CHAMP

Assembler/Monitor package for the MSX range of computers

452, Stoney Stanton Road,
Coventry,
CV6 5DG.

Telephone: (0203) 667556

CHAMP

Assembler/Monitor package for the MSX range of computers.

To Load

(1) Reset your computer

(2) Type BLOAD " PSS ", R

(3) Ensure the tape is fully rewound

(4) Press play on your tape player

(5) Press 'RETURN'

(6) The program will automatically execute once loaded.

The Package

Champ will auto-run when loading is complete, so, having isssued the LOAD command , you need do nothing until the screen clears and displays the copyright message. Stop the tape, remove it, and replace it with a blank data tape if you intend to save program files from Champ.

⟨Assemble⟩ mode
is used after you have typed in an Assembly language program, in order to assemble it into machine code.

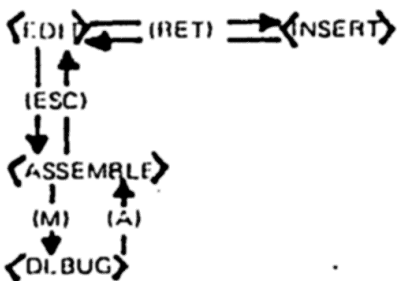
⟨Insert⟩ mode
is what you use to type in an Assembly language program.

⟨Edit⟩ mode
enables you to modify an existing Assembly language program.

⟨Debug⟩ mode
allows the inspection or modification of the contents of the memory, or the execution of a machine code program.

Both ⟨ASSEMBLE⟩ and ⟨DEBUG⟩ modes are command modes. In these modes various keys represent commands which make something happen to your program or to memory. On the other hand, ⟨INSERT⟩ and ⟨EDIT⟩ are text modes; with these you can move program text around on the screen, and add to, or modify it

You can change from one mode to another as shown here:



You now have CHAMP successfully loaded into your computer, you will see the ⟨ASSEMBLE⟩ prompt. At this point, the computer is waiting for you to type in an assembly language program, but don't do anything yet.
see figure 1.

```
;
;    CHAMP EXAMPLE PROGRAM


;
     ORG $C000
;
;    VARIABLES
;
I    DB    0
J    DB    0
;
;    PROGRAM
;
;    10 FOR I = 100 TO 1 STEP-1
               LD   A, $64
NEXTI          LD   (I), A
;    20 FOR J = 255 TO 1 STEP-1
               LD   A, $FF
NEXTJ          LD   (J), A
;    30 NEXT J
               LD   A, (J)
               DEC  A
               JR   NZ, NEXT J
;    40 NEXT I
               LD   A, (I)
               DEC  A
               JR   NZ, NEXTI
;    50 RETURN:  REM TO BASIC OR
     CHAMP
               RET
```

Figure 1

The first thing to notice is that there are a lot of semi colons (;) about. These may look strange, but are very simply the equivalent of the BASIC REM. In machine code programming they are known as COMMENTS, and they are extremely important if you want to understand something you may have written weeks ago. You can put anything you like inside a comment, which must begin on a new line. I have put the equivalent BASIC program lines in comments so that you can see how the machine code instructions can be made to operate in the same way as BASIC.

Any line which does not start with a ; is an assembly language statement. The first one is ORG $C000. This is not a machine code instruction but is short for ORIGIN, it tells the assembler where to put your program when you tell it to turn your assembly language into machine code. In BASIC, you did't need to worry about where your program was because the interpreter looked after all that for you. Now, you have the whole of the computer under your control, and that includes where you want your program to go.

You also tell the assembler where to put your variables. Once again, BASIC used to do all this for you, but BASIC isn't as good at using the full power of the computer as a human being is, so you have to do it yourself. For large machine code programs it is usually a good idea to put all your variables together in one block, but for small programs it's okay to put them next to the program for clarity. The next few assembly language statements in our example tell the assembler that you want to use two variables, I and J and that they will be found right at the beginning of the program straight after the ORIGIN address. Because we are not going to use numbers larger than 255, we only need one byte for each of the two variables, so the DB (Define Byte) statement is used. If we had wanted to use larger numbers we could have used the DW (Define Word) command to reserve 2 bytes for each variable. BASIC would have automatically used up 5 bytes for each of I and J. The storage define commands very simply tell the assembler not to use an area of memory because you're going to store some variables there. They also tell the assembler how big each area is and what the areas are going to be called. (In this case I and J).

The storage commands, the ORG command and the COMMENT are all called Pseudo-ops (ops short for operations) because the assembler doesn't generate any machine code from them, they are just there for your convenience.

All the other assembler language statements will be translated by the assembler into executable machine code instructions. That is, instruction codes which will cause the microprocessor to do something for you. Among these are Load a register, test a flag and Jump to a new address. These make up the rest of the example program listing, their operation is covered in several good text books on the subject. (See Bibliography).

If you're not too clear on any of the three types of instruction, Pseudo-ops, Comment, Org or Storage, then please reread this section, it's not at all difficult once you get the hang of it, but do take it at your own pace.

Now that we have covered the different types of statement understood by the assembler you can enter the example program.

Hands On!

The < ASSEMBLE > prompt is telling you that the assembler is waiting for you to do something. We wan't to enter our first program so tell CHAMP you want to EDIT by pressing{ESC}. The prompt changes to < EDIT > and shows a flashing underscore at the cursor position. The first thing you always want to put in is a comment to say what the program does, so type a semi colon together with the title of the program and any other information you think might be useful. Press Enter before you reach the right hand side of the screen, the line you just typed will move up one place and the cursor will be flashing at the beginning of the new line. The prompt will now show < INSERT > , this is because your are now inserting new information into the assembler. If you make a mistake before pressing ENTER, then use the cursor keys (which operate as normal) to correct your mistake, just type over any mispelt words. If you pressed ENTER before noticing your error, don't worry, you can correct it in a minute.

As you can see, you can also use blank comment lines to space your program listing out to make it more readable.

When you have finished typing the top comment, and the cursor is at the beginning of a new line try typing ENTER once more, you will find that you go back to < EDIT > mode. In < EDIT > mode, you can use the cursor keys to scroll the listing up and down and move the cursor through any line you may want to change. Correct any mistakes you may have (but don't press ENTER) and move the cursor back down to the bottom of the text. Now press ENTER again and you should once more have the cursor on a blank line with text above it and nothing below. If not, use ENTER to toggle between < EDIT > and < INSERT > , and use the cursor keys to get you to correct position at the bottom of the text.

Now type a space without a semi colon, the cursor will skip to the second field (or column), because CHAMP knows that if you type a space, you don't want anything in the first field. Now type in ORG followed by a space. Once again, when CHAMP gets the space, it knows to skip to the next field, and you can now complete the ORG instruction by typing $C000 followed by ENTER. All instructions except comments are typed into the assembler in this way; when you've finished using a particular field (or don't want to use it at all) then use the space bar to move to the next or ENTER to complete the line.

If you want to type a line containing a label, then it is exactly the same. Start in the LABEL field (the leftmost one), type your label followed by a space, then carry on with instruction and operand fields.

Some typing errors will be recognised by CHAMP when you press ENTER and will cause an error message to be displayed. Possible errors at this point are LABEL, INSTRUCTION or OPERAND errors. These correspond respectively to the three fields in the assembler display, so if you get an error it is most likely that your mistake will be in the field referred to in the

error message. Use the cursor keys to go back and correct the mistake when you've found it.

When you've typed the listing in, press ENTER one last time to return to <EDIT> mode, then use the cursor keys to check over the listing to ensure that it looks like the example. When you are satisfied, press ESC to return to <ASSEMBLE> mode and SAVE your text using the S command (this can be loaded back at any time using the L command). This is a good habit, it is much easier to lose what you are doing when using machine code since you can't use CTRL-C or STOP to stop a runaway program.

Having SAVEd your listing, you can now Assemble it. Type A (for Assemble), and CHAMP will display the ASSEMBLE=> prompt. Type in a 3, this is your assembly option, it tells the assembler what sort of listing you want and is explained more fully elsewhere in the manual. Press ENTER at this point and assembly should commence.

If all is well, CHAMP will have printed a version of your listing with some extra numbers on the left hand side. The left most column of numbers show the addresses to which each instruction has been assembled. They may look a little odd because they are expressed in hexadecimal notation. If your are not sure of this, then refer to your Z80 book for a full explanation before proceeding.

Notice that the addresses do not increase after comment lines, this is of course because they do not produce any machine code.

This is also reflected in the second two columns of numbers, these contain the actual machine code values loaded into memory. You can see that comment lines once again produce no machine code values, as you would expect.
After the listing you will also see a table of all the the lables you used, this is called the Symbol Table and CHAMP produces it for your convienience when using it to produce large amounts of machine code. It enables you to find the parts of the program you want quickly.

You might like to note that addresses of variables and jump labels are held in the symbol table inthe same way, this is because the microprocessor holds them internally in a similar way in its registers.
Having successfully assembled your example program, enter the monitor part of CHAMP by pressing M then ENTER. The screen should now display the < DEBUG > prompt.

Now we want to look at the program you assembled into memory. The start address of your program is not $C000, but $C002, because of the two variables you reserved. So type D (for dissassemble) followed by C002 (you don't need a $ sign in < DEBUG > mode).

```
C002 3E64     LD    A,$64
C004 3200C0   LD    ($C000),A
C007 3EFF     LD    A,$FF
C009 3201C0   LD    ($C001),A
C00C 3A01C0   LD    A,($C001)
C00F 3D       DEC   A
C010 20F7     JR    NZ,$C009
C012 3A00C0   LD    A,($C000)
C015 3D       DEC   A
C016 20EC     JR    NZ,$C004
C018 C9       RET
```

FIGURE 2

When you press ENTER, you will see a listing similar to Figure 2, without comments and Pseudo-ops. You will remember that these produce no machine code.

If this listing doesn't look like the machine code you typed in to the assembler then return to < ASSEMBLE > mode by typing A,ENTER and REASSEMBLE the program, checking that you use the correct option and that the assembler finishes by saying that it found no errors.

And now for the moment of truth, if the listing printed by the dissassemble command looks correct, you can execute it by typing GC002 ENTER. If all is well, the <DEBUG > prompt will return almost immediately, telling you that your program has completed execution.

You might like to type in the BASIC program in order to appreciate the magnitude of difference in speed between the two programs. You could even modify the programs to put an extra loop around the outside of the two loops already present and use a stop-watch to calculate exactly how much faster machine code is by using the formula:-

BASIC TIME (SECONDS)

MACHINE CODE TIME (SECONDS)

Be prepared to wait a long time for the BASIC!

When entering a new program, emember,
LABELS must start with a letter, and must not be more than six alphanumeric characters long.
INSTRUCTION MNEMONICS must be standard Z80: two, three, or four letters long.
OPERANDS must follow standard Z80 formats. They can contain arithmetic expressions comprising symbols or hex constants and a '+' or '-' operator, and can fill,

but not exceed, the entire operand field. COMMENTS must start on a new line with ';'. They can fill, but not exceed, the entire line, and are not subject to syntax or format checking.
When you have successfully typed in the program, enter < EDIT > mode. In this mode you can change the text on the Edit Line, and you can move the entire text file up and down on the screen using the following keys:

KEY	EFFECT
↑	Moves the Edit Line up one line.
↓	Moves the Edit Line down one line.
(CTRL)+(T)	Moves to the top of the text
(CTRL)+(B)	Moves to the bottom of the text.
(CTRL)+(U)	Moves text up one screen page.
(CTRL)+(D)	Moves text down one screen page.
(CTRL)+(Z)	Deletes the contents of the Edit Line.

These keys without (CTRL) have the same effects in < ASSEMBLE > mode, but you cannot delete or otherwise modify your text in that mode.

If the results are successful, then you might want to SAVE the machine code (called the Object Code to distinguish it from the Assembly language Source Code) to tape, using the 'W' command in < DEBUG > . Having done that, you might like to try altering some of the object code in memory using the '@' command, also in < DEBUG > . Once you've started to understand roughly what's going on in Champ, you should simply play around with any and every command or option that meets your eye— you can't damage anything, and it's really the only way to learn.

< EDIT > MODE COMMANDS

In < EDIT > mode, source text is displayed with the cursor on the Edit Line, and < EDIT > on the Command Line. Text on the Edit Line can be overwritten or deleted (using (DEL) or (SP). (RET) causes the Edit Line contents to be checked for syntax and format. An error message will appear if the line is faulty, and the text will remain on the Edit Line. If the line is acceptable, it will be entered into the source text, and mode will change from < EDIT > to < INSERT > . (RET) toggles these two modes, while (ESC) toggles < EDIT > and < ASSEMBLE > modes.

The following keys can be used to move the source text on the screen, assuming the text on the Edit Line is correct. If a line is edited & text is valid, then any of the following keys has the effect of entering the new line into the source text without changing the mode.

N.B. The text movement keys have the same effects when used in < ASSEMBLE > mode, but they then do not require (CTRL) to be pressed. Thus (U) in < ASSEMBLE > mode moves the screen text up one page.

KEY	EFFECT
↑	Moves one line up the text.
↓	Moves one line down the text.
(CTRL)+(U)	Moves the screen text up one page.
(CTRL)+(D)	Moves the screen text down one page.
(CTRL)+(T)	Moves to the top of source text.
(CTRL)+(B)	Moves to the bottom of source text.
(CTRL)+(Z)	Deletes the Edit Line contents.
(ESC)	Enters < ASEMBLE > mode.
(RET)	Enters < INSERT >

⟨INSERT⟩ MODE COMMANDS

It is this mode that you actually type your Assembly language program into the Assembler. The Command Line shows ⟨INSERT⟩ , and a flashing cursor appears on the Edit Line. The Edit Line (and the whole screen) is divided into three coloured columns, corresponding to the Label Instruction, and Operand Fields of an Assembly language program.

Label Field

A label is any alphanumeric string of up to six characters. There must be a letter in the first position of the Field. A label does not require a colon (or any other character) as delimiter.

Instruction Field

Instructions are Assembly language mnemonics as in Zilog Z80 specifications. They may be two, three or four letters long, starting in the first position of the Field.

Operand Field

Operands may be hex constants (which must be preceded by $), labels, symbols, or expressions comprising two operands seperated by + or −. Decimal, octal, and binary constants are not permitted. Operand formats for the various addressing modes are as specified by Zilog.

Text entry in ⟨ INSERT ⟩ is subject to Field Formating. This means it is impossible for you to type a seven-character label, or a five character instruction. Typing an extra character, or hitting (SPACE), causes the cursor to skip to the first position of the next field.

The (CRSRR), (CRSRL), and (DEL) keys act as normal in ⟨ INSERT ⟩ mode – subject to Field Formating – but the delete keys acts on the cursor character rather than on the character to the left of the cursor.

When you hit (RET) in ⟨ INSERT ⟩ mode, the contents of the Edit Line are checked for syntax and format; if an error is found, then a message appears on the Error Line. If no error is found, then the contents of the Edit Line enter the source text, and the Edit Line is cleared for the entry of a new line. Hitting (RET) when the Edit Line is blank toggles between ⟨ EDIT ⟩ mode and ⟨ INSERT ⟩ mode.

⟨ DEBUG ⟩ MODE COMMANDS

This mode combines the following functions:

MEMORY MONITOR – allows you to inspect and alter the contents of memory.

HEX DISSASSEMBLER – allows you to interpret the contents of memory as machine code to be converted back into Assembly language.

DEBUGGER allows you to execute machine code programs in an error-trapping environment.

⟨ DEBUG ⟩ is a command mode, but the Command Line/EditLine/Field Format display of the other modes is not used: the screen is a blank page showing only the prompt and a cursor. In this mode all constants without the '$' prefix, although the 'H' command supports decimal constants.

ABBREVIATIONS

addr	any hex address
saddr	start address of a block of memory.
faddr	finish address of a block of memory (=1+ address of last byte of block).
daddr	destination address in hex.
hx	a hex value (hx ⟨ = FF)
regname	CPU register name (see below).
expr	any arithmetic expression in one or two operands; operands may be decimal constants, '$' – prefixed hex constants, or legal symbols; operators are '+' or '−'.
bystr	a string of hex byte values seperated by spaces.
chstr	a string of characters (exactly as it appears, no seperators).

CPU REGISTER NAME ABBREVIATIONS

A= Accumulator; F = Flag/Status Register; H,L,B,C,D,E, = Registers; SP = Stack Pointer IX, IY = IX, IY registers.

COMMAND	EFFECT
@ addr or 0 addr	Memory from the given address onwards displayed one byte at a time, in hex and ASCII equivalent. Hit (RET) to advance to next byte, hit (ESC) to return to command level, or type a hex constant to replace the existing content of the byte.
A	Return to ⟨ ASSEMBLE ⟩ mode.
D addr f addr	Memory from the given address onwards is displayed in screen pages; hit any key to continue, or (ESC) to return to command level.
F saddr f addr hx	Every byte between saddr and faddr is filled with hx.

K	Print source and symbol table usage
M daddr saddr faddr	The block of memory between saddr and faddr is copied to the block starting daddr.
Q addr	Memory from addr onwards is disassembled; hit (RET) to contiue, and (ESC) to return to command level.
G addr	The code starting at addr is executed (returnable).
C addr	Execute from addr (non-returnable).
Bn=addr	A breakpoint number n, (between 1 and 8) is set at addr, to cause a break in execution of any program which accesses the contents of addr as an instruction; press (C) (RET) to continue from breakpoint.
En	Eliminates breakpoint n.
T	Displays the addresses of all the breakpoints.
R regname	Displays the contents of a CPU register and accepts a new value (similar to the function of '@' above).
J addr	Executes the code from addr onwards, one instruction at a time, giving a full register display. Hit (J) to continue, (ESC) to return to the command level.
H expr	Displays the decimal, hex, and binary value of expr.
S bystr	Searches the memory from $0000 onwards for every occurrence of bystr. The word 'searching' is displayed while the program is searching, and the address displayed when bystr is found. Hit (RET) to continue the search, or (ESC) to return to command level
Nchstr	As 'S' above.
W	Load, Save, and Verify machine code to tape, see BASIC panel.

INSTRUCTION FORMAT

Z80

Instruction		Addressing Mode
LD	A,B	Register (Direct)
LD	A,$ 9F	Immediate
LD	($ED46),A	Absolute (Direct)
LD	A,(HL)	Register (Indirect)
LD	A,(IY+d)	Indexed (Indirect)
CCF		Implied

ASSEMBLY LANGUAGE FORMATS

Psuedo-Op-Codes	Meaning
ORG addr	origin; assemble machine code in memory from addr onwards. The program line with ORG on it cannot take a label.
EQU	equate; set the symbol in the Label Field equal to the constant, symbol or expression in the Operand Field.
DB const/ chstr	define byte(s); load this location, and as many following as required with the value(s) of const or chrstr
DW const/ symb	define word; oad this location with the lo-byte, and the next location with the hi-byte of the operand
DS const/ symb	define storage; add the value of the operand to the location address of this instruction.

Abbreviations;

addr	a $-prefixed hex address
const	a $-prefixed hex constant; as an operand of DB, const must be a single-byte value. A string of constants such as (DB const(SP) const (SP) const . . . etc) is valid.
chstr	a string of characters enclosed in single quotes (e.g. 'AB3%9K10')
symb	any valid symbolic operand

LINKING MACHINE CODE AND BASIC

Once you're familiar with both Champ and Assembly language programming, you'll probably want to be able to call special purpose machine-code routines from BASIC programs, rather than write entire programs in machine code. The easiest way of doing this is:

1. Using Champ, develop the Assembly language routine util it works.

2. From ASSEMBLE mode, SAVE the Assembly language routine to tape for future reference.

3. Assemble the routine into memory, choosing an ORG address near the top of User RAM (see your computer User Manual for Memory Map and advice).

4. From DEBUG mode, SAVE the block of memory containing your machine code to tape.

5 Quit Champ

6 Write your BASIC program, starting with the instructions necessary to set the Top of User RAM pointers to an address safely below the ORG address of your routine. Follow those instructions in the program with a LOAD instruction that will load your machine code routine from tape to the location from which it was SAVEd (consult your User Manual).

7. Whenever you need to execute the machine code routine in the BASIC program use a USR instruction with your routine's address.

8. Save the BASIC program as usual.

CHAMP ERROR MESSAGES

Error messages appear on the Error Line in all modes except DEBUG, which prints 'ERROR' at the current cursor position.

Message	Meaning
Label Error	A syntax or format error in the Label Field.
Instruction Error	A syntax or format error in the Instruction Field.
Operand Error	A syntax or format error in the Operand Field.
Undefined Label	The Label or Symbol displayed on the Edit Line has not been assigned an address or a value.
Jump out of Range	The relative jump in the instruction on the Edit Line requires a displacement of more than 127 bytes forward or 128 bytes backward.
Overflow	Assembling the instruction on the Edit Line into memory would overwrite CHAMP itself, or some protected memory, or would be out of range.
Error	The operand of a DEBUG command contains illegal symbols or is too large a quantity, or is a bad address etc.

ASSEMBLE MODE COMMANDS

Find = string (RET)
Searches the Assembly language program from the start of the program for the first occurrence of the given string.
Next = string (RET)
Searches the Assembly language program for the next occurrence of the given string. The search begins from the end of the program line currently on the Edit Line.
Find = (RET) and
Next = (RET)
As above, but this searches for the string defined in last 'F' or 'N' command. While a search is preceeding, the message 'searching' appears on the Error Line. If the search is successful, the line containing the string being seached for appears on the Edit Line. If the search is unsuccessful, the last line of the program appears on the Edit Line.
Load = > Save = > Verify =>
These must all be followed by a filename; double quotes are not needed, but the filename must be legal for user's machine.
Print = > expression (RET)
This prints the hex value of the given expression on the Error Line, eg.
Print = > $F8-$C1 $37
symbols already defined in source text can be used in expressions; but only one Operator (+ or—) is allowed per expression.
Quit = > (Y)
This quits Champ and returns control to the BASIC system only if (Y) follows the prompt; any other response aborts the command.

(M)
Enter DEBUG mode. Return from there to ASSEMBLE mode by pressing (A) (RET).
(ESC)
Toggle EDIT and ASSEMBLE modes.
Assemble => (option number) (RET).
This assembles the source text in one of a variety of ways, depending upon which numerical option is chosen.

KEY	PROMPT	FUNCTION
(F)	Find = >	Find a string
(N)	Next = >	Find a string
(L)	Load = >	Load a source file.
(W)	Save. =>	Save a source file
(V)	Verify =>	Verify a source file
(P)	Print =>	Print value of expression
(Q)	Quit = >	Quit to BASIC
(M)		Enter DEBUG mode
(ESC)		Enter EDIT mode
(A)	Assemble =>	Assemble program.

ASSEMBLY OPTIONS

To tell the assembler what to do, you must choose from the table below, add the numbers together and type the resulting number in reply to the Assemble ⇒ prompt.

	Option No.
Syntax check only	0
Display full list on screen	+1
Load M/Code into memory	+2
Copy screen to printer	+4
Double line list	+10
Supress display of symbol table	+40

EG.– To assemble machine code into memory, with a double line listing to the printer you type 17 (RET).

Full list	1
M/C to memory	2
Screen to printer	4
Double line list	10
	17

Note These are hexadecimal numbers, so bear this in mind when adding up.

Assemble => 2 (RET)

causes the source text to be assembled with error-checking and the resulting machine code to be loaded into memory as directed by the ORG pseudo-op-code. The symbol table is displayed on the screen, but no assembly listing appears on the screen, and there is no output to the printer.

If an error is found during assembly, a message will appear on the Error Line, assembly will cease, and the screen will display the source text with the faulty instruction appearing on the Edit Line.

CHAMP

Continuation Sheet

Bibliography.

Book	Author	Publisher
Programming the Z-80	Zaks	SYBEX
Z-80 Assembly Language Programming	Leventhal	Osbourne/ McGraw Hill

USEFUL ADDRESSES

Source stored at: $ AAØØØ

Symbols stored at: $ EØØØ

Best ORG Address: $. CØØØ

To re-enter CHAMP type:– DEF USR & H84ØØ (For Normal Entry)

DEF USR & H84Ø3 (To clear source buffer)

To Execute CHAMP Type:– PRINT USR (Ø)