ABC 4.14 Release Notes

ABC originally written by Paul Fellows for Solent Computer Products
Current maintainer: Alan Glover for Pineapple Software
Current distributors: Castle Technology

1 Introduction

ABC is a compiler for programs written in BBC BASIC, producing programs which run faster than the original interpreted code, and are often smaller too. ABC works on all versions of RISC OS above 3.00 and with 26 and 32 bit processors.

BASIC programs compiled with ABC 4.14 must be run in conjunction with the ABCLibrary module, version 4.14 or later. Programs compiled with this combination will run on 26 or 32 bit platforms with no modifications necessary. Programs compiled with earlier versions of the compiler will run with ABCLibrary 4.14 on 26 bit platforms but will be faulted on 32 bit platforms.

Applications compiled with earlier versions of ABC must be recompiled with ABC 4.14 or later to run 32 bit. This is due to changes in the linkage scheme between the application and ABCLibrary. In addition various 32-bit-illegal code sequences have been eliminated from both compiler (and hence future compiled code) and ABCLibrary.

In the past one of the selling points of ABC was the generation of ‘super-fast machine code’. With improved processors, more memory, and larger disk 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.

2 Installation

There are two components to ABC; !ABC – the compiler application, and ABCLibrary which provides runtime support to ABC-compiled programs (which includes the compiler itself).

Having older versions of ABCLibrary around can cause unstable behaviour, particularly if a newer ABCLibrary is loaded whilst another application is still using an older one. We recommend searching through your hard disk(s) and replacing every instance of ABCLibrary that you find with 4.14, There is only one exception to this advice – if you find a version inside !Killer is out of date do not replace it; !Killer will detect the change and refuse to run. !Killer will however run if a newer ABCLibrary than it RMEnsures is already in memory.

!ABC requires no installation other than copying it to a location of your choice.

3 Getting Started

Compilation of programs is very simple. To try this out create a simple BASIC program such as

```
10 PRINT “Hello brave new world”
```

and save it to disk.

Now double click on !ABC to load it. It will indicate its presence by the appearance of an icon in the right hand group of the icon bar. If an error is generated indicating that DDEUtils could not be loaded (it will be in !System if you have the C/C++ tools installed), comment out the RMEnsure line for DDEUtils in !ABC.!Run and then double-click again on !ABC to load it.

Drag the BASIC file onto this icon. A save box will pop up. Drag this back to the same directory as the BASIC program. !ABC will then begin the compilation. A Compilation Progress window will be shown indicating input and output filenames, number of warnings generated, and progress through the file. Once the compilation is complete this will be replaced by the Compilation Complete window which reports some statistics about the program.

To try out the compiled program simply double-click on the newly created output file. It will display the message in the PRINT statement and then you will be prompted to press SPACE or click with the mouse to close the command window used for the text display.

ABCLibrary

ABCLibrary must be loaded for ABC-compiled programs to run. In the event that ABCLibrary is not present the error ‘ABCLib not loaded’ will be given. However, any desktop application should be written to rmensure for the minimum version of ABCLibrary that it needs. Doing this needs to be slightly more sophisticated than normal to avoid causing instability by replacing the ABCLibrary a program was already using with the newer version. The sequence below is recommended:-

```
RMEnsure ABCLibrary 0.00 RMLoad System:Modules.ABCLib

RMEnsure ABCLibrary 4.12 Error An older version of
ABCLib is already loaded
```

The first RMEnsure checks for ABCLibrary in memory and loads it from !System if necessary. The second RMEnsure checks the version in memory (whether just loaded or already resident) to ensure that it is recent enough.

Permission is granted to ship the most recent version of ABCLibrary with any program that requires it. The recommended minimum ABCLibrary for any given version of the compiler may be found inside the compiler’s directory, and will often be shipped as part of a computer’s disk content within !System too.

4 Warnings and errors

Usually trying to compile a program of any size will result in Warnings or Errors for the first few attempts at compilation. ABC is more rigorous than BASIC in its checking of the program (this is essential since it must be able to interpret the source unequivocally in order to render it correctly as compiled code) and also imposes a small number of restrictions on the structure and content of the BASIC code.

Reporting using Throwback

ABC will attempt to report warnings and errors via the throwback system used in other compilers/assemblers. This requires the DDEUtils module to be loaded (ABC rmensures for it when started) and a throwback-capable editor such as Zap or StrongED which also understands BASIC programs. If you have neither you can use SRCEdit to manage the throwback window, but will have to perform the actual editing with Edit. If you are using Zap or StrongED you may double-click on a warning/error in the throwback window to have the editor open the source file on that line.

If you wish to suppress the use of throwback by default include the `REM {NOTHROWBACK}` directive in the source.

Reporting without 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. The window has Edit and OK buttons. The Edit button attempts to get an editor such as Zap, StrongED or Edit to load the source file (but unlike throwback, you are not taken automatically to the line that caused the problem). The editor must be loaded for this feature to operate. The options at this point are either to click on OK to continue compilation after a warning or click Abort in the progress window to abort the compilation so that the source may be corrected. If an error has been reported clicking on OK will end the compilation and close ABC. If the Warning/Error window has obscured the Abort button on the progress window simply drag the Warning/Error window out of the way.

When OK is selected the compiler will attempt to complete the compilation using its best interpretation of the intended meaning of the line. In most cases we recommend doing one compilation to take note of all the warnings, selecting OK each time, but do not try running the code – go back and fix the warnings and then compile again before testing. A warning can be interpreted as an indication that the compiler is having to guess what you really wanted, and it is unlikely to guess right every time.

Warnings and Errors

An Error differs from a Warning in that ABC is unable to continue the compilation. Once again Edit and OK options are presented, but the recommended action at this point is simply to note the error, abort the compilation from the progress window, and then fix the problem.

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.

In some cases more warnings will be counted than are reported. This comes about either because the Pause option is turned off or the same effect has been achieved explicitly in the code by use of the `REM {NOWARNINGS}` directive.

5 Compiler Options

Clicking the Menu button on !ABC's icon once loaded brings up a menu with Info, Options, Save Options and Quit.

Options leads to a dialogue box which presents these options:-

X-ref

When X-ref is on cross-referencing information is accumulated during compilation and will be displayed upon successful completion

Quick

Produces smaller and faster code by asserting a number of directives about the code content. This option should only be used if you are confident that your program is safe. The directives are:-

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

Compiler directives are covered in more detail later in this Release Note.

While most of these are unlikely to cause problems, `REM {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 which involve a lot of head scratching to find. 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.

RAM

This directs the compiler to hold both input and output files in memory rather than accessing them via file operations. Compilation will be significantly faster, but more memory will be used.

Pause

When on the compiler will pause on each Warning. When off it will continue and only present a count of the total warnings at the end of compilation. Initially this option should be turned on so that all warnings are displayed, but once a program is stable compiler directives can be used to turn Pause off when it is known that warnings will be generated and back on for parts of the source that are not expected to generate warnings.

Code Size

This allows you to preset a memory buffer size for the output code. It should be left empty to direct the compiler to manage this automatically. As well as making sure that the input field is empty, also ensure that the option is not ticked.

6 BASIC and ABC compared

This section notes the rules that ABC imposes above and beyond those required by the BASIC interpreter.

Structures

The following rules apply:-

- There must be only one UNTIL for each REPEAT
- There must be only one ENDWHILE for each WHILE
- There must be only one NEXT for each FOR
- Multiple PROC exits are allowed. The body of the procedure begins with the DEF and persists until either:-
  - o An ENDPROC which is not within a structure is found
  - o The end of the program is reached
  - o Another DEF is encountered
- Multiple FN exits are allowed. The rules for PROCs above apply, plus:-
  - o A function must return the same data type from all its exits
- Multiple entry points should be avoided.

Keyword positioning

The following rules apply:-

- THEN and OF must still be the last item on a line
- WHEN, OTHERWISE, ENDCASE, ENDIF do not have to be the first items in a line (although we strongly recommend avoiding the exploitation of this, since it will create a program that operates differently when compiled and when run as BASIC)
- Dangling ‘ELSE’
  - o Consider: `IF A%>B% THEN IF A%>C% THEN PRINT “A” ELSE PRINT “C” ELSE PRINT “B OR C”` – BASIC will only ever print A or C. ABC will resolve the nesting and give the expected output. However, once again we do not recommend writing potentially ambiguous code like this – with modern systems there is little excuse not to lay out structures over multiple lines with indentation to aid readability.
- NEXT,
  - o The compiler does not permit NEXT, to be used as a substitute for NEXT:NEXT
  - o In the case of `FOR I%=… FOR J%=… NEXT I%` the compiler will fault the implied NEXT J%

Scope Rules

The compiler considers that all variables are global, save when explicitly defined LOCAL. Notably when a LOCAL variable is passed to another procedure the interpreter will use the local value, but the compiler will use the global value.

Expressed another way, LOCAL variables are only in scope within their procedure/function – any calls out will cause the global value to be used instead.

Avoid GOTO/GOSUB to jump between bodies of procedures or functions – apart from being poor programming practice, the compiler cannot figure out what the scope rules should be at that point.

Error handlers

Local error handlers may be defined within a procedure or function with LOCAL ERROR <statement> or ON ERROR LOCAL <statement>. The local handler will be inherited by any procedures or functions called from the one where the local error handler was established.

Local error handlers can be explicitly turned off with LOCAL ERROR OFF, ON ERROR LOCAL OFF, RESTORE ERROR, or by returning from the procedure/function which defined them.

The handler established with ON ERROR at global level will be called unless a local error handler is in scope. A global handler can be turned off with ON ERROR OFF.

ERL will always return zero.

Stacked GOSUB/RETURN is lost when an error is reported; use of PROC and FN is strongly recommended instead.

Floating point

Double-precision variables can be declared by using a back-tick (ASCII 96) suffix, eg A`. Variables suffixed with &, eg A&, are extended precision.

If a variable name ends with a-z, A-Z, 0-9 or _ it will be treated as single precision.

BASIC does not allow & as the last character of a variable name.

Print formatting and @%

Bits 25 and 24 define how PRINT# writes to a file:-

- 00 – single 4 byte
- 01- Acorn 5-byte
- 10- double 8 byte
- 11 – extended 12 byte

Compiled programs using INPUT# will detect these formats, but uncompiled programs will not.

If Bit 31 is set STR$ follows the print format setting, if clear it does not.

Bits 20-16 specify the number of significant digits to print.

Bits 12-8 specify the number of decimal places for fixed format numbers.

Bits 7-0 specify the field width to print the number in.

The interpretation of the top byte of @% can be modified with the TYPE directive.

Numeric Input

Under the interpreter INPUT only accepts decimal numbers whilst READ is more versatile. ABC only allows numbers, but with these extensions:-

- Optional +/- sign
- Optional %/& to indicate binary or hex
- Valid digits (depending on number base in use, ie 0-1, 0-9 or 0-F)
- An optional decimal point followed by more valid digits
- An optional E for exponent followed by an optional +/- and up to four decimal digits

VAL

ABC allows VAL to use a hex string, eg VAL("&"+addr$)

RETURN Parameters

Up to eight parameters may be returned. Each must be a simple variable or integer array element. Indirect expressions, eg `base%!4` are not allowed.

Arrays

ABC supports all BASIC IV array handling features. However, there are some limitations compared to BASIC V:-

- Whole array operations are not supported
- LOCAL array handling is not supported
- Array elements may not be the control element of a FOR loop

Indirection operators

There are some restrictions on assignment to indirect expressions. Valid examples:-

- PRINT $(buffer%+I%)
- PROCfred(A%?B%)
- $(buffer+I%)="Hello"+CHR$0

However these are not valid:-

- 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"

Programming environment pseudo variables

HIMEM – top of memory
END – top of heap
LOMEM – bottom of heap
TOP – top of program
PAGE – start of program
EXT – current stack pointer
EXT LIM – bottom of stack
QUIT – address of exit handler
The default memory organization is:-

- PAGE – program

- TOP – workspace
- LOMEM – heap
- END/EXT LIM – free stack space
- EXT – used stack space
- HIMEM – 4K workspace at top of allocated memory

If the `REM {NEWHEAP}` directive is being used this changes to:-

- PAGE – program
- TOP – workspace, fixed heap
- EXT LIM – free stack space
- EXT – used stack space
- HIMEM – 4K workspace
- LOMEM – start of dynamic heap
- END – top of dynamic heap

The compiler does allow indirection on these pseudo variables, eg `PRINT END?1`.

Assignments to these pseudo-variables are not permitted

Calling machine code

CALL/USR <expression> results in A%-H% being loaded to R0-R7.

When CALL/USR is used inside a procedure/function it will first look for LOCAL A%-H%. If they do not exist a Warning is given and global ones used instead.

Extended syntax: CALL/USR <expression> (param1,….,param8)

Extended syntax: CALL <expression> (param1,…,param8) TO var1,…var8;var9

The return from USR will be in R0.

See the `REM {MANUALCODESYNC}` directive for information about StrongARM code cache issues.

Compiler 4.11 onwards allows R8 to be used with SYS.

Using assembly language in compiled programs

If you use the inline assembler the code is not assembled until run time (just as would happen under the BASIC interpreter). ARM architectures 4 and 5 are not supported.

The interpreter pre-defines r0-r15, R0-R15, pc, PC, pC, Pc. These must be defined explicitly as having values 0-15 corresponding to their register numbers in the compiler. The directive `REM {REGISTERS}` can be used to define them all.

OPT bit 1 for listing is supported, however some information is no longer available once compiled, notably the name of the label.

Additional EQU directives:-

- EQUFS – single precision floating point
- EQUFD – double precision floating point
- EQUFE – extended precision floating point
- EQUF – floating point (as governed by TYPE directive)

Banned keywords

The compiler will give a warning and ignore any of:-

- AUTO
- LVAR
- DELETE
- NEW
- EDIT
- OLD
- HELP
- RENUMBER
- INSTALL
- TWIN
- LIBRARY
- TWINO
- LAST
- TRACE

These are legal in the interpreter, but not allowed in the compiler:-

- APPEND
- CHAIN
- LOAD
- SAVE
- COUNT
- EVAL
- SUM
- WIDTH

TAB and SPC operate slightly differently. TAB(n) outputs spaces until POS(n). SPC(n) outputs n spaces

The omission of EVAL places some restrictions on DATA/READ. Consider READ A%,B%:DATA X*StepX,10*20. Both these data statements are effectively EVAL instructions. Since these could be legal strings the compiler cannot fault these at compilation time, but will not evaluate the expressions at run time.

CLEAR

CLEAR has a different function in the compiler. It is used in conjunction with the `REM {NEWHEAP}` memory model to release an area of claimed memory. The syntax is CLEAR <expression>. See the compiler directives section for more information

7 Compiler directives

Compiler directives all take the form `REM {directive}.` Most also have a complementary form `REM{NOdirective}` to turn that facility off. Directives take effect from wherever they occur in the source. Directives can be placed at the start of the program to take effect for the whole compilation, or be used to surround specific areas (eg to turn the pause on Warning feature on or off).

Code Cache Synchronisation (StrongARM)

On StrongARM there are separate data and code caches. Without getting into too much detail the net effect of this is that if the code changes dynamically at runtime (eg by using the inline assembler) the processor must be told to update its caches to ensure that it doesn't use a cached version of the memory as it was before the inline code was built.

To preserve backwards compatibility with older programs the StrongARM ABCLibrary (4.05) 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, loaded from disk, etc. 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 synchronization, and it does not need to do the cache flush on every CALL/USR. The programmer would then code in cache flushes only when they are needed, ie after building inline code, or loading a code fragment from disk into the program space. This directive is supported in !ABC compiler 3.15 forwards.

The default is NOMANUALCODESYNC, and this can also be set explicitly with `REM {NOMANUALCODESYNC}.` Programmers are strongly advised to use MANUALCODESYNC though.

Ignoring sections

Sometimes there are situations where code should only execute when interpreted or only execute when compiled. `REM {NOCOMPILE}` directs the compiler to skip everything that follows until a `REM {COMPILE}` is encountered. Typical uses for this include debugging output that should not appear in the compiled program, or cases where different code is needed in each case (though we recommend writing source that works the same compiled or interpreted by obeying the various rules ABC imposes). Another necessary use of this is to avoid the interpreter faulting ABC's extensions, eg the CLEAR <expr> extension when using the `REM {NEWHEAP}` memory model.

Memory usage

These directives are for advanced use – ABC will apply sensible defaults. When a compiled program is running there are three areas of workspace; the BASIC heap, the BASIC stack and the system area.

> The BASIC heap is where all global variables, strings and arrays are held. DIM also claims memory from here. It is located immediately above the program code.
> The BASIC stack is where temporary information such as parameters, local variables and procedure call information is held. It is located high in memory and grows down towards the top of the BASIC heap
> The system area is a 4K area for a stack and some internal data.
> By default 4K is reserved at the top of memory for the system area, with the remainder equally divided between heap and stack. This will often be sufficient, but in some cases a program may have more intensive use of variables, or of temporary data.

The size of heap or stack can be set explicitly, eg:-

```
REM {HEAP=10000}
REM {STACK=6000}
```

If you only specify one of these, the other will be allotted all the remaining space.

HIMEM returns the address of the top of the stack, and EXT returns the current stack pointer, thus HIMEM-EXT will report the amount of stack space being used.

Heap and stack size may be defined in percentages, eg:-

REM {HEAP% 90}
REM {STACK% 10}

You may combine absolute and percentage setting

Memory usage with `REM {NEWHEAP}`

NEWHEAP may only be used with desktop applications. NEWHEAP sets up an expanding/contracting heap, eg:-

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

This sets up 8K for both areas plus a dynamic heap. When a dynamic heap is in use the fixed heap will hold global variables. Strings, arrays and DIMmed memory will be placed in the dynamic heap.

CLEAR can be used to release dynamic memory, eg:-

- `DIM fred% 512`
- `CLEAR fred%`

The wimpslot will automatically grow or shrink as the dynamic heap alters size. `REM {NEWHEAP}` is the recommended way of directing ABC to manage its memory

Variable type management with `REM {TYPE}`

In its simplest form {TYPE} can be used to force variables to a specific type. In the example below all the variables are treated as double precision:-

- `REM {TYPE=DOUBLE}`

- `INPUT x`
- `INPUT power`
- `PRINT x^power`

A {TYPE} directive must occur at the start of the program before any executable code. {TYPE} can also be used to assign a suffix to a variable type, eg:-

- `REM {TYPE ` = EXTENDED}`

If these are combined as in the example below it becomes possible to test floating point code with the interpreter (though at less precision obviously):-

- `REM {TYPE=EXTENDED}`
- `REM {TYPE ` = EXTENDED}`

By using this you avoid the interpreter's restriction on & as a suffix to a variable name

By default the floating point indirection operator transfers four bytes. If {TYPE} has been used to change the size of simple variables to 8 or 12 bytes that number will be transferred instead.

It is also possible to force the compiler to transfer 4,8 or 12 bytes by using the S, E or D directives as in the example below:-

- `REM {TYPE=DOUBLE}`
- `DIM block% 1000`
- `|{S}block%=37.5: REM forces 4 byte transfer`

It is possible to force variables not ending with % to be treated as integers, eg:-

- `REM {TYPE=INTEGER}`
- `fred=10.3`
- `PRINT fred: REM prints 10 in compiler, 10.3 in interpreter`

The meaning of % and $ may not be changed with {TYPE}. If you wish to completely avoid floating point operations, use `REM {NOFLOAT}`.

With {NOFLOAT} in effect these keywords become prohibited:-

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

With {NOFLOAT} in effect the fifth parameter to ELLIPSE cannot be used (it specifies angle of rotation, which requires trig calculations and floating point numbers).

Large programs (Over 256K)

The compiler has two forms of addressing scheme. The default has a limit on object code size of 256K and generates more compact code. If the object code will exceed this size use the directive `REM {LONGADRS}`. The opposite is `REM {SHORTADRS}`.

CASE statements

The compiler can either generate a series of comparisons and branches or a jump table when encoding a CASE statement. The jump table method is usually more efficient, but can occupy more memory since a jump has to be created for every value in the range expressed in the CASE statement.

The `REM {MAXCASES=n}` directive sets the decision point between a jump table or comparisons/branches, using the latter if the value is exceeded. The default is 256.

Assembly language

`REM {REGISTERS}` creates manifest constants for R0-R15 and all case permutations of PC. This will avoid a Warning being generated every time the compiler has to assign it for you. Using {REGISTERS} is also a useful safeguard against getting caught out by accidental use of possible register names elsewhere in the program with incorrect values. Specifically an instruction like MOV R0,#10 will only assemble as expected if the variable R0 has the value 0. The same applies to all register names.

Whenever [is encountered an automatic OPT 3 is executed. To change these options an OPT opt% or similar must be used within each fragment of assembler. You can save on code size by using `REM {NOOPT}` to force OPT statements (whether automatic or literal) to generate no code. This allows the start of the program to have the sequence below, which sets the prevailing OPT level, with all further OPTs generating no code. The initial `REM {OPT}` is not strictly necessary, but is a good precaution:-

- `REM {OPT}`
- `[OPT opt%]`
- `REM {NOOPT}`
- `[OPT opt%` - this generates no code now!

Resolving SWI names

By default the compiler attempts to resolve a SWI name to a number at compile time, but will embed the name for run-time resolution if it cannot resolve it at compile time.

`REM {SYSKNOWNONLY}` asserts to the compiler that all SWI names are expected to resolve to numbers. A Warning will be raised for any that do not. This behaviour can be disabled with `REM {NOSYSKNOWNONLY}`.

Controlling the generation of Warnings

`REM {NOWARNINGS}` will suppress the generation of Warnings. They will still be counted. This is useful if you know that the compilation will generate a number of warnings that are known to be false alarms. The default behaviour can be turned back on with `REM {WARNINGS}`

Optimisations

Stack Checking

`REM {NOSTACKCHECK}` tells the compiler not to generate code to test whether the stack has grown downwards too far and collided with the top of the heap. `REM`

`{STACKCHECK}` enables the check. Selecting the Quick compilation option turns on {NOSTACKCHECK}

Checking for the Escape key

`REM {NOESCAPECHECK}` tells the compiler not to generate checks for the escape key being pressed during execution. This is recommended for wimp programs. The check may be enabled (the default) with `REM {ESCAPECHECK}`. Selecting the Quick compilation turns on {NOESCAPECHECK}

Checking array subscripts

`REM {NOARRAYCHECK}` tells the compiler to suppress generation of code to check whether arrays are being accessed within their valid range of subscripts. The default can be restored with `REM {ARRAYCHECK}`. Selecting the Quick compilation option turns on {NOARRAYCHECK}.

Checking for traps

`REM {NOTRAPS}` tells the compiler not to generate code that detects falling into a function/procedure (eg because an END statement is missing). The default behaviour can be restored with `REM {TRAPS}`. Selecting the Quick compilation option turns on {NOTRAPS}.

Initialising local variables

`REM {NOZEROLOCALS}` tells the compiler not to initialise LOCAL variables to zero upon their creation. This implies that the program initializes all LOCAL variables before using them, and can be a source of strange bugs that take a while to track down. `REM {ZEROLOCALS}` restores the default behaviour. Selecting the Quick compilation turns on {ZEROLOCALS} and this particular option is usually the most common reason for problems being seen with programs compiled with Quick turned on.

Unused registers with SYS

`REM {NOZEROSYSREGS}` tells the compiler not to zero any of R0-R7 for a SYS call that have not been explicitly assigned values. The effect of this is that SYS "Something_Op",a%,,c% would pass 0 in R1 under the interpreter, but an unpredictable value when compiled. The default is `REM {ZEROSYSREGS}`.

CALL and USR

`REM {NOCALLREGS}` tells the compiler not to generate code that copies A%-H% to R0-R7. It also tells the compiler not to initialize any unassigned registers to zero in the extended forms of CALL and USR. The default is `REM {CALLREGS}`.

Aligned addresses

By default the compiler assumes that addresses accessed by ! might not be word-aligned. However if they always are it can generate more efficient code. `REM {ALIGNEDPLING}` tells the compiler to consider all accesses word-aligned. The default is `REM {NOALIGNEDPLING}`

Use of GOTO and GOSUB

By default the compiler assumes that the program is going to use GOTO or GOSUB and has to keep track of every line as a potential branch target, which is an overhead on the compiler and a hit on its register usage optimization. This can be overridden with `REM {NOGOTOSUSED}`. The default is `REM {GOTOSUSED}`. When {NOGOTOSUSED} is in effect a Warning will be raised for any GOTO/GOSUB encountered.

Resolution of SWI names

`REM {SYSCONSTONLY}` tells the compiler that all SYS/SWI names are string constants or held in variables (rather than general string expressions). This allows more efficient code to be generated. The default is `REM {NOSYSCONSTONLY}`.

Use of throwback

By default the compiler will attempt to use throwback. This may be disabled by using `REM {NOTHROWBACK}` and re-enabled with `REM {THROWBACK}`

Squeezing of the compiled code

If `REM {SQUEEZE}` is used the compiler will attempt to run the compiled code through the squeeze program to reduce its size (squeeze is one of the supporting utilities in the C/C++ tools).

Squeeze must be on the Run$Path for it to be found. If you are uncertain whether it is there bring up a command window (or press F12) and type `squeeze –h.` If squeeze is found you will see some help information which includes the version number.

For code which will run on 26 and 32 bit environments you must use squeeze 5.08 or later. Earlier versions had a bug that corrupted the code, and prior versions only generated 26 bit-capable code.

The use of squeeze is turned off by default, and may be explicitly turned off with `REM {NOSQUEEZE}.`

8 Manifest constants

A manifest constant is a variable declaration where the value of the variable will not change during runtime. Using manifest constants helps to prevent bugs where intended constants get modified by the program and allows ABC to generate more efficient code. However since they are not supported in the interpreter care is needed to generate them in a way that allows a program to still work whether interpreted or compiled. The example below shows this in action:-

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

Of course, there is the slight disadvantage that care must be taken to keep the constant’s value the same in both declarations.

A%-Z% and @% may not be defined as manifest constants, nor may a manifest constant be negative.

9 Conditional compilation

Manifest constants allow a more sophisticated scheme for conditional compilation to be implemented than is possible with {COMPILE}/{NOCOMPILE}. Consider this example:-

```
    DEF debug%=1
    REM {IF debug%}
          PRINT “*** Debug version ***”
    REM {ELSE}
    …
    REM {ENDIF}
```

If the result of the test is True (non-zero) the PRINT will be compiled. If it is False (zero) any code after the ELSE will be compiled instead.

Only expressions may be used with `REM {IF…}`. So `REM {IF English}` is legal but `REM {IF lang$=”english”}` is not.

Tests may be nested, for instance:-

```
    DEF debug%=1
    DEF demo%=0
    REM {IF demo%}
          PRINT “This feature is only available in the 
full version”
    REM {ELSE}
          PROCa
          PROCb
          REM {IF debug%}
                PRINT “Returned values: “;a,b
          REM {ELSE}
          REM {ENDIF}
    REM {ENDIF}
```

10 Writing Modules

Although ABC has support for compiling relocatable module code, it has a number of restrictions and caveats. Details are available on request from Pineapple Software, but we recommend that modules are written entirely using BASIC’s in-line assembler, C or directly in assembler.

11 Library Modules

ABC has the capability to use libraries of pre-compiled procedures and functions, however there are a number of restrictions on libraries. More details are available on request.

12 The history of ABC

ABC was an early arrival on the Archimedes scene, issued initially by Solent Computer Products and written by Paul Fellows. By 1991 the title had passed to Oak Solutions and in that year version 3 of the compiler was released. However, following years marked the decline of Oak's interest in the desktop computing market and by the advent of StrongARM the product was in danger of becoming obsolete.

One of Pineapple Software's key products, its anti-virus program !Killer, is built with ABC so it was essential for them to have continuity of ABC. With this in mind they negotiated successfully for the rights and produced a new StrongARM-compatible version of ABCLibrary. Previously compiled programs continued to run successfully on StrongARM – the changes were done in such a way that the compiled application was unaware of StrongARM necessities such as flushing the code cache after generating dynamic code with inline assembler in a compiled program.

The version Pineapple took over was an interim version, more recent than Oak's last major version, so it needed some work to resolve a couple of development bugs as well as establishing support on the compiler side for StrongARM with new directives to tell ABCLibrary that the compiled code knew about cache flushing and would handle it itself rather than ABCLibrary applying its brute force approach.

Regrettably the market for a BASIC compiler has never been huge, and with the decline of the desktop market there seemed little wisdom in investing time, effort and money in overhauling ABC to enhance it much beyond its capabilities in its Oak days.

With the advent of 32 bit both compiler and library once again needed changes – much more major this time than for StrongARM. Both compiler and ABCLibrary used 32-bit-illegal code sequences, so a new ABCLibrary was essential, as was recompilation of code intended to run 32 bit. More seriously still the linkage scheme used between the application and ABCLibrary breaks down in a 32 bit environment, and the new linkage scheme needs more space in the application than old applications allowed.

Pineapple have now made a deal with Castle where Pineapple will support the compiler whilst Castle handle the distribution (typically as part of their Tools CD).

One of the casualties of the period of hiatus has been the electronic text of the manual. Copies of the 1991 manual exist in paper form, and are for the most part still entirely valid – the only thing actually missing is the StrongARM compilation directives. However we do not have this in electronic form, hence the creation of this Release Note as a highly abridged form of the 1991 manual plus additions.

During the ‘dark years’ of the desktop market ABC has had the bare minimum of maintenance; indeed it has been stable since the StrongARM work and has only needed revisiting for 32 bit conversion. As a consequence it has a few shortcomings. The most notable is that its inline assembler emulation has not been extended to incorporate ARM architectures 4 or 5. At present our intention is to perform maintenance only, but we will reconsider that position if there is renewed interest in ABC.

13 Recent versions

Compiler 3.10 was the last Oak Solutions version to see major release, though 3.13 and perhaps 3.14 are in use by some people. Pineapple took on the compiler from version 3.14's sources.

ABCLibrary 4.05 was released by Pineapple to support StrongARM processors running previously compiled code. Compiler 3.15 gained the compiler-side introduction of StrongARM support with the addition of the `REM {MANUALCODESYNC}` directive.

The next major flurry of ABC activity was the work to make ABC 32-bit compatible. The compiler's version number has been changed to be in step with ABCLibrary (or at least closer!).

Compiler 4.10 and ABCLibrary 4.12 were the first 32 bit versions considered stable. Compiler 4.11 was released in response to feedback and contains the following changes from 4.10:-

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 (eg Edit, Zap, StrongEd)
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.

Compiler 4.12 refines some of 4.11's new features and adds a few more changes:-

New options {THROWBACK} (default) and {NOTHROWBACK}.
New options {SQUEEZE} and {NOSQUEEZE} (default).
The warning/error window is now centred.
The compilation complete window is now centred.
Fixed the bottom y position of the icon bar menu.
Fixed the line spacing in the 2D templates set in the warning/error window
Revised set of templates – thanks to Richard Hallas for combining the best of his first version based on 4.10's with Paul's 4.11 version.

ABCLibrary 4.13 fixes a bug around reading the returned flags on a 32bit processor from CALL, USR and SYS calls where the SWI number is resolved at runtime rather than compile time.

Compiler 4.13 introduced {MODULE 32BIT} and {LIBRARY 32BIT}, though using ABC to generate module code remains deprecated.

ABCLibrary 4.14 fixes a bug with EQUW on 32 bit systems (the EQUW would fail to generate any data)

Compiler 4.14 fixes a bug where {ZEROLOCALS} wouldn’t take effect whilst the Quick compilation switch was turned on.