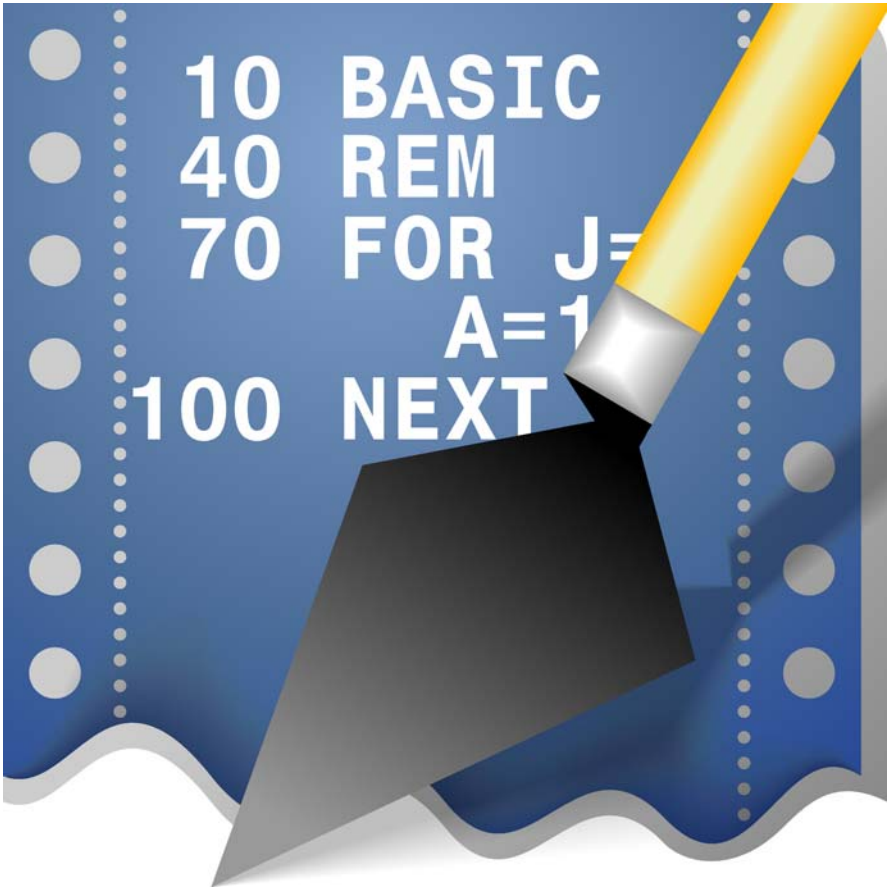
Archimedes
BASIC Compiler



Issue 1 published by Oak Solutions Ltd.

Issues 2 and 3 published by RISC OS Open Ltd.

The product described in this manual is not intended for use as a critical component in life support devices or any system in which failure could be expected to result in personal injury.

The product described in this manual is subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by the publisher in good faith. However, the publisher cannot accept any liability for any loss or damage arising from the use of any information or particulars in this manual.

If you have any comments on this manual, please complete the form at the back of the manual and send it to the address given there.

Published by RISC OS Open Ltd.

Issue 1 for ABC 3, November 1991 (Oak Solutions edition).
Issue 2 for ABC 4, November 2016 (updates by RISC OS Open Ltd).
Issue 3 for ABC 4, October 2019 (updates by RISC OS Open Ltd).

Contents

1 Introduction

The Archimedes BASIC Compiler forms part of the Desktop Development Environment, giving developers the option to quickly produce applications and modules in the familiar BBC BASIC language built in to every version of RISC OS.

Interpreters and compilers

All versions of Acorn's BBC Microcomputer and all machines which run RISC OS have been supplied as standard with a version of BBC BASIC which runs as an *interpreter*. An interpreter is a program which reads the program line by line, analyses and evaluates the instructions, then executes them immediately.

The alternative to an interpreter is a *compiler*. A compiler also reads the whole program, but without execution. Instead, the instructions of the input program (usually called the *source*) are converted into machine code and the resultant output, the object program, is normally saved as a separate file.

This compiled program can then be executed just as if it had originally been written in machine code, rather than a 'higher level' or more user friendly language. With improved processors, more memory, and larger disc space this is less important than it used to be. ABC still has a valuable role as a means of packaging BASIC software for release in a way that deters trivial observation of the program code.

Interpreters have the advantage that a program can be typed in and run straight away without any extra operations. However, they also have two major disadvantages:

- The first is that, especially for large programs, you can never be absolutely sure that a program is free of syntax errors. These will be found only if the interpreter tries to evaluate the instructions in which they occur. If, when a program runs, a particular path through the code is not followed, possible errors in that area of the program remain undetected. Since a compiler always processes and evaluates every instruction, such errors are always discovered immediately.

- The second disadvantage of interpreted programs is that the speed of execution is much lower than for compiled programs. This is because each statement must be re-interpreted every time it is executed. Under a compiled system, most of this analysis is done only once - during the compilation - and so does not waste time when the compiled program executes.

The ideal is to have both an interpreter and a compiler. The interpreter is used to develope and test logic and debug the algorithms, while the compiler fulfils two roles. It can be used initially to detect syntax errors, and finally, when development is complete, to produce the finished compiled program. The Archimedes BASIC compiler has been produced to provide you with this ideal development tool for your RISC OS programs.

Compatibility

ABC works on all versions of RISC OS from 3.10 upwards, and with 26 or 32 bit processors.

For compatibility with 32 bit processors you must recompile with ABC version 4.10 or later and use ABCLibrary version 4.12 or later also. This is due to changes in the linkage scheme between the application and ABCLibrary. Programs compiled with this combination will run on 26 or 32 bit platforms with no modifications necessary.

2 Installation

We detail installation for both the developer using ABC and an end user wishing to use the resulting software created.

Elements required

!ABC

The main Archimedes BASIC Compiler product.

Find this in the `Apps.DDE` directory.

Developer !System

A collection of modules required by developers using ABC.

Find this in the `AcornC/C++.Developer` directory.

End user !System

A smaller collection of modules required by users wishing to run software created by ABC.

Find this in the `AcornC/C++.EndUser` directory.

Examples

A directory containing examples referred to in this text.

Find these in the `Sources.DDE-Examples.ABC` directory.

ABC is designed for use under the RISC OS desktop. On the distribution disc, the software is present in a ready-to-run form. However, it is strongly recommended that you should install the system onto a day-to-day working disc. You should then put the original distribution disc away in a safe place.

Programs compiled with ABC, which includes ABC itself, make use of a *run time library* called ABCLibrary. This library is encapsulated in a RISC OS module. Only one copy of ABCLibrary needs to be loaded at once, which can serve any number of ABC applications or modules.

Installation

Copy the `Apps.DDE` folder to your day-to-day working disc in its entirety. On a standard RISC OS harddisc layout there will already be an Apps directory present; this update will add the Desktop Development Environment (DDE) directory within it.

Copy the `Sources.DDE-Examples.ABC` to your day-to-day working disc too. Keep this somewhere convenient as later chapters in this Guide refer to the examples in several places.

Merging the developer System modules

Do not use a Filer copy operation to copy the !System directory over your existing one, as this would overwrite any newer versions of modules that have been released since the ABC distribution disc was created.

An obey file called `!SysMerge` is provided to safely merge the newer resources with your existing System. ABC works with RISC OS 3 and later and the System update is similarly compatible.

Your licence does not permit the developer System to be redistributed.

Merging the end user System modules

For software created with ABC you will need to provide end users with a copy of ABCLibrary and a handful of other System updates. Use the end user version for this purpose.

Instructions are provided in an accompanying text file for how to install these resources which varies slightly between different versions of RISC OS. As a developer, the new modules will already have been installed in the earlier instructions for merging the developer System modules, so no further action is needed.

Your licence permits you to include this end user System with any software you produce using the Desktop Development Environment.

3 Getting started

Once ABC has been installed onto either a working disc or your hard disc, you will need a BASIC program to compile to try it out. This chapter works through the process of getting started.

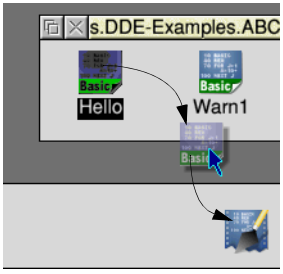
Compiling a program

A very simple example called `Hello` is provided in the `Examples` directory. This consists of the following:

```
10 REM >Examples.Hello
20 PRINT "Hello"
```

You are now ready to find out how easy the compiler is to use.

To compile the program, open a directory viewer onto the `Examples` directory and drag the `Hello` icon from the directory viewer and drop it on top of the ABC icon on the icon bar:

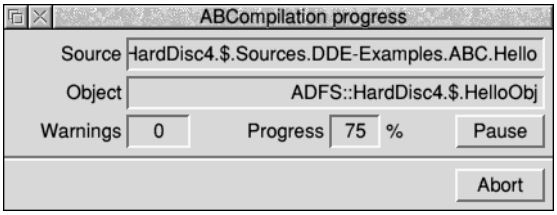


The following will appear:

This represents the object file which you are about to create.

You can, if you wish, alter the name of this file from the default, `Object`. To do this, press the Delete key until the writable icon is empty and then type in the name which you want to use, for example `HelloObj`.

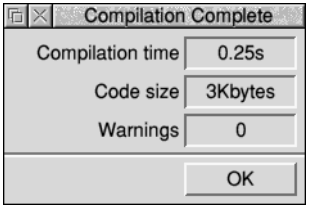
Click select to drag the object code icon from the dialogue box to a directory viewer where you want the file to be created. A good place to save it would be in the `Examples` directory alongside the source. When you drop the icon in the directory, the compilation will start and the following window will appear:



This confirms the *path names* of the source and object files and shows the progress through the compilation in terms of both the number of problems which have been found so far and the percentage of the compilation which has been completed.

Click **Abort** if you wish to halt compilation.

The compilation will take just a few moments, after which the following will be displayed:



This verifies that the compilation has been completed successfully and provides information about the time taken and the size of the object code produced. The object code always contains a *header block* of about 2 kbytes. Hence the length of the object will always be at least this size.

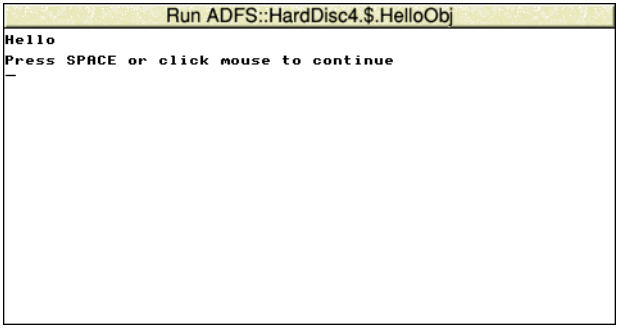
Executing the code

The object code produced during the compilation is saved using the name you provided earlier. Therefore you should be able to see an application icon called `HelloObj` (or whatever you chose to call it) in the `Examples` directory viewer.

To execute the code, double-click on this icon. A command window will be opened and the output from your code, the string “Hello”, will be printed in it. Then, when execution is complete, the message

```
Press SPACE or click mouse to continue
```

will appear:



ABCLibrary

To execute the object code produced, ABCL*ib*, the ABC library module, must be active. This library is automatically loaded when the compiler is initialised. Therefore, if you compile a file immediately before running the object code, you won't have anything to do.

However, if you reset your computer and then double-click on the object code icon to execute the code, you will get the error message:

```
ABCLib not loaded
```

You need to double-click on the ABCLib module before executing this code.

You may find it convenient to create an application directory to hold your object code. Then you can add the following line to the `!Run` file so that it automatically loads the ABCLib module for you:

```
RMensure ABCLibrary 0.00 RMLoad System:Modules.ABCLib
RMensure ABCLibrary 4.12 Error An older version of ABCLib is already loaded
```

This assumes that ABCLib is included in !System which is where you are advised to place it. Notice how the two commands are used to firstly load the library if no library is currently loaded, and secondly to take special care not to load the module if it is already loaded. Loading a newer library when an older one is already in use with other clients could result in system instability.

If you provide an application to someone else, you should include the end user !System resource holding ABCLib on the disc and advise the recipient to install as described in the instructions for M*erging the end user* S*ystem modules* on page 4.

Warnings and errors

If the compiler discovers a problem with your BASIC program then it will stop and display either an error message or a warning. A large range of different messages have been implemented to allow the description of the problem to be as specific and meaningful as possible.

- **Errors** are the more serious. After an error the compiler is unable to continue generating code.
- **Warnings** are given when there is something wrong with the code but the compiler has been able to make a guess at what you meant or ignore the incorrect statement. The compiler will continue generating code after a warning, however the object code might not do what you intended!

The compiler attempts to avoid throwing errors if it can. Notably when undefined variables are found each will generate a warning rather than an error, but an error will be thrown at the end of compilation. This allows you to get substantially further through the compilation at each attempt.

However, what will sometimes happen is that the number of problems reaches the point where the compiler cannot continue and the compilation must be aborted for errors to be fixed. The **Pause** option must be turned on to see these warnings during compilation.

Warnings

To see a warning, try compiling the `Warn1` file in the `Examples` directory. This contains the following program:

```
10 REM >Examples.Warn1
20 A$ = "Hello : REM a simple message
30 PRINT A$
```

The error is on line 20 - a double quote is missing from the end of the string.

Reporting using throwback

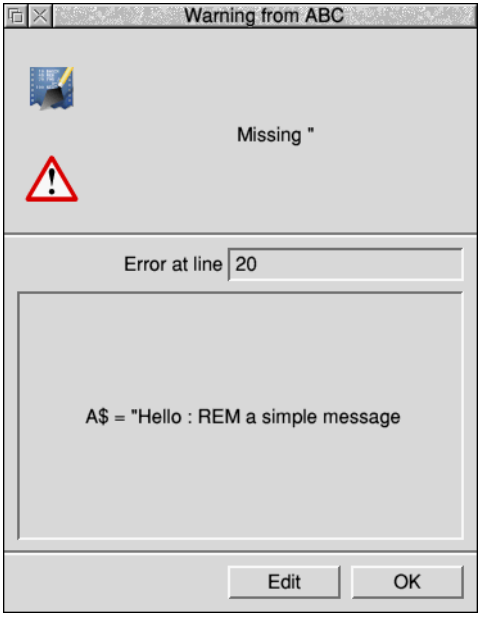
ABC will attempt to report warnings and errors via the *throwback protocol* used by other compilers and assembler tools in the Desktop Development Environment.

This facility requires the DDEUtils module to be loaded and a throwback-capable editor, such as SrcEdit, which can also load tokenised BASIC programs. If you double-click on a warning/error in the throwback window then the editor will open the source file on that line.

It is possible to disable throwback with a directive as desribed in *Throwback suppression* on page 47.

Reporting without using throwback

If ABC is unable to use throwback it will open its own error reporting window which will report the line, nature of the problem, and line number:



Having obtained a warning, you have two alternatives. You can either click on **OK** to continue with the compilation or click on **Edit** to load the source program into your chosen BASIC editor.

Choosing OK

If you choose the former option then the compiler will continue with the compilation, having guessed that you meant the missing double quote to be at the end of the line. When the compilation is complete, the dialogue box produced will confirm that 1 warning was issued during the compilation.

If you try running the code produced by double-clicking on the object file, the following will be printed:

```
Hello : REM a simple message
```

Although object code has been produced, it does not do what was intended, which was to print “Hello”. Hence you should be very wary of the code produced by a compilation which generated a warning.

One reason for continuing with the compilation is that it allows you to detect further problems using just a single run of the compiler.

However, a note of caution is required here as well. Any subsequent warnings may be spurious. They may be generated because the compiler guessed incorrectly at the cause of the first problem. For example, try compiling `Warn2` which contains the following:

```
10 REM >Examples.Warn2
20 FOR I% = 1 TO 10
30 PRINT I%
40 NET I%
50 FOR J% = 1 TO 10
60 PRINT J%
70 NEXT J%
```

This contains a single mistake - the NEXT on line 40 has been mis-spelt. However, when line 40 is reached, the warning generated is:

```
= expected
```

This is because the compiler makes the (incorrect) assumption that NET is a variable which is being assigned the value I%. Then, when the last line is reached, the compiler still thinks that the first FOR loop hasn't been ended and so produces the message:

```
Unclosed FOR loop
```

Choosing Edit

This option will attempt to load the source program into an editor which can load BASIC programs, such as SrcEdit provided with the Desktop Development Environment. The editor must already be running for this feature to work.

Errors

An error is a problem which is serious enough for the compiler to be unable to guess what you meant. Therefore, compilation is unable to continue after an error has occurred.

The best course of action is to immediately correct the program and start again.

With throwback in use, the errors will be listed with a higher priority alongside warnings in the throwback window.

With throwback not in use the dialogue box which appears when an error occurs is similar to that for a warning except that the title is "Error from ABC" rather than "Warning from ABC". You are still given the two options **OK** and **Edit**.

Because the cycle of:

- Find a bug
- Correct it
- Recompile

can be time-consuming, the compiler will do its best to avoid giving errors. The majority of syntax problems, for example, produce just a warning with a sensible guess being made as to what you intended.

A common cause of errors is trying to use an *unassigned variable*. This is a variable which is never assigned a value within the program. This situation frequently occurs due to the mis-spelling of variable names in expressions. However, particularly with a large program, you may inadvertently have several such variables. It would be very frustrating to have them reported one at a time, with you fixing a single occurrence and recompiling from the beginning inbetween.

So, what the compiler does, is to report them as warnings as it finds them. Then, at the end of the first pass when it is sure that no further ones exist, you will be given the error:

```
Unknown object(s) found
```

For example, try compiling **Error1**. This file contains the following program:

```
10 REM >Examples.Error1
20 A% = a
30 B% = b
40 C% = 3
50 D% = A% * B%
```

The compilation will produce two warnings:

```
a is not defined
```

and:

```
b is not defined
```

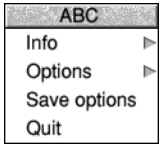
Then, when the pass is complete, the following error will be given:

```
Unknown object(s) found
```

The compilation will then end. This allows you to find out about and fix all unassigned variables with just one pass of the compiler.

Icon bar menu

If you menu click on the ABC icon on the icon bar, the following menu will appear:



The first and last options are the standard ones provided by the majority of applications.

Info

This gives you information about the application.

Options

This leads to the options dialogue, described below.

Save options

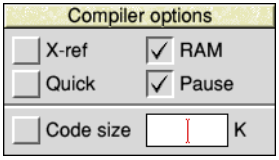
Choosing this will save the current settings of the options so that they are recalled automatically the next time you use ABC.

Quit

Quit ends the ABC application, removing the ABC icon from the icon bar.

Compiler options

The **Options** sub menu reveals the following dialogue:



Each of the option boxes represents a setting which affects how the compiler behaves.

X-ref

When selected, this option instructs the compiler to build up details of all the variables, procedures and functions used in the program; the calling sequence of routines etc.

Then, when the compilation is complete, you can interrogate this store of information. Full details of this option are given in the chapter C*ross referencing* on page 69.

The default is for the option to be **off**.

Quick

When selected, this option applies a number of compiler directives to produce code which is smaller and faster. The list is as follows:

```
REM {NOTRAPS}
REM {NOSTACKCHECK}
REM {NOESCAPECHECK}
REM {NOARRAYCHECK}
REM {NOZEROLOCALS}
```

Details of these directives are given later in the chapter C*ompiler directives* on page 35.

The directives make a number of assumptions about your code. Therefore, this option should be used with care.

While most of these are unlikely to cause problems, NOZEROLOCALS is the exception. This asserts that your code does not rely on local variables in procedures or functions having a value of zero until they are explicitly assigned a value. With the assertion in place an uninitialised local variable will almost always have a non-zero value until an assignment is made. This can cause some very mysterious bugs, so if you find that your program becomes unreliable with the quick option turned on try adding

```
REM {ZEROLOCALS}
```

at the start of the program to override the directive. If your program then works properly you should then either leave this in place or track down the uninitialised variables.

The default is for the option to be **off**.

RAM

When selected, this option causes ABC to perform RAM-*to*-RAM *compilation*. The alternative is for the compiler to read the source from disc, a few lines at a time, and write out the object produced in a similar manner.

RAM-to-RAM compilation is faster but does require a larger amount of memory and so may not be practical for compiling large programs on smaller machines.

The default is for the option to be **on**.

Pause

When selected, causes the compiler to display a warning dialogue box every time it finds a problem, and wait until you click on **OK** or **Edit** before continuing with the compilation. This gives you chance to act on the mistakes and stop the compilation if you wish.

If you de-select the option, messages will not be displayed. The only information you will be given will be a count of the number of warnings when the compilation is complete. Deselecting this option is particularly useful for long compilations which you wish to leave unattended.

The default is for the option to be **on**.

Code size

This option and adjacent writeable allows you to preset a memory buffer size for the output code. To direct the compiler to manage this automatically, leave the text field empty and ensure that the option is not ticked.

The default is for the option to be **off**.

4 ABC versus BASIC interpreter

In an ideal world, it would be possible to have complete source and object compatibility between the BBC BASIC interpreter and the ABC compiler. Unfortunately this is not the case, these differences are highlighted below.

Nature of differences

There are certain small differences between the syntax for some instructions, as well as in the output generated. An existing program may, therefore, need some minor modification before compilation.

Some of these differences are inherent in the execution of compiled or interpreted code. Others are deliberately implemented to improve the performance of the compiled code - since the compiler is primarily intended as a development tool, not just a new way to run existing programs.

Structures

The compiler supports all the flow control structures available in BASIC V with just a few minor differences.

Closed structures

The interpreter insists that there is a one-to-one correspondence between opening and closing structure markers in the case of CASE...ENDCASE and IF...ENDIF structures. The compiler takes this rule one step further and insists that the one-to-one correspondence is maintained between REPEAT...UNTIL, WHILE...ENDWHILE, and FOR...NEXT.

Multiple exits from procedures & functions

For procedures and functions, the compiler's rules are more relaxed. It does not insist that there is only one ENDPROC for every DEFPROC and only one function return for every DEFFN.

The rules governing the use of ENDPROC in ABC programs are as follows:

Procedures

The body of a procedure starts at the DEFPROC and lasts until either:

- an ENDPROC is found which is not nested within a structure
- the end of the program is reached
- another DEF is encountered.

An ENDPROC which is nested within some other structure will be compiled as a branch to the 'real' end of the procedure.

Thus examples of legal structures are:

```
1000 DEF PROCex1
1010 REPEAT
1020 IF X = 0 THEN ENDPROC
1030 .....
1040 UNTIL FALSE
1050 .....
1060 ENDPROC
```

```
2000 DEF PROCex2
2010 .....
2020 CASE X OF
2030 WHEN 1 : ENDPROC
2040 OTHERWISE
2045 .....
2050 ENDCASE
2060 ENDPROC
```

```
3000 DEF PROCex3
3010 .....
3020 IF X = 0 THEN
3030 .....
3040 ELSE
3050 .....
3060 ENDPROC
3070 ENDIF
3080 .....
3090 ENDPROC
```

Whereas the following is not allowed:

```
1000 DEF PROCex4
1020 IF X = 0 THEN 1050
1030 .....
1040 ENDPROC
1050 .....
1060 ENDPROC
```

This example is not allowed because neither of the ENDPROCs on lines 1040 or 1060 are nested within some other structure. This program should be converted to use the multi-line IF...THEN...ELSE...ENDIF to make it acceptable.

Note: PROCex1 may cause some problems for earlier versions of the interpreter. Under BASIC IV and previous versions, the REPEAT...UNTIL loop would have been left 'pending' if the ENDPROC on line 1020 had been executed. This was corrected under BASIC V - this version flushes the stack and discards the loop. ABC has no problems with this type of structure, whilst executing it does not have the concept of 'current' structures.

Functions

The rules for the use of function returns are the same with one addition:

- No function may return values of incompatible data types.

For example, the following is not allowed:

```
100 DEF FNx(A)

200 IF A = 0 THEN
210 = "HELLO"
220 ELSE
230 = PI / 4
240 ENDIF
```

This example isn't much use anyway because just about all you can do is print it!

Multiple entry points

A listing such as:

```
1000 DEF PROCa
1010 DEF PROCb
1020 PRINT "Hello"
1100 ENDPROC
```

is treated differently under the compiler and the interpreter.

Under the interpreter, it leads to two identical procedures being available; PROCa and PROCb. Calling either of these will result in the string “Hello” being printed.

Under the compiler, two procedures will exist but will be different.

Under the rules given in the section above, PROCa ends when the DEF on line 1010 is encountered. Hence PROCa is a null procedure which does nothing when it is called. Note that a warning will be given to let you know that the procedure body is unclosed.

See also the compiler directive NOTRAPS which is documented in the section *Falling into* PROCs/FNs on page 49.

Keyword position

Under the interpreter there are strict rules about where certain keywords must occur in the text. These rules are more relaxed under the compiler.

The compiler still insists on the THEN of a block structure IF...THEN...ELSE...ENDIF being at the end of a line. Similarly the OF in a CASE...OF...WHEN...ENDCASE.

However, the keywords WHEN, OTHERWISE, ENDCASE, ENDIF etc no longer have to be the first non-space objects on a line. For example, the following is quite acceptable:

```
CASE i% OF
WHEN 1:PRINT"1":WHEN 2:PRINT"2"
OTHERWISE PRINT "ERROR":ENDCASE
```

The 'Dangling ELSE'

The compiler corrects one of the 'bugs' in the BASIC interpreter, by correctly handling nested single-line IF...THEN...ELSE statements.

```
IF A% > B% THEN IF A% > C% THEN PRINT "A"
ELSE PRINT "C" ELSE PRINT "B or C"
```

Under the interpreter, the first ELSE applies when either of the preceding IF statements returns FALSE. This means the only possible output is A or C under all circumstances. This is clearly incorrect as demonstrated in the following table:

A	B	C	Output
1	2	3	C
1	3	2	C
2	1	3	C
2	3	1	C
3	1	2	A
3	2	1	A

Under the compiler, the innermost ELSE applies to the innermost IF, and the outermost ELSE applies to the outermost IF. The correct results are returned as follows:

A	B	C	Output
1	2	3	B or C
1	3	2	B or C
2	1	3	C
2	3	1	B or C

A	B	C	Output
3	1	2	A
3	2	1	A

Short cuts

The compiler does not allow the use of:

```
NEXT,
```

to terminate two (or more) FOR loops. Similarly, it does not allow the use of only one NEXT statement to terminate more than one FOR loop, through the use of the outer loop's control variable only.

For example, the following program will be rejected:

```
10 FOR I% = 0 TO 10
20   FOR J% = 0 TO 10
30   A%(I%,J%) = 0
40 NEXT I% : REM terminate both loops
```

Scope rules

It is not uncommon for a program to contain several variables with the same name. For example, there may be a global name created at the 'top' level and several independent local versions, created within procedures and functions.

Clearly, when a variable is referenced the program must know which one to use. This is determined by the scope rules. A variable can be referenced only when it is in scope. So, for example, a LOCAL variable can be referenced only within the procedure or function to which it belongs.

The scope rules for the compiler are *static*. This means that a specific occurrence of a non-unique variable name in the text of the program always references the same variable, whether local or global. Furthermore its meaning can be determined just by looking at the program text.

This view of the scope rules is not shared by the interpreter, its scope rules are said to be dynamic. That is, determination of which variable is referenced happens while the program is running and may depend on code elsewhere in the program.

For example:

```
10 A = 1
20 PROCp(A)
30 END
40 DEFPROCp(X) : LOCAL A
50 PRINT X,A : PROCq(X)
60 ENDPROC
70 DEFPROCq(Y)
80 PRINT Y,A
90 ENDPROC
```

Under the interpreter, the value of A in PROCq depends on where PROCq has been called from. If it is from the main program A takes the global value (1). If it is from PROCp the variable A takes the value local to PROCp (0).

Under the interpreter this program will produce:

Output	Implication
1 0	A passed to PROCp in X but A made local to PROCp
1 0	X passed to PROCq in Y but A still local to PROCp

The compiler classes all variables as global, unless they are declared explicitly to be local to the procedure or function in which they are used. In the example above, therefore, the local value of A applies only to PROCp, not to any other procedure. In the compiled version PROCq will always use the global value of A, regardless of where PROCq was called from.

Under the compiler this program will produce:

Output	Implication
1 0	A passed to PROCp in X but A made local to PROCp
1 1	X passed to PROCq in Y but A is now the global value

Variables such as A are known as *dynamic free variables*.

An additional point to note about scope is that the use of GOTO or GOSUB to jump from one procedure or function into a different one will produce unpredictable results, since the variables which will be in scope after the jump are unknown.

Local error handling

ABC provides a full package of local error handling. This differs from the way in which the interpreter handles it - the specification is as follows:

Setting local error handlers

```
LOCAL ERROR <stmt>
ON ERROR LOCAL <stmt>
```

Both mean the same thing, namely 'set an error handler for this procedure'. The error handler will be inherited at run-time by any PROC or FN which is called from the scope of the local error handler.

```
10 ON ERROR PRINT "Error detected"
.....
99 END

100 DEF PROCone
110 LOCAL ERROR PRINT "Local error detected"
.....
150 ENDPROC

200 DEFPROCtwo
210 ERROR 1,"Crash!"
220 ENDPROC
```

If PROCtwo is called directly from the main program, then the error which it generates will be caught by the global error handler which is set up on line 10.

Alternatively, if PROCtwo is called from PROCone, the local error handler set up by PROCone on line 110 will be activated.

In a more complicated case there might be a third procedure, PROCthree, which made calls of PROCtwo and was itself called from either the main program or from PROCone. The error will be trapped by the global error handler if the sequence of calls has not been through PROCone. If PROCone has been called its local handler will trap the error.

Once a procedure returns to the place it was called from, any local error handler which it may have set up will be deactivated. Subsequent errors will then be trapped by the previously active error handler.

When an error occurs, the stack is wound back to the level at which the error handler was set up and hence variables have their values restored. The interpreter does **not** do this, which can be very inconvenient.

Cancelling local error handlers

Use the construct:

```
LOCAL ERROR OFF
ON ERROR LOCAL OFF
RESTORE ERROR
```

These delete the record of the error handler at the current stack level. Using them inside a procedure which has no local error handler will have no effect.

Leaving a procedure (by ENDPROC) which set up a local error handler will automatically remove the handler, i.e. ENDPROC automatically does RESTORE ERROR.

Global error handlers

The effect of ON ERROR and ON ERROR OFF will be to set up the global error handler. The global error handler will only be called if either:

- the program is at the global level
- no local error handler is in force.

It must be noted that when an error occurs, compiled programs still lose track on any stacked GOSUB/RETURN information. This is another good reason why PROCs should be used instead!

ERL

ERL always returns the value 0.

Floating point

The compiler makes use of the F*loating* P*oint* E*mulator* (and hence the floating point coprocessor if fitted). This allows you to choose between using single-, double- or extended-precision formats for storing floating point numbers. This contrasts with the interpreter which always uses a non-standard five-byte format.

The BASIC interpreter uses the presence of $ and % characters to distinguish between floating point numbers, strings and integers. This scheme has been extended to control the different types of floating point through the use of the ` and & characters.

Note that all references to the ` character are describing the use of the character whose ASCII value is &60 (96). This appears as the back-tick character when using the ISO fonts and will be produced by the back-tick key on the keyboard, unless RISC OS has been reconfigured to a different KEYBOARD or COUNTRY setting in which case it may appear as £.

If the name of a variable

- ends in a `, the compiler will treat it as double-precision
- ends in an &, the variable is assumed to be extended-precision.

By default, the precision of a floating point variable is assumed to be single if its name ends in a letter (A...Z or a...z), a digit (0...9) or the underscore character (_).

Thus the following program is assumed to use two single-precision (hot and cold) and one double-precision (double`) variables.

```
10 REM >Examples.FP
20 hot = 27.453
30 cold = -10
40 double` = 4.6692106090
```

Under the BASIC interpreter, this program will run although all three variables will be represented in BASIC's unique five-byte format.

The rules which apply to the names of simple variables also apply to arrays, thus:

```
10 DIM block` (1000)
```

will, by default, set up an array of 1000 double-precision numbers occupying 8000 bytes.

This does pose a compatibility problem with the use of & as a variable name suffix in that the BASIC interpreter will not allow it.

Thus programs which use & to specify the use of extended precision cannot be tested under the interpreter. However, this problem can be worked around through the use of the TYPE compiler directive described in the chapter *Compiler directives*.

Single-precision numbers provide only very limited accuracy, typically about six decimal digits. Use of double-precision increases this to about 15 while extended-precision provides up to 19. This is illustrated in the table below:

Precision	Digits	Exponent range	Storage per variable
Single	6	-44 to +38	4 bytes
Double	15	-324 to +308	8 bytes
Extended	19	-4950 to +4932	12 bytes

@% and print formatting

To allow for the use of higher precision it has been necessary to change the meaning of the print control variable @%. Each of the four bytes now has a special meaning:

Byte four - the flags byte

This controls the format in which PRINT# sends numbers to data files as follows:

Format	Bit 25	Bit 24
Single precision	0	0
Acorn 5 byte	0	1
Double precision	1	0
Extended precision	1	1

Compiled programs will read data files in any of these formats and carry out the necessary conversion automatically. This enables full compatibility with the interpreter. However, if your compiled program produces data to be read by INPUT# under the interpreter, you will have to stick to the Acorn format.

This byte also controls whether STR$ obeys the remaining bytes. If the top bit is set, STR$ obeys the print formatting bytes, otherwise it does not.

The default value of the flags byte is **zero**.

Byte three - significant digits

The lower five bits of this byte specify the number of significant digits which are allowed. A number will be printed in such a way that the total number of significant digits is not more than this limit. Thus, if the number of digits before the decimal point plus the number of decimals specified by byte two of @% is less than or equal to this limit, then the number is printed in full using fixed-point format.

Otherwise the number will be printed in exponential format with this number of significant digits.

Byte two - decimal places

The lower five bits of this byte specify the number of decimal places which will be used for fixed-format numbers.

Byte one - field width

This controls the number of character places in which the number will be output. If the number has fewer digits than this it will appear right-justified with the appropriate number of leading spaces. If the number won't fit in the field width it is printed in as few places as possible.

The top byte of the default value of @% is affected by the TYPE compiler directive in order to control the default format in which floating point numbers are output to data files.

The default value of @% always has its bottom three bytes set to **&0A0A0A**. This means that, by default, floating point numbers will be printed in general format using 10 significant digits in a field of 10 places.

@% and floating point in data files

The BASIC V interpreter use a five-byte representation for floating point numbers in data files - marked by a type byte of &80. Earlier 6502 BASICs used &FF as the type marker. Compiled programs will put one of &F0, &E0, &D0 or &80 in the type byte according to the value of @%:

- &F0 Floating point, i.e. single-precision
- &E0 Floating point, i.e. extended-precision
- &D0 Floating point, i.e. double-precision
- &80 Acorn 5 byte format

The 4, 5, 8 or 12 bytes of the floating point number will be output directly to the file, lowest byte first.

Variables, arrays, parameters etc

Numeric input

Under the interpreter, the rules about what can be supplied when numeric input is called for vary depending on the keyword used. For example, INPUT allows only decimal numbers, whereas READ takes decimal, hex or binary numbers, variables and some simple expressions.

Under the compiler, the rules for numeric input have been standardised. Only numbers are allowed, but these may be given in decimal, hex or binary. So the format required for a number in all cases is as follows:

- An optional + or - sign
- An optional radix indicator (% or &)
- A string of digits:
  - Binary digits (0 or 1) if radix indicator = %
  - Hex digits (0-9 & A-F) if radix indicator = &
  - Decimal digits (0-9) if no radix indicator
- An optional decimal point followed by a string of decimal digits
- An optional E to introduce the exponent followed by:
  - an optional + or - sign
  - a string of up to four decimal digits.

Numeric conversions

Another extension supported by ABC is that VAL is allowed to take a hexadecimal string. The interpreter restricts VAL to decimal strings. For example:

```
VAL ("&" + addr$)
```

RETURN Parameters

ABC supports RETURN parameters, that is parameters to procedures or functions which return their final values back to the variables which were used when the call was made.

The syntax for these is compatible with that of the interpreter but there are a couple of minor restrictions:

- The actual parameter may only be a simple variable or an array element, it may not be an indirect expression such as base%!4.
- Each routine may have a maximum of eight RETURN parameters.

Arrays

The compiler supports all of the array handling features of BASIC IV. However, it provides neither the extended features of BASIC V for handling whole arrays nor the LOCAL array handling. One further restriction is that array elements may not be used as the control variables of FOR loops. Items which are affected by this are:

- The DIM function
- The SUM functions
- Passing arrays as parameters
- Array assignment
- Array arithmetic and vector processing
- LOCAL arrays

Indirection operators

Whilst reading values from indirect expressions is fully supported, assignment to them is restricted to use in straightforward assignment statements. The following are allowed:

```
PRINT $(buffer%+I%)
PROCfred(A%?B%)
$(buffer%+I%)="Hello"
```

But the following types of operation are not allowed:

```
INPUT $buffer%
INPUT #file%,buf%?I%
SWAP I%!(J%+0),I%!(J%+4)
[:.labels%!I% :]
READ $buffer%
DEF PROCaction(block%?1)
LOCAL !FNfred
SYS A,B,C TO $buffer%
LEFT$(buf%)="XXX"
```

Pseudo variables

Under the interpreter, the *pseudo variables* such as PAGE and TOP hold values defining the location of the BASIC program and the workspace it uses when running. The interpreter also allows certain of these pseudo-variables, such as HIMEM, to have values assigned to them by the user to limit the memory used.

Under the compiler, the values returned by these pseudo-variables are obviously different, since the object code is in memory during execution, not the source. Also, the compiler uses workspace differently to the interpreter.

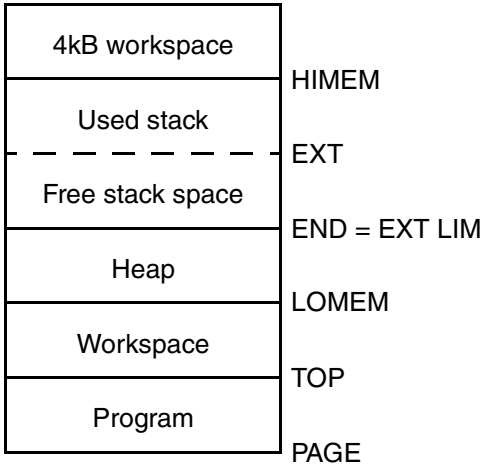
Nevertheless, the pseudo-variables do return sensible values as follows:

- HIMEM Top of memory
- END Top of heap
- LOMEM Bottom of heap
- TOP Top of the program
- PAGE Start of the program

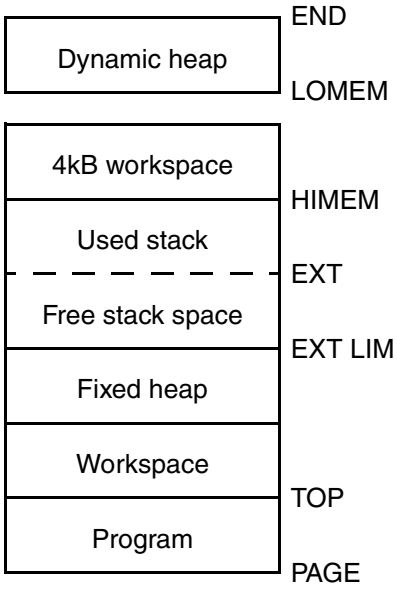
In addition, the compiler provides extra pseudo-variables:

- EXT Current stack pointer
- EXT LIM Bottom of stack
- QUIT Address of exit handler

There are two different ways in which memory can be arranged whilst using the compiler. The default method is shown below:



The alternative method applies when the NEWHEAP directive is being used. For more details see the chapter *Compiler directives*.



The restriction in the interpreter which prevents pseudo-variables being used as indirection operators does not apply to the compiler.

For example:

```
PRINT END?1
```

is allowed and will work correctly. However, assignments to pseudovariables are not permitted by the compiler.

Calling machine code

The two keywords for calling external machine code routines are CALL and USR. There are two different ways in which these can be used under the compiler.

Standard form

```
CALL/USR <expression>
```

The expression gives the address of the routine which is to be called. The values of A%, B%, ..., H% are exported to the routine in the registers R0, R1, ..., R7.

When used inside a procedure or function, the compiler will first try to find LOCAL variables A%...H% belonging to that particular routine. If the procedure or function does not have its own LOCAL A% etc, then the compiler will use the global assignments and issue a warning that it is doing so.

Note that the global integer variables A%...Z% are always pre-declared.

Extended form

```
USR <expression> (param1,...,param8)
CALL <expression> (param1,...,param8) TO var1,...,var8;var9
```

Both USR and CALL can be followed by a list in brackets of up to eight parameters which will be assigned to the registers R0...R7. It is not necessary to supply all eight values, but those supplied will be assigned in sequence.

The single value returned by the USR function is the value held in R0 when the routine ends.

CALL can be followed by the keyword TO and a list of up to eight variable names, into which the values R0...R7 are placed when the routine ends. This provides a method of getting several values back from a routine at once. In addition, the state of the flags can be returned by following the variables by a semicolon and a further variable name. For example:

```
CALL code%,2 TO x%,y%;flags%
```

In the above example, x% and y% will hold the values of R0 and R1 respectively and flags% will hold the state of the system flags, i.e. Negative, Zero, Carry, Overflow.

Operating system calls

Handling of CALL

On a 6502 based BBC Microcomputer, operating system routines can be accessed by calling particular addresses at the top of memory. For example:

```
10 A% = 138
20 X% = 0
30 Y% = 65
40 CALL &FFF4
```

This program calls the OSBYTE routine in the operating system to insert an A into the keyboard buffer.

The BASIC interpreter in RISC OS has a table of 'legal' 6502 entry points and checks every CALL or USR to see if the address called is one of these. If it is, the interpreter translates the call into the equivalent SWI.

This attempt at compatibility with earlier Acorn micros has a problem. These addresses now lie in user memory because the very large address space and increased amount of RAM up to and above 64kB.

Hence, if you generate object code and try to CALL it, you may be unlucky and hit one of these special addresses. If this occurs, the 6502 routine will be called, not your own code.

For this reason, and the fact that the compiler is aimed at helping the development of new code rather than running existing code, this feature is not supported by the compiler. If the compiler observes a call of this nature it will give a warning.

However, for the compiler to be able to recognise the calls, the address has to be given as a constant. If the call is of the form:

```
CALL os_byte%
```

where os_byte% has been assigned the value &FFF4, then the compiler cannot detect it.

SYS keyword

Use the SYS keyword in preference to CALL to call RISC OS software interrupts. These accept up to 8 input parameters and 8 output parameters which correspond to the ARM's registers R0-R7.

Nine registers (R0-R8) can be returned in ABC version 4.11 and later.

Uninitialised registers can be skipped and will be set to zero as the interpreter does, unless the directive controlling I*nitialising* SYS *registers* on page 48 is in use.

Assembly language

The in-line assembler is fully supported. The facility is included for compatibility, where this is vital. Note, however, that it is far more efficient and logical to assemble machine code as separate external routines and then call these as general subroutines, than to include them in a compiled program.

Remember that the assembler does not take over from the compiler, you will actually be compiling a program which assembles a routine at run-time.

If this facility is used, certain differences should be noted.

Register names

Under the interpreter, the variables R0...R15 are assumed to automatically contain the values 0...15, and hence refer to the registers.

Under the compiler these variables must be defined.

The compiler will perform the definition for the following variables for you if necessary, i.e. if you have used one without defining it:

```
r0...r15, R0...R15, pc, Pc, pC, PC
```

but will issue a warning each time it does so. A compiler directive has been provided to allow the names to be declared in one go:

```
REM {REGISTERS}
```

This is covered in more detail in the chapter C*ompiler directives*.

OPT

Bit zero of the OPT setting (enable/disable assembly listing) is available. However, the compiled code cannot produce a full listing because to do so would require access to the source text.

For instance, branches are shown as the mnemonic followed by the address of the destination. All information about the name of the label has been lost at that stage.

Any undefined label references will be found by the compiler rather than by the assembler. The meaning of bit one in the OPT command (enable/disable assembler errors) is therefore altered.

An assembler directive:

```
REM {NOOPT}
```

can be used to make the compiler take no notice of OPT statements.

This is covered in more detail in the chapter C*ompiler directives*.

EQUate directives

In addition to the standard EQUate directives for bytes, words and strings, the following are provided:

- EQUFS **EQ**Uate **F**loating point **S**ingle precision
- EQUFD **EQ**Uate **F**loating point **D**ouble precision
- EQUFE **EQ**Uate **F**loating point **E**xtended precision
- EQUF **EQ**Uate **F**loating point

These place the value given to them in-line in the assembly code file to their respective precisions. In the case of EQUF, the precision is governed by the setting of the TYPE compiler directive described in the chapter C*ompiler directives*. EQUF will use the same precision as will be used by a simple variable.

Banned keywords

The following BASIC keywords are commands that cannot be used in programs either under the interpreter or the compiler:

AUTO	LVAR
DELETE	NEW
EDIT	OLD
HELP	RENUMBER
INSTALL	TWIN/TWINO
LIST/LISTO	TRACE

If the compiler discovers one of these in a source program it will give a warning and ignore it.

In addition, there are certain keywords which are allowed under the interpreter but not under the compiler:

APPEND	COUNT
CHAIN	EVAL
LIBRARY	SUM
LOAD	WIDTH
OVERLAY	
SAVE	

The commands in the left-hand column are forbidden because they all load or store BASIC programs and at run-time the source program is not available.

EVAL would require the compiled program to have access to the complete source text of the program during execution and the means to decode it. This would be analogous to including a copy of the interpreter with every program and is therefore not a viable option.

COUNT and WIDTH are not implemented since they are so rarely used that the reduction in the speed of output routines which they cause is not justified.

SUM is not supported since it forms part of the whole array manipulation package which is not available.

Effect on TAB & SPC

Note that because COUNT is not implemented, TAB and SPC work differently. TAB(n) outputs spaces until POS = n. SPC(n) outputs n spaces.

Effect on DATA & READ

Note also that the omission of EVAL affects DATA/READ statements. Under the interpreter it is possible to place expressions in DATA statements and have them evaluated when READ. This also applies to variables names: For example:

```
READ A%,B%
DATA X*StepX, 10*20
```

The above would result in A% being set to the value of X*StepX and B% to 200 (assuming both X and XStep exist).

The compiler cannot support this because it requires the expression evaluator EVAL. Unfortunately, it is not possible for the compiler to detect this situation since the data could be used quite legally as a string. Therefore, a program such as this will compile but will produce different results (A% = 0 and B% = 10).

CLEAR

The keyword CLEAR is not banned under the compiler, but it does perform a different task.

Under the interpreter it clears any existing variables and forgets about existing procedures, subroutines etc.

Under the compiler it forms part of the memory management system and can be used with the:

```
REM {NEWHEAP}
```

directive to release a block of memory which has previously been claimed. Its syntax is:

```
CLEAR <expr>
```

This is covered in greater detail in the chapter C*ompiler directives*.

5 Compiler directives

ABC provides some special directives which are used to control the way in which the compiler treats the source program.

General format

These directives mostly take the form of specially constructed REM statements and as such are ignored if the program is used under the interpreter. A typical directive is:

```
100 REM {NOESCAPECHECK}
```

Most ABC directives take this form, with a REM followed by some text enclosed in curly brackets. The text must always be in upper case if ABC is to recognise it. For directives that control and on/off state, where there is a directive to enable

```
REM {feature}
```

there is also its logical opposite

```
REM {NOfeature}
```

to disable that feature.

Some of the directives are associated with the compilation of relocatable modules and are listed separately in the chapter *Relocatable modules* on page 55. Others allow conditional compilation to be used, these can be found in the chapter *Conditional compilation* on page 53. The others follow below.

Program directives

Ignoring sections

If there is a section of the program which you do not wish to be compiled, for instance debugging procedures, you can instruct the compiler to omit it. This is accomplished by placing the directive:

```
REM {NOCOMPILE}
```

before the section to be ignored and by placing:

```
REM {COMPILE}
```

following it to resume compilation.

This provides a much better solution than 'commenting-out' a whole section. Note that you cannot nest these directives. A warning will be given if you attempt to do so.

For existing code, most of the differences listed in the chapter ABC *versus* BASIC *interpreter* can be dealt with by making minor modifications. However, there may be some circumstances in which the code required for the compiler is different to that for the interpreter.

To avoid the need for two versions of a program, one for the interpreter and one for the compiler, the following 'trick' can be used.

Write two procedures, with the same name, one containing the code needed for the interpreter and one for the compiler. Place these at the end of the program, the interpreter one first and surround the interpreter procedure by the NOCOMPILE and COMPILE directives.

For example, when using the NEWHEAP directive, the following is useful:

```
REM {NEWHEAP}
.....
PROCfree(block%)
.....
END
.....
REM {NOCOMPILE}
DEFPROCfree(addr%)
IF addr% < LOMEM OR addr% > END THEN
ERROR 1,"Not a heap block"
ENDIF
ENDPROC
REM {COMPILE}
.....
DEFPROCfree(addr%)
CLEAR addr%
ENDPROC
```

When the program is run under the interpreter, the first version of the procedure will be found and used. The compiler directives are simply treated as remarks and so are ignored. The fact that the procedure name exists twice does not cause a problem. This is because the interpreter always searches from the beginning of the program for procedures. When the name is found the search terminates.

However, the REM statements instruct the compiler to ignore the interpreter version of this procedure, so only the second version will be included in the compiled program.

ARM image format

Some versions of RISC OS require that all executables are prefixed with an AIF header in order to accept that they contain runnable code, despite their filetype.

Use the directive:

```
REM {AIFHEADER}
```

to request that ABC constructs a valid AIF on the compiler's output. The addressing mode and three flags bytes at offset +&30 will have bit 5 set by default to denote the addressing mode is 32 bit compatible. Alternatively, the flags word can be overridden with the directive:

```
REM {AIFFLAGS = &1020}
```

if this is desired.

For a more detailed description of the AIF see *Code file formats* in the *Desktop Tools* manual.

Handling large programs

By default, the compiler generates offsets within the object code using a short addressing mode. This helps to keep the size of the object code low, however it does place a limit of 256 kilobytes on the size of the object code. For most people this will not be a problem.

However, if you do want to write a mammoth application, you need to include the directive:

```
REM {LONGADRS}
```

at the start of your program. Its inverse, which gives the default situation, is:

```
REM {SHORTADRS}
```

CASE statements

When ABC comes across a CASE statement it can generate the code in two different ways. One is to generate a series of comparisons and branches:

Compare the CASE expression with the WHEN expression
Branch if not equal to the next WHEN expression

The other method is to build a jump table which contains the address of the code to be used for each possible value of the WHEN expression. The possible values are determined by the maximum and minimum of the range given.

For example, a CASE statement such as:

```
CASE var% OF
WHEN -30 : ...
WHEN 0 : ...
WHEN 30 : ...
ENDCASE
```

The jump table would require 61 entries, one each for the numbers -30, -29, -28...28, 29, 30.

This method produces code which executes more quickly in general but may occupy more memory, particularly if the table is sparsely populated, as in the above example.

You can determine the cut off point at which the compiler chooses to use a series of comparisons and branches rather than a jump table by using the following directive:

```
REM {MAXCASES = n}
```

If the number of entries which would be required in the table is less than or equal to MAXCASES, a jump table is used, otherwise the comparisons & branches are created.

The default value is **256**.

Names of operating system routines

Under the interpreter, providing the identifier of an operating system routine as a number is more efficient than providing it as a name.

Names have to be converted into the appropriate numbers before the call can be made.

The compiler is capable of converting names into numbers at compile time and so producing more efficient code. However, to do so, it needs to 'know' all the routines being called. For example, if your code is to access a routine from a module, then the compiler can only convert the name you supply into its number if the module is loaded at compile time.

The compiler will always perform this optimisation where it can. In addition, if you use the compiler directive:

```
REM {SYSKNOWNONLY}
```

ABC will produce a warning message every time it comes across the name of an operating system routine which it doesn't know. Without this directive, no such warning will be given so the inefficient version of the call will be generated without you knowing about it.

The alternative is:

```
REM {NOSYSKNOWNONLY}
```

Compressed compiler output

If the directive:

```
REM {SQUEEZE}
```

is used the compiler will attempt to run the compiled code through the squeeze program to reduce its size.

The utility `squeeze` is provided with the Desktop Development Environment and must be located on the *Run$Path* for it to be found.

For code which will run on 26 and 32 bit environments you must use squeeze 5.08 or later.

The opposite directive:

```
REM {NOSQUEEZE}
```

turns off the compression step, and is the default setting.

Memory directives

There are a set of directives which control the way in which your program will allocate memory to its various tasks. If neither of these is specified then a sensible default setting will be used.

Stack and heap allocation

When a compiled program is running it requires three separate areas of workspace, the ABC heap, the BASIC stack and the system area. The functions of each of these are outlined below.

1 **The BASIC heap**

This is where all global variables, strings and arrays are held. It is also where any DIM statement will claim memory from. It is located immediately above the top of the program in memory.

2 **The BASIC stack**

This is where all parameters, local variables and procedure call information is held. It is located high up in memory and grows downwards towards the top of the heap.

3 The system area

This is a 4kB area for a small stack and some private information. This will be at the highest address in memory which is used.

The default action is to reserve 4kB at the top of memory for the system area and then divide the remainder equally between the heap and the stack. In most cases this will be suitable, but some programs will need more of one type of memory than of the other.

For example, a program which operates on very large arrays may well need most of the memory to hold them whilst at the same time having only very modest stack space requirements.

To give you control over this, the amount of heap and stack space required by a program can be set in one of two ways.

Absolute values

The number of bytes to be reserved for either the heap or stack can be specified. For example, the heap space can be specified using a directive of the following format:

```
REM {HEAP = 10000}
```

This will reserve 10000 bytes for the heap. Alternatively, to specify the stack space:

```
REM {STACK = 6000}
```

This will reserve 6000 bytes of memory for the stack. When your program is running, HIMEM will return the address of the top of this stack and the EXT function will return the current value of the stack pointer. Thus, the amount of stack space in use at any given time is:

```
HIMEM - EXT
```

If you specify either HEAP or STACK, but not both, then the other will use up the rest of the available memory. Thus, a good way to maximise the amount of heap space is to specify the minimum stack size and leave the heap to grab the rest of memory.

If both options are specified then the two areas will be adjacent in memory. There will then be a free area of memory above the top of the system area. See the diagrams in the description of P*seudo variables* on page 27 for more details.

Percentages

Often a more useful way of specifying the space is to specify the percentage of free memory to be used by either the heap or the stack. This can be achieved as follows:

```
REM {HEAP% 90}
```

where the number specifies the percentage of the free memory which will be allocated to the heap. Similarly:

```
REM {STACK% 10}
```

would try to allocate 10% of the available memory to the stack.

Obviously, the total cannot exceed 100%.

These directives can be used in conjunction with the absolute versions of the directives where appropriate. This is a useful method if you are attempting to compile under different memory configurations.

Dynamic

There is a further directive:

```
REM {NEWHEAP}
```

which can be used in addition to a fixed heap and stack to set up an *expanding/contracting heap*. Note that this can only be used by RISC OS applications.

For example:

```
REM {STACK = 8192}
REM {HEAP = 8192}
REM {NEWHEAP}
```

This sets up 8 kilobytes for the stack, 8 kilobytes for the fixed heap and a dynamic heap as well. When both types of heap exist, the fixed heap will be used for holding global variables and for system use.

The dynamic heap will hold strings, arrays and DIMmed memory. The keyword CLEAR can be used in conjunction with the NEWHEAP directive, This can be used to release memory.

For example, statements such as:

```
DIM fred%
```

will reserve blocks of memory from the expanding/contracting wimpslot heap. If you have finished with one of these blocks, you can use:

```
CLEAR fred%
```

to free it again and make it available for other things.

The wimpslot for the application will automatically increase when necessary and will decrease again once a sufficient block of memory at the top of the wimpslot has been freed.

Variable directives

Variable types

ABC provides a directive to allow full control over the relationship between the type of a variable and the last character of its name.

This is the TYPE directive. In its simplest form it can be used to force the compiler to change the precision used for simple floating point variables. The following program shows this in use:

```
10 REM {TYPE = DOUBLE}
20 INPUT x
30 INPUT power
40 .....
50 variable = x^power
60 .....
70 PRINT variable
```

All three variables (x, power and variable) will be created as eight byte double-precision variables.

Note: The TYPE directive must be placed at the top of the program before any executable code.

A more complex version of the TYPE directive is available which allows you to specify the meaning of one of the special characters, thus overwriting the default meanings. For example:

```
10 REM {TYPE ` = EXTENDED}
```

This tells the compiler to use extended-precision for all variables which end in the back-tick character. It can be used to get around the limitation imposed by the BASIC interpreter on the use of & to imply extended-precision.

In most cases a program will probably only want to use one sort of floating point for all its variables. In this case, the simplest solution is to avoid using the & suffix and to specify the same precision for variables with no suffix and for those with a `.

For example:

```
10 REM {TYPE = EXTENDED}
20 REM {TYPE ` = EXTENDED}
```

The program can then be tested under the interpreter although the results will be rather less accurate than those of the compiled program.

If, for some reason, you need to use two forms of floating point within a program you can still maintain testability. Again, you should avoid use of & and specify the precision for the other two possibilities. For example:

```
10 REM {TYPE = SINGLE}
20 REM {TYPE ` = EXTENDED}
30 DIM array(100000)
.....
100 num` = array(I%) * array(J%)
```

This program uses single-precision for the array since 100000*4 bytes is a very large amount of memory (400kB). Using extended-precision would require three times as much memory - 1.2Mbytes!

However, the product of two array elements is put into the extended-precision number num`. This is useful since it is certain that the calculation will not overflow. An extended-precision variable is easily capable of holding the square of the largest possible single-precision number.

Floating point indirection

By default, floating point indirection only transfers four bytes.

However, if the TYPE compiler directive has been used to change the size of the simple variables to eight or twelve bytes, the floating point indirection operator will transfer this number instead.

It is possible to force the compiler to use one particular size and thus override the TYPE directive in the following way:

```
10 REM {TYPE = DOUBLE}
20 DIM block% 1000
.....
100 |{S}block% = 37.5
```

The {S} after the | character forces the compiler to perform a single-precision (four-byte) transfer. {D} for double-precision and {E} for extended-precision operations may be used in a similar fashion.

Forcing integers

It is also possible to force the compiler to treat variables which do not end in the % character as integers.

For example:

```
10 REM {TYPE = INTEGER}
.....
100 fred = 10.3
110 PRINT fred
```

This will print 10 because variable fred is an integer. Of course, the interpreter will still treat fred as being floating point.

Note: It is not possible to change the meaning of the % and $ suffix characters.

Checking for floating point use

In some applications it is useful to know that your program is free of all use of floating point so that the Floating Point Emulator need not be loaded.

The directive:

```
REM {NOFLOAT}
```

will cause the compiler to fault any attempt to use floating point arithmetic (or an attempt to use library routines which do so). Thus, if the program compiles successfully with this directive set, then it can be assumed that it does not use the Floating Point Emulator.

When the NOFLOAT directive is active, the following keywords may not be used:

ACS	ASN	ATN	COS	DEG
EXP	INT	LN	LOG	PI
RAD	RND	SIN	SQR	TAN

Note also that the ELLIPSE keyword may only be used in its four parameter format. The fifth parameter is the angle of rotation and involves trigonometric calculations which uses floating point numbers. Finally, the use of READ and INPUT for integers may call up floating point operations if the data which is to be read is in floating-point form.

Note: The use of these keywords is not forbidden but this limitation must be observed.

Assembly language directives

Register initialisation

In assembly language routines, the registers 0 to 15 are normally referred using the names R0...R15. An alternative name for R15 is PC since this register holds the program counter.

If your program includes any assembly language, it is a good idea to include the directive:

```
REM {REGISTERS}
```

at the top of your program. This creates manifest constants with the names R0...R15, r0...r15, PC, pc, Pc and pC which have the appropriate values.

If you don't include this directive, any of the names which you use which you haven't created yourself as either a manifest constant or a variable, will automatically assigned the appropriate value by ABC.

However, for every register name which has to be assigned to by ABC, a warning will be issued. So use of the directive suppresses these warnings which can become tedious to respond to.

In addition, it may be that you have used a variable with the name R0, say, elsewhere in your program for some other purpose. Hence the compiler will use the value currently assigned. For example, if R0 is holding the value 10 then an instruction such as:

```
ADD R0,R0,#4
```

will add 4 to register 10 rather than to register R0 as intended. The REGISTERS compiler directive is a safe-guard against this happening.

Use of OPT

Whenever the interpreter comes across an opening square bracket, [, it performs an automatic OPT 3. Hence you need to include a statement such as:

OPT opt%

as the first statement after the bracket to ensure that the correct OPT setting is used.

Under the compiler, you can use the directive:

```
REM {NOOPT}
```

to cause all OPT statements, whether explicit or automatic as described above, to do nothing.

One method of using this compiler directive is to have:

```
REM {OPT}
[OPT opt%]
REM {NOOPT}
```

at the start of your program. The OPT directive is not strictly necessary for this state as this is the default. It ensures that the following OPT is taken note of. Hence the OPT setting is set to the value of opt%. The NOOPT directive then tells the compiler to ignore all further OPT statements. Hence future:

```
[OPT opt%
```

statements generate no code.

This can have a dramatic effect of the size of the object code - for example it saved 18kB in the case of ABC itself.

Code cache coherency

Processors from ARM v4 (StrongARM) and later employ a technique to speed up execution by splitting the caches, one for data and one for machine code. Earlier processors used unified caches.

With a unified cache, generating machine code at run time was not a problem as any instructions emitted by the assembler were visible to the processor core in the cache.

With a split cache, the two halves are in parallel and it is possible that something the assembler output is still in the data cache only, and not yet visible in the code cache for the processor to execute. It is necessary to *flush* the caches to synchronise them, a step known as making them *coherent*.

To preserve backwards compatibility with older programs the ABCLibrary version 4.05 and later takes the draconian approach of performing a full code cache flush whenever CALL/USR is called. This approach is necessary because ABCLibrary cannot deduce whether any new code has been freshly assembled or loaded from disc in the meantime. This is a performance hit, but serves to maintain compatibility for pre-StrongARM compiled programs.

For more efficient operation the program can declare the directive:

```
REM {MANUALCODESYNC}
```

which indicates to ABCLibrary that the programmer is managing the code cache synchronisation, and can skip the costly cache flush on every CALL/USR.

It will then be necessary to add the appropriate cache flushes only when they are needed, i.e. after building inline code, or loading a code fragment from disc into the program space. There is an Application Note called AN295 - I*ntroduction to* S*trong*ARM *and* P*rogramming* G*uidelines* which contains further guidance.

The default is NOMANUALCODESYNC, and this can also be set explicitly with:

```
REM {NOMANUALCODESYNC}
```

but programmers are strongly advised to use MANUALCODESYNC.

Warning directives

Global suppression

A pair of compiler directives are provided which enable you to tell the compiler whether or not to issue warning messages. Note that these affect warning messages only - the production of error messages cannot be turned off.

```
REM {NOWARNINGS}
```

will cause the compiler not to issue warning messages. However, the total number of warnings will still be counted and reported when the compilation has ended. This directive may be useful if you need to leave your compilation and so will not be around to tell it to continue if it finds a problem.

The directive:

```
REM {WARNINGS}
```

re-enables the production of warning dialogue boxes when problems are found.

These directives may be used in pairs to mark particular sections of a program.

Throwback suppression

During compilation line numbers of warnings and errors, along with the error text, will be sent via the throwback protocol to any registered throwback capable editors. SrcEdit is an example of a throwback capable editor.

The directive:

```
REM {NOTHROWBACK}
```

will stop ABC from sending throwback. Its inverse, which gives the default situation, is:

```
REM {THROWBACK}
```

Optimisation directives

The compiler has to be very careful when it is generating code to ensure that it always copes with all situations which can occur. In many cases, this leads to it having to include, for example, checks for particular cases and the code to handle any problem ones. Including this code has two detrimental effects - it increases the size of the object file and it makes the object code slower to execute.

There are a number of different compiler directives supplied which each look at one particular area and allow the generation of the extra code to be turned off. If you use any of these, you must ensure that the situation which is no longer being handled never occurs. In some cases it will be possible for the compiler to warn you that the situation has happened, in others it can end up overwriting memory without warning!

Each of these situations is covered by a pair of directives, one of which returns you to the default situation. These can be used to turn the code generation off around a particular area of the program, for example, and then on again later. Alternatively, you can turn the generation off at the beginning of the program and leave it off throughout.

Stack limit checking

The stack grows downwards towards the top of the heap. If the two collide an error is generated. Hence, by default, code is generated to check that the stack hasn't overflowed into the heap. You can instruct the compiler to turn off the generation of the stack checking code by using the directive:

```
REM {NOSTACKCHECK}
```

The alternative is:

```
REM {STACKCHECK}
```

Initialising local variables

By default, the compiler generates code to initialise all local variables to zero. If you are initialising them all yourself explicitly, you can instruct the compiler to turn off the generation of the initialisation code by using the directive:

```
REM {NOZEROLOCAL}
```

The alternative is:

```
REM {ZEROLOCALS}
```

Initialising SYS registers

By default, the compiler generates code to initialise any of the registers R0-R7, which aren't explicitly assigned to in a SYS call, to zero. You can instruct the compiler to turn off the generation of the initialisation code by using the directive:

```
REM {NOZEROSYSREGS}
```

The alternative is:

```
REM {ZEROSYSREGS}
```

Initialising CALL registers

By default, when the compiler compiles a standard CALL/USR statement, it generates code to load the values of the integer variables A%...H% into the registers R0...R7. Similarly, when it compiles an extended CALL/USR statement, it generates code to initialise any of the registers R0-R7, which haven't been explicitly assigned to, to zero. You can instruct the compiler to turn off the generation of the assignment/initialisation code by using the directive:

```
REM {NOCALLREGS}
```

The alternative is:

```
REM {CALLREGS}
```

Falling into PROCs/FNs

By default, the compiler generates code at the start of every procedure and function definition to check that it hasn't been 'fallen into'. This can happen, for example, due to a missing END. You can instruct the compiler to turn off the generation of the incorrect entry checking code by using the directive:

```
REM {NOTRAPS}
```

The alternative is:

```
REM {TRAPS}
```

Escape checking

By default, the compiler will produce machine code which will respond to the Escape key being pressed. This either calls the program's own error handler (if it contains one) or terminates execution with the message ESCAPE. You can instruct the compiler to turn off the generation of the Esc checking code by using the directive:

```
REM {NOESCAPECHECK}
```

Note that this does not disable the normal action of Esc with respect to INPUT operations such as GET. These will still respond in the usual way.

The alternative is:

```
REM {ESCAPECHECK}
```

Array bound checking

Arrays, when they are dimensioned, are given a range for their subscripts. For example:

```
DIM pos%(9,19,29)
```

This means that the first subscript can be 0...9, the second 0...19 and the third 0...29. Hence pos%(4,4,4) is a valid element but pos%(14,14,14) is not because the first subscript is too large.

By default, the compiler generates code whenever an array element is used, so that the subscripts are checked at run-time to ensure that they are within the allowed range. You can instruct the compiler to turn off the generation of the array bound checking code by using the directive:

```
REM {NOARRAYCHECK}
```

The alternative is:

```
REM {ARRAYCHECK}
```

Word alignment

The ! indirection operator stores or reads four consecutive bytes of memory. By default, the compiler assumes that the addresses are not necessarily word aligned, i.e. the addresses at which they start are not necessarily multiples of 4. If they are, you can instruct the compiler to turn off the generation of the general non-aligned code, and hence produce much more efficient code, by using the directive:

```
REM {ALIGNEDPLING}
```

The alternative is:

```
REM {NOALIGNEDPLING}
```

Use of GOTOs

The compiler optimises the object code produced by remembering, in certain cases, what it has stored in its registers. It is able to do this far more effectively if it knows that the program doesn't contain any GOTO or GOSUB statements.

Note: When these statements are used, any line of the program is potentially the target for an arbitrary 'jump'.

By default, the compiler assumes that you might be using GOTOs/GOSUBs. However, you can instruct it otherwise by using the compiler directive:

```
REM {NOGOTOSUSED}
```

The alternative is:

```
REM {GOTOSUSED}
```

If you use the NOGOTOSUSED directive and then try to use a GOTO or GOSUB, the compiler will produce a warning.

Operating system routine names

By default, the compiler assumes that the names of routines called using SYS or SWI have been provided as general string expressions. If you are supplying them as string constants or variables, you can instruct the compiler to turn off the generation of the general code by using the directive:

```
REM {SYSCONSTONLY}
```

The alternative is:

```
REM {NOSYSCONSTONLY}
```

6 Manifest constants

Using symbolic names for constants in your program is generally a beneficial aid to writing programs that are easy to read and debug.

Using manifests

It is generally much clearer to use named variables than to use explicit constants within a program. However, there is a penalty to pay for this as the program will be slower in operation and will consume more memory than if the constant approach is used.

To solve this problem, the compiler supports the definition of *manifest constants*. Hence it is possible to define a named object to represent a constant value throughout the program.

Declaring a manifest

A manifest constant may be defined as follows:

```
DEF height% = 1023
```

Wherever height% is used within the program, the value 1023 will be substituted instead. This helps to make programs much more readable. In addition, it is not possible to assign a new value to a manifest constant. This helps to make programs more secure against accidental alterations of 'constants'.

Similarly, it is possible to define string or floating-point constants. For example:

```
DEF text$ = "some text"
DEF approx_pi = 3.2
```

A good technique is to use these in conjunction with the COMPILE and NOCOMPILE directives to set up a program which will run both under the interpreter and the compiler, whilst at the same time allowing the compiler to generate efficient code.

For example:

```
DEF minimum% = 200
DEF maximum% = 1000
REM {NOCOMPILE}
minimum% = 200
maximum% = 1000
REM {COMPILE}
```

Under the interpreter, minimum% and maximum% will be variables, under the compiler they will be manifest constants.

Static integer variables

You cannot define manifest constants with the names A%...Z% or @%. These names are reserved for the global integer variables which are always pre-declared. For example:

```
DEF A% = 100
```

will generate the error:

```
DEF A% is already defined
```

Numeric values

You can define numerical manifests with a number given in either decimal or hex. However, it is not possible to have negative values. Therefore, the following are allowed:

```
DEF max% = 100
DEF offset% = &2C
DEF max = 2.5E10
```

but the following is not:

```
DEF min% = -100
```

7 Conditional compilation

Coupled with manifest constants, conditional compilation is a powerful technique to select options within the program at compile time.

Using conditional compilation

Through the use of manifest constants, it is possible to control whether the compiler looks at or ignores sections of the source program.

Basic operation

This is best illustrated with an example:

```
10 DEF debug% = 1

100 REM {IF debug%}
110 PRINT TAB(0,0)"Values are "I%,J%
120 REM {ELSE}
130 REM {ENDIF}
```

The manifest constant debug% is set to the value 1 on line 10. This will be taken as TRUE by the IF directive on line 100 and so the compiler will compile line 110. If debug% is set to 0, the IF directive will treat this as FALSE and line 110 will be ignored by the compiler.

Source text between the ELSE and ENDIF directives is only compiled if the condition is FALSE.

Nested operation

It is possible to nest conditional sections within one another. This allows, for example, debug sections to be placed within areas which are already being compiled conditionally for other reasons:

```
10 DEF debug% = 1
20 DEF demo% = 0
```

```
100 REM {IF demo%}
120 PRINT "Can't do - this is a demo version"
130 REM {ELSE}
140 PROCa
150 PROCb
160 REM {IF debug%}
170 PRINT TAB(0,0)"Values are "I%,J%
180 REM {ELSE}
190 REM {ENDIF}
200 REM {ENDIF}
```

In addition, it allows a choice of more than two alternatives to be applied. For example, consider the problem of creating a program so that messages are given in a choice of English, French and German:

```
10 english = 0
20 french = 1
30 german = 0
```

```
100 REM {IF english}
110 PRINT "Hello"
120 REM {ELSE}
130 REM {IF french}
140 PRINT "Bonjour"
150 REM {ELSE}
160 PRINT "Guten Tag"
170 REM {ENDIF}
180 REM {ENDIF}
```

Note that the IF can only be followed by a simple variable - not an expression. For example, the following is not allowed:

```
10 DEF lang$ = "french"
```

```
100 REM {IF lang$ = "english"}
110 .....
```

The IF is interested in just two values:

- zero treated as FALSE
- non-zero treated as TRUE.

Therefore, if the value of the variable is non-zero, the IF part is compiled. Otherwise the ELSE part is compiled.

8 Relocatable modules

RISC OS makes extensive use of modules to extend the functionality of the operating system. With ABC, you can create modules from BASIC programs.

Module types

ABC provides directives which allow the output to be formatted into a relocatable module. There are four types of modules:

- Application
- Utility
- Service
- Library

Library modules are covered in the chapter L*ibraries* on page 63. Each of the others is discussed below.

Application modules

Programs which are compiled into an application module are run via a * command. The program will make use of all the application workspace, or as much as may be specified using the STACK and HEAP compiler directives. When the program finishes, or an untrapped error occurs, it exits by executing OS_Exit (similar to the effect of QUIT from BASIC).

The module can be executed directly on loading via `*RUN` or `*RMRUN`. It can be re-executed using the * command which it provides.

Utility modules

This is much like an application module with the exception that the program does not make use of the application workspace. Instead, it uses memory claimed from the relocatable module area. When the program finishes, or there is an untrapped error, it exits by executing OS_Exit.

The module can be executed directly on loading via `*RUN` or `*RMRUN`. It can be re-executed using the * command which it provides.

Service modules

Again this works in a similar fashion to an application module in that the program is called by a * command. As with utility modules the program will not make use of the application workspace, instead it uses memory claimed from the relocatable module area. When the program finishes, or there is an untrapped error, it exits by returning to the calling program. Thus a service module provides a command which can be called up from within another program.

The program which is incorporated within a service module is executed with the CPU in supervisor mode. This means that certain limitations are imposed:

Assembler subroutines

Be very careful if your program includes any assembly language. In particular, you must ensure that assembler subroutines preserve the processor mode. Also, you should note that the use of SWI instructions when in supervisor mode causes register R14 to be corrupted. Thus, a subroutine which calls a SWI must preserve R14 on the stack.

To make use of floating point assembler instructions you must ensure the Floating Point Emulator version 4.09 or later is present. Prior versions did not work when called from supervisor mode without first manually preserving R14 on the stack.

Execution of the module directly via `*RUN` or `*RMRUN` will cause the module to be loaded and initialised. To execute the embedded program you must issue the * command which it provides.

Module compiler directives

A number of directives are provided to specify the details of the module:

Module type

The type of module required must be specified using the MODULE TYPE directive, thus:

```
REM {MODULE TYPE <type>}
```

The three possible types should be specified as follows:

```
REM {MODULE TYPE APPLICATION}
REM {MODULE TYPE UTILITY}
REM {MODULE TYPE SERVICE}
```

Title string

Use the directive

```
REM {MODULE TITLE <string>}
```

to set the title string of the module to <string>.

Version string

Use the directive

```
REM {MODULE VERSION <string>}
```

The version string of the module will be set to <string>. This will be incorporated into the module help string as given in response to:

```
*HELP MODULES
```

The version string should be of the form x.yz.

Command string

This is the command which is used to run the module.

```
REM {MODULE COMMAND <string>}
```

If the command string expects any arguments it should be followed by the number required:

```
REM {MODULE COMMAND <string> <args>}
```

If some of the arguments are optional then both the minimum and the maximum number should be given:

```
REM {MODULE COMMAND <string> <min> <max>}
```

Command help string

The command should be provided with a help string. The text of the help string will begin with the command and then be followed by the text specified by this directive:

```
REM {MODULE HELP <text>}
```

Module memory

For utility or service modules which use their own RAM rather than the application RAM, it is necessary to specify the amount of memory which will be claimed when the module is initialised.

```
REM {MODULE MEMORY = <size in bytes>}
```

The size must be at least 8 kilobytes. Usually it is possible to have a good guess how much they need. It is simpler to overestimate than to spend time trying to calculate the exact figure.

The way to work out how much RAM your module will require is a long and complex process, which goes roughly as follows:

1. Count the number of real variables and multiply by 4, 8 or 12 depending on whether they are single-, double- or extended-precision variables.
2. Count the number of integers and multiply by 4.
3. Count the number of strings and multiply by their maximum length.
4. Add the amount of RAM which has been DIMmed for use with assembler.

You must remember to include local as well as global variables in your calculations.

32 bit compatibility

To mark the resulting module as being 32 bit compatible in its header, add the directive:

```
REM {MODULE 32BIT}
```

Recall that a module marked as 32 bit compatible module can still be loaded on a 26 bit system, but requires a more recent ABCLibrary as described in *Compatibility* on page 2.

Command argument tail

Star commands are used to invoke module commands and quite often the * command may require extra information which would generally be typed after the command.

ABC provides a mechanism that allows the command line to be investigated so that information such as filenames can be extracted. This is done by assigning the * command string to the variable LINE$.

This variable is like any other BASIC string variable and can be treated as such. It can therefore be scanned to see how many parameters have been passed, and you can work out what they were. You can use this process of investigating LINE$ to make your module respond to more than one * command. All you would have to do is to look at the first parameter in LINE$ and respond accordingly, using a CASE...ENDCASE statement.

For example:

```
100 N% = 1
110 WHILE MID$(LINE$,N%,1) = "":N%+=1:ENDWHILE
120 REM Misses out leading spaces
130 REM Start% = N%
140 WHILE MID$(LINE$,N%,1) <> " " AND N% < LEN LINE$:N%+=1:ENDWHILE
150 Command$ = MID$(LINE$,Start%,N%-1)
160 REM We now have the first word
170 CASE Command$ OF
180 WHEN "Save": PROCsave
190 WHEN "Load": PROCload
200 ENDCASE
```

SWI handlers

Service modules are capable of supporting SWI handlers. We don't recommend that you use BASIC to create SWI handlers for commercial products because they need to execute extremely quickly and so should be written in highly optimised assember.

However, you may like to experiment with SWI handlers out of interest.

You need to include one extra directive in your program:

```
REM {SWIBASE = n}
```

to define the base of the SWI chunk you wish to use. You should apply to RISC OS Open Limited for a SWI chunk to ensure that you don't clash with any existing products. Hence the value of n will be different for everyone.

In addition, there are two functions which are useful:

SYSDATA

This returns the address of a block of memory containing the values of R0 to R7 as set up when the SWI was issued. You can use:

```
reg0% = SYSDATA!0
reg1% = SYSDATA!4
.....
reg7% = SYSDATA!28
```

to obtain the values. In addition, you can pass back values as follows:

```
block = SYSDATA
block!0 = R0%
block!4 = R1%
.....
block!28 = R7%
```

before returning.

SYSFN

This returns the SWI number code actually used. Hence, if you are supporting more than one SWI, you can use this function to identify the one required. You can set up names for the SWIs as follows:

```
DEF SYS Init = 0
DEF SYS Exit = 1
DEF SYS Error = 2
```

where the numbers are their offsets from SWIBASE. For example:

```
CASE SYSFN OF
WHEN Init: PRINT "Initialise"
  FOR I% = 0 TO 7*4 STEP 4
    PRINT "R";STR$(I% DIV 4);"=&";~I%!SYSDATA
  NEXT I%
WHEN Exit: PRINT "Exit"
  FOR I% = 0 TO 7*4 STEP 4
    I%!SYSDATA = I%!SYSDATA + I %
  NEXT I%
WHEN Error: ERROR 1,"Get away"
ENDCASE
```

An example module

The following example module illustrates the use of the various directives to produce a service module. The module provides the command SCREEN which must be followed by a parameter. This parameter is the textual name of the colour to which the screen background will be set.

```
10 REM >Examples.Screen
20
30 REM {MODULE TITLE DemoModule}
40 REM {MODULE VERSION 1.00}
50 REM {MODULE TYPE SERVICE}
60 REM {MODULE COMMAND Screen 1}
70 REM {MODULE HELP sets screen background colour}
80 REM {MODULE MEMORY = 8192}
90 REM {MODULE 32BIT}
100
110 I% = INSTR(LINE$," ")
120 Command$ = LEFT$(LINE$,I%-1)
130 Command$ = FNUPPER_case(Command$)
140
150 CASE Command$ OF
160 WHEN "RED" : COLOUR 0,1
170 WHEN "YELLOW": COLOUR 0,3
180 WHEN "GREEN" : COLOUR 0,2
190 WHEN "BLUE" : COLOUR 0,4
200 OTHERWISE
210 ERROR 1,"Bad Colour"
220 ENDCASE
230 END
240 DEF FNUPPER_case(A$)
250 LOCAL I%,B$,C$
260 FOR I% = 1 TO LEN(A$)
270   C$ = MID$(A$,I%,1)
280   IF "a" <= C$ AND C$ <= "z" THEN
290     C$ = CHR$(ASC(C$)-32)
300   ENDIF
310   B$+=C$
320 NEXT I%
330 =B$
```

This module makes use of the LINE$ function to read the text of the command line which was used to start it. The operating system will already have ensured that only one command-line argument was present following the SCREEN command as requested in the module specification. Thus, all that is necessary is to skip to the first space which is found using the INSTR function, take the rest of the string and convert to upper case.

The CASE statement then checks to see whether the argument is one which it recognises and performs the necessary COLOUR commands if it is. An error is generated if the argument was not one of the colours which the module provides. This error will be reported back to the program which issued the *SCREEN command.

If an incorrect number of parameters is given, then the operating system causes an error message to be generated. This message is constructed by the compiler from the information about the module and is bound into it when the module is made. It will indicate what syntax the command will accept.

The module requires the minimum amount of workspace, i.e. 8192 bytes (8kB).

Once the module is compiled and loaded, the *SCREEN command can be used from another program.

For example:

```
10 REPEAT
20 INPUT colour$
30 OSCLI ("SCREEN " + colour$)
40 UNTIL FALSE
```

9 Libraries

One variation on modules described earlier is a library, where a collection of frequently used routines can be kept together to be reused many times.

Library modules

Besides allowing you to compile stand-alone programs, ABC also provides the facility for producing libraries of pre-compiled procedures and functions. These libraries are in the form of relocatable modules which can be activated and killed as required.

When one of these modules is activated, the procedures and functions within it can be called from 'normal' programs compiled with ABC. These procedures and functions are shared by all such programs which are currently running.

This feature allows you to split off frequently used procedures from your main programs. Consequently, your programs become smaller and compile quicker. In addition, you only have to worry about one version of each library procedure and you can be sure that each program is calling exactly the same code.

Making a library

You can choose to have your source code compiled as a library by using the LIBRARY compiler directives. These will cause the compiler to produce a library module which will automatically provide one star (*) command which lists the names of the procedures and functions within it, hence producing a catalogue of the library routines it contains.

For example, including the directives:

```
REM {LIBRARY TITLE BasLib1}
REM {LIBRARY VERSION 1.00}
REM {LIBRARY INFO BasLib1Info}
REM {LIBRARY 32BIT}
```

at the top of your source program will result in ABC producing a library module called BasLib1 whose * command is *BasLib1Info.

Not all versions of ABC can produce 32 bit libraries, see *Compatibility* on page 2 for more details.

Accessing library routines

To call a library procedure or function you must provide a dummy definition within the main program. This enables the compiler to produce the external procedure call code. It also allows the compiler to check that the call to the routine has the right number of parameters and that their types are correct.

Accessing procedures

A dummy definition for a procedure takes the form:

```
DEF EXT PROCprocname(arg, ..., arg) ENDPROC
```

The following should be noted:

- The EXT and PROC must be separated by at least one space.
- The whole definition must be on a single line.
- The procedure cannot contain any code.
- The ENDPROC is not preceded by a colon.

For example:

```
DEF EXT PROCfred(first%, second$, RETURN third)
ENDPROC
```

or

```
DEF EXT PROCjim ENDPROC
```

Accessing functions

Dummy definitions of functions are similar to those of procedures. The one difference is that they are terminated by giving the type returned rather than by the keyword ENDPROC.

For example:

```
DEF EXT FNfnname(arg, ..., arg) = INTEGER
```

or

```
DEF EXT FNfnname(arg, ..., arg) = FLOAT
```

or

```
DEF EXT FNfnname(arg, ..., arg) = STRING
```

Note that the type of floating point number is irrelevant. FLOAT applies to functions returning either single-precision, double-precision or extended-precision values.

For example:

```
DEF EXT FNadd(a1, a2) = FLOAT
```

Implementing library routines

A library has to be able to distinguish between procedures and functions which can be referenced from other programs and those which are local to the library itself. Any routine which is to be callable externally has to be defined as:

```
DEF LIBRARY PROCprocname(arg, ..., arg)
```

or

```
DEF LIBRARY FNfnname(arg, ..., arg)
```

Procedures and functions defined in the normal manner are treated as being local to the library and cannot be referenced from the main program.

When ABC comes across a call to an external routine in the main program, it searches through all the currently active libraries until it finds a routine of the correct name. Up to 32 libraries can be active at any given time. If a routine with a particular name occurs in more than one of these then the first one found will be used, the others will be ignored.

Note: A procedure in one library may call an external routine from another library in the same way as the main program can. The main program checks the parameters used when an external routine is called against those of the dummy definition. It does not check them against those of the external routine. It is therefore very important to make sure that the external definition matches the dummy one exactly.

When supplying libraries to other people, you should also supply them with a file containing the appropriate dummy definitions for them to append to their main programs.

Note that if a procedure is declared as a library procedure, it cannot be called directly from within that library (i.e. from the same source file). It must always be called as a library procedure. Thus the following is illegal:

```
DEF LIBRARY PROCpr1(x,y)
PROCpr2(y)
ENDPROC

DEF LIBRARY PROCpr2(x)
PRINT x
ENDPROC
```

However, if PROCpr2 is not declared as a library procedure, then it becomes allowed. This has an implication for recursive procedures in libraries. Each recursive procedure, within the body of a library, which is to be exported must be declared in two parts; the main procedure itself and a dummy library header.

For example:

```
DEF LIBRARY FNfactorial(x)
= FNfac(x)

DEF FNfac(x)
IF x <= 1 THEN
  = 1
ELSE
  = x*FNfac(x-1)
ENDIF
```

Restrictions on libraries

There are certain restrictions imposed on the source files which are to be converted into Library modules. These are listed below.

Global variables

Library routines do not have access to any of the variables of the main program, other than those passed to it as parameters and the static integer variables @%, A%, B%, ..., X%, Y%, Z%. They cannot create their own global variables. Any attempt to create or read a global variable will result in an error message being given at compile time.

Since all arrays are global, this means that library routines cannot use arrays. Hence DIM may not be used to initialise one.

GOTO/GOSUB

GOTO/GOSUB statements within library routines would be ambiguous since it is not clear whether the line numbers they refer to are in the library code or the main code. Therefore they are not allowed.

RESTORE, READ, DATA

To be fully versatile, library routines need to be able to handle different data depending on the program calling them. ABC adopts the convention that all data read by library routines refers to data in the main program. This is similar to the view taken by the BASIC interpreter to its LIBRARY routines:

- RESTORE restores to a line in the main program.
- READ reads data from the main program.

When ABC encounters a READ statement within a library source file, it will issue a warning to remind you that this is the case.

Note: DATA is not allowed since data in the library can never be accessed.

An example library

The `Examples` directory contains some example files:

- StringFns the source of four functions for manipulating strings
- StringLib a module containing these routines as library functions
- StringDum the dummy definitions for appending to calling programs
- LibUse the source of a program which uses them

StringLib was produced by compiling the StringFns file as a Library module. To install the library module type:

```
*StringLib
```

Then type:

```
*Stringroutines
```

to list the names of the routines it provides.

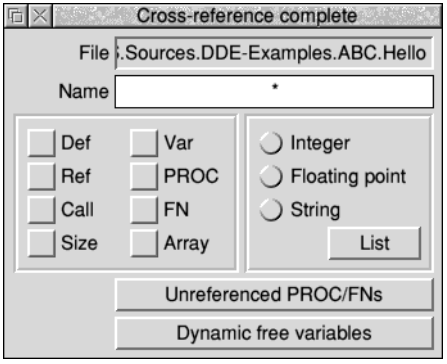
LibUse, which uses these routines, has already had the dummy definitions of them appended to it. Compile this program using ABC in the usual manner and then run the object code by double-clicking on it. It will behave as a normal program.

10 Cross referencing

Cross referencing provides a detailed analysis of the of the use of variables, arrays, procedures and functions within a BASIC program.

Using the X-ref option

To ask the compiler to build up cross-referencing information, enable the X-ref option in the Options menu as described in *Compiler options* on page 12. The next time you compile a program, when the compilation is complete, the following dialogue box will appear:

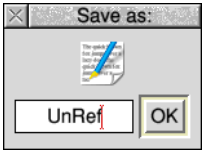


Extra report detail

The two options at the bottom allow you to check the program for:

- Any PROC/FNs which are never executed. This could be because no calls to them exist or because they are only called from PROC/FNs which themselves are never called etc. The only effect these routines are having is to make your object code larger.
- Dynamic free variables. These are described in *Scope rules* on page 19. The existence of dynamic free variables is one of the common causes of programs producing different results under the compiler and interpreter.

To check your code for either of these, click on the appropriate option box. The following dialogue box will appear:



The default filename will be `UnRef` or `DFV` depending on whether the **Unreferenced PROC/FNs** or **Dynamic free variables** option was chosen. You can change this, if you wish. Then, drag the file icon onto a directory viewer. A file will be created which can be examined using Edit or a similar word processor.

Unreferenced procedures/functions will be reported as follows:

```
Examining PROCunused
PROCunused is never called
```

Dynamic free variables will appear as:

```
Checking PROCz
Warning: References to x% in PROCz are ambiguous!
```

Report information

The other options all work together to allow you to obtain various information about objects of a particular type and name.

The default configuration is such that the cross referencer will provide all possible information about all objects within your program when **List** is selected.

Name

Allows a wild-carded name to be entered. Then only objects of that name will be checked. You can enter specific names such as colour. Alternatively you can ask for a range of names such as col* which will match all objects whose names start with the letters col.

The most general name is *, the default, which matches all names.

Def

Provides the definition of the object, for example:

```
F.P. Number Global e
PROCa
Integer Local x% belonging to PROCa
```

Ref

Lists the references made to the objects.

For example:

```
F.P. Number Global e
This e is referred to from main program as a Global

PROCa
PROCa refers to -
Integer Local x% belonging to PROCa Twice
PROCb
```

Call

Provides the calling sequence for procedures and functions.

For example:

```
PROCa
PROCa is called from -
  *** Main program ***

PROCb
PROCb is called from -
  PROCa
    *** Main program ***
```

Procedures and functions which are referenced should all lead back to the Main program. Recursion is dealt with. A cut off occurs once the loop has been listed twice.

Size

This option provides details of the sizes of arrays, procedures and functions.

For example:

```
PROCa
PROCa uses 24 bytes of stack
```

This allows you to calculate how much stack space your program will require.

Matching on constructs

Var

If this is selected, variables which match the name will be checked.

PROC

If this is selected, procedures which match the name will be checked.

FN

If this is selected, functions which match the name will be checked.

Array

If this is selected, arrays which match the name will be checked.

Int, FP, Str

Allow variables, functions and arrays to be restricted to either integers, floating points or strings, depending on which of the radio buttons is selected. If non are selected, then all types will be allowed and checked.

List

Carries out the checking, based on the current settings. Clicking on this option displays a **Save as** dialogue box. The default name will be `List`. The checking takes place when this is dropped onto a directory viewer.

Appendix A: Significant changes

This chapter details the significant changes made to the ABC compiler, for the period where records exist, and the companion ABCLibrary.

Version history

Version 3.10

- The final version from Oak Solutions. Some copies of 3.13 and 3.14 exist, but were never formally released.

Version 3.15

- Pineapple Software takes over development.
- First version to be StrongARM compatible, with the introduction of MANUALCODESYNC directive. Released with ABCLibrary 4.05.

Version 4.10

- First version to be suitable for running in 32 bit mode, and generating 32 bit programs. Released with ABCLibrary 4.12.

Version 4.11

- R8 may now be used with SYS.
- The Edit button on the warning/error window now sends a DataLoad message in the hope that a suitable editor is loaded (e.g. SrcEdit).
- A scenario where the compiler could generate an infinite sequence of errors has been fixed.
- Throwback support added, which will be used automatically if DDEUtils is loaded. Thanks to R-Comp for assistance with this.
- New template file added. Thanks to Paul Reuvers.

Version 4.12

- New options {THROWBACK} (default) and {NOTHROWBACK}.
- New options {SQUEEZE} and {NOSQUEEZE} (default).

- Several minor user interface improvements to the ABC windows.

Version 4.13

- Introduced {MODULE 32BIT} and {LIBRARY 32BIT}, though using ABC to generate module code remains deprecated.
- ABCLibrary 4.13 fixes a bug around reading the returned flags on a 32 bit processor from CALL, USR and SYS calls where the SWI number is resolved at runtime rather than compile time.

Version 4.14

- Fixes a bug where {ZEROLOCALS} wouldn't take effect whilst the Quick compilation switch was turned on.
- ABCLibrary 4.14 fixes a bug where EQUW would fail to generate any data on 32 bit systems.
- ABCLibrary 4.15 fixes a bug with top bit set hexadecimal numbers being set to zero in DATA statements.

Version 4.16

- Introduced the {AIFHEADER} directive. Some copies of 4.15 and 4.17 exist, but were never formally released.

Version 4.18

- Accepts END as a function and the pseudo variable END= to adjust the application slot size, restoring it on QUIT.
- Introduced the {AIFFLAGS} directive.
- Fix to propagate the error message on return from a SWI in a service module when the ERROR keyword is used. Previously R0 was restored and so the pointer to the error block was lost.
- Internal changes to avoid accidentally accessing zero page and unaligned memory locations, and to ensure cache coherency before calling machine code instructions produced with ABC at run time.
- Library modules no longer abort on 32 bit systems during initialisation.
- ABCLibrary 4.18 fixed to compute the addresses of external PROCs and FNs in library modules correctly on 32 bit systems, required because the library module is now further from application memory than the range of a branch instruction.

Version 4.20

- Hexadecimal numbers are now accepted in lower or mixed case. This requires a corresponding update to ABCLibrary 4.20 for numbers parsed at run time.
- Made END as a function to match the description given on page 27, fixing a bug introduced in 4.17. Assignments to END= set the application slot size as described in the change log for 4.18, above.
- Modules created by ABC now use the datestamp of the source file as the date show in the help string, rather than today's date.

Index

Symbols

A

C

D

E

I

L

M

O

S

Reader's Comment Form

Archimedes BASIC *Compiler, Issue 3*

We would greatly appreciate your comments about this Manual, which will be taken into account for the next issue:

Did you find the information you wanted?

Do you like the way the information is presented?

General comments:

If there is not enough room for your comments, please continue overleaf

How would you classify your experience with computers?

☐ **Used computers before** ☐ **Experienced User** ☐ **Programmer** ☐ **Experienced Programmer**

Please send an email with your comments to:

manuals@riscosopen.org

Your name and address:

This information will only be used to get in touch with you in case we wish to explore your comments further