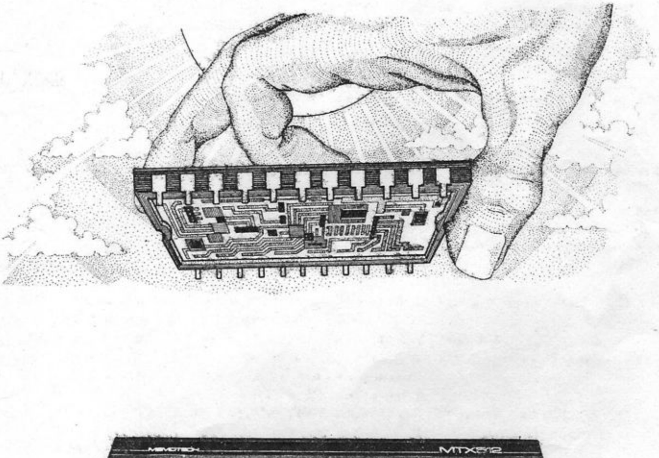
MEMOTECH MTX SERIES

SYNTAX SOFTWARE

EDASM 8 BIT MACRO/ASSEMBLER

FOR

MTX SERIES COMPUTERS



3. Bulcock Street. Burnley. BB10 1UH 0282-57427

SYNTAX
SOFTWARE

MTX - EDITOR/ASSEMBLER Version 1.00 (tape)

Thankyou for your purchase of the MTX-E/A package. The facilities included within this package makes this one of the most advanced Editor/Assembler packages currently available for any tape-based micro.

NOTE: Any keywords will be signified by enclosing them within '<' and '>' characters (e.g. <RET> signifies the key 'RET') in order to avoid confusion.

LOADING

1. Connect the EAR socket on your cassette recorder to the EAR socket on your MTX computer.

2. Make sure the cassette is fully rewound.

3. Type:

```
LOAD "ED/AS" <RET>
```

Set your cassette recorder to play mode and wait.

4. If you can hear the tones of the tape from your television speaker but the computer gives no indication of loading, then rewind the tape and try again with the recorder on a different volume setting.

NEW COPIES

Making back-up copies that are identical to the one supplied is very simple procedure.

1. Follow the section on LOADING to load the Editor/Assembler.
2. Save as though it was just another Basic program with any name that you prefer.

Making new copies after you have modified the Z80 macro library (on the reverse side of the tape provided), or replaced it completely, for example, with a 6502 macro library, and assembled it onto the Editor/Assembler (see assembler manual), is a slightly longer process.

1. Assemble the new macro library onto the Assembler using the assembler 'K' option.
2. Quit the editor using the 'Q' command.
3. Re-load an old copy of the editor/assembler but do not RUN it.
4. Follow exactly the section on RUNNING under the sub-heading "Running after a SYSTEM RESET" for entry back into the new Editor/Assembler.
5. Answer the "New/Old file ?" with a 'N' followed by <RET>.
6. Use the 'X' command answering the "New copy (Y/N) ?" question with a 'Y'.
7. Use the 'Q' command to quit the editor and save the new Editor/Assembler as you would a Basic program with any name you prefer.

MEMORY USAGE

The Editor/Assembler occupies the memory from 0000H to 4000H, which is situated under the system ROMs. Source files begin at 4010H, so even if the system has to be RESET the file will remain intact and can be recovered by following the section on RUNNING.

Although this means that no Basic programs can be entered while there is a source file present, there is just room for one line that can be used to execute the generated code (e.g. code at A000H can be executed by entering the following Basic line.

```
10 RAND USR (40960)
```

and by typing RUN to execute the code

RUNNING

(a) Running from load up

Type:

```
RUN <RET> - Inintializes the program for use
USER <RET> - Enter EDITOR
```

Now follow the section on INITIALIZATION.

(b) Running from monitor

NOTE: This will not work after a SYSTEM RESET. If you are not sure that a system reset has occured then try this entry method and if you get a 'Mismatch' error then follow method (c).

Type:

```
USER <RET>
```

Now read the section on INITIALIZATION.

(c) Running after a SYSTEM RESET

Depending on what caused the system to reset you may find this method does not work. In that case you must re-load the Editor/Assembler from tape.

1. Enter panel by typing: PANEL <RET>

2. Set address by typing: DF000 <RET>

3. Now enter the following code:

```
F3 <RET>
3E <RET>
80 <RET>
D3 <RET>
00 <RET>
CD <RET>
4E <RET>
01 <RET> and now press '.' to leave command.
```

Type: 'B' & 'Y' to exit from panel.

4. Type: RAND USR (61440) <RET>

INITIALIZATION - warm or cold start ?

At this stage you should be faced with a clear screen (yellow) except for the question "New/Old file ?".

```
ANSWER 'N' - (Creates an empty source file)
Used:      - after loading Editor/Assembler.
           - when file has been corrupted.

ANSWER 'O' - (Enter existing source file)
Used:      - when re-entering editor.
```

Press <RET> to enter EDITOR.

INSIDE THE EDITOR

The display is divided into just two areas. The top 22 lines are a window on your source file. By use of the cursor keys you have the ability to move the cursor to any column and any line of your source file.

The very bottom line has two uses:

1. Status

The status line is divided into three areas. The first displays the current input mode, which is either CHANGE or INSERT (see 'INS' command). The second shows the coloumn number of the cursor, which has a range from 001 to 128. The third section is used for displaying error messages or other reports connected with commands in use. For speed considerations, the status line is only updated when no keys are being pressed.

2. Command information entry

When a command is activated, that requires input from the user of some kind, the bottom line will be cleared and used for this purpose. Commands will always prompt you for for this information.

INPUTING TEXT

You are now in the position to enter text into your source file. Initially the file is made up of just one empty line and up/down cursor keys therefore have no effect. By pressing <RET> you will immediately create a new empty line and the cursor will be set to start of it. To demonstrate text inputing, enter the following.

```
<TAB> LD HL,1000H <TAB>; Comments can go here <RET>
<TAB> LD DE,2000H <RET>
; Comments can also go here <RET>
```

There are other keys that will make the entering of text easier, such as DEL, F1 and F5 etc. For details of these commands and more read through the following section.

COMMANDS IN DETAIL

All letter commands are activated by holding down the CTRL key at the same time as pressing one of the letter keys. Other commands are used as stated.

A - Assemble file
Enter assembly options (see assembler user guide) followed by <RET>. You will now be prompted to enter the assembler workspace address in hexidecimal. When getting this, you must make sure it is within free memory. By pressing <RET> only, the workspace will automatically be placed after the current source file.

B - Bottom line of file
Set the cursor to the very last line of the source file and update screen.

C - Copy text block

To copy a block of text, simply place a signal string at the start of the line immediately before the block (the string must be unique to avoid confusion), and a delimeter code '-' (must always be a minus sign) at the start of the line immediately after the block.
eg., '-**1' before, and '-' after.
Set the cursor to where you want the block to be copied to (the cursor **must not** be positioned inside the delimetered block). Now activate command 'C' which will prompt you for the signal string (eg., '-**1'). Enter this and then press <RET>. The signal string will be located and the text will be copied to the current cursor position.

If you now wish to delete the first block, use command 'Z' followed by command 'D'.

The copy can be aborted by holding down the BRK key.

D - Delete text block

You are prompted for whether you want to delete up to a delimeter code, positioned at the very start of a line, or right up to the end of the file. Notice that by pressing <RET> only, it defaults to delimeter mode. The delimeter must always be a minus sign.

F - Find text string

Enter the string you wish to be found and press <RET>. Remember that the cursor position determines the end of the string. You are now prompted to specify where the string should be. Specifying 'S' will only look for the string at the start of lines and is therefore useful for finding labels. Specifying 'A' wil look for a string anywhere along a line. Use command 'Z' to find subsequent strings.

J - Join two source files

Similar to the 'L' command, but instead of looking over the current source file, wil load in at the end of the source file. Useful for moving routines from one program to another.

L - Load source file from tape

Enter the name of the source file you wish to load, or clear to spaces to load the next source file, and press <RET>. Now switch on tape and wait for file to load. BRK can be used to abort command, but only while data is being scanned from the tape recorder.

M - Load object file from tape

This command will load an object code file at a specified memory address.

N - Save object file to tape

Save a named object file to tape. You will prompted for start and end addresses.

P - Print text block

Print the file from the cursor line to either, the end of the file, or until a '-' delimeter is found at the start of a line. Printing will start when a 'Y' is entered in response to the prompt. BRK will abort printing at any time.

Q - Quit editor

Upon a 'Y' response to the prompt, control will be handed back to the system ROM.

R - Replace text string

Firstly you will be prompted to enter the string you wish to replace. After entering this and pressing <RET>, you now have to enter the string you wish to replace it by, followed by <RET>. Now enter the number of strings you wish to replace (from 1 to 99) or a '*' indicates all.After pressing <RET> you will now be prompted on whether you want the editor to stop it before it replaces a string, and allow you to decide, or whether you want it to replace them continuously.

As with the FIND and COPY commands the end of each string is determined by the cursor position.

S - Save source file to tape

Enter the name you wish the file to be saved with and press <RET>. Now enter the number of copies you wish it to save (1 to 99). This has the advantage of saving a number of copies, equally spaced, without you having to stand over it. Switch tape to record and press <RET>.

T - Top line of file

Set the cursor to the very first line of the source file and update screen.

V - Verify source file from tape

Verify the current source file with a named source file on tape. Enter name, press <RET> and switch tape to play.

W - Back tab

Moves the cursor backwards, clear as it goes, to the previous tab stop.

X - Copy ed/as for making new copy

This command is used for the sole purpose of creating new copies of the editor/assembler after re-assembling the inbuilt macro library (see section on NEW COPIES), and **should not** be used while there is a source file present (other than completely empty).

RET - Split line

This has the affect of splitting the cursor line at the cursor position.

EOL - Clear rest of line

Clear the line, from the current cursor position right up to the very end of the line (column 128).

BRK - Abort

Abort from **any** command while the cursor is flashing, and some while not flashing.

TAB - Tab forwards

Move forward, clearing as it goes, to the next tab stop (every eight columns).

DEL - Delete character

Move back one and clear character. When used on column one, the current cursor line is appended to the preceding line.

ARROW UP - Cursor up

Move cursor up one line. Hold key down for fast downward scrolling.

ARROW DOWN - Cursor down

Move cursor down one line. Hold key down for fast upward scrolling.

ARROW LEFT - Cursor left

Move cursor left one character. When used in column one, the cursor will move to the first non-space of the previous line and the screen will be updated to display this.

ARROW RIGHT - Cursor right

Move cursor right one character. The cursor will move to the proceding line after reaching column 128 of the present line.

HOME - Cursor home

Move the cursor to column one and update the screen display if necessary.

INS - Insert/change mode toggle

This is very similar to the normal use of this key, but it is worth just pointing out the difference between the two modes. Note that the status line shows you which mode you are currently using.

CHANGE MODE

1. Characters over-write other characters.
2. 'DEL' only clears characters.

INSERT MODE

1. Characters are inserted between characters.
2. 'DEL' clears characters and uses the 'F1' comand to move the text right of the cursor.

F1 - Text towards cursor

Move all the text right of the cursor (on current line) towards to the cursor.

F5 - Text away from cursor

Move all the text right of the cursor (on current line) away from the cursor.

F4 - Scroll file right

Scroll the file horizontally right eight characters, making the cursor move eight columns to the left.

F8 - Scroll file left

Scroll the file horizontally left eight characters, making the cursor move eight columns to the right.

SHIFT F4 - Delete cursor line

Delete the current cursor line. This is similar to the 'D' command but only deletes one line.

COMMANDS IN BRIEF

COMMAND	DESCRIPTION
A	Assemble the current source file.
B	Move cursor to the very bottom line of the source file.
C	Copy a specified text block to the current cursor position.
D	Delete a block of text, or delete from the cursor to end of source file.
F	Find a specified string.
J	Join a named tape file to the current source file.
L	Load a named source file from tape.
M	Load a named object file from tape.
N	Save a named object file to tape.
P	Send file, beginning at the cursor position, to the printer.
Q	Quit Editor/Assembler.
R	Replace one specified string of text with another.
S	Save the current source file to tape.
T	Set cursor to the very top line of the source file.
V	Verify a named source file on tape with the current file.
W	Move back, and clear, to the previous tab stop.
X	Transfer the editor/assembler to saveable memory.
Z	Find next occurrence of the string stored in the 'FIND BUFFER'.
RET	Split the current line at the cursor position.
EOL	Clear all of the line right of the cursor.

BRK	Abort command.
TAB	Move cursor forward to next tab stop.
DEL	Move cursor left one and clearposition.
UP	[Arrow] Move cursor up line.
DOWN	[Arrow] Move cursor down one line.
LEFT	[Arrow] Move left one character.
RIGHT	[Arrow] Move right one character.
HOME	Move cursor to column 001 and update screen.
INS	Change/Insert mode toggle.
F1	Move text, right of cursor, towards cursor.
F5	Move text, left of cursor, way from cursor.
F4	Scroll file horizontally right.
F8	Scroll file horizontally left.
SHIFT F4	Delete the current cursor line.

MACRO ASSEMBLER

OVERVIEW

There are many types of macro assembler to be found these days. Most of these offer a "hardwired" instruction set for a particular machine and allow macros to be defined in terms of these instructions. This is quite reasonable as assemblers are generally used on a single machine to generate code for that machine and a built in instruction set does decrease assembly time. So why depart from this tried and trusted recipe ?. Well, this assembler was conceived with two main goals in mind - (i) to be able to assemble code for CPU's other than the host, (ii) to be able to assemble code in times comparable to assemblers with hardwired instruction sets. In our opinion both these goals were realised.

USER GUIDE

This guide assumes a knowledge of normal assembly code.

Code format

Considerable freedom exists in the format of files acceptable to this assembler. Multi-statement lines are allowed and lines can be up to 128 characters long. But there are a few basic conditions that must be adhered to in order that the assembler knows what the code is supposed to mean, these are.

(a) Labels/variables may appear anywhere along a line, but if they do not start in column one then they must be immediately followed by a colon (unless they are part of/or form an expression).

(b) Multi-statement lines may be constructed by separating instructions with the backslash character "\".

(c) Files must be terminated by an END statement.

Basic primitives: DB & EQ

It was revealed earlier that the Z80 mnemonic set was not built into the assembler but were defined as macros in terms of more primitive instructions. We will now consider these primitives and define their syntax.

1) Define byte "DB".

This is the most used primitive in instruction set macro definitions (have a look at the Z80 macro library source). Its job is to generate one byte of code representing the value of the expression or list of expressions following it. These expressions must yield values between -256 and 255 inclusive. Here are some examples of the use and forms of the DB instruction and the code it generates. (This also shows off the expression handling capabilities of this assembler, so see the section on expressions for a detailed description of these facilities.)

```
0000 00          DB 0                  ;A comment
0001 1234        DB 12H,34H
0003 48454C4C
0007 4F205448
000B 45524521    DB "HELLO THERE!"
000F FE          DB 11111111B - 1
0011 2C          DB "H"-7o<<2
0012 00010203
0016 04050607    DB 0,1,2,3,4,5,6,7
001A FF00FF      DB 1<2 , 5>20 , "B"<=42H
   ^        ^    ^
Program     ^    ^
counter     ^    Code as entered in file
            ^
       Code generated
           by DB
        instructions
```

2) Label equate "EQU"

This is the same as in other assemblers and is used to set labels to particular values (which cannot be changed - more on this later). For example:

```
1234            LABEL1:   EQU 1234H
AAAA            A_NAME:   EQU 1010101010101010B
0CBC            $TT_TT$:  EQU 123*64-LABEL1
003F            COLZERO   EQU 77o                 ;Label in column zero
^
^

Result of
assignment
is put here
```

Notice that "$"'s and "_"'s are allowed as part of label names. Labels can be of any length but only the first eight characters are significant.

1) Expressions

As stated earlier, this assembler supports quite extensive arithmetic and logic capabilities. These facilities are no luxury and are used extensively later in macro definitions. Algebraic logic is supported with nestable parenthesises and four number bases. Here is a list of operators, their functions and precedence.

OPERATOR	PRECEDENCE	USE	FUNCTION
+	5	+A	Unary Plus
-	5	-A	Negate
!	5	!A	Logical complement
>>	4	A>>B	Rotate A right B bits
<<	4	A<<B	Rotate A left B bits
*	4	A*B	Multiply
/	4	A/B	Integer division
%	4	A%B	A modulo B
+	3	A+B	Addition
-	3	A-B	Subtraction
&	2	A&B	Logical AND
|	2	A|B	Logical OR
~	2	A~B	Logical EX-OR
=	1	A=B	Test equality
>	1	A>B	Greater than
>=	1	A>=B	Greater or equal
<	1	A<B	Less than
<=	1	A<=B	Less or equal

Precedence (the relative binding of an operator) is given as a number from 1 to 5 - where 5 indicates the highest binding power. Of course brackets may be used to alter the order of evaluation of an expression.

Labels and undefined values

In the preceding section on the EQU instruction it was shown that labels could be set to the value of an expression, and that expression could contain label references. In such situations the reference must be to a previously equated label otherwise a default value of zero will be used. ie.,

In this group of statements the first label wil be set to zero when it is meant to be set to 200H:

```
0000          LABEL1: EQU LABEL2*100H
0002          LABEL2: EQU 2
```

With each label there is stored a flag which indicates whether or not the label contains a totally correct value, this will be set true in the second equate but false in the first one. Certain statements that use expressions test this flag and will give an error if the expression contained an uninitialised value. This flag is propogated whenever a label/variable is set to an expression, even through macro arguments (see section on macros with arguments).

2) Macro definitions

For the uninitiated a macro is a single entity which represents a number of other entities. ie., A macro instruction will invoke a number of other instructions which can themselves be macros. To illustrate this further let us now define a macro.

```
        DEFMAC  ("MACRONAME")           ;This is how we
                                        ;define the macros name

                DB 77o                  ;Generate a byte
                                        ;containing 77 octal
                END.                    ;This is how we end
                                        ;a macro definition

                                        ;Now let's use this macro

0000 3F         MACRONAME               ;This is how we
        END                             ;End of source code
```

Notice that the macro definition did not generate any code but when the macro was invoked it generated a byte containing 77 octal (3F hexadecimal). So the upshot is that invoking a previously defined macro assembles the code present inside its macro definition. Macros can be thought of as procedures in some high level languages, such as PASCAL, which must be defined before their use. This type of macro is of limited use though, because it always generates the same code when invoked and in the folowing sections we will deal with how to define macros which generate different code when invoked in different ways, but it is worth looking at how we can define macros to be used as Z80 instructions with this type of macro.

```
                ;Definitions of the Z80 instructions NOP & HALT
                DEFMAC ("NOP")
                        DB 00H
                END.

                DEFMAC ("HALT")
                        DB 76H
                END.

0000 00         NOP
0001 76         HALT

                END
```

It is also possible to define macros which consist of more than one word as in this example.

```
                DEFMAC ("DO-THREE-NOPS")
                       DB 00,00,00
                END.

0000 000000            DO THREE NOPS
```

Although this is a silly example it illustrates that "-"'s are used to separate the words in multiple word macros. Notice that spaces canot be used inside the DEFMAC statement.

Macros with arguments

In order that macros can change the code they generate - we must have some method of passing information to macros when they are invoked. We do this by embedding arguments into the macro when we invoke it. There are two types of argument that we need to differentiate between.

1) Numbers, including expressions labels/variables and numeric constants.

2) Constants, including register names and some opcode mnemonics.

Most macros that have register names as arguments usually only use a subset of the total number of registers available and similarly macros that require numeric arguments require only numbers within certain bounds. So to cater for this requirement the assembler has the ability to define sets of registers (formally called constants) or subranges of numbers.

Constants

Constants are names which can take on a value (a bit like labels) but which are totally distinct from labels and variables in that they cannot be used in expressions. The value a constant takes depends on its use. Here is how we define constant names.

```
DEFCONST (BC,DE,HL,AF).
```

This definition does not generate any code, of course, and its purpose is to inform the the assembler which names are constant names thus preventing them from being used accidentally as labels.

Sets of constants

We are now in a position to use the constants we defined earlier. To do this we must collect them up into sets, we do this in the following way.

```
DEFSET REG16 = (BC,DE,HL).
```

We have now defined a set called REG16 containing the constants BC,DE and HL these constants now have a value if used in the context of REG16 - their values are assigned starting at zero and increment for each name in the set, so BC has the value 0, DE has the value 1 and HL has the value 2. It must be emphasised that constants are **not** treated in the same way as labels and cannot be used in expressions and further more, they only have a value when they can be found in a set and their value depends on which set they are used from. To illustrate the use of sets and show how arguments are used we will now consider how we would define a macro which will generate the code for all the Z80 8 bit register load instructions. ie., LD B,L etc.

Firstly we must define a set of constants which contains the name of all the registers this instruction is valid for.

```
DEFCONST (A,B,C,D,E,H,L,INVALID).
DEFSET   R8 = (B,C,D,E,H,L,INVALID,A).
```

Notice that the order in which constants are entered into the DEFCONST statement is not important but the order they are put in the DEFSET statement is, and also that a "junk" constant was needed to pad out the list. ie., we need L to have the value 5 and A to have the value 7 and no register has a value corresponding to 6 so a junk name is used.

Now let us define the macro.

```
            DEFMAC ("LD*,*",R8,R8)
                   DB 40H + £0*8 + £1
            END.
```

The asterisks in the macro name define where the arguments are to appear and the list of setnames after the macro name string define which set each argument belongs to. Each setname corresponds to each asterisk in the name string. The argument values are passed to the macro in a set of local variables that only the macro may access, these have the names £n, where n has a value in the range 0 to 255.

A look at the opcode for the instruction we are defining shows that its value is a combination of some preset data and the numbers of the registers is is to use. The DB instruction in the macro will faithfully generate the opcode for this instruction.

Now let us use this macro.

```
0000 78          LD A,B
0001 5F          LD E,A
0002 5A          LD    E  ,  D
```

Notice how spaces are ignored when invoking macros. Spaces are allowed anywhere except in the middle of a name (ie., HELLO is one name and HE LLO is two names) so spaces can be used to separated arguments from the mnemonic for example.

Number subranges

It is also desirable to define ranges of numbers and attach a name to then as we did with constant sets and so another function of the DEFSET statement is to do just that.

```
            DEFSET BYTE = 0 TO 255.
```

This has now defined a subrange of numbers called BYTE containing all the numbers from 0 to 255 inclusive. We can now use this set to define another Z80 macro - load immediate 8 bit register. ie., LD C,4

```
            DEFMAC (\"LD*,*",R8,BYTE)
                   DB 6 + £0*8
                   DB £1
            END.
```

Notice that only difference between this macro and the last is the type of the second argument, and, so that the assembler knows that the macro name string "LD*,*" has already been defined, the "\" is inserted before it in the definition. This must be done whenever a macro name string is repeated otherwise the "Double identifier" error message will be generated.

Here is how we can use this new macro.

```
0003 0620        LD B,00100000B
0005 3EFA        LD A,0FAH
0007 2645        LD H,"E"
```

It is worth pointing out that constants may be used in any number of different sets and the user must make sure that ambiguities do not arise because of this.

Certain Z80 instructions require an argument that has only one value to follow them (such as EX DE,HL where DE and HL are constant). Using the above methods of defining macros we would have to define two sets, one with HL in and the other with DE in. Which is a bit silly. So in order to obviate the need to do this constant names may be used in macro definitions and will mean a set with one element. To illustrate this let us define the macro EX DE,HL.

```
;Assuming that HL and DE have been defined as constants

             DEFMAC ("EX*,*",DE,HL)
                    DB OEBH
             END.

0000 EB             EX DE,HL        ;Use of this macro
```

By now you may wondering why we did not define a macro that used no arguments in the above case.

```
             DEFMAC ("EX-DE,HL")   ;This is illegal
                      DB OEBH
             END.
```

If the names DE and HL were not defined as constants this macro would be perfectly acceptable but as they have been defined as constants the assembler would search for a macro mask like this - "EX*,*" and so the "Undefined" error will occur.

With the facilities so far covered it should now be possible for the user to write his/her own macros which can simulate any mnemonic in the Z80 instruction set and much more.

Conditional assembly

The main conditional assembly statement supported by this assembler is the IF-END and the IF-ELSE-END statement combinations. These are used to enclose parts of the assembly code and enable/disable the assembler assembling it. This is mostly used for allowing one source listing to produce a number of different versions of a program depending on 'switches' set at the start of the source code. Here is a brief example of its use.

Here is a notional data storage routine - for disk systems it must contain code to write to the disks and for tape systems it must contain code to write to tape.

```
DISK:   EQU 0
TAPE:   EQU 1

SWITCH: EQU DISK
        ...Lots of source code...
 WRITE:
        IF SWITCH=TAPE
        ...Tape interface code...
        END              ;*
        IF SWITCH=DISK   ;*
        ...Disk interface code...
        END
        ...Rest of source file...
```

* - The code so indicated can be replaced with a single ELSE statement.

One point worthy of note is that the IF statement works by evaluating the expression and if it is zero it is taken as 'false', if it is non-zero considered 'true'. This means that the first IF statement could have been replaced with the following.

```
        IF SWITCH
```

But in this example it makes the statement less clear.

The IF statement may also be used inside macro definitions, enabling macros to change at invocation time the way they assemble.

Here is a small but useful example to illustrate this.

Supposing we wish to define a macro which performs the following operation - LD rr,(HL) where rr is another 16 bit register from the set HL,DE,BC. So if we were to invoke LD BC,(HL), the following instructions would be used.

```
1)      LD C,(HL)       ;Get low byte
2)      INC HL
3)      LD B,(HL)       ;Get high byte
4)      DEC HL          ;Restore HL
```

A similar sequence would be required for the register DE.

On the other hand we might wish to do this instruction - LD HL,(HL). In this case the following instructions would be needed.

```
1)      PUSH AF         ;Save AF
2)      LD A,(HL)       ;Get low byte
3)      INC HL
4)      LD H,(HL)       ;Get high byte
5)      LD L,A          ;Load up low byte
6)      POP AF          ;Restore AF
```

So we need three different code sequences for each type of instruction. This can be done with the IF statement in a macro definition.

```
              DEFCONST (HL,DE,BC).

              DEFSET RR = (BC,DE,HL).

              DEFMAC ("LD*,(*)",RR,HL)

              IF £0=0                  ;Case for LD BC,(HL)
                     LD C,(HL)
                     INC HL
                     LD B,(HL)
                     DEC HL
               ELSE IF £0=1            ;Case for LD DE,(HL)
                     LD E,(HL)
                     INC HL
                     LD D,(HL)
                     DEC HL
               ELSE IF £0=2            ;Case for LD HL,(HL)
                     PUSH AF
                     LD A,(HL)
                     INC HL
                     LD H,(HL)
                     LD L,A
                     POP AF
              END END END
              END.                     ;One for each IF

 0000 F57E2366
 0004 6FF1           LD HL,(HL)        ;Invocations of the macro
 0006 4E23462B       LD BC,(HL)
 000A 5E23562B       LD DE,(HL)
```

Argument modification

When using macros inside macros it is possible to modify the value that a macro definition sees a constant argument as. This is done by immediately following the argument in the macro invocation with a value to be added on to the constants (set related) value in square brackets. ie., LD C[1],A if invoked inside a macro would generate the code for LD D,A (see the Z80 macro library).

This facility can be used to simplify the above example.

```
              DEFMAC ("LD*,(*)",RR,HL)
              IF £0<2                  ;BC & DE case
                    LD C[£0*2],(HL)
                    INC HL
                    LD B[£0*2],(HL)
                    DEC HL
              ELSE
                    PUSH AF
                    LD A,(HL)
                    INC HL
                    LD H,(HL)
                    LD L,A
                    POP AF
              END
              END.
```

So although this is a complex facility to use it does simplify the code in certain circumstances.

Iteration and the variable

It is often useful for macros to produce tables etc., and to do this they must have a conditional looping statement - this assembler has two. The $WHILE - END sequence and the $REPEAT - $UNTIL sequence and there is a $BREAK statement for breaking out of these sequences. Their use is best illustrated by example because they are not very different from similar statements in high level languages.

Here is a macro which will fill a number of bytes with a particular value at assembly time.

```
                DEFVAR (COUNTER).

                DEFSET NN = 0 TO 256.

                DEFMAC ("FILL*BYTES-WITH*",NN,NN)
                COUNTER:= £0

                $WHILE COUNTER>0
                        DB £1
                        COUNTER:= COUNTER-1
                END
                END.

0000 AAAAAAAA
0004 AAAAAAAA
0008 AA                 FILL 9 BYTES WITH 0AAH  ;Here is how we use it
```

Notice that we used what looked like a label as a counter. This was in fact a variable and was defined as such in the DEFVAR statement which works in much the same way as DEFCONST. Variables may be used anywhere that labels are used but must be assigned to using the "=" rather than the EQU for normal labels. The advantage of variables over labels is that variables may be assigned to more than once without the assembler objecting. They can, of course, be used anywhere labels are used and can sometimes be used to advantage instead of labels but it is best to restrict their use, otherwise mistakes can be made which will not be picked up by the assembler.

The $REPEAT is used in much the same way as the $WHILE except that the condition is tested at the end of the loop rather than at the beginning.

```
ie.,            $REPEAT
                        ...some code...
                $UNTIL expression
```

When the $BREAK statement is encountered inside one of the above loops, assembly breaks from its current position and continues after the end of the loop. This can also be used to break out of macros.

Assembler terminating information

When the assembler terminates it prints out on the screen a few statistics about the assembly - the final value of the pseudo program counter ($), the final value of the LOAD pointer, the start and end addresses of the workspace used and, if any errors, the number of errors and the assembler pass they occurred on.

Creating a library

As has been stated earlier this assembler comes with a built in macro library containing all the Z80 instruction set. It is possible for the user to change this library and thus personalise the assembler as little or as much as he/she likes!. It would be possible, for instance, to get rid of the Z80 set and put in the macro definitions for the 6502 instruction set or probably any other microprocessor. One need not even put in micro instructions sets, the user could create his own language made out of macros.

To add a new library the user must first write it as he/she would any other assembler program (the Z80 macro library built into the assembler is supplied on the tape provided so that the user may modify its contents). The file is then assembled using the "K" option. The tables thus generated are then put at the end of the assembler. To save the modified editor/assembler see the section on NEW COPIES at the begining of the manual.

GENERAL USER GUIDE

Assuming that a file has been created, containing assembly source code in the correct format, it is assembled as follows.

1) Type CTRL 'A' while in the editor.
2) Enter option letters required and press <RET>.
3) Enter workspace address, or simply press <RET> in which case the workspace will go immediately after the source file.
4) Either, the assembler will return with no errors or any errors will be indicated and the user must correct and reassemble.

Guide to options

E - Stop and indicate error position with cursor.
L - Produce listfile.
S - Produce symbol table of labels/variables.
T - Produce other symbol tables (macros, sets & constants).
P - Send any output to the printer.
M - If file contains a load directive then store object in memory.
K - Create a new "built in library" (see section on libraries).

LOAD directive

This will cause any object generated to be put into memory at address following the load address (if the 'M' option is on).

ie.,

```
                ORG 1000H
                LOAD 8000H

1000 47         LD B,A
1001 21 02 00   LD HL,2

                END
```

will put hex codes 47,21,02 & 00, starting at 8000H, into RAM (although the code is assembled as if it was to go at 1000H).

LIST directive

This is a macro in the built in library which will switch on and off the list option (above) whilst printing a source listing. It is used as follows.

```
....source code (A)

LIST    OFF

....source code (B)

LIST    ON

....source code (C)
```

In this example (providing the list option was used in invoking the assembler) section (A) would be listed, section (B) would be missed and section (C) would be listed. If the assembler was invoked with the list option then only section (C) would be listed.

System variables

Certain system variables are available for use (with care) by the source file during assembly. These are accessed by putting "$n" where n is between 0 & 9. These may be used anywhere a standard label/variable is used and contain the following.

$0 (or just $)	Pseudo program counter.
$1	Load address pointer.
$2	Option flags.

-- bit 0	reserved
bit 1	print flag
bit 2	reserved
bit 3	list option
bit 4	symbol option
bit 5	reserved
bit 6	reserved
bit 7	error stop option
bit 8	'T' option
bit 9	'M' option
bit 10	Store to memory enable
bit 11	reserved
bit 12	'0' option
bit 13	'1' option
bit 14	'2' option
bit 15	'3' option

The '0','1','2' & '3' options are not used by the assembler but are available as a method of externally passing information into an assembling source file. This could be used in conjunction with the conditional IF statements to perform assembly switches without affecting the source file.

ie.,

```
IF $2 & 0001000000000000B

...source code (A)

ELSE

...source code (B)

END
```

So, if the assembler was invoked with the '0' option only source code (A) would be assembled and if the '0' option was omitted then only source code (B) would be assembled.

Errors

There are ten different error responses and each of these will cover a number of different but similar error conditions.

Here is a list of their meanings.

PHASE -- A label had a different value on pass three to that on pass two. This can be caused by macros generating different code on each pass, or labels not being initialized correctly.

SIZE -- This emcompasses a number of conditions mostly concerned with the size of a value.

SYMBOL -- A symbol of one type was found where one of another type expected.

ARGM -- Argument error. Usually on macro definitions.

UNDEF -- A symbol/macro has not been defined.

SYNTAX -- Syntax error often occurs in macro name strings when illegal characters are used.

INFINITE -- A loop using $WHILE or $REPEAT has iterated over 65535 times and is possibly an infinite loop.

INIT -- Label initialization error.

EOF -- Unexpected end of file found. Usually too few ENDs in file.

SYS -- System error - this is generated if a library was made which overran the start of the source file.