MANUAL

FOR

μSource

ASSEMBLER — FORTH — DEBUG

INITIALISING MICROSOURCE.

Microsource automatically links into the Spectrum's operating system on start up but when used with Interface 1 a Basic 'error' must be caused (eg. type in RETURN and ENTER) to correctly link up Microsource, Interface 1 and the ZX Spectrum. This operation must be performed after every power-up or NEW.

MICROSOURCE OPERATION - GENERAL

Microsource incorporates two languages and a Debug aid which are all 'invoked' by the use of Basic 'reserved variables'. The reserved variables which affect Microsource's operation are given below. You should not use any of these variables in a program other than for invoking Microsource's facilities.

Reserved Variable	Operation
Assemble	Invokes Assembler
Forth	Invokes Forth
Loader	Invokes object code manipulator
Debug	Invokes Debug aid
Forsv1	Forth stack pointer
Forsv2	Forth stack pointer

... Assembler ...

Microsource's Z-80 Assembler is invoked by LET assemble = 1. The assembler scans the user's Basic program for valid assembly language lines, which are held in REM ! statements as follows: (the exclamation differentiates the Assembler lines from other types of comments).

```
10  LET assemble = 1
20  REM !           org 8000h
30  REM ! code      ld a, 2
40  REM !           out (254), a
50  REM !           ret
```

Microsource, when invoked, assembles the code into memory (or outputs it to Microdrive, ZX Net, etc.) and produces a formatted listing showing object code etc. The listing may be directed to the Screen, ZX Printer, Microdrive, ZX Net etc.

Microsource automatically creates new Basic variables corresponding to the labels the user has defined in his assembly language program.Thus in the above example, RANDOMIZE USR code will run the assembled machine code program, located in memory at 8000 hex.

ASSEMBLY LANGUAGE STATEMENTS - SYNTAX

An assembly language program consists of labels, opcodes, operands, comments and pseudo-ops in a sequence which defines the user's program.Usually a source program begins with an ORG pseudo-op (which tells the assembler where to start putting the assembled code), otherwise it defaults to a starting address of 0000H.

Every line of assembly language has the form:

```
XX  REM ! label opcode operand;  comments
```

Where:

XX — is a Basic line number.

REM — is the Basic REM Keyword.

! — is the assembly language statement marker.

label — is an optional label. The label, if used, must start immediately after the '!' and can be any continuous string of alphanumeric characters. Spaces in labels are not allowed.

opcode — is a Z-80 opcode. The opcode must have at least one leading space between it and the '!' (or the label preceding the opcode) otherwise it will be taken to be part of the label.

operand — is the operand required after some opcodes.

; — starts a comment. Comments are ignored during assembly.

A typical line with one statement is shown below:

```
10  REM !loop ld hl, 1234h ; load hl with data
```

Note that the '!' marker can be used to terminate one assembler statement in a line and begin another, making multi-statement lines possible. This program:

```
10  REM !loop ld hl, 1234h
20  REM !     inc hl
30  REM !     ret
```

may be rewritten as:

```
10  REM !loop ld hl, 1234h! inc hl! ret
```

Note that with every new '!' the syntax has to be obeyed all over again, hence the leading spaces before inc hl and ret.

UPPER/LOWER CASE

Microsource Assembler does not differentiate between upper and lower case in labels, opcodes or operands. Comments are treated as a string and are repeated identically in the listing. In general, Basic keywords are not allowed, the exception being in arithmetic expressions where they constitute a valid mathematical function (see "expressions"). The = and = symbols must be typed out in full as the = = keywords are not allowed.

LABELS

A label is composed of a continuous string of one or more characters, and only the alphanumerics are allowed (ie. 0-9 and A-Z). Labels may be of any length. Examples of allowable labels are:

label5 transferthedata lopt99 P
loop dontreturnfromhere e5 Z1044

Labels may be declared as Basic variables (eg. LET datablock = 30000) and "imported" into an assembler program. Thus ld hl, datablock would be evaluated as "load register pair hl with 30000". For this reason no error messages are generated if a label is declared twice or more, and the first declaration of a label sets the value during the rest of assembly.

Labels are evaluated to addresses by the assembler and stored in the Basic variables area as integer variables. This allows a Basic program to call any portion of a machine code routine by simply referencing the label.

EXPRESSIONS

An expression is an operand entry consisting of one or more terms that can be evaluated to a valid assembler operand. Microsource's assembler differs from most other assemblers as regards expressions by allowing any Basic expression. For instance:

ld A, 2 + (3*5)
would be evaluated as ld A, 17.

Where the expression evaluates to a value with an integer and fractional part, only the integer value is taken. For instance:

ld A, SIN 2 (NB use the SIN keyword)

would be evaluated as ld A, 0. Where constants are required in an expression, they can be declared as Basic variables and imported as such:

ld A, datablock * offset value

Expressions may include labels declared in the assembly language program. This is useful in accessing lists of data eg.

ld hl, datablock + 2

NUMBER BASES in EXPRESSIONS

Decimal, hex and binary numbers are allowed in expressions in the form:

Hex numbers (ending in 'h') eg. 10FFh

Binary numbers (ending in 'b') eg. 10110011b
Decimal numbers eg. 1234

Hex numbers beginning with a letter (eg. FFh) should be preceeded by an '0' (eg. 0FFh) to avoid confusion with labels. Note that the Basic BIN notation for binary numbers is not accepted.

Number bases may be mixed in expressions eg. ld a, 12*12-10h+101b.

STRINGS in EXPRESSIONS

A 1 character string in single or double quotes will be evaluated to its ASCII value eg. 'k' and "k" both evaluate to 6BH.
String expressions can be of the form 'xxxxxx...xxx' or "xxxx...xxxx". These will be evaluated to the sequence of ASCII bytes corresponding to the string.

Note that the ! character cannot be used in a string as it starts an assembly language statement - it would have to be entered as DEFB 33 (see Pseudo ops) or imported as a Basic string variable.

Any Basic string variable can be imported and will be evaluated as a string expression. The following are examples of valid string expressions:

a$ 'hello' "don't say that" 'p'

The following are examples of invalid expressions:

"good grief!" '!' "mixed delimiters' 'no delimiter

When subsequent sections refer to -expression-, simply refer back to this section to find out what constitutes a valid expression.

ASSEMBLER PSEUDO-OPS (Assembler Directives)

There are 14 pseudo-ops which the assembler will recognise. These assembler directives, although written much like processor instructions (and entered in the opcode field) are commands to the assembler instead of to the processor.They direct the assembler to perform certain tasks during the assembly process but have no meaning to the Z-80 processor.

ORG -expression- Sets address reference counter to value of the -expression- eg. ORG 8000h or ORG CODE

LIST -expression- Listing from here on is sent to channel specified by expression as follows:
Channel 0 = No listing although errors are still reported.
Channel 2 = Normal screen (the default)
Channel 3 = ZX printer
Channels 4---15 = Any opened channel. eg. Microdrive data file/RS232/ZX Net. Note that channel MUST have been opened from Basic before the assembler is invoked.

OBJ -expression- Object code is sent to channel specified by expression as follows:

Channel 0 = No object code generated.
Channel 1 = Into memory directly (the default)
Channels 2---15 = To relevant stream, which MUST have been opened from Basic before Assembler is invoked. Code sent to channels 2-15 has 'headers' proceeding blocks of code. See 'object and list files'.

OPT -expression- Various options may be specified. The expression should evaluate to a number between 0 and 127 and alters all 7 options. Each option is represented by one bit as follows:

OPTION

1 Turn error checking off for assembly into protected areas of memory.
2 List the symbol table after assembly.
4 Wait for 'symbol shift' or 'break' on encountering an error.
8 Suppress macro expansions in listings.
16 List output in normal printer (ie.80 column) format.
32 List an object code map at the end of assembly (up to 8 blocks will be listed).
64 Terminate program on error to prevent accidental 'run' of faulty program.

EG. 10 REM ! OPT 2+8+32 - will list the symbol table, suppress all macro expansion and list the object code map.

DEFW -expression- Defines the contents of the two-byte word at the current address reference counter to be the byte specified by the expression. The least significant byte is located at the address reference counter while the most significant byte is located at the reference counter plus one. Note that DEFW can usefully be assigned to a label eg.

```
DEFW  code
DEFW  1020H
DEFW  5H  (evaluates to 0005H)
```

DEFB -expression- Defines the content of the byte at the current address reference counter to be the byte specified by the expression. The expression must evaluate to a single byte.
eg.DEFB 's' or DEFB 32h or DEFB 10101010b or DEFB 30h,31h,33h

DEFM -string expr- Defines the contents of n bytes of memory to be the ASCII representation of a string, where n is the length. The expression must be a valid string expression (see expressions). The string must not exceed 255 bytes in length.

DEFV -string expr- The string expression is parsed into allophones in standard CURRAH uSPEECH format eg. DEFV 'he(ll)(oo)'. Intonation changes with upper/lower case. See CURRAH uSPEECH programming manual for full list of allophones allowed in string.

DEFS -expression- Reserves n bytes of memory starting at the current address reference counter. The counter is incremented to step over the reserved area. This is useful for reserving areas of RAM for data, system variables etc.

EQU
Format:
-label- EQU -expression-

Equates a label to be equal to the expression eg. code EQU 1234H. Similar in principle to Basic LET command.Note that the expression is always evaluated to be 16 bits.

MACRO
Format:
-name- MACRO (—parameters—)

Begins a macro definition. An ENDM MUST end a macro definition. Macros cannot be defined inside other macros. See 'Macros'. Parameter list must be enclosed in brackets.

ENDM Ends a macro definition. No expressions allowed.

ASIF -expression- Assembly is turned off if the expression evaluates 'false'. When assembly is turned off, only ASIF and END pseudo ops are recognised. This conditional assembly is very powerful and allows overlays etc. (See 'Conditional Assembly').

END Terminates a block of code subjected to conditional assembly. No expression follows.

MACROS

Macros provide a means for the user to define his own opcodes. A macro defines a body of source text which will be automatically inserted in the source stream at each occurence of a macro call. In addition,parameters may be passed to the macro, providing a cpability for making limited changes at each macro call.

Macros can be used by the programmer to handle repetitive instruction sequences where only certain parameters are changed with each repetition. Macros offer the advantages over recoding each instruction sequence of simplified program coding and reduced chances of error.

The main disadvantage of macros in comparison with subroutines is that the amount of code generated can be significantly increased.

The ideal use of macros is in conjunction with subroutines; a call to the subroutine can be included in the body of the macro, whereas the mechanics of picking up the parameters and storage of results can be handled by the macro.

There are two elements in the utilisation of macros. Definition, where the body of code to be generated each time the macro is called is defined, and invocation, when the macro is called.

A macro definition consists of the block of code required, set between a MACRO pseudo-op(to start the definition)and an ENDM psuedo-op(to end the definition).

Each statement between the MACRO and ENDM is part of the macro body. These statements are placed in a macro file for use when the macro is called. At expansion time an error will be generated if another macro is defined within a macro.No statements are assembled at definition time including any Assembler psuedo-op within it.

A parameter may exist anywhere in the macro body and for any statement field, including labels, opcodes and comments. A parameter is an unknown quantity at definition time which is assigned a quantity or value at macro call time. A parameter is denoted by a hash (#) followed by any character from A to Z. 26 parameters can thus be referenced in a macro definition. For instance, ld a, #B at definition time would be expanded to ld a, c at call time if the #B parameter was assigned the value 'c' when the macro was called. As another example, DEFM '#R' at definition time would be expanded to DEFM 'this is a message' at expansion time if #R was assigned to be 'this is a message'.

For every macro definition there is an internally defined macro parameter $n. This parameter may be used usefully to generate local labels and control recursion within the macro body but should not appear in the formal parameter list in the definition. When the macro is called, each occurence of $n is replaced by a string representing a number between 001 and 255. This string is constant over a given level of macro expansion and increases by one for every macro call.

A typical use of the $n string is to provide unique labels to a macro that is expanded multiple times to avoid duplicate label problems.

$n = A number
LOOP$n = Eg of a label

MACRO DEFINITION

A macro may be defined in the following format:

```
-name-   MACRO   ( --parameter list-- )

     block of code

         ENDM
```

WHERE: -name- is the name of the macro. Any further use of this name will cause an expansion of the macro. The name follows all the usual rules for labels.

MACRO is the 'macro' pseudo-op.

(-parameter list-) is the list of all the parameters that will be needed inside the block of code, enclosed in brackets. eg. if the macro definition includes parameters #A, #B and #Z, then the list MUST be (#A,#B,#Z). Do not reference $n in this list.

MACRO CALL

A macro may be called after it has been defined in the following format:

```
          -name-    -actual parameters-
```

The (name) is placed in the opcode field (preceded by a label,if desired) and the actual parameter list in the operand field. Parameters are supplied to the macro as a list, separated by commas. Do not enclose the actual parameter list in brackets. Where the parameter is a string, it must be enclosed in delimiters ('-' or "-"). Whatever lies between the delimiters will be passed as a string. A single outer level of quotes acts as a null parameter.

Any parameter not supplied at call time will be set to blanks.

Symbols that are passed as parameters are passed by name and not by value. In other words the parameters are not evaluated until the expansion is produced- the process is simply one of character substitution.

An example of a macro definition and its call and expansion are shown below.

Definition

```
        Moveblock  Macro (#A,#B,#C)
                   ld hl, #A
                   ld de, #B
                   ld bc, #C
                   ldir
                   ret
                   Endm
```

Macro Call

```
        Move 99     Moveblock 1000H, 2000H, 256
```

Macro Expansion During Assembly

```
        Move 99     Moveblock 1000H,2000H,256
                    ld hl, 1000H
                    ld de, 2000H
                    ld bc, 256
                    ldir
                    ret
```

Note that the Macro call line does not generate any code. Parameters must be supplied in the order they are given in the formal parameter list and if a parameter is to be deliberately omitted, a null parameter ("") should be used to ensure that the order is not changed.

Recursion may be employed to generate multiple calls of a macro:

Definition

```
        Borderpaint   Macro (#A)
                      asif $n #A
                      call flash
                      borderpaint
                      endm
                      -
```

Call

```
borderpaint 3
```

Expansion

```
call flash
call flash
call flash
```

CONDITIONAL ASSEMBLY

Conditional assembly allows blocks of code to be assembled into memory when a certain condition is "found". The block of code which is under conditional control is enclosed by the ASIF -expression- and END pseudo-ops, and is not assembled if the -expression- evaluates 'false'. For example:

```
asif b = 1
ld a, 3
call rewind
ret
end
asif b = 2
ld a, 2
call fastforward
ret
end
```

Here the first block of code will be assembled if b = 1 and the second if b=2. b could be a Basic variable or even the Address counter $. Conditional assembly is extremely useful in a macro:

```
moveblock macro (#A, #B, #C, #S)
          asif #S  ""
          push hl! push de! push bc!
          end
          ld hl, #A! ld de, #B, ld bc, #C
          asif #S  ""
          pop bc! pop de! pop hl
          end
          ret
          endm
```

Here all the registers are pushed on entry and popped on exit if the fourth parameter is present. When register preservation is required the macro call could thus be:

```
moveblock 1000H,2000H,256,SAVE
```

The fourth parameter could be missed off from the macro call if register preservation was not important. When used with conditional assembly macros can, if used intelligently, generate very readable listings and yet generate no more object code than if it had been done longhand. Sometimes, macro expansions are not desirable on assembled listings and they can be suppressed with OPT 8.

ASSEMBLER OPERATION

Microsources Z-80 assembler is invoked from Basic by the command:

```
LET assemble = 1
```

Actually any number can be substituted for the '1' as it is the action of creating the variable which invokes the assembler.

The assembler, on invokation, tries to make room for its work space (about 1100 bytes) which it borrows from the machine stack. If there is no room the assembly will be aborted with the error 'M Out of Memory'. If all is well,the assembler scans the program, from start to end, looking for assembly language lines, and assembles them as instructed by the various pseudo-ops and conditional directives. If Basic variables are to be imported into the assembly language program, they must be declared by LET statements before the Assembler is invoked.

Note that during assembly, BREAK can be pressed to abort altogether or caps shift can be pressed to pause the listing. Symbol shift can then be pressed to continue, or BREAK to abort.

The Assembler works in two passes:

PASS 1 generates values for all labels and assembles them into a symbol table, which in fact is the Sinclair's variable area. All symbols can be accessed from Basic after assembly is complete.

PASS 2 performs the actual assembly. Object code is generated, expressions are evaluated and the various options take effect. Macros are also expanded. The object code is sent to memory or to the various streams specified in the OBJ pseudo-op and a listing is generated. The listing is provided so the user can see which bytes have been assembled where and various list options provide symbol table dumps etc.

During Pass 2, any errors which may have occurred are also reported. If OPT 64 is specified, the assembly will automatically abort with Error X 'Program Terminated' if an error is found. Note that LIST 0 still allows errors to be reported.

Once assembly is over and the listing etc output if required the assembler terminates and returns control to the next Basic line after the Assembler invokation. If the Assembler was invoked from a directly entered line, then control will be returned to the user as normal, or the next statement if it was a multi-statement line.

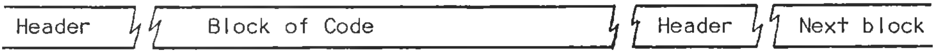
The various object code and list file formats are specified next.

OBJECT CODE FORMAT

By default, object code is assembled into memory, starting at the assembly program address counter set by ORG and defaulting to 0000H. If code is attempted to be assembled between 0000H and 3FFFH, then none is actually sent as this area is ROM. This only applies to assembly directly into memory.

OBJ 0 prevents any code from being assembled, although a list file may still be produced.

OBJ 2--15 assembles the code to the relevant stream, which MUST have been opened from Basic prior to the assembler invocation. When code is sent to any of these streams it is sent in blocks of code preceded by 'headers' as follows



The block length is variable and a new block is begun every time a fresh ORG is encountered. In other words, every block represents one contiguous block of machine code, so many programs may only require one block.

The header for each block consists of five bytes:

A marker byte set to OFFH
Load address lo
Load address hi
Length lo
Length hi

When all the blocks have been sent, a header is output with its length bytes set to zero. This marks the end of the object code output.

AUTO LOADING of OBJECT FILES

Object files down loaded in this format (eg. to a Microdrive data file,ZX Net etc.) may be automatically loaded by the command:

```
          Let loader = -stream-
```

Where -stream- is a number corresponding to an input stream, which MUST have been opened from Basic before invoking the loader. Microsource automatically handles the removal of the headers and places the incoming code in the correct place in memory. For example, suppose that an object file has just been assembled to a Microdrive 2 data file called "object" and the output stream has been closed. Then:

```
          OPEN #4; "m"; "object"
          LET loader = 4
          CLOSE #4
```

This will automatically load the incoming data into the correct place in memory. System error "W, Bad Stream or Data" will be reported at the bottom of the screen if the input stream has not been opened or if the incoming data is not a Microsource object file.

PROTECTED AREAS etc.

Microsource will normally not generate any object code for assembly into 0000H - 3FFFH, as this is ROM in the standard Spectrum. Error messages of the form 'Bad Org' will be issued and no code generated if code is attemted to be assembled into the following areas:

```
4000H - 5800H  (the Screen RAM)
5801H - 5B00H  (the Atributes RAM)
5B01H - 5AE9H  (the Basic System Variables)
```

In addition, the Interface 1 'extra' System Variables and the Microdrive Channels and Maps are 'protected' when they are present. Microsource also 'protects' its own system area which it steals from the machine stack.

Note that the above does not prevent you from writing and running a machine code program which alters where the stack is although if you wish to return to Basic afterwards you will have to restore the stack pointer.

In certain circumstances the user may require all the protection features turned off; this may be done by specifying OPT 1. This option may be required for sending code to another Spectrum or for users who simply want to be awkward. But beware! If you overwrite the Basic system variables or Microdrive maps, then on exit from the Assembler you may find that Basic will crash, or you may be left with an unclosed file on your Microdrive.

LIST FILES

Unless directed otherwise by the LIST pseudo-op, the assembled listing will be sent to the screen. A typical listing is shown below. The normal listing format has been optimised for the Spectrum screen :

```
10  8000
               Org 8000h
20  8000 3E AF
               pixad ld a, 0afh
30  8000 90
               sub b
40  8003 C9
               ret
0   total errors
```

Each line of listing occupies two lines to make it clearer. The format is:

Signon message

```
Basic line number   Assembly address Object code
                    Label            Opcode  Comments
```

WHERE :

Basic line number - is the line on which the code was written. The line number is simply repaeted where multi statement assembly lines are employed. Note that multi statement source lines are separated automatically in listings.

Assembly address - address at which code is to be placed if code is generated. It is also the current address of the ORG counter. Unless an ORG is specified, it starts off at 0000H.

Object code - the hexadecimal representation of the machine code which each line of assembly language generates.

Label - as source text.

Opcode - as source text.

Comments - as source text.

Note - Microsource will try to make the listing neat even when long lines are encountered. This is particularly useful in DEFM statements where a lot of object code is generated.

Using the LIST pseudo-op, the list format may be redirected to another device or stream. LIST 0 prevents any listing from being produced. Errors are,however reported. LIST 2 sends the listing to the screen. You will need to specify LIST 2 if you have directed the first half of a listing to the printer, say, and wish to view the last half on the screen. LIST 2 is the default if no list is specified. LIST 4--15 will output the list file to the relevant channel, which MUST have been previously opened from Basic. You can send the list file to a Microdrive data file if you wish, using this command. Do not forget to CLOSE the stream after you have finished with it.

The listing may be modified by use of the OPT pseudo-op (see under pseudo-ops)

OPT 2 Lists the symbol table after assembly. Note that LIST 0 will suppress OPT 2. The symbol table is a list of all the labels and their assigned values, and is very useful in debugging a program.

OPT 4 Halts the listing every time an error is encountered and allows the user to examine what has gone wrong. The listing can be resumed by pressing 'symbol shift' or the assembly aborted altogether with a 'break'.

OPT 8 Suppresses all macro expansions in listings. The macro will still be expanded into object code but will only appear as its source statement in the listing. This prevents listings being cluttered with repetitive chunks of code.

OPT 16 Outputs the listing in a format suitable for an 80 column printer. If you have an 80 col.printer attached to the RS232 output of your Spectrum you can use this option to obtain a professional listing.

OPT 32 Outputs an object code map at the end of assembly. This map tells the user where the blocks of code start and how long they are. This information is provided in both hex. and decimal. This map will be useful for users with a tape-based system who wish to save their machine code or tape using the SAVE .... CODE command, which expects a start address and a length.

AUTO ABORT

In some circumstances it may be desirable to prevent an assembled program with errors from being automatically run. For instance, this program will crash the computer:

```
10  LET ASSEMBLE = 1
20  REM ! org 8000h
30  REM ! crash rubbish
40  RANDOMIZE USR crash
```

The assembler will obediently try to assemble the source, report the errors, and then drop back into Basic, which will promptly crash when the USR call is made.

OPT 64 will automatically abort a Basic program if an error is detected in assembly, and the error message "X : Program Terminated" will be issued. For instance, if OPT 64 is specified in the above program, the program will generate a listing and terminate as follows:

```
20  8000
                  org 8000h
30  8000
                  crash rubbish
no such macro
1 total errors
```

```
X : Program Terminated, 10 : 1
```

Z-80A - CPU

ASSEMBLY LANGUAGE MNEMONICS

The Assembler follows standard Zilog mnemonics with no departures from the norm for opcodes or operands.

EXAMPLE ASSEMBLY LANGUAGE PROGRAMS

Some examples are given in the final manual of programs to handle pixel plotting, text handling and sound output. The following example makes a "zap" sound suitable for a game:

```
10  LET assemble = 1
20  REM !     org 32000
30  REM ! zap ld bc, 16
40  REM ! zpl xor 16
50  REM !     push bc
60  REM ! del djnz del
```

```
 70  REM !      pop bc
 80  REM !      out (254), a
 90  REM !      djnz zpl
100  REM !      ret
110  RANDOMIZE USR zap
```

NOTES FOR USERS OF TAPE BASED SYSTEMS

It is not possible to download assembled object code directly to tape during the assembly process in a format suitable for direct re-loading and execution on the Spectrum. The programmer must assemble the object code into memory,then save it onto tape using the SAVE ... CODE command. OPT 32 should be used to devise an object code map for use with this command.

NOTES FOR USERS OF MICRODRIVE SYSTEMS

Microsource is fully compatible with Interface 1. However before invoking Microsource, the user MUST cause a Basic error (eg. type in RETURN and ENTER) This is to ensure that the 'shadow' system variables (see p.45 Interface 1 manual) are set up and Microsource correctly linked into Interface 1's error handling mechanism.

Note that this only applies after a power-up or a new. You don't have to do it at all if you are using the Microdrive's auto-run facility.

ASSEMBLY TIME ERRORS

Assembly errors are printed out in the list file output to the stream specified by the LIST pseudo-op. When LIST 0 is specified, the errors are output to the screen, along with the offending line.

There are 3 assembler errors possible. These are:

ERROR	CAUSE
Bad expression	An expression doesn't have the correct number of brackets or couldn't be evaluated eg. if a variable is referenced which is as yet undefined.
No such macro	Couldn't find the macro; possibly a bad label or a mis-spelt opcode.
Bad operands	Wrong operands for that opcode.
No room	Assembler has run out of memory space.
Bad allophone	Illegal mnemonic for allophone in DEFV pseudo-op expression.
Out of range	A number in a relative jump, restart, IN or OUT instruction is out of the allowed range.
Bad org	The current address of the ORG pointer would result in Basic or Assembler workspace being overwritten. OPT 1 can suppress this.
Bad macro	A bad macro definition or call eg. if too many parameters are specified in a macro call.

In addition to those assembly-time errors which are reported in the list file there are two system errors which cause Microsource to abort. These are reported at the bottom of the screen like a Basic error - (n, m are the line and statement numbers where the errors ocurred, just like a Basic error).

W Bad stream or Data, n:m - will be issued by LET loader = n command if no stream opened or if incoming data in stream is not a Microsource object file.

X Program Terminated, n:m - will be issued if OPT 64 is specified and an error is found.

------[END of SECTION]------

...Forth...

FORTH - AN INTRODUCTION

Forth is a stack-orientated language which utilises words to manipulate information held as items on the stack. Words may be built up out of other words and compiled into the central Forth "dictionary". Once in the dictionary Forth will respond to the new words as if they were an innate part of the language.

The user in effect writes his own language as he goes on, defining new words to perform specific functions. Over 70 words and 6 control operators are instantly available to the user as they are held in Microsources ROM. The users dictionary is built up in RAM and linked to the ROM dictionary and various Forth words allow the user dictionary to be saved, reloaded and linked to existing dictionaries or back into the ROM dictionaries.

USE OF WORDS

A word is invoked simply by placing it in the string of Forth words following the invocation LET forth=1. Words are separated by a space between each word. Numbers are entered in a similar way in either decimal,binary,or hex; when Forth encounters a valid number it simply places it on the stack.The following sequence of Forth words places the number 19 on the stack and performs the three Forth words DUP, +("add") and .("dot"):

```
19 DUP + .
```

FORTH INVOCATION

Microsource Forth is invoked by LET forth=1. The forth scans the users Basic program for valid Forth lines, which are held in REM # lines as follows: (the hash differentiates Forth lines from other types of comments).

```
10  LET forth=1
20  REM # 12 12 DUP * + . CR
```

Forth, when invoked, interprets all the Forth lines it finds following the invocation. As many Forth blocks as you want may be used inside a program, as long as each block is preceded by a LET forth=1 statement, to invoke the interpreter. As soon as the interpreter finds a line which does not start with REM # the Forth execution is terminated and control returned to Basic.
The following program, for instance, has two Forth blocks and three Basic blocks:

```
10  LET forth=1
20  REM # 1 DUP DUP DROP DROP DUP .
30  REM # DUP SWAP + DUP .
40  LET a=2 : REM  forth is terminated now
50  LET a = a+1
60  LET forth=1
70  REM # DUP * .
80  PRINT a
```

Use of Forth in this manner allows Forth blocks to be integrated and run with Basic either as large units or many separate portions. Forth words allow Basic variables to be "transported" between Basic and Forth, so parameters may be passed to and fro.

Forth may be used in 'direct mode' just like Basic by preceding with an invocation:

```
LET forth=1 : REM # 12 12 # .
```

Upper and Lower Case.

Microsource's Forth, unlike its Assembler, is 'case-dependent', that is, it is fussy about whether you use upper or lower case. All ROM-defined words (ie.the words that are in the system at power-up) must be invoked in UPPER CASE eg. DUP, ROT, SWAP, etc. All user-defined words (the ones you build up yourself) are defined and invoked in LOWER CASE eg.stop, go, printmyname, etc. Although this is not standard, it allows the user to quickly spot his own words in a program and differentiate them from the system's.This greatly improves the readability of long programs.

Like the Assembler, the symbols > =, <= and <> must be typed out in full ie.less than or equal to, (rather than by using the keywords ie. <=)

THE STACK

Forth supports a parameter stack which should not be confused with the Z-80 's machine stack. A stack is a 'first in last out' buffer which holds items pushed onto it. The last item to be pushed onto the stack will be the first item to be popped off it. All numbers, addresses, data etc. in Forth are held on the stack and various stack operators may be invoked to manipulate items on the stack. Arithmetic is also performed on the stack, for instance, the word + ("add") pops the last two items off the stack, adds them together, and places the result back onto the stack.

A consequence of this implicit use of the stack is that Forth uses POSTFIX or Reverse Polish(as it is often known) notation for all its sequential operations. The sequence PRINT 2+3+4 in Basic is equivalent to the Forth 4 3 2++. A pleasant consequence of using Postfix as opposed to infix notation is that brackets and precedence rules are redundant, although the programmer has to think harder when doing the ordering operation.

There is also a return stack which is similar in operation to the parameter stack but which is solely concerned with handling internal loop and control indices. The return stack is manipulated implicitly by the forth control operators and explicitly by the return stack operators. Unless mentioned otherwise all reference to the stack refer to the parameter stack. Note that the stacks are not cleared by LET forth=1 but are cleared by Basic RUN and CLEAR.

FORTH and NUMBERS

Microsource Forth supports 16-bit integer numbers and depending on which words are employed to operate on the stack, numbers can be treated as either unsigned 16-bit integers or signed 2's complement numbers. For a review of 2's complement arithmetic, refer to Fig. 3.

This topic can cause newcomers to Forth considerable confusion as numbers popped off the stack which are greater than 32767 mysteriously appear as -32768 when printed using . ("dot"). The word U. ("U-dot") is used to print out a number which is a 16-bit integer and this simple example shows that the representation of signed and unsigned numbers is not inherently different - whether a number on the stack is considered to be unsigned or signed depends on the conversion routines applied to it.

When transporting variables to and from Basic careful attention should be paid to the number format when using GET and PUT. (This does not apply to COPY). The range for GETing variables from Basic and transferring them to Forth is -32768 to +32767. Numbers outside this range are still transferred but their representation will be wrong and it will be easier to use COPY.When Forth transfers the top of the stack to a Basic variable, it is always transferred as an unsigned integer ie. as a number between 0 and 65535 and will be stored accordingly in the Basic variable. A signed 2's complement number therefore needs special treatment if it is negative. In the manual, the notation n is used for signed 2's complement numbers and u for unsigned 16-bit integers.

Microsource Forth does not support floating point arithmetic - if floating point calculations are required then either scaled integer arithmetic may be used or the Spectrum's floating point calculator may be called from a machine language program (see Example Assembler Programs) and parameters passed using the USR word.

Fig. 3 REVIEW OF TWO'S COMPLEMENT ARITHMETIC

When using unsigned arithmetic all values are positive and the maximum value allowable depends on the number of bits. In 16 bits the lowest value is:

    0000000000000000

which equals zero. The highest value is:

    1111111111111111

which is equivalent to decimal 65535 (ie. 65536 numbers, including 0).

When using signed 16-bit arithmetic,on the other hand,the number of values that can be expressed is still 65536 but the range here is -32768 to +32767.

In signed 16-bit arithmetic the highest value (32767) is represented in binary form as:

    0111111111111111   (1 zero and 15 ones)

whereas the lowest value (-32768) appears in binary as:

    1000000000000000   (1 one and 15 zeros)

The high-order bit called the "sign-bit" is zero for positive numbers and one for negative numbers.

It is important to realize that the representation of signed and unsigned numbers is not inherently different. That is,

    1000000000000000

represents either -32768 or +32768 depending on the context. Whether a binary number is considered to be signed or unsigned is a function of the conversion routines applied to it.

The computers representation of negative numbers is called "two's complement arithmetic". To understand two's complement think about what happens when you exceed the limit of 16-bit arithmetic. The highest (unsigned) value is 65535. If you add one to it, you need a 17-bit value to record the result:

    10000000000000000

To the computer the one in the seventeenth place would be lost and the "result" would be zero. If you had added a two the result would be one, and so on. In effect the 65535 acts as if it were -1. In 16-bit signed arithmetic the value represented by 65535 is called the two's complement of one (-1); 65534 is the two's complement of two (-2), etc. All negative two's complement numbers have a one in the most significant bit (the sign bit) because their unsigned equivalents exceed 32767.

NUMBER BASES

Numbers may be passed to the Forth stack in the input word string as Decimal, Binary, or Hex in the forms:

Hex numbers	(ending in h)	eg. 103Bh
Binary numbers	(ending in b)	eg. 10101101b
Decimal numbers		eg. 1028

Note that you cant place a negative number directly in the input string. Ie. use NEG after a number to input it and negate it eg. 123 NEG will result in -123 on the stack.

Numbers may be printed out from the stack to the current stream in Decimal or Hex form using the . , U., and C. words. Output in the other number bases (eg. Octal) may be handled quite easily - see Example Forth Programs for a suggested way to do this.

COMPILING NEW WORDS

New words may be defined in forth by invoking the Forth compiler.The compiler is invoked from within Forth by the % symbol and the new word definition terminated by ; (semicolon). The % sign for opening a word definition is not standard (the colon is usually used) but is unavoidable as the colon forces the Spectrum into K mode.

Suppose you wanted to define a new word CUBE which takes off the last number on the stack and replaces it with its cube.
The way to perform this operation in Forth is DUP DUP * *. So the new word CUBE would be defined like this.

```
10  LET forth=1
20  REM # % cube DUP DUP * * ;
30  REM # 3 cube .
```

Once a word has been compiled it will be recognised as a valid Forth word as it has been linked into the dictionary. Forth may be terminated, Basic resumed, and Forth re-invoked and still the new word will be recognised. The only way the definition is lost is when a RUN or CLEAR is executed (this clears the variables area and resets the Forth stacks) or when the Forth words FORGET and EMPTY are invoked. Note that LET forth=1 does not clear the stacks or lose any definitions.

The Forth dictionary may be saved to tape or Microdrive cartridge as a block of code using the Basic SAVE ... CODE command. The address and length information for the dictionary is obtained using the HERE and LAST words (see Saving and Loading Dictionaries) and the code block may be reloaded from Basic. The first words in any Forth program to use a reloaded dictionary should be STACK and LINK, to link up the code so that all the previously defined words may be used.
Note that whenever a RUN is executed, any dictionary in existence is not actually destroyed it is just the linking pointers which have been reset. Providing the user has previously recorded the values for the start and end of the dictionary he will be able to relink an existing dictionary in with LINK.

SAVING and LOADING DICTIONARIES

Since a dictionary is just a contiguous block of memory,it can be saved using the normal SAVE ... CODE commands, given the start and end.

A dictionary starts at parameter stack top +1. Thus if you have used the STACK word, it will be at the number you gave, plus one. eg. after STACK 33000 then the dictionary start will be at 33001. If you have not used the STACK word at the start of your Forth program then HERE U. will iprint out the required value. You must use HERE U. at the start of your program (before any definitions).

The dictionary end can be found by invoking HERE after all your words have been defined.

So if the start was 33001 and the end 33087, then the dictionary may be saved to tape or Microdrive using the Basic command SAVE "name" CODE 33001, 87.

Three values are vital for a dictionary to be reloaded. They are the start and end of the dictionary and also the pointer to the last word. This can be found by using the word LAST after all your words have been defined.

As an example:- assume that LAST . returns the value 33070.

Given these three numbers - 33001, 33087 and 33070 - your dictionary may be reloaded and linked. This is achieved using the STACK and LINK words.

After loading the code back into memory using LOAD ... CODE the first line of your Forth program should read:

```
10  LET forth=1
20  REM # STACK 33000 33087 33070 LINK
```

Which will link in the dictionary. The STACK word takes the number 33000 and relocates the stack here; then LINK takes the two numbers on the stack (33087 and 33070) and uses them as dictionary end and last word pointers.

Note that you must always use the STACK word when reloading dictionaries although it can be omitted if you are not loading a dictionary and you are happy with the default value when Forth is first invoked (see Fig.4)

RECURSION and WORDS WITHIN DEFINITIONS

If you think about it there is nothing to stop you from defining a word which contains other user defined ones, providing they are already compiled and in the dictionary. For instance, a Forth word called driveoff might be defined as:

```
% driveoff getincar shutdoor startengine putingear brakeoff;
```

WHERE: getincar, shutdoor etc. are all words which have been defined earlier.

You cannot open a new definition (ie. use a %) inside a definition, or use the words CONST, VAR and '. Comments may be used inside definitions but they must NOT be the final word - ie. avoid these -) ;

However one of the things you can do is use the word which you are defining, inside its own definition. This apparant paradox can be explained by considering the way in which the compiler builds up the word definition. Consider the following:

```
10  LET forth=1
20  REM # % loop DUP DROP loop ;
30  REM # loop
```

The compiler starts by setting up the word loop in the next free cell in the dictionary. It then considers what is inside the definition and finds "loop". Since "loop" by now is set up, its name is recognised as beingvalid and so the compiler places its address in the code field along with the addresses of DUP and DROP. When the word "loop" is invoked DUP and DROP are executed, then DUP and DROP, and the operation repeats without end. Left to itself this word would recurse indefinitely until the computers memory had run out. However , if used carefully, recursion can be used to great advantage. The key to writing recursive definitions is to always ensure that the recursion will close ie.stop recursing at some point. As an example, consider this. A word fact! is to be defined which will generate the factorial of a number on the stack and leave the factorial on the stack. The word definition could be:

```
% f DUP DUP 1-SWAP IF >R * R> f ELSE DROP DROP THEN;
% fact! DUP 1- f ;
```

The word 'f' is the recursive bit and performs the successive multiplication of the components of the factorial. The IF ... ELSE ... THEN clause tests to see if the calculation is complete each time and if not calls 'f' again to perform the next multiplication. When the multiplier is 1 the recursion ceases.

The word 'fact' sets up the conditions for the recursion.

ERRORS in FORTH

Forth error messages are given at mn-time at the bottom of the screen in a similar way to Basic errors. They conform to the same format - a code letter the message itself, and the line and statement number where the program stopped due to the error. The messages are:

S Bad Forth Word This could be caused by a misspelt word, spaces being omitted between words or using a word which has not yet been defined. System words will not be recognised unless they are in upper case - see 'Upper and Lower Case'.

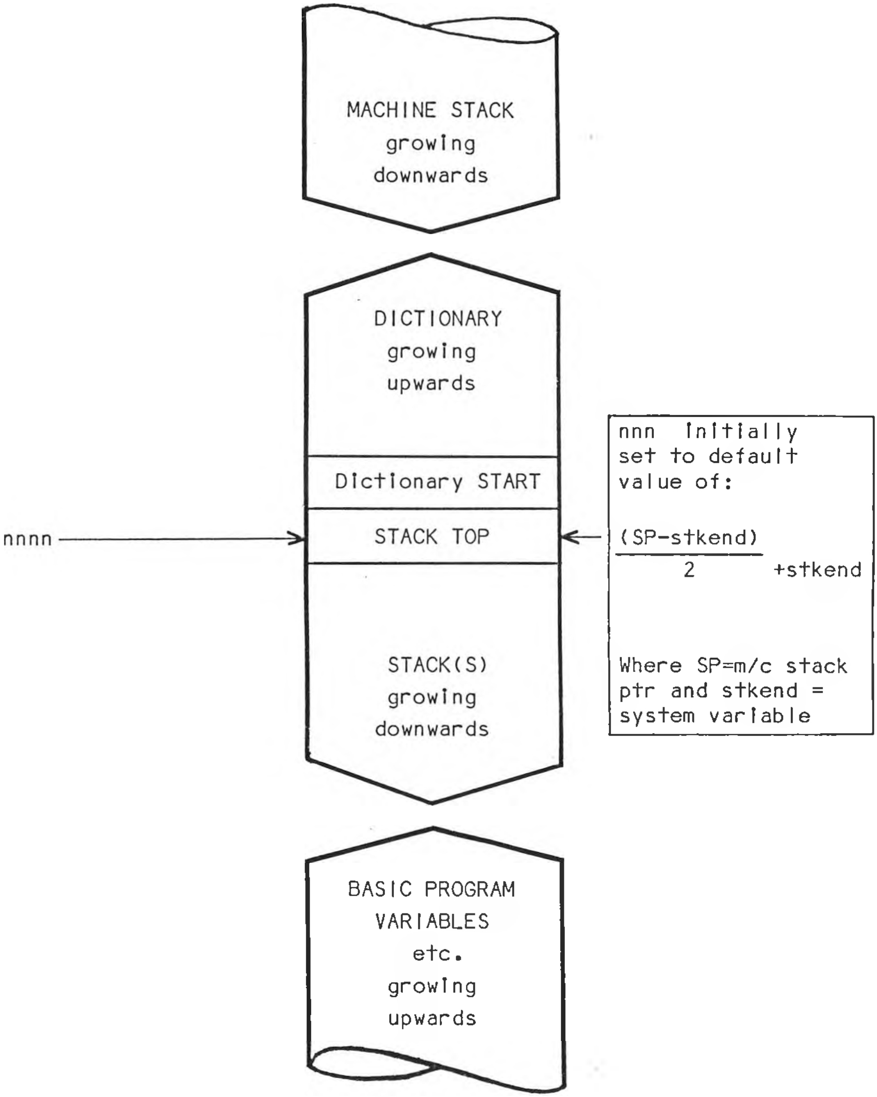
T Bad Structure This error is flagged by the compiler if definitions are left unclosed or control structures incorrectly specified.

U Transfer Error Caused by referencing a Basic variable which does not exist.

V Out of Stack Stack empty when an item was attempted to be removed.

All Forth control structures may be broken out of (once compiled) by pressing BREAK, as for Basic.

Fig. 4 : FORTH MEMORY ALLOCATION



nnn initially set to default value of:

$$\frac{(SP-stkend)}{2} + stkend$$

Where SP=m/c stack ptr and stkend = system variable

Fig. 5 : DICTIONARY CONSTRUCTION (for one word)

THE WORD cube IS USED AS AN EXAMPLE

<table>
<tr><th>BYTES USED</th><th>PURPOSE</th></tr>
<tr><td>2</td><td>Pointer to previous word</td></tr>
<tr><td>1</td><td>Length of the word (=n)</td></tr>
<tr><td>n</td><td>C
U
B
E
characters of the word</td></tr>
<tr><td colspan="2">There then follows the code for executing the word. This is normally (Forth-79) a series of addresses pointing to the constituent words. Microsource Forth is different in that the addresses it holds here, are actual call addresses to the required routines, ending with a RET. This has the advantage of greater speed in executing and is called directly threaded code.</td></tr>
<tr><td>1</td><td>RET (C9H)</td></tr>
</table>

GLOSSARY OF FORTH WORDS

It is beyond the scope of this preliminary manual to give full explanations of all the Forth words embodied in Microsource, especially where these words are standard Forth-79. The words are therefore presented in the form of tables with brief explanations and longer explanations where appropriate.

DIFFERENCES BETWEEN MICROSOURCE FORTH and FORTH-79

(Table references refer to Tables 1 - 9)

STACK MANIPULATION OPERATORS (See Table 1)

SWAP, DUP, ?DUP, DROP, OVER, ROT as standard
PICK (new word) enables selective picking of any item off stack.
RP (new word) enables selective picking of any item off return stack.

OUTPUT OPERATORS (See Table 2)

., ?, U	as standard but no trailing spaces printed.
H. (new word)	Outputs 16-bit no. as hex digits.
C. (new word)	Outputs 8-bit no. as hex digits.
TYPE	as standard.
."XXX"	as standard.
EMIT, CR, SP	as standard (SP = Forth-79 SPACE)

ARITHMETIC and LOGICAL STACK OPERATORS (See Table 3)

+, -, 1+, 1-, 2*, 2/, *, /, MOD, /MOD, NEG, MAX, MIN, ABS, AND as standard.

OR, XOR (new words)	like Z-80 OR and XOR, but operating on last two stack items.
RL, RR (new words)	Bitwise left and right rotate of last item on stack.

ADDRESS OPERATORS (See Table 4)

@, !, +!, C@, C!, '	as standard.
CONST	same as Forth-79 CONSTANT.
VAR	same as Forth-79 VARIABLE

COMPARISON and TESTING OPERATORS (See Table 5)

=, <>, >, <, 0<, NOT >=, <=,	as standard. 0 = is not included.

RETURN STACK OPERATORS (See Table 6)

I, I', J, >R, R>	as standard.
RP (new word)	enables selective picking of any item off return stack.

CONTROL OPERATORS (See Table 7)

(only usable within definitions)

IF xxx ELSE yyy THEN zzz	as standard.
DO ... LOOP	as standard.
DO ... +LOOP	as standard.
BEGIN ... UNTIL	as standard.
BEGIN xxx WHILE yyy REPEAT	as standard.
ABORT (new word)	forces exit from Forth, whether in loop or not.

DICTIONARY OPERATORS (See Table 8)

HERE, ALLOT	as standard.
,	as standard.
C,	as standard.
H (different to Forth-79)	places address of dictionary end pointer onto stack.
FORGET xxx, EMPTY	as standard.
LAST (new word)	Pushes current pointer to last word to be defined onto stack. Used in loading/saving dictionaries.

SPECIAL OPERATORS (See Table 9)

ABORT	Forces an exit from Forth or Forth loops.
# (new word)	Used in changing output stream.
STACK, LINK (new words)	Used in moving stack area and re-linking dictionaries loaded from tape or Microdrive.
(XXX ... XXX)	Start and end a comment.

The special operators USR GET PUT COPY and TOK deserve special mention as these are radical departures from the normal Forth. USR enables a call to a previously assembled machine code routine and a parameter may be passed to it in the BC register pair. On return the BC register pair is pushed onto the stack. This enables the Forth to use the full power of Microsource Assembler.

GET and PUT enable variables to be taken from Basic, placed on the Forth stack used in a Forth program, and passed back to any Basic variable. Care should be exercised in the use of these words to pass negative numbers between the two languages because of the different internal representations of numbers in the two languages. (See 'Forth and Numbers')

The following program will GET two variables from Basic, add them together and pass the sum back to a Basic variable for Basic to print:

```
10  LET number1 = 10 : let number2 = 20 : let result = 0
20  LET forth=1
30  REM # GET number1 GET number2 + PUT result
40  PRINT result
```

The new word COPY works in a similar way but is more flexible as it copies the ADDRESS of a variable and the ADDRESS and LENGTH of a string onto the Forth stack. This enables easy manipulation of strings and complex operations. on Basic variables, whilst they are still in the variables area.

Note that GET PUT and COPY only apply to single variables or single strings,
- DO NOT TRY TO REFERENCE ELEMENTS OF ARRAYS -

TOK allows the user to access some of the symbol tables held in Microsource's ROM and is useful for anyone writing a disassembler or even (cheek!) another assembler. TOK pops the last item off the stack and prints out a character string to the current channel if the number is 114 or less.The tokens printed out, are as follows:

0=LD	1=ADC	2=ADD	3=AND	4=CP	5=OR	6=SUB	7=XOR
8=INC	9=DEC	10=SBC	11=JP	12=JR	13=DJNZ	14=CALL	
15=RST	16=RET	17=POP	18=PUSH	19=BIT	20=RES	21=SET	
22=RLC	23=RRC	24=RL	25=RR	26=SLA	27=SRA	28=SRL	
29=EX	30=IN	31=OUT	32=IM	33=CCF	34=CPL	35=DAA	
36=DI	37=EI	38=EXX	39=HALT	40=NUP	41=RLA	42=RLCA	
43=RRA	44=RRCA	45=SCF	46=CPD	47=CPDR	48=CPI		
49=CPIR	50=IND	51=INDR	52=INI	53=INIR	54=LDD		
55=LDDR	56=LDI	57=LDIR	58=NEG	59=OTDR	60=OTIR		
61=OUTD	62=OUTI	63=RETI	64=RETN	65=RLD	66=RRD		
67=ORG	68=MACRO	69=DEFB	70=DEFW	71=DEFM	72=DEFS		
73=EQU	74=DEFV	75=LIST	76=OPT	77=OBJ	78=ASIF		
79=ENDM	80=END	81=?PC	82=F	83=B	84=C		
85=D	86=E	87=H	88=L	89=(HL)	90=A		
91=AF	92=BC	93=DE	94=HL	95=SP	96=IX		
97=IY	98=(IX	99=(IY	100=NZ	101=Z	102=NC		
103=C	104=PO	105=PE	106=P	107=M	108=(BC)		
109=(DE)	110=(SP)	111=I	112=R	113=AF	114=(C)		

TABLE 1 : STACK MANIPULATION OPERATORS

WORD	**STACK**	**ACTION**
SWAP	(n1 n2 → n2 n1)	Reverses the top two stack items.
DUP	(n → n n)	Duplicates the top stack item.
?DUP	(n → n n) or (n → n)	Duplicates only if **n** is non zero.
DROP	(n →)	Discards the top stack item.
OVER	(n1 n2 → n1 n2 n1)	Makes a copy of the second item and pushes it on top.
ROT	(n1 n2 n3 → n2 n3 n1)	Rotates the third item to the top.
PICK	(n → n)	Picks n'th item off the stack and copies it to top of stack.
RP		Like PICK, but operating on Return Stack. See Return Stack Operators.

KEY Stack Notation:- (before → after); top of stack on right
n, n1, ... 16-bit signed numbers.

TABLE 2 : OUTPUT OPERATORS

WORD	**STACK**	**ACTION**
.	(n →)	Prints the signed 16-bit number.
?	(addr →)	Prints the contents of the address.
U.	(u →)	Prints the unsigned 16-bit number.
H.	(u →)	Prints the unsigned 16-bit number as hex digits.
C.	(u →)	Prints the least significant byte of 16-bit number as hex digits. The most significant byte is discarded.
TYPE	(addr u →)	Types the string of u characters, starting at addr.
." XXX"	(→)	Types out XXX. The character " is the delimiter.
EMIT	(c →)	Prints the ASCII character.
CR	(→)	Performs a carriage return.
SP	(→)	Types out one space.

KEY Stack notation:- (before → after); top of stack on right.
n, n1, ... 16-bit signed numbers c ASCII character value.
u, u1, ... 16-bit unsigned numbers addr. 16-bit address.

Table 3 : ARITHMETIC and LOGICAL STACK OPERATORS

WORD	STACK	ACTION
+	(n1 n2 ⟶ n-sum)	Adds.
-	(n1 n2 ⟶ n-diff)	Subtracts (u1-u2).
1+	(n ⟶ n+1)	Adds one.
1-	(n ⟶ n-1)	Decrements by one.
2*	(n ⟶ n*2)	Multiplies by 2. (arithmetic left shift).
2/	(n ⟶ n/2)	Divides by 2. (arithmetic right shift).
*	(n1 n2 ⟶ n-prod)	Multiplies.
/	(n1 n2 ⟶ n-quot)	Divides (n1/n2).
MOD	(n1 n2 ⟶ n-rem)	Returns module (ie.remainder from division)
/MOD	(u1 u2 ⟶ u-rem u-quot)	Divides. Returns remainder and quotient.
NEG	(n ⟶ -n)	Changes sign.
MAX	(n1 n2 ⟶ n-max)	Returns the maximum.
MIN	(n1 n2 ⟶ n-min)	Returns the minimum.
ABS	(n ⟶ /n/)	Returns the absolute value.
AND	(n1 n2 ⟶ n1 ⊕ n2)	Returns logical AND.
OR	(n1 n2 ⟶ n1 ∧ n2)	Returns logical OR.
XOR	(n1 n2 ⟶ n1 ⁘ n2)	Returns logical XOR.
RR	(n ⟶ n)	Rotate right, bit 0 ⟶ bit 15.
RL	(n ⟶ n)	Rotate left, bit 15 ⟶ bit 0.

KEY Stack notation :- (before ⟶ after); top of stack on right.
n, nl, ... 16-bit signed numbers.
n, u1, ... 16-bit unsigned numbers.

Table 4 : COMPARISON and TESTING OPERATORS

WORD	STACK	ACTION
+	(n1 n2 ⟶ f)	Returns true if n1 = n2.
<>	(n1 n2 ⟶ n-diff)	Returns true (ie.the non-zero difference) if n1 and n2 are not equal.
>	(n1 n2 ⟶ f)	Returns true if n1 > n2.
<	(n1 n2 ⟶ f)	Returns true if n1 < n2.
0<	(n ⟶ f)	Returns true if n is negative.
NOT	(f ⟶ f)	Reverses the result of the previous test.
>=	(n1 n2 ⟶ f)	Returns true if n1 >= n2.
<=	(n1 n2 ⟶ f)	Returns true if n1 <= n2.

KEY Stack notation:- (before → after); top of stack on right.

n, n1, ... 16-bit signed numbers.
f Boolean flag (true = non-zero).

Table 5 : ADDRESS OPERATORS

WORD	**STACK**	**ACTION**
@	(addr → n)	Replaces the address with its 16-bit contents. ("fetch")
!	(n addr →)	Stores a 16-bit number into the address. ("store")
+!	(n addr →)	Adds a 16-bit number to the contents of the address ("plus-store")
C@	(addr → b)	Fetches an 8-bit value from the address.
C!	(b addr →)	Stores an 8-bit value into the address.
XXX	(→ addr)	Finds the address of word XXX in dictionary and places it on stack.
CONST XXX	(n →) XXX: (→ addr)	Creates a constant called XXX with value n; the word XXX returns n when invoked.
VAR XXX	(→) XXX: (→ addr)	Creates a variable called XXX; the word XXX returns its address when invoked.
'XXX	(→ addr)	Finds address of XXX in dictionary.

KEY Stack notation:- (before → after); top of stack on right.

n, n1, ... 16-bit signed numbers
u, u1, ... 16-bit unsigned numbers

8-bit byte (stored as 16 bits; most significant 8 bits set to zero).
addr 16-bit address.

Table 6 : RETURN STACK OPERATORS

WORD	**STACK**	**ACTION**
I	(→ n)	Copies the top of the return stack without affecting it. Used to obtain current loop index.
'I	(→ n)	Copies second item of the return stack without affecting it.Used to obtain limit inside a loop.
J	(→ n)	Copies the third item of the return stack without affecting it.Used to obtain index of next outer loop from within inner (nested) loop.

> R	(n →)	Pops a value off p-stack and pushes it onto r-stack.
R >	(→ n)	Pops a value off return stack and pushes it onto p-stack.
RP	(n → n)	Picks n'th element from return stack and copies it to p-stack. (I=1RP,I'=2RP,J=3RP)

KEY Stack notation: (before → after); top of stack on right.

n, n1 ... 16-bit signed numbers.
f Boolean flag (true = non zero)

Table 7 : CONTROL OPERATORS

The stack operation refers to the parameter stack. All these words must be used inside definitions.

WORD(S)	STACK	ACTION

IF xxx
ELSE yyy
THEN zzz
	IF: (f →)	If f is true execute xxx; otherwise execute yyy; then continue with zzz regardless. The phrase ELSE yyy is optional.

DO ... LOOP	DO: (end start →) LOOP: (→)	Sets up a finite loop, given the index range.
DO ... +LOOP	DO: (end start →) +LOOP: (n →)	Like DO ... LOOP except adds the value of n (instead of always "1") to the index.
BEGIN ... UNTIL	UNTIL (f →)	Sets up an indefinite loop which ends when f is true.
BEGIN xxx WHILE yyy REPEAT	WHILE: (f →)	Sets up an indefinite loop which always executes xxx and also executes yyy if f is true. Ends when f is false.
ABORT		See special operators.

KEY Stack notation:- (before → after); top of stack on right.

end limit of loop
start start of loop (initial value of index)

Table 8 : DICTIONARY OPERATORS

WORD	STACK	ACTION
H	(→ n)	Places address of dictioary and pointer on stack. This is thus a pointer to the 4th byte of the value of variable forsv2.
HERE	(→ h)	Pushes the contents of h onto the stack.
ALLOT	(n →)	Leaves a gap of n bytes in the dictionary.
, ("comma")	(n →)	Compiles n into the next available cell in the dictionary. Increments h by two bytes.

C,	(b →)	Compiles an 8-bit value into the next available byte in the dictionary. Increments h by one byte.
FORGET xxx	(→)	Forgets all definitions back to and including xxx.
EMPTY	(→)	Forgets the entire contents of the user partition.
LAST	(→ addr)	Pushes current pointer to last word onto stack. Necessary when saving/loading dict.

KEY Stack notation:- (before → after); top of stack on right.

n, n1, ...	16-bit signed numbers	addr	address
b	8-bit byte (stored as 16-bit byte with most significant 8-bits set to zero)		

Table 9 : SPECIAL OPERATORS

WORD	STACK	ACTION
MOVE	(from to u →)	Copies a region of memory u bytes long (from, to). Similar to "ldir" on Z-80.
ABORT	(→)	Forces an exit from FORTH (whether in loop or not). Return stack cleared, parameter stack NOT.
#	(n →)	Changes output stream to n.All output will now go to stream n (3=printer etc.) Note that for streams 4---15 the channels must be opened first from Basic.
STACK n	(→)	Moves stack and dictionary compilation area. Value must follow eg.STACK 8000h. This is so that no memory is corrupted should this be the first command.
(	(→)	Start a comment. ')' ends it. A comment must end in the same contiguous block of Forth and cannot be the last text within a definition or a contiguous block.
USR	(n1 n2 → n3)	Call machine code routine. n1 is forced into BC register pair; n2=call address.On return, BC is pushed onto stack. No registers preserved.
GET xxx	(→ n) (Input range) (-32768 to +32767)	xxx = name of single Basic variable. Value of variable pushed onto stack. eg. GET number.
PUT xxx	(u →)	xxx name of single Basic variable. u is stored in value field of variable.Variable must exist or error U generated.

IMPORTANT GET PUT and COPY only apply to single variables or single strings DO NOT TRY TO REFERENCE ELEMENTS OF ARRAYS.

COPY xxx	xxx numeric: (—>n) xxx string: (—>n1 n2)	If xxx numeric, addr of value field pushed onto stack. If xxx a string, then addr(n1) and length pushed. Thus COPY a$ TYPE will print a$.
LINK	(n1 n2—>)	Links in a reloaded dictionary n1 is value of HERE and n2 value of LAST for dictionary being linked.
TOK	(n—>)	Prints taken off stack if n 114 or less. eg. 0 TOK will print out LD. Useful for disassemblers etc.

KEY	Stack notation:-	(before —>after); top of stack on right.
	n, n1, ...	16-bit signed numbers
	u, u1, ...	16-bit unsigned numbers

------[END of SECTION]------

... Debug ...

INTRODUCTION to the DEBUGGING AID

Microsource's DEBUG AID may be used to step through any machine code program held in the Spectrums ROM or RAM. (note that you cant step through the Microsource's ROM or the Interface 1 ROM !!) It is an essential tool for developing a machine code program as machine code does not usually generate error messages when a problem is encountered - it either "hangs" or "crashes" and finding out what went wrong is very difficult without a debug-facility.

The Debug Aid is effectively an emulation of what is going on inside the Z-80 and when first invoked displays a 'front panel' of the Z-80's internal registers and memory. The user can **single-step auto-step** examine and modify memory contents and set up **'trace masks'** to display selected registers during debugging.

DEBUG INVOCATION

Debug is invoked by typing LET debug= -expression- where -expression- evaluates to an integer number which represents the address where you would like to start debugging.

Type in LET debug=0 (and ENTER). You will see the 'front panel' display appear. This is a representation of the state of the Z-80 before it embarks on executing the code at memory location zero.

The display is largely self-explanatory but a few points should be noted.

Where 16 bit registers or register pairs are shown their current value is indicated and the contents of memory at that point is shown as hexadecimal bytes.

The user is now ready to enter debug commands.

DEBUG COMMANDS

GENERAL

Debug commands are entered by pressing the appropriate key, and, in the case of expressions, ENTER as well. Some keys operate as soon as they are pressed. These are denoted in the command list. The command is acknowledged as it is entered in the bottom left hand corner of the screen.

S (SINGLE STEP) Immediate action when pressed

Single step, when initiated, causes the opcode pointed to by the PC to be "executed". All registers and memory contents affected are altered and the display updated.

X (EXIT) Immediate action when pressed

Exit from Debug, back to Basic. Unless Basic's variables or workspace have been tampered with (eg.by single-stepping through the NEW routine!) normal program/line execution will resume.

E (EXCHANGE) Immediate action when pressed

Display the alternate register bank. AF,BC,DE and HL are swapped with their alternate pairs. The front panel display will show 'ALT' if the alternate pairs are the ones currently in use.

ENTER (Display Front Panel) Immediate action when pressed

Displays the front panel. Useful after a memory dump to restore the front panel display.

G (GO) Immediate action when pressed

Repeat single-stepping from the current PC until a breakpoint has been encountered (see B) or BREAK is pressed. Note that the display during the GO command depends on the 'call levels' set with the 'L' command (see below). By default, only the topmost level is displayed.

C (CALL) Immediate action when pressed

The opcode pointed to by PC is examined and unless it is a CALL or RST it is executed as a single-step (see S command). If it is a CALL or RST the entire call is single-stepped, only displaying when finished. This is useful if you dont want to know what happens in a call but only want to know what happens at entry and exit. If a breakpoint is encountered during the call or BREAK is pressed, then Debug will halt in the call and display the front panel showing the condition at that time.

D (MEMORY DUMP) Immediate action when pressed

Memory contents is displayed around the value of M (the Memory Pointer), as hex bytes. When Debug is first invoked, M is set to 0000, but it can be modified (see U below).

$ (ASCII) Immediate action when pressed

GO into memory dump, as above, but bytes are displayed as ASCII characters. A '?' is printed if the byte is not a printable ASCII character.

Q (BACK) Immediate action when pressed

Decrement the memory pointer by 00B0H - useful for stepping through memory - memory dump is modified automatically if in D mode.

A (FORWARD) Immediate action when pressed

Increment the memory printer by 00B0H. Else similar to Q above.

COMMANDS REQUIRING ARGUMENTS

B nnnn (BREAKPOINT)

Set a breakpoint at nnnn, where nnnn is a decimal or hex number. Press ENTER to set the breakpoint. Only one breakpoint can be set, and it will be active until you use the B command again. Both C and G commands cease single-stepping on encountering a breakpoint.

J nnnn (JUMP)

Causes Debug to temporarily suspend and allows the processor to do a direct JP to nnnn, where nnnn is a decimal or hex number. Press ENTER to initiate the jump. If the code "jumped-to" ends with a RET, Debug will be re-entered (but PC will still show the same value as on exit).

This feature allows the user to execute code directly whilst still in Debug.

O nn (OUTPUT)

Change display stream to nn, where nn is a number coresponding to a channel. eg. 03 will direct output to the ZX printer. If the stream is directed to Microdrive/ZX Net/RS232 then these channels must previously have been opened from Basic. Press ENTER to initiate the change.

Note that 00 supresses all display but Debug will still respond to commands eg. 02 will restore the display stream to the screen. 01 directs the display to the bottom half of the screen, resulting in rather a strange display which scrolls up and then disappears, but Debug will still respond to commands as normal.

UPDATE COMMANDS

All of the commands in this class require a pseudo-assignment. In the pseudo-assignment -reg- can be any of:-

A B C D E H L PC SP IX IY BC DE HL

Updates are entered by pressing 'U' followed by the pseudo-assignment,followed by ENTER to perform the update eg. U a = 3Dh ENTER will update the accumulator to contain 3D hex.

-reg- = nnnn

Alter register or register pair eg. U a = 3Dh or U bc = 1234h or U sp = 0001H. nnnn can be a decimal or hexadecimal number. Note that if a 16-bit value is attempted to be assigned to an 8-bit register, the least significant 8 bits will be used eg. U a = 1234h results in the accumulator being loaded with 34h.

nnnn = mm

Alter byte of memory at nnnn to contain mm, where mm is an 8-bit value, eg. U 8000h = 32h.

%nnnn = mmmm

Alter word of memory at nnnn and nnnn+1 to be mmmm (lo, hi). eg. U 8000h = 1234h will place 34h in 8000h and 12h in 8001h.

# -reg- = nnnn

Alters byte at which the register pair points to nnnn. eg. if HL is 8000h then #HL = 5Dh will load 5Dh into location 8000h.

@ -reg- = nnnn

Alters word pointed to by the register pair to nnnn (lo hi). eg.if BC = 5C30h then U @BC = 1234h will place 34h in location 5C30h and 12h in lacation 5C31h.

# nnnn = mmmm

Alters the byte pointed to by the word stored in location nnnn, nnnn+1 to mmmm. eg.If memory at 9000h holds 12h and 9001h holds 34h, then U #9000H = 52h will place 52h in location 3412h.

@ nnnn = mmmm

Alters the word pointed to by the word stored in loaction nnnn, nnnn+1 to mmmm (lo hi). eg.if memory at 7EAFh holds 17h and 7EB0h holds 19h, then U @7EAFh = 3637h will place 37h in location 1917h and 36h in location 1918h.

The last two commands in particular are rather complex and you should try them out to see their effect.

M = nnnn

Alters the memory pointer M (which acts as a pseudo-register) to nnnn. On entry to Debug, M is set to 0000, but U M = 1234h will alter M to 1234h, and pressing D will display a memory dump of the region round 1234h. M is also altered by the Q and A commands detailed in the previous section.

R (repeated alteration to memory)

R (repeated alteration to memory)

When R is pressed Debug waits for a byte to be input and places it at the memory location pointed to by M. eg. R : 12h (ENTER). M is then automatically updated and the R : prompt given again so that the next location can be updated. In this way repeated alteration to an area of memory can be made.

Pressing ENTER when the R : prompt is given leaves R mode and returns to the front panel.

TRACE MASK COMMANDS

Debug allows a trace mask to be set up so that the user can observe the effects of a program on a specific set of registers or memory locations etc.

To set up a mask, enter M followed by a series of pseudo-assignments separated by one or more spaces, specifying what is to be displayed. The pseudo-expressions are the same as in the Update command section, so

```
M: PC #PC HL A %8000h
```

will set up a mask to print:

PC - The contents of PC.

#PC - The byte PC points to (ie.the first byte of the next instruction)

HL - The contents of HL.

A - The contents of the Accumulator.

%8000H - The contents of location 8000h, 8001h
(Note: all word prints are in order hi, lo)

The M command automatically turns the mask display on, and lists a first line showing the state of the register etc. specified.

Subsequent S and G commands will now cause new mask lines to be printed. C and J commands can also be used.

Masks are useful because you can step through a program much faster than with the front panel, and information which you do not require is suppressed. To leave mask mode, simply press ENTER to return to the front panel.

K (RETURN TO MASK)

K will return the user to mask mode from the front panel display. If a mask has not yet been defined then blank lines will be printed, otherwise the first line of the mask is printed as before.

CALL SUPPRESSION

In complex machine code routines, calls to nested sub-routines may be tedious to step through using S or G commands. The C command allows you to skip through a call and the J command to allow the Z-80 to execute it directly,but these commands actually require you to spot that a call is about to be made. The H and L commands allow nested calls to be processed more effectively.

H nn

Single-step, not displaying until nn levels of returns have been encountered (nn max 255). eg.H 1 (ENTER) will single step (not displaying) to the end of a routine and stop after the RET. This command is very useful for emulation of message printing.

L m, n

The L command sets 'upper and lower' call levels specified by m and n (upper, lower). Initially m and n default the zero. These control the display during the G command only. Thus L 0, 0(the default) means that a trace or display is only displayed at the top-most level. L 0, 1 would display at the top level and at one level of call; L 2, 2 would display at call level 2 only.

The L command, when used in conjunction with a trace mask,the G command and a breakpoint, is a very powerful debugging tool.

HOW DEBUG WORKS

Debug is a virtual machine that is, it is a piece of software which simulates the action of the Z-80 processor. To do this, it basically saves the status of the processor somewhere on the stack, extracts the opcode from memory and allows the Z-80 to execute it. The Z-80 is then halted and the new status displayed. This allows a constant display of what the Z-80 is doing but the Z-80 actually executes the code, so you have to be careful !!

A classic example is a machine code program which zeroes the memory (an example of this is to let Debug GO from 0000). Since the Z-80 actually does this, the memory, including the stack, will be wiped, resulting in Debug crashing. Similarly the execution of stack manipulation instructions will cause problems although Debug does its best to anticipate these and only the more complex stack instructions will cause it to crash.

An important point to note if you desire to return to Basic after using Debug is that if the Basic system variables area has been corrupted the computer may crash on exiting from Debug. A more frequent problem may possibly be the appearance of spurious error reports on exiting.

It is therefore GOOD PRACTICE to use the B command to set a breakpoint where possible to prevent Debug executing too much code.

QUADHOUSE COMPUTERS (UK)

REGENT HOUSE
VICTORIA ROAD
MIDDLESBROUGH

CLEVELAND
(0642) 221102