESTINATION
        MOV  R0,@FAC12    VDP ADDRESS TO FAC12
        BLWP @XMLLNK      USE XML
        DATA CSN          CONVERT STRING TO F.P.
        LI   R9,FAC       POINT AT ACCUM.
        LI   R10,NUM2     MEM LOCATION FOR NUM2
        BL   @MOVNUM      PLACE THE NUMBER
        LI   R0,7*32+1    ROW 8, COL 2
        LI   R1,ADDSTR    PLUS
        BL   @DISSTR      DISPLAY
        LI   R9,NUM1      NUM1
        LI   R10,ARG      TO ARGUMENT
        BL   @MOVNUM      PLACE NUM1
        BLWP @XMLLNK      USE XML
        DATA FADD         ADD ARG TO FAC
        A    R2,R0        ADD LENGTH TO ADDRESS
        LI   R4,18        18 CHARS
        BL   @BLNFLD      CLEAR THE 18 CHARS
        BL   @DISNUM      DISPLAY THE NUMBER AT FAC
        LI   R9,NUM1      POINT AT NUM1
        LI   R10,ARG      AND ARG
        BL   @MOVNUM      PUT NUM1 AT ARG
        LI   R9,NUM2      POINT AT NUM2
        LI   R10,FAC      AND FAC
        BL   @MOVNUM      MOVE NUM2 TO FAC
        LI   R0,9*32+1    ROW 10, COL 2
        LI   R1,SUBSTR    SUBTRACT
        BL   @DISSTR      DISPLAY
        BLWP @XMLLNK      USE XML
        DATA FSUB         SUBTRACT FAC FROM ARG
        A    R2,R0        ADD LENGTH
        LI   R4,18        18 CHARS
        BL   @BLNFLD      BLANK FIELD
        BL   @DISNUM      DISPLAY THE NUMBER
*
* FOLLOWING REPEATS THE PROCESS FOR MULTIPLY AND DIVIDE
```

```
*
       LI   R9,NUM1
       LI   R10,ARG
       BL   @MOVNUM
       LI   R9,NUM2
       LI   R10,FAC
       BL   @MOVNUM
       LI   R0,11*32+1
       LI   R1,MULSTR
       BL   @DISSTR
       BLWP @XMLLNK
       DATA FMUL
       A    R2,R0
       LI   R4,18
       BL   @BLNFLD
       BL   @DISNUM
       LI   R9,NUM1
       LI   R10,ARG
       BL   @MOVNUM
       LI   R9,NUM2
       LI   R10,FAC
       BL   @MOVNUM
       LI   R0,13*32+1
       LI   R1,DIVSTR
       BL   @DISSTR
       BLWP @XMLLNK
       DATA FDIV
       A    R2,R0
       LI   R4,18
       BL   @BLNFLD
       BL   @DISNUM
*
* FOLLOWING COMPUTES AND SHOWS SIN(NUM1)
*
       LI   R9,NUM1      1ST NUMBER
       LI   R10,FAC      TO FAC
       BL   @MOVNUM      MOVE THAT
       BLWP @GPLLNK      USE GPLLNK
       DATA SINE         TO COMPUTE SIN(NUM1)
       LI   R0,15*32+1   ROW 16, COL 2
       LI   R1,SINSTR    "SIN OF NUM1"
       BL   @DISSTR      DISPLAY THAT
       A    R2,R0        MOVE POINTER
       LI   R4,18        18 CHARS
       BL   @BLNFLD      BLANK FIELD
       BL   @DISNUM      DISPLAY THE NUMBER
*
* FOLLOWING COMPARES NUM1 AND NUM2
*
       LI   R9,NUM1      1ST NUMBER
```

```
       LI   R10,ARG      TO ARGUMENT
       BL   @MOVNUM      MOVE
       LI   R9,NUM2      2ND NUMBER
       LI   R10,FAC      TO FAC
       BL   @MOVNUM      MOVE
       BLWP @XMLLNK      USE XML LINK
       DATA FCOM         COMPARE F.P. NUMBERS
       CB   @STATUS,@ANYKEY IS STATUS BYTE = >20?
       JEQ  SEQ          THEN NUMBERS EQUAL
       MOVB @STATUS,R3   MOV TO R3
       ANDI R3,>4000     MASK TO > BIT
       JEQ  SLT          IF ZERO, JUMP
SGR    LI   R1,GRTSTR    ELSE SET GREATER
       JMP  SHWCMP       THEN JUMP
SEQ    LI   R1,EQUSTR    ARG = FAC
       JMP  SHWCMP       THEN JUMP
SLT    LI   R1,LESSTR    NUM1 < NUM2
SHWCMP LI   R0,17*32+1   ROW 18, COL2
       BL   @DISSTR      DISPLAY STRING
       LI   R0,19*32+5   ROW 20, COL 6
       LI   R1,PAK       "PRESS ANY KEY"
       BL   @DISSTR      DISPLAY THAT
       A    R2,R1        NEXT STRING
       LI   R0,21*32+4   ROW 22, COL 5
       BL   @DISSTR      DISPLAY "OR FCTN-8"
       LI   R0,23*32+15  ROW 24, COL 16
       CLR  @>8378       CLEAR TIMER
       MOVB @CURSOR,R1   CURSOR CHAR
       BLWP @VSBW        ON SCREEN
       MOV  @INTLOC,@>83C4 ENABLE USER INTERRUPT
       MOVB @ANYKEY,@ALTKEY ALTERNATE SPACE
       BL   @KEYLOO      USE KEY LOOP
       CLR  @>83C4       STOP USRINT
       MOVB @ANYKEY,R1   SPACE IN R1
       BLWP @VSBW        WRITE THAT
       CI   R8,6         WAS FCTN-8 STRUCK?
       JNE  EXIT         IF NOT, EXIT
       B    @RESTR       ELSE RE-START
EXIT   LWPI >83E0        GPL WORKSPACE
       B    @>6A         TO GPL INTERPRETER
*
* SUBROUTINES
*
ACCEPT MOV  *R11+,R0     R0 HAS START POSITION
       JNE  GETLEN       IF NOT 0, JUMP
       INC  R0           ELSE POINT AT 1
GETLEN MOV  *R11+,R2     R2 HAS MAX LENGTH
       MOV  *R11+,R3     R3 HAS CLEAR FIELD SIGNAL
       MOV  *R11+,R9     R9 HAS STRING DESTINATION
       CLR  @INSFLG      NOT IN INSERT
```

```
       MOV  R0,R7        SAVE START POSITION
       MOV  R2,R4        SAVE LENGTH
       DEC  R0           POINT ONE BACK
       MOVB @EDGE,R1     EDGE CHARACTER
       BLWP @VSBW        WRITE A BYTE
       INC  R0           BACK TO START
       A    R2,R0        ADD LENGTH
       MOV  R0,R6        SAVE THAT POSITION
       DEC  R6           LAST ALLOWED
       BLWP @VSBW        WRITE EDGE CHAR
CLRSNS MOV  R7,R0        BACK TO START
       MOV  R3,R3        CHECK SIGNAL
       JEQ  KEYFRC       IF ZERO, JUMP
       MOV  R4,R2        GET LENGTH BACK IN R2
       MOVB @ANYKEY,R1   SPACE CHAR
CLRFLD BLWP @VSBW        WRITE ONE SPACE
       INC  R0           MOVE AHEAD ONE
       DEC  R2           DEC COUNT
       JNE  CLRFLD       IF NOT 0, RPT
       MOV  R7,R0        GET START BACK
*
* KEYFRC GETS THE CURRENT CHARACTER
* FROM THE SCREEN, FORCES THE CURSOR
* TO THAT POSITION, THEN ACTIVATES THE
* USER INTERRUPT TO BLINK CURSOR
*
KEYFRC BLWP @VSBR        READ BYTE AT R0 POSITION
       MOVB R1,@ALTKEY   PLACE AT ALTKEY
       MOVB @CURSOR,R1   PUT CURSOR IN R1
       BLWP @VSBW        WRITE CURSOR
       CLR  @>8378       CLEAR TIME COUNTER
       MOV  @INTLOC,@>83C4 ENABLE USER INTERRUPT
*
* KEYIN IS THE PART THAT GETS KEYSTROKES
*
KEYIN  BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTERRUPTS
       LIMI 0            STOP THEM
       CB   @STATUS,@ANYKEY KEY STRUCK?
       JNE  KEYIN        IF NOT, REPEAT
*
* FOLLOWING CODE USES THE KEYSTROKE
*
       MOV  @KEYADR,R8   KEY AS WORD IN R8
       MOVB @ALTKEY,R1   OLD CHAR IN R1
       BLWP @VSBW        WRITE TO SCREEN
       CB   @KEYVAL,@ENTERV "ENTER" STRUCK?
       JEQ  KEYEX        IF YES, EXIT
       CB   @KEYVAL,@BACKUP FCTN-S?
       JNE  KEY0         IF NOT, JUMP
```

```
*
* FOLLOWING IS CODE THAT HANDLES FCTN-S
* IT MOVES CURSOR ONE SPOT, THEN GOES TO
* RPTKEY, WHICH DELAYS BEFORE ALLOWING REPEAT
*
       DEC  R0           DEC SCRN POSITION
       BLWP @VSBR        READ BYTE
       CB   R1,@EDGE     EDGE CHARACTER?
       JNE  BCKX         IF NOT, JUMP
       INC  R0           ELSE INC POSITION
       JMP  KEYFRC       THEN BACK
BCKX   B    @RPTKEY       AHEAD FOR REPEAT ACTION
KEY0   CB   @KEYVAL,@FWARD FCTN-D?
       JNE  KEY1         IF NOT, JUMP AHEAD
*
* FOLLOWING IS CODE THAT HANDLES FCTN-D
* IT MOVES CURSOR ONE SPOT, THEN GOES TO
* RPTKEY, WHICH DELAYS BEFORE ALLOWING REPEAT
*
       INC  R0           POINT AHEAD
       BLWP @VSBR        READ BYTE
       CB   R1,@EDGE     EDGE CHAR?
       JNE  FWKX         IF NOT, JUMP
       DEC  R0           ELSE POINT BACK
       B    @KEYFRC      THEN BRANCH BACK
FWKX   JMP  RPTKEY       AHEAD FOR REPEAT ACTION
KEY1   CI   R8,32        COMPARE TO SPACE BAR
       JLT  FUNCT        IF LESS, CHECK FOR FUNCT
       CI   R8,122       CHECK L.C. z
       JGT  CHKINS       IF GREATER, JUMP
       CI   R8,97        CHECK L.C. a
       JLT  CHKINS       IF LESS, JUMP
       SB   @ANYKEY,@KEYVAL ELSE CONVERT TO U.C.
*
* FOLLOWING HANDLES KEY VALUES 32 AND ABOVE
*
CHKINS MOV  @INSFLG,R1   INSERT MODE?
       JEQ  KEY1A        IF NOT, JUMP AHEAD
*
* FOLLOWING HANDLES INSERT IF IN INSERT MODE
*
       C    R0,R6        AT END OF FIELD?
       JEQ  KEY1A        IF SO, NO INSERT
       MOV  R6,R2        GET LAST POSITION
       S    R0,R2        SUBTRACT CURRENT POSITION
       MOV  R9,R1        USE ASSIGNMENT SPACE
       BLWP @VMBR        PUT BYTES THERE
       INC  R0           POINT AHEAD ONE
       BLWP @VMBW        WRITE THERE
DEC0   DEC  R0           BACK TO OLD POSITION
```

```
       JMP  KEY1A        PUT IN THE KEYSTROKE
*
* FOLLOWING HANDLES FUNCTION KEYS WITH VALUES BELOW 32
*
FUNCT  CB   @KEYVAL,@DELKEY DELETE KEY?
       JNE  FUNCT2       IF NOT, JUMP
*
* FOLLOWING HANDLES DELETE WITH FCTN-1
*
       MOV  R0,R3        STASH AWAY R0
       MOV  R6,R2        GET END OF FIELD
       S    R0,R2        SUBTRACT CURRENT POSITION
       JEQ  NULDEL       IF ZERO, JUMP AHEAD
       INC  R0           ELSE POINT AHEAD ONE
       MOV  R9,R1        POINT AT ASSIGNMENT PLACE
       BLWP @VMBR        READ TO THERE
       DEC  R0           POINT BACK ONE
       BLWP @VMBW        WRITE TO THERE
NULDEL MOV  R6,R0        GET END OF FIELD
       MOVB @ANYKEY,R1   SPACE CHAR
       BLWP @VSBW        WRITE A SPACE
       MOV  R3,R0        GET OLD POSITION BACK
       JMP  KEYFRC       JUMP TO GET NEXT KEY
FUNCT2 CB   @KEYVAL,@INSKEY FCTN-2 PRESSED?
       JNE  FUNCT3       IF NOT, JUMP
*
* FOLLOWING SETS INSERT MODE ON FCTN-2
*
       INC  @INSFLG      SET INSERT FLAG
       JMP  KEYFRC       THEN BACK
FUNCT3 CB   @KEYVAL,@ERSKEY FCTN-3 PRESSED?
       JNE  FUNCT9       IF NOT, JUMP
*
* FOLLOWING ERASES FIELD IF FCTN-3 STRUCK
*
ERSFLD MOVB @ANYKEY,R3   SET R3 NON-ZERO
       B    @CLRSNS      BRANCH TO CLEAR FIELD
*
* FCTN-9 EXITS FROM ROUTINE
*
FUNCT9 CI   R8,15        FCTN-9?
       JEQ  KEYEX        IF SO, EXIT ROUTINE
*
* FCTN-8 CAUSES ERASE OF FIELD
*
       CI   R8,6         FCTN-8?
       JEQ  ERSFLD       IF SO, ERASE
       B    @KEYFRC      ELSE IGNORE KEYSTROKE
*
* FOLLOWING PUTS CURRENT KEYSTROKE ON SCREEN
```

```
* THEN MOVES CURSOR TO NEXT SPOT
*
KEY1A  MOVB @KEYVAL,R1   GET KEY VALUE IN R1
       BLWP @VSBW        WRITE THAT
       INC  R0           POINT AHEAD
       BLWP @VSBR        READ A BYTE
       CB   R1,@EDGE     EDGE?
       JNE  KEY1X        IF NOT, OKAY
       DEC  R0           POINT BACK
KEY1X  B    @KEYFRC      THEN BRANCH BACK
*
* KEYEX IS THE EXIT FROM THIS ROUTINE
*
KEYEX  CLR  @>83C4       KILL USER INTERRUPT
       MOV  R4,R2        GET LENGTH
       MOV  R6,R0        AND LAST POSITION
RDBYT  BLWP @VSBR        READ A BYTE
       CB   R1,@ANYKEY   SPACE?
       JNE  RDSTR        IF NOT, JUMP
       DEC  R0           ELSE DEC POSITION
       DEC  R2           AND CHAR COUNT
       JNE  RDBYT        IF NOT ZERO, GO BACK
RDSTR  MOV  R9,R1        GET STRING LOCATION
       MOV  R7,R0        AND START POSITION
       SWPB R2           SWAP BYTES
       MOVB R2,*R1+      PUT LENGTH BYTE AT STRING LOCATION
       JEQ  NULSTR       IF ZERO, JUMP
       SWPB R2           SWAP R2 AGAIN
       BLWP @VMBR        READ STRING CONTENT
NULSTR RT                RETURN TO CALLER
*
* UPON EXIT, THE ENTRY IS PLACED AS A STRING WHERE ASSIGNED,
* AND REGISTER 8 HAS THE KEYSTROKE THAT CAUSED THE EXIT
*
*
* FOLLOWING IS THE REPEAT-KEY FUNCTION FOR LEFT AND RIGHT
* MOVEMENT OF THE CURSOR
*
RPTKEY BLWP @VSBR        READ CURRENT CHAR
       MOVB R1,@ALTKEY   PLACE AT ALTKEY
       MOVB @CURSOR,R1   GET CURSOR
       BLWP @VSBW        WRITE THAT
       CLR  @INSFLG      CLEAR INSERT MODE
       CLR  @>8378       CLEAR TIMER
       CLR  @>83C4       DISABLE USRINT
*
* THE LOOP STARTING AT RPT1 DELAYS REPEAT MOTION FOR
* 32/60THS OF A SECOND UNLESS KEY IS RELEASED
*
RPT1   BLWP @KSCAN       SCAN KEYBOARD
```

```
       LIMI 2            ALLOW INTS
       LIMI 0            STOP INTS
       CB   @KEYVAL,@NOKEY NO KEY?
       JEQ  RPTEX        IF SO, EXIT
       CB   @>8379,@ANYKEY COMPARE TO 32
       JLT  RPT1         IF LESS, JUMP
RPT1A  CLR  @>8378       CLEAR TIMER
       MOVB @ALTKEY,R1   GET ALTKEY BACK
       BLWP @VSBW        WRITE
       CB   @KEYVAL,@BACKUP BACKWARD?
       JNE  RPTF         IF NOT, JUMP
       DEC  R0           ELSE BACK ONE
       BLWP @VSBR        READ CHAR
       CB   R1,@EDGE     IS THAT EDGE CHAR?
       JNE  RPTF1        IF NOT, JUMP
       INC  R0           PUT POSITION BACK
       JMP  RPTEX        THEN EXIT
RPTF   INC  R0           AHEAD ONE
RPTF1  BLWP @VSBR        READ CHAR
       CB   R1,@EDGE     EDGE?
       JNE  RPTFA        IF NOT, JUMP
       DEC  R0           BACK ONE
       JMP  RPTEX        THEN EXIT
RPTFA  MOVB R1,@ALTKEY   STASH CURRENT CHAR
       MOVB @CURSOR,R1   CURSOR IN R1
       BLWP @VSBW        WRITE CURSOR
*
* THE LOOP AT RPT2 DELAYS 8/60THS UNLESS KEY IS RELEASED
*
RPT2   BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            INTS ON
       LIMI 0            THEN OFF
       CB   @KEYVAL,@NOKEY NO KEY?
       JEQ  RPTEX        IF SO, EXIT
       CB   @>8379,@BACKUP COMPARE TO 8
       JLT  RPT2         IF LESS, REPEAT
*
* AFTER 8/60THS, CURSOR ADVANCES ANOTHER STEP
*
       JMP  RPT1A        ELSE JUMP BACK
RPTEX  MOVB @ALTKEY,R1   OLD CHAR
       BLWP @VSBW        WRITE THAT
       B    @KEYFRC      THEN BRANCH BACK
*
* FOLLOWING IS THE "BLINK", DONE WITH USER INTERRUPT
* EVERY 20 60THS, THIS WILL BLWP @CHVECT TO CHANGE
* FROM CURSOR TO CHARACTER OR VICE VERSA
*
USRINT CB   @>8379,@TWENTY TIMER=20?
       JLT  INTEX        IF LESS, EXIT
```

```
       BLWP @CHVECT      ELSE CHANGE CHAR
INTEX  RT                RETURN TO INTERRUPT HANDLER
*
* CHVECT CHANGES FROM CURSOR TO CHAR AND VICE VERSA
* EVERY 20/60THS OF A SECOND.  (THAT'S 1/3 SECOND)
*
CHVECT DATA WS,CHG1      OUR OWN WORKSPACE, CHANGE CODE
CHG1   BLWP @VSBR        READ CURRENT BYTE FROM SCREEN
       CB   R1,@CURSOR   IS THAT CURSOR?
       JEQ  CHG2         IF YES, JUMP
       MOVB @CURSOR,R1   ELSE GET CURSOR
       BLWP @VSBW        AND WRITE THAT
       JMP  CHGX         THEN EXIT
CHG2   MOVB @ALTKEY,R1   PUT OLD CHAR IN R1
       BLWP @VSBW        WRITE THAT
CHGX   CLR  @>8378       CLEAR TIMER
       RTWP              THEN RETURN
*
* BLNFLD CLEARS A SCREEN AREA
* STARTING AT R0 POSITION, EXTENDING R4 SPACES
*
BLNFLD MOV  R0,R3        SAVE R0 IN R3
       MOVB @ANYKEY,R1   SPACE CHAR IN R1
       MOV  R4,R2        COPY R4 TO R2
BLN1   BLWP @VSBW        WRITE A SPACE
       INC  R0           MOVE POINTER
       DEC  R2           DEC COUNT
       JNE  BLN1         IF NOT ZERO, RPT
       MOV  R3,R0        GET OLD R0 BACK
       RT                THEN RETURN
*
* DISSTR DISPLAYS A STRING ON SCREEN
*
DISSTR MOVB *R1+,R2      GET LENGTH BYTE
       SRL  R2,8         RIGHT JUSTIFY
       JEQ  DISX         IF ZERO, EXIT
       BLWP @VMBW        ELSE WRITE STRING
DISX   RT                RETURN
*
* DISNUM CONVERTS A FLOATING POINT NUMBER TO A STRING,
* THEN DISPLAYS THAT STRING
*
DISNUM CLR  @FAC11       SET FOR BASIC FORMAT
       BLWP @GPLLNK      USE GPL LINK
       DATA CNS          CONVERT F.P. AT FAC TO STRING
       MOVB @FAC12,R2    STRING LENGTH TO R2
       SRL  R2,8         RIGHT JUSTIFY
       MOVB @FAC11,R1    STRING ADDRESS TO R1
       SRL  R1,8         RIGHT JUSTIFY
       AI   R1,>8300     ADD >8300 OFFSET
```

```
       BLWP @VMBW        DISPLAY THE STRING
       RT                RETURN
*
* KEYLOO WAITS FOR A KEYSTROKE, THEN RETURNS
* IN THIS INSTANCE, WE'VE MADE THE CURSOR BLINK
* WHILE KEYLOO IS EXECUTING.
*
KEYLOO BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTS
       LIMI 0            THEN STOP
       CB   @STATUS,@ANYKEY ANY KEY?
       JNE  KEYLOO       IF NOT, REPEAT
       MOV  @KEYADR,R8   KEY AS WORD INTO R8
       RT                THEN RETURN
*
* MOVNUM MOVES A FLOATING POINT NUMBER FROM
* THE LOCATION POINTED BY R9 TO
* THE LOCATION POINTED BY R10
*
MOVNUM LI   R4,8          EIGHT BYTES TO MOVE
MOVBYT MOVB *R9+,*R10+    MOVE ONE, INC POINTERS
       DEC  R4            DECREMENT COUNT
       JNE  MOVBYT        IF NOT ZERO, REPEAT
       RT                 RETURN

*  GENERAL PURPOSE GPL LINK
*  BY WARREN/MILLER

GPLLNK DATA GLNKWS
       DATA GLINK1
RTNAD  DATA XMLRTN
GXMLAD DATA >176C
       DATA >50
GLNKWS EQU  $->18
       BSS  >08
GLINK1 MOV  *R11,@GR4
       MOV  *R14+,@GR6
       MOV  @XTAB27,R12
       MOV  R9,@XTAB27
       LWPI GPLWS
       BL   *R4
       MOV  @GXMLAD,@>8302(R4)
       INCT @STKPNT
       B    @LDGADD
XMLRTN MOV  @GETSTK,R4
       BL   *R4
       LWPI GLNKWS
       MOV  R12,@XTAB27
       RTWP
*
```

```
* DATA SECTION
*
WS     BSS  32           OUR WORKSPACE
INTLOC DATA USRINT       USER INTERRUPT ADDRESS
INSFLG DATA 0            INSERT FLAG
NUM1   BSS  8            STORAGE FOR FIRST NUMBER
NUM2   BSS  8            STORAGE FOR SECOND NUMBER
DELKEY BYTE 3            FCTN-1 VALUE
INSKEY BYTE 4            FCTN-2 VALUE
ERSKEY BYTE 7            FCTN-3 VALUE
TEMSTR BSS  30           TEMPORARY STRING
ALTKEY BYTE 0            CURRENT CHARACTER FROM SCREEN
ENTERV BYTE 13           ENTER KEY VALUE
CURSOR BYTE 30           CURSOR CHAR
BACKUP BYTE 8            FCTN-S
FWARD  BYTE 9            FCTN-D
ANYKEY BYTE 32           SPACE OR COMPARISON BYTE
TWENTY BYTE 20           CURSOR BLINK NUMBER
NOKEY  BYTE >FF          NO KEY INDICATION
EDGE   BYTE 31           EDGE CHAR
N1STR  BYTE 18
       TEXT 'ENTER FIRST NUMBER'
N2STR  BYTE 19
       TEXT 'ENTER SECOND NUMBER'
ADDSTR BYTE 11
       TEXT 'NUM1+NUM2= '
SUBSTR BYTE 11
       TEXT 'NUM1-NUM2= '
MULSTR BYTE 11
       TEXT 'NUM1*NUM2= '
DIVSTR BYTE 11
       TEXT 'NUM1/NUM2= '
SINSTR BYTE 13
       TEXT 'SIN OF NUM1= '
EQUSTR BYTE 24
       TEXT 'NUM1 IS EQUAL TO NUM2   '
GRTSTR BYTE 24
       TEXT 'NUM1 IS BIGGER THAN NUM2'
LESSTR BYTE 24
       TEXT 'NUM1 IS LESS THAN NUM2  '
PAK    BYTE 21
       TEXT 'PRESS ANY KEY TO EXIT'
OR8    BYTE 19
       TEXT 'OR FCTN-8 TO REPEAT'
       END
```

1.72. The Art Of Assembly — Part 72. Still Afloat

By Bruce Harrison

This month's column is a continuation of last month's. You'll need to get the Sidebar from last month in front of you to understand what we're writing about. Got it? Okay, ready or not, here we go.

Sidebar 71 is a complete program to demo some typical floating point math operations. You'll notice that the source starts off with more than the usual EQUates. These are of course not absolutely necessary, but are used so that there will be mnemonics available to make the bulk of the source code a bit easier to grasp. FAC stands for the Floating point ACcumulator. This is a very important memory location for any of the floating point operations. It refers to the eight bytes starting at address >834A in the RAM Pad. That's where floating point numbers get placed by the ROM and GROM floating point math routines that we'll use in this program. ARG, which stands for floating point ARGument, is another eight byte portion of RAM Pad, starting at >835C. This is used for a second floating point number. For example, the addition routine adds the number in FAC to the one in ARG, and puts the result at FAC. The subtraction routine subtracts the number at FAC from the one at ARG, and puts the difference at FAC. Some operations, such as the computation of the sine, use only the number at FAC, but also use a stack area in VDP RAM to store intermediate results. To accommodate those operations, we've set up a stack address (VSTACK) in VDP RAM using >1000 as the start of the stack.

1.72.1. Starting Up

The code section, which begins at label START, first sets the workspace pointer to our own workspace at label WS. Next it clears the word at >8374 (KEYADR) to insure that we're using key-unit zero. Next we set R0 to the value of our VDP Stack (>1000), then stash that number in RAM Pad at >836E, which serves as the pointer for stack use by some routines. Thus those routines will put stuff into VDP RAM at a place which won't interfere with our screen displays or character definitions.

1.72.2. Getting The Numbers

At label RESTR, set up for getting our first number input from the user. First we "point" R0 at row 1, column 4, then put a prompt on the screen. Now we BL @ACCEPT to allow user input. ACCEPT uses four data lines following the BL to determine its parameters. The first word after the BL determines the screen position for accepting keyboard input. In this case, that's row 2, column 3. The next data word is the field length, in this case 28 characters, which is actually more than enough for numeric inputs. The third data word is a signal to the routine that determines whether or not to clear out the input field before accepting input. In this case, it's 1, so the field will be cleared by the ACCEPT routine. Any number here other than zero will make the field clear. Zero will allow any previous content to remain in the field. The fourth parameter is of no importance in this case, but is the address of a block of memory to store the user's input as a string. Here, we've set that to the address of TEMSTR, a block of 30 bytes in our data section. The string placed there won't actually be used in this program, but there has to be a block of at least one byte more than the field length. This way, we can use the same ACCEPT routine for either strings or numbers.

After the four data words, we take the content of R0, and place it at location FAC12 (>8356). This happens just after we've exited from the ACCEPT routine. R0 at this point contains the VDP address of the first byte of the input field. Placing that address at >8356 is necessary to allow us to use the Convert String to Number (CSN) routine via XMLLNK. That routine examines the contents of VDP RAM starting at the address that we've put in FAC12 (>8356), and converts what's there into a floating point number at FAC. If what's there does not represent a number, FAC will contain zero in its first two bytes, meaning the input is regarded as zero. The routine CSN keeps reading the string until either it runs out of digits or it finds a character that's not part of the numeric set. For this routine, the numeric set consists of the numbers 0-9, the plus or minus sign, and the capital E (for exponent). Any other character found in the input string will be regarded as non-numeric, and will terminate the conversion routine. Thus if we put a number, like 2.135, into the 28 byte input field, the conversion routine will stop when it finds the space just after the 5. It will correctly report the number 2.135 into FAC in floating point format. (We explained that format last month.) The way we've written this program, the number input must be left-justified in the input field. Any leading spaces before the number starts will cause the conversion to yield zero.

Once this first number has been accepted, we want to save it to our own data memory, so that accepting another number can be done without losing this one. For that, we use a special little subroutine called MOVNUM. To use that, we load R9 with the address FAC, and R10 with the address NUM1, which is a block of eight bytes set aside in our Data Section. MOVNUM then copies the eight bytes from FAC to the block at NUM1.

All of this now repeats for the second number input, except that number gets copied into the block of bytes at label NUM2. Now the first math operation we want to perform is just to add these two floating point numbers. First we put a string on the screen to label this as the sum of the numbers, then we use MOVNUM to take the number we placed at NUM1 into the ARG block. The number at NUM2 is still present in FAC, so we don't need to put it there for the addition. Thus we have two floating point numbers at ARG and FAC, these being NUM1 and NUM2, respectively. We now use the ROM routine FADD through XMLLNK to add these two numbers. XMLLNK finds and executes the FADD routine, which places the result at location FAC as eight bytes. Now we clear an 18 character area just after the label on the screen, and then use our subroutine DISNUM to display the number taken from FAC. The subroutine DISNUM uses a GPL routine called Convert Number to String (CNS) through GPLLNK. That routine takes the floating point number at FAC and converts it to a string located in RAM Pad. When that routine exits, the byte at FAC12 (>8356) contains the length of the string, and the byte at FAC11 (>8355) contains the low order part of the address at which the string is to be found. To get this string on the screen, we take the byte at FAC12 into R2, right justify it, and then take the byte from FAC11 into R1, right justify that, then add >8300 to R1 so it points at the string's location in RAM Pad. The desired address on the screen is already in R0, so a simple VMBW operation puts the number string on the screen at the correct location. The string will always have a length of at least one, so we needn't check for a zero length string. If the number was zero or positive, there will be a space in the first character of this string. If the number is negative, the first character will be a minus sign.

1.72.3. Other Math Operations

We proceed now to perform other math operations on the two numbers we accepted. They have been left unsullied at the locations NUM1 and NUM2 in memory, so we can re-use them at will. In all cases from here on, we have to assume that whatever was left at ARG and FAC have been corrupted, so our first order of business before any more math operations is to use MOVNUM to place NUM1 at ARG and NUM2 at FAC. Remember that for subtraction, the number at FAC gets subtracted from the number at ARG. Thus when our FSUB finishes (via XMLLNK) FAC will contain ARG-FAC. We go through this process a couple more times, putting the product of NUM1 * NUM2 on the screen, and the result of NUM1 / NUM2 on the screen.

The next to last operation we perform is to take the Sine of the number at NUM1. This computation uses a routine in GROM, so we have to use GPLLNK instead of XMLLNK. In our source, we've included the Warren/Miller GPLLNK, mainly to avoid the well known problems that TI's GPLLNK presents. We don't have to put anything into ARG in this case, but just put the number from NUM1 into FAC. This routine uses that VDP stack we mentioned at the beginning, putting "God-knows-what" into the stack as intermediate results. When it's finished, there's a floating point number at FAC that equals the Sine of whatever number (in radians) was at FAC when we called the GPLLNK routine. This result is always a number between -1 and 1, inclusive.

The final operation is a comparison of the two numbers. For this, we place NUM1 at ARG and NUM2 at FAC as usual. After the XMLLNK performs the FCOM routine, we have to examine the GPL status byte (>837C) to see the result. We check first to see if the numbers are equal, since that's the easiest test. We just do CB @STATUS,@ANYKEY, and if those two bytes are equal, so are the numbers at FAC and ARG. If they're not equal, we put the STATUS byte in a register (e.g. R3), then strip off all but the >4000 bit. If that result is not zero, then the number at ARG was greater than the one at FAC. If the result is zero, given that the numbers are not equal, then the one at ARG must be less than the one at FAC. The code in the Sidebar performs just this way, and puts one of three messages on the screen to indicate the relationship between NUM1 and NUM2. One can also test for a "logical high" relationship using the >8000 bit, but we can't see any sense in doing that for floating point numbers.

1.72.4. Variations You Can Try

In your own work, you might want to try out the idea of allowing leading spaces to be present in the input field. This might come in handy if, for example, a default positive number string were in the input field to start with. Such a string starts with a space, which must then be skipped over after an ACCEPT operation. The following discussion assumes that our own version of ACCEPT, as included in last month's Sidebar, is being used. Among other things, that means that the length of the string typed in the field, excluding trailing spaces, is in R2 upon return from the routine. We'll show here the code that would allow your routine to skip over any leading spaces.

```
            BL   @ACCEPT            Use Accept subroutine
            DATA 32*5+2             Row 6, Col 3
            DATA 28                 Field Length 28
            DATA 0                  Don't Clear field
            DATA TEMSTR             String Buffer
READ1       BLWP @VSBR              Read byte from field
            CB   R1,@ANYKEY         Is that a space?
            JNE  GNUM1              If Not, jump
            INC  R0                 Next spot on screen
            DEC  R2                 Dec string length count
            JGT  READ1              If positive, repeat read
GNUM1       MOV  R0,@FAC12          Put R0 at >8356
            BLWP @XMLLNK            Use XML linkage vector
            DATA CSN                Convert string to number
```

This method will work even if the field was left blank. In such a case, after the DEC R2, R2 will become a negative number, and the JGT test will fail, so the CSN routine will be used right away, and will report zero at FAC. No doubt some of our readers will find a more efficient way to do this, but the way we've shown is certain to work. Each time a leading space is found in a non-null entry, the pointer in R0 advances one spot and another read is done until a non-space character is found.

1.72.5. The "Caret" Case

We've fooled around with the other operations allowed through GPLLNK, and found that other functions work as given in the E/A Manual, with one notable exception. That exception is the "Raise number to power" routine (A ^ B in BASIC or XB). The E/A Manual says that you can use this routine, which we call the "caret" routine, by placing the first number at ARG, the power to which it's to be raised at FAC, then using GPLLNK. This doesn't seem to work! The routine seems to lose its way somewhere along the line, returning to our code with a meaningless number placed in FAC. Here then is another plea to our readers. If any of you has discovered some trick to make the "caret" routine work from Assembly code, please let us know, and we'll pass that on to others.

We hope these two articles will satisfy your hunger on the subject of floating point numbers for a while. Next month's topic is undecided at present, but since we're now writing more than a year ahead of publication, we've got plenty of time to decide on that topic.

1.73. The Art Of Assembly — Part 73. Where Is Bryn Mawr?

By Bruce Harrison

Depending on which side of the Atlantic you're on, it's either in South Wales or on Philadelphia's "Main Line". Of course in our case it's a trick question, because today's column deals with the subject of a Word Search program, not directly with Geography. As we'll explain, however, this program could be used to help in teaching that subject.

This project was the direct result of a call for help from Ed Morse, of Phoenix, AZ. Ed had been trying to modify an old TI Basic program which generated Word Search puzzles to suit his own needs. For those unfamiliar with the concept, a Word Search puzzle (or Word Find puzzle) is a rectangular matrix of letters in which words are "hidden". The words can run horizontally, vertically, or diagonally, in a total of eight directions through the matrix. There are also letters in the matrix which don't belong to any of the words.

Ed Morse was trying to use this old Basic program to create puzzles containing the names of members of his family. He had 108 names in all, and the matrix of only 25 by 25 was nowhere near large enough for that purpose. Further, the Basic program required him to type in each name on each run. That's one whale of a lot of typing! Ed had expanded the original program's matrix, but still found that not all the names would fit.

Our first problem was to deal with all that typing on each run. Why would anyone design such a program to require typing in each name each time? Perhaps this was done for a "console basic" case in which no disk drive was available. Our first order of business, then, was to fix the input section so that one could have the names in a D/V 80 file, which only needed to be typed once, then have the program read the names from the file. We made that and a couple other small changes, making the matrix 32 by 32 letters, and passed a copy back to Ed Morse. Still, this Basic program would not always fit all of his 108 names into the puzzle, and it was painfully slow in trying.

1.73.1. Assembly To The Rescue!

Having failed to completely understand why the Basic program couldn't do the job, we set out to make an Assembly version that would. To do this, we "borrowed" source code from some of our other programs, then devised our own algorithm for placing the words. Like the Basic version, ours used random numbers to decide in which direction each word would be placed. Since there are eight such directions, the random number had to range only from 0 through 7. That much was easy. We devised our scheme so that 0 meant straight across left to right, 1 meant downward and right, 2 meant straight downward, and so on through 7, which meant upward and to the right. For each of those directions, we would establish a "last row" and "last column" for placement, based on the length of the word being placed. In other words the program would not try to place the word where there was not enough room. But we're already getting ahead of the story.

1.73.2. The Matrix Structure

In the special case for Ed Morse's family, the matrix has to have 32 rows of 32 columns. In our Assembly version, this is simply a block of 1024 (32 times 32) bytes which are all set to 0s before we start placing words. To place a word, we start with its first letter and a starting point in the matrix. If that element of the matrix is a zero or if it's got a matching letter, then we proceed to examine the next appropriate spot in the matrix. This continues until we've found the end of the word's length or we've found a letter that doesn't match the word. If we got through the whole word, then the letters of that word are placed into the matrix, and we can move on to the next word. If that word wouldn't fit in that place, we try another starting point and repeat the process. If we find that this word can't be fitted in that direction, then we go back and try the next direction. We make eight direction trials, so the word has every possible chance to be placed. Each time a word is fitted into the matrix, it gets added to a list of used words in memory. This list gets used later by the program, to print out the list of words to be found. The used list is kept separate from the list as read in from the disk. That's important in case there are one or more words that won't fit into the matrix. In this "special for Ed Morse" version, that wasn't a problem, as all 108 names would fit nicely into the 32 × 32 matrix.

1.73.3. The Public Domain Version

For the more general-purpose release version, we used the original 25 × 25 matrix size as in the old Basic program. The program starts by showing a title screen, which stays on screen until a key is pressed. The delay between startup and the pressing of a key is used to "seed" the random number process. Once a key has been pressed, the user is asked to supply a file name. There are a number of "ready made" word files supplied on the disk, so the user can try out the program immediately. Each is a short D/V 80 file (3 or 4 sectors) with words related to a single topic. For example, the file ANIMALS contains the names of animals, COUNTRIES contains the names of countries from around the world, FOODS contains names of things we eat, and so on. Each file contains between 50 and 70 words, depending partly on the length of words involved. Files with many long words may have as few as 50 entries, while those with mostly shorter words may have 70 or more entries.

Given a complete file name (e.g. DSK1.FOODS), the program proceeds to open that file and read each word. As they're read, the words appear on the screen and scroll up quickly. When the end of the file is found, the program reports on the screen how many words were found, then puts the legend "PLACING THE WORDS" on the screen. In the next couple of seconds, all of the words get placed into the matrix. Compared to the BASIC version, which takes a very long time to place words, this is practically instantaneous.

1.73.4. The First Eleven

Just to make the puzzles a bit tougher to solve, we take additional random numbers for the first eleven words in the list. Each of them gets a randomly chosen row and column for its starting point. Thus these eleven words are, in general, more "hidden" than the others. After the first eleven are placed, the others are fitted in according to an algorithm that places their initial starting points based solely on the direction the word is to go. For example, if a word is being placed down and right, its first trial placement will be at the upper left corner. If up and left, its first trial position will be at the lower right corner, and so on. The program then cycles through all possible positions where this particular word can fit, until either the word is placed or it's found that there's not room for this word in this direction. The program will then try placing this word in seven other directions, and will usually find a place for it. Letters placed in the matrix can of course belong to more than one word where they match up. In many cases, for example, you may find a C at the upper left corner that's the first letter of three words, one going across left to right, one going straight down, and one on the diagonal down and right. The same sharing may happen anywhere, as words going in different directions may share a letter where they cross over each other. In most cases, all of this gets done in about two seconds, then the puzzle is almost ready for printing.

The last step in forming the matrix is the "random fill" process, in which the program examines each of the 625 bytes in the matrix. For each one that still contains zero, the program inserts a randomly chosen upper case letter. This takes essentially no time because of the rapid and efficient way the program generates random numbers.

1.73.5. The Exceptional Case

Sometimes a word list will contain long words toward the end, and it's possible that one of these won't be able to be placed because earlier placements have not left a suitable spot for this word. When that happens, the program stops and offers the user three options on which to proceed. Option 1 is to re-start placement. Option 2 is to skip this particular word and continue with the next one. Option 3 is to just stop and print the matrix with those words already placed.

If the user chooses Option 1, he actually has two possible ways to go after making that choice. The program asks whether the user wants to make a new try with the same data. If yes, the program re-clears the matrix to all zeros, then starts the placement process again, using the word list that's already in memory from the file. This is why the list of "used" words is kept separate from the original list read from the file, so that the placement process may start again without re-reading the file. If the user answers this "same data" prompt with any keypress other than **Y** or **y**, the program goes back to the file name input prompt so a different file may be tried.

Usually even a "difficult" word list can all be placed in one or two more tries, simply because a different set of random numbers creates a new situation for each placement attempt. On rare occasions a word list may be found that may take five or six tries to get placed. When that happens to us, we go back and re-edit that word file, leaving fewer or shorter words in the list.

1.73.6. The Printing Process

Once the program has filled the matrix in memory, we're ready to print. The user gets a prompt for the printer file name, with the "PIO" option already in place as a default entry. If that's the user's desire, just pressing **ENTER** will start the printing process. When we first distributed a few test copies of this program, we had good reaction from almost every tester, but got one surprise answer. That was from John H. Bull, of Knoxville, TN. Seems what he wanted was to make many identical copies of just one puzzle. To do that, he saved the puzzle to a disk file instead of sending it to the printer. In that early version, this idea worked, but the resulting disk file contained unwanted carriage returns and line feeds. In the latest update, that condition has been corrected, so that if the puzzle is sent to disk, no carriage returns or line feeds will be included.

The matrix itself gets printed centered in the eighty columns on the paper, with a space between each letter going across the paper. An escape sequence gets sent out before the matrix, to set the line feed so that the spacing between letters will be the same in both horizontal and vertical directions. That makes it easier to find words along the diagonals. As we all know by now, setting a different line feed amount for 9 pin printers and for 24 pin printers is a tricky process. Thus on the Public Domain release disk there are two versions of the main program, so that the disk can be used with either type of printer and gives satisfactory results in both cases. Another escape sequence is sent before the word list gets printed, to set the vertical spacing back to the normal 1/6 inch. These escape sequences are also included if the file goes to disk, so that a puzzle saved to and then printed from disk will look no different from one printed directly from the program.

The list of words gets printed in columns, after the program determines how many columns are needed and how many words are to be included in each column. Between the matrix and the word list, the program prints a line which says "FIND THESE XX WORDS IN THE PUZZLE ABOVE". The XX is of course the number of words actually placed in the matrix, which may not be the same as the number of words in the original word file. (Remember that the user can skip over words that didn't fit.) This way, the user is assured that all words printed in the list are actually somewhere in that puzzle above, hard though it may be to find them. On rare occasions, "included" words may get placed over a longer "including" word. Suppose that a list included both the words JONES and JONESBORO. It's possible, though unlikely, that the word JONES will get placed, then JONESBORO placed right on top of that, so that nowhere in the puzzle will JONES appear by itself. In hundreds of test runs with various word files, this has only happened to us once, but be warned that it can happen, and can drive the solver crazy.

1.73.7. The Finishing Touch

After the matrix and word list are printed, the program places the name of the original word file at the bottom of the sheet. This allows the user to quickly determine what kind of words he's looking for, as well as just knowing where the list came from. For example, one wouldn't be looking for ROBIN or MEADOWLARK if the file name was CARS. Of course there are some cases where a word may appear in more than one list. COUGAR, for example, is in the list file called CARS, and also in the one called ANIMALS. There are 21 word files supplied on the disk, including many topics. They range from ANIMALS through TREES, and include such things as FOODS, SPORTS, etc.

1.73.8. The Geography Lessons

Among the files supplied are ones that could be used as part of Geography lessons. For example, there's a file called CITIES, which includes names of cities large and small in the United States. For our friends overseas, there are ones called CANCITY (for Canada), BRITCITY (for the U.K.), and IRELAND and GERMANY for those countries. There are also other Geography oriented files such as LAKES (of North America), RIVERS (of the World), and MOUNTAINS (of the world). Using these, a student could first find the words in the puzzle, then go to the Atlas and find the locations. Alex Trebek [of the TV show Jeopardy] would be proud of you for doing this!

Just for the fun of it, there are files such as FOODS, which may have the side effect of making you hungry, and SPORTS, which includes esoteric ones like curling, biathlon, and so on. Of course this is all intended just to spark your own imagination. John Bull, for example, made a file of words for his wife's hobby of making miniatures. Over in Dorset, England, Mr. John Murphy made files of towns in England, Scotland, and Wales.

You'll notice that there's no Sidebar this month. Mainly that's because the whole program is much too large to publish all its source code, but also because nothing exotic was done that hasn't been covered before in this column. The complete source code is included on the Public Domain disk for those who care, although it's not completely annotated. As always, the disk containing this program has been made available through the Lima user group, so it's accessible to the whole community. Contact Dr. Charles Good at P.O. Box 647, Venedocia, OH 45894.

1.73.9. Extra Added Attraction

When we are making up word files for use with this program, we don't necessarily think up words in alphabetical order, but that's how we want them in the file. Thus we went back a long way in this series (to Part 24) and found the source for our File Sorter. We modified that slightly, and saved it as an Option 5 program file called simply SORT. SORT is supplied on this P.D. disk, so that once you've made a file of words, you can sort that file in a couple of seconds using this SORT program. The file gets sorted while it's being read from disk, so that as soon as the file closes on input, it's ready to be saved back to disk in sorted form, and its existing file name is already there in the input field for OUTPUT NAME. This sort program can actually be used for other purposes, as it will correctly sort any D/V 80 file that's 100 sectors or less in length.

The limits on word files are fairly simple. The program will accept words of any length from 2 through 20 letters. If there are blank lines in the file, the program will ignore them, as it will words longer than 20 or shorter than 2 letters. All words should be in Upper Case only, but the program doesn't check that. This could be a way to cheat, by using word files all in lower case, as the random fill letters are all upper case. (We don't recommend cheating, but it is possible.)

Next month we'll open a whole new world, concerning use of color printers to print TI-Artist Picture files. See you then.

1.74. The Art Of Assembly — Part 74. A Colorful Experiment

By Bruce Harrison

As we promised in our last column, today we take up a new subject, that of printing in color from TI-Artist Picture files. The development of a program to perform that function was no easy task, partly because we have no color printer here at the Harrison residence on which to test results. Thus we express sincere thanks to Lew King of Industry, PA and to Gary Cox of Memphis, TN for their help with this project. Lew and Gary were the "guinea pigs" for this development, and thanks to them we have two versions of the TIACOLOR product, one for use with 24 pin dot matrix color printers, and one for use with Canon Bubble Jet color printers.

The idea started when Lew King mentioned that his printer was a Star Micronics 24 Pin color unit. He sent along the list of escape codes and their corresponding colors. That happened to match exactly the color control codes from a Panasonic printer manual which we have courtesy of Harley Ryan. Given this match, it occurred to us that this might be common to all "Epson Compatible" color models. Our first step, then, was to make up a short test program in Extended Basic, then ship that off to both Lew King and Gary Cox. Results were similar, except that Gary's Bubble Jet model makes a better red than the impact model makes.

1.74.1. How Do They Work?

In the impact case, the ribbon has four colors, these being magenta, cyan, yellow, and black. In the Bubble Jet case, there are four separate inks in one cartridge, with those same three subtractive primaries plus black. Taken with combinations, these four colors result in a total of seven colors available (including black). Red is made by combining magenta and yellow, green is made by combining cyan and yellow, and blue by combining cyan and magenta. These same four colors are used to print color glossy magazines and such. Results of our test with Extended Basic showed that the red on the impact printer was more like orange, and its blue more like purple. On the Bubble Jet, the "combined" colors red and blue were just about what we'd expect. Apparently Canon uses very high quality inks.

1.74.2. The Mapping Problem

On its screen, our faithful old TI produces sixteen colors. Sixteen is more than seven, so we had to devise a "mapping" scheme so that these sixteen colors would reduce gracefully to seven for the printer. Black was of course no problem, and white is made by printing nothing, but the TI has three shades of red and three of green, two of blue and yellow, and so on.

Thus was born a lookup table, whereby the colors on the TI are mated up and combined into seven colors for the printer. This of course limits the accuracy of color rendition, so that things which are in different shades of red or blue will appear exactly alike on paper. Also, without getting into half-tone rendering, we allowed gray to simply appear as white.

There are two tables used, one of which has the numbers of screen colors which map to each printed color, while the other has the screen color codes for comparison. Since we're going to have to do a lot of comparing and moving of bytes from both the pattern and color tables, our program first copies those tables from VDP RAM into the 32K memory. This is done by two VMBR operations before the actual printing starts.

1.74.3. Multiple Passes

In order to print a Bitmap image in color, we must examine each pixel first for its color, then to see whether that pixel is on or off. The colors for the printer start with black, so that's the first color in our lookup table. We start at the bottom left corner of the image, and get the color byte that corresponds to this byte. Since we're looking at the foreground color in this first pass, we take the color byte into a register and then shift it four bits to the right. so that the left byte of the register contains the foreground color for the byte. Now we compare that to the first byte in our lookup table (01), and if it matches, then we'll examine the corresponding byte from the pattern table. In other words, if this pixel's foreground color is not a match, we ignore this byte entirely. If the color matches, then we examine the high bit of this byte, and if that's a 1, we save this byte in our temporary storage for printing. We proceed in like manner until each pixel in the left-hand dot-column has been accounted for, then send out ESC "r" 0 to set the printer for black and print this one dot-column to the paper. Note that after printing this one column in black, we issue a carriage return without line feed to the printer, because we have six more colors to print in this same dot-column.

The next color in the map is magenta, which corresponds to ESC "r" 1 control sequence. As with black, the TI has only one shade of magenta, so this is a one pass process, working on that same dot-column from the screen. As before, each color byte is examined for a match with magenta, then those bytes that have the most significant bit turned on get saved in our temporary buffer. When we've again finished this dot-column, we set the printer for magenta and print anything whose foreground was magenta and left bit 1. When we reach colors like blue, there are two such matchings done, one for each shade that we'll print as blue. All of this gets repeated until we've done all seven printer colors for this one dot-column. When that's done, we're ready to repeat the whole process again, but this time examining the background color each time, and printing with the pattern byte inverted, so that we print something only if the background color matches and the left bit was a zero, which became 1 when inverted. Like the foreground, this background printing cycles through each of the seven colors with just a carriage return sent to the printer on each pass. Finally, after the background colors have been printed for this first dot-column, we're ready to send a special linefeed and start processing the next dot-column of the picture.

To do this, we access the very same bytes from the stashed color and pattern tables, but we shift each byte left by one bit, so that we're processing the second dot-column. We perform the same processing, moving through the colors in both foreground and background. In like manner, we take these same bytes six more times, shifting by two through seven bits, until all eight bits of each byte have been printed. This completes one of the 32 character columns in the picture.

All of the above gets done 32 times, which completes one picture. Then and only then do we send a form feed and reset code to the printer, so it's back in its "normal" powerup condition, ready for whatever's next from the computer.

1.74.4. Today's Sidebar shows. . .

The Sidebar is only a "snippet" of the source code, plus a portion of the data section. The file is called PRNMODE, and as you may guess, this is the part of the code that takes the picture data from memory and processes it for output to the printer. Although it's not immediately obvious, the bulk of this code is a set of nested loops. The nesting is intricate, to say the least, because various start and end points overlap each other depending whether we're at color, bit, or byte boundaries in the processing.

In this code we have used an instruction that we've never used before in this series, and we learned something about this instruction's use. The instruction is SLA R7,0 just before the label ANDI7 in the source code. This instruction means that the content of register 7 gets shifted left by the number in the low nybble of Register 0. What we learned is that this doesn't work if R0 contains 0. Thus when we move the data word SFTQTY into R0, we skip over the SLA instruction if the data word was zero, which happens when we're processing the high order bit in any byte. By incrementing the number at label SFTQTY, we walk through the bits in each byte for the printing process. The data word at SFTQTY gets incremented after each dot-column just after label CKBIT. Thus it ranges from 0 through 7 as the counter BITCNT decrements from 8 to 1. When one character column has been completed, the code loops back to label PRNST2, at which the lookup table indexing re-starts and SFTQTY goes back to zero.

1.74.5. The Public Domain Disk

The disk containing this software has been released and made available through the usual channel, the Lima Users Group library. On the disk are two versions of the program file, one for 24 pin impact color printers, and one for the Bubble Jet type. They are called COLOR24 and COLORBJ, respectively. The source code supplied on the disk is for the 24 pin version only, so that users won't get confused. If any of our readers would like to see the source for the Bubble Jet version, just request that disk from yours truly.

1.74.6. Caveat Emptor

While we were pecking away at the source code for this product, we got a plea from another TI owner in Pennsylvania, concerning the use of our Drawing program. This user had obtained a copy of that Public Domain disk, and was trying to use it with his new Hewlett-Packard Deskjet printer. He was getting "total nonsense" results, so he sent along a copy of the escape code pages from its manual. These sequences bore no resemblance at all to the sequences we're used to finding on the "Epson Compatible" printers that most TI owners have. Even the words used to describe the actions of these escape sequences were very different from the descriptions used in the usual printer manuals. What, for example, does "Raster Graphics" mean? Thus we were unable even to understand what the sequences would accomplish. We advised this user to look carefully in the manual and on the printer to see whether the printer had a Dip Switch setting (as do many printers) to put it into an Epson Compatible mode. NO SUCH LUCK!

The escape sequences used by his Hewlett-Packard printer are in a whole new "language" called PCL, which HP considers the new standard for printers. It seems that HP decided that all users would have modern PC computers with Windows capability and the ability to install special drivers for this new language, and that thus there was no need for catering to the older crowd (like your author) who still depend on using Epson-type escape sequences to control printers. Caveat Emptor is Latin for Buyer Beware, and that certainly applies in this case. If you plan to buy a new printer, we recommend you avoid Hewlett-Packard like the Black Plague. Our fear, though, is that other makers of printers will follow HP's lead in this matter, so we'd better keep our "Epson Compatibles" in good condition, else give up using a printer with our TI for anything but straight text applications. When I purchased the Canon Bubble Jet model that I now use, I first checked the description carefully to be sure that "Epson Emulation" was among its features. That's our best advice to anyone buying a new printer.

Our topic for next time is undecided. It will in all likelihood depend on what problems readers throw our way, so if there's something you'd like to see, just ask.

```
* SIDEBAR 74
*
* PRINTER OUTPUT SECTION
* 24 PIN IMPACT COLOR VERSION
*
* CODE BY Bruce Harrison
* PUBLIC DOMAIN
* NOTE: THIS IS NOT A COMPLETE PROGRAM
*
PRNOUT LI   R1,PPABDT    PRINTER PAB DATA
       BL   @FILOP       OPEN THE FILE
       JNE  PRNSET       IF SUCCESS, JUMP
       BL   @CLOSE       ELSE CLOSE
       BL   @BLNK        BLANK SCREEN
       BL   @SETGM       BACK TO GRAPHICS
       BL   @CLS         CLEAR SCREEN
       BL   @UNBLNK      UNBLANK
       LI   R1,PNAMSG    PRINTER NOT AVAILABLE
       BL   @ERRRPT      REPORT ERROR
       B    @SAVE0       THEN BACK TO MAIN PROGRAM
PRNSET CLR  @BGFLG       CLEAR BACKGROUND FLAG
       LI   R0,PRNBUF    PRINT BUFFER
       LI   R1,RSTSTR    RESET STRING (ESC '@')
       BL   @DISSTR      PUT IN VDP BUFFER
       BL   @PRNSND      SEND TO PRINTER
PRNST1 LI   R5,32        32 COLUMNS OF PICTURE
       LI   R12,>1707    START AT LOWER RIGHT CORNER
*
* LABEL PRNST2 IS THE START OF ONE CHARACTER COLUMN
* EACH CHARACTER COLUMN HAS EIGHT DOT COLUMNS
*
PRNST2 LI   R9,MAPLUT    COLOR MAPPING
       LI   R13,QTYLUT   NUMBER OF MAPPED COLORS
       CLR  @SFTQTY      START WITH NO SHIFT
```

```
       MOV  @EIGHT,@BITCNT EIGHT BITS PER CHAR COLUMN
*
* LABEL PRNOTL IS THE START POINT FOR ONE DOT COLUMN
*
PRNOTL MOV  R12,R3       PUT ADDRESS IN R3
       MOV  R3,@SAV3     STASH AWAY R3
       MOV  @SEVEN,@CLRCNT SEVEN COLORS
       MOVB @PPABDT,@COLSTR+3 START WITH BLACK (27)"r"(0)
*
* LABEL NXTCLR IS THE START POINT FOR ONE PRINTED COLOR
* IN ONE DOT-COLUMN
* THE LOOP AT CLST1 CLEARS THE MEMORY BUFFER
*
NXTCLR MOV  *R13+,@MAPCNT NUMBER OF MAPPED COLORS FOR THIS PRINT COLOR
       CLR  R14          R14=0
       LI   R0,TEMSTR    TEMPORARY STRING
       LI   R2,192       192 COUNT IN R2
CLTS1  MOVB R14,*R0+     ZERO A BYTE
       DEC  R2           DEC COUNT
       JNE  CLTS1        RPT IF NOT ZERO
*
* LABEL LTEM10 IS THE START POINT FOR ONE SCREEN COLOR
* R9 POINTS AT ONE SCREEN COLOR BYTE IN A LOOKUP TABLE
* R4 COUNTS DOWN THE 24 CHARACTER ROWS TO BE DONE
*
LTEM10 LI   R10,TEMSTR   TEMPORARY STRING
       LI   R4,24        24 CHAR ROWS
       MOV  @SAV3,R3     PUT R3 BACK
*
* LABEL PRNMIL IS THE START A GROUP OF EIGHT DOT-COLUMNS
*
PRNMIL LI   R6,8         8 BYTES
*
* LABEL PRNMIL IS THE START OF PROCESSING FOR ONE PIXEL
*
PRNINL MOVB @PIXBUF+>1800(R3),R7 COLOR BYTE TO R7
       MOV  @BGFLG,R0    BACKGROUND PASS?
       JNE  NOSRL        IF SO, JUMP
       SRL  R7,4         ELSE MOVE FOREGROUND NYBBLE BY 4 BITS
NOSRL  ANDI R7,>0F00     MASK TO ONE NYBBLE
       CB   R7,*R9       COLOR MATCH?
       JEQ  GPB          IF SO, GET BYTE FROM PATTERN
       JMP  PDEC3        THEN JUMP
GPB    MOVB @PIXBUF(R3),R7 GET A BYTE
*      JEQ  PDEC3        IF ZERO, JUMP
       MOV  @BGFLG,R0    TEST BACKGROUND FLAG
       JEQ  MOVR7        IF ZERO, FOREGROUND PASS
       INV  R7           ELSE INVERT THE 1'S AND 0'S
MOVR7  MOV  @SFTQTY,R0   SHIFT AMOUNT TO REG 0
       JEQ  ANDI7        IF ZERO, SKIP THE SHIFT
```

```
       SLA  R7,0         SHIFT LEFT BY THE NUMBER IN R0
ANDI7  ANDI R7,>8000     MASK TO MSB ONLY
       JEQ  PDEC3        IF ZERO, JUMP
       MOVB R7,*R10      ELSE PUT BYTE IN TABLE
       INC  R14          AND INCREMENT R14
PDEC3  INC  R10          ADD ONE TO TABLE INDEX
PDEC3A DEC  R3           BACK ONE BYTE
       DEC  R6           DONE 8?
       JNE  PRNINL       IF NOT, JUMP
       AI   R3,-248      NEXT CHAR IN COLUMN
       DEC  R4           DONE 24?
       JNE  PRNMIL       IF NOT, REPEAT
       INC  R9           NEXT MAPPED COLOR
       DEC  @MAPCNT      USED ALL MAPPED COLORS?
       JNE  LTEM10       IF NOT, REPEAT FOR NEXT MAPPED
*
* R14 INDICATES WHETHER ANY PIXELS WERE FOUND FOR THIS PRINT COLOR
* IF NONE WERE FOUND, THEN WE CAN SKIP SENDING ANYTHING TO PRINTER
*
       MOV  R14,R14      CHECK REG 14
       JEQ  ADDBYT       IF ZERO, NO NEED TO PRINT
       LI   R0,PRNBUF    POINT AT PRINT BUFFER
       LI   R1,COLSTR    COLOR ESCAPE STRING
       BL   @DISSTR      PLACE IN BUFFER
       BL   @PRNSND      SEND TO PRINTER
*
* CODE STARTING AT PLRTS SENDS BIT GRAPHICS DATA TO PRINTER
*
PLRTS  LI   R4,4         FOUR GROUPS
       LI   R0,PRNBUF    POINT AT BUFFER
       LI   R1,BGRSTR    BIT GRAPHICS CONTROL STRING
       BL   @DISSTR      PLACE IN BUFFER
       BL   @PRNSND      SEND TO PRINTER
       LI   R10,TEMSTR   POINT AT BYTES FOR 1 COLUMN
PRLBUF LI   R0,PRNBUF    PRINT BUFFER
       LI   R2,48        GROUP BY 48
PRMV1  MOVB *R10+,R1     GET A BYTE
       ANDI R1,>8000     MASK ONLY MSB
       SRA  R1,2         REPLICATE IN THREE MSBS
       BLWP @VSBW        WRITE THAT
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE AGAIN
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE AGAIN
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE AGAIN
       INC  R0           NEXT ADDR
       BLWP @VSBW        WRITE 5TH TIME
       INC  R0           NEXT ADDR
       DEC  R2           DONE 48?
```

```
       JNE  PRMV1        IF NOT, REPEAT
       LI   R2,240       240 BYTES IN BUFFER
       BL   @PRNSND      SEND THOSE TO PRINTER
       DEC  R4           DONE 4 GROUPS?
       JNE  PRLBUF       IF NOT, ANOTHER GROUP
*
* CODE AT PRCRLF SENDS ONLY A CARRIAGE RETURN TO PRINTER
*
PRCRLF LI   R0,PRNBUF    PRINT BUFFER
       LI   R1,CROSTR    CR ONLY (NO LINE FEED)
       BL   @DISSTR      PLACE IN BUFFER
       BL   @PRNSND      SEND
*
* ADDBYT ADDS ONE TO PRINTER COLOR ESCAPE CODE
*
ADDBYT AB   @BYTONE,@COLSTR+3 SET NEXT PRINTING COLOR
       DEC  @CLRCNT       DEC COUNT OF COLORS PRINTED
       JNE  NXTCLR        IF NOT ZERO, NEXT LOOP
       LI   R9,MAPLUT     RESET MAP LOOKUP POINTER
       LI   R13,QTYLUT    AND MAPPED COLORS POINTER
       MOV  @BGFLG,R0     CHECK BACKGROUND FLAG
       JNE  CKBIT         IF NOT ZERO, JUMP
       INC  @BGFLG        ELSE SET BGFLG
       B    @PRNOTL       THEN START BACKGROUND PRINT
CKBIT  CLR  @BGFLG        CLEAR BGFLG FOR NEXT PASS
       INC  @SFTQTY       ADD ONE TO SHIFT QUANTITY
       LI   R0,PRNBUF     POINT AT PRINTER BUFFER
       LI   R1,CRLSTR     CARRIAGE RETURN AND SPECIAL LINE FEED
       BL   @DISSTR       SEND TO BUFFER
       BL   @PRNSND       THEN TO PRINTER
       DEC  @BITCNT       EIGHT DOT COLUMNS DONE?
       JEQ  PRDEC5        IF SO, JUMP AHEAD
       B    @PRNOTL       ELSE BACK FOR NEXT DOT COLUMN
PRDEC5 AI   R12,8         NEXT CHAR COLUMN
       DEC  R5            DONE 32?
       JEQ  PRFF          IF SO, AHEAD TO FORM FEED
       B    @PRNST2       IF NOT, CONTINUE
PRFF   LI   R0,PRNBUF     POINT AT PRINT BUFFER
       LI   R1,FFSTR      FORM FEED/RESET
       BL   @DISSTR       PUT IN BUFFER
       BL   @PRNSND       THEN TO PRINTER
PRNCLS MOV  @PABLOC,R0    PAB LOCATION
       MOVB @ONE,R1       CLOSE OPCODE
       BLWP @VSBW         WRITE THAT
       BL   @FILOP3       CLOSE THE FILE
       B    @SAVE0        BACK TO MAIN PROGRAM
*
* DATA - AN EXCERPT
*
QTYLUT DATA 1,1,1,2,2,3,3
```

```
MAPLUT BYTE 1,>0D,>07,>04,>05,>0A,>0B,>06,>08,>09,>02,>03,>0C
BGFLG  DATA 0
COLSTR BYTE 3,27,'r',0
FFSTR  BYTE 3,12,27,'@'
RSTSTR BYTE 2,27,'@'
LSPSTR BYTE 3,27,'+',5
BGRSTR BYTE 4,27,'L',192,3
CRLSTR BYTE 5,13,27,'+',15,10
CROSTR BYTE 1,13
SPLF   BYTE 5,13,27,'+',120,10
MARSTR BYTE 3,27,'@',13
```

1.75. The Art Of Assembly — Part 75. Reading Disk Sectors

By Bruce Harrison

This time we're into some really deep stuff, reading things directly by sectors from disks. Normally, of course, when we deal with files on disks, we let the drive controller handle all the hard stuff, like finding the contents of the file and putting records into a buffer in VDP RAM. Why, then, should we have to read a disk sector by sector? To help a friend in need is the answer.

Back in 1996, your author looked at the video tapes from the MUG conference in Cleveland. On those was a lecture and demo by Mickey Cendrowski, showing her Load Master program. This program was one of those inspired by Mickey's own need for some way to make sense of the West Penn User Group's disk library. She wanted, among other things, to have a program that would identify clearly many different file types that are available for the TI. The program was written in Extended Basic, and performed very well but slowly. The biggest problem seemed to be that files of the Program (a.k.a Memory Image) type had to be lumped into large categories because there's no way in Extended Basic to tell the difference between E/A Option 5 files and those created by Basic or Extended Basic. Mickey had done the best she could, but for program files under 34 sectors in size, Load Master could not determine what was Basic or XBasic and what was E/A.

1.75.1. Headers The Answer

The difference between Basic/XB programs and E/A Option 5 programs is in the content of the file header. This is the first six bytes of the file's content. When either Basic or E/A loads a "program" file, the loader examines this header information, and can tell if the wrong type is being loaded. That idea wouldn't be useful, however, since trying to load the program file would destroy the Load Master program itself.

The only way to read the header in a controlled manner was to read by sectors from the disk. The end product, then, would have to be able to find the directory sector for each file on the disk, find from that the sector number of the file's header, read that sector, and then examine its content.

1.75.2. The Keys To The Disk

The keys that unlock all this are in Sector 1 on the disk. In that sector are the sector numbers for the directory sectors of all the files on the disk. Sector 1 contains up to 127 words (two bytes each) that give the sector numbers for the files on the disk in sorted order. That is, if two files are named AAA and AAB, the AAA file's directory sector number will come before that for AAB in sector 1. When new files are put onto a disk, the contents of sector 1 get sorted so that the new file is in the correct place by an ASCII sort. Unlike some other things we'll get to shortly, there's no mystery about reading the sector numbers in sector 1. Each pair of bytes, read as a word value, is the number of the sector containing the directory information for one file. If that number is zero, it means we're past the number of files stored on this disk.

1.75.3. Some Assembly Required

Like it says on the boxes at Toys R Us, some Assembly is required to actually access the directory sectors and then the files' headers. It's impossible on the TI to read disk sectors from Extended Basic, except of course by using CALL LINK to an Assembly routine which reads the sectors. It was obvious that Mickey's program would need such capability, so your author offered to help in that effort. Mickey sent me the Version 2.1 disk, and away we went. There were some Assembly routines in Version 2.1 already, embedded by Todd Kaplan's ALSAVE method. It became obvious fairly quickly that the amount of Assembly stuff that would need to be added would exceed the space available in low memory, so "method two" became the method of choice. This means that we keep a program called LOAD on the disk, but it only places the previous Assembly routines (plus one) into low memory before RUNning another program called LOADMASTER. That second program contains the Extended Basic stuff plus a lot of Assembly code embedded via Harry Wilhelm's High Memory Loader. This way, we get to have two sets of Assembly routines in use at the same time. The "old" routines plus Boot Tracking are kept in low memory, while the routines to read the catalog by sectors and identify various file types sits in High Memory along with the LOADMASTER program. There was still some leftover space in Low Memory, and some of that was used for temporary storage by the routines in High Memory. Thus we've made very efficient use of the whole expansion memory.

1.75.4. Thanks To Travis Watford

Through our friend Barry Traver, we had a disk in our collection that contained Travis Watford's T-Shell source code. Among other things, that source code contained a complete DSRLNK and the code to read sectors in an Extended Basic environment. We had to modify Travis' DSRLNK slightly for our purposes, but the code that actually reads the sectors is largely his. In today's Sidebar are portions of that code as modified. Travis' DSRLNK as modified is a very general-purpose one, which can be used in any environment and can perform linkage to just about any device service routine. Thank you, Travis!

The device service routine that we're using to read the sectors is of the "call" variety, in that the BLWP to DSRLNK is followed by DATA >A instead of the usual DATA 8. The PAB used is just two bytes in length, that being one byte of 1 (length) and a byte of >10 to call the sector service. The specifics as to the sector number, whether to read or write, etc. are placed in specific locations in RAM Pad before the DSRLNK call. The result of the call (barring error) is a dump of 256 bytes at our chosen buffer location in VDP RAM. We found it less troublesome to put both the PAB and the Buffer in the area above >37D7 in VDP. This way it doesn't get in the way of any other file accesses, nor does it interfere with the use of VDP RAM for the lookup tables and string variables that XB puts there. In other words, we found a "safe area" in VDP RAM to do our sector reads.

Before actually trying to integrate our Assembly stuff into Mickey's program, we ran a series of tests on this process, and found a real problem that we hadn't anticipated. In our system we have two "normal" floppy drives of the DS/SD variety, plus a number of Horizon Ramdisks. In our first trials, we found that our sector reading would work fine for Drives 1, 2, and 3, but for drives 4 and above it wouldn't work! Instead of a sector being read, we'd get the infamous 'I GOTCHA' report. This did no harm, but still it was maddening to see that on the screen.

We consulted with Bud Mills, who quickly surmised that the problem, although being reported from our Ramdisk cards, was actually a result of something happening in our TI Disk Controller. He was RIGHT!

Through a series of carefully controlled experiments, we found that if one asks the TI Controller to sector-read a disk with a number higher than 3, the TI Controller reports an error in location >8350 of the RAM Pad. Through our experiments, we were able to determine that the error code reported in such a case is unique and different from the "no disk" or any other common problem. Thus we put in a test after an error in the DSRLNK process, and if this unique error code showed up when we were accessing through CRU 1100, we could be sure it was simply the TI Controller's problem. At that point, we modify the starting CRU address so that the DSRLNK will start looking at CRU address >1200, and re-try the DSRLNK process. This makes for another use of the controversial self-modifying code idea, but it works as intended. Our Drive 3 Ramdisk, by the way, is at CRU address >1000, so we had no trouble reading sectors from that, since the DSRLNK found drive 3 before it got round to the >1100 TI Disk Controller. Ramdisks 4 and above were all at CRU addresses above >1100, and so fell victim to the error.

Having cleared that hurdle, we were ready to proceed with integration of the Assembly with Mickey's Extended Basic program. Early in that process, we decided to put all of the process of identifying file types into the Assembly code, so that cataloging and identification would become a single process instead of two separate ones. Doing this in Assembly made the whole process easier to manage and made execution much faster than it was in the Version 2.1 of Load Master. This meant that tons of Mickey's original XB code were eliminated from the finished product, replaced by a couple of well-chosen CALL LINKs.

1.75.5. The Catlog/Ident Process

To do a complete cataloging job, we have to start by knowing which disk drive is desired, and have to read Sector number 0 of the disk. Sector 0 contains the name of the disk, the total number of sectors initialized on that disk, and the sector use map, which tells us which sectors have been allocated to files on the disk. We take the disk name and assign that to a string variable in the Extended Basic realm. We use the total capacity and use map and create from them the numeric variables that indicate the used and available sectors. This all gets reported back to XB variables in one CALL LINK. Before exiting back to XB, this CALL LINK also reads sector 1 from the disk, and places that 256 bytes in some leftover space in low memory.

The next CALL LINK is a "biggie". It starts with the sector 1 data from the previous LINK. For each file, we take the two bytes from the sector 1 data that give us the directory sector location for that file. If this number is zero, we're past the last file. Otherwise, we go ahead and read the directory sector for one file into our VDP RAM buffer. Since we don't need all 256 bytes of that, we read only a portion into a storage location in low memory. The first ten or fewer bytes are the name of the file, so we extract that. The file's principal characteristics (e.g. type, size, protection) are contained early on in the directory sector. We have to separate that data on a bit-by-bit basis to determine what kind of file we're dealing with.

If the type indicates Program, then we have to do more work before we can specify what kind of "program" file this is. Again we take some data bit-by-bit to find the number of the first sector of file content, then read that sector, and take eight bytes of it into low memory for examination. In most cases the first six bytes allow a complete identification of the program file. For example, if the first two bytes are >FFFF, then this is an E/A Option 5 program file. If the first two bytes are zero, then this could be either an E/A Option 5 or a CHARA1 type file. If the fifth and sixth bytes are >07FA, then this is a CHARA1 file, not a program. Of course if the file is bigger than 33 sectors, it's automatically not an E/A Option 5 nor a CHARA1.

In order not to take up this whole issue with our Sidebar, we've omitted large parts of the source code, including all the detailed code that identifies file types. If you're vitally interested, send me $1.00 and ask for the Load Master source code, and I'll send the complete source and its data files.

1.75.6. The Sidebar Code

Today's Sidebar starts with the part that reads sector zero of the disk. When we enter this code, Register 4 already has the drive number in its high byte, and that register doesn't get changed until the whole cataloging and identifying process is finished. Error checking is done for each sector read at the end of the subroutine UDSR by moving the byte at >8350. If that byte has been cleared, then the sector read was successful. If not, then an error has occurred. When we get that error from the TI Disk Controller for an attempt to read a sector from drive 4, we check to see if the CRU address in >83D0 is >1100, and if that's true whether the error code in >8350 is 7. If both conditions are true, we modify the DSRLNK code at two bytes past label DSR2A, then try again. This time the DSRLNK will start at CRU Address >1200, so it will find a Ramdisk at or above that CRU address.

The code at label UDSR will look strange. Even though we're using a DRSLNK process, the PAB takes only two bytes, and other parameters for the CALL process are placed in locations in the RAM Pad before the BLWP. The word at >834C first gets set to all ones to indicate a read operation, then its left byte is set to the drive number. The sector number, as a word, is passed along to >8350. The location of the name length byte for the PAB is passed to >8356 as usual, but the PAB consists of only two bytes, one being the name length, and the other the "name", consisting only of a byte set to >10. The buffer address, instead of being part of the PAB, is placed at >834E. After all that is done, we BLWP @DSRLNK with DATA >A to perform the sector read operation.

A lot has been left out, so the Sidebar is not anywhere near a complete entity. Its purpose is to supply some neat pieces of source code that you can excerpt for use in your own programs. The Travis Watford DSRLNK shown here will work for just about any "environment", even on a Geneve. Testing for the sector reading has shown that it works on any floppy disk drive with any drive controller, and also works with any Ramdisk of either the Horizon or Quest type.

1.75.7. The Extra Little Goodies

There are two in this Sidebar that may prove useful. First is the tiny routine RSXB. To use this you'll need to include the Warren/Miller GPLLNK routine, which we've omitted. This re-sets all conditions to "startup" in XB without affecting the program in memory. The second little "goodie" provides a way to RUN another XB program using a string variable as the file name. Let's say your XB program has taken an input of "DSK1.MYPROG" into the string variable F$. You could then have your XB program run that by CALL LINK("RUNIT",F$). Include in your Assembly code all the stuff from label RUNIT through label TWO. This is used in Load Master to allow running a program selected from the catalog listing.

Hope you'll find some of this useful in your own programs. The topic for next time is undecided as usual. See you then.

```
* SIDEBAR 75
*
* FRAGMENTS OF SOURCE FOR LOAD MASTER V.2.2
* CODE BY BRUCE HARRISON EXCEPT AS NOTED
*
       DEF  DISKS,FILES,RUNIT,RSXB
NUMASG EQU  >2008        NUMERIC ASSIGN
NUMREF EQU  >200C        NUMERIC REF
STRASG EQU  >2010        STRING ASSIGN
STRREF EQU  >2014        STRING REF
XMLLNK EQU  >2018        XML LINKAGE
KSCAN  EQU  >201C        XB'S KEYSCAN
VSBW   EQU  >2020        XB'S VDP SB WRITE
VMBW   EQU  >2024        XB'S VDP MB WRITE
VSBR   EQU  >2028        XB'S VDP SB READ
VMBR   EQU  >202C        XB'S VDP MB READ
VWTR   EQU  >2030        XB'S VDP REG WRITE
ERR    EQU  >2034        XB'S ERROR REPORT
IOERR  EQU  >2400        CODE FOR I/O ERROR
CALPNT EQU  >832C        CALL POINTER
PAB    EQU  >3BE9        PAB VDP ADDRESS
PABUF  EQU  >3CEF        VDP BUFFER ADDR
GPLWS  EQU  >83E0        GPL WORK SPACE
GR4    EQU  GPLWS+8      GPL REG 4
GR6    EQU  GPLWS+12     GPL REG 6
FAC    EQU  >834A        F.P. ACCUMULATOR
NAMLEN EQU  >3600        LEFTOVER LOW MEM
NAMBUF EQU  NAMLEN+2       "
FSCBUF EQU  NAMBUF+32      "
DNBUF  EQU  FSCBUF+8       "
SEC1   EQU  DNBUF+12       "
TOTL   EQU  SEC1+256       "
NUMFLS EQU  TOTL+2         "
FILSIZ EQU  NUMFLS+2       "
RECSIZ EQU  FILSIZ+2       "
*
```

```
* FIRST SECTION GETS DISK NAME, CAPACITY, FREE SPACE
*
DISKS  LWPI WS           LOAD OUR WORKSPACE
       MOV  @ONES,R12    R12 NON-ZERO
       CLR  R3           SECTOR 0
       MOV  @OHEFF,@DSR2A+2 >0F00 TO START DSR
GTS0   BL   @UDSR        USE DSR
       JEQ  RDDN         IF NO ERROR, JUMP
       MOV  @>83D0,R0    CRU ADDR
       CI   R0,>1100     >1100?
       JNE  S0ERR        IF NOT, ERROR
       CB   @>8350,@SEVEN CHECK ERRCODE 7
       JNE  S0ERR        IF NOT, ERROR
       MOV  R0,@DSR2A+2  >1100 TO START DSR
       JMP  GTS0         THEN TRY AGAIN
S0ERR  B    @ERROR       REPORT ERROR
RDDN   LI   R0,PABUF     POINT TO BUFFER
       LI   R1,DNBUF     AND STORAGE
       LI   R2,256       WHOLE SECTOR
       BLWP @VMBR        READ TO LOW MEM
       MOV  R1,R6        COPY ADDR TO R6
       AI   R1,9         ADD 9 TO R1
       LI   R2,10        TEN IN R2
GLLOP  CB   *R1,@H20     CHECK FOR SPACE AT END OF NAME
       JNE  LENFND       IF NOT, R2=LENGTH OF NAME
       DEC  R1           BACK UP ONE
       DEC  R2           DEC LENGTH
       JNE  GLLOP        REPEAT IF NOT 0
LENFND MOV  R2,@DNBUF-2  PUT NAME LENGTH IN PLACE
       MOV  @10(R6),R7   DISK CAPACITY
       CLR  R8           R8=0
       MOV  R7,R1        CAPACITY TO R1
       LI   R9,DNBUF+>38 START OF USE MAP
NXTWRD MOV  *R9+,R2      ONE WORD FROM MAP
       LI   R5,16        16 BITS TO EXAMINE
DEC1   DEC  R1           DEC COUNT BY 1
       JLT  ENDMAP       IF <0, JUMP
       SLA  R2,1         SHIFT R2 LEFT 1 BIT
       JNC  DEC5         JUMP IF NO CARRY
       INC  R8           ELSE INC COUNT OF USED
DEC5   DEC  R5           DECREMENT BIT COUNT
       JNE  DEC1         JUMP IF NOT 0
       JMP  NXTWRD       ELSE NEXT WORD
ENDMAP CLR  R0           NON-ARRAY
       LI   R1,1         2ND PARAM DISK NAME
       LI   R2,DNBUF-1   ADDR OF NAME STRING
       BLWP @STRASG      DISK NAME TO XB
       MOV  R7,@FAC      TOTAL SECTORS
       DECT @FAC         MINUS 2
       INC  R1           NEXT PARAMETER
```

```
       BL   @SNDINT      SEND TO XB
       MOV  R7,@FAC      TOTAL SECTORS
       S    R8,@FAC      - USED = FREE SECTORS
       INC  R1           NEXT PARAM
       BL   @SNDINT      SEND TO XB
       CLR  @SEC1        CLEAR WORD AT SEC1
       INC  R3           R3=1 - SECTOR 1
       BL   @UDSR        USE DSR LINK
       JEQ  RDSEC1       IF NO ERROR, JUMP
       B    @ERROR       ELSE REPORT ERROR
RDSEC1 LI   R0,PABUF     POINT AT BUFFER
       LI   R1,SEC1      AND LOW MEMORY ADDR FOR SECTOR 1
       LI   R2,256       WHOLE SECTOR
       BLWP @VMBR        READ TO LOW MEM
       B    @EXIT        EXIT TO XB
*
* NEXT SECTION GETS FILE DIRECTORY SECTORS
*
FILES  LWPI WS           OUR WORKSPACE
       CLR  R15          ARRAY ELEMENT 0
       LI   R9,SEC1      SECTOR 1
NXTFIL INC  R15          NEXT ARRAY ELEMENT
       LI   R0,11*32+9
       LI   R2,13
       XOR  @ONES,R12    "WORKING" INDICATION ON-OFF
       JEQ  WRKOFF
       LI   R1,WRKSTR
       JMP  WRTWRK
WRKOFF BL   @CLA
       JMP  NXTOK
WRTWRK BLWP @PRSTR
NXTOK  CI   R9,TOTL      PAST END OF SEC1?
       JEQ  BEEKEY       IF SO, FINISHED
       MOV  *R9+,R3      NEXT DIRECTORY SECTOR NUMBER
       JNE  NXT1         IF NOT 0, JUMP
BEEKEY B    @SNDNUL      DIRECTORY FINISHED
NXT1   BL   @CLRCEE      CLEAR C$
       BL   @CLREMM      CLEAR M$
       BL   @UDSR        READ DIRECTORY SECTOR
       JEQ  RDFNM        JUMP IF NO ERROR
       B    @ERROR       REPORT ERR
RDFNM  LI   R0,PABUF     VDP BUFFER
       LI   R1,NAMBUF    FILE NAME LOCATION
       LI   R2,32        ONLY 32 BYTES
       BLWP @VMBR        READ TO LOW MEM
       MOV  R1,R6        COPY ADDR TO R6
       AI   R1,9         ADD 9 TO R1
       LI   R2,10        TEN IN R2
CHSPC  CB   *R1,@H20     SPACE AT END OF NAME?
       JNE  GNLEN        IF NOT, GOT LENGTH
```

```
        DEC  R1           BACK ONE
        DEC  R2           DEC LENGTH IN R2
        JNE  CHSPC        JUMP IF NOT 0
GNLEN   MOV  R2,@NAMLEN   COPY R2 TO FILE NAME LENGTH
        MOV  @>E(R6),@>835E FILE SIZE WORD (SECTORS)
        INC  @>835E       ADD 1 FOR DIRECTORY SECTOR
        MOV  @>835E,@FILSIZ PUT AT FILE SIZE
        LI   R0,CEESTR+5  C$ PLUS 5
        BL   @SHWINT      NUMBER TO C$
        MOVB @>D(R6),R5   A FILE TYPE
        JEQ  ISPGM        IF ZERO, PROGRAM TYPE
        B    @NTPGM       ELSE NOT PROGRAM
ISPGM   LI   R1,PGSTR     "PROGRAM"
        LI   R0,CEESTR+5  INDICATE PROGRAM
        BL   @DISSTR      INTO C$
        C    *R10+,*R10+  ADD 4 TO R10
        MOVB @WS+21,@CEESTR C$ LENGTH
        CLR  @RECSIZ      CLEAR RECORD SIZE
        BL   @RHSEC       READ HEADER SECTOR
*
* FILE IDENTIFICATION STUFF OMITTED
*
SHWID   LI   R0,EMSTR+3   FILE IDENT
        BL   @DISSTR      PLACE IN M$
        INCT R10          ADD 2 TO R10
        SWPB R10          SWAP
        MOVB R10,@EMSTR   LENGTH OF M$
        MOV  R15,R0       ARRAY MEMBER TO R0
        LI   R1,1         1ST PARAM A$()
        LI   R2,NAMLEN+1  SEND A$()
        BLWP @STRASG      FILE NAME TO XB
        INC  R1           2ND PARAM
        LI   R2,CEESTR    C$
        BLWP @STRASG      SEND C$()
        INC  R1           3RD PARAM
        LI   R2,EMSTR     M$
        BLWP @STRASG      SEND M$()
IDEX    B    @NXTFIL      GO GET NEXT FILE
SNDNUL  LI   R2,NAMLEN    POINT AT NAME LENGTH
        CLR  *R2          MAKE THAT 0
        MOV  R15,R0       CURRENT ARRAY MEMBER IN R0
NXTNUL  CI   R0,127       IS THAT 127?
        JGT  GNUMF        JUMP IF GREATER
        LI   R1,1         FIRST PARAM (NAME)
        LI   R5,3         THREE TO SEND
SND1    BLWP @STRASG      ASSIGN NULL STRING
        INC  R1           INC PARAM NUM
        DEC  R5           DEC COUNT
        JNE  SND1         REPEAT IF NOT 0
        INC  R0           NEXT ARRAY ELEMENT
```

```
        JMP  NXTNUL        JUMP BACK
GNUMF   DEC  R15           DEC LAST ARRAY MEMBER
SNF     CLR  R0            NON-ARRAY
        LI   R1,4          4TH PARAM (N)
        MOV  R15,@FAC      NUMBER OF FILES
        MOV  R15,@NUMFLS   SAVE IN LOW MEM
        BL   @SNDINT       SEND N TO XB
        CLR  R14
        DIV  @FIFTEN,R14   DIVIDE R14-R15 BY 15
        MOV  R15,R15       ANY REMAINDER?
        JNE  DIVOK         IF REMAINDER, JUMP
        DEC  R14           ELSE DECREMENT QUOTIENT
DIVOK   MOV  R14,@FAC      PLACE AT >834A
        INC  R1            5TH PARAM (PP)
        BL   @SNDINT       SEND PARAM TO XB
        MOV  @NUMFLS,@FAC NUMBER
        C    @NUMFLS,@FIFTEN COMPARE TO 15
        JLT  SNFL          IF LESS, JUMP
        MOV  @FIFTEN,@FAC ELSE 15 TO FAC
SNFL    INC  R1            6TH PARAM (S2)
        BL   @SNDINT       SEND
        MOV  @OHEFF,@DSR2A+2 RESET DSR FOR >0F00 START
EXIT    LWPI GPLWS         GPL WORKSPACE
        B    @>6A          EXIT TO GPL INTERPRETER
ERROR   LWPI WS            OUR WS
        LI   R0,11*32+9
        LI   R2,13
        BL   @CLA          CLEAR "WORKING"
        MOV  @OHEFF,@DSR2A+2 RESET DSRLNK START
        LI   R0,IOERR      I/O ERROR CODE IN R0
        BLWP @ERR          USE XB ERROR REPORT
RSXB    BLWP @GPLLNK       USE GPLLNK (NOT SHOWN)
        DATA >6917         UNDOCUMENTED FEATURE
        LWPI >83E0         LOAD GPL WS
        B    @>6A          BACK TO GPL INTERPRETER
RUNIT   LWPI RNWS          PRELOADED REGISTERS
        MOVB @TWO,@>83C6   UNDO THE 3 KEY UNIT
        MOVB R5,*R2        MAX LEN 40
        BLWP @>2014        GET STRING VARIABLE
        MOVB *R2,@RNWS+13 ACTUAL LENGTH TO LOW BYTE R6
        MOVB R6,@LENBYT+1(R6) A ZERO AT END OF FL1
        MOV  R3,@>832C     ADDR FL1 TO >832C
        MOV  R4,@>2000     INIT VAL TO >2000 (8192)
        LWPI >83E0         LOAD GPL WS
        B    @>6A          GO TO GPL INTERPRETER
RNWS    DATA 0,1,LENBYT,FL1,>205A,>2800,0  R0 THRU R6
FL1     BYTE >82,>A9,>C7   TOKENS FOR ::,RUN,QUOTED STRING
LENBYT  BSS  41            NAME GOES HERE
TWO     BYTE 2             TWO AS A BYTE
*
```

```
* SUBROUTINES
*
RHSEC  MOV  @>1C(R6),R3  FIRST SECTOR WORD TO R3
       MOV  R3,R7        COPY INTO R7
       ANDI R7,>000F     MASK ONLY LOW NYBBLE
       SWPB R7           PUT IN HIGH BYTE
       SRL  R3,8         MOVE HIGH BYTE R3 TO LOW BYTE
       MOVB R7,R3        ADD HIGH BYTE R7 TO R3
       MOV  R11,R10      SAVE R11 IN R10
       BL   @UDSR        USE DSR TO READ FIRST CONTENT SECTOR
       JEQ  RDFSC        IF NO ERROR, JUMP
       B    @ERROR       ELSE REPORT
RDFSC  LI   R0,PABUF     BUFFER
       LI   R1,FSCBUF    LOW MEM LOCATION
       LI   R2,8         FIRST 8 BYTES
       BLWP @VMBR        READ INTO LOW MEM
       B    *R10         RETURN (LOCATION IN R10)
UDSR   SETO @>834C       SET TO READ
*
* NOTE: TO WRITE A SECTOR, YOU'D USE CLR @>834C INSTEAD
* AND WOULD NEED TO PRE-LOAD PABUF WITH DESIRED CONTENT
*
       MOVB R4,@>834C    DRIVE #
       MOV  R3,@>8350    SECTOR #
       LI   R0,PAB       PAB VDP LOCATION
       LI   R2,2         TWO BYTES
       LI   R1,PABDT     PAB DATA
       BLWP @VMBW        WRITE PAB
       MOV  R0,@>8356    ADDR TO >8356
       LI   R5,PABUF     BUFFER IN VDP
       MOV  R5,@>834E    PLACE AT >834E
       BLWP @DSRLNK      USE DSR LINK
       DATA >A           "CALL" FUNCTION
       MOVB @>8350,R2    CHECK ERROR
       RT                RETURN
* T-SHELL SOURCE CODE BY TRAVIS WATFORD
* excerpt taken by B. Harrison
*-------------------------------------------------------------------------------
*  DEVICE SERVICE ROUTINE
*DSRLNK
NPNTR  EQU  >8356
NLEN   EQU  >8354
CRULST EQU  >83D0
SAVADD EQU  >83D2
VDPWA  EQU  >8C02
VDPRD  EQU  >8800
DSRWS  BSS  32
DSRWS5 EQU  DSRWS+10
DSRWS0 EQU  DSRWS+1
DSRLNK DATA DSRWS,DSR
```

```
H20     BYTE >20
H2E     BYTE >2E
HAA     BYTE >AA
FNAME   BSS  7
        EVEN
DSR     MOV  *R14+,R5     GET DSR OFFSET
        SZCB @H20,R15     ZERO ALL BUT >20
        MOV  @NPNTR,R0    GET POINTER TO NAME LENGTH
        MOVB @DSRWS0,@VDPWA SET VDP READ ADDRESS
        NOP
        MOVB R0,@VDPWA
        AI   R0,-8        SET A POINTER TO ERROR RETURN BYTE
        MOVB @VDPRD,R1    GET THE NAME LENGTH
        MOVB R1,R3
        JEQ  DSR9         IF LEN=0 ABORT
        SRL  R3,8
        SETO R4
        LI   R2,FNAME     MOVE NAME TO CPU
DSR1    INC  R4
        CI   R4,7         SEE IF NAME LENGTH > 7
        JH   DSR9
        C    R4,R3        SEE IF NAME IS MAX LENGTH
        JEQ  DSR2
        MOVB @VDPRD,R1    GET CHAR
        MOVB R1,*R2+
        CB   R1,@H2E      SEE IF PERIOD
        JNE  DSR1
DSR2    CLR  @CRULST
        MOV  R4,@NLEN     SAVE NAME LENGTH
        INC  R4
        A    R4,@NPNTR    ADJUST NAME POINTER
        LWPI GPLWS
        CLR  R1
DSR2A   LI   R12,>0F00    START AT >1000
DSR3    SBZ  0
DSR3A   AI   R12,>0100    NEXT CRU BASE ADDRESS
        CLR  @CRULST
        CI   R12,>2000    QUIT AFTER CRU >1F00
        JEQ  DSR8
CRUOK   MOV  R12,@CRULST
        SBO  0
        LI   R2,>4000
        CB   *R2,@HAA     SEE IF A CARD IS PRESENT
        JNE  DSR3
        A    @DSRWS5,R2   ADD THE DSR OFFSET
        JMP  DSR5
DSR4    MOV  @SAVADD,R2
        SBO  0
DSR5    MOV  *R2,R2       SEE IF THERE ARE ANY ROUTINES
        JEQ  DSR3
```

```
       MOV  R2,@SAVADD
       INCT R2
       MOV  *R2+,R9      GET ROUTINE ADDRESS
       MOVB @NLEN+1,R5   GET NAME LENGTH
       JEQ  DSR7
       CB   R5,*R2+      SEE IF NAME LENGTH MATCHES
       JNE  DSR4
       SRL  R5,8
       LI   R6,FNAME     SEE IF NAME MATCHES
DSR6   CB   *R6+,*R2+    COMPARE CHARS
       JNE  DSR4
       DEC  R5
       JNE  DSR6         REPEAT
DSR7   INC  R1
       BL   *R9          EXECUTE THE ROUTINE
       JMP  DSR4
       SBZ  0
       LWPI DSRWS        RESTORE REGISTERS
       MOVB @DSRWS0,@VDPWA SET VDP READ ADDRESS
       NOP
       MOVB R0,@VDPWA
       NOP
       MOVB @VDPRD,R1    GET THE NAME LENGTH
       SRL  R1,13        SEE IF ANY ERRORS
       JNE  DSR10
       RTWP
DSR8   LWPI DSRWS        RESTORE REGISTERS
DSR9   CLR  R1
DSR10  SWPB R1
       MOVB R1,*R13      PUT ERROR CODE IN R0 OF WS1
       SOCB @H20,R15     SET EQUAL BIT FOR ERROR
       RTWP
*
* DATA
*
WS     BSS  32           OUR WORKSPACE
PABDT  BYTE 1,>10        SECTOR READ/WRITE PAB DATA
*
* NOTE:  OTHER DATA HAS BEEN OMITTED
```

1.76. Art Of Assembly — Part 76. New And Improved

By Bruce Harrison

We've said before in this column that even after years of making various programs in Assembly Language we are still able to learn new tricks and improve things. Today's column is about old versus new ways of doing various operations. Most are in the form of subroutines, so there's no complete program here, but it should be a learning experience and provide better ways to do some little things.

1.76.1. Screen Clearing

As with so many things, there are many ways to do the screen clearing operation. For example, one can use VSBW 768 times, incrementing R0 each time. That works, but it's the very slowest method. In many of our programs, we've used a line in the DATA section of the program called BLNKLN, which is 32 or 40 spaces (depending whether we're in 32 or 40 column mode) and is used with a loop using VMBW 24 times. This method also works, of course, and is much faster than the VSBW method, but it takes up more memory both in the code segment and in the DATA part.

While working on the improved version of Load Master, we came up with the "new and improved" method that uses very little memory, requires no DATA, and is faster than any other method. It affects only registers 0, 1, and 2 of your workspace, and can be modified slightly to provide other services such as partial screen clearing operations. What we've shown in the Sidebar is a very simple screen clear that can be used in the "normal" E/A environment, plus slightly modified versions for use in the Basic/XBasic environments and so on. Take your pick, but be assured any of these "new and improved" subroutines will do the job as quickly as possible and in a minimal amount of memory. We've even included a version that can do an "HCHAR" type service many times faster than the XB version. Take care, though, that your calling code does not make this subroutine go beyond your screen area, as no provision has been made here for the "wraparound" to the top of the screen as HCHAR does.

These routines work faster than the old ones mainly because we've taken full advantage of the auto-incrementing feature when writing to VDP RAM via >8C00. Thus once we've written the starting VDP address to >8C02, we don't have to write an address again, as was the case when we used BLWP to either VSBW or VMBW. Rather, every time we MOVB @>8C00, the VDP address increments all by itself. We also save some time by avoiding the BLWP "overhead" in favor of the less time consuming BL.

1.76.2. Slightly Better Keyscan

In our own "normal" key routine, we need somewhere in the DATA a byte of >20 (ANYKEY BYTE >20) so that we can compare the value at >837C (GPL Status byte) to ANYKEY, and repeat scan if that's not equal. Here's another method, which works just as well but doesn't need the byte of >20 in DATA. This, as shown in the Sidebar, does the LIMI 2 and LIMI 0 operations, so that waiting for a keystroke won't stop anything that needs the interrupts, such as sprite motion or background music. That has the disadvantage of enabling the **FCTN =** to get back to the title screen. If you don't need the interrupts serviced, just use this without the lines LIMI 2 and LIMI 0.

1.76.3. Slightly Better Speed

You can't do this in stuff that links from Extended Basic, but in "pure Assembly" programs, speed of execution overall can be slightly improved by placing your workspace registers in the >8300 (Ram Pad) area. Since that part of memory is accessed on a 16-bit word basis rather than byte-by-byte, all register actions will happen slightly faster. To give a concrete example, we recently took a modified (by Bob Carmany) version of Shawn Baron's AMSTEST program, which runs an exhaustive test on the AMS card, and instead of having the registers in the program code as BSS blocks, we put in EQUates so that the main program registers were at >8300 and the utilities' registers at >8320. With no other changes, the execution time for this program on our 256K AMS card went from 2 minutes 25 seconds to 2 minutes flat. That's not a big improvement, (about 20 %) but points in the right direction.

1.76.4. MUCH Better Boot Tracking

Last but not least, for those using Assembly routines with Extended Basic, here's a much improved method for finding out what disk drive an XB program loaded from, then modifying all occurrences of "DSK1" in that program to whatever device it loaded from. This will work for "non-DSK" devices as well, including those starting with WDS (old hard drives) and SCS (new SCSI hard drives.)

The impetus for this new version of our boot tracking routine came from our experience in modifying Load Master. One of the things we did was to add boot tracking to the LOAD program, so that Load Master could be started from any disk drive and would always "know" where the Load Master disk was located. In our early attempts, we found that while our boot tracking worked just fine on our own system's drives, both the floppies and the Ramdisks, it would cause a lockup on Mickey Cendrowski's system. Mickey has a Myarc HFDC as her main disk controller, which handles both her floppy and hard drives. That old version of boot track used a particular method of turning the ROM on and off, which worked okay for most disk controllers, but not for the Myarc HFDC.

In recent work, we'd done some things with CRU access on the AMS card, and in that case used SBO and SBZ operations to turn the card's "ROM" on and off. It looked as though that might be a good idea for the on-off switching required in the boot tracking case, so we tried that. Yes, we now had a means of boot tracking that worked with every kind of controller that it was tested on. This means it worked on TIs with Myarc disk controllers, on Geneves, and even with the SCSI controller on Charlie Good's Geneve. As you can see in the Sidebar, once we've set R12 to the CRU address for the controller's ROM, we simply SBO 0 to turn the ROM on so we can read it, then SBZ 0 to turn the ROM off again.

This new version of boot track also has the ability to work with programs that contain a zero byte in the middle of a program line, so long as that's after a >C9 or >C7 token. This is a bit esoteric, but hang in there a moment. Suppose a line contains IF XYZ THEN 40. When this is tokenized, the line number 40 becomes (in hex) C9,0,28. In early versions of our boot track, that byte of 0 following the C9 would be mistaken for the 0 that ends a program line, so any DSK1 that came later in that line would get skipped. A similar problem can occur when a program sets a string variable to null value. Let's say the line includes A$="". That "" will become tokenized as >C7,0, which again raises the possibility of a false ending to the line.

In the version of boot track shown in the Sidebar, both of these problems are taken care of, so that if >C9 or >C7 is found, the comparison process will skip over that 0 byte and thus will correctly find DSK1 anywhere in any XB program line. As in older versions, wherever DSK1 is found, it will be changed to the device name found by the earlier part of the routine.

1.76.5. Using the TRACK6 Routine

Perhaps it's best to describe by example. In the case of Load Master, the LOAD program has this code embedded by Todd Kaplan's ALSAVE routine. Let's suppose that we've copied LOAD, LOADMASTER, DEFAULTS, OPTIONS, and the mysterious EA file into the root directory of a hard drive named SCS2 (a SCSI drive). We then get into Extended Basic and type RUN "SCS2.LOAD" **ENTER**. When LOAD starts, it first dumps all of its Assembly routines, including TRACK, into low memory. There's a line in LOAD that says RUN "DSK1.LOADMASTER", but before that line executes, there's one that says CALL LINK("TRACK"). In that CALL LINK, the TRACK routine gets the device name SCS2 from the controller ROM, then searches the XB part of LOAD (in memory) until it finds DSK1. It changes DSK1 to SCS2, so that instead of RUN "DSK1.LOADMASTER", that line says RUN "SCS2.LOADMASTER". When that line executes, LOADMASTER is loaded and run from SCS2.

When LOADMASTER starts running, it too executes a CALL LINK ("TRACK"). This time the code in TRACK sees that it already has the device name, because its R0 is non-zero, so it jumps ahead to the part that scans the XB content of LOADMASTER, where it replaces all occurrences of DSK1 with SCS2. Now if the user selects OPTIONS, that program will be sought on SCS2 as well. OPTIONS, in turn, will also perform a CALL LINK("TRACK"), and thus will look for DEFAULTS on SCS2. When OPTIONS is done, it will save the revised DEFAULTS and then re-load LOADMASTER from SCS2, and so on.

There are places in the OPTIONS file where we want the four characters DSK1 to stay "as-is" when TRACK is running. To make sure of that, we put DSK1 in as a string expression like this: D$="DSK"&"1". Having that ampersand in the middle insures that TRACK won't find DSK1 in four successive bytes, and thus will not replace it with SCS2. This same trick can be applied in your own work if "DSK1" is supposed to stay that way. In this set of programs, only LOAD contains a CALL INIT, so that the code for TRACK remains in place and available while any of these programs is running. There are additional Assembly routines included in LOADMASTER, but those are embedded using Harry Wilhelm's High Memory Loader, and so don't affect the routines that LOAD placed in Low memory.

To summarize, use TRACK6 this way:

1. Embed TRACK6/O using ALSAVE in the "main" program.
2. Make sure CALL LINK("TRACK") is performed before any RUN "DSK1.XXXX" lines in that program.
3. In chained programs, (e.g. LOADMASTER) perform CALL LINK("TRACK") early in the program. Make sure the chained programs don't perform CALL INIT.

For your convenience if you get *MICROpendium* on disk, we've included TRACK6/O as an object file in our submission, and asked John and Laura to put that on the disk version. Otherwise, just type in the section of the Sidebar that's source for TRACK6 and assemble that so it can be embedded with ALSAVE. If you need help with any of this, don't hesitate to call us any time from 9AM through Midnight Eastern time at (301) 277-3467.

Again our next topic is undecided. See you in two months.

```
* SIDEBAR 76
* NEW AND IMPROVED SUBROUTINES
* all Public Domain
*
* CLEAR THE SCREEN - SIX NEW VERSIONS
*
* 1. "PLAIN-JANE" VERSION FOR 32 CHAR SCREEN IN E/A
*
CLS    LI   R0,>0040     SET >40 IN LOW BYTE R0
       LI   R1,>2000     SPACE IN LEFT BYTE R1
       LI   R2,768       768 TIMES (32 * 24)
CLS1   MOVB R0,@>8C02    SEND HIGH BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    SEND >40 BYTE
CLS2   MOVB R1,@>8C00    WRITE A SPACE
       DEC  R2           DECREMENT COUNT
       JNE  CLS2         IF NOT ZERO, REPEAT
       RT                ELSE RETURN
*
* 2. "PLAIN-JANE" VERSION FOR 32 CHAR SCREEN IN XB
*
CLSXB  LI   R0,>0040     SET >40 IN LOW BYTE R0
       LI   R1,>8000     SPACE W/OFFSET IN LEFT BYTE R1
       LI   R2,768       768 TIMES (32 * 24)
CLS1   MOVB R0,@>8C02    SEND HIGH BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    SEND >40 BYTE
CLS2   MOVB R1,@>8C00    WRITE A SPACE
       DEC  R2           DECREMENT COUNT
       JNE  CLS2         IF NOT ZERO, REPEAT
       RT                ELSE RETURN
*
* 3. "PLAIN-JANE" VERSION FOR 40 COL SCREEN IN E/A
*
CLS40  LI   R0,>0040     SET >40 IN LOW BYTE R0
       LI   R1,>2000     SPACE IN LEFT BYTE R1
       LI   R2,960       960 TIMES (40 * 24)
CLS1   MOVB R0,@>8C02    SEND HIGH BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    SEND >40 BYTE
CLS2   MOVB R1,@>8C00    WRITE A SPACE
       DEC  R2           DECREMENT COUNT
```

```
       JNE  CLS2         IF NOT ZERO, REPEAT
       RT                ELSE RETURN
*
* 4. "FANCY" VERSION FOR 32 OR 40 CHAR SCREEN IN E/A
* USE BL @CLS32 FOR 32, BL @CLS40 FOR 40
*
CLS32  LI   R2,32*24     768 FOR 32 CHAR
       JMP  CLS0
CLS40  LI   R2,40*24     960 FOR 40 CHAR
CLS0   LI   R0,>0040     SET >40 IN LOW BYTE R0
       LI   R1,>2000     SPACE IN LEFT BYTE R1
CLS1   MOVB R0,@>8C02    SEND HIGH BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    SEND >40 BYTE
CLS2   MOVB R1,@>8C00    WRITE A SPACE
       DEC  R2           DECREMENT COUNT
       JNE  CLS2         IF NOT ZERO, REPEAT
       RT                ELSE RETURN
*
* 5. "FANCIER" VERSION FOR 32 OR 40 CHAR SCREEN IN E/A
*  IN THIS VERSION, YOU CAN CLEAR ANY SELECTED AREA
*  OF SCREEN BY FIRST SETTING R0 TO THE START POINT
*  AND R2 TO THE NUMBER OF SPACES TO WRITE, THEN
*  BL  @CLS0A
*
CLS32  LI   R2,32*24     768 FOR 32 CHAR
       JMP  CLS0
CLS40  LI   R2,40*24     960 FOR 40 CHAR
CLS0   CLR  R0
CLS0A  ORI  R0,>4000     SET >4000 BIT IN R0
       SWPB R0
       LI   R1,>2000     SPACE IN LEFT BYTE R1
CLS1   MOVB R0,@>8C02    SEND HIGH BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    SEND >40 BYTE
CLS2   MOVB R1,@>8C00    WRITE A SPACE
       DEC  R2           DECREMENT COUNT
       JNE  CLS2         IF NOT ZERO, REPEAT
       RT                ELSE RETURN
*
*  6. FANCIER YET VERSION:
*  IN THIS VERSION, YOU CAN CLEAR ANY SELECTED AREA
*  OF SCREEN BY FIRST SETTING R0 TO THE START POINT
*  AND R2 TO THE NUMBER OF SPACES TO WRITE, THEN
*  BL  @CLS0A, BUT YOU CAN ALSO PUT ANY CHARACTER
*  ASCII YOU WANT IN R1'S LEFT BYTE, THEN THE STARTING
*  SCREEN ADDRESS IN R0 AND NUMBER OF REPEATS IN R2,
*  THEN BL @CLS0B TO DO A VERY SWIFT "HCHAR" TYPE
*  OPERATION.
*
```

```
CLS32  LI   R2,32*24     768 FOR 32 CHAR
       JMP  CLS0
CLS40  LI   R2,40*24     960 FOR 40 CHAR
CLS0   CLR  R0
CLS0A  LI   R1,>2000     SPACE IN LEFT BYTE R1
CLS0B  ORI  R0,>4000     SET >4000 BIT IN R0
       SWPB R0
CLS1   MOVB R0,@>8C02    SEND HIGH BYTE
       SWPB R0           SWAP
       MOVB R0,@>8C02    SEND >40 BYTE
CLS2   MOVB R1,@>8C00    WRITE A SPACE
       DEC  R2           DECREMENT COUNT
       JNE  CLS2         IF NOT ZERO, REPEAT
       RT                ELSE RETURN
*
* IMPROVED KEY SCAN SUBROUTINE, DOESN'T REQUIRE
* EXTERNAL DATA FOR OPERATION
* DOES REQUIRE THAT KSCAN BE REF'D OR EQU'D
*
KEY    BLWP @KSCAN       SCAN KEYBOARD
       LIMI 2            ALLOW INTS
       LIMI 0            STOP INTS
       MOVB @>837C,@>837C MOV GPL STAT BYTE TO ITSELF
       JEQ  KEY          IF 0, SCAN AGAIN
       RT                ELSE RETURN
*
* TRACK6/S  - SOURCE CODE FOR CHANGING
* DEVICE NAME IN AN XB PROGRAM
* FINDS DSK1 ANYWHERE IN XB PROGRAM
* AND CHANGES IT TO XXXX, WHERE XXXX IS THE
* DEVICE FROM WHICH XB PROGRAM WAS LOADED
* EVEN WORKS FOR NON-DSK DEVICES, e.g. WDS1, SCS1
* 18 DEC 1996
* PUBLIC DOMAIN
* CODE BY Bruce Harrison
*
       DEF  TRACK
TRACK  LWPI WS           USE OUR WORKSPACE
       MOV  R0,R12       USE OLD CRU ADDR IF AVAILABLE
       JNE  GET30        THEN JUMP AHEAD
GETD0  MOV  @>83D0,R12   GET THE CRU BASE IN R12
       JEQ  EXIT         GET OUT IF 0
       MOV  @>83D2,R9    GET THE ROM ADDRESS FOR DEVICE
       JEQ  EXIT         GET OUT IF 0
       SBO  0            ENABLE THE DEVICE ROM
       AI   R9,5         ADDING FIVE PUTS US AT DEVICE NAME
       LI   R4,4         4 BYTES TO GET
       LI   R10,TEXT+1   POINT TO TEXT BUFFER+1
MOVIT  MOVB *R9+,*R10+   MOV ONE BYTE FROM ROM TO TEXT BUFFER
       DEC  R4           FINISHED?
```

```
       JNE  MOVIT        NO, DO ANOTHER BYTE
       SBZ  0            DISABLE THE ROM (R4 IS ZERO AT THIS POINT)
       MOV  R12,R0       SAVE R12 IN R0
GET30  MOV  @>8330,R13   PUT START OF LINE NUMBER TABLE IN R13
NEWLI  INCT R13          POINT TO BYTE CONTAINING ADDRESS OF LINE
       C    R13,@>8332   ARE WE PAST END OF LINE NUMBER TABLE
       JGT  EXIT         IF SO WE ARE FINISHED
       MOVB *R13+,R4     GET HIGH ORDER BYTE OF LINE ADDRESS IN R4
       SWPB R4           SWAP R4
       MOVB *R13+,R4     GET LOW ORDER BYTE OF LINE ADDRESS
       SWPB R4           SWAP SO R4 CONTAINS STARTING ADDRESS OF A LINE
*
* AT THIS STAGE R4 POINTS TO THE BEGINNING OF A LINE IN THE XB PROGRAM,
*
NEXT
       MOV  R4,R10       SET R10 EQUAL TO R4
CHECK
       LI   R9,DSK1      POINT AT TEXT 'DSK1'
       LI   R3,4         SET FOR 4 CHARACTER COMPARE
CMPB   CB   *R10,@CEE9   CHECK FOR C9 TOKEN
       JNE  CMPC7        IF NOT EQUAL, JUMP AHEAD
       INCT R4           ELSE MOVE R4 POINTER AHEAD TWO
       JMP  NOCMP        THEN JUMP AHEAD TO NOCMP
CMPC7  CB   *R10,@CEE7   IS THIS A BYTE OF >C7
       JNE  CMPC         IF NOT, JUMP
       INC  R4           ELSE INCREMENT POINTER BY ONE
       JMP  NOCMP        THEN JUMP
CMPC   CB   *R10,@ZERO   IS THE BYTE WE'RE LOOKING AT A ZERO?
       JEQ  NEWLI        IF SO, IT'S THE END OF A PROGRAM LINE
       CB   *R9+,*R10+   COMPARE BYTES AND INCREMENT
       JNE  NOCMP        IF NOT EQUAL, GET OUT
       DEC  R3           ELSE DECREMENT COUNT
       JNE  CMPB         IF NOT ZERO, REPEAT
       LI   R9,TEXT+1    DSK1 WAS FOUND. POINT TO BOOT TRACKED DEVICE NAME
       MOV  R4,R10       R10 POINTS TO LOCATION WHERE "DSK1" WAS FOUND
*
* THE LOOP AT MOV2 OVERWRITES "DSK1" IN THE XB PROGRAM LINE
* WITH DEVICE NAME FOUND IN THE BOOT TRACK PROCESS
*
       LI   R5,4         FOUR BYTES TO MOVE
MOV2
       MOVB *R9+,*R10+   MOVE ONE, INCREMENT POINTERS
       DEC  R5           DECREMENT COUNTER
       JNE  MOV2         IF NOT ZERO, REPEAT
       MOV  R10,R4       START OF NEXT GROUP OF BYTES
       JMP  CHECK        JUMP BACK
NOCMP  INC  R4           GO START AT NEXT BYTE IN XB PGM LINE
       JMP  NEXT         AT LABEL NEXT
EXIT   LWPI >83E0        LOAD GPL WORKSPACE
       B    @>006A       RETURN TO GPL INTERPRETER
```

```
WS      DATA 0            R0 STARTS AS ZERO
        BSS  30           REST OF WS
TEXT    BYTE 4            LENGTH OF DEVICE NAME
        BSS  5            BUFFER FOR DEVICE NAME
ZERO    DATA 0            ZERO BYTE FOR COMPARISON
DSK1    TEXT 'DSK1'       COMPARISON TEXT
CEE9    BYTE >C9          TOKEN FOR LINE NUMBER
CEE7    BYTE >C7          TOKEN FOR QUOTED STRING
        END
```

Unpublished articles

This section contains material that was submitted to *MICROpendium*, but was not published before it ceased publishing. Thanks to Bruce Harrison for supplying his "Art Of Assembly" material.

1.77. The Art Of Assembly — Part 77. Breaking New Ground

By Bruce Harrison

For the most part, this edition of The Art will deal with some details of ways of making use of the AMS memory, but we'll also get into some stuff dealing with the SCSI hard drive.

1.77.1. MIDI Album Without Mini-Memory

While we were developing our AMS version of MIDI-Master, our friend Richard Bell, who was testing our efforts, asked what turned out to be an interesting question, "Why does Album still need the Mini-Memory when there's plenty of memory in the AMS?" The reason had to do with address mapping. The Album feature of MIDI-Master was set up to use memory in the >7000 block (nearly all of that block). The mapper chip in the AMS does not allow any of the AMS' pages to be mapped to addresses other than the >2000->3000 and >A000->F000 blocks. We know this is true because we did controlled experiments that proved this to be fact. Still, Richard Bell's question was one of those things that nags at the programmer's mind night and day. Why indeed?

1.77.2. The Sudden Inspiration

Yes, we know what Thomas Edison said about inspiration versus perspiration, and most of the time that applies, except that without the inspiration the perspiration is just sweat. The inspiration had to do with pages 0 and 1 of the AMS memory. Until now, to our knowledge, nobody who's programmed for the AMS card has made any use whatsoever of pages 0 and 1. Many don't even use pages 4 through 9 either, and that seems a waste. Our AMS version of MIDI-Master uses pages 2 and 3 for its main program code, and pages 4 through whatever for storing the music that it plays. That's reasonably good use of memory, but there were still 8192 bytes of perfectly good memory in pages 0 and 1, while we were having to plug in the Mini-Memory to run Album. It simply didn't make good sense. Finally the idea hit the brain: Use page 0 or 1 mapped as if it were some legal area in high memory. In the actual end product, that wound up being the use of page 1 mapped to the area >C000 through >CFFF.

1.77.3. Code That "Moves Itself"

This isn't strictly speaking true, but the idea came from that AMS Testing program that we helped Bob Carmany with. In that program there were sections of code that re-located themselves from one AMS page to another so that all pages of the AMS could be tested without destroying or losing the program's own code. In our Album case, we decided to use a small section of code outside the actual Album code to move Album.

To do that, we first AORG to location >BFB8, so there's enough room between there and >C000 to do a few things. First, the code starting at >BFB8 initializes the AMS memory. At that point the mapping registers are set so that page 0 is "mapped" to >0000, page 1 to >1000, etc. Pages 2 and 3 are then mapped to the area containing the main MIDI-Master code, and page >B and >C are mapped to >B000 and >C000, where the Album code resides. Next, the little section of "move it" code checks to see if an AMS has been found, and branches to a "sorry, no AMS found" message display in the main program if no AMS is present. So long as an AMS is there, the code proceeds to do its "move" operation. First, it sets page 1 to map at >A000. Next it moves everything from >C000 to the end of Album's code into page 1, now masquerading as >A000. Now with the Album's code all safely stashed in page 1, it re-maps page 1 to appear as >C000, then jumps to >C000 to start executing Album.

Confused? Please don't get confused yet, as this soup gets thicker with continued stirring. Album operates to do a few things on its own, such as cataloging a disk, but to actually load and play music, it uses routines in the main program part, which is sitting there in pages 2 and 3. Whenever that happens, the main program does a re-mapping of the High Memory addresses, so that >A000 thru >FFFF are actually pages 4 through 9 of the AMS. Thus while the main program's routine is executing, >C000 is actually page 6 (or higher) of the AMS card. None the less, the Album part is still kept in page 1, so it doesn't get overwritten when music data gets put into >C000.

Before Album branches into the main code, it sets a flag that's in the main program's data section. Each routine that gets called by Album has been modified so that before an exit from the routine, that flag is checked, and if it's set, a mapping is done on the AMS to set page 1 back to the >C000 address space. This way, control returns to Album, since page 1 is once again treated as >C000. Yes, it sounds complicated, but it really isn't, once you understand how the AMS card and its mapping registers work.

1.77.4. Workspace Registers

One potential problem when doing this kind of fooling around is the possibility of losing one's workspace registers. We "headed that off at the pass" by using two sets of registers, both of which are in RAM Pad. This way, our register spaces don't get affected by the mapping of pages in the AMS. The Album part uses >8300 thru >831F, and the main program uses >8320 thru >833F. Right at the start of the "move" section of Album, the workspace is set by LWPI >8320, so that the "move" addresses can be kept in registers and won't be affected by the move. Also, the code that does the re-mapping operations, except for one operation in the "move" section, is all kept in the main program's part, where pages 2 and 3 remain mapped to >2000 and >3000 throughout all phases of the program's running.

1.77.5. A Quick Review

MIDI-Master and its Album feature are contained in two E/A Option 5 program files. The first, MASEXA, has a header starting with >FFFF, so that once it's been loaded, the second program file MASEXB will also be loaded. MASEXA's header includes a load-point specified as >BFB8, and a length just a little under 4096 (>1000) bytes. Thus the loader will place this code into the high memory starting at >BFB8 and running through most of the >C000 area. At this point the AMS Mapper is inactivated, so this stuff actually goes into pages >B and >C of the AMS memory. MASEXB, which is nearly >2000 bytes, loads into >2000 and >3000, which correspond to pages 2 and 3 of the AMS memory. These pages are not re-mapped at any time during the program. After the code at >BFB8 does its "move" operation, the Album code resides in page 1 of the AMS, which is mapped to act as >C000 while Album itself is in control.

1.77.6. Fringe Benefits

By our using the >B000 and >C000 sections of high memory as the initial load area for Album, we not only made it possible to run Album without Mini-Memory, but also made it possible to load the two program files in many ways. There is even a LOAD program on the disk which will load both programs from Extended Basic. Both can also be loaded via Load Master by selecting MASEXA from its catalog listing. These two programs can also be loaded via Ramdisk menu, from Funnelweb, or from E/A Option 5 itself. Yes, there is a Santa Claus, and he's offered up an interesting gift in the AMS version of MIDI-Master.

1.77.7. Cataloging The SCSI Drive

Our friend Lew King has inspired several of our AMS products since he first purchased his AMS card. This time, however, he's inspired a product that doesn't need the AMS.

Lew reported that he was to a degree happy with his SCSI, except that the cataloging program supplied with SCSI was written in Extended Basic, and was painfully slow in operation. Lew had used our AMS Slideshow with a sub-directory on his SCSI, so he knew that we had ready-made routines capable of reading the catalog of any sub-directory on his SCSI (or the root directory if desired). The cataloging part of AMS Slideshow is tailored so that it "filters" the file names, showing only those ending in _P, which are assumed to be TI-Artist picture files. Lew figured (correctly) that we could remove that file name filter and create a general-purpose cataloger for his SCSI.

It took only a few days to have an initial working version, and then some refinements were added to make it a more complete product. The thing Lew wanted most was speed, and this program gives him that. It reads a directory or sub-directory from the SCSI in about a second or two depending how many files are there. The list appears on screen with the first 22 files showing. **FCTN 4**, **X**, **x**, **CTRL X** or **FCTN X** "pages" through the list in the forward direction and **FCTN 6**, **E**, **e**, **CTRL E** or **FCTN E** pages backward.

Two features are important to Lew, and probably to anyone else with a SCSI. The initial prompt field allows up to a 40 character path name, so that the exact path desired can be cataloged right away. For example, if Lew wants to see what's in his UT sub-directory, he types SCS1.UT. at the prompt, then presses **ENTER** and very quickly gets the contents of SCS1.UT on screen. The second important feature is that by simply pressing **P** or **p** while the catalog is on screen, he gets a very rapid printout of that sub-directory sent to his printer.

1.77.8. More "Fringe Benes"

SCSICAT works on Lew's SCSI drive, but also will catalog any of my Horizon Ramdisks, any Floppy drive, and even the older Myarc Hard Drive systems. Thus it's sort of a general purpose quick cataloger.

1.77.9. Today's Sidebar

Not too big this time, just a couple of sample "snippets" for your amusement. First in the Sidebar is the "move" part of the MIDI Album code. This uses subroutines in the main program code, and if no AMS is found, it branches to a place in the main code unconditionally. We've also shown the code from the main program that sets page 1 to act as >C000, just so you can make the connection.

There's also a small portion of source code from SCSICAT. In this case, we wanted you to see a small trick we used to speed up the process of getting the numeric variables from the VDP Buffer. By reading those directly from VDP into FAC and ARG, we speed up the process of dealing with doing the necessary floating point math operations. In the case of the numbers for capacity of the SCSI drive, we keep these in floating point format because they are generally too large to be converted to integer values. For types, sizes and record lengths of files, we convert to integers before doing anything else with them. The file records from the catalog get tucked away in memory as groups of 16 bytes each. That's ten for the name, (which we fill out to ten with spaces) then two bytes for each number after conversion to integer format.

1.77.10. Another Myarc Mystery

For some reason, files of the D/V 80 type that we have gotten from Geneve users do not have the usual 2 reported in the TYPE number from the catalog operation. Instead, their type is given as 18! On our Ramdisk menu's Show Directory function, these show up as H80 instead of d80. It took a while to figure out what to do with these, so they'd be properly identified as DIS/VAR 80. In the end it was quite easy, since 18 when expressed in hex is >12. Thus if we simply ANDI the value to 7, that strips off the 1 hex digit, leaving just the 2, which our program translates to DIS/VAR for display. For those who want to play around with such things, we'll pass along that we found out the TYPE number for sub-directory names on SCSI drives. They show up as type 6, when you might have thought type numbers didn't go past 5. In SCSICAT, those get reported on the screen and printouts as SUB-DIR, with no sector size (always 2) or record size shown. We don't have a clue why those Geneve files show up as TYPE 18, but at least we found an easy way to deal with them. Load Master, by the way, is not confused by these files, as it examines the directory sector bit-by-bit and correctly identifies these that way. Load Master cannot, however, catalog SCSI or hard drives of any kind. It works on floppies and Ramdisk drives only.

That's it for this time. Hope you'll enjoy this rather heavy stuff.

```
* SIDEBAR 77
* SOME SNIPPETS FOR YOUR AMUSEMENT
* CODE BY BRUCE HARRISON
*
* PART ONE FROM NEW MIDI ALBUM
*
       DEF  ALSTRT
       AORG >BFB8
ALSTRT LWPI >8320     WORKSPACE IN RAM PAD
       BL   @AMSINI   INITIALIZE AMS MAP
       C    R1,@>401E IS AMS PRESENT?
       JEQ  ALSTR0    IF SO, PROCEED
       B    @NOAMS    ELSE BRANCH INTO MAIN PROGRAM
ALSTR0 SBO  0         TURN ON AMS CARD
       LI   R1,>100   1 IN LEFT BYTE R1
       MOVB R1,@>4014 SETS PAGE 1 TO >A000
       SBZ  0         CARD OFF
       SBO  1         MAPPER ON
       LI   R0,ALENT  START OF ALBUM CODE (>C000)
       LI   R1,>A000  START OF PAGE 1 (AS >A000)
       LI   R4,ALEND->C000  LENGTH OF STUFF TO MOVE
MOVALB MOV  *R0+,*R1+  MOVE A WORD OF ALBUM TO PAGE 1
       DECT R4        DEC COUNT BY 2
       JGT  MOVALB    IF >0, REPEAT
       SETO @ALBFLG   SET ALBUM FLAG IN MAIN PROGRAM
       BL   @SETP1C   SET PAGE 1 TO >C000
       JMP  ALENT     JUMP TO START OF ALBUM
ENDMOV EQU $
       AORG >C000     LOADS AT >C000 IN HIGH MEMORY
ALENT  B    @ALBNTR   BRANCH TO START OF CODE
*
* FOLLOWING ARE SUBROUTINES IN MAIN
* MIDI-MASTER THAT ARE USED BY THE ABOVE
*
* AMSINI - INITIALIZES MAPPING OF AMS CARD TO "NORMAL"
* MEANING PAGES ARE AT THEIR NAMED ADDRESSES
* E.G PAGE 2 IS >2000, 10 IS AT >A000, ETC
* MAPPER IS NOT TURNED ON
AMSINI LI   R12,>1E00    AMS CRU BASE
       SBO  0            TURN ON AMS
       LI   R1,>FEFF     -257 IN R1
       LI   R0,>4000     START OF MEMORY
AMSLP  AI   R1,>0101     ADD 1 PAGE (257)
       MOV  R1,*R0+      MOVE 2 BYTES TO MEM-MAPPER
       CI   R0,>4020     ALL DONE?
       JLT  AMSLP        NO, INIT MORE
       RT                RETURN
*
* ROUTINE SETP1C
* SETS AMS PAGE 1 TO >C000
```

```
*
SETP1C LI   R12,>1E00    AMS CARD CRU
       SBZ  1            TURN OFF MAPPER
       SBO  0            TURN ON CARD
       LI   R1,>100      1 IN LEFT BYTE R1
       MOVB R1,@>4018    PAGE TO >C000
       SBZ  0            TURN OFF CARD
       SBO  1            TURN ON MAPPER
       RT                RETURN
*
* PART TWO - SNIPPET FROM SCSICAT
*
* AT THIS STAGE, CATALOG FILE IS OPEN
* AND WE'RE READING THE DISKNAME RECORD (#0)
*
RDDNAM LI   R1,>0200     READ OPCODE
       BL   @DSOP3       SUBROUTINE PERFORMS READ
       JNE  RDNM1        JUMP IF NO ERROR
       B    @REDERR      ELSE REPORT ERROR
RDNM1  LI   R0,>1000     VDP BUFFER
       MOV  R0,R6        SAVE ADDRESS IN R6
       INCT R6           ADD 2 TO R6
       BLWP @VSBR        READ LENGTH OF DISK NAME
       MOVB R1,@DNAME    MOVE TO STORAGE
       MOVB R1,R2        AND TO R2
       SRL  R2,8         RIGHT JUSTIFY
       MOV  R2,R4        STASH IN R4
       JEQ  CLRROW       IF ZERO, SKIP AHEAD
       INC  R0           R0 POINTS AT NAME ITSELF
       LI   R1,DNAME+1   TEXT STORAGE
       BLWP @VMBR        READ THE DISK NAME
CLRROW CLR  R0           SCREEN ORIGIN
       LI   R2,40        40 CHAR ROW
       BL   @CLFLD       CLEAR TOP ROW OF SCREEN
       CLR  R5           R5=0
       LI   R1,DNAME     POINT AT STASHED DISK NAME
       BL   @DISSTR      DISPLAY THAT
       A    R4,R5        ADD SAVED LENGTH TO R5
NONAM  MOV  R6,R0        GET R6 BACK INTO R0
       A    R4,R0        ADD LENGTH OF DISK NAME
       AI   R0,9         SKIP OVER THE F.P TYPE NUMBER
       LI   R1,>835C     POINT AT F.P. ARGUMENT IN PAD
       LI   R2,8         EIGHT BYTES IN F.P. NUMBER
       BLWP @VMBR        READ TOTAL CAPACITY ON DRIVE
       AI   R0,9         AHEAD TO NEXT F.P. NUMBER
       MOV  R0,R6        SAVE R0 IN R6 AGAIN
       LI   R1,>834A     POINT AT F.P. ACCUMULATOR IN PAD
       BLWP @VMBR        READ EIGHT BYTES (SECTORS FREE)
       BLWP @XMLLNK      USE XMLLNK
       DATA >0700        SUBTRACT FAC FROM ARG (RESULT AT FAC)
```

```
       LI   R0,USDNUM   SECTORS USED STORAGE
PUTUSD MOVB *R1+,*R0+   COPY A BYTE FROM FAC TO USED
       DEC  R2          DEC COUNT
       JNE  PUTUSD      REPEAT IF NOT 0
       MOV  R5,R0       GET SCREEN LOCATION BACK FROM R5
       BL   @DISFPN     SUBROUTINE DISPLAYS FLOATING POINT # (SECTORS USED)
       INC  R0          POINT AHEAD ONE SPOT
       MOV  R0,R5       SAVE ADDR IN R5
       LI   R1,USDSTR   "USED"
       BL   @DISSTR     DISPLAY THAT
       A    R2,R5       ADD LENGTH
       MOV  R6,R0       GET FILE RECORD ADDRESS BACK
       LI   R2,8        EIGHT BYTES
       LI   R1,>834A    FLOATING POINT ACCUMULATOR
       BLWP @VMBR       SECTORS FREE NUMBER AGAIN
       LI   R1,FRENUM   AND OUR STORAGE FOR LATER
       BLWP @VMBR       READ INTO STORAGE
       MOV  R5,R0       SCREEN ADDRESS BACK IN R0
       BL   @DISFPN     DISPLAY FLOATING POINT #
       INC  R0          POINT AHEAD ONE
       LI   R1,FRESTR   "FREE"
       BL   @DISSTR     DISPLAY THAT
*
* THAT'S IT
```

1.78. The Art Of Assembly — Part 78. Selective Cataloging

By Bruce Harrison

This time we're doing disk cataloging, but in a special way. The idea is simple enough, in that we want to read the catalog of a disk and report out only those files suitable for use with a particular program.

Our friend Lew King inspired the first example for this treatise, when he said that he'd used our Font Designer to create many special fonts for his 24-pin printer, but had caused himself a problem in using the program. That is, he had trouble remembering the names of the font files he has on his SCSI drive. Lew suggested we add a catalog function to that program, so he could pick one for loading right off the catalog screen.

1.78.1. Ending With A Dot

To make the idea as user-friendly as possible, we used the regular FILE NAME input routine that was already in the program, but added a new feature. If you know what file name you want, you simply type in the whole name and that file gets loaded. If you don't, you just enter a "path" name (e.g. DSK1.) and press **ENTER**. The program sees that the last character in the name you entered is a period, and catalogs that path for you. This works for floppy disks, Ramdisks, and hard drives, including the SCSI type. In Lew's case, he could type SCS1.FONT. at the prompt, and the program catalogs the FONT sub-directory on his SCS1 drive.

When we take this directory, we first check the type of each entry read. If the type is not 2, we know this is not a Display/Variable file, so we don't need to include it in our list. Next we skip over to the record size, and check for a record size of 120. If that doesn't match, then this file can be rejected as not suitable for use with the Font Designer program. What we're left with is a list of only those files that are the correct type and record size for our program.

On the screen, Lew gets a list of file names in one to three columns on the screen, but only those that are of the D/V 120 type, which means they are Font files for use with our Font Designer program. The cataloging continues until the end of the Catalog file is found, which means we're through reading the catalog entries. That's when we proceed to put the selection cursor on the screen, and let the user select any file from that path for loading. By simply moving the cursor next to a file name and pressing **ENTER**, Lew gets his desired Font file to load. We've added a similar catalog function to the QDUMP program on that disk, which is just for downloading fonts to the printer without editing.

In the Sidebar (Part One) is a portion of the source code that shows how each catalog file record is read and how we select only the appropriate files. The name for each file (padded to 10 characters if necessary) gets tucked away in a temporary name location. There are three floating point numbers in the catalog record. The first is the file type, which we check for the value 2. If the type is not 2, we skip to the next record in the file without further checking. The second number is the file size in sectors, which we ignore, and the third is the max record size for the file, and this gets checked for the value 120. If the file type is 2 AND the max record size is 120, then the name gets put on the screen. Otherwise we just try the next file in the catalog.

1.78.2. Flushed With Success

Having gotten that to work correctly, we decided to tackle a somewhat more difficult case, in our AMS Video Titler program. This case was more complicated because the files that can be used by this program are of two kinds. Both kinds are of the type 5 (Program, a.k.a. Memory Image). The TI-Artist format pictures always have a file name ending in _P, and are always 25 sectors in size. The Harrison Drawing type picture files don't have the _P at the end of their names, but are always 43 sectors in size.

To accept both of these kinds while rejecting all other kinds of files, we use a process similar to what was done for the font files, but with a twist or two. First the name gets checked to see if the _P is at its end. If that's so, we set a flag for use later in the selection process. Next, we check for type, and if that's not 5, we reject the name. Finally, if the type is correct, we check that flag and compare the size in sectors to either 25 or 43. Those conversant with source code will notice that this code also checks for sector size 24 and 42. That's needed because on a SCSI drive, sector sizes are reported as one smaller than on other drive types. We think that's because the designers forgot to add one for the directory sector that's used for each file, but not included in the number contained in that sector. Which number is checked depends on the state of the flag. If the tests are all passed, the file name gets listed on the screen.

The code that does this is shown in Part two of the Sidebar. It's similar in many respects to the code in part 1, but shows the checking of the name for _P endings and the checking for file size in sectors, which is important as a criterion in this program.

Again when all of the catalog has been read, the catalog file is closed and a flashing cursor is placed next to the first name on the screen. **ENTER** selects that file for loading, and it gets loaded into the current "frame" in the AMS memory.

1.78.3. Another Tough Case

The AMS slideshow program was originally designed for use with floppy drives. Its catalog function simply filters by the presence or absence of the _P on a file's name, meaning a TI-Artist format picture. Since each TI-Artist Picture file takes up 25 sectors, and since floppy disks generally don't have more than 1440 sectors total (DSDD), and since most TI-Artist pictures have an associated 25 sector color file (_C) on the same disk, there wouldn't be many cases where more than 28 picture files would be kept on one disk. The size of the catalog list was then arbitrarily set at 46.

Nobody objected to that until people with SCSI drives showed interest in the AMS Slideshow. On a SCSI drive, one can't have more than 127 files in a sub-directory, but that's the only limit. It was thus possible to have 63 color TI-Artist pictures or 127 black and white ones able to be read from a single directory. What, then, would we do with the file names once we were past 46? The answer was: Page 1 of the AMS. Page one of the AMS provides 4096 bytes of storage space If we find a 47th, then that whole screen gets stashed away in Page 1 (addressed as >A000 thru >AFFF). The screen then clears and there's room for another 46 names. If there's a 93rd name, this process repeats, storing the second screen starting at >A300, and continuing right up to the 127th name. The user can then switch back to previous screens by **CTRL E** and forward to later screens by **CTRL X**. The selection process works much the same as the older version, allowing selection of names until enough have been selected to completely fill the AMS Card's pages from 4 thru the end. In the case of 1 Meg AMS cards, that would be 84 slides in the sequence.

1.78.4. While We Were There

In the original Slideshow and the earlier AMS versions, if one wanted each slide to remain on screen for a particular amount of time, one could enter a time delay factor from 0.1 second through 300 seconds. When first we developed the AMS version, Jim Krych asked whether one could put a zero in for the time delay, so that slide changing would happen as fast as possible. At that time, the answer was no, because we'd put in some protection against time entries shorter than 0.1 second.

While we were revising the code, though, we looked again at how the timing was done, and determined that using zero as the delay time would do no harm. We therefore took out the lower limit protections, so one can now either leave the time entry field blank or put in a zero with the same result, and the slides will change as rapidly as possible.

1.78.5. Two More "Goodies"

After putting in the "zero time" feature, we thought it might be nice if the user could pause the show at any frame and then allow it to continue when he'd finished looking at that picture. There was already a keyboard scan included in the timing loop, but it looked only for FCTN-9 to stop the showing and return to the catalog list. By adding just a couple of lines of code, we set this up so that when in the timed mode, one can press and hold the **SPACE BAR**, and the current picture will remain until the **SPACE BAR** is released. While the **SPACE BAR** is down, time stops counting, and resumes upon its release. Thus if the time set were two seconds and the user pressed the **SPACE BAR** after one second, the picture would remain for its another second after the **SPACE BAR** was released.

The other minor annoyance we encountered was the business of "How many have I already selected?" This was especially difficult with more than one screenful of names, as we'd have to scan through the screens and count the selection marks. Not good! One can easily lose count and have to start over. We then went back to our bag of tricks and pulled out a small subroutine used in SCSICAT. By slightly modifying that routine, we were able to put the number of slides selected in the upper right corner of the screen. This number goes along when screens change, so that at any time while selecting, the user has the number selected readily available on the screen.

The number increments each time a selection is made, and decrements each time one is deleted, so it always tracks the current number. For our 256K card, that number won't go past 20, at which point selection is disabled because we've selected enough to totally fill the memory. For those with 512K capacity, the number will go to 41, and for those 1 meg users it will go to 84.

All of these upgrades are available for the sum of $1.00, including media and S&H. In the case of the two AMS programs, you must be a current owner of the program to get the upgrade, but the Font Designer is Public Domain, so it's not limited to current owners. To become a current owner of either AMS program (TITLER or SLIDESHOW) will cost you $5.00, including S&H.

```
* SIDEBAR 78
* SOME CODE FRAGMENTS
* TO ILLUSTRATE SELECTIVE CATALOGING
* BY BRUCE HARRISON
* PUBLIC DOMAIN
*
* PART ONE - FROM FONT DESIGNER
* STARTS AT THE INPUT FILE NAME SECTION
*
INFNT  BL   @CLS          CLEAR THE SCREEN
       LI   R0,40*10+9    ROW 11, COL 10
       LI   R1,EFNSTR     "ENTER FILE NAME"
       BL   @DISSTR       DISPLAY THAT
       LI   R0,40*12+1    ROW 13, COL 2
       LI   R1,IPABDT+9   OLD FILE NAME
       BL   @DISSTR       DISPLAY
INFBP  BL   @BEEP         SOUND BEEP
       BL   @ACCEPT       ACCEPT INPUT
       DATA 40*12+1       ROW 13, COL 2
       DATA 38            38 CHARS
       DATA 0             DON'T CLEAR FIELD
       DATA IPABDT+9      RESPONSE ADDRESS
       CI   R8,15         FCTN-9?
       JNE  INOF          JUMP IF NOT
       CLR  @IPABDT+8     CLEAR FILE NAME
       B    @MENU0        BACK TO MENU
INOF   MOV  R2,R2         NAME LENGTH 0?
       JEQ  INFBP         IF SO, TRY AGAIN
       DEC  R1            R1 POINTS AT IPABDT+9
       MOVB *R1,R3        NAME LENGTH TO R3
       SRL  R3,8          RIGHT JUST.
       A    R1,R3         ADD R1 SO R3 POINTS AT LAST CHAR IN NAME
       CB   *R3,@PERIOD   IS THAT A PERIOD?
       JNE  INOFA         IF NOT, JUMP AHEAD
       B    @INFCAT       ELSE CATALOG PATH
INOFA  LI   R9,IPABDT+9   TAKE INPUT NAME
       LI   R10,OPABDT+9  TO OUTPUT NAME
       BL   @MOVSTR       COPY THE STRING
       LI   R1,IPABDT     POINT AT INPUT PAB
```

```
       BL   @OPNF         OPEN THE FILE
       JNE  INP1          JUMP IF NO ERROR
* code omitted here that reads the
* font file
* Starting at INFCAT is the stuff that catalogs path
*
INFCAT MOV  R1,R9         COPY R1 TO R9
       LI   R10,DIRPAB+9  POINT AT DIRECTORY PAB NAME
       BL   @MOVSTR       COPY STRING
       BL   @CLS          CLEAR THE SCREEN
OPN0   LI   R1,DIRPAB     DIRECTORY PAB DATA
       CLR  R9            R9=0
       MOV  @ONE,@INSFLG  SET INSERT FLAG
       BL   @OPNF         OPEN CATALOG FILE
       JNE  AMB0          JUMP IF NO ERROR
       B    @INFNO        ELSE BRANCH TO ERROR TRAP
AMB0   LI   R13,2         R13=2
       LI   R14,14        R14=14
AMB1   BL   @REDCAT       READ A CATALOG RECORD
       JNE  AMB2          JUMP IF NO ERROR
       SRL  R0,8          RT. JUST R0
       CI   R0,5          IS THAT END OF FILE?
       JEQ  ALBCLS        IF SO, JUMP TO CLOSE FILE
       BL   @CLOSF        ELSE CLOSE ANYWAY
       B    @INPERR       BRANCH TO ERROR REPORT
AMB2   LI   R0,>1080      POINT AT LENGTH OF NAME BYTE
       BLWP @VSBR         READ THAT TO R1
       MOVB R1,R2         MOVE TO R2
       JEQ  ALBCLS        IF ZERO, WE'RE DONE
       SRL  R2,8          RT. JUSTIFY
       LI   R6,10         R6=10
       LI   R4,TEMFN      TEMPORARY FILE NAME
       INC  R0            ADD ONE TO R0
AMB2B  BLWP @VSBR         READ A BYTE FROM NAME
       MOVB R1,*R4+       PUT INTO TEMFN
       INC  R0            NEXT SPOT IN VDP
       DEC  R6            DEC R6
       JEQ  AMB2A         EXIT IF ZERO
       DEC  R2            DEC ACTUAL LENGTH
       JNE  AMB2B         READ NEXT IF NOT 0
AMB2BA MOVB @ANYKEY,*R4+  SPACE INTO TEMFN
       DEC  R6            DEC MAX COUNT
       JNE  AMB2BA        REPEAT IF NOT ZERO
AMB2A  LI   R6,3          THREE NUMERIC VALUES IN RECORD
REDNUM INC  R0            POINT PAST LENGTH BYTE (ALWAYS 8)
       LI   R1,>834A      POINT AT FAC IN RAM PAD
       LI   R2,8          8 BYTES TO GET
       BLWP @VMBR         READ F.P. NUMBER TO FAC
       BLWP @XMLLNK       USE XML ROUTINE
       DATA >1200         CONVERT F.P. TO INTEGER
```

```
       MOV  *R1,R5        MOVE INTEGER TO R5
       A    R2,R0         ADD 8 TO R0
       CI   R6,3          IS THIS THE FIRST PASS?
       JNE  CHKR61        IF NOT, JUMP
       ABS  R5            TAKE ABSOLUTE VALUE OF R5
       ANDI R5,7          LIMIT TO 7
       CI   R5,2          IS IT 2 (IF 2, THEN IT'S D/V TYPE)
       JNE  AMB1          IF NOT, SKIP THIS RECORD
CHKR61 CI   R6,1          IS THIS THE THIRD PASS?
       JNE  RDEC6         IF NOT, SKIP
       CI   R5,120        IS RECORD LENGTH 120?
       JNE  AMB1          IF NOT, SKIP RECORD
       INC  R9            R9 COUNTS FILES
       MOV  R0,R3         SAVE R0 IN R3
       MOV  R13,R0        COPY R13 TO R0
       LI   R1,TEMFN      POINT AT TEMP FILE NAME
       LI   R2,10         10 CHARACTERS
       BLWP @VMBW         WRITE TO SCREEN
       MOV  R3,R0         GET OLD R0 BACK
       AI   R13,40        MOVE DOWN ONE ROW
       CI   R13,960       PAST SCREEN BOTTOM?
       JLT  RDEC6         IF NOT, JUMP
       MOV  R14,R13       ELSE SET FOR NEXT SCREEN COLUMN
       AI   R14,12        ADD 12 FOR NEXT TIME
       CI   R14,40        CHECK 40
       JGT  ALBCLS        IF >, CLOSE FILE
RDEC6  DEC  R6            DEC NUMBER IN R6
       JNE  REDNUM        IF NOT 0, READ ANOTHER NUMERIC
       JMP  AMB1          ELSE READ NEXT RECORD
ALBCLS BL   @CLOSF        CLOSE THE FILE
       MOV  R9,@NFILES    SAVE R9 AS NUMBER OF FILES
       JNE  SELFIL        IF NOT 0, JUMP
       B    @NFLS         ELSE REPORT NO FILES FOUND
SELFIL LI   R3,1          START AT TOP OF SCREEN
       MOV  R3,R4         COPY TO R4
SELF0  AI   R4,12         ADD 12
SELF1  MOV  R3,R0         PUT R3 IN R0
       BL   @KEYBLN       KEY IN WITH BLINK
       BL   @KEYDLY       SHORT DELAY TO PREVENT RUNAWAY
       CI   R8,15         FCTN-9?
       JNE  SELCKX        IF NOT, JUMP
       B    @INFNT        ELSE BACK TO START
SELCKX CI   R8,'X'        CAPITAL X?
* from here on is code to allow user to
* select a file for loading
*
* END OF PART ONE
*
* PART TWO - FROM AMS TITLER
AMB0   LI   R13,2       ROW 1, COL 3
```

```
       LI   R14,14      R14=14 FOR LATER
AMB1   BL   @REDCAT     READ A RECORD
       JNE  AMB2        JUMP IF NO ERROR
       SRL  R0,8        RT. JUST R0
       CI   R0,5        END OF FILE?
       JEQ  ALBCLS      IF SO, CLOSE
       BL   @CLOSF      ELSE CLOSE ANYWAY
       LI   R1,RDRSTR   READ ERROR
       BL   @ERR40      REPORT
       B    @RELOAD     BRANCH BACK
AMB2   CLR  @PGNUM      PGNUM=0
       LI   R0,>0C80    POINT AT FILE BUFFER
       BLWP @VSBR       READ NAME LENGTH
       MOVB R1,R2       COPY TO R2
       JEQ  ALBCLS      IF ZERO, CLOSE FILE
       SRL  R2,8        RT. JUST
       LI   R6,10       MAX NAME LENGTH
       LI   R4,TEMSTR   TEMPORARY STRING
       MOV  R2,R5       SAVE LENGTH IN R5
       INC  R0          POINT AT 1ST CHAR
AMB2B  BLWP @VSBR       READ
       MOVB R1,*R4+     STASH
       INC  R0          NEXT CHAR
       DEC  R6          DEC 10 COUNT
       JEQ  AMB2A       JUMP IF ZERO
       DEC  R2          DEC LENGTH
       JNE  AMB2B       REPEAT IF NOT 0
AMB2BA MOVB @ANYKEY,*R4+ SPACE FILL
       DEC  R6          TIL END OF 10 CHARS
       JNE  AMB2BA
AMB2A  LI   R7,TEMSTR   POINT R7 AT TEMSTR
       DECT R5          DEC LENGTH BY 2
       JLT  AMB1        IF <0, JUMP
       A    R5,R7       POINT AHEAD TO NEXT-TO-LAST
       LI   R3,UNDP     POINT AT _P
       CB   *R7+,*R3+   CHECK _
       JNE  AMB2C       IF NOT, JUMP
       CB   *R7,*R3     CHECK 'P'
       JNE  AMB2C       IF NOT, JUMP
       SETO @PGNUM      SET PGNUM AS FLAG
AMB2C  LI   R6,2        TWO NUMBERS TO CHECK
       CLR  R7          R7=0
REDNUM INC  R0          POINT AT F.P. NUMBER
       LI   R1,>834A    AND AT FAC
       LI   R2,8        8 BYTES
       BLWP @VMBR       READ F.P. NUMBER TO FAC
       BLWP @XMLLNK     USE XML
       DATA >1200       CONVERT F.P. TO INTEGER
       MOV  *R1,R5      MOVE TO R5
       A    R2,R0       ADD 8 TO R0
```

```
       CI   R6,2         FIRST NUMBER?
       JNE  CHKR61       IF NOT, JUMP
       ABS  R5           ABSOLUTE VALUE R5
       ANDI R5,7         LIMIT TO 7
       CI   R5,5         IS THIS PROGRAM TYPE?
       JNE  AMB1         IF NOT, SKIP RECORD
CHKR61 CI   R6,1         SECOND NUMBER?
       JNE  RDEC6        IF NOT, SKIP
       MOV  @PGNUM,R1    FLAG SET?
       JEQ  CHK43        IF NOT, CHECK 43
       CI   R5,25        ELSE IS FILE LENGTH 25?
       JEQ  REDIN9       IF SO, JUMP
       CI   R5,24        CHECK LENGTH 24
       JEQ  REDIN9       IF SO, JUMP
       JMP  RDEC6        ELSE JUMP AHEAD
CHK43  CI   R5,43        LENGTH 43?
       JEQ  REDIN9       JUMP IF SO
       CI   R5,42        LENGTH 42?
       JNE  RDEC6        JUMP IF NOT
REDIN9 INC  R9           R9=R9+1
       MOV  R0,R3        SAVE R0 IN R3
       MOV  R13,R0       SCREEN POSITION FROM R13
       LI   R1,TEMSTR    FILE NAME
       LI   R2,10        10 CHARS
       BLWP @VMBW        WRITE TO SCREEN
       MOV  R3,R0        GET R0 BACK
       AI   R13,40       ADD ONE ROW
       CI   R13,960      END OF SCREEN?
       JLT  RDEC6        JUMP IF LESS
       MOV  R14,R13      COPY R14 TO R13
       AI   R14,12       ADD 12 FOR NEXT TIME
       CI   R14,40       PAST RIGHT EDGE?
       JGT  ALBCLS       CLOSE FILE IF SO
RDEC6  MOV  R7,R7        R7=0
       JNE  AMB1         JUMP IF NOT
       DEC  R6           DECREMENT R6
       JNE  REDNUM       IF NOT 0, REPEAT
       JMP  AMB1         ELSE NEXT RECORD
```

1.79. The Art Of Assembly — Part 79. Playing It In

By Bruce Harrison

Back when we started on the revisions of MIDI-Master, there was a nagging void in all the improvements we wanted to make. Most "sequencer" programs, including the Cakewalk that we use on our PC, allow the user to get music into the system by simply playing that music on the MIDI instrument while its MIDI OUT port is connected to the computer's MIDI IN. Alas, that was not the case with MIDI-Master.

Mike Maksimik, the original author of MIDI-Master, had tried some experiments toward having the Play-In feature, but with less than acceptable results. After we completed both the AMS and non-AMS versions of the "new" MIDI-Master, it was very clear that Play-In could not be fitted into the structure of those programs, mainly because memory simply would not allow enough extra code, and still allow the entire high memory to be used as music data storage.

1.79.1. A Separate Program

Thus we started with the idea that this capability would have to be made available as a separate program, not part of the MIDI-Master per se, but able to be used ultimately with MIDI-Master. Through some selective dis-assembly and modification of code supplied by Mike, we were able to make the core of a Play-In program and got that core to work fairly quickly.

1.79.2. Borrowing A Concept

The problem facing us, once we were able to receive bytes through the MIDI port, was how to store those bytes in the TI's high memory. Many ideas were considered and rejected for various reasons, mainly because they lacked the necessary efficiency of memory use to be practical on the TI. As part of the "thinking through" process, we tried playing some music into Cakewalk on our ancient PC, and looked at what Cakewalk did with the data. Eureka! There on the PC screen was a concept just waiting to be "borrowed" for the TI.

Cakewalk keeps a metronome running while music is being played in. The metronome keeps track of musical time. When events (pressing or release of keys) are recorded, what goes into memory is the time of the event by measure, beat, and "tick" numbers. To make this simple, let's consider that we're working in 4/4 time, four beats per measure, and a quarter note gets one beat. In Cakewalk, a quarter note gets 120 "ticks" of internal time, while in MIDI-Master a quarter note is 48 "ticks". No matter this difference, the concept of putting an event into memory was important.

On the TI, our internal metronome counts ticks, beats, and measures. In the 4/4 case, 48 ticks constitute a beat, four beats make a measure. Each "tick" actually represents a large number of passes through a delay loop. Typically that "delay factor" is a number between 250 and 1000. When a note event happens, five bytes get written to the high memory. These are the current Measure, Beat, and Tick of the metronome, plus the note value and its "velocity" value. When a note key is struck, the "velocity" byte is always a number between 1 and 127, and when that key is released, the velocity value is 0. Note that nothing gets put into memory between note events. Thus the playing of a note, including its release, requires just ten bytes to get stored in memory.

In the program, before we actually start taking bytes from the MIDI Instrument, we provide the user with a number of prompts so that the parameters can be set. This starts with the DELAY FACTOR, with a default entry of 750 in place. Next comes the TIME SIGNATURE, with 4/4 as a default. Both can be changed easily by just typing in a different choice and pressing **ENTER**. The third prompt concerns the METRONOME. This defaults to ON, which is what we recommend. The metronome works by sending short bursts of sound via the TI Sound Chip. The starting beat of each measure is accented to make it easier to track measures by sound. Finally there's a prompt for ECHO SERVICE, with N as the default answer. In almost all cases N for NO is the correct answer. Some Yamaha model keyboards will require this service in one operating mode, but generally it is not needed. If your setup needs it, however, it's there by just pressing **Y** or **y** at the prompt. If the TI system being used for Play-In has the AMS card, that fact will be known to the program. Also known will be the number of 24K "groups" of memory available in that AMS card. If the AMS is present, then, there will be one final prompt, for the memory group to be used as high memory. This always defaults to the next group that doesn't already contain some music.

1.79.3. Gotta Be Quick

A good player on a keyboard can generate a lot of notes in very little time. These get sent out the MIDI port at a blazing 31,200 baud rate. To keep up with bytes coming into our RS-232 port at such a rate requires a whole different approach to using the RS-232 port. Using a concept borrowed from MIDI-Master, we activate the port by direct actions on the CRU lines, thus avoiding the delays that would be involved in using the RS-232 as a file device. The program "speaks to" the UART in the RS-232 card by what amounts to a direct line, so that bytes can be sucked into a workspace register in very little time. The loop that accepts bytes from the UART is a very tight and fast one. It uses mainly registers, which are in the >8300 memory block (Ram Pad) so that all register actions happen as fast as possible. There are no BLs or BLWPs in this loop, just "straight line" code. The time counts for Measure, Beat, and Ticks are kept in the left bytes of registers, etc.

In the early stages of development, we tried putting a BLWP @KSCAN in the loop so that the user could stop input with a keystroke, but that took far too much time, and notes being played got lost. The current source, then, runs as fast as we could manage, and seems to catch all the notes even with a skilled pianist doing the playing. Our main testers for the program have been Richard and Valeda Bell, of Staten Island. Their advice and encouragement have contributed in a very large way to the development of the program. Encouragement has also come from Mike Maksimik, without whose original MIDI-Master source code none of this would be possible.

Today's Sidebar is the portion of code that handles the playing in of notes and storing them in memory. The program resides in low memory, so all of the note events are stored in high memory. You'll see that this code has been made to operate very quickly, with almost everything governed by the values in registers. The workspace is at >8300, just to make all these registers quicker for access.

1.79.4. Once It's In The Memory . . .

Having solved the major problem of taking bytes from the MIDI keyboard into the memory in an organized and efficient manner, we had two other problems to solve. First was "How do we know when the player has stopped playing?" After a lot of thought, we decided that if two complete measures passed by without any note events (note struck or note released) we would simply stop and return to the program's menu. We reasoned that there are almost zero musical works in which the instrument is at rest for two complete measures. (There is a case in the very comical Hoffnung Festival records where a mythical piece has two measures of silence, and "The first measure of silence is in 4/4 time, while the second is in 8/8 time.") We don't think anyone will be playing in mythical pieces from Hoffnung, so the two measure silence criterion has stayed.

So now there's music in the memory as played in by the family musician. What can we do with it? The first and most obvious thing is to simply play the music back through the instrument. For that, we've got Option 2 on our menu, Play Back. Since we might want to hear the music at a faster or slower tempo than what was played in, there's a prompt for the DELAY FACTOR, with a default the same as the play-in already in place. Once the prompt has been answered, the computer starts a count of measures, beats, and ticks running, but without making sound. Instead, it keeps comparing its present "time count" to the next event stored in memory. When the time matches, that event gets sent out to the MIDI instrument. Thus each event goes out at the relative time of its storage, and play of notes is paced just like the original playing. The relative timing is accurate to the nearest tick, which is 1/48th of a quarter note duration. That's the same "resolution" as used in MIDI-Master.

1.79.5. Saving Your Work

There's always the possibility that you might not want to keep your TI system up and running 24 hours a day just to keep that little minuet in memory, so we added two more menu choices to the list. SAVE IMAGE takes the music from memory and puts it out to a disk file just as it sits in memory. The one exception is the six bytes from >A000 through >A005. The actual music stores starting at >A006. The bytes before that are filled up with first the delay factor in use, then the beat length in ticks, and finally the number of beats per measure. This means that the output file will store those "vital stats" about this piece, so that when it's later brought back in, the tempo and time signature will be set correctly for playing.

The number and size of files created by SAVE depends on how much music was played in. Only that memory which was filled with music will be saved to one or more files. This can range from just one file of two sector size through a whole series of files of 33 sector size. Because we've used a whole new way of storing what's played in memory, the files saved by this program in its own memory-image format are NOT compatible with any of the versions of MIDI-Master, nor are files saved by MIDI-Master compatible with this program. The ones saved by this program can of course be re-loaded by this program and played back by it, and can also be used to create source files (see below) which ARE compatible with the updated versions of MIDI-Master.

1.79.6. Loading Saved Works

Item 4 from the menu is for loading a previously saved file or series of files. As with saving, just type in the name of the first file in the series, and the program will keep loading until it finds a non-existing member of the series, then it will return to the menu.

1.79.7. The De-Compiler

Item 5 from the menu, CREATE SNF, is the "payoff" for having played some music into this program. This option takes the music in memory and creates from it one or more D/V 80 files in SNF source format. These files are designed for use with the updated versions of MIDI-Master, either version 2.5Z or 2.5A. If a work is too long to make the source file a length that's "editable" on the TI, the output file will be split into a series of such files, each of which is of a size that can be edited with the E/A Editor, with the TI-Writer Editor or with either of the Funnelweb Editors. After editing, these files can be re-combined using our TOOL2UT program into a master file for the MIDI-Master compiler.

The idea here was to allow the user, having gotten a piece into the "system" through play-in, to be able to revise and refine his work. In these editable source files, such things as note durations and rests can be changed as desired. Mistakes made in the playing can be corrected, patch changes can be introduced to change instruments, and so on. In other words, Play-In gives the user a solid and consistent starting point for making a really finished musical work available to MIDI-Master. With the new versions of MIDI-Master, there are numerous possible additions that can be made. For example, with the "Voice Volume" feature, the playing volume of each track in the composition can be tailored just as desired by the musician. Since the music has been played in by a musician, you might want to leave the numeric note and rest durations just as they were, since this will leave the finished work with a more "human" quality than it would have if "perfected" by edits.

1.79.8. Loop Closed

This addition to the MIDI-Master "family" of programs closes the loop. You can now start with music that's played in, make SNF source files from that, edit those files to your own satisfaction, and then compile these SNF files through the new editions of MIDI-Master. Play-In is available to all MIDI-Master owners for the paltry sum of $5.00, including S&H.

See you again in two months.

```
* SIDEBAR 79
* SNIPPET FROM PLAY-IN
* CODE BY BRUCE HARRISON
* PUBLIC DOMAIN
*
* FIRST, THE SETUP SUBROUTINE IS USED
* TO SET THE RS232 FOR MIDI BAUD RATE
*
GOSET  BL   @SETUP          SET UP RS232
**********************
* THE NEXT PART SETS UP THE INTERNAL
* SOUND CHIP FOR THE METRONOME SOUNDS
**********************
       LI   R2,BEEP         OUR BEEP SOUNDS
       LI   R4,6            SIX BYTES
SNDBEP MOVB *R2+,@>8400     SEND TO SOUND CHIP
       DEC  R4              DEC COUNT
       JNE  SNDBEP          REPEAT IF NOT 0
**********************
* NEXT PART SETS UP INITIAL CONDITIONS
* FOR PLAYING IN MUSIC
**********************
       MOV  @DELAY,R13      DELAY FACTOR IN R13
       LI  R15,>0100        LEFT BYTE R15=1
       SETO R0              R0=>FFFF
**********************
* R5 SET TO -4 TO SERVE AS A COUNTDOWN
* IF NOTHING GETS PLAYED IN
* R0 WILL BECOME 0 ONCE SOMETHING'S PLAYED
**********************
       LI   R5,-4           R5=-4
       CLR  R1              R1=TOGGLE FLAG
       LI   R2,>B090        CTRL & CHAN1
       MOV @BEATL,R9        BEAT LENGTH IN HIGH BYTE R9
       LI  R3,BUFFER        POINT AT >A006
       CLR R6               MEASURE = 0
       MOV @CARD,R12        CARD CRU ADDR IN R12
       SBO 7                TURN ON LED
       MOV @UART,R12        UART CRU ADDR IN R12
********************
* FROM HERE ON, THE ACTIVITY LIGHT ON
* THE RS-232 STAYS ON, AND R12 HAS THE
* CRU ADDRESS FOR THE UART CHIP ON THAT CARD
********************
       MOV @ECHFLG,R11      R11 SIGNALS ECHO
NMEAS  CLR R7               BEAT 0
       CLR R4               R4=0
       MOV R3,@OLD3         SAVE MEMORY POINTER
******************
* AT THE START OF EACH BEAT, THE SOUND CHIP
```

```
* IS GIVEN A VOLUME BYTE SO THAT IT WILL
* SOUND AS A METRONOME FOR ONE TICK DURATION
* THE START BEAT OF EACH MEASURE IS LOUDER
* THAN THE OTHER BEATS. (WHEN R7=0)
*******************
NBEAT  MOV R7,R7            IS R7=0?
       JNE  BEAT            JUMP IF NOT
       MOVB @LOUD,@>8400    LOUD VOLUME TO SOUND CHIP
       JMP   CLRTIC         THEN JUMP
BEAT   MOVB @SOFT+1,@>8400  SOFT VOLUME TO SOUND CHIP
CLRTIC CLR R8               TICKS=0
*******************
* AFTER A BEAT HAS RUN ONE TICK,
* WHICH IS 1/48TH OF A QUARTER NOTE,
* THE SOUND CHIP IS SILENCED.
*******************
NTICK  CI  R8,>0100         TICKS=1?
       JNE LOAD10           JUMP IF NOT
       MOVB @SILENT,@>8400  SHUT UP METRONOME
       MOVB @SILENT+1,@>8400 BOTH GENERATORS
LOAD10 MOV  R13,R10         DELAY COUNT TO R10
*******************
* CODE HERE DETERMINES IF THE MIDI INSTRUMENT
* HAS SENT A BYTE, AND ACTS ACCORDINGLY
*******************
GETBYT TB  >15              TEST BIT FOR BYTE READY
       JNE NOBYTE           JUMP IF NO BYTE
       STCR R4,8            TAKE 8 BITS INTO R4
       SBZ  >12             SHUT OFF INPUT
       CB   R4,R2           LEFT BYTE R4=>B0?
       JEQ  CTRLPI          IF SO, INPUT PEDAL
       MOV  R4,R4           CHECK R4
*******************
* AT THIS POINT IF THE BYTE JUST TAKEN
* FROM THE INSTRUMENT IS EITHER A NOTE OR A VOLUME,
* R4 WILL BE EITHER 0 OR A POSITIVE NUMBER
* IF IT'S A NEGATIVE OTHER THAN >B0, WE'LL REJECT IT
*******************
       JLT  GETBYT          IF NEGATIVE, IGNORE BYTE
       MOV  R0,R0           IS R0=0?
       JEQ  MOV1            IF SO, JUMP
       CLR  R0              SET R0=0
       CLR  R6              SET MEASURE=0
MOV1   MOV  R1,R1           CHECK R1
       JNE  MOVR4           JUMP IF NOT 0
******************
* A NOTE EVENT COMES IN AS TWO BYTES, ONE FOR THE
* NOTE VALUE AND ONE FOR THE VELOCITY (A.K.A. VOLUME)
* R1 STARTS OUT 0, SO THE TIMING GETS SENT TO
* MEMORY BEFORE THE NOTE.  AFTER THAT HAPPENS,
```

```
* R1 IS INVERTED SO THAT THE VOLUME BYTE WILL
* GO INTO MEMORY JUST AFTER THE NOTE IT'S SUPPOSED
* TO ACCOMPANY.  AGAIN R1 GETS INVERTED AFTER THE
* VOLUME IS STORED, SO IT'S ZERO FOR THE NEXT NOTE
******************
       MOVB R6,*R3+         MEASURE TO MEMORY
       MOVB R7,*R3+         BEAT TO MEMORY
       MOVB R8,*R3+         TICK TO MEMORY
MOVR4  MOVB R4,*R3+         INPUT BYTE TO MEMORY
       CI   R3,>FFDE        MEMORY FULL?
       JGT  MEMER           IF SO, ERROR
******************
* INVERT R1 MEANS THAT R1 TOGGLES BETWEEN
* BEING ZERO AND BEING >FFFF (ALL BITS ON)
******************
       INV  R1              INVERT ALL BITS IN R1
       MOV  R11,R11         ECHO REQUIRED?
       JEQ  NOBYTE          IF R11=0, NO ECHO
******************
* FOR MOST KEYBOARDS, THIS NEXT SECTION WILL BE
* SKIPPED BECAUSE R11 WILL BE ZERO.
* FOR CERTAIN YAMAHA MODELS IN CERTAIN OPERATING
* MODES, R11 WILL BE NON-ZERO, SO THE BYTE JUST
* RECEIVED WILL BE SENT BACK OUT TO THE KEYBOARD
******************
       SBO  16              SET BIT FOR OUTPUT
ECLP1  TB   22              CHECK AVAILABLE
       JNE  ECLP1           REPEAT IF NOT
       LDCR R4,8            SEND 8 BITS TO OUTPUT
       SBZ  16              RESET OUTPUT BIT
NOBYTE DEC  R10             DECREMENT DELAY COUNT
       JNE  GETBYT          REPEAT IF NOT 0
******************
* WHEN A DELAY FACTOR CYCLE IS DONE, THE
* NEXT SECTION WILL INCREMENT THE TICK, BEAT, AND MEASURE
* COUNTS AS APPROPRIATE.
******************
INCTIC AB   R15,R8          ADD 1 TO TICKS
       C    R8,R9           EQUAL 1 BEAT?
       JEQ  INCR7           IF SO, JUMP
       JMP  NTICK           NEXT TICK
INCR7  AB   R15,R7          ADD 1 TO BEAT
       CLR  R8              TICKS=0
BPM1   CI   R7,>0400        = BEATS PER MEASURE?
       JEQ  INCR6           JUMP IF SO
       JMP  NBEAT           NEXT BEAT
INCR6  AB   R15,R6          ADD 1 TO MEASURE
       C    R3,@OLD3        HAS POINTER CHANGED?
       JNE  LEDIN           IF YES, JUMP
******************
```

```
* IF R3=OLD3, A MEASURE HAS PASSED WITHOUT ANY
* INPUT FROM THE KEYBOARD.  AT STARTUP, R5 WAS SET
* AT -4, SO R5 WILL BECOME POSITIVE IF WE'VE JUST
* STARTED AND FIVE MEASURES HAVE PASSED WITHOUT INPUT.
* IF THERE HAS BEEN INPUT, CONTROL WILL JUMP TO
* LEDIN, AND R5 WILL BE SET TO -1
* AFTER THAT, TWO MEASURES OF SILENCE WILL
* CAUSE R5 TO BE POSITIVE AND WE'LL EXIT
******************
       INC  R5              R5=R5+1
       JGT  EXIT            EXIT IF >0
       JMP  NMEAS           ELSE NEW MEASURE
LEDIN  SETO R5              R5=-1
       JMP  NMEAS           NEW MEASURE
*****************
* THE SECTION BELOW TAKES A CONTROL ACTION
* (USUALLY SUSTAIN PEDAL ACTION) INTO MEMORY
* THAT CONSISTS OF A >B0, >40, >7F STRING
* AFTER THAT STRING, IF A NOTE EVENT COMES IN
* DURING OUR TIMEOUT, IT WILL BE SENT STARTING
* WITH A >90 BYTE, WHICH WILL ALSO BE STORED
* AND WILL CAUSE EXIT FROM THIS LOOP
* IF TIMEOUT HAPPENS (R1 BECOMES 0) THEN THE
* >90 WILL BE SENT ANYWAY AND WE'LL RETURN TO
* THE NORMAL LOOP ABOVE.
*****************
CTRLPI SWPB R2              R2=>90B0
       MOV  R0,R0           R0=0?
       JEQ  CTRL0           JUMP IF YES
       CLR  R0              SET R0 TO 0
       CLR  R6              SET MEASURE TO 0
CTRL0  MOVB R6,*R3+         MEASURE TO MEMORY
       MOVB R7,*R3+         BEAT TO MEMORY
       MOVB R8,*R3+         TICKS TO MEMORY
       MOVB R4,*R3+         >B0 BYTE TO MEMORY
       LI   R1,50           TIMEOUT COUNT IN R1
CTRL1  TB   >15             BYTE READY?
       JNE  CTRL2           IF NOT, REPEAT
       STCR R4,8            BYTE TO R4
       SBZ  >12             RESET BIT >12
       MOVB R4,*R3+         BYTE TO MEMORY
       CB   R4,R2           WAS THAT >90?
       JNE  CTRL1           IF NOT, GET NEXT BYTE
CTR0A  SWPB R2              ELSE SWAP R2 TO >B090
       CLR  R1              R1=0
       JMP  GETBYT          BACK FOR NOTE BYTE
CTRL2  DEC  R1              DEC TIMEOUT COUNT
       JNE  CTRL1           REPEAT IF NOT 0
       MOVB R2,*R3+         PUT >90 IN MEMORY
       JMP  CTR0A           THEN JUMP
```

```
******************
* WE GET TO EXIT IF EITHER OF TWO THINGS HAVE
* HAPPENED.  IF WE STARTED THE LOOP AND NOTHING
* CAME IN FROM THE KEYBOARD FOR FIVE MEASURES,
* WE EXIT.  IF SOMETHING WAS PLAYED IN AND THEN
* TWO COMPLETE MEASURES HAVE SEEN NO INPUT, WE
* EXIT.
******************
EXIT   MOV  R3,@ENDMUS      SAVE ENDING POINTER
       MOV  @CARD,R12       CARD CRU ADDR IN R12
       SBZ  7               SHUT OFF LED
       MOVB @SILENT,@>8400  SILENCE METRONOME
       MOVB @SILENT+1,@>8400 BOTH GENERATORS
       B    @MENU           BACK TO MENU
```

1.80. The Art Of Assembly — Part 80. What's A NewLine?

By Bruce Harrison

Is this number 80? How can that be? How can he keep on cranking these out? Well, besides being a pretty "cranky" old guy, it helps if the Assembly guy keeps writing new Assembly programs on his TI. That way, there's always new ground to cover in this column.

1.80.1. The PC Connection Problem

The subject of today's column starts with a small change in our use of the PC computer. We have for a long time been "holdouts" from the world of Internet. That changed forever when our friend Lew King gave a presentation at the 1998 Lima MUG Conference. Lew used a TI computer and Term80 to connect to the Internet live during his lecture. Two things about this impressed my partner Dolores. First, just how easy it could be to get into various websites, and second, how even with a text-only connection one could reach so much nifty information. We talk to Lew on the phone frequently, but seeing his demo of internet access was a revelation. After the demo, we talked to Lew at length. Like us, Lew also has a PC at home, and he assured us that what he did from the TI could also be done very easily from our "ancient" Tandy 1000 SX PC.

Lew even found us a modem at a "Ham Fest" somewhere in his state (PA) and shipped it to us. He also provided a program called TELIX, made by a now-defunct Canadian company, that would allow our old PC to talk to the world as a VT-100 Terminal. Thus we're now able to get into that other world right from our living room. We can get to many resources, including the catalogs of various libraries, the articles in newspapers, scientific journals, and so on.

1.80.2. Meanwhile, There's Cakewalk (TM)

Dolores is our family musician. She's the one who did all that Assembly Language music for the sound chip, and who's done numerous pieces on MIDI-Master. She helped at every stage in the re-vamping of MIDI-Master by testing and suggesting improvements. Still, when she wants to program some MIDI music quickly, she uses the Cakewalk program on our Tandy 1000 SX. That program is far easier to use, and allows direct entry of music into a file that can be played instantly without having to be compiled. Hmmm. . . Maybe someday we'll have "son of MIDI-Master" with that capability on the TI. Hmmm. . . In some cases, she's done a musical work first on the PC using Cakewalk, then did the same work over again on MIDI-Master. That's a very hard way to do things. Starting completely over again on the TI is really not an acceptable way of doing things.

With Cakewalk, we have a little PC program called CAKE2ASC. That takes one of the .WRK files made with Cakewalk and converts it into an ASCII format, or in other words a human-readable file format. That being the case, she asked, why could we not take those ASCII files over an RS-232 connection into the TI and then have the TI convert them into SNF source files for MIDI-Master. The resulting files might need some editing to really sound right when compiled by MIDI-Master, but that would be lots quicker and easier than starting over from scratch.

1.80.3. Pre-AMS Ways

Some years back, we developed a little system called Smart Connect, to transfer text files between our PC and TI in both directions. Others, most notably Dr. Charles Good, came up with other ways of getting text files from PC to TI. He used Funnelweb's LOAD FILE by simply using a file name like RS232 etc. Still, there's always the problem that many of the files found on PCs are much too large to fit into memory on the TI. In our Smart Connect system, we used Basic on the PC to do the sending and receiving, and allowed the user at the TI to stop now and then and change file names. That worked, but it isn't really a sound solution.

So it was concluded that we needed something new to handle the rather large files created by CAKE2ASC. Using that, even for fairly short musical works, creates files in the 100 Kbyte range. Having done four programs already that use the AMS card to handle large amounts of data, the answer to handling the ASCII files from Cakewalk was obvious. Write a program that would take files of enormous size in through the RS-232 and store all those file records in the many pages of AMS memory.

Having had this brainstorm led within a few days to a program for our own use only, just to take those Cakewalk ASCII files over the 25-wire ribbon cable between our PC and TI. With the 256K AMS available, it was easy to take a 100 Kbyte file over the line. Once it was all in memory, our little program would write it out to disk in pieces of about 12 Kbytes each, so that each file on the TI side would be editable via the Funnelweb editors or TI-Writer. This worked well after we removed a couple of bugs. For the converted Cakewalk files, that was all we needed.

1.80.4. Once That Was Working. . .

As so often happens in these things we develop just for our own use, it occurred to us that with some slight changes it could be made into a more General Purpose tool, suitable for use by Lew King and perhaps others in the TI Community who have both TI and PC computers. Here's where that "NewLine" question comes into play.

When one captures or downloads text files through the Internet or from a BBS, the files usually contain what are called NewLine characters. These are no mystery, they're just Line Feeds without a Carriage Return. Putting in Line Feeds [CHR$(10)] makes it easier to send these files through various communication paths where Carriage Returns should not be sent except to signal the end of a record.

In editor programs like Funnelweb, however, we want a carriage return [CHR$(13)] as the end of a line and at the end of a file record. In some cases, one finds files in which there are both Carriage Returns and Line Feeds used together at the end of a line. When the file is being brought in from the RS-232, we simply take it as it comes, as a D/V 80 input. When we're writing it out to TI disk files, however, we run each input string through a "filter" process before it goes to the disk file.

1.80.5. The Filter Rules

Each character in each string gets examined to see if it's a Carriage Return or Line Feed. If it's neither of those, it goes out unchanged. If it's a carriage return, it gets changed to a space. If it's a Line Feed, (a.k.a. NewLine) several things happen. First, the Line Feed gets changed to a Carriage Return. Next, the current record up to and including this CR gets sent out as a record to the output file, then the remainder of the input string goes back to resume the filter process. If there are more LFs within the rest of the string, this process repeats until all of the input string has been examined and sent out. If there are no Line Feeds [as in those converted Cakewalk files] then each record (80 characters or less) gets sent to the disk file as-is, that is a record with its original length.

1.80.6. The Editing Process

Having this filter in the program makes the final editing process much easier. Let's say, for example, that what would normally be one line in a text file winds up split between two records in the output file, and that there was a NewLine at the end of the second part. In Funnelweb, we can correct this very easily by putting the cursor at the start of the first of those two lines and pressing **CTRL 2**. This will re-join to two lines into one if possible, ending its work at the CR which was a LF in the original file. If the split happened in the middle of a word, Funnelweb will leave a space in the middle of that word, which you'll have to delete with **FCTN 1**. If a line ended up with leading spaces in front of it, those leading spaces can be removed by just pressing **CTRL 2** with the cursor on that line. In short, having CRs where those LFs used to be makes cleaning up the file with Funnelweb's Text Edit much easier.

Today's Sidebar

The Sidebar today is a portion of the source code from the output part of the program. It shows the filtering process and the means used to write out part of an input record when we've found that LF character. It's fully annotated, so we probably don't need to discuss it in detail here.

1.80.7. The Disk Is Available

The disk that has this program is available either from me or from Lima for only $1.00. It's Public Domain, so you may also find a friend who can make you a copy or get it from your User Group's library. It's SS/SD format, so even those with only single sided drives can use it. As usual the disk contains complete instructions, a LOAD program to run it from XB if needed, and an XB program to print the instructions for you.

Note: Cakewalk is a registered trademark of Twelve Tone Systems, Inc.

```
* SIDEBAR 80
* PART OF OUR TRANSFER PROGRAM
* SOURCE CODE - NOT A COMPLETE PROGRAM
*
* THIS PART SAVES THE STRINGS FROM MEMORY
* INTO A SERIES OF DISK FILES
* IT USES NUMEROUS SUBROUTINES (NOT SHOWN)
* COMPLETE SOURCE IS ON THE TRANSFER P.D. DISK
*
* PUBLIC DOMAIN
* CODE BY Bruce Harrison
*
SAVE   MOV  @AMSFLG,R4   AMS IN USE?
       JEQ  SAVEN        IF NOT, JUMP
*
* IF AMS IS IN USE, THE STUFF BELOW
* RECORDS THE HIGHEST GROUP NUMBER THAT
* HAS CONTENT, THEN RESETS TO GROUP 1
* TO START THE OUTPUT FILES.
*
       MOV  @CURGRP,@HIGRP HIGHEST USED = CURRENT GROUP
       LI   R12,1        GROUP 1 (PAGES 4 THRU 9)
       BLWP @SETGRP      SET TO THAT GROUP
SAVEN  BL   @CLS         CLEAR SCREEN
       LI   R0,10*32+4   ROW 11, COL 5
       LI   R1,SPMSTR    SAVE NAME PROMPT
       BL   @DISSTR      DISPLAY
*
* THE SECTION STARTING AT SAV2 ACCEPTS FIRST OUTPUT
* FILE NAME FOR THE SERIES
*
SAV2   LI   R0,12*32+1   ROW 13, COL 2
       LI   R1,OUTPAB+9  OUTPUT FILE NAME STRING
       BL   @DISSTR      DISPLAY
       BL   @ACCEPT      ACCEPT FILE NAME
       DATA 12*32+1,30,0,OUTPAB+9
       MOV  R2,R2        NAME LENGTH 0?
       JEQ  SAV2         IF SO, TRY AGAIN
       CI   R8,15        FCTN-9 PRESSED?
       JEQ  EXIT2        IF SO, EXIT PROGRAM
       BL   @CLOSE       CLOSE INPUT FILE
       LI   R0,PABLOC    PAB VDP ADDR IN R0
       BL   @DSROP2      OPEN OUTPUT FILE
       JNE  LDR9         JUMP IF NO ERROR
       LI   R0,22*32+2   ROW 23, COL 3
       LI   R1,FNOSTR    "FILE DID NOT OPEN"
       BL   @DISSTR      DISPLAY
       BL   @KEYHNK      HONK & AWAIT KEYPRESS
       LI   R2,32        32 BYTES
       BL   @CLFLD       ERASE ERROR MESSAGE LINE
```

```
       JMP  SAV2          TRY AGAIN
LDR9   MOV  @WREFLG,R4    HAS THERE BEEN A WRITING ERROR?
       JEQ  LDR9A         IF NOT, JUMP
       MOV  @SAVR3,R9     GET OLD VALUE R9 BACK
       CLR  @WREFLG       CLEAR THE FLAG
       MOV  R9,R6         STASH R9 IN R6
       JMP  SAV6          JUMP AHEAD
LDR9A  LI   R9,>A000      START OF HIGH MEM
LDR9B  MOV  R9,R6         STASH R9 IN R6
       CLR  @PTTFLG       NOT PART TWO
SAV6   MOV  @PTTFLG,R4    IN PART 2?
       JNE  SAV6A         JUMP IF SO
SAV61  CI   R9,>D000      HALFWAY THROUGH HI MEM?
       JLT  SAV6A         SKIP IF LESS
       JMP  INCSAV        ELSE START NEW FILE
SAV6A  MOVB *R9+,R2       STRING LENGTH TO R2
       CB   R2,@NOKEY     IS THAT >FF?
       JEQ  NMSAVE        IF SO, JUMP
SAV6B  LI   R0,PABLOC+5   POINT AT PAB VDP ADDR +5
       MOVB R2,R1         LENGTH IN R1
       BLWP @VSBW         WRITE TO PAB+5 IN VDP
       SRL  R2,8          RT JUST R2
       MOV  R9,R1         MOVE R9 POINTER TO R1
       MOV  R9,R3         AND TO R3
       MOV  R2,R4         COPY LENGTH TO R4
*
* THE SECTION STARTING AT CMPLF IS THE
* "FILTER" THAT CHECKS FOR CR AND LF AND
* TAKES ACTION IF FOUND
*
CMPLF  CB   *R3,@LFBYTE   LINE FEED?
       JNE  CMPCR         IF NOT, JUMP
       MOVB @ENTERV,*R3   PUT CR THERE
       B    @PTREC        THEN WRITE PARTIAL STRING AS RECORD
CMPCR  CB   *R3,@ENTERV   CARRIAGE RETURN?
       JNE  SVINC3        ELSE JUMP
       MOVB @ANYKEY,*R3   PUT SPACE THERE
SVINC3 INC  R3            NEXT BYTE OF STRING
       DEC  R4            DEC LENGTH
       JGT  CMPLF         RPT IF >0
SAV7A  LI   R0,P2BUF      VDP OUTPUT BUFFER
       BLWP @VMBW         WRITE STRING THERE
SAV7   A    R2,R9         ADD LENGTH TO POINTER
SAV8   LI   R0,PABLOC     0TH BYTE IN PAB
       MOVB @WRITEF,R1    WRITE RECORD OPCODE
       BLWP @VSBW         WRITE TO VDP
       BL   @DSROP4       PERFORM DSR OPERATION
       JNE  SAV6          IF NO ERROR, REPEAT
WRTERR BL   @CLOSE        CLOSE FILE
       LI   R1,SVESTR     "SAVE ERROR"
```

```
SHWWRE LI   R0,22*32+2   ROW 23, COL 3
       BL   @DISSTR      DISPLAY
       BL   @KEYHNK      SEND "HONK", AWAIT KEY
       LI   R2,32        32 BYTES
       BL   @CLFLD       CLEAR AWAY MESSAGE
       MOV  R6,@SAVR3    STASH AWAY R6
       SETO @WREFLG      SET WRITE ERROR FLAG
       B    @SAV2        BACK TO FILE NAME
NMSAVE MOV  @AMSFLG,R4   AMS IN USE?
       JEQ  QUTSAV       IF NOT, JUMP
       MOV  @CURGRP,R12  CURRENT GROUP IN R12
       INC  R12          NEXT GROUP
       C    R12,@MAXGRP  CHECK MAXIMUM
       JGT  QUTSAV       IF GREATER, OUT OF HERE
       C    R12,@HIGRP   CHECK HIGHEST GROUP USED
       JGT  QUTSAV       IF GREATER, OUT OF HERE
       BLWP @SETGRP      SET NEW GROUP
       LI   R9,>A000     POINTER TO START
INCSAV MOV  R9,R6        STASH IN R6
       BL   @CLOSE       CLOSE CURRENT FILE
       BL   @INCNAM      INCREMENT NAME
       INV  @PTTFLG      INVERT PART TWO FLAG
       BL   @DSROP2      OPEN NEW FILE
       JNE  GOSAV6       JUMP IF NO ERROR   G
       BL   @CLOSE       CLOSE EVEN IF NOT OPEN
       LI   R1,FNOSTR    FILE DIDN'T OPEN
       JMP  SHWWRE       ISSUE ERROR
GOSAV6 JMP  SAV6         BACK TO SAV6
QUTSAV BL   @CLOSE       CLOSE FILE
       B    @MENU        BACK TO START
*
* SECTION STARTING AT PTREC TAKES THE
* PART OF THE CURRENT RECORD UP TO AND
* INCLUDING THE NEW LINE (NOW A CR)
* OUT AS A SEPARATE DISK RECORD
*
PTREC  INC  R3           PAST THE LF (NOW CR)
       DEC  R4           DEC COUNT
       JGT  PTREC1       IF >0, JUMP
*
* IF R4 HAS BECOME 0 HERE, THE CR IS AT THE END
* OF THE INPUT FILE RECORD, SO WE CAN PROCEED
* TO SEND AND THEN MOVE ON TO NEXT RECORD IN
* THE MEMORY
*
       B    @SAV7A       ELSE WRITE RECORD
PTREC1 MOV  R3,R7        R3 TO R7
       S    R9,R7        SUBTRACT R9 - R7=LENGTH
       MOV  R7,R2        COPY R7 TO R2
       LI   R0,PABLOC+5  PAB LENGTH BYTE
```

```
       SWPB R7           LEN IN HIGH BYTE
       MOVB R7,R1        COPY TO R1
       BLWP @VSBW        WRITE LENGTH TO PAB
       LI   R0,P2BUF     OUTPUT BUFFER
       MOV  R9,R1        R1=START OF CONTENT
       BLWP @VMBW        WRITE TO VDP BUFFER
       LI   R0,PABLOC    0TH BYTE OF PAB
       MOVB @WRITEF,R1   WRITE OPCODE
       BLWP @VSBW        WRITE THAT
       BL   @DSROP4      OUTPUT TO FILE
       JNE  PTREC2       JUMP IF NO ERROR
       B    @WRTERR      ELSE TO ERROR REPORT
PTREC2 MOV  R3,R9        R3 TO R9 POINTER
       MOV  R4,R2        R2=REMAINING LENGTH
       SWPB R2           IN LEFT BYTE R2
       B    @SAV6B       BACK TO MAIN PROCESS
* HERE, THE PROCESS RE-ENTERS THE MAIN
* PART TO PROCESS THE REST OF THE INPUT
* RECORD
*
* THAT'S IT
```

1.81. The Art Of Assembly — Part 81. Strange Happenings

By Bruce Harrison

Now that we're "connected" to the rest of the world via our beloved Tandy 1000 SX PC, and even have an E-mail address, we can experience things that never happened to us before. If you don't know what a Tandy 1000 SX is, perhaps a little story will make it clear.

In a phone conversation with our friend John Bull, we described our PC as being a "stone age" model. John asked if that meant it was a 286. 286? Try 8088! I guess it says something about the PC business that John would think my expression "stone age" would mean a 286. Later in the same phone call, John agreed with my putting the 286 in the "bronze age". Of course when we bought that Tandy 1000 SX brand new, it was the "latest and greatest" PC. In less than a year it was becoming obsolete.

Had an E-mail from Stephen Shaw (received over that same "stone age" Tandy) telling how his three year old PC is too old now, and can't run any of the newly-released software. Luckily for us, we've still got all that "stone age" software for our PC, and it still does what we want it to do. Won't run Windows (TM) (R) of course, and won't support Netscape, or run PC99, but it's still a useful and reliable tool. Most of the time it is, anyway.

1.81.1. But We Digress. . .

Now that we're "connected", we get to experience all the good things of Internet access, and some of the bad things, too. We can now capture text from the Internet onto our PC's hard drive by the ton. This creates two problems. First, what are we to do with it all. Some of it can be read on the screen using the TYPE with MORE command, but that gets tiresome. We can print it out, but then where do we store all that paper? In just a few days we're already suffering some information overload.

Then there are those "Frames". Many of today's web sites seem to consist of them, and these sites always remind us that we can't handle them. In many such cases, we're offered a download of a new browser. We know better than to try taking one of those offers. For one thing, there are only about 8 Megabytes left on our 10 Megabyte hard drive, and that's most likely not near enough for any browser we're offered. Also, of course, the preferred browser will expect us to be running under some version of Windows. No way, Jose!

Still, even as a VT100 terminal emulation, there are some good deals out there. We found a site with thousands of MIDI files available for download, and took some of those. Many of them worked okay with Cakewalk (TM) and played pretty nicely on our new Casio keyboard. But then there were some that sent "mystery" control codes to the Casio, and after a while these "locked up" our keyboard. We don't know why. Getting the MIDI keyboard unlocked required removing its battery power, letting it sit overnight, then sending it some music from MIDI-Master on the TI. Even then, Lory found some channels not responding. She wound up sending a "reset all controllers" code to each and every channel. That worked perfectly, and after that we could again use it with Cakewalk on the PC, provided we didn't play any of those downloaded MIDI files.

1.81.2. And Then Came E-mail

Shortly after getting equipped to "browse" via a service provided by the Maryland Public Library system, we found our way to Hotmail, established our E-Mail hookup, and opened yet another Pandora's box. E-mail now arrives from various people on a regular basis. Seems every day there are new messages in our in-box that must be dealt with. Most, fortunately, are of a nature that they can be read once or twice and then deleted. But what of the others? Stephen Shaw sent what he'd previously done by regular mail. Just three pages or so, but not something I'd throw away. Hotmail's help screens are not the most helpful. If I want to download Stephen's letter to my own hard drive, I find the menus on Hotmail give no indication how to do that. The only time "download" is mentioned anywhere on any of their displays is to offer me a browser! Of course since Hotmail is a Microsoft service, guess whose browser that would be, and guess what operating system it would need. Bill Gates rules the world. Actually there is one thing I can download from Hotmail. Michael Becker sent an e-mail with another file attached. In that case, Hotmail offers the option of downloading the attached file. Took me three tries to get it right, but I did manage to download that file.

In many such cases, I consult with Lew King. By ordinary voice phone, that is. Lew can usually help out very quickly that way, and we can reach a mutual understanding of what we're talking about much more easily when we're talking, as opposed to writing E-Mail back and forth. Still, the phone bill isn't affected by E-Mail, but those half hour conversations can add up. By the way, if you can help, that E-mail address is: rottencat13@hotmail.com. Seems rottencat was already taken, so we added 13 (a lucky number?) to have our own unique address. This probably proves that we're not the only people with E-mail who have a rotten cat. My partner, Dolores P. Werths, has her very own E-Mail address: Lorysmandolin@hotmail.com.

1.81.3. The List Server Saga

Thank you Tom Wills for providing the TI99 list server. This is a valuable service to the community. We, however, had some trouble getting started with it. In fact, as we write this, we're still not subscribed. On our first attempt we put a hyphen between TI and 99. WRONG! It has to be just TI99. We've tried twice more, but all we've gotten is return e-mail from majordomo@TheRiver.com. Each time this let us know that we were not subscribed. Since we didn't get anything from TI99 we could sort of figure that out.

On our fourth try we got everything right, and got subscribed on the List Server. Like the Internet itself, this is a mixed blessing. Yes, some valuable stuff shows up there, but one has to read a lot of less valuable stuff to sort it out.

But then that's life. One of our U.S. Senators here in Maryland has lots of stuff on her web site, including her favorite recipe for crab cakes. (Maryland is for Crabs - why else would I live here?) There's one other complaint while I'm being a Maryland crab. Some people don't answer their E-Mail. Of course some don't answer regular paper mail either. Mainly the non-answers have been people outside the TI realm. The TI people have been very good about responding, sometimes with more text than necessary, but at least they respond. Of course it's easier if I get their addresses right on the outgoing.

It's very easy working from a printed source to mistake a 1 (number) for an l (lowercase L). Such mistakes cause much grief, and of course the two characters are just as hard to tell apart on the computer screen as in print. (Maybe even more so.)

1.81.4. The Other Penalty

As this is written, it's been about a week since we started getting into the Internet, and less than that since we've had e-mail. Your author has not written a single line of source code in that period. There's a project "in work" for yet another new program for our beloved TI, but now that we're busy "browsing" web sites and sending/receiving e-mail, not one lousy line of source has made it into the project. Maybe on Saturday night. Today happens to be a Friday, and we don't want to miss this week's re-run of Sabrina, The Teenage Witch. We don't watch it for Sabrina, but for Salem the cat, who has all the best lines. We've never had a cat that could talk, but have had cats that were truly as "wiseacre" as Salem.

Still, on the internet we can visit the Fermi Laboratory and learn about fundamental particles of matter, particle accelerators, methods for detecting sub-atomic particles, the relationships between astronomy and physics, etc. etc. etc. Our boy Jean-Guy can visit a website that's all about Spawn, and spend hours reading up on his favorite action hero.

No, there's no Sidebar this time. Authors who haven't done any source code don't have Sidebars. Shame on me! Next time I PROMISE. . .

1.82. The Art Of Assembly — Part 82. Click On Icon Again

By Bruce Harrison

It's been an exciting few weeks since we wrote Part 81 of the series. During that time, we bought a brand-new PC, of the "modern" variety, with Cyrix 233 processor, 32 Megabytes! 4.3 Gigabytes Hard Drive, 32X CD-ROM drive, etc. Came with Windows 98 and a whole bunch of other stuff pre-loaded on the Hard Drive.

With the new machine, we've gotten connected to America Online, got a new e-mail address, new and better capabilities for sending and receiving e-mail, and a whole new outlook on the world of the PC in its present evolution. Of course being the pessimist that I am, I'm quite certain this model will be rendered obsolete soon after the credit card charges are paid off, but for the present, it's wonderful!

Before we bought this machine, I had tried computers with the GUI concept twice before. Once on a MacIntosh at the office, and once on a Windows PC at the library. They mystified me. I was sure it would take me a long time to learn to use Windows 98. WRONG! It took me very little time to get used to the "click on icon" idea, and not much longer to become at ease with the new computer. Yes, there are things I don't like. It takes quite a while to boot up, but of course the TI is kind of a "spoiler" in that respect. The second thing I don't like is having to go through a "shutdown" procedure before turning it off. With the TI I can just power it down any time. With the Tandy 1000, I did have to park the hard drive, but I used a batch file for that, so it was just typing the four letters PARK **ENTER** to get ready for the OFF position. The new one CAN just be powered down, but it remembers that you did that awful thing, and on the next powerup it chastises you for that evil act and checks the hard drive for "damage". That makes boot-up take longer, so the machine gets even with me for the time I saved by just turning it off.

Don't get me wrong, though, I'm starting to really love the new machine. Among other things, we could visit the MUG98 web site and see all those very nice pictures. We could download a selected subset of them and enjoy them any time. With a little help from AOL, we were able to create two web site of our very own:

```
http://members.aol.com/rottencat1/homepage.html
http://members.aol.com/flatpickin/loryspage.html
```

The first of these is called The Assembly Guru, named for your author. Among other things it has links to pages full of various kinds of software product descriptions, a "bio" of your author, and a page of recipes for various very unhealthy dishes. Thanks to our friend Dr. Charles Good, there's a picture of your author available at that website, as a "click" off of the Brief Bio page. The picture was taken by Charlie with his digital camera at the MUG 98 in Lima, put on his MUG98 page, downloaded to the hard drive on our new PC, then uploaded to our space on AOL to be available with our web site. The subject of the picture is just as awful looking after all those down and up loadings, but the picture itself is terrific.

The second of those web sites is by my partner, Dolores P. Werths. This deals mostly with her exploits as a musician in various venues, including Renaissance Faires (& Festivals) and numerous other gigs. The irony of all of the above is that I'm writing this column on the trusty old TI, using Funnelweb, while our boy Marcel uses the new PC to play "Atomic Bomberman" in the other room. I too, though, have been known to play more than a few games of solitaire on the new machine, but I try not to do that when Marcel's home, because he always kibitzes over my shoulder, seeing moves I've missed. I still lose the game when he does that, it just takes longer to get really stuck.

So what does all this have to do with Assembly Language on the TI? Not much except that if my "fans" have noticed a slight decline in my production of new programs (all the way to zero) they can blame the new PC and the really excellent game of solitaire that was "bundled" with it. By The Way, nobody bothers to "bundle" any programming languages with the PCs now. Back when we bought our "stone age" Tandy, GW Basic came with it as a matter of course, and Assemblers were easy to find. Not now. I suppose if one went to the right store, one could buy some kind of C++++++ compiler for about $150.00, but there is no more free lunch for programming. (Yes, I know that Peter Drucker said there's no such thing as a free lunch, and once again he's been proven correct.) (Peter Drucker is ALWAYS correct!)

It's been a couple of weeks since the part above was written, during which again no TI source code was written except to make a minor correction in a program we produced some time back. We know we promised some kind of Sidebar with source code this time, but have broken that promise. The power of the new PC is just too much of a temptation. Finally we can look at all those "Frames" websites and see what's there. Sometimes they're a bit of a let-down, particularly those that produce what I call "lists of lists". In those, each choice made from a list produces yet another list, and maybe five or six such levels go by without producing any "meat" on the screen. On my own web site, I've tried very hard to avoid that pitfall. There's some "meat" on every page, but there are also "Details" links so that those interested in a particular subject can dig deeper, but those just passing through won't have to suffer all the details. My partner's page is structured that way, too.

The other thing we've done is to keep checking out the web site on that old Tandy 1000 SX, so that we'll know that it looks okay for those without the modern PC, and even those using TERM80 on a TI will be able to extract all the information from the site. (All but the picture of yours truly, but that's no great loss anyway. On our Tandy, we get offered the choice of downloading that picture, but since we already have it on the new computer there's really no need. Maybe Lew King could download it onto his SCSI, but then it's a .jpg file, so what exactly he'd be able to do with it from there is questionable. The JPG format is now the standard for photographs on the PC. Our daughter Karen works at a large photo-finishing plant, and she can have our regular photos put on 3 1/2 inch (HD) disks in JPG format. They render pretty nicely on our new PC, and can of course be uploaded to the web sites for others to enjoy, but there's not much hope of having them rendered at all on the TI. Maybe some ambitious Geneve programmer will come up with a "JPG Show" program for that machine. There, now a challenge has been issued!

1.83. The Art Of Assembly — Part 83. The End Of An Era

By Bruce Harrison

This is IT! The END! No More! If you're hurting for something to read, there are 82 previous parts to keep you entertained. By the time this is published, it will be old news, because the folks who attended MUG 99 spread the word around that Harrison has "thrown in the towel" for good. In today's column, we'll try to explain why.

1.83.1. Third Party Hardware

Those three words summarize the reason this particular Assembly programmer won't be writing any more programs. As more and more new things get developed to "enhance" the TI-99/4A computer, it gets more and more difficult to test any software and have reasonable confidence that it will work on the systems owned by the clients. At this point I consider it quite impossible. That's because there are simply too many kinds of hardware setups other than the one you're writing and testing on. No matter what you're trying to do, chances are that somebody out there has a "combination of ingredients" in his system that will defeat your program.

1.83.2. My First Experience

Many years ago, while I was still perfecting the "Assembly Music" programs that made Harrison Software a "household name" in the TI community, I made the trip to a TI Faire in Ottawa. At the time, I had absolutely no third party hardware in my own system. To save on lugging the system (and explaining it to customs officials) I decided to borrow a system from the Ottawa UG to do my table demos.

Looked fine until it was turned on, and then there was no familiar color bar display, but some kind of menu instead. This was my first time seeing a system with the Horizon Ramdisk in use. Since my "Assembly Music" programs ran in Extended Basic, I chose that, and discovered one way that such things can wreck one's software. In the startup of my XB code I wanted to have a copyright notice come up only on the first showing of the program's main menu. To do that I checked to see if there were no definition for character 143, which is normally undefined when one enters XB. Unfortunately, the Ramdisk's own menu defines that character for its own purposes, so my program skipped its copyright notice, and also skipped the steps that loaded the "defs" file from my disk. As a result, my programs would simply lock up the computer.

The system's owner looked at what was happening, turned off the power and then removed his Ramdisk card. After that, everything worked fine with my disks on his system, users were happy, etc. But this was a harbinger of doom! I didn't see it at the time, but the end was already in sight. I was able later to re-work the main menu part of my Assembly music products so they could co-exist with Horizon Ramdisk equipped systems.

1.83.3. The Big "G"

At about that same time, Lou Phillips was starting to make and sell the Geneve 9640 in quantity. Enthusiasts sprang up and put cash in Lou's pockets for the new "improved" capability. In theory, the Geneve (in its GPL Mode) would run any existing TI-99/4A software, so the person buying it would not lose the ability to use software he already owned, and would be able to use new software still being written for the original machine. That theory never quite worked out.

My own first experience with the Geneve came when I sold an Assembly Music disk to a guy named Don West. Don had both a TI and a Geneve at his house, so he was able to do some comparing. As it turned out, my disk actually worked on both machines! The only problem was that the speed of playing was very different. On the Geneve, the music played about twice as fast as on the TI. Not good.

Eventually, we were able to develop a "self calibration" method for the music's pacing, so that a few of our Assembly Music disks could actually be used on either the TI or the Geneve, and would play at the same pace on either one. That did not last long, however, as another improvement we introduced for the TI, which involved using a DSRLNK in our assembly code, made the newer products completely incompatible with Geneve, and so they remained.

The claimed compatibility of the GPL Mode with TI software was always more than a little "iffy", and remains so to this day. One never knows what little trick that works perfectly on the TI will cause total lockup on the Geneve. For those who don't own the Geneve, and therefore can't test anything on it, this is particularly annoying. There's always a "Geneve Surprise" waiting to happen every time you send out new TI Software. (See "The Final Straw", below)

1.83.4. Other Fun Hardware

Of course not only Lou Phillips' Myarc made third party hardware. There are also Horizon (Ramdisks and P-Gram) and CorComp (various things) to consider. One of my all-time favorites involved the CorComp Mini System, in which the 32K etc. plugged into the side of the console instead of having to use a P-Box. Sent off some package or other to a gentleman in Maine. Soon heard those dreaded words "didn't work". There was, as I recall, nothing exotic being done by the software, no tricks outside of the usual Assembly stuff, but the program got just so far on his CorComp mini system and then froze up so that only the on-off switch had any effect. Never did find out why that happened, but had to refund his money.

The Horizon Ramdisk usually causes no trouble, except in cases like trying to read or write sectors, and then only if the CRU for the Ramdisk is set above >1100. It can, however, cause other problems, as a recent (Feb 1999) spate of messages on the TI List Server showed. Seems the original suspected culprit for a problem with TI-Artist was the AMS Card. Turned out instead to be the Horizon Ramdisk. At the 1998 Chicago Faire, we were given a disk from Europe that included a new Disk Manager program called DM2K. This was intended to work with floppies, hard drives, and SCSI drives. Back at home, I tried it with my system's floppy drives and Ramdisk. It would catalog any of them just fine, but would not copy files to or from the Ramdisk. As it happens, my Ramdisk card is at CRU >1200. Another program for the "doesn't work with" file.

We have not even mentioned such things of beauty as the Myarc HFDC, which can cause a simple boot-tracking routine to lock the system up, the SCSI card, on which boot tracking does not work, and the infamous Rocky Mountain Memory card (32K) which could not properly run our Assembly Music products. And now there are still more surprises just waiting for us, in that whole slew of cards that Michael Becker and his friends are building. So far, my only experience with those had to do with the AMS Emulation that's included in the three-card package. I had to modify my AMS packages so they would work on both the SW99ers SAMS card and the German emulation. A simple enough fix was devised with help from Michael, and tested okay on both AMS systems, but as a result all of those packages are now TOTALLY incompatible with Geneve. Is that a surprise?

1.83.5. The Helpless Community

There was a time, not too long ago, when most of the people using the TI-99/4A computer were conversant at the very least with TI Extended Basic. If they encountered a problem with a program in that language, they would at the very least have a look at the listed program to see whether it could easily be fixed. Not any more! Recently I was sent an entire package that was written in Extended Basic to help with a problem that took ten minutes or so to find in the XB code, and just a few seconds to correct. Testing the fix took a lot longer, because it involved printing several pages of graphics through XB (painfully slow).

The message is, TI users don't try to help themselves any more. They're becoming just as helpless as the PC owning "Windows Wizards" who never understand anything beyond the point and click operation. Of course in the case of the PC, keeping users from knowing what's going on is the whole purpose of Windows, but it used to be different for TI users. Just a few years back, any TI owner faced with a problem in an XB program would, at the very least, LIST the program and examine it for anything obvious that might explain the problem. In the case mentioned above, the problem had to do with what happened in the year 2000, and right there in the XB code was an IF-THEN with the expression IF (Y=2000). Really obvious if one just takes a look! Is it too much to expect that a user would be able to figure out that Y might mean Year, and that an exception being made when Y=2000 might have something to do with the observed problem?

1.83.6. The Final Straw

Over all the years that I've been writing Assembly programs, never had I written one that used the TI Speech Synthesizer Module. I'd always considered that just a bit too "Toys R Us" for my liking. I changed my mind about that just enough to make a program for playing Bingo on the TI that would use the Speech module (if available) to say the letter-number combinations as they came up. Worked just beautifully on the TI-99/4A here at home, and just beautifully on the system provided for my lecture at Chicago (also a TI-99/4A). I was happy with the results until I got my copy of the Jan-Feb 1999 *MICROpendium*. There in Charlie Good's MICROreviews column were the "final straw" words: "Unfortunately the speech of TI Bingo doesn't work on my Geneve. I have a Rave speech card in my Geneve. . ."

There it was, a double third-party whammy! Not only is the computer in question a Geneve, but the speech device is also a third-party item. Finished! No More! My Bingo has been discontinued, removed from the web site, no longer offered for sale, etc. No more software for the TI-99/4A, period. In Navy terminology, we have gone to the engine order telegraph and rung up Finished With Engines.