# Accelerator

## Instruction Manual

Computer Concepts

# *Errata extra*

First of all I would like to thank Jalov that he was willing to scan the whole manual for me so I could create this digital version.

There are some changes comparing to the original manual.
During the conversion I could not keep the same page numbering as the original one. So I had to create a new Contence page with different page numbers.

Also the index pages at the end of the manual are not there. This because this is a lot of figuring out what has changed comparing to the original manual and this manual.
Together with this manual I have included the original DFS disk (in image format). But I also have included an ADFS version of this disk (also as disk image)

There is a slight difference between the DFS and the ADFS disk image.
On the DFS disk there are files with an '&' symbol in it.
I had to change the name of these files on the ADFS disk image, because ADFS does not support this character in their filenames.
All the files that start with 'L1' have been moved to a directory called L1. This is the same for L2 and L3. So L2 files are in directory L2 and L3 files are in L3.
Then I have renamed the filenames. e.g. L3&1900 is 1900 in directory L3.
So to call the file you have to type L3.1900 instead of L3&1900.
(The files are explained on the 'Errata pages' from Computer Concept.)

If there are any corrections to be done in this manual, (spelling or style errors or any other) please let me know.
Hope you Acorn people like the new addition to the library.

K. Keevel (k.keevel@chello.nl)

# Accelerator

## *Errata Sheet*

**1. Disc contents**

The disc supplied with Accelerator is readable by both 40 and 80 track drives. Side zero is recorded in 40 track format and the other side is in 80 track format. It is supplied write-protected and so should be backed up onto a fresh disc as soon as possible. You can do this using the normal DFS *BACKUP command, taking care to specify the correct side of the disc for your drives.

The disc contains the following files:

**CONVERT** - The machine code converter, which should be executed by typing *RUN CONVERT.

**RELOC** -The G-code relocator, which should be executed by typing *RUN RELOC.

**LIBGEN** - The library generator, which should be executed by typing *RUN LIBGEN.

If any of the above programs are accidently CHAINed, an appropriate error message will be given.

**L3&1900** - A level 3 library for use at &1900.
**L2&1900** - A level 2 library for use at &1900.
**Ll&1900** - A level 1 library for use at &1900.
**L3&0E00** - A level 3 library for use at &0E00.
**L2&0E00** - A level 2 library for use at &0E00.
**Ll&0E00** - A level 1 library for use at &0E00.
**L3&0800** - A level 3 library for use at &0800 on the second processor.
**L2&0800** - A level 2 library for use at &0800 on the second processor.
**Ll&0800** - A level 1 library for use at &0800 on the second processor.
**L3&8100** - A level 3 library for use at &8100 in sideways ROM generation.
**L2&8100** - A level 2 library for use at &8100 in sideways ROM generation.
**Ll&8100** - A level 3 library for use at &8100 in sideways ROM generation.

**LIFE** - A simple version of John Conway's Life simulation. This program is supplied for demonstration purposes only, and should not be construed as an attempt at writing the ultimate version of Life. Note that the program cannot be used on the Tube. It will compile to G-code and machine code without problems. When the program is run, it will display a blank mode 7 screen, waiting for you to type in the starting configuration. To do this, you can manipulate the cursor over the screen using '2' for left, 'X' for right, '*' for up and '?' for down. To fill the current cell, press the return key. The program will start computing subsequent generations when the 'E' key is pressed.

**CIRCLE** - The demonstration program from page 3 of the manual.

**CHNl, CHN2** - The demonstration programs from page 34 of the manual.

**2. VAL**
The function VAL normally returns a floating point result. If you want it to return an integer result (to allow it to be converted to machine code for example), you must use VAL % instead. '

**3. The machine code converter**
The machine code converter issues another prompt when generating sideways ROMs. The prompt is 'Variable address:'. The response should be the hexadecimal address where variable storage should begin. A typical value would be &2000. The converter needs this information so that it can assign new storage addresses to all the variables in the program under conversion. It will then use all the space from the address specified up to HIMEM to store strings, arrays and non-resident integer variables.

**4. Resident integer variables**
As mentioned in the manual, the machine code converter stores the resident integer variables in zero page rather than page four. This presents problems when ROMs like the Graphics ROM are used, which expect to find the values of these variables in page four. There are two solutions. Either access page four directly, using the! operator, or use the OSCLI statement. For example, rather than *FORWARD T%, use OSCLI "FORWARD" +STR$ (T%).

**5. Code length**
The G-code compiler gives one extra line of information when it has finished compiling a program. This is the length of the G-code program generated, given as a decimal number of bytes.

**6. *ACCELERATOR**
To reduce clashes with the DFS *ACCESS command, Accelerator will ignore *ACC. or *A. if any further text is included on the same line. Thus, *A. will be treated as *ACCELERATOR, but *A. PROG will be ignored by Accelerator, allowing the DFS to treat it as an *ACCESS command. Accelerator also follows the Computer Concepts standard of allowing a letter C to be used as the first letter of *ACCELERATOR and *GRUN, to prevent further clashes.

# CONTENTS



Accelerator is designed and distributed by Computer Concepts.

This manual was typeset direct from Wordwise files by Quorum Technical Services Ltd., Cheltenham.

The abbreviation BBC Micro for British Broadcasting Corporation Microcomputer has been used throughout this book.

# 1. Introduction

Accelerator is a BASIC compiler system for the BBC Micro that can make BBC BASIC programs run up to ten times faster than normal. It is compatible with all the standard SSC Micro configurations, including the 6502 second processor, discs, Econet and, to a lesser extent, cassettes.

Accelerator increases the speed of programs by translating them either into machine code or a specially designed 'artificial' machine code called G-code. In the case of the former the resulting program can be executed with *RUN on any BSC Microcomputer, regardless of the presence of Accelerator. G-code programs are executed by a module within Accelerator, and so can only be run on machines fitted with Accelerator.

As a rough guide, G-code programs are up to about five times faster than the original BASIC program and machine code programs increase in speed by about three times faster still.
The actual speed increase measured for a particular program depends greatly upon the nature of the program. For example, a program doing extensive file handling should not be expected
to show a dramatic increase, since the computer (whatever language it is running in) will spend more time waiting for the comparatively slow disc and cassette systems than in actual computation. Conversely, a program written using long variable names and other 'readability' features can be expected to show a quite dramatic speed increase.

Although G-code programs execute much more slowly than machine code, Accelerator can compile to G-code in a fraction of the time it takes to compile all the way to machine code.

The vast majority of BBC BASIC programs can be translated into G-code without modification, but only programs that confine themselves to integer arithmetic can be further refined to machine code.

Special features of Accelerator include provision for developing sideways ROMs from BASIC programs. When a program is compiled to machine code, Accelerator offers the option of producing a special sideways ROM format file which can be blown into an EPROM or loaded into sideways RAM. Once the ROM is installed, the program in it can be run with a *command of your choice.

The Accelerator package comprises this manual, a quick reference card, two ROMs and a disc. The two ROMs contain the BASIC to G-code compiler and the G-code interpreter. The disc contains the machine code converter module and some associated utilities.

The Accelerator system requires OS 1.2 or later. It is designed to mimic BASIC II (users with BASIC I will find that `OPENIN` is treated as `OPENUP` and the `OSCLI` statement is not accessible). The disc or Econet system (or another filing system which allows three files to be open at the same time) is required by the machine code converter module, although both the G-code compiler and interpreter make provisions for tape users. Accelerator works with and takes advantage of the 6502 second processor and HIBASIC.

# 2. Sample session

This section leads you through a sample session with Accelerator, covering compiling a BASIC program into G-code and converting the G-code program into machine code.

First ensure that the computer is in BASIC, if necessary by issuing a *BASIC command, then type in the following program:

```
10 MODE4
20 VDU 29,640; S 12; 30 TIME=0
40 FOR Radius%=100 TO 10 STEP -10
S0 PROCcircle(Radius%)
60 NEXT Radi us%
70 PRINT TIME
80 END
90
100 DEFPROCcircle(Radius%)
110 LOCAL Xco%,Yco%,Di ff% 120 Xco%=0
130 Yco%=Radius%
140 Diff%=3-2*Radius%
1S0 IF Xco%>=Yco% THEN GOTO 200
160 PROCref lect (Xco%,Yco%)
170 IF Diff%<0 THEN Diff%=Diff%+4*Xco%+6 ELSE
Diff%=Diff%+4*(Xco%-Yco%)+10:Yco%=Yco%-1 180
Xco%=Xco%+1
190 GOT0150          ____
200 IF Xco%=Yco% THEN PROCref lect (Xco%, Yco%)
210 ENDPROC
220
230 DEF PROCref lect(Xco%,Yco%)
240 LOCAL Xfact%,Yfact%
2S0 FOR Xfact%=-1 TO 1 STEP 2
260 FOR Yfact%=-1 TO 1 STEP 2
270 PLOT 69,Xco%*Xfact%*4,Yco%*Yfact%*4
280 PLOT69,Yco%*Yfact%*4,Xco%*Xfact%*4
290 NEXT Yfact%,Xfact%
300 ENDPROC
```

Execute the program by typing RUN. It will draw a series of different sized circles centred on the middle of the screen. when the program finishes, note down the time taken (typically 21 seconds).
To compile the program, type *ACCELERATOR, or a suitable abbreviation. Accelerator will respond by printing its name, a copyright message and a prompt:

```
>*ACCEL.
ACCELERATOR

(c) 1985 Computer Concepts

Source filename :
```

Press the RETURN key. Accelerator will then issue another prompt:

```
Object filename:
```

Press the RETURN key again and Accelerator will compile the program. As it does so, it prints the final line number of the program three times. After compilation, the G-code version of the program will be stored in memory along with the original BASIC program.
Accelerator will then ask you if you wish to run the compiled program:

```
Execute program (Y/N) :
```

Press the RETURN key again, which in this context is taken as meaning 'Y' or 'yes'. (Pressing 'N' would drop you straight back into BASIC without running the program).
Accelerator responds by printing:

```
G-CODE INTERPRETER
```

and then running the G-code version of the program. When the program has finished, compare the time taken with the original timing under BASIC. A typical reading is 11 seconds.
The G-code interpreter will then issue the prompt

```
Re-execute (Y/N):
```

Pressing the RETURN key or 'Y' will re-execute the program and pressing 'N' will return you to BASIC. Notice that an OLD command is

automatically given as soon as BASIC is re-entered, which restores the original BASIC program.
The method used above can be summarised into this golden rule:

## To compile and execute a program, enter Accelerator and press the RETURN key three times.

Using Accelerator in this way makes it act rather like a supercharged version of the RUN command of BASIC since no disc or tape accesses are required, and the actual compilation is virtually instantaneous.
The speed increase gained in this way can be bettered by converting the G-code program to machine code using the machine code converter and then running the machine code. The machine code converter will not run on cassette, but the programs it generates will.
When you drop back into BASIC after running the program, the original BASIC program is still present. You can verify this by LISTing it.
First, save the program on disc, calling it ORIGNL. The next step is to compile it to G-code again, but this time to save the resulting program on disc. The following dialogue shows how to do this:

```
>*ACCEL.
ACCELERATOR

(C) 1985 Computer Concepts

Source filename:ORIGINAL
Object filename:GCODE

300
300
300

Execute program (Y/N):N

BASIC
>OLD
>
```

Next, invoke the machine code converter itself. Type:

```
*RUN "CONVERT"
```

The converter will clear the screen and print:

```
MACHINE CODE CONVERTER

G-code filename:
```

Type the filename `GCODE`. The converter will then ask for the:

```
Library filename:
```

Type the filename `L3&1900`. The converter will then print:

```
Machine code filename:
```

Enter `MCODE`. The converter will come back with:

```
Sideways ROM format (Y/N):
```

Press the 'N' key.
The converter will then set about converting the G-code program called `GCODE` into a machine code program called `MCODE`. This will take about a minute.
At the end of the process it will print:

```
Execute program (Y/N):
```

If you press 'N', the system drops back into BASIC, but pressing 'Y' or the RETURN key will load the machine code back off the disc and execute it. Under machine code, the program takes about six and a half seconds to run.

# 3. G-code theory

Accelerator is a tool like any other; to make effective use of it, it helps to understand a little of how it works. This section covers the theory behind the G-code system and explains why a G-code program is faster than the original BASIC prig-ram. By understanding this, you can judge what speed increase to expect for a particular program and write programs, or modify existing ones, in such a way that a good speed increase is obtained.

As mentioned in the introduction, it is useful to think of G-code as being a sort of mega-machine code. It is designed so that most fundamental BASIC operations can be implemented with a single G-code. This means that the G-code form of a given BASIC program is small compared with the machine code form and not much bigger than the original BASIC version.

The only slight problem with G-code is that the BBC Microcomputer is not designed to directly execute programs written in it. Instead, part of one of the Accelerator ROMs contains a 12K machine code program called the G-code interpreter. The interpreter 'executes' G-code programs by reading G-codes one at a time from the program and acting upon them. Since G-code was designed to be executed in this way, the process is efficient.

There are several fundamental differences between G-code and BASIC, and it is these differences that make G-code faster. The most important one is that variables are not referred to by their names, but by the address in memory where they are stored. For example, every time you refer to the variable `AREA%` in a BASIC program, the BASIC interpreter has to search through an internal list of all the variable names in use to discover the associated value of the variable. This time-consuming process would have to be repeated a thousand times if the variable were referenced inside a loop of a thousand iterations. With G-code programs, this conversion process is carried out once and for all when the program is compiled. Thereafter when the program is executed, the G-code interpreter knows precisely where to find the value of `AREA%`. To extend our example, take a statement such as `SPACE%=SPACE%+AREA%`. The only actual computation taking place is an addition. In BBC BASIC the addition itself takes a tiny percentage of the total time required by the statement, because it takes much longer to find the addresses of the variables and to retrieve their values. Under G-code, the actual addition is no faster but there is no overhead in finding the addresses of the variables and hence the whole statement is executed much faster. One benefit is that since variable names themselves do not exist in G-code programs, there is no speed or space penalty for using long ones. You can demonstrate this easily with a short program written using long variable names and the same program written

using single character variable names and comparing the running times reported by BASIC and G-code. Since `REM` statements and spaces are also weeded out when conversion to G-code takes place, the compiler makes it possible to write readable, well-documented programs that execute at a respectable speed.

The performance of G-code in graphic applications is subject to a similar effect. Since both G-code and BASIC use the operating system for all graphic operations, G-code cannot offer an appreciable speed increase for raw graphics statements like MOVE, DRAW and PLOT. Instead, its speed advantage has to be gained from speeding up the calculations leading to the actual drawing. For example, a BASIC graphics program taking thirty seconds to run may spend 15 seconds of that time calling operating system routines to draw the actual graphics. This means that there is no possible way for Accelerator to increase the speed of the program by more than 50%. In real life, the program will probably only run in 70% to 80% of the original time.

A problem with BASIC is that numbers in programs are stored as strings, in exactly the same way as they are typed. Before a number can be used in a computation, it has to be converted into an internal binary form. Normally, this conversion is carried out every time a given number is required, so that in a statement like `IF G%>147 THEN GOTO 400` the string "147" has to be converted into the number 147 every time the statement is executed. With the G-code compiler, all the numbers in a program are converted to binary at compilation time. Since converting numbers from strings to binary is quite slow, this preconversion provides an important part of the overall speed increase offered by using G-code.

When BBC BASIC executes a `GOTO` or `GOSUB` statement, it searches through the text of the program from beginning to end until it finds a line with the correct line number. This is fine if the line is near the start of the program, but becomes increasingly slow as the program gets longer and the line gets nearer the end of it. In G-code programs, on the other hand, the destinations for `GOTO`s and `GOSUB`s are given as the actual addresses where control should resume. It should be clear that the speed increase gained by this particular dodge depends upon the length of the program. This is a common feature of many of the speed improvements offered by G-code. Many trivial programs show a trivial increase of speed when executed in G-code. As programs get longer and more complex, so BASIC becomes increasingly inefficient while G-code becomes more and more of an improvement.

This only scratches the surface of the ways in which G-code gains an advantage over BASIC; most of others are rather more subtle and don't make a very noticeable improvement on their own.

There are, of course, a few disadvantages to using G-code. It is impossible to directly debug a G-code program, since you cannot print the values of variables after an error, nor can you use TRACE or even edit the program. The intention is that the debugging of a program should take place before you compile it. If an error occurs in a G-code program, an error report will be given in the same way as BASIC. At this point you should revert to the original BASIC program and try to recreate the error and debug from there. When the error has been removed, you can re-compile. The other side of the coin is that by going back to the original BASIC program for debugging, you will be able to take advantage of your REM statements and long variable names.

An important conclusion to draw from all this is that the dodges often used by BASIC programmers to try to increase the speed of programs will not always result in faster G-code programs. For example, in heavily executed loops, it is common practice to replace a numeric constant (such as 12.347282) with a variable with the same value. This can often yield quite a dramatic speed increase. Under G-code, both representations will run at approximately the same speed.

# 4. G-code compiler restrictions

This section covers the restrictions imposed by the G-code compiler. Most of them are keywords and features that have not been implemented, either because they are inappropriate in a compiler or because they are impossible to compile. Three keywords are used slightly differently, or have different effects.

The keywords not implemented are `LOMEM, PAGE, TOP, EVAL, TRACE` and assembly language and the keywords used differently are `FN, RND` and `CHAIN`. The rest of this section is devoted to an explanation of why these differences occur, which should soften the blow and help you to remember what the differences are.

Under BBC BASIC, `LOMEM` points to the start of the area used for variable storage. The intention is that by altering `LOMEM` and `HIMEM`, you can change the area of memory used for variable storage, allowing sophisticated programmers to make room in memory for their own devices. Under the compiler, however, the addresses of all variables are fixed at the time of compilation. Thus, it is not possible to alter `LOMEM` in a compiled environment. On the other hand, `HIMEM` is implemented entirely normally, which permits you to move the top of the area used for string storage and arrays, which in many circumstances will achieve a similar effect.

`PAGE` is not implemented because every G-code program is tied to its own particular starting address, which can only be altered using the relocation utility. Since the G-code interpreter automatically moves programs to the correct address before attempting to run them, `PAGE` is just not needed.

`TOP` is not implemented for a slightly different reason. The internal organisation of G-code programs is radically different from that of BASIC programs, to the extent that there is no direct equivalent of `TOP`.

`EVAL` is used to 'work out' a string as if it were an expression. For example, in BBC BASIC you can write `PRINT EVAL("short*45.67")` and the computer will respond with 45.67 times the variable `short`. Under the compiler, variable names are lost after compilation which makes it impossible to know at execution time which variable `short` refers to, let alone its value. Thus it is not possible to implement `EVAL` in a compiled environment. However, the most common use of `EVAL` is to permit hexadecimal numbers to be used in response to `INPUT` statements, as in:

```
16370 INPUT "Address:"A$:ADD%=EVAL(A$)
```

This usage of EVAL is not needed under Accelerator, since the INPUT statement and the VAL function have been extended to allow hexadecimal numbers to be entered directly. The example above could therefore be re-written as:

```
16370 INPUT "Address:"ADD%
```

Under BBC BASIC, TRACE works by printing every line number encountered as the program progresses. Since this slows down programs, it has not been implemented. As mentioned previously, the intention is that a program will have been debugged before it is compiled, which makes TRACE unnecessary .

The reason the compiler does not handle assembly language is rather more complex. Assembly language has been implemented in BBC BASIC by including an assembler as part of the BASIC interpreter. So, at first sight, the solution is to include an assembler in the compiler. While this is possible, it could not be done in a way that is compatible with BBC BASIC. For example, consider the assembly language instruction JSR OSBYTE. At compile time, the value of the variable OSBYTE is not known (although its address is), so the statement cannot be fully assembled. The next solution would be to defer the assembly until the compiled program is executed. This is actually feasible, but pointless for most applications. For example, suppose you have written a program in BBC BASIC assembly language. Normally, you just run the program to assemble the code and then use the CALL statement to execute it. Under the compiler the sequence of events would be to compile the original BASIC program containing the assembly language, then to run the G-code version of the program and then to execute the machine code with CALL. It is possible that the G-code interpreter could assemble code quicker than BBC BASIC (but not much), but it would add the overhead of compilation to the whole process of assembly.

The solution is to assemble machine code subroutines with BBC BASIC in the normal way, and then to load and run the machine code from BASIC or G-code.

It may not be immediately obvious that BBC BASIC commands such as LOAD, SAVE and RENUMBER are not implemented for the simple reason that they are not permitted to appear in programs, but can only be used in immediate mode. Since immediate mode is inn applicable to the compiler, there is no need for any commands to be handled.

The first statement implemented slightly differently by the compiler is `CHAIN`. Under BBC BASIC this statement loads a BASIC program to the current value of `PAGE` and executes it. Under the compiler this statement loads another G-code program and runs that. The new program will be moved to the correct execution address before being run, so there is no need to set `PAGE` in advance (even if it were possible to do so ... ).
There is much scope for confusion in the way `RND` is handled. The problem occurs because under BBC BASIC the function sometimes returns an integer result and sometimes a real result, depending upon its argument. Under the compiler it behaves as follows:

1. `RND` with no argument returns a random 32 bit integer in the normal way.
2. `RND(0)` returns the last random number generated as a real number between zero and one. This is only true if the argument is the number zero, and not an expression or variable with the value zero.
3. `RND(something)` , where 'something' is a variable or expression (but not a number) that turns out to be zero at execution time, returns the most recent random number as a 32 bit integer. Thus, `PRINT RND` followed immediately by `X=0:PRINT RND (X)` will print the same 32 bit integer..
4. `RND(1)` , where 1 is a number and not a variable or expression, returns a real random number between zero and one.
5. `RND(something)` , where 'something' is a variable or expression (but not a number) that evaluates to one at execution time, returns a random number between zero and one as an integer. For reasons that should be clear, this will always be zero.
6. `RND(something)`, where 'something' is a variable, a number or an expression that evaluates to be greater than one will return a random integer between one and the argument in the normal way.
7. `RND(something)` , where 'something' is a negative variable, number or expression, will reseed the random number generator in the normal way.

The compiler handles user defined functions and procedures normally, with one exception. When you are calling a user defined function, if it will return a string result (as opposed to a numeric one), you must add a $ to the end of the name of the function at each point in the program where it is called. You must not add the $ to the name at the definition. A partially complete program might be:

```
450 PRINT FNyesno$
-
-
1000 DEF FNyesno
1010 LOCAL key$
1020 key$=GET$
1030 IF key$="y" OR key$="Y" THEN="YES"
1040 IF key$="n" OR key$="N" THEN="NO"
1050 GOTO 1020
```

If you don't include the $ in the name, a `Type mismatch` error will be given when the function call is executed.
The reason you have to do this is so that the compiler knows what type of result to expect from a function. ); a function is called and there are no symbols after the name. the compiler assumes the function is going to return a real result. If a $ is included at the end of the name, the compiler assumes the result is going to be a string. If a % is used, it assumes the result will be an integer. Most of the time, the % is optional, since the compiler can automatically convert between integer and real results as necessary. For example:

```
260 PRINT A% EOR FNexample
-
-
1000 DEF FNexample
1010 LOCALA%,B%
-
-
1500=A%+B%+NORM%
```

In this case, the function returns an integer result. Line 260 includes a call to the function, specifying that a real result is expected. When the program is run, the function will be called, yielding an integer result. The integer will then be converted to a real number. This is so that the function returns the real result specified in line 260. In this example, the result will then be turned back into an integer, so that it can be used as an argument to `EOR`. Thus, in this example, using the % would have saved quite a bit of time by removing the need for all the conversion. On the other hand, omitting it is not very serious, since the program will still run.
One further potential problem arises because the compiler . extracts the names of variables used in a 'program in the second pass by looking out for statements which define variables. If a variable is defined which it has not

already logged, it adds it to its list. This only becomes a problem when a variable is used in a program that is only defined once, and that is in an `IF` statement like this:

```
IF value=target flag=TRUE
```

No problem arises if the word `THEN` is included, or if f lag is also defined elsewhere (e.g. in a `LOCAL` statement). If a variable such as this occurs in a program, the compiler will give a `No such variable` error when it comes to it on the third pass. The solution is to include the word `THEN`.
A couple of brief points:
The `INSTR` bug of version 1 of BBC BASIC does not occur in Accelerator.
All `REM` statements are weeded out of programs at the compilation stage, and so do not add to the bulk of programs.

# 5. Using the G-code compiler and interpreter

The G-code compiler is at the heart of the Accelerator system. This section explains in rigorous detail how it and the G-code interpreter are used to compile and execute G-code programs. It also covers using the relocate utility to change the address of a G-code program.

The G-code compiler is invoked with the *ACCELERATOR command. An abbreviation can be used, but beware of the similarity with the disc filing system command *ACCESS. You may find it useful to put the two Accelerator ROMs in higher priority sockets than the disc filing system, which should permit you to use the ultimate abbreviation *A.. The disadvantage of doing this is that it forces you to use the full *ACCESS command. The G-code interpreter is invoked with the command *GRUN. This can often be abbreviated to simply *G..

Throughout both programs the terms 'source' and 'object' program are used. The source program is the BASIC program that is to be compiled, while the object program is the G-code program generated from the source program.

The G-code compiler takes a source program and translates it into the equivalent object program, while the G-code interpreter is used to execute an object program. The complicating factor in using the two programs is that the source and object programs may be taken directly from memory or from disc/cassette.

When you enter the G-code compiler, it will print:

```
ACCELERATOR

(c) 1985 Computer Concepts

Source filename:
```

Here it is prompting you to enter the name of the program you wish to compile. If the source program is on disc or cassette, you should enter the filename and press RETURN. If the program is already in memory, just press RETURN. in the latter case, the compiler will look for the program at the default value of PAGE for your machine (generally &1900 for disc based. machines and &E00 for cassette based machines). This means that it is impossible to compile a program stored at any other PAGE address. (You

can change the default PAGE address by using *FX180 , n followed by *BASIC, where n is the address you want divided by 256. This will not work on the 6502 second processor)
There is a further option in response to this prompt. If you type an operating system command beginning with an asterisk, the compiler will execute the command and then re-issue the prompt. For example, if you cannot remember the filename of the source program, you can give a *CAT command to get a disc catalogue and read the filename from there.
After you have entered the source filename, the compiler will ask you for the object filename in a similar manner. This is the filename under which the G-code program will be saved. You can either enter the filename and press RETURN, or, if you do not wish to save the object code, just press RETURN. The advantages of saving the object code are that it enables you to re-execute the program at a later date without the need to re-compile it and it allows you to feed the object code through the machine code converter.
After this information has been entered, the compiler starts to do the actual compilation. To do so, it scans through the source program three times. As it goes through, each line number encountered is printed on the screen over the previous one. The end result is that the final line number of the program is printed three times, once for each 'pass' over the source program. If you are using the cassette filing system, the compiler prompts you with either Play: or Record: followed by the filename to be used. When the appropriate controls have been set on the cassette recorder, press any key (except BREAK or ESCAPE) and the compiler will start to read or write the file in question. The source program will be read in three times, and the object code written out once.
If the source program is 'bad' or any other error occurs (such as ESCAPE being pressed in the middle of compilation), the compiler prints an error message and drops you back into BASIC (or attempts to - more on this later).
Internally, the compiler builds the object program on top of the BASIC program at the default PAGE value. If the source program is being read from disc or cassette, the compiler first constructs a BASIC program at the default PAGE value by carrying out a NEW operation.
When the program has been successfully compiled, the compiler issues the prompt:

```
Execute program (Y/N):
```

Pressing 'N' will re-enter BASIC, while pressing 'Y' or RETURN will enter the G-code interpreter to run the program.

A word about returning to BASIC: both the compiler and interpreter return you to BASIC when they have finished. To do this, they use an internal `OSBYTE` call (number 187) to establish the ROM number of the BASIC ROM. This is fine, unless you don't happen to have a BASIC ROM in your machine. In this case, the compiler issues the message `No BASIC ROM` and prints an * prompt. You can then enter any number of operating system commands, the ultimate intention being that you will enter some other language in preference to BASIC.
If the G-code interpreter is entered with `*GRUN` it issues the prompt:

```
Object filename:
```

It expects you to type the name of the G-code program you wish to execute. If the program is on disc/cassette, you can enter the filename directly. If you press RETURN without entering a filename, the interpreter looks for a G-code program above the BASIC program at the default value of `PAGE`. This option is used, bypassing the prompt, if you enter the G-code interpreter directly from the compiler. If there is no BASIC program present, or no G-code program above it, the interpreter issues the appropriate error message and attempts to drop back into BASIC.
If the object program is to be read from cassette, the interpreter will prompt you to `Play:` the relevant filename and wait for you to press a key to signify that you have done so.
It then attempts to execute the program.
When the program has finished, or after an error has occurred, the following prompt is given:

```
Re-execute (Y/N):
```

Pressing 'N' returns you to BASIC, while 'Y' or return will re-execute the program.
When using the G-code interpreter you should be aware of the fact that every G-code program is designed to be loaded and executed at a particular address in memory. This address is the top of the BASIC program upon which the compiler built the program. In order to run a G-code program loaded from disc or cassette, the interpreter may need to move it from where it loaded it to the address required for running it. A problem arises when the 6502 second processor is used, since the default value of `PAGE` drops to &800 on this machine. In this case, it is quite possible that the compiler will construct the object code to start at an address like &802 if the source program was read from disc/cassette. If you then try to run this object program on the I/O processor, the interpreter will load the program and then attempt to move it down in memory to &802. When the program

is run, it will sooner or later crash, since it is situated right where the operating system has its workspace. The solution is to use the relocation program, described below, to move the object program to a new address. Note that this problem will only arise when a second processor is being used.
You can discover the address at which a given G-code program is designed to run using the following short program:

```
10 INPUT"Object filename:"FILE$
20 HANDLE%=OPENIN(FILE$)
30 ADDRESS%=BGET#HANDLE%+256*BGET#HANDLE%
40 CLOSE#HANDLE%
50 PRINT"Start address is";~ADDRESS%
```

The object code relocation program is supplied on the same disc as the machine code converter module. It takes a given object program and converts it to run at a new address. The most common use of it is to enable programs compiled on the second processor to be run on the I/O processor and to make programs on disc based machines run at &EOO. In the latter case, the program will be loaded at the normal default PAGE address and then moved down to &EOO. Since this overwrites the disc workspace area, you will not be able to use the disc system again. It is therefore advisable to make *TAPE the first statement in such a program, to ensure against accidental use of DFS commands, which would erase the program.
To use the relocate program, insert the utility disc and type *RUN RELOC. The program will prompt first for the filename of the object program to be changed, the new filename to be used for it and the new address required. For example:

```
>*RUN RELOC
G-CODE RELOCATOR

Object filename:GCODE
New object filename:GCODE2
Newaddress:&E00
>
```

# 6. Machine code converter theory

The machine code converter can be used to further increase the speed of G-code programs by converting them into 6502 machine code. It does this by intelligently substituting one or more machine code instructions for each G-code instruction in the object program.
The advantage of converting to machine code is that it increases the speed of the program by removing the need to individually interpret each G-code instruction. The main disadvantage is that conversion to machine code often makes a program three or four times larger than the original BASIC program.
Many G-code instructions are substituted by a few 6502 machine code instructions, but some of the more complex ones are implemented as subroutines. In this case a `JSR` instruction is used whenever the G-code occurs. All the subroutines required in this way are kept in a file called a 'library' which the converter incorporates into every program it generates. In order to provide all the routines that could be required, the library can be as big as 6K. However, if a program uses only a few of the subroutines, it can use one of the smaller libraries supplied on the utility disc.
Thus, the machine code converter combines a direct machine code version of the G-code program with a library of subroutines to produce the final program. The advantage of this approach is that since the library is integrated into the machine code program, the finished program can be run on a machine not equipped with Accelerator.

# 7. Machine code converter restrictions

Like the G-code compiler, the machine code converter will only translate certain parts of BBC BASIC. The restrictions include those imposed by the G-code compiler, with some new ones added. This section describes those extra restrictions.
The most significant difference is that the machine code converter will not translate real arithmetic. This means that you cannot use real variables (like `ANSWER=RESULT`) nor can you include real calculations in an expression, like `A%=B%/5` (in this case you would use `DIV` instead of /). It follows that functions that use or give a real result are either not translated or are treated differently. The complete list of banned functions is:

`ACS, ASN, ATN, COS, DEG, EXP, INT, LN, LOG, PI, R AD, SIN, SQR` and `TAN.`

The `RND` function can always be translated, except when `RND(0)` or `RND(1)` is used, which gives a real result and hence cannot be translated. Using `RND` with a variable argument equal to one or zero is permissable, but the result will De rounded down to an integer in the same way as under the G-code system.
The second significant difference arises out of the fact that BBC BASIC normally stores integers in four bytes or 32 bits, whereas the machine code converter uses two byte or 16 bit integers. This reduces the range of integers from +/- 2,147,483,647 to +/- 32,767. This can create problems in situations where a given expression (such as `100*100`) can be stored as an integer _ under the G-code system or normal BBC BASIC, but will overflow under the machine code converter.
Advanced BASIC programmers might wish to refer to addresses above &7FFF. To do this you need to know how negative numbers are implemented internally by the machine code converter. Since it uses 16 bit integers, the actual range of values is zero to 65,535. To make provision for negative numbers, the convention is that numbers over 32,767 are treated as though they were negative. For example, 65,535 is treated as -1, 65,534 as -2 and 65,525 as -10. The reverse relationships are also true -the value -1 is stored as 65534 etc. Thus, you can access addresses above 32767 using the formula 'new-address=old-address-65535'. For example, the address of the 6845 video controller in the I/O processor (normally written as &FEOO)

comes out as -511. When hexadecimal numbers over &7FFF are entered, this conversion is carried out automatically. This means that `PRINT &FE00` would actually print -511. So, in essence, you can access the top half of memory by writing the address in hexadecimal. It is important to be aware that this provision is really only a trick, and so must be used with care.
There is a nasty side effect of all this when the `ADVAL` function is used to read analogue values. It normally returns a positive 16 bit value. Under the machine code Converter, any values over 32767 will appear to be negative. The only way around the problem is to use a function like this:

```
DEF FNadval(Channel%)
Result%=ADVAL(Channel%)
IF Result%<&7FFF THEN=Result% DIV 16
ELSE=4095+Result% DIV 16
```

Using this function, `ADVAL` returns a 12 bit value from zero to 4,095. Since this is actually the internal accuracy of the device, nothing but speed is lost by using this patch. If you are concerned about speed, the only solution is to write a machine code version of the function definition above.
A similar problem occurs with `USR`. Normally, the four bytes of the integer returned by `USR` contain the contents of the four 6502 registers after returning from the machine code subroutine. Since the value returned by `USR` can only be two bytes in machine code, you may well have to extend machine code subroutines to return results in some other way. The most significant of the two bytes contains the X register and the least significant contains the accumulator. This is the same as the lower two bytes of the standard `USR` call.
Both BBC BASIC and the G-code system store the resident integer variables in the same place (page four of 6502 RAM). This allows communication between programs in the two languages. The machine code converter stores the resident integer variables in zero page, which makes them rather faster to access. This difference does not present any problems except when transferring values between programs by using the resident integer variables. (The way around the problem is to use the ! operator in the machine code program to extract the variable values directly from memory).
`CHAIN` operates slightly differently; it has been implemented to operate in the same way as the operating system `*RUN` command. This allows it to be used for chaining between machine code programs.
As discussed in the section on G-code restrictions, Accelerator assumes a user defined function is going to return a real result unless you append `a%` or `a$` to the end of the function name when you call it. Since the machine

code converter can not cope with real arithmetic, functions can only return integer or string results. Thus, you MUST append one or other of the symbols on the end of each function name as it is called.
GOSUB and GOTO may not be used with computed destinations or in an ON...GOTO/GOSUB construction.
The ESCAPE key is handled slightly differently in the machine code converter. In BASIC and G-code the system checks whether the ESCAPE key has been pressed after executing every statement or G-code. In the machine code system, this would entail riddling the program with instructions to check the ESCAPE key. Instead, it is turned off with *FX229 just before the main program is run and turned back on before returning to BASIC at the end of the program. This means that if you want to write a program that can be interrupted with ESCAPE you'll have to check for it yourself:

```
10 *FX 229,0 - Turns the ESCAPE key back on
20 REPEAT
30 PRINT "Escape hasn't been pressed yet"
40 UNTIL (?&FF AND &80)
50 *FX124
60 PRINT "Escape has been pressed"
```

The reference to location &FF reads the ESCAPE key flag'. Strictly speaking, it is not considered good practice to do this, but all Acorn BBC Micro BASICs do it and it works on the tube.
Although this method is slightly laborious, it does afford a degree of protection by allowing you to choose when and where you want to check for ESCAPE.
Error handling itself is also rather different under the machine code converter. After the program has been converted, all line number information is lost. This means that when an error does occur, the computer cannot discover what line caused the error. Instead, it just prints the error message itself and then stops. You then have to work out whereabouts the error occurred yourself. You can do this by seeing how far the program got before it stopped, either by examining the screen or any files the program may have created. As a last resort, insert lines like PRINT "This is line 27640" around where you suspect the error to be. Then re-convert and re-execute the program. You can then start to narrow down the cause of the error. If the error also occurs in the G-code or BASIC version of the program, you would be better off trying to debug that. Generally, the only reason a program might not work under the machine code converter even if it works perfectly under the G-code system

is that you have failed to take into account the differences between G-code and machine code.
In view of the above, the function ERL is not implemented. Some of these restrictions lead to very subtle problems which are not reported by the converter. For example, when the author was testing Accelerator, he tried to translate a program that contained the following line:

```
FOR T%=0 TO 1023 STEP 4
:T%!&7C00=T%!&7800:NEXT
```

This line was used to copy a screen full of MODE 7 graphics from a buffer at &7800 into the actual screen at &7C00. The program operated perfectly under BBC BASIC and G-code, and seemed to convert to machine code. The problem was that the screen looked a bit strange. After much head scratching, the error was pinned down and corrected. Since the machine code version of ! only works with two bytes at a time, the step size in the FOR loop should have been only two.
Another typical example would be:

```
IF (A%+B%)>C% THEN GOTO 4560
```

Normally this line will work perfectly. But imagine that A % is 30000, B% is 6000 and C% is 200. This time the program will not work, since A%+B% computes to 36000. Since this is over 32767, the system treats it as the negative number -29535 (which is 36000-65535). Thus, the > operator will report an incorrect result, possibly causing havoc. In this example, the error message, if any, will probably occur miles away from the line in question, making it very difficult to track down.
Problems like these can be difficult to solve if you are not familiar· with the internal operation of the program under conversion. It really is just a matter of acquired skill.

# 8. Using the machine code converter

The first stage in using the machine code converter is to decide which library to use. This decision is based on two criteria: the address where you want the finished code to be run and the complexity of the program under conversion.
Under normal circumstances, the address you choose to run the code at will be the default PAGE address for your machine. Typically, on a cassette based BBC Micro this will be &E00, rising to &1900 when discs are installed and dropping back to &800 if a second processor is being used. Using the default PAGE address gives you the maximum space for the code. You may wish to generate code for a slightly higher address to leave space for machine code subroutines, or data storage. The address must be a multiple of 256, like PAGE.
The complexity of the program under conversion affects the choice of library because simpler programs can save space by using a smaller, pared down, library. The standard level 3 library supports the following:

*commands, ABS,ADVAL,ASC,BGET,BPUT#,CALL,CHR$,CLEAR, CLG,CLOSE#,CLS,COLOUR,COUNT,DATA,DEF,DIM,DRAW,END, ENDPROC,ENVELOPE,EOF#,ERR,EXT,FALSE,FN,FOR,GCOL,GET, GETS,GOSUB,GOTO,HIMEM,IF,INKEY,INKEY$,INPUT, INPUT#, INSTR,LEFT$,LEN,LET,LOCAL,MID$,MODE,MOVE,NEXT,ON ERROR, OPENIN,OPENOUT,OPENUP,OSCLI,
PLOT,POINT,POS,PRINT,PRINT#,PROC,PTR#,READ,REPEAT, REPORT,RESTORE,RIGHT$,RND,SGN,SOUND,SPC,STOP, STR$, STR$~,STRING$,TAB,TIME,TRUE,UNTIL,USR,VAL,VDU,VPOS and WIDTH.

!,?, *, -, +, DIV, MOD, EOR, AND, OR, NOT, <, >, =, <>, <= and >=

The next level down, level 2, supports everything in level 3, except:

```
INSTR, STRING$, READ, DATA, RESTORE, PRINT#, INPUT#,
PROC, FN, DEF, ENDPROC, LOCAL
```

The lowest level only supports the following:

```
*commands, ABS, ADVAL, CALL, CLEAR, CLG, CLS, COLOUR,
DIM, DRAW, END, ENVELOPE, ERR, FALSE, FOR, GCOL, GET,
GOSUB, GOTO, HIMEM, IF, INKEY, LET, MODE, MOVE, NEXT,
ON ERROR, PLOT, POINT, POS, PRINT, REPEAT, REPORT, RND,
SGN, SOUND, SPC, STOP, TAB, TIME, TRUE, UNTIL, USR,
VDU, VPOS and WIDTH.

!, ?, *, -, +, DIV, MOD, EOR, AND, OR, NOT, <, >, =,
<>, <= and >=
```

N.B. Strings are not supported in level 1.
Level 3 - approximately 6K long
Level 2 - approximately 4.5K long
Level 1 - approximately 3K long
The choice comes down to choosing the lowest level that still supports all the features you need. Although the level 3 library may seem big at 6K long, under many conditions you can at least start off using it, only switching to one of the lower level ones if memory becomes a problem.
Having selected the target address and the library level required, combine the two into a single filename. For example, a level 2 library for a target address of &1900 would become `L2&1900`.
Then catalogue the utility disc and see if a file of the correct name is on the disc. If the file is present, you have picked one of the pre-defined libraries. Otherwise, you'll have to create the library.
To create a library, run the program called `LIBGEN` (which is on the utility disc along with the machine code converter):

```
CHAIN "LIBGEN"
```

The library generator will prompt you for the address and level of the library you require, followed by the drive number where the finished library should be stored .
The library generator will then construct the library from the existing libraries on the disc.
Having obtained the library file you require, you can run the machine code converter:

```
*RUN "CONVERT"
```

The computer will respond with:

```
MACHINE CODE CONVERTER
G-code filename:
```

Here, type the filename of the G-code program to be converted. The computer will then ask for the

```
Library filename:
```

Enter the filename of the library you selected. The computer will respond by asking you for the

```
Machine code filename:
```

This is the filename it will use to save the machine code. Enter a suitable filename and press RETURN.
The computer will finally ask:

```
Sideways ROM format (Y/N):
```

For the moment, press 'N'. The machine code converter will then start work. When it has finished, it will ask

```
Execute program (Y/N):
```

Pressing 'N' returns you to BASIC, and pressing 'Y' or RETURN will reload and execute the machine code.
You can load and run the machine code file with a *RUN command followed by the filename.
Advanced programmers can also execute machine code programs as subroutines from other environments. In this case, end the program with another RETURN statement. Then CALL or JSR the start address of the file. You should be aware that when control returns from the program, zero page will be totally corrupted. Thus, it is not practicable to execute machine code programs as subroutines from BASIC, unless they are stopped with an END statement, which carries out a *BASIC and so restores zero page.

# 9. Chaining between BASIC, G-code and machine code

The CHAIN command in BASIC,G-code and machine code can be used to transfer control to another program. The problem with CHAIN is that it can only be used to transfer to a program in the same language. For example, from a G-code program, CHAIN refers to another G-code program and from a BASIC program, it refers to another BASIC program.

There are, however, many circumstances when it is useful to chain between programs in different languages. For example, consider a program that draws a graphic pattern. Suppose that the pattern algorithm needs a table of the coordinates of 36 points on the perimeter of a circle with a radius of 500.

The best way of implementing this program would be to work out the circle table in G-code or BASIC and then switch to machine code to draw the actual pattern.

This method calls for two separate programs, which implies that some means of communicating between them is necessary. The simplest method is for the first part of the program to lower HIMEM and construct the table in the space freed. When the machine code program takes over, it simply sets HIMEM to the same address, and the table is accessible above HIMEM.

In these examples, I have assumed the first part of the program is to be compiled to G-code, but for a truly independent (of Accelerator) program, it could be left in BASIC.

Here is a listing of the complete program, starting with the G-code section:

```
10 MODE4
20 HIMEM=HIMEM-36*4-2
30 A%=HIMEM
40 FOR angle=0 TO 359 STEP 10
50 xdist=SIN(RAD(angle))*500
60 ydist=COS(RAD(angle))*500
70 !(HIMEM+(angle/10)*4)=xdist
80 !(HIMEM+(angle/10)*4+2)=ydist
90 NEXT angle
```

```
100 *RUN MACHINE
```

The machine code section could be:

```
10 MODE4
20 HIMEM=!&404
30 VDU 29,640;512;
40 FOR start%=0 TO 35
50 FOR end%=start% TO 35
60 MOVE!(HIMEM+start%*4),!(HIMEM+start%*4+2)
70 DRAW!(HIMEM+end%*4),!(HIMEM+end%*4+2)
80 NEXT end%,start%
90 END
```

The coding in both programs veers on the side of redundancy, which should help to make the programs slightly easier to understand. Here are notes on what the individual lines do, starting with the G-code program:
Line 10 clears the screen to `MODE 4`. This is done because the circle table has to be accessible when the screen is in `MODE 4`. If `MODE 7`, say, were used for this section, then the table would be built up below the `MODE 7` screen. In that case, when the next part of the program cleared the screen to `MODE 4`, the table would be wiped.
Line 20 moves `HIMEM` down to make room for the table. The table has 36 entries, each consisting of a pair of coordinates. Under the machine code converter, integers only take up 16 bits, so it makes sense to allot only two bytes to each coordinate, which makes four bytes for a pair of coordinates. The `-2` at the end of the line reserves an extra two bytes. This is done because `!` under G-code writes four bytes to memory at a time. Thus, when the very last coordinate is written to the table, there will be an 'overhang' of two bytes at the end. To prevent the two bytes doing any harm, they are reserved.
Line 30 sets the resident integer variable `A%` to the new value of `HIMEM`.
Line 40 starts a loop through the angles which feature in the table.
Lines 50 and 60 work out the horizontal and vertical component respectively of the circumference point in question. The formula used is a standard method for drawing circles. Notice that the circle is based around 0,0 and so some values will be negative.
Lines 70 and 80 put the values into the table. `HIMEM` is the base address of the table. `(angle/10)*4` gives the offset of the current coordinate pair. In line 80, the extra `+2` specifies the Y coordinate of the pair. It should be

realised that line 50 could be combined with line 70 and line 60 with line 80, which would probably be slightly faster.
Line 100 chains the machine code program using the *RUN command.
In the machine code program:
Line 10 clears the screen to MODE 4 again.
Line 20 restores the old value of HIMEM. Since BASIC and G-code's A% is not directly accessible from machine code, the ! operator has to be used to get the value directly from the memory location where A% is stored. Note that this is not strictly good programming practice, but since it works on the Tube and everything else, it seems permissable to do it.
The rest of the program draws the pattern itself. All it does is to join every point on the circumference of the circle with every other one, using straight lines. The resulting pattern is quite pretty. The only useful points in this section of the program is the way that the circle table is accessed in lines 60 and 70.
To sum up, *RUN can be used to chain from a BASIC or G-code program to a machine code program.
To chain from a BASIC program to a G-code program, use a section of code like this:

```
1000 *KEY 9 "*GRUN|MEXAMPLE|M"
1010 *FX 21
1020 *FX 138,0,137
1030 END
```

This example chains to a program called EXAMPLE, but you can change the program name by altering line 1000. The method used is to prime the keyboard buffer with the string of commands needed to switch into the G-code interpreter and execute a given program, and then stop the current program to let the commands be acted upon in BASICs immediate mode. Exactly the same method can be used to go from machine code to G-code. A slight alteration of the same routine will transfer control from a G-code program to a BASIC program:

```
1000 *KEY 9 "NCHAIN""EXAMPLE""|M"
1010 *FX 21
1020 *FX 138,0,137
1030 END
```

The 'N' at the start of the string in line 1000 answers the Re-execute (Y/N) : prompt given when a G-code program stops.

To go from machine code to BASIC, use exactly the same routine, but leave out the 'N'.

# 10. Developing sideways ROMs

This chapter describes how the machine code converter can be used to generate language ROMs from a G-code program.
The BBC Micro supports two types of ROM; language ROMs and utility ROMs. Utility ROMs provide extra operating system commands to be used from BASIC or direct from the keyboard. Typical utility ROMs include Disc Doctor, Caretaker and the Graphics Extension ROM. Language ROMs, as their name implies, provide new languages. BASIC itself is a language ROM, as are the two Accelerator ROMs. They don't have to contain languages - any program can be put into a language ROM. The machine code converter constructs language ROMs in such a way that your program can be entered with a simple operating system command. It outputs the ROM in the form of a file. To use it, you must either load the file into an EPROM programmer and blow an EPROM from it, or load the file in 'sideways RAM', as provided by many popular sideways ROM expansion boards. If you have a second processor, you can load the resulting file straight into it, and it will be treated as a language.
To generate a sideways ROM, you need to use a library created to work at an address above &8000. The address &8100 has been found to be ideal. The library can be any level.
Use the machine code converter normally, until it asks

```
Sideways ROM format (Y/N):
```

Reply in the affirmative by pressing 'Y' or RETURN. The converter will then ask you for the command to be used to invoke your program:

```
Invocation command:
```

Type a suitable command name and press return. Don't include the asterisk. You will then be asked to enter the title string. This is the string printed when the language is entered. For example Accelerator's two ROMs have the title strings `ACCELERATOR` and `G-CODE INTERPRETER`.

```
Title string:
```

The title string cannot be longer than a single line. The program then asks

you for the string it should print in response to the *HELP operating system command:

HELP message:

The HELP message must be a single line of text.
Once all this information has been accepted, the system will convert the program in the normal way. When the process has been completed, you will not be asked whether you wish to run the code. The only way the program can be run is to load it into sideways RAM, an EPROM programmer or the second processor.
To run the language file from the second processor, type *RUN name, where name is the name of the machine code file. The 'ROM' will be loaded and executed as a language directly.
Once you have the finished ROM installed in a machine, the program can be run with the operating system command you specified, regardless of whether the machine has Accelerator installed.
To do anything more sophisticated with sideways ROMs, the best method is to produce a normal machine code conversion to run at &8400 (for example), and then to write a ROM header yourself to fit between &8000 and &83FF. The ROM header can JMP to &8400 to execute the main program.

# 11. Technical information

This section describes the environment under which G-code and machine code programs are executed. None of the information in this section is actually necessary in order to use the compiler under normal circumstances, but advanced programmers will find it useful for specialist applications.
A word of warning: While we will make every effort to make future releases of Accelerator compatible with this information, we cannot guarantee it.
The layout of memory when a G-code program is being executed is as follows:

## Zero page

Locations &00 to &85 inclusive are used by the G-code interpreter. Strictly speaking, then, only locations &86 to &8F are free for use by machine code subroutines. However, locations &73 to &85 inclusive are only used to build up the parameters for `SOUND` and `ENVELOPE` statements. Thus, if you bear in mind that these locations will be altered when a `SOUND` or `ENVELOPE` statement are used, you can use them with impunity for your own purposes.
This list gives the uses of selected zero page locations:
&00 and &01 - Hold `OSHWM` They are set with OSBYTE &83 at initialisation and are not changed afterwards. The interpreter might use memory below this address if a G-code program needs to be executed at a lower address.
&02 and &03 - Hold the address of the top of the dynamic variable storage area. Memory above the G-code program up to the stack pointer is used to store strings and arrays. These locations hold the address of the next free location in this area. &04 and &05 - Hold the address of the stack pointer. This is initialised to `HIMEM` and moves down in memory.
&06 and &07-Hold `HIMEM`.
&08 and &09 - Hold the line number of the most recent error.
&00 to &11 - Hold the random number generator seed.
&IE - Holds `COUNT`.
&23 - Holds `WIDTH`.
&37 and &38 - Holds the address of the next G-code to be executed.

## Pages 4 to 7

Locations &400 to &7FF are used primarily for the purposes outlined below:

&400 to &46B - Hold the values of the resident integer variables. These are the same locations as BASIC uses, so they may be used to transfer values between G-code programs and BASIC programs.
&46C to &4BF - Used for temporary real workspace.
&4CO to &4FF - Used primarily for building up control blocks for operating system calls.
&500 to &5FF - Holds various system stacks. &600 to &6FF - The string buffer.
&700 to &7FF - Used as a keyboard buffer in the INPUT statement.
When a machine code program is executing, memory is used as follows:

# Zero page

&10 - Holds `COUNT`.
&11 - Holds `WIDTH`.
&12 and &13 - Hold `HIMEM`.
&14 and &15 - Hold the stack pointer.
&19 to &10 - Hold the random number generator seed.
&20 to &2F - Used as a buffer by `SOUND` and `ENVELOPE`, and thus not used most of the time.
&38 to &6F - Hold the resident integer variables `@%` to `Z%`, with two adjacent bytes for each variable.

# Pages 4 to 7

&480 to &4FF - Theoretically used as a buffer for operating s\'stem calls. In real life, only the first 30 or so bytes will ever be used.
&500 to &55F - Used for various internal stacks.
&560 to &5FF - Not used.
&600 to &6FF - String buffer.
&700 to &7FF - Keyboard buffer.

# 12. Keyword summary

**-**
G-code: Standard.
M-code all levels: Integer only.

**+**
G-code: Standard.
M-code level 3: Integer and string only.
M-code level 2: Integer and string only.
M-code level 1: Integer only.

**?**
G-code: Standard.
M-code all levels: Standard.

**!**
G-code: Standard.
M-code all levels: Reads or writes two bytes at a time, rather than four.

**5**
G-code: Standard.
M-code all levels: Not implemented.

*****
G-code: Standard.
M-code all levels: Integer only.

**/**
G-code: Standard.
M-code all levels: Not implemented.

**= (when used as an operator)**
G-code: Standard.
M-code level 3: Integer and string only.
M-code level 2: Integer and string only.
M-code level 1: Integer only.

### <>
G-code: Standard.
M-code level 3: Integer and string only.
M-code level 2: Integer and string only.
M-code level 1: Integer only.

### <
G-code: Standard.
M-code level 3: Integer and string only.
M-code level 2: Integer and string only.
M-code level 1: Integer only.

### >
G-code: Standard.
M-code level 3: Integer and string only.
M-code level 2: Integer and string only.
M-code level 1: Integer only.

### <=
G-code: Standard.
M-code level 3: Integer and string only.
M-code level 2: Integer and string only.
M-code level 1: Integer only.

### >=
G-code: Standard.
M-code level 3: Integer and string only.
M-code level 2: Integer and string only.
M-code level 1: Integer only.

## ABS
G-code: Standard.
M-code all levels: Integer only.

## ACS
G-code: Standard.
M-code al1levels: Not implemented.

## ADVAL
G-code: Standard.
M-code all levels: See the machine code restrictions chapter.

## AND
G-code: Standard.
M-code all levels: Standard.

## ASC
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## ASN
G-code: Standard.
M-code all levels: Not implemented.

## ATN
G-code: Standard.
M-code all levels: Not implemented.

## BGET#
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## BPUT#
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## CALL
G-CODE :Standard.
M-code all levels: Standard, except for the use of 16 bit variables. Thus parameter type 4 (User Guide, P.215) is now a 16 bit integer variable. Parameter type 5, meaning a real variable, is not implemented.

## CHAIN
G-code: Chains another G-code program.
M-code all levels: Chains another machine code program.

## CHR$
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## CLEAR
G-code: Standard.
M-code all levels: Standard.

## CLOSE#
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## CLG
G-code: Standard.
M-code all levels: Standard.

## CLS
G-code: Standard.
M-code all levels: Standard.

## COLOUR
G-code: Standard.
M-code all levels: Standard.

## COS
G-code: Standard.
M-code all levels: Not implemented.

## COUNT
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code-level 1: Not implemented.

## DATA
G-code: Standard.
M-code level 3: Standard.

M-code level 2: Not implemented.
M-code level 1: Not implemented.

## **DEF** (see also FN)

G-code: Standard.
M-code level 3: Standard
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## **DEG**

G-code: Standard.
M-code levels: Not implemented

## **DIM**

G-code: Standard.
M-code all levels: Real arrays not implemented.

## **DIV**

G-code: Standard.
M-code all levels: Standard.

## **DRAW**

G-code: Standard.
M-code all levels: Standard.

## **ELSE**

G-code: Standard.
M-code all levels: Standard.

## **END**

G-code: Standard. Causes a return to the Re-execute (Y/N) : prompt.
M-code all levels: Standard. Causes a return to BASIC.

## **ENDPROC**

G-code: Standard.
M-code level 3: Standard.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## **ENVELOPE**

G-code: Standard.
M-code all levels: Standard.

## EOF#
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented .

## EOR
G-code: Standard.
M-code all levels: Standard.

## ERL
G-code: Standard.
M-code all levels: Not implemented.

## ERR
G-code: Standard.
M-code all levels: Standard.

## EVAL
G-code: Not implemented.
M-code all levels: Not implemented.

## EXP
G-code: Standard.
M-code all levels: Not implemented.

## EXT#
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## FALSE
G-code: Standard.
M-code all levels: Standard.

## FN
G-code: When a function is called, the compiler must be informed what type you expect the result to be. This is done by appending a % to the end of the name to indicate an integer result or a $ to indicate a string result. If neither symbol is added to the name, the compiler assumes you wish the function to return a real result. When the function returns, the G-code

interpreter tries to convert the result to the type you indicated.
For example:

```
10 PRINT FNanimal$
20 END
30 DEFFNanimal
40 LOCAL A$
50 INPUT "Enter an animal:"A$
60 IF FNnot-animal%(A$) THEN GOTO 50
70 =A$
```

M-code level 3: See above. Notice that since real numbers are not implemented, you must always use either % or $ with a function to indicate whether you want a numeric or string result.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## FOR

G-code: Standard.
M-code all levels: Integer only.

## GCOL

G-code: Standard.
M-code all levels: Standard.

## GET

G-code: Standard.
M-code all levels: Standard.

## GET$

G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## GOSUB

G-code: Standard, computed destinations (e.g. `GOSUB SELECT*1000+5000`) are all owed.
M-code all levels: Standard, except computed destinations are not allowed.

## GOTO

G-code: Standard. Computed destll1J:lOns (e.g. `GOTO CHECK*100+6500`) are allowed M-code all levels: Standard. (except computed destinations are not allowed.

## HIMEM

G-code: Standard.
M-code all levels: Standard

## IF

G-code: Standard.
M -code all levels: Standard.

## INKEY

G-code: Standard.
M-code all levels: Standard.

## INKEY$

G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## INPUT

G-code: Standard. Extended to allow hexadecimal input.
M-code level 3: Standard. Extended to allow hexadecimal input.
M-code level 2: Standard. Extended to allow hexadecimal input.
M-code level 1: Not implemented.

## INPUT#

G-code: Standard.
M-code level 3: Standard. Attempting to read a real number will result in an error.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## INSTR

G-code: Standard.
M-code level 3: Standard.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## INT

G-code: Standard.
M-code all levels: Not implemented,

## LEFTS
G-code: Standard
M-code level 3: Standard
M-code level 2: Standard
M-code level 1: Standard

## LEN
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## LET
G-code: Standard.
M-code all levels: Integers and strings only.

## LN
G-code: Standard.
M-code all levels: Not implemented.

## LOCAL
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## LOG
G-code: Standard.
M-code all levels: Not implemented.

## LOMEM
G-code: Not implemented.
M-code all levels: Not implemented.

## MID$
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## MOD
G-code: Standard.
M-code all levels: Standard.

## MODE
G-code: Standard.
M-code all levels: Standard.

## MOVE
G-code: Standard.
M-code all levels: Standard.

## NEXT
G-code: Standard.
M-code all levels: Integer only.

## NOT
G-code: Standard.
M-code all levels: Standard.

## ON ERROR
G-code: Standard.
M-code all levels: Standard.

## ON ... GOTO/GOSUB
G-code: Standard.
M-code all levels: Not implemented.

## OPENIN
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## OPENOUT
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## OPENUP

G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented,

## OR
G-code: Standard.
M-code all levels: Standard,

## OSCLI
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard
M-code level 1: Not implemented.

## PAGE
G-code: Not implemented.
M-code all levels: Not implemented.

## PI
G-code: Standard.
M-code all levels: Not implemented.

## PLOT
G-code: Standard.
M-code all levels: Standard.

## POINT
G-code: Standard.
M-code all levels: Standard.

## POS
G-code: Standard.
M-code all levels: Standard.

## PRINT
G-code: Standard.
M-code all levels: Mostly standard; since all numbers are integers there is no need for the variety of formats offered by `PRINT`. Instead, only the bottom byte of `@%` is taken into account to justify numbers. At level1, strings cannot be printed.

## PRINT#
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## PROC
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## PTR#
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## RAD
G-code: Standard.
M-code all levels: Not implemented.

## READ
G-code: Standard.
M-code level 3: Integers and strings only.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## REM
G-code: Ignored.
M-code all levels: Ignored.

## REPEAT
G-code: Standard.
M-code all levels: Standard.

## REPORT
G-code: Standard.
M-code all levels: Standard.

## RESTORE
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## RETURN
G-code: Standard.
M-code: Standard.

## RIGHT$
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard
M-code level 1: Not implemented.

## RND
G-code:
1) `RND` with no argument returns a random 32 bit integer
2) `RND(0)` returns the last random number generated as a real number between zero and one.
3) `RND(1)` returns a real random number between zero and one.
4) `RND(X)` , where X is zero, returns the most recent random number as a 32 bit integer.
5) `RND(X)`, where X is one, returns zero.
6) `RND(X),` where X is greater than one, returns a random integer between 1 and X.
7) `RND(X)` , where X is negative, restarts the random number generator.
M-code all levels: See above. Variations 2 and 1 cannot be translated and all integers are 16 bit.

## RUN
G-code: Standard.
M-code all levels: Standard.

## SGN
G-code: Standard.
M-code: Integer only.

## SIN
G-code: Standard.
M-code all levels: Not implemented.

## SOUND
G-code: Standard.
M-code all levels: Standard.

## SPC
G-code: Standard.
M-code all levels: Standard.

## SQR
G-code: Standard.
M-code all levels: Not implemented.

## STEP
G-code: Standard.
M-code all levels: Standard.

## STOP
G-code: Standard.
M-code all levels: Standard,

## STR$
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## STR$~
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Standard.
M-code level 1: Not implemented.

## STRING$
G-code: Standard.
M-code level 3: Standard.
M-code level 2: Not implemented.
M-code level 1: Not implemented.

## TAB
G-code: Standard.
M-code all levels: Standard.

## TAN
G-code: Standard.
M-code all levels: Not implemented.

## THEN
G-code: Standard.
M-code all levels: Standard.

## TIME
G-code: Standard.
M-code all levels: Only the bottom 16 bits of the TIME value are available.

## TO
G-code: Standard,
M-code all levels: Standard,

## TOP
G-code: Not implemented.
M-code all levels: Not implemented.

## TRACE
G-code: Not implemented.
M-code all levels: Not implemented.

## TRUE
G-code: Standard.
M-code all levels: Standard.

## UNTIL
G-code: Standard.
M-code all levels: Standard.

## USR
G-code: Standard.
M-code all levels: Standard. The 16 bit value returned has the X register in the most significant byte and the accumulator in the least significant byte.

## VAL
G-code: Standard, with an extension to allow hexadecimal numbers to be converted if preceded by a '&'.
M-code level 3: Integer only including the provision for hexadecimal numbers.

M-code level 2: Integer only including the provision for hexadecimal numbers.
M-code level 1: Not implemented.

## VDU

G-code: Standard.
M-code all levels: Standard.

## VPOS

G-code: Standard
M-code all levels: Standard

## WIDTH

G-code: Standard
M-code all levels: Standard

# 13. Error message summary

This section lists the error messages issued by the G-code compiler, the G-code interpreter, the machine code converter and the machine code run time system.

## The G-code compiler error messages

`Assembly Language`
Caused if assembly language is used.

`Bad DIM`
Caused by a badly formed DIM statement.

`Bad PROC/FN name`
Caused by declaring a procedure or function without a name.

`Bad call`
Caused by including too many parameters in a procedure or function call.

`Bad hex`
Occurs when & is used without any valid hexadecimal digits following it.

`Can't complete function`
Caused by using one of the functions or pseudo variables `PAGE, LOMEM, EVAL` or `TOP`.

`Escape`
Caused by pressing ESCAPE during compilation.

`FOR variable`
Occurs when a string or byte variable is used as a FOR loop index.

`File not found`
Occurs when the source filename you specified doesn't exist.

`Incorrectly formed source program`
Caused by not having a valid program at the default `PAGE` address or by trying to compile a file that isn't a BASIC program.

`Memory full`
Occurs when all available memory has been used up. If you are compiling from memory, the cure is to start compiling from disc/cassette. You can also try to reduce the size of workspace required by Accelerator by using shorter variable names and fewer line numbers.

`Missing "`
Caused by the omission of the terminating quote symbol of a string.

`Missing #`
Caused by the omission of the # symbol after `BGET` , `PTR` , `EXT` or `EOF`.

`Missing )`
Caused by omitting the closing bracket of an expression or after the parameter list of a function.

`Missing ,`
Caused by omitting the comma between parameters for statements and functions.

`Missing =`
Caused by the omission of the equals sign from an assignment statement.

`Missing PROC/FN after DEF`
Caused by omitting `PROC` or `FN` after `DEF`.

`Missing parameters`
Occurs when a procedure or function that requires parameters is called without any.

`Mistake`
Caused by omitting the equals sign when assigning a new value to `HIMEM` or `TIME`.

`No TO`
Caused by the omission of the word `TO` from a `FOR` loop.

`No such FN/PROC`
Occurs when an undefined procedure or function is called.

`No such array`
Caused by a reference to an array for which there is no corresponding DIM statement.

`No such line`
Caused by a reference in a `GOTO, GOSUB` or `RESTORE` statement to a non-existent line.

`No such variable`
Caused by a reference to a variable that is not defined in the program. For example, `PRINT here`, where the variable `here` is not defined elsewhere in the program.

`ON syntax`
Caused by using something other than `GOTO/GOSUB` or `ERROR` after `ON`.

`Redefined PROC/FN`
Caused by defining the same procedure or function at two or more points in a program.

`Redimensioned array`
Caused by attempting to declare an array in more than one `DIM` statement.

`Syntax error`
Caused by completely incomprehensible statements or attempting to use `TRACE, LOMEM` or `PAGE`.

`Too big`
Occurs if a real number is used in circumstances that require an integer and the number is too big to convert to an integer. For example, `PRINT EXT#(2.2334↑34.)`

`Type mismatch`
Caused by attempting an operation on an inappropriate type (e.g. `PRINT -"HELLO"`) or by attempting to assign a string value to a numeric variable or vice versa.

# The G-code interpreter error messages

The error number reported by ERR is given for trappable errors.

`-ve root-21`
Occurs if an attempt is made to take the square root of a negative number. This may also occur with ACS and ASN.

`Accuracy Lost-23`
Occurs if an attempt is made to calculate trigonometric functions with very large angles.

`Bad G-code`
An internal error meaning that the program under execution uses a G-code wrongly or attempts to use an undefined G-code. This can only occur if the program being run has been tampered with or was not produced by the compiler.

`Bad hex-28`
Occurs when & is used without any valid hexadecimal digits following it.

`Bad mode-25`
Occurs if there is not enough memory spare to go to the mode specified in a MOD E statement or if MOD E is used in a procedure or function.

`Bad program`
Given when running G-code direct from memory if there is no valid BASIC program at the default P AGE address.

`DIM space-11`
Occurs if there is not enough memory to allow an array to be dimensioned.

`Division by zero-18`
Occurs if an attempt is made to divide a number by zero.

`Escape-17`
Caused by pressing ESCAPE while the program is executing.

`Exp range-24`
Occurs when an attempt is made to evaluate a power greater than 88.

`Log range-26`
Occurs if an attempt is made to take the logarithm of a negative number or of zero.

`Memory full`
Occurs if all available memory has been exhausted.

`Missing"`
Occurs if the closing quote mark of a string is omitted in the `INPUT` or `READ` statements.

`No FN-7`
Occurs if a function terminator is executed outside a function definition.

`No FOR-32`
Occurs if a `NEXT` statement is encountered without a corresponding `FOR` statement having been previously executed.

`No G-code program`
Occurs If the G-code program in memory or pulled off disc/cassette was in some way badly formed.

`No GOSUB-38`
Occurs if a `RETURN` statement is encountered without a corresponding `GOSUB` statement having been executed previously.

`No PROC-13`
Occurs if an `ENDPROC` statement is executed outside a procedure definition.

`No REPEAT-43`
Occurs if an `UNTIL` statement is encountered without a corresponding `REPEAT` statement having been executed.

`No such array-14`
Occurs if an array is referenced before it is dimensioned,

`No such Line-41`
Occurs if the line number in a computed `GOTO, GOSUB` or `RESTORE` does not exist.

`Not LOCAL-12`
Occurs if a `LOCAL` statement is executed outside a procedure or function definition.

`ON range-40`
Occurs if the value of the expression in an `ON GOTO/GOSUB` statement is too large. The remedy is to include an `ELSE` clause at the end of the construction.

`Out of DATA-42`
Occurs if a `READ` statement is executed when no more `DATA` is available.

`Re-dimensioned array-10`
Occurs if an attempt is made to re-dimension an array.

`STOP`
Caused by execution of the `STOP` statement.

`Subscript too big-15`
Occurs if a subscript in an array access is too large.

`Too big-20`
Caused by a number becoming too large to be manipulated.

`Too Long-19`
Occurs if a string becomes longer than 255 characters, either through the `STRING$` function or by concatenation.

`Too many FORs-35`
Occurs if the `FOR` loop nesting limit of 10 is reached.

`Too many GOSUBs -37`
Occurs when the maximum `GOSUB RETURN` nesting limit of 26 is reached,

`Too many REPEATs -44`
Caused by nesting more than 20 `REPEAT/UNTIL` loops.

`Type mismatch -6`
Occurs if the type of a datum read by `INPUT#` does not match the type of the destination variable or if the type of the result of a function does not match the type specified when it is called.

# The machine code converter error messages

`Cannot trans Late G-code`
Occurs when the converter encounters a G-code which cannot be translated, even if a level 3 library is used.

`G-code not in Library`
Occurs when level 2 or 1 libraries are used and a feature only found in higher level 1ibraries is encountered. The solution is thus to use a higher level 1ibrary.

`Mismatched REPEAT-UNTIL`
Occurs at the end of a program if there is an outstanding `REPEAT` for which no matching `UNTIL` has been encountered.

`Out of memory`
Occurs when the converter runs out of memory, usually because the program under conversion uses too many variables. The usual solution is to split the original program into subsections and chain between them. If this is not practicable, you can try to cut down on the overall number of variables by using the same variable for as many different purposes as possible.

`Real arithmetic`
Occurs when the converter encounters real arithmetic, variables or real `INPUT`s, `DIM`s or `READ`s, non of which can be translated.

`Too many REPEATs`
Occurs if an attempt is made to nest more than 20 `REPEAT/UNTIL` loops.

`Too many dimensions`
Occurs if an attempt is made to dimension an array with more than 15 dimensions.

`UNTL without REPEAT`
Occurs if `UNTIL` is encountered without a corresponding `REPEAT` having been processed.

# The machine code run time system error messages

`Badmode-25`
Occurs when an attempt is made to change display mode inside a procedure or when there is not sufficient memory to allow the selected mode to be used.

`Can't match FOR-&21`
Occurs when the variable given in a `NEXT` statement cannot be matched to a corresponding `FOR` statement.

`DIM space-11`
Occurs when there is not enough room to dimension an array.

`Division by zero-18`
Occurs when an attempt is made to divide a number by zero.

`HIMEM too low-47`
Occurs when an attempt is made to set HIM E M below the top address currently used for dynamic variable storage.

`No FN-7`
Occurs when a function termination (e.g. `=RESULT%`) is encountered outside a function definition.

`No FOR-32`
Occurs when a `NEXT` statement is encountered without a corresponding FOR statement having been executed.

`No PROC-13`
Occurs when `ENDPROC` is encountered outside a procedure definition.

`No such Line in RESTORE-41`
Occurs when a computed `RESTORE` is attempted to a non-existent line.

`Not LOCAL-12`
Occurs when `LOCAL` is encountered outside procedure or function definitions.

`Out of DATA-42`
Occurs when an attempt is made to use `READ` when no more `DATA` can be read.

`Re-dimensioned array-10`
Occurs when an attempt is made to re-dimension an array.

`STOP`
Occurs when the `STOP` statement is executed.

`String too Long-19`
Occurs when a string becomes longer than 255 characters, either through the use of `STRING$` or concatenation.

`Subscript too Large-15`
Occurs when an attempt is made to access an array where one of the subscripts is larger than the corresponding dimension in the `DIM` statement.

`Too big-20`
Occurs when the result of an arithmetic computation is larger than 32,767.

`Too many FORs-35`
Occurs when the maximum `FOR` loop nesting depth of 10 is reached.

`Type mismatch-6`
Occurs when `INPUT#` encounters a real number, or attempts to read a string into an integer or vice versa, Can also be caused by the 'wrong' type being returned by a function.

`Undimensioned array-14`
Occurs when an attempt is made to access an array before it has been dimensioned.

# 14. Specifications

Accelerator is a BBC BASIC to pseudo-code code compiler, a pseudo-code interpreter (both in ROM), a pseudo to native code converter and utilities (all on disc). The pseudo-code is a specially designed system called G-code.

## The BBC BASIC to G-code compiler

This module takes a BBC BASIC source program from disc/cassette or direct from memory and compiles it to produce a G-code object program, which it optionally saves to disc/cassette.
The compiler will compile all of BBC BASIC version II with the following differences:
`LOMEM, PAGE, TOP, EVAL, TRACE`, assembly language Not implemented
`VAL, INPUT` - Extended to accept hexadecimal numbers
`RND, FN` - Rationalized for predictable type conversion
`CHAIN` - Loads and runs another G-code program
When compiling from memory, there must be enough room for the original BASIC program, the G-code program and workspace. This generally limits programs to about 10K in `MODE 7`. When compiling from disc/cassette, there need only be enough room for the G-code program and workspace, which allows programs up to about 25K to be compiled.

## The G-code interpreter

This module takes a G-code object program from disc/cassette or direct from memory and executes it. To do so, it moves the code to the correct execution address and repeats the cycle of 'get the next G-code byte; call the routine associated with it'. This cycle is halted either by an error or the end of the program

# The machine code converter (native code generator)

This module takes a G-code object program off disc and converts it into machine code, combining it with a specified run-time library file. The finished code is output to another file. It can optionally generate machine code in sideways ROM format, which allows the program to be executed from sideways ROM/RAM.
Since the machine code converter uses a G-code file as its starting point, the restrictions that apply to the BBC BASIC to G-code compiler apply by extension to the machine code converter. There are several additional differences:
Floating point variables, arithmetic and functions - Not implemented
Integers - Implemented in 16 bits
RND - Only integer variants may be used
Resident integer variables - Moved from page 4 to zero page
CHAIN - Loads and runs another machine code program
ESCAPE - No longer automatically trapped Error handling - Line numbers of errors are not available
There is no intrinsic limit on the size of program that can be converted.

# Utility - G-code relocator

This program takes a G-code program from disc/cassette and relocates it to give it a different execution address.

# Utility - Library generator

This program uses the standard libraries supplied on the utilities disc to construct a new library to your specifications.