USER MANUAL

Gremlin

Machine Language Monitor & Debugging ROM for the BBC Micro

Computer Concepts

Serial No. 003584

Please Note

IT IS VITAL THAT THE REGISTRATION CARD SUPPLIED WITH GREMLIN IS RETURNED TO US, WITH YOUR NAME AND ADDRESS FILLED IN. THE CARD IS POSTAGE PAID FOR THE U.K. IF FOR ANY REASON A REGISTRATION CARD IS NOT SUPPLIED, YOU MUST CONTACT THE DEALER FROM WHOM THE PACKAGE WAS PURCHASED. THE SERIAL NUMBER ON THE REGISTRATION CARD SHOULD BE PRINTED INSIDE THE MANUAL. YOU MUST QUOTE YOUR SERIAL NUMBER IN ANY CORRESPONDENCE WITH REGARD TO DISC DOCTOR. RETURN OF THE CARD IS FOR YOUR OWN BENEFIT.

DUE TO INCREASING SOFTWARE PIRACY, A REWARD OF £100-£500 IS OFFERED TO ANYONE PROVIDING INFORMATION LEADING TO A SUCCESSFUL LEGAL SETTLEMENT AGAINST ANY DEALER, SCHOOL, INDIVIDUAL, ETC. MAKING COPIES OF THIS OR ANY OTHER COMPUTER CONCEPTS SOFTWARE PACKAGE.

Computer Concepts accept no responsibility for loss of data, programs or time, due to the use of GREMLIN.

1 INTRODUCTION

The GREMLIN ROM is a very powerful tool for examining and helping to debug machine-code programs. Being ROM-based it occupies none of the user's valuable program space, allowing a program being debugged to sit virtually anywhere in memory. It has all the facilities expected of a debugging tool, as well as some rather unusual and highly advanced features, such as the full expression evaluator and the two-pass assembler.

The first few chapters in this manual are quite brief in their explanation of the commands, but section 11 describes a session with GREMLIN which should serve to clarify most points. Section 12 also gives a summary of commands, system variables and switches, plus the syntax of the expression evaluator, which will be useful when you have read the rest of the manual. It is worth pointing out that, regardless of how impatient the user may be, trying to use GREMLIN without first reading the manual is quite pointless, not to mention frustrating.

TECHNICAL ADVICE

Technical problems should be sent in letter form to Computer Concepts. Again, please read the manual first. Letters asking for answers given clearly in the manual are likely to be given little if any attention. All correspondence must state name, address, and most important of all - the registration serial number which is printed on the inside cover of this manual. In order to hasten your reply, please enclose a self-addressed adhesive label, and give details of any non-standard aspects of your machine - e.g. what other ROMs are plugged in.

1.1 GETTING STARTED

Follow the fitting instructions supplied with GREMLIN and before screwing the lid back on, ensure that the ROM is working in the machine. To do this type the command:

*HELP GREMLIN <RETURN>

If the answer on the screen is simply "OS 1.20" then GREMLIN is NOT being recognised, so read the fitting instructions again to ensure that you have done everything correctly. If you cannot get GREMLIN working, then report it to its place of purchase. If however, the answer given is the name "GREMLIN x.xx" (where x.xx is the version) followed by a list of commands etc. then all is well. When the lid is replaced on the machine and you are ready to use GREMLIN, type the command:

*GREMLIN <RETURN>

to enter GREMLIN. The above command may be reduced to a minimum abbreviation as is usual for ROMs.

On entry, GREMLIN will clear the screen and show the STATUS DISPLAY at the top of the screen. The status is a central feature in the GREMLIN, giving the user all the general information about the processor. Below a dividing line the title GREMLIN is shown, followed by an exclamation-mark used throughout as a prompt to show that it is waiting for the next command. In mode 5 or mode 2 only the title and prompt appear,because the status needs more than a 20 column screen to make sense. GREMLIN has the command MODE which can be used just as in BASIC to change the screen mode to anything else, to save having to go into BASIC and back just to change the mode.

The STATUS DISPLAY

Printed below is a typical status display shown on entry to GREMLIN. The top line shows the contents of the A,X and Y registers, plus the current flag settings. Below these are the contents of the Stack and Program Counter registers which are two-byte pointers; to their right is an area of memory pointed to by each register respectively. On the line which is mainly blank is shown a disassembly of the next instruction, pointed to by PC. The example shows three question marks "???" indicating that the instruction is unrecognised, i.e. PC is not actually pointing to the start of a valid instruction at present.

```
  A=FF    X=FF    Y=FF
S =01FF 89 10 E3 DA 92 93 DC 89 ........
PC=FFFF DC FF FF 02 1B 00 7C 00 ......|.
        ???
   06F0 00 00 00 00 00 00 00 00 ........
   06F8 00 00 00 00 00 00 00 00 ........
M =0700 A9 01 85 06 A9 00 85 07 ........
   0708 4C 11 07 A9 FF 85 06 85 L.......
   0710 07 20 92 07 A0 00 A5 04 . ......
   0718 05 05 D0 04 20 92 07 60 .... ..#
----------------------------------------
GREMLIN
!
```

Most of the status display (from the disassembled instruction downward) is taken up with display of an area of memory. A GREMLIN 'system variable' named 'M' is the memory pointer, and defines the area of memory to be displayed. All values shown are in Hexadecimal. Each line is made up as follows: two-byte address for the first byte of that line, a number of bytes shown as pairs of Hex digits, then the same bytes shown as ASCII characters. Any byte which cannot be displayed as an ASCII character is instead shown as a dot.

In 80-column modes there are sixteen bytes of memory shown across the width; in 40-column modes there are eight bytes across, as shown, but in 20-column modes the status display is not shown at all. The entire status display may be either disabled so that it is not updated, or omitted altogether.

When debugging programs producing screen output (e.g. graphics) which must be examined, the status display would be omitted entirely, and an abbreviated list of registers and flags can be sent to the printer instead. When, say, assembling a program from file, the status would be updated after every instruction and assembly would consequently be slow. Whilst assembling long programs it is advisable to disable the display's automatic update. Internal 'switches' or 'flags' are provided to allow this control over the status display. Switches are discussed later in the manual.

2 ALTERING MEMORY CONTENTS

This section deals primarily with editing memory. The topics discussed are moving the memory pointer, editing memory, searching, and moving areas of memory.

2.1 Moving around in memory.

The first step toward editing memory is learning how to use the status display to examine different areas of memory. The system variable 'M' is used to point the display to a particular memory region. System variables will be discussed later, but it is necessary to know at this stage that all system variables are prefixed by the dollar symbol '$' in order to reference them. Therefore the system variable 'M' will be referred to as $M. Four commands are provided to move the memory pointer ($M) and so look at different areas of memory. Practise gained in use of these commands now will save time later.

Note that all command words or letters should start immediately after the exclamation-mark prompt. The reason for this is explained later, but if this simple rule is ignored, the command will not be performed as expected and a 'No such variable' error will probably occur.

M <expr>

Sets $M directly. The full details of an expression <expr> are explained in great depth in section-6; for the moment, suffice it to say that some obvious operations are allowed and numbers are dealt with in Hex format by default. Therefore the command:

$M=1900 <RETURN> will cause the area of memory around 1900 Hex to be shown in the status display.

+ <expr>

Adds the result of the specified expression to $M. Therefore, assuming $M still points to 1900 from the previous example, the command:

+100 <RETURN> will add 100 Hex to $M, resulting in $M containing 1A00 Hex and the area around that address being displayed.

- <expr>

Subtracts the result of the specified expression from $M. This is similar to the above command and shouldn't need an example.

Function keys may be programmed as normal while using GREMLIN, so you may find it useful to try some definitions such as "+8", "-8" etc. while becoming proficient in using $M.

IND

This command indirects $M through itself. It is useful for following vectors and linked lists. For instance, if $M contains 1A00 Hex it is said to be 'pointing' at memory location 1A00. If locations 1A00 and 1A01 contain 00 and 01 respectively, then as a pair they are pointing at location 0100 Hex. Therefore if we give the command IND (which has no parameters) then $M is indirected through itself and ends up containing (i.e. pointing to) &100. This command may take a little more thought than the previous ones for users who are unfamiliar with memory pointers. Remember that pointers in memory hold the address in the usual convention of low-byte, high-byte, so 0100 Hex is stored as 00 followed by 01.

2.2 Punching data into memory.

The powerful 'P' command allows almost any data to be 'Punched' into memory. After putting data into memory with the P command, $M is automatically incremented to point to the next location after the inserted data.

P <data>
Punches <data> into memory at the current location pointed to by $M. The <data> following the P command may be any of the following:

<expr> [,]
Punch BYTE. The result of the expression will be treated as a single byte and $M incremented by 1. Since this is the default use of the P command the comma is optional.

<expr> ;
Punch WORD. The expression result will be treated as a double-byte word and $M incremented by 2. The use of a comma to follow a byte and semi-colon to follow a word is the same convention used by BBC BASIC's VDU statement.

<"string">
Punch STRING. The given string must be enclosed within double-quote marks. The string may be of any length and is evaluated by the O.S. routine used to evaluate *KEY statements. This means that the same syntax applies (if you are unfamiliar, see the Users Guide).

<assembler code>
Punch ASSEMBLY STATEMENT. One of GREMLIN's strongest points is its assembler, which is a FULL implementation. Mnemonics must be typed in capitals, expressions can be used and all addressing modes, etc, are supported. The assembler is dealt with more fully in Section 8. The assembler code needs no terminator; this does not usually cause ambiguity. One point to note, do not type BNE$M+3 or a similar reference to $M in the "P" command, because it is unlikely to function as expected. All other numeric operands should assemble as expected.

2.3 Searching for data

F <data>
Searches forward from $M to find the specified data. The argument given to the command as <data> follows the same rules described above for the P command. $M is set to the start of the data found. If the data is not present in memory then it will always be found in GREMLIN's own buffer area where it is stored for comparison with the rest of memory. Do not be suprised either if the search string is found in the keyboard buffer, or in the screen memory.

2.4 Moving data

IM <expr1> <expr2> <expr3>
INTELLIGENT MOVE. Moves data from <expr1> to <expr2>, transferring <expr3> bytes, e.g. to move Hex 100 bytes of memory from 800 to 2000 Hex, type

IM 800 2000 100 <RETURN>

Memory may be moved up or down and the destination may overlap the source. $M is not affected by the IM command.

3 DISPLAYING MEMORY

In addition to the status display, memory contents may be shown in two further ways: a straightforward tabulation in Hex and standard ASCII (as used in the status) or a disassembly. The disassembler is extremely versatile, with numerous options including production of a SPOOL file which may be later reassembled (possibly at a different location) by either BASIC or GREMLIN, depending on another option for output format.

3.1 Tabulation

T <expr1> <expr2>
Tabulates memory from <expr1> to <expr2> inclusive, in the same format as that used in the memory display. ESCAPE may be used to exit at any time during a tabulation.

3.2 Disassembly

D <expr1> <expr2> (S) (F"file"(title))
Disassemble from <expr1> to <expr2> inclusive. ESCAPE may be used to exit if necessary. Further arguments are optional, a description of each is given below, together with some relevant switches:

OPTION

S - DISPLAY SOURCE CODE.The source code is shown, comprising instruction address; the instruction as:- up to three bytes in Hex; the same bytes in ASCII; followed by the instruction mnemonics as usual. Any unknown instruction is disassembled as "???". This option must be omitted if files are to be produced and used to re-assemble the code.

F "file"(title) - SEND OUTPUT TO FILE. The title may be something like AUTO 100,1 so that BASIC-compatible files may be easily reassembled by *EXECing the file. GREMLIN can also reassemble the output.

The following options are controlled by switches, which control various options within GREMLIN. They are described more fully in section-5, but they are used simply by typing the command 'SW' followed by the two-letter switch name, followed finally by a '1' to turn the switch on, or a '0' to turn the switch off, e.g. SW DR 1 would turn on the Disassemble Relative option.

DR - DISASSEMBLE RELATIVE SWITCH (Default=0) - causes a relative disassembly format to be employed, wherein branch instructions are disassembled not as

BNE 7800

for instance, but as

BNE $M-5

This enables spooled files to be reassembled at a location different to the original.

BF - BASIC FORMAT (Default=0) - causes BASIC format to be employed. This means that relative output will refer to P% rather than $M, and Hex numbers will be preceded by an ampersand "&". This overcomes the differences between assembling in BASIC and GREMLIN.

HX - HEX NUMBER FORMAT (Default=1) - controls the number base. Setting to 0 causes decimal output - and also requires decimal input. See Section 5 for more details on numeric format and the use of switches in general.

4 EXECUTING CODE

Once code is in the memory, it can be executed in a variety of ways:

1) SINGLE STEPPING. It can be executed one instruction at a time, allowing observation of each instruction's effect on registers and memory.
2) WITH BREAKPOINTS. A number of breakpoints may be set such that, when reached, cause execution to be temporarily suspended while registers and memory are examined. Execution may then proceed to the next breakpoint.
3) CALL. An entire routine may be executed without pause by CALLing the start address. This would mainly be used for calling debugged subroutines, O.S. routines etc.

ENABLING EXECUTION

Executing sections of code at random is likely to accidentally corrupt valuable areas of memory. Misuse of the 'single-stepping' and 'execution with breakpoints' commands is therfore hazardous. To reduce the risk of invoking the commands (which are single letters and therefore easily mis-typed) a special software 'switch' is provided to enable or disable their use. Named the JE switch (meaning Jump Enable) it initially defaults to the disable setting to prevent use of the S and J commands. Attempting to use either command will result in the message "Not enabled" until the switch setting is changed. The JE switch is set to enable or disable by the following commands:

```
SW JE 1  <RETURN>  ....Enable execution instructions.
SW JE 0  <RETURN>  ....Disable execution instructions.
```

4.1 Single-stepping.

To single-step a program it is necessary to first set the program counter (using the system variable $PC) to the address of the first instruction. e.g. if the start address is 2000 Hex, then use the instruction:

```
$PC=2000     <RETURN>
```

To start the single-stepping type the single letter command:

```
S     <RETURN>
```

Typing S initiates single-stepping mode, executes the first instruction and updates the status display. Pressing <RETURN> will cause a further instruction to be executed. Subsequent presses of <RETURN> cause individual single-steps to be executed. Pressing any other key will abort the single-step mode and the exclamation mark prompt will reappear to show that another command may be entered.

Each time a single step is performed, the instruction shown on the status line is executed, the registers and flags are updated, the next instruction is displayed and GREMLIN awaits either <RETURN> to execute the next instruction or any other key to exit back to command mode.

Continued....

Whilst stepping through a program it would prove useful to treat an entire subroutine as a single step. For instance, stepping through the O.S. OSWRCH routine each time a character is printed would accomplish nothing (since it is presumably debugged), it would be very tedious and waste a great deal of time. GREMLIN provides a method of coping with this problem by single-stepping ONLY within a specified area, called the debugging area. Any calls to routines outside the debugging area are executed as a whole in one step. For instance entering the following short program should serve to demonstrate, but first, enable single-stepping and set the memory pointer to a reasonable postion with the commands:

```
SW JE 1     <RETURN>
M2000     <RETURN>
```

Then enter the short program as follows:

```
LDA #"A"
JSR FFEE
$PC=$M-5
S <RETURN> <RETURN> <SPACE-BAR>
```

The two single-steps performed at the end show the accumulator first being loaded with the letter "A" (41 Hex), followed by a call to an O.S. subroutine to print a character - OSWRCH (FFEE Hex) and the letter A appears on the screen. The point to notice is that the subroutine which consists of many instructions is executed in one step. If you wish to see the difference this makes then enter the command:

```
$DH=OFFFF
```

The debugging area is defined by the contents of system variables named $DL and $DH (Debug Low and Debug High). Initially $DL=0 and $DH=8000, causing subroutines in the area 0..8000 to be "expanded" into their individual steps, whereas ROM routines are treated as a single unit. Altering the debugging area is accomplished by altering $DL and $DH. For instance if a routine between 5800 and 5900 is being debugged and any calls made outside that area are to previously debugged subroutines, we might use

```
$DL=5800     <RETURN>
$DH=5980     <RETURN>
```

The time saved by using this feature is well worth the trouble of setting $DL and $DH before starting to single-step.

Single-stepping through ROMS

Not only is it possible to step through the Operating System or GREMLIN; any sideways ROM may be selected and stepped through. For instance, if BASIC is in socket 3, typing

```
$ROM==3     <RETURN>
```

will enable you to step through any part of BASIC. (The "==" operator must be used since $ROM is a single byte variable - see later). The memory display allows inspection of any area of the ROM; the IM command will move ROM code down into RAM where it may be disassembled etc. Some operations are not possible with ROMs and are described later.

Continued....

The single-step mechanism is implemented by moving the instruction down into low memory, and using BRK handling routines. This imposes limitations on the sort of program that may be single-stepped. Any BRK instructions in a program should not be stepped through; simply set the program counter to the following instruction. Subroutines which rely on artificial instructions, sometimes called self-altering code, cannot be stepped through. Certain parts of the service sections of sideways ROMs may cause problems too. A sideways ROM header will usually start by disabling interrupts... executing this instruction will effectively disable single-stepping and is NOT advisable.

4.2 Continuous execution and breakpoints.

J <addr>

The J command causes execution of code at location <addr>. It, too, must be enabled by the JE switch. When a 'BREAKPOINT' is encountered a warning 'beep' will sound and the status display is updated. At this point the user may either press <RETURN> to continue execution to the next breakpoint, or press any other key to return to command mode and issue commands.

There are up to eight breakpoints, $B0..$B7, which are set by the user. Several commands are provided to handle these breakpoints:

CB - Clear Breakpoints. All breakpoint positions are set to FFFF so that they are effectively unused.

LB - List Breakpoints. Addresses of all breakpoints are listed.

Setting Breakpoints - No instructions exist specifically to set breakpoints, since they are in system variables which may be assigned in the normal way using the expression evaluator,

e.g. $B0=1234

would set breakpoint $B0 to location 1234. There are two limitations with breakpoints: firstly they must be set to the beginning of an instruction, since they are implemented by "ghosting" BRK instructions into the code - otherwise they will be interpreted as data; secondly they may not reside in ROM or even sideways RAM - though code being executed can of course call O.S. routines as normal.

C - Continue. After a breakpoint has occurred (or at any other time) and the user has chosen not to continue immediately, the C command allows continuation from the current value in the Program Counter. Its use is therefore directly equivalent to the command: J $PC <RETURN> but is much more convenient to type and to remember. The C command must be enabled by the JE switch.

4.3 Direct execution

CALL <addr> - CALL subroutine at address <addr>. CALL will not call into sideways ROMs other than GREMLIN or the O.S.

4.4 More on ROMs

Due to the complexity of implementation, sideways ROMs may only be used in a limited number of contexts. These are:

Single-stepping through ANY ROM.
The IM command.
The status display, T and D commands.
The P and F commands and the assembler (for writing into sideways RAM)

However, because GREMLIN is itself a sideways ROM it cannot respond to:

CALL, J or C commands.
Use of * in expression evaluator (except in the case of a DESTINATION in sideways RAM on certain systems).

5 SYSTEM VARIABLES AND SWITCHES

5.1 System variables

These may be divided into the following six groups.

5.1.1 Memory pointer $M

This is used to control the status display, the assembler, data-punching and searching functions, as described in the preceding sections.

5.1.2 Processor registers $PC, $A, $X, $Y, $S, $P

All these variables are BYTE except $PC. This means they must be assigned to using "==", and retrieved using ">" (see section 6). However, $X assigned to as a word register (using = instead of ==) corresponds to YX as a register pair. Y and X are often used in a pair, especially for O.S. routines and assigning both together is much easier, especially when assigning from another variable, e.g. $X=$PC.

5.1.3 Program Debugging area $DL, $DH.

These two delimit the debugging area, being respectively the lower and upper boundaries. The upper boundary is one byte higher than the last byte in the debugging area (cf PAGE and HIMEM in BASIC). $DL defaults to zero; $DH to 8000 Hex. See section 4.1 for details of use.

5.1.4 Variable storage area $VL, $VH.

These delimit the area allocated to variable storage. Note they do NOT show the current extent of the variable table, but the maximum possible extent. Initially only one page is allocated to variable storage, in the language ROM workspace area: 600..700 Hex. This will not be enough for a large assembly program, and the table may be moved by typing the series of commands as follows:

```
$A==83            (OSBYTE call to)
CALL FFF4           (find OSHWM)
$VL=$X
$A==84           (and similarly the)
CALL FFF4      (bottom of display RAM)
$VH=$X-1000
CLEAR
```

This sequence sets the variable table from "PAGE" to "HIMEM-&1000" (using BASIC parlance). Note the use of the $A register and the $X register pair. The CLEAR statement is necessary to set the internal pointer to the actual top of the variable table to $VL - if this is not done the machine may crash when you next try to define a variable, rather like changing PAGE in BASIC and not typing NEW.

5.1.5 Breakpoints $B0..$B7

Used only by the J and C commands, $B0..$B7 are not initialized by GREMLIN. They may be individually set and printed or alternatively the CB command clears them all and the LB command lists them.

5.1.6 Sideways ROM handling: $ROM

The restrictions which apply to GREMLIN commands in relation to debugging and examining ROMs have been described in section-4. On entry to GREMLIN, examining memory above 8000 Hex (Sideways ROM area) will reveal GREMLIN itself. Ordinarily it would only be possible for a ROM to examine itself, i.e. the currently selected sideways ROM. The $ROM system variable in GREMLIN allows user-selection of any resident ROM to be examined. For instance, if the BASIC ROM were resident in socket number 15 (&OF Hex) then after entering the command:

```
$ROM=&OF      <RETURN>
```

the BASIC ROM could be disassembled, single-stepped, etc. Changing $ROM to another value will instantly select that ROM for examination by GREMLIN.

It may be useful to set the memory pointer ($M) to &8000 and then select different values of $ROM. Each time a new value is selected, the header of the ROM in that socket is displayed, so that all socket contents can be catalogued without even opening the machine.

5.2 Switches

Switches basically provide the user with a choice between two options (just as a light switch provides the choice of having a light ON or OFF). Some switches have already been encountered in previous sections; some switches will be described later. The purpose of this section is simply to describe the SW statement and give a summary of all switches.

5.2.1 The switch command

SYNTAX : SW (<switchlist> 0/1)*

The syntax of the SW command may look complicated, but is quite straightforward in reality. One or more switches may be set in a SW command, so the name(s) of the switch(es) are refered to in the syntax as the <switchlist>, which may be any series of valid switch names. The switches listed in the <switchlist> are all set to the value '1' or '0', i.e. ON or OFF respectively, according to the following digit given specifically as a 1 or 0 character (and NOT a variable). Another list may follow the first, allowing one list of switches to be set 'ON' and another list of switches 'OFF', all in a single command. For instance, to set the JE switch to 1 (i.e. ON) and reset the DR and BF switches (i.e. turn them OFF), the command would be:

```
SW JE 1 DR BF 0      <RETURN>
```

It is important to note that switches may not take the value of variables, but must be explicitly set to the value '1' or '0'.
The eight valid switch names are listed below, together with their default settings

SWITCH NAME		DEFAULT SETTING
HX	...	1
HN	...	1
SE	...	1
AO	...	0
SE	...	0
DR	...	0
BF	...	0
JE	...	0

Resetting all the switches to their defaults would be achieved with the command:

```
SW HX HN SE 1  AO SE DR BF JE 0      <RETURN>
```

Note that it is perfectly acceptable to set just one switch with the SW command, e.g. to disable update of the status display use the command:

```
SW SE 0      <RETURN>
```

Summary of switches

Presented below is a summary of all eight switches and their uses.

5.2.2 Number format :

HX - Use Hex numbers
HN - expected Hex input format
BF - BASIC Format

Format of all numbers input or output is controlled by these switches.
HX sets the number base:
HX=1 sets Hex input/output,
HX=0 sets decimal.
With either setting of HX the "&" prefix may be used to force the number to be understood as Hexadecimal (similar to BASIC). The two other flags HN and BF are only relevant when HX=1.
BF governs ONLY output, causing the '&' symbol (representing Hex) to be printed as a prefix to all Hex numbers. This could be simply for user preference, though it is designed to allow output from the disassembler to be reassembled by BASIC, in which the '&' symbol is a necessary part of the syntax.
HN governs both output and input. It causes GREMLIN to expect subsequent numbers input to be in Hexadecimal if they start with 0-9 or A-F. Therefore the command PRINT A would print 'A' as the Hex number, rather than assuming it to be the variable A. Setting HN 'off' will cause the opposite to be assumed, i.e. all Hex numbers must start with one of the characters (0-9), so that inputting the number &FF as just FF would cause it to be interpreted as a variable named "FF", whereas it should be input as OFF. Alternatively, use the '&' symbol to force Hex input regardless of the setting of HX or HN, i.e. input the number as &FF. Assuming Hex numbers to start with the digits 0-9 is inconvenient for entering lots of Hex numbers, but is far more convenient wheninputing a mixture of variables and numbers.

Examples: inputting the numbers 255 and 127 decimal (&FF and &7F Hex) :-

HX	BF	HN			
0	.	.	255	127	
1	1	.	&FF	&7F	(output only)
1	0	1	FF	7F	(default setting)
1	0	0	OFF	7F	
.	.	.	&FF	&7F	(input forced with '&')

In summary HN and BF affect numbers as follows:
HN=0: leading character = "0".."9"
HN=1: leading character = "0".."F" ("0".."9" for decimal)
BF=0: standard printing
BF=1: BASIC-format printing using "&" for Hex.

5.2.3 Disassembler effects :

DR - Disassemble Relative
BF - BASIC Format

DR controls the disassembly of branch instructions into either absolute or relative format. For example, while normally we may have :-

```
        701E D0 EO    ..  BNE 000
```

setting DR=1 would give :-

```
        701E D0 EO    ..  BNE $M-1E
```

BF flag has an additional effect in this respect; it causes P% to be used instead of $M, so setting BF=1 would give :-

```
        701E D0 EO    ..  BNE P%-&1E
```

So that the disassembled branch instruction is compatible for re-assembly in BASIC.

In summary:

DR 0: absolute disassembly
DR 1: relative disassembly
BF 0: normal output
BF 1: relative branches refer to P%.

Try finding a branch instruction in the MOS, setting P% to point to it, and watching the disassembly on the status display change as you alter these flags.

5.2.4 Status display :

SE - Status Enable
PF - Printer Flag (Send status to printer only)

SE enables or disables the status display. When enabled, the status is automatically updated after each command. The actual updating takes time and can therefore be an annoyance whilst *EXECing an assembly file for instance. Disabling it does not immediately remove it from the screen, rather it is simply not updated, providing a large increase in speed of assembly from file, amongst other things. Disabling the status and pressing CTRL-Z will restore the full screen for use, rather than just the lower half.
PF controls printer output, and is explained fully in section 7.

In summary :-

PF 0 Printer under ctrl-B control
PF 1 Commands and special status to printer
SE 0 No status
SE 1 Enable status.

5.2.5 Two Pass Assembly :

AO - Assembler Option.

AO allows the assembler to be used in a two-pass mode by EXECing a file, and is explained fully in section 8. In short, it determines whether the current value of the location pointer $M (equiv. P% in BASIC) is assigned to undefined variables (forward references to labels...first pass) or whether an error message is generated (...pass two) as the alternative.

In summary :-

AO 0 enables normal "No such variable" error
AO 1 assignes $M to undeclared variables.

5.2.6 Enabling Jump commands :

JE - Jump Enable

JE enables use of commands which may cause a machine crash/data corruption etc. if used inadvertently. The commands enabled/disabled are :

J - Jump to address and execute with breakpoints;
C - Continue execution after breakpoint exit;
S - Single-step execution.

Initially these commands are disabled (JE=0) so that an attempt to use any of the above will result in a "Not enabled" error message.

In summary :-

JE=0 disables use of S, J, C Commands.
JE=1 enables use of S, J, C Commands.

6 THE EXPRESSION EVALUATOR

GREMLIN's expression evaluator is based on that of the language called "C". As such it combines economy of expression with power in use, and has extreme flexibility. To the BASIC programmer GREMLIN's many features may seem rather unusual and take some getting used to, but at the simplest level it may be used just like BASIC's with only a few exceptions.

Since GREMLIN is primarily a debugging tool, floating point mathematics etc. would be a little too excessive, so number handling is limited to two-byte unsigned integers. Bytes are treated in a special way, and string manipulation is minimal being limited to single-character usage. No comparison operators are provided, since they have proved unnecessary in use.

Trying to use GREMLIN's expression evaluator without reading the details of this section is not recommended. Read it through once so that you know what to expect. If you are impatient to try everything then you are likely to obtain seemingly unpredictable results. You have been warned !
The final arbiter of questions concerning the expression evaluator is the syntax specification in section-11. The syntax description may look rather incomprehensible, but at least all the operators are listed and explained there.

PRINT <expr> (,<expr>) ...
The PRINT command will display the values of one or more expressions, similar to the statement of the same name in BASIC. (It does not have the string printing capabilities of BASIC). Where a list of expressions is given, the expressions are separated by commas. The PRINT command will be used widely in this chapter for illustration.

6.1 Operators: + - * / % & ¦ † << >>

An expression consists of TERMS separated by OPERATORS (referred to as 'ops'). The definition of a 'term' will be given later but, for the present, suffice to say that variables, numbers, etc, are all classed as 'terms'. There are 10 ops (operators) as listed above. Unlike most expression evaluators, GREMLIN's has no operator precedence, so the command :-

```
PRINT 1+2*3
```

for instance, will yield 9 as the result (i.e. 1+2, all multiplied by 3). In BASIC this statement would result in 7, since the multiplication has a higher precidence and would be performed before the addition.

Summary of Operators

a+b	Add a and b.
a-b	Subtract b from a.
a*b	Multiply a and b.
a/b	Divide a by b. Then take integer result.
a%b	Take MOD or 'remainder', such that a/b*b+(a%b) equals a.
a&b	Perform bit-wise a AND b.
a¦b	Perform bit-wise a OR b.
a†b	Perform bit-wise a EOR b ('Exclusive OR').
a<<b	Shift a left by b bit positions.
a>>b	Shift a right by b bit positions.

All of these operators except << and >> are present in BASIC (though with different names/symbols). Their effects should be sufficiently obvious to avoid lengthy explanations.

6.2 Terms

The terms of an expression may be anything from a simple number to a whole sub-expression in brackets. Before discussing these, a more detailed look at variables is necessary.

6.2.1 Variables

Variable names must start with a letter and continue with alphanumeric characters. They may be upto 254 characters long and all characters are significant. Upper and lower-case characters are distinguished (as in BBC BASIC).

Variables must be DECLARED before they may be used (undeclared variables are reported with a "No such variable" message). The "." command is used to define variables, e.g.

```
.VAR1
```

declares VAR1 and assigns to it the current value of $M. VAR1 may be initialized to another value if desired ($M is used to support the use of variables as labels in assembly code), e.g.

```
.VAR1=1234
```

sets VAR1 to the value 1234. Now typing :

```
PRINT VAR1 * 2
```

(spaces optional) or even :

```
PRINT VAR1<<1
```

will yield 2468 as expected (remember from the summary of ops that '<<n' means logically shift-left by n bit positions, i.e. <<1 is equivalent to multiplying by two).

Declaring a variable already in existence is legal: the variable is simply reassigned the current value of $M (or the expression on the right-hand side of the "=" if present).

Associated Commands :

CLEAR
Deletes all variables by setting the internal top-of-variable-table pointer to $VL (n.b. $VL is the low pointer of the variable table).

LVAR
Lists all current variables and their values.

6.2.2 Numbers

A term may be simply a number, rather than an expression. The format of numbers depends on switch settings as explained in section 5. In addition to this, a character may be given between quotes, in fact an entire string of characters may be given but only the first character will be used, resulting in its ASCII value. As with strings used in the "P" and "F" commands, the full *KEY syntax is available.

6.2.3 unary operators: + - ! < >

Like BASIC, GREMLIN provides a number of unary operators. In GREMLIN, they are all implemented as symbols, as listed below :-

```
+    ...  Has no effect. Used in the assembler to
          avoid confusion in certain circumstances.
-    ...  Performs two's complement, or negation.
!    ...  Performs one's complement, or NOT operation.
<    ...  Takes high byte
>    ...  Takes low byte - these two are useful
          with the assembler.
```

For instance, typing

```
.X=2052     or     .X=&804
PRINT <X
```

gives :-

```
8
```

which you could also have obtained with

```
PRINT X>>8        (shift right one byte)
```

or

```
PRINT X/256       (X MOD 256)
```

In summary :-

```
X*256/256  equals  X<<8>>8  equals  >X
```

etc.

The "-" unary operator will never yield a "negative" result but will perform a correct two's complement operation, for instance :-

```
PRINT -1
```

gives

```
65535    (or FFFF Hex).
```

6.2.4 Null operator : @

The @ operator is simply a means of accessing one of GREMLIN's internal calculation registers. It returns the value of the last expression evaluated. It saves repeating an immediately previous expression and is useful in commands like "D" and "T" where :

```
D  *(*OFFFE+1)  *(*OFFFE+1)+0A0
```

(which disassembles the first 160 decimal bytes of the IRQ routine), for instance, may be replaced by

```
D  *(*OFFFE+1)  @+&A0
```

Apart from this, the value of @ is unpredictable.

6.2.5 Assignment operators (1): = ==

Assignment (setting a variable to a value) is the first area in which GREMLIN departs radically from BASIC. GREMLIN does not have conditional statements and does not have to allow a comparison of equivalence. Therefore if it finds VAR1=5 it can only be an assignment operation and nothing else. So that in GREMLIN the statement :-

PRINT VAR1=0 will carry out the assignment, then print the result.

The first two assignment ops perform simple assignment, "=" assigns double-byte words; "==" assigns a single byte. (The choice of == for one byte and = for two bytes may seem a little illogical, but a single = is the 'normal' case and so the shortest option to type is sensible.)

A 'term' may be composed of such an assignment :

```
PRINT X=3
PRINT (X=3)+2
PRINT X=3+2
```

etc. Assignment ops must have an ADDRESS on the left hand side to assign to. Variables count as addresses but numbers obviously do not (but see "*" and "?" below). The right-hand side of an assignment op is an expression, not a term. This means that assignops have a very low priority, hence the brackets in the second example above. It also means they evaluate right-to-left, allowing multiple assignment. Try typing

```
.A
.B
A=B=X=0
PRINT A,B,X
```

and see that all are zero. Be careful with brackets:

A=(B=X=0) is legal enough but (A=B)=(X=0) is not, as (A=B) is not an address. This example also shows that the "=" sign is never used as a comparison operator as in BASIC: GREMLIN has no looping constructs and thus needs no comparisons.

The byte assignop "==" is not important except assigning to registers (see section 5) and using with "*" (see below).

6.2.6 Assignment operators (2).

There are ten more assignment ops, all of the form :

```
op=
```

where "op" is one of the ops and precedes an '=' symbol. For example :-

```
X+=2
```

performs the same function as

```
X=X+2
```

This is a very common construct and the assignment ops are extremely useful in abbreviating expressions. In general, any expression of the form

```
X = X op Y
```

may be abbreviated to :-

```
X op= Y
```

This extends even to using multiple assignments etc. For instance, it is possible to produce a sequence of square numbers by typing

```
.X=0
.Y=-1
*KEY4 PRINT X+=Y+=2|M
```

and repeatedly pressing key f4. In BASIC, f4 would have to be

```
*KEY4 Y=Y+2:X=X+Y:PRINT X|M
```

6.2.7 Incrementing ops: ++ --

It is often desirable to be able to automatically increment a variable after using it. This is much more useful when looping constructs are allowed but can be of use in the assembler too. To print X and increment it, type :

```
.X=0
PRINT X++
PRINT X
```

Similarly X-- decrements X after using it. It is also possible to increment or decrement before using X, for example :

```
.X=0
PRINT ++X
```

will yield 1. Naturally these 'inc ops' apply only to variables: ++1 is as meaningless as 1=X+3. Inc ops have high precedence and may be used in expressions; for instance to produce a series of triangle numbers, type

```
.X=0
.Y=0
*KEY4 PRINT X+= ++Y¦M
```

and keep pressing f4. Note that the "+" and "-" symbols are somewhat overloaded: ambiguities may be resolved by putting spaces in suitable positions. In fact the expression above would work perfectly without spaces, but

```
PRINT X=X+++Y
```

is NOT equivalent - it would be interpreted as

```
PRINT X=X++ +Y
```

6.3 Addresses and numbers: *, ÷.

Inc ops and assignment ops by their nature only operate on addresses. A number may be used as an address by preceding it with "*", while the reverse operation is performed using "÷". For instance, typing

```
CLEAR
.X
PRINT ÷X
```

yields the same value as

```
PRINT $VL+2
```

which is the start of the variable table, plus 2 for the name "X" and a length marker. The "*" operator performs the reverse operation:

```
SW HX 1
PRINT *202
```

will give the start of the BRK routine - i.e. the contents of location 202 Hex. The effect of "*" is similar to BASIC's "?" except that it operates on a different word size. Thus

```
PRINT *÷X
```

and

```
PRINT ÷*X
```

are both equivalent to simply PRINT X. Beware, however, of using '*' in assignments. Two things to watch out for: firstly, the ever present problem of confusion between the expression evaluator and command interpreter, which is simply resolved by adding spaces; secondly the syntax of "*" belies the fact that it operates on a whole TERM, including assignment ops etc. Thus :-

```
*1234=2
```

works perfectly, sending 2 to location 1234 - because "=" cannot work on 1234, it must use an address as provided by the "*". But

```
X=1234
*X=2
```


assigns 2 to X because the "=" can operate on X, as it is an address. The correct syntax is

```
*(X)=2
```

which has the desired effect. Note that with

```
(*X)=2
```

the "=" will be ignored altogether because (*X) is a - 'the contents of' location X. We emphasise, USE CARE with "*".
Finally "*" may be used to assign bytes:

```
* $M++ == 123
```

is exactly the same as

```
P 123
```

in that it puts a byte into memory and increments $M.

6.4 Points to watch

The expression evaluator stops as soon as it finds something which it cannot understand. Depending on how deep into an expression an error occurs and what type it is, you will get either a "No such variable" or "Missing)" message, or sometimes nothing at all, when the mistake is subtle - such as placing an assignment op after a number (not an address) - it may just stop and not report an error, in which case the wrong result will have been evaluated. It is this that makes it very important to practise the expression evaluator - assignment ops, incrementing ops, "*" and "÷" in particular - if you're not sure of it, to eliminate all possibility of undetected error during assembly. The worst errors are those that are not reported.

7 PRINTERS AND GRAPHICS.

Special thought has gone into the area of debugging graphics programs. This is handled by sending status output to the printer and/or spool file while the graphics output from the user program is allowed full use of the screen. This allows even single-stepping of multi-byte PLOT instructions whilst monitoring the screen.

7.1 Screen modes.

GREMLIN works in any screen mode. However, no status is output in 20-column modes and mode must be changed using the MODE command or GREMLIN will get confused about screen dimensions. The screen is measured every time MODE is invoked and status display altered to suit. This ensures compatibility even with the U.S. versions of the BBC Micro.

MODE <expr>
Sets screen mode to <expr>. Note that mode 7 is usually preferable from a speed point of view because the status display takes so long to update in other modes.

7.2 GREMLIN output devices

These are controlled by the switch PF. On entry PF=0. PF controls four output devices of the BBC Micro, which the are enabled and disabled in various ways by the O.S. Below, FX3<x> represents the xth bit of the FX3 byte:

VDU: The normal screen display.
FX3<1> disables;
in addition VDU6 enables, VDU21 disables.

BPRN: the printer enabled by ctrl-B.
FX3<2> disables;
in addition VDU2 enables, VDU3 disables.

SPOOL: the spool file.
FX3<4> disables;
in addition *SPOOL <fsp> enables, *SPOOL disables.

PRN: the independent printer.
FX3 bit 3 enables.

PF controls the setting of FX3 during status output, user machine-code output and GREMLIN command output. Thus any attempt to set FX3 on the user's part will have very temporary effects. However, the user has complete control over VDU6/21, VDU2/3 and *SPOOL, and can separately enable BPRN, the VDU and SPOOL using these commands.

7.3 Sources of output.

GREMLIN divides its output sources into 3 groups, command, status and user. These are routed to different devices according to PF.
Command output includes the typing of the command line, the output of such commands as PRINT, D and T, and error messages.
User output consists only of that which is produced by JSR OSWRCH (OSASCI, OSNEWL) instructions within the user's code, while he is single-stepping or continuous-executing it (not CALLing it; this is treated as a command).
Status output is simply the status display, as sent after command execution and during single-stepping or continuous execution after breaks.

7.4 Normal debugging: PF=0

GREMLIN is entered with PF=0. This causes status to be sent to VDU (unless in a 20-col mode) while commands and user output go to VDU, BPRN and SPOOL. This means it is perfectly legitimate to use ctrl-B and *SPOOL and the output will be what appears on the lower half of the screen.

7.5 Debugging graphics programs: PF=1

Graphics programs would suffer from two complaints in normal mode: GREMLIN output getting in the way, and interference with multi-byte PLOT (etc) sequences caused by using the VDU. So the VDU is given over to the user alone in PF=1 mode, while commands and status are sent to PRN and SPOOL. BPRN is disabled.
A special status panel is used with the printer, as it is obviously desirable to keep information down to the minimum usable. It shows the registers, program and one line of memory only, in a format that has been designed to look good on 40-, 80- and 132-column printers. As usual the disassembled instruction is the next one to be executed, so some predictions can be made. While single-stepping etc in this mode, it will be necessary occasionally to step out of the debugging process to alter register, look at a different area of memory, etc. While doing this, it would perhaps be desirable to turn the status display off to avoid wastage of paper, for which purpose the SE switch must be used.
Other effects of PF=1 are that tabulations in response to the "T" command are always produced in 16-column format, even in 40-column screen modes.

8 THE ASSEMBLER - Introduction

The GREMLIN assembler is designed to be used either for single-instruction entry, e.g. for making small changes to routines already assembled in memory, or for the entry of whole programs. In either case, the assembler has access to the variable-handling and expression-evaluating facilities of GREMLIN. Long assembly files are handled simply as individual instructions being entered, except that they will be input by *EXEC from a pre-prepared file instead of direct from the keyboard. This means that assembly files are produced in conjunction with an external text editor, and corrected by that means as well.

It is assumed throughout this section that SW HX 1 HN 0 is active, thus Hex numbers beginning with a numeric digit are recognised, giving freedom in variable usage. The '&' symbol will be used for clarity in certain instances (& indicates a Hex number).

8.1 The assembler

The GREMLIN assembler syntax is quite standard and should not cause any problem. The purpose of this section is to describe the use of related commands and facilities for two-pass assembly etc.

Assembly statements are entered one per line and may start with a '.label' as in BBC BASIC. It is NOT necessary to start with a special symbol (e.g. the left-arrow in BASIC), neither is a termination symbol required. This means that assembly statements may be mixed freely (on separate lines) with other GREMLIN commands.

Commands may be entered individually at any time, but the source statements are not retained, rather they are assembled and forgotten. To assemble programs of more than a few statements it is strongly recommended that a text editor (e.g. WORDWISE) is used to create a source file. The source file is then assembled by using the Operating System *EXEC command, causing the file to be read in as if entered as individual statements from the keyboard. Some additional support commands will be required for assembly of long files, e.g. assigning an area of memory for variable storage, performing two passes to cater for forward references, setting the code origin, etc.

The memory pointer $M is used by the assembler as a pointer to the next address into which code will be assembled. Setting $M to the code origin should therefore be one of the first steps in an assembly file.

Two pass assembly is achieved by use of the AO switch, rather than with the 'OPT pass' method employed in BBC BASIC. On the first pass, the AO switch should be set 'ON' with the command : SW AO 1 and turned off again for the second pass with SW AO 0. This means that on the first pass an undeclared variable will be assigned the current value of $M. On the second pass any variable which is still undeclared will be reported as such with an error message and assembly will be halted. This is designed specifically to allow the use of jumps, branches, calls, etc. to be used with variables (labels) which are not defined until a later section in the code (forward references).

Certain steps will be carried out whenever one or more files are to be assembled. They can be assigned to a function key for ease of use when repeated assembly/re-assembly is required. Listed over the page are some of these general steps.

Assembling one or more long files, general steps required :

```
COMMAND  PURPOSE
   1) .origin=&3000          Define a variable as code origin.
   2) .VL=&1A00    Set variable storage LOW pointer.
   3) .VH=&2FFF    Set variable storage HIGH pointer.
   4) CLEAR        Reset variable storage with CLEAR.
   5) SW HX 0      Set switch for default Decimal input ?
Pass one...
   6) $M=origin    Set memory pointer to code origin.
   7) SW AO 1      Set Assembler Option switch 'ON'.
   8) *EXEC progl Assemble the file.
Pass two...
   9) $M=origin    Set memory pointer to code origin.
  10) SW AO 0      Set Assembler Option switch 'OFF'.
  11) *EXEC progl Assemble the file.
Assembly complete...
  12) PRINT $M     Print next free address (optional).
  13) $M=$PC=origin          Reset $M and $PC to start of code.
```

8.2 Labels

Labels in GREMLIN operate in much the same way as in BBC BASIC. A variable is created (or re-assigned) by preceding it with a dot (.), for example :-

.label

would create a variable called 'label' and assign to it the current value of $M, the memory pointer.

8.3 Using GREMLIN commands in assembly files

The 'P' command will be frequently used within assembly files, being the substitute for the BASIC-(II) commands EQUS, EQUB, EQUW, EQUD. Users of BASIC-(I) will not be familiar with these, but simply they allow strings, bytes, words, or double words to be 'put' into memory. The 'P' command is more versatile, allowing any mixture of strings, bytes, etc. to be put at the current $M address. Programs designed for assembly by BASIC will need changing to make use of the 'P' command if they currently use either the indirection operators or the BASIC-(II) EQU commands mentioned. A search-replace operation from the wordprocessor can be employed if a lot of changes are necessary. See section-2.2 for details of the 'P' command.

Setting the $M variable to a different value during assembly is perfectly acceptable. It will cause further assembly code to be positioned at the new location according to $M. For instance, to reserve &100 bytes of memory, use the command :-

$M=$M+&100 (or $M+=&100)

at the required position in the file.

Other commands such as PRINT will be used within assembly files, though few other commands would be used as frequently.

Both passes over several files (being assembled one after another) can be achieved with relative ease. The last command in each file should be *EXEC followed by the name of the next file. The final file will obviously not end with such a command. It is suggested that when assembling several files, a general 'start' file and a general 'end' file are created. The start file would perform some of the steps described earlier in section 8.1, but would also define some general variables such as O.S. routines, zero page memory usage etc. An example is shown below :

```
*¦ General Start File.
SW HX HN 1
SW SE 0

*¦ STANDARD VARIABLES:-
.OSRDCH=OFFE0
.OSASCI=OFFE3
.OSNEWL=OFFE7
.OSWRCH=OFFEE
.OSWORD=OFFF1
.OSBYTE=OFFF4
.OSCLI=OFFF7
.delt=7F
.cr=OD
.lf=OA
.bell=7
.cls=OC
.ctrlu=15
.keybuf=5000
*¦ Zero Page Usage :
.acc16a=70
.acc16b=72
.acc8a=74
.ptr1=75
.ptr2=77
.maxlen=78
*¦ Set Code Origin
.ORG=3000
$M=ORG
*¦assemble first file
*EXEC file1
```

Notice that the status is disabled at the start, in order to speed assembly by removing the delay caused by automatic status update. Status is re-enabled in the 'end' file. An example of an end file is given below :

```
*¦End Routine For Multi File Assembly
SW AO 0
SW SE 1
*¦ Reset Status Display
$M=$PC=ORG
```


8.4 Setting breakpoints

The eight breakpoints can be set from within an assembly program file by setting the system variables $B0-$B7. It is also possible to allocate the next available breakpoint at a position by making use of the expression evaluator's facilities, together with the way that breakpoints are implemented. The breakpoints are stored three bytes apart, starting with $B0. If a counter is set up called, say, NBP for the number of breakpoints so far allocated, then at each location where a breakpoint is to be set a single expression can be used to allocate the next available breakpoint. The expression would be :-

```
*(3 * (NBP++ % 7) + &$B0) = $M
```

which simultaneously calculates the address of the next breakpoint, sets it to $M, and increments NBP. The "% 7" part is included to ensure that memory beyond the eight breakpoints is not overwritten if NBP accidentally exceeds the value of eight. Spaces are used above for clarity and can be omitted (except the first one, which would cause an OSCLI call if omitted). The beginning of the assembler program should Clear Breakpoints with a CB command. An LB command to list all the breakpoints set may be usefully added at the end of the file.

8.5 Commenting assembly files

No special symbol is required to precede comments. After interpreting an assembly statement, GREMLIN moves onto the following line. Therefore it is possible to put comments on lines following assembly statements. An important exception to this rule is caused by the internal workings of GREMLIN. When a label appears at the start of a line, multiple assembly statements WILL be interpreted, up to the end of the line. A comment appearing on such a line would be misinterpreted as an assembly statement. In summary : DO NOT COMMENT A LINE THAT STARTS WITH A LABEL.

Because no symbol is used to indicate that a comment follows, the syntax does not allow for a line containing only a comment and nothing else. The reason this is not implemented is that the facility is already provided by the operating system. Any line that starts with the asterisk symbol '*' followed immediately by the double-bar symbol '¦' is ignored. For instance, a line entered as :-

```
*¦ This is a comment
```

would be passed to the Operating System, which would ignore it.

Examples :-

```
*¦ example statements follow
.OSWRCH=&FFEE
.start
      LDA £'z'      load A with letter z
      JSR OSWRCH    print character
      RTS           return.
```

Ensure that a space or other delimeter appears before comments to separate them from the preceding command.

Possible problems with syntax

Like many 6502 assemblers this one treats any opening round-bracket after the mnemonic as an attempt at indirect addressing, and any "A" as an attempt at accumulator addressing (where valid). For example,

```
LDA(ADDR+1)<<2
```

would produce a "Bad addressing mode" error because (indirect) addressing would be expected, which is not supported by LDA. Similarly,

```
ROL ACCUM
```

would be assembled as ROL A without any error being flagged at all - which is far worse. If it is necessary to start the operand with these symbols, use the "+" OPERATOR to avoid confusion. In this circumstance, the '+' symbol serves no function, except as a delimeter, whereas it will be used for addition in the correct place. For instance:-

```
LDA+(ADDR+1)<<2
ROL+ACCUM
```


9 SUMMARY OF COMMANDS AND SYNTAX

COMMANDS - Must all begin with the first character of the command, not preceded by spaces.

SETTING THE MEMORY POINTER :

```
M <expr>
$M=<expr>
```

Add to Memory Pointer :-

```
+ <expr>
```

Subtract from Memory Pointer :-

```
- <expr>
```

Indirect Memory Pointer via value currently pointed to by $M :-

```
IND
```

ASSEMBLY STATEMENTS :

Comments may optionally follow the assembly statement, EXCEPT when the line starts with a '.label', when further assembly statements may (optionally) follow on the same line.

Either: <.label> (<Assembly statement>)
Or: <Assembly statement> (<comment>)

Inserting data at Memory Pointer :-

```
P <data>
```

Data is inserted according to $M. <data> may be any series of:

<expr>; - assembled as a word
<expr>(,) - assembled as a byte
<assembler statement>
"string" - *KEY syntax

Searching with the 'F' command :-

Search memory for specified data, starting at $M+1. When is found the status is updated and $M points to the data. <data> may be any of the types listed for the 'P' command above.

```
F <data>
```

Searching for several occurrances is easily achieved by repeatedly issuing the find command (by copying it from the input line above).

Displaying memory in Tabulated form :-

Display memory between two limits in tabulated form.

```
T <start> <end>
```

Disassembly :-

Memory is Disassembled according to the parameters listed below.

```
D <strt> <end> (S) (F"filename"(heading))
```

<strt> - Start address, expression allowed.
<end> - End address, expression allowed.
(S) - Capital letter S, show source code.
(F"string" - Capital letter F followed by a filename, sends output to a spool file.
(title)) - Any string, placed at start of spooled file.

Moving an area of memory :-
Move a specified amount of memory from source address to destination address. source and destination areas may overlap.
IM <source> <dest> <amount>

Executing Code :-
Similar to BASIC, the CALL statement causes execution of code until a return from the routine.
CALL <address>

Single-stepping :-
Single-stepping will commence from the current location held in the PC register. The 'S' command MUST be enabled with the Jump Enable switch before it will operate.
SW JE 1
S
Press the RETURN key after each step to perform the next step, or any other key to enter other commands.

Execute code with breakpoints :-
Code is executed until a breakpoint is encountered. The 'J' command MUST be enabled with the Jump Enable switch.
SW JE 1
J <address>
When a breakpoint is encountered, press RETURN to continue execution, or any other key to enter other commands.

Continue after breakpoint :-
Execution (as with 'J' command above) continues from address currently in $PC. The 'C' command MUST be enabled with the Jump Enable switch.
SW JE 1
C

Clear all Breakpoints :-
CB

List all breakpoints :-
LB

Operating System commands :-
The entire line following an asterisk '*' is passed to the O.S. in the usual manner.
*<OS command>

Printing values on screen :-
Printing is according to current format, set by HX, BF, switches etc.
PRINT <expr> (,<expr>)..

Initialising and assigning variables :-
Variables are first declared with the dot symbol '.' and may optionally be assigned a specific value at the same time. Once a variable has been created, it need not be preceded by the '.' symbol and may be used in expressions.
.<varname> (= <expr>)

Clear all variables :-
CLEAR

List all variables :-
LVAR

Select screen mode :-
MODE <expr>

Setting switches :
Any switch(es) can be set to the 'on' or 'off' settings with the 'SW' command.
SW <switch list> <setting> (<switch list> <setting>)
Where:-
<switch list> - is a list of one or more two-character switch names;
<setting> - is the digit 1 or 0. Variables are not allowed.

SWITCHES - Listed with initial value :

HX 1 HeX number base
0 = decimal
1 = hex

HN 1 Hex Number interpretation
0 = must start with numeric
1 = may start with alphabetic

DR 0 Disassemble Relative
0 = normal disassembly
1 = Branch instructions disassembled into relative format.

BF 0 BASIC Format
0 = normal numeric printing
1 = BASIC-compatible hex printing preceded by "&", and P% in relative disassembly

AO 0 Assembler Option
0 = Error message if variable not declared
1 = Assign value of $M to undeclared variable

PF 0 Printer Flag
0 = normal use
1 = debugging graphics programs

SE 1 Status Enable
0 = no status display
1 = normal display

JE 0 Jump Enable
0 = S, J, C commands disabled
1 = S, J, C commands disabled

SYSTEM VARIABLES :

All used in expressions with $ prefix. Initial value in brackets if initialized.

M - Memory pointer
PC - Program counter
A, X, Y, P, S - corresponding processor registers
VL (0600), VH (0800) - Low and High ends of variable area
DL (0000), DH (8000) - Low and High ends of user program area.
B0, .., B7 - Breakpoints.
ROM - Current sideways ROM.

SYNTAX OF EXPRESSIONS :

expr: term [op term] ...

term:
- number
- address
- address assignment op expr
- address ++
- address --
- ++ address
- -- address
- ÷ address
- unary op term
- @
- (expr)

address:
- variable
- * term

op:

+	add
-	subtract
*	multiply
/	divide
%	modulus
&	bitwise AND
¦	bitwise OR
↑	bitwise EOR
<<	rotate left
>>	rotate right

assignment op:

=	word assign
==	byte assign

+=, -=, *=, /=, %=, &=, ¦=, ↑=, <<=, >>=

unary op:

+	no operation
-	negate
!	complement
<	high byte
>	low byte

10 ERROR MESSAGES

All give error code zero (though there is no way of finding this out by standard commands) and print a message on the screen. All files are closed - to prevent a *EXECed assembler file from running amok - but screen mode, printer control, etc, remain unaltered on receipt of an error. There are several ways in which errors may arise: this section is accordingly subdivided.

It is necessary to know the workings of the command interpreter to interpret some error messages: the interpreter first checks input to see if it represents a valid assembler mnemonic; then it checks for a command - this MUST begin in column zero, ie with no leading spaces; if it had still had no success it passes the command line to the expression evaluator. This means that, where there might be confusion between an expression like

```
 S=3
```

and a command like

```
S
```

the expression must be preceded by spaces. This is especially relevant when using the "*" operator to start an expression.

10.1 Command errors.

Escape
Results from pressing ESCAPE during command input, disassembly or memory tabulation.

Bad switch
Attempt to define a non-existant switch. Will also result if the SW command string does not end in "0" or "1", or possibly if an expression is used instead of "0" or "1".

Not enabled
Attempt to use the S, J or C commands without first typing

```
SW JE 1
```

to enable them.

Filename?
Bad filename after the "F" option in the disassembler command. Quotes must be used.

10.2 Assembler errors

Addressing mode?
Attempt to use illegal addressing mode. May also be result of misinterpretation, which can be avoided by using "+".

Assembler syntax?
Bad formation of assembler operand, eg wrong index or missing closing bracket on indirect addressing modes.

Byte
Immediate data will not fit into byte. Use ">" operator.

Out of range
Branch destination out of range. Must be replaced by JMP.

10.3 Expression evaluator errors

10.3.1 Variables

No room
An attempt to define a variable with the "." command with insufficient room left in the variable table area. Try increasing $VH.

Bad variable
An attempt to define a variable with an illegal name was made.

No such variable
An attempt to access a variable which had not been defined, or a mistyped command interpreted as an expression, or an attempt to type a hex number not beginning with a numeric character while HN=0, or certain malformed expressions, will give this message. Like its BASIC counterpart, this covers a multitude of sins.

No such system variable
Whatever follows a "$" sign must be a system variable. This message will result if it isn't.

10.3.2 Others

Missing)
Closing bracket needed and not found in correct place.

Divide by zero
The left-hand operand of the /, %, /= or %= operators is zero.