ACORNSOFT

Archimedes

UTILITIES

GUIDE

ACORNSOFT

Archimedes

UTILITIES

GUIDE

Within this publication, the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

First published 1988
Issue 1 1988
Published by Acorn Computers Limited
Part number 0481,845

CONTENTS

This manual describes the utilities in the Archimedes Software Developer's Toolbox package (the Arthur Symbolic Debugger is described in a separate manual). The Toolbox includes a dozen utilities which are designed to enhance the productivity of programmers developing software under the Arthur operating system.

The scope of these utilities is quite wide, and some programs will be used more frequently than others. For example, you will rarely need the memory tester routine, but if you are developing software in a compiled language like Pascal or C, you will have constant recourse to the linker.

Here is an alphabetical list of the utilities provided with the package, with a brief description of what they do:

- Amu — Control compilation of multiple sources in large programs
- Common — List the most common words in a list of files
- Dbug — Machine code debugger for use from multiple languages
- Diff — Compare two files and list their differences
- Grope — Search files for a given pattern (regular expression)
- Libfile — Manipulate Acorn Library Format (ALF) files
- Link — Link AOF files and libraries to form a runnable image
- Memtest — Test the application memory of the machine
- Objlib — List the symbols in a library file
- Shell — Allow BASIC to be used as a command language
- Squeeze — Compress Acorn Image Format (AIF) file to conserve disc space
- Wc — Count characters, word and lines in a file

TEXT FILE MANIPULATION describes those utilities which are concerned with manipulating text files. These are: Common, Diff, Grope and Wc.

PROGRAM MANIPULATION UTILITIES covers the utilities which manipulate program or library files. These are: Libfile, Link, Amu, Objlib and Squeeze.

THE DBUG UTILITY looks at the debugger program, Dbug.

THE BASIC SHELL LIBRARY is about the BBC BASIC Shell library.

THE MEMTEST PROGRAM describes the Memtest program.

All of the programs (except BASIC Shell) are 'applications': that is, they load at address `&8000` and take over the foreground control of the Archimedes workstation. When they terminate, they pass control back to the last program to set up an 'exit handler'. This is generally the operating system (the Arthur * prompt), but could also be another application such as Twin or BASIC (assuming the utility was called using the Shell library). You should therefore ensure that before issuing one of these commands from BASIC, the current program is saved.

APPENDIX A: FILE FORMATS gives a brief description of three important types of file that a programmer will come across: object files, image files and libraries.

APPENDIX B: ARM PROCEDURE CALLING STANDARD describes the standard that allows programs written in various high-level languages and assembler to communicate through a well-defined procedural interface.

EXPLANATION OF SYNTAX

When the format of a command is described in the subsequent chapters, the notation makes use of some special characters to denote optional items and classes of item. Square brackets (`[` and `]`) enclose items which are optional. Classes of object are shown in *`italics`*, which indicates that these words are not to be typed literally. The vertical bar, `|`, denotes a choice of objects (and is read 'or'), and an ellipsis, `...`, denotes that the previous item may be repeated an arbitrary number of times.

Here is an example which uses all of the syntax characters:

`Common [-first` *`n`*`] [-n- | -n+]` *`filename`*`...`

Thus the Common command may be followed by the keyword `-first`, which itself is followed by a number (the *`n`*). This is followed by a list of filenames, each of which may be prefixed by the flags `-n-` or `-n+`.

Note that it is not explicit in the syntax above just how much of the preceding text the `...` refers to. In this case, it encompasses the text `[-n- | -n+]` *filename*. In such ambiguous cases, the explanation of the command will make things clear.

All of the commands documented in this manual respond to the `-help` option, so this is not mentioned explicitly in the text.

You will have noticed the use of a typewriter-type font above. This is used in syntax descriptions,

- to show something that is typed literally at the keyboard, and
- to show output that is displayed on the screen.

The four utilities described in this chapter operate on one or more text files. They provide information about the contents of the file(s), eg the most common words, whether particular patterns occur in the file etc.

COMMON - LIST COMMON WORDS IN FILES

Description of use

This program scans a file or several files for strings of characters which look like words. It keeps a count of how many times each word occurs, and prints a table in decreasing order of frequency. All the words that were found can be printed, or just the most popular few.

The definition of a word is 'any sequence of one or more upper or lower case alphabetical characters'. Thus `Hello`, `APPLE` and `TUTTIFRUTTI` are all words. The case of the letters is significant, so `Hello` is treated as a different word from `HELLO`.

You can broaden the definition of a word to encompass numeric characters (the digits 0 to 9) as well. This is useful, for example, when you are analysing assembly language source files, where you may want to treat labels such as `DIV010` as complete words. It also allows you to count the number of occurrences of register names in instructions, eg `R0`, `R13` etc.

Syntax

The Common command has the syntax:

`Common [-first` *n*`] [-n- | -n+]` *filename* `...`

If the `-first` option is specified, only the first *n* most popular words are listed, otherwise all words are given. The switch `-n+` enables digits to be counted as word characters for subsequent filenames, and `-n-` disables digits again. The default is `-n-`.

Example

```
Common -first 10 c.find -n+ asm.find
```

This will print the 10 most common words found in the two files `c.find` and `asm.find`, where numerics are allowed in words for the second file but not the first.

Here is an example of the results printed by Common when it was run on a file whose contents were the output of the command `*HELP`:

```
Total number of different words: 664
167 of 'on'
149 of 'the'
147 of 'Help'
144 of 'keyword'
122 of 'Syntax'
57 of 'a'
48 of 'to'
43 of 'file'
41 of 'of'
33 of 'filename'
...
```

DIFF - COMPARE TWO FILES

Description of use

This command compares the contents of two files. It then prints a series of comments which indicate the operations that must be performed on one of the files to make them identical (ie to make the first named file identical to the second one).

Typical comments that are generated by the program are:

```
remove filename line number: text of the line

after filename line number: text of line
add filename line number: text of line

change filename1, line number
  line number: text of line
to filename2, line number
  line number: text of line

Identical
```

It is most useful to send the output from Diff to a file. You can then use Twin to edit the first file specified in the Diff command line in one window, and inspect the changes to be made in the other; see the example below. (Alternatively you could run Diff in one of Twin's windows, with the file to be altered in the other window.)

Syntax

The syntax of the Diff command is very straightforward:

```
Diff file1 file2
```

As mentioned above, the instructions output by Diff give you the changes to make to the first file in order to make it identical to the second one.

Example

The command below compares two files, and sends Diff's output to a third file which can subsequently be inspected in Twin.

```
*Diff asm.srce asm.backup.srce { > diffs }
```

GROPE - FIND PATTERNS IN FILES

Description of use

The Grope command is a pattern searcher. It looks for a string of characters in a list of files and displays the lines which contain the string. Because the search string can contain various wildcards and other special characters, Grope is a very versatile utility. The formal name for the type of search string that Grope can find is 'pattern' or 'regular expression'.

Grope can take several options in addition to the usual `-help` flag. These are described in the syntax section below. They may all be abbreviated to the first letter of the keyword (as shown in upper case).

Syntax

The syntax of Grope is:

`Grope` *`patterns files`* `[`*`flags`*`]`

The three parts may appear in any order, The *`patterns`* part has the syntax:

`[-Pattern]` *`pattern`* `[`*`pattern`*`]...`

The keyword `-pattern` (or just `-p`) may be omitted if the patterns don't immediately follow the *`files`* part. Otherwise it is required to separate the files list from the pattern list. The exact form that a *`pattern`* can take is described below.

The *`files`* part has two forms:

`-Files` *`file`* `[`*`file`*`]...`

and

`-VIA` *`viafile`*

In the first case, the keyword is followed by a list of filenames to be searched for the pattern. They are searched in the order specified. The second form uses the filename given to find a list of files to be searched. The filenames should be listed one per line in the *viafile*. Again, the search order is the same as the ordering of the filenames.

You can also combine these keywords. If they are both given, the `-VIA` file is taken to be a list of directory names. The `-Files` list are files which are searched in each of those directories. For example, suppose the file `Cpaths` contains:

```
$.arm.clib.c
$.arm.clib.h
```

then the command:

```
GROPE Init ( -files main expr print -via cpaths
```

will look for the pattern `Init (` in the files:

```
$.arm.clib.c.main
$.arm.clib.c.expr
$.arm.clib.c.print
$.arm.clib.h.main
$.arm.clib.h.expr
$.arm.clib.h.print
```

in that order.

There are three optional *flags*. These are:

`-Nocase`	Make the search case insensitive
`-Verbose`	Display each file's name before it is searched
`-Describe`	Describe the *pattern* syntax

Searches are usually performed in a case sensitive manner, so that the pattern `HELP` would not match the string `Help` in a file. Specifying `-nocase` reverses this, so that the case of letters in the pattern and file is not significant.

If you use the `-describe` flag, Grope just gives a description of the pattern syntax and exits immediately, as with `-help`.

Pattern syntax

This section describes the syntax of Grope patterns.

> Note: The pattern syntax is very similar to that used in the EDIT program provided with the Acorn Master computer and 6502 Development Package. Users of either of those products should have little difficulty understanding Grope patterns. (The patterns are also similar to those used by Twin, but use normal punctuation symbols as special characters, instead of Twin's function keys.)

Any character which does not have a special meaning in a Grope pattern is referred to as a 'normal' character. This includes alphabetic and numeric characters, and many of the punctuation characters. The first six characters described below (*any* to *lit* inclusive) are know collectively as 'special' characters.

`.` *any*: matches any single character

`@` *id*: matches any 'identifier' character (same as `[A-Za-z0-9_]`)

`#` *dig*: matches any single digit (same as `[0-9]`)

`|` *ctrl*: control character. `|c` matches Ctrl c, where `c` is in the range `@` to `_`, yielding ASCII 0 to ASCII 31. `|?` matches ASCII DEL (127). `|!` matches 128 plus the following character (or *ctrl* character). After a `|!`, `|` is the only character which is not treated literally.

`$` *newl*: matches the end of line character (ASCII 10)

`\` *lit*: matches the following character literally, even if this would normally have a special meaning in patterns. For example, `\#` means the ASCII character `#`, not *dig*. It must also be used before spaces in patterns, as these are used as pattern-list separators.

`[ ]` *class*: matches a set or range of characters. For example `[abc]` matches any one of `a`, `b` or `c`. The characters in a *class* may be

normal characters or the special characters. That is, the characters `@ # | $ \` retain their special meanings, and all others are treated literally.

- *range*: matches a range of characters. This can only be used within a *class*. The pattern `[a-b]` matches any single character in the range a to b, where those are normal characters or *ctrl* chracters. For example, `[|@-|_]` would match any control character. Several ranges and single characters may appear in a single *class*, as shown in *id* above.

~ *negation*: matches anything but the following character, which may be a normal character, a special character, or a *class*. For example, `~A` matches anything but ASCII 65, `~#` matches any non-digit, and `~[|@-|_]` matches any non-control character.

* *many*: matches zero or more of the following character, which may be a normal character, a special character, a *class*, or a *negation*. The shortest string possible will be matched, so a *many* should be followed by another character to stop the null string from being matched. For example `*#` will always match the null string, whereas `*#~#` will match the longest sequence of digits possible, followed by a non-digit. `LDM*.{*~;PC` will match any ARM LDM instruction involving the PC.

^ *most*: matches one or more occurrences of the following character, which may be a normal character, a special character, a *class*, or a *negation*. Unlike *many*, it matches the longest possible string. For example, `^#` matches a string of decimal digits, `[A-Za-z]^@` matches an identifier of at least two characters.

Note: The backslash character \ has a special meaning for the arguments decoder used by Grope. This means that in order to get a backslash through to the program itself, it should be 'escaped' thus: `\\`.

Examples

The examples below use a file `hlp` which contains the output of the command `*HELP`. Below each command line is some of the output produced by the command.

```
Grope basic -f hlp
```
No output

```
*Grope -nocase basic -f hlp
"hlp", line 257:BBC BASIC V           1.00 (05 Jun 1987)
...
"hlp", line 731:Syntax: *BASIC [-help|-chain|-load|-quit] <filename>
*Grope Syntax:^~$ -f hlp
```
All the syntax lines in the help file are printed

```
*Grope \\*Voices -f hlp
"hlp", line 1061:*Voices lists the installed voices and channel
allocation
```

WC - COUNT WORDS IN FILES

Description of use

The Wc command reports the number of lines, words and characters found in a file (or list of files) which is given as the parameter. In fact, two types of words are counted: those which consist of sequences of one or more upper and lower case letters, and 'identifiers' which can contain alphabetic characters and digits. A line is defined as a sequence of characters terminated by a newline character (ASCII 10). Note that Wc is designed for use with ASCII text files and will not give useful results with tokenised BASIC files.

If a list of files is given on the command line, the counts for the individual files are given, followed by the total for all files.

Syntax

The syntax of the Wc command is as follows:

`Wc [filename]...`

If no filenames are given, the help information for the command is displayed (as if the `-help` option had been specified).

Example

The following commands might produce the output below:

```
*Wc memsrc
Word count of 'memsrc':
      Lines: 234, Words: 595, Alphanumerics: 719, Characters: 3519

*Wc shellinf helptxt
Word count of 'SHELLINF':
      Lines: 508, Words: 2438, Alphanumerics: 2866, Characters: 16132
Word count of 'HELPTXT':
      Lines: 729, Words: 3068, Alphanumerics: 3253, Characters: 22690

Total: Lines: 1237, Words: 5506, Alphanumerics: 6119, Characters: 38822
```

This chapter describes the utilities which are concerned with the manipulation of object program files (AOF files), image files (AIF files) and libraries. Descriptions of all three types of file can be found in *Appendix A: File formats*.

AMU - A 'MAKE' UTILITY

Description of use

Amu is designed to facilitate the management of large programs which use multiple source and object files. Its input is a file which contains 'targets', called the makefile. Associated with a target are one or more dependencies. Amu's job is to produce a list of commands which, when executed, ensure that the target is consistent with respect to the things (usually files) that it depends on.

Typically a target is the name of a compiled program. It depends on the source and object files which are compiled and linked to produce the final program. To be 'consistent', a target must be newer than the files it depends on, ie must have been updated more recently. (If a source file was altered more recently than the object file that it produces, the latter is probably out of date and therefore the whole system described by the 'makefile' is in an inconsistent state.

As mentioned above, the output of Amu is a set of commands. These are derived from the makefile, where they are listed along with the dependencies, and are usually sent to a command file for execution when Amu terminates. (Amu does this automatically, but the file could be re-executed using `*EXEC`.) Alternatively, the commands may be displayed on the screen instead. When executed, the commands put the system being made into a consistent state with the minimum amount of work (eg the fewest compilations).

The exact format of the contents of a makefile is described in the section below called *The makefile*.

Syntax

The Amu command has the following syntax:

```
Amu [-f makefile] [-o cmdfile] [-n] [-t] [target...]
```

The `-f` option precedes the name of the *`makefile`*, ie the file which contains the targets and dependencies to use. If it is omitted, the file called `makefile` in the current directory is used.

Following the `-o` option is the name of the file to which the list of commands must be sent. If omitted, the filename `!make` is used. Alternatively, you can specify the `-n` option. This causes the commands to be displayed on the screen instead of being sent to a file.

The option `-t` causes Amu to generate commands which make the target(s) up to date by setting the time/date-stamps on source files. The Arthur command `*STAMP` is used. All the source files must exist for this option to succeed (`*STAMP` will not create a file).

Finally the *`targets`* specify which of the targets in the makefile should be processed. If none is given, the first target in the file is used, so it is usual for the first target to be one on which the whole of the system being made depends. Other typical targets are given in the next section.

You can also define macros in the *`targets`* section, using the form *`name=value`*. These act as if they were defined in the makefile, as described in the section *Text substitution* below.

The Makefile

As already mentioned, the makefile is a list of targets, dependencies and commands. Here is a listing of a typical makefile:

```
all:        grope

grope:      o.grope o.pattrn o.output
            link -o grope o.grope o.pattrn o.output $.arm.clib.o.ansilib

o.grope:    c.grope h.grope
            cc -c c.grope
```

```
o.pattrn:  c.pattrn h.grope
           cc -c c.pattrn

o.output:  c.ouput h.grope
           cc -c c.output

h.grope:   h.ctypes
           stamp h.grope

install:   ;copy grope $.library.grope ~cfq

clean:     ;wipe o.* ~cf;delete grope
```

The first line contains the target `all`. In the absence of any targets on the command line, Amu will try to satisfy this one. Immediately after the target is a colon and one or more spaces (or tabs) which separate it from the files it depends on. In the case of `all`, the only dependency is `grope`. Thus in order to make sure that the target `all` is satisfied, Amu has to generate the commands to ensure that `grope` is consistent.

`grope` depends on the three object files. You can read this line as '`grope` must be younger than all of the object files on the line'. If it isn't, the command:

```
link -o grope o.grope o.pattrn o.output $.arm.clib.o.ansilib
```

must be executed. Notice, though, that each of the object files has its own target line and dependencies, so before the `link` command can be output, any or all of the source files may have to be re-compiled. For example, the lines:

```
o.grope:   c.grope h.grope
           cc -c c.grope
```

say that `o.grope` must be newer than both `c.grope` and `h.grope` if it is to be considered up to date, and if it isn't, the `cc` command below must be executed.

The dependency for the file `h.grope` says that it is out of date if it is newer than `h.ctypes`. In order to make it up to date, it is simply stamped with the current time. This will in turn make all of the object files out of date, as they all depend on `h.grope`, resulting in the re-compilation and re-linking of the files.

The final two lines of the file are targets which can be used to produce commands to copy the newly compiled and linked `grope` into the library, and delete the object files from the home directory, respectively. They will be explained below.

Dependency relations

The example above covered most of the common items found in makefiles. Now follows a more formal description.

A dependency relation consists of a single logical line containing a space-separated list of target names, followed by `:` or `::`, followed by a space-separated list of source files on which each target depends. It is important that at least one space (or tab) follows the `:` or `::`. The difference between these two variations is explained below.

You can continue the 'logical' line that the relation occupies on to the next physical line by preceding the newline with a backslash character (`\`). Another way to deal with long dependencies is to have more than one target of the same name, eg:

```
o.alloc:    c.alloc h.alloc
o.alloc:    h.printf h.exit h.mcsuppt
            cc alloc
```

The right hand sides are effectively merged, so that in this example, `o.alloc` is dependent on all five filenames. Note that only one of the relations may have an associated list of command lines below it (the second one, in the example above).

A command line starts with at least one space or tab character. The command is terminated by a newline, and may be followed by more command lines of the same form. Alternatively, you can use a semicolon to separate the dependency from the first command, and commands from each other, as this line taken from the first example above shows:

```
clean:      ;wipe o.* ~cf;delete grope
```

This says that the target `clean` has no dependencies (so the commands are always sent to the output file) and the commands associated with it are a `wipe` command and a `delete` command. An Amu command to invoke this target might be:

```
Amu all copy clean
```

This would re-compile the system (if necessary), copy the new object file to the library, and then delete all the unrequired object files. (Note that targets are made in the order in which they are specified on the command line.)

If you use `::` to separate the target from the dependencies, then the right hand sides are not merged but remain independent. For example:

```
o.t1::      c.t1 h.t1
            cc -g -c c.t1

o.t1::      c.t1 h.t2
            cc -c c.t1

all:        o.t1
            link -o t1 o.t1 $.arm.clib.o.ansilib
```

In this example, the first `cc` command is executed if (and only if) either of the files `c.t1` or `h.t1` is newer than `o.t1`. The second `cc` command executes if either `c.t1` or `h.t2` has changed. The `all` part depends on either of the re-compilations taking place.

Text substitution

You can 'parameterise' makefiles to a degree by using the textual substitution mechanism. A macro is defined by a line of the form:

name=text

Examples are:

```
CFLAGS = -fa -fh
CC = cc160a
```

To use a defined name in the makefile body, it is placed in a string of the form `${name}` or `$(name)`. For example, the names defined above might be used in this relation:

```
o.hash      c.hash h.hash
            ${CC} ${FLAGS} hash
```

By placing the macro assignments at the start of the makefile, you can quickly make all of the commands use, for example, a new version of the compiler, or different options.

Comments

Any text following a hash character # is regarded as a comment and is ignored by Amu, up to the end of the line. Comments can be useful for explaining non-obvious relations, or planting Twin in-core filenames at the start of the file, eg:

```
# >mkgrope - make file for the grope utility
```

LIBFILE - LIBRARY FILE TOOL PROGRAM

Description of use

Libfile is a general-purpose tool for maninpulating compiled libraries. The two main applications of Libfile are complementary. It can be used to construct a library from compiled object files, or to decompose a library into its constituent files. Other uses are listing the contents of a library, and creating a library. As usual, the different options are accessed using command keywords.

Syntax

To avoid a complex command syntax line, each variant of the Libfile command is listed separately below. Capitalisation is used to show the alternative short form of each keyword.

```
Libfile library -Create [nfiles len] [files]
Libfile library -Insert files
Libfile library -Delete files
Libfile library -Extract files
Libfile library -List
```

In all cases, a *library* must be specified. This has the form:

```
[-LIbrary] filename
```

The keyword is only required if the *filename* could otherwise be taken to be part of a *files* list. If you make the library filename the first argument, you will never need to use the keyword.

The *files* part, which is required by the `-insert`, `-delete` and `-extract` variants, and optionally in the `-Create` variant, consists of a keyword followed by one or more filenames. The keyword may be `-Members` or `-Files`. The former is more natural in the delete and extract cases, where existing members of a library are being manipulated, and the latter makes sense in the create and insert cases, where names of object files are being quoted.

Create keyword

This form creates a new library file. The file may be empty, but with enough 'slots' to allow the susbequent insertion of object files, or it may be formed from existing files. The first variant looks like this:

```
Libfile library -create nfiles len
```

Two numbers follow the `-create` keyword. The first gives the number of slots to create in the file. Each of these slots can subsequently be filled by a `-insert` command. The second number gives the average length of the filename of each file. In effect, *`nfiles*(K+len)`* bytes (where *K* is a constant) are reserved in the 'directory chunk' of a new library file to hold the names and other information of its component files. See the section on the `-list` option below for more details.

You can insert some files into the new library at the same time that it is created by specifying them after a `-files` option. This variant has the form:

```
Libfile library -create [nfiles len] -files filenames
```

The files in the list are used to form the new library. They should be AOF files, eg generated by a compiler or ObjAsm, or extracted from a library. If you omit the numbers after the `-create` keyword, enough slots are created for about half as many files again. For example, if ten files are used to create the library, 15 slots will be created.

You may use wildcards in the *`filenames`* list. If you do, all files which match the wildcard specification will be inserted into the library.

Insert keyword

To insert one or more new files into an existing library file, you use a command of the form:

```
Libfile library -insert files
```

Before the list of filenames you can put the keyword `-files` or `-members`, or omit it altogether. Each of the AOF files listed is inserted into the library file. If a file has the same name (ignoring the case of letters) as an existing one, it replaces the current version. Wildcards may be used in the filenames and are expanded to a list of all the files matching the specification.

If there is not enough room in the library file, you will get an error of the form:

`Not enough space in the directory chunk of '`*`library`*`'`

or

`Not enough chunks in header of file '`*`library`*`'`

Delete keyword

This variation of the command removes one or more files from the library. It has the form:

`Libfile` *`library`* `-delete` *`files`*

The *`files`* list may be preceded by the keyword `-files` or `-members`, or nothing at all. The names specified must match exactly those stored in the library (except for the case of the letters). A warning message is given if any of the files cannot be found in the library. It is a fatal error to specify the same file more than once, and wildcard characters cannot be used in the filenames.

Extract keyword

This performs a complementary function to `-insert`. It copies the contents of one or more files in the library into separate (AOF) files of the same names. The format of the command is:

`Libfile` *`library`* `-extract` *`files`*

As with the other variants, you may precede the list of filenames by either of the keywords `-files` or `-members`, or omit it altogether. Note that the files are not removed from the library; they are just copied into the filenames given. If a file cannot be located in the library, a warning of this form is given:

```
libfile: (Warning) Failed to find member 'filename'
```

List keyword

You can list all the component files of a library using this version of the Libfile command. For each file the name, length and date of creation are given. After the list, the number of free entries left in the header (ie the total allocated at creation minus the number currently used) and the number of free bytes in the directory chunk are printed. The size of the latter is determined by the number of entries in the library and the length of their names.

When a library is created, the directory chunk size is calculated as *`nfiles*(5+(len+4) div 4)`* words. For each entry, five words are allocated for fixed information such as creation date and length, then some extra words are added depending on the average length of the filenames to be stored. It is possible to exhaust space in the directory chunk before all of the entries have been used up, for example if many files are inserted whose names are much longer than the 'average' previously given.

The syntax of this form of the command is simply:

`Libfile` *`library`* `-list`

(You can reverse the order of the arguments if you prefer.)

Examples

Below are examples of all variants of the Libfile command.

```
Libfile ansilib -create 20 10
Libfile ansilib -create 10 12 -files signal stdio ctype string
Libfile ansilib -insert math lib.*
Libfile ansilib -delete -members sort osalloc time error
Libfile ansilib -extract locale startup
Libfile ansilib -list
```

LINK - LINK OBJECT PROGRAMS AND LIBRARIES

Description of use

The Linker is an essential program for anyone developing programs in a high-level compiled language on the Archimedes personal workstation. Its purpose is to combine the contents of one or more object files (the output of a compiler or Assembler) with one or more library files, producing a final executable program.

Syntax

The format of the Link command is:

`Link -output` *`file`* `[`*`options`*`]` *`files`*

The *`files`* argument is a list of input files; this is described below. `-output` is the only compulsory keyword.

Below is a list of the command line options that the Linker can take. Most of these will only be used occasionally. In the descriptions below, the important, frequently-used options are given first, followed by the less common ones. As usual, capitals are used to denote the alternative shortened form of the keyword.

`-Output`	Name of the linked output file
`-VIA`	Use a file to obtain (further) input file names
`-Case`	Make matching of symbols case insensitive
`-Base`	Set base address for output file
`-Verbose`	Print messages indicating progress of the link operation
`-Relocatable`	Generate relocatable output file
`-Dbug`	Generate an AOF image for use with the Dbug program
`-Module`	Generate an Arthur relocatable module

Notes

- The keyword `-base` is followed by a numeric argument. You can use the prefix `&` to specify hexadecimal, and the suffixes `k` for 2^10 and `m` for 2^20.

- The default base address for the output file is `&8000` (32K). If `-dbug` is specified, the default base address is `&50000` (ie 320K).

- The item *`files`* above is a list of one or more filenames, separated by spaces. This part of the command must be given. Each of the files in the list must be in Acorn Object Format (compiled files) or Acorn Library Format (libraries). They may contain references to external objects (procedures and variables) which the Linker will attempt to resolve by matching them against definitions found in other files.

- You can use wildcards in the filename list. Names using wildcards will be expanded into the list of files matching the specification. For example, the name `o.bas*` might yield `o.basmain`, `o.basexpr`, `o.bascmd`.

- Usually, at least one library file will be specified in the list. A library is just a collection of AOF files stored in a single Acorn Library Format file. You can call the procedures in the library for one language from programs written in another, as long as both languages conform to the ARM Procedure Calling Standard and both run-time libraries use the common run-time kernel. For example, an assembler program could use the C `printf` function, as long as the C run-time system had been initialised, through the common run-time kernel.

- Libraries differ from object files in the way the Linker uses them. Object files' symbols are scanned only once when the Linker attempts to resolve external references. Libraries are scanned as many times as necessary. If a required symbol is found in one of the library's component files, the whole component is incorporated into the output file.

- Two common errors given during a link are caused by unresolved and multiple references. In the first case, a symbol has been referenced from a file (whose name is given in the error), but there is no corresponding definition for the symbol. This is usually caused by the omission of a required object or library file from the list, or the mis-spelling of a symbol in the original source program.

- The second error is caused by a clash of names. For example, a procedure might have been defined with the same name as a library procedure, or as a procedure in another object file. The version of the procedure used in any situation is the one local to the reference to it.

- The `-output` keyword is obligatory. It is followed by the name of the file to which the final linked program should be written. If you just want to use the Linker to check object files for unresolved references, you can specify the device `null:` as the output file; the final object code will be discarded. The output is usually in Arthur Image Format, which can be executed directly. Alternative formats allow low-level debugging with Dbug or the creation of an Arthur relocatable module.

Simple examples

Before we move on to describe the rest of the `Link` command's options, we give some examples using the syntax described so far:

```
Link -OUPTUT p.sieve o.sieve, ansilib
Link -o %.mybasic o.bas* lib.f77
Link -o null: o.comp*
```

Via keyword

Sometimes you may want to link a large number of input files which would be tedious to type on a command line, and whose names can't conveniently be matched by a wildcard specification. Using the `-via` keyword, you can store a list of input filenames in another file and use this to access them. For example, suppose you created the file `basfiles` with the contents:

```
o.main
o.expr
o.cmd
o.stmnt
o.lex
o.filing
o.tokens
```

If you then used the command:

```
*link -o basic -via basfiles lib
```

then the files listed in `basfiles` would be linked, together with the AOF file `lib`.

Case keyword

If you specify `-case` in the command line, then the Linker will not treat the case of letters as significant in identifiers. By default, the identifiers `main` and `Main` refer to different objects, as they are spelt differently. However, with `-case` set, they are the same identifier.

Base keyword

By default, the base address of the output file of the Linker is `&8000`. This corresponds to the start of application workspace on the Archimedes workstation. Alternatively, if the `-dbug` option is given, the base address is set to `&50000`. This is so that the debugger program Dbug can load at `&8000` as a normal application, and load the file to be debugged above itself. (There are other changes when `-dbug` is given, as described below.)

Using the -base keyword, you can set the base address of the output file to any desired value. For example, you may want a program to have a high load address (as with the -dbug option set), but still be directly executable (which a dbug file in AOF format isn't).

The keyword is followed by a number giving the base address desired for the output file, eg -base &80000, -base 256k etc. When this is done, all relocatable objects in the input files are relocated using that base instead of the default.

Verbose keyword

If you specify -verbose on the command line, the Linker gives a report of its progress. A message is printed as each file is opened and as each module is being relocated. For example:

```
link: opening p.basic
link: opening o.bas1
link: opening o.bas2
link: relocating module o.bas1
link: relocating module o.bas2
link: relocating module ansilib (fpprintf)
...
```

Relocatable keyword

Usually, when an image file is produced, it will execute correctly only at the base address given (or the default). This is because the object program will contain references to absolute addresses within the data area. However, if you specify the -relocatable option, the final program will be relocatable. That is, it can be loaded and executed at any address.

This feat is achieved by adding a relocation table and a small program to perform the relocation to the final object code. The relocation table is a list of offsets from the start of the program to words which need relocating. These words are adjusted by the difference between the base address of the

program and the address where it was loaded. Once the relocation has been performed, the program proper starts executing.

The relocation process is very fast, and once it has been performed, the space occuupied by the table is available as part of the program's heap space when it starts executing.

Note that although this ability can be used to make a program statically relocatable, it does not confer true position-independence on the program. That is, the program could not be moved in memory once it has started and still be expected to work.

Dbug keyword

If a program is linked using the `-dbug` keyword, an executable image is not formed. Instead, an AOF file is created which contains all of the symbols found in the original source files. The code segment of the file can be executed under the control of a Dbug program, and the contents of the code and data segments may be examined (and altered in the case of the data segment).

Module keyword

If you specify this keyword, the output file is created in Arthur relocatable module format, suitable for loading into the RMA (relocatable module area). To use this facility, there must be an area called `!!!Module$$Header`, which contains a standard RM header.

> Notes: To be of use, this feature requires the module support in the common run-time kernel. This is not provided in early releases of some compiled languages. Consult your language documentation for details. Note also that the `-relocatable`, `-module` and `-dbug` keywords are mutually exclusive; only one of them can be given on the command line.

Predefined Linker symbols

There are several symbols which the Linker knows about independently of any of its input files. These start with the string `Image$$` and, along with all other external names containing `$$`, are reserved by Acorn.

The symbols are:

`Image$$RO$$Base`	Address of the start of the read-only (program) area
`Image$$RO$$Limit`	Address of the byte beyond the end of program area
`Image$$ZI$$Base`	Address of the start of run-time zero-initialised area
`Image$$ZI$$Limit`	Address of the byte beyond the zero-initialised area
`Image$$RW$$Base`	Address of the start of the read/write (data) area
`Image$$RW$$Limit`	Address of the byte beyond the end of the data area

Although it will often be the case, Acorn does not guarantee that the end of the read-only area corresponds to the start of the read/write area. You should therefore not rely on this being true.

Note also that programs can reside in read/write areas, as they sometimes contain local writeable data (eg modifiable code), and it is possible to have read-only data (eg floating-point constants and string literals in C).

These symbols can be imported as relocatable addresses by assembly language routines that might need them.

The Linker joins all areas (from all input files) with the same name and attributes together to form a single area. It then creates the two symbols *name*`$$Base` and *name*`$$Limit` to mark the start and end of the area. It is an error for two areas to have the same name but different arttributes.

OBJLIB - CREATE/LIST A LIBRARY'S SYMBOL TABLE

Description of use

The Objlib command is used to create or examine the external symbol table of library file. External symbols are those which are visible to modules outside the one in which they are defined. Typically they are the names of library functions (eg `printf` in the C standard library) and global variables (eg `errno`, also from the C library).

When a library is first created (using Libfile), there is no external symbol table. Objlib is therefore used to create the table, which is stored in the library as a chunk of the name OFL_SYMT. See *Appendix A: File formats* for more details of library file formats.

Syntax

The Objlib command has the form:

`Objlib [-File]` *`file`* `[-Create] [-List]`

The *`file`* is the name of a library file. You cannot list the contents of the symbol table before it exists, so to list the symbols in a library for the first time you would use something like:

```
Objlib libfile -c -l
```

Thereafter, the symbols may be listed using just `-l`. The output from the command goes to the screen, but may be redirected using the usual Arthur method, eg:

Objlib libfile -l { > syms }

Note that if the library is altered by Libfile, then the symbol table will become out of date and should be re-created using Objlib with the `-c` option. (Objlib will tell you if the symbol table is out of date.)

Output from the program looks like this:

```
External Symbol Table, generated Thu Mar 17 09:30:34 1988
clock from object file armsys
time from object file armsys
_sys_msg from object file armsys
....
....
pow from object file armstart

End of Table
```

Of course the symbol and filenames will depend on the contents of the library file being examined.

SQUEEZE - COMPRESS AN AIF FILE

Description of use

The Squeeze utility is a program compactor. It takes an AIF file (eg the product of an execution of the Link program) and compresses it by a factor of just less than two. The compressed program can be executed directly and it 'expands' automatically when it is run. The advantages of using Squeezed programs is that they occupy less space on a floppy disc, and therefore take less time to load.

The exact saving in space depends on the contents of the image file. If it has many zeros (eg a large area of initialised static data in a C program), a factor of greater than two may be achieved. A hand-coded assembly language program, which contains a greater diversity of instructions than one produced by a compiler, would not achieve such a high compression ratio (3:2 being typical).

Syntax

The Squeeze command has the format:

```
Squeeze [-v] srce-file [dest-file]
```

If the -v flag is given, a more verbose form of the progress report messages is produced. Even if you omit the option, Squeeze still produces some information about the process. In particular, it tells you what reduction factor was achieved and how long it took.

The form with only one filename will reduce the given file *in situ*, overwriting the original with the new compacted form. If you give both filenames, the original is left intact, and the compressed version is stored in the second named file.

Examples

Below are two examples of the use of Squeeze. After the first example, typical output from the command is also reproduced.

```
*squeeze mint
-- squeezing 'MINT' to 'MINT'
-- encoding stats (0, 1, 2, 4) 9% 70% 19% 0%
-- compressed size 17519 is 57% of 30388
-- compression took 68csec, 44688 bytes/cpusec
-- getting timestamp from Arthur

*squeeze -v mint lib*.mint
```

INTRODUCTION

The Dbug utility is a versatile and powerful machine code debugger which can be used with high-level language programs. The input to Dbug is a file containing executable code. This file might have been linked using the `-dbug` option, in which case it will also contain symbol table information. See the section *Link - link object programs and libraries* in the previous chapter, *Program manipulation program utilities*, for more details. Alternatively, other types of code file may be loaded, as described below.

As with many programs of its type, Dbug is interactive and is characterised by short command names. All commands may be abbreviated to one character (usually a letter). Before we describe the commands available and their syntax, we give the options available on the Dbug command line itself.

Syntax

The Dbug command has the form

`Dbug [`*`file`*`] [-Limit `*`addr`*`] [-Rs423 [-Baud `*`tx/rx`*`]]`

As all of the arguments are optional, you can just issue the command on its own. This enters Dbug, which starts with the prompt:

```
Dbug:
```

At this stage, you could use various Dbug commands, such as those to examine and alter memory locations, call particular routines etc. However, many Dbug commands require the presence of a file so that addresses of code and data may be expressed in terms of symbols (and so that there is actually a program to debug). Thus, the command usually includes at least a file name, eg:

```
Dbug o.prog
```

When Dbug starts, it will load the program part of the given file at its code base address, which is usually `&50000`. The symbol table of an AOF file is loaded into the free area that Dbug has between the end of its own code and

data and the upper limit which it can use. Typical values for these two addresses are `&28000` (160K) and `&50000` (320K), leaving 160K for symbol tables.

If the file isn't an AOF one, it can still usually be loaded and debugged, but symbol information will not be available. If the load address of the file is greater than or equal to 320K, it is loaded there. If the file is a relocatable Arthur Image Format file, it is loaded at 320K and the relocation code executed to allow execution there. If none of the above applies, Dbug will ask if it should load the file at 320K. The file should contain position independent code in order for 'yes' to be a sensible answer.

The other parts of the command line are seldom required. The `-limit` keyword gives the upper limit of memory that Dbug can use. It is followed by a number which may be prefixed by `&` for hexadecimal and suffixed by `K` for *1024. The default is 384K. You could increase this to `&80000` (512k) if absolutely required, but usually the debugger will function without the extra space.

The `-rs423` keyword tells Dbug to use the RS-423 port for its input and output, instead of the keyboard and screen. This can be useful if you are debugging a program whose interaction with the screen might be confused by the presence of Dbug messages. The baud rate defaults to 9600 for transmit and receive, but may be altered using the `-baud` keyword.

Command examples

Below are some typical ways in which Dbug might be called:

```
*Dbug
*Dbug o.myfile
*Dbug o.myfile -limit 512k
*Dbug -rs423
```

DBUG COMMAND MODE

Dbug responds to about 20 commands. As mentioned above, these may all be abbreviated to one character, and all but two of them are letters. In fact, Dbug

only looks at the first character of a command, and ignores subsequent characters until the first space. Upper and lower case are not differentiated in Dbug command names.

Before using Dbug, you should be aware of its limitations. Under an operating system such as Arthur, which provides no memory protection, it is difficult to guard against a fault in the debugged programmer crashing the debugger. For example, if the program being examined starts to write all over Dbug's program code, there is little that can be done about it. Thus if the debugger appears to crash in mysterious circumstances, you should suspect your program first.

You should also note that Dbug is not designed to be used on programs which execute in SVC (or IRQ or FIQ) mode. It is a user-mode program utility.

Many of the commands take parameters of various types, so we will describe the form these take before moving on to the commands themselves.

EXPRESSIONS

In general, when Dbug requires a number, eg an address or a count, a general expression can be used. This section described the elements that expressions may contain.

Operands

The operands in an expression are numbers in various bases, register numbers, symbols found in the loaded symbol tables, and a couple of special characters.

Numbers are treated as decimal unles prefixed with an `&` for hexadecimal. Examples are `123` and `&1ff` (511). Alternatively, you can use the form *`base_digits`*, where *`base`* is the base to be used (in decimal), and *`digits`* is a string of one or more digits in the ranges 0-9, A-Z, as appropriate to the base.

A character constant has the form `'`*`chars`*`'`. *`Chars`* is between one and four characters. The ASCII value of the first character is placed in the least

significant byte of the 32-bit integer, the second character in the next byte, and so on. Missing characters are taken as zero. For example, `'A'` evaluates to 65, `'01'` to `&3130`. Characters may contain all of the standard C language escape sequences, eg `'\n'` for the newline character.

A register number is the letter `r` followed immediately by a decimal number in the range 0 to 15. Examples are `r0` and `r13`. Usually a register evaluates to its number, so `r15` just means 15. However, in some commands, an expression is used as an address whose contents are required. In these commands, a register number is replaced by the register's contents.

Floating point registers have the letter `f` followed by a digit in the range 0-7. The floating point status register is accessed using `FPPSW`. Note that the floating point emulator (or hardware) must be present for you to be able to use these registers.

There are some special register names which correspond to those defined in the Acorn procedure calling standard:

`fp`	`r10`	Frame pointer
`ip`	`r11`	Work register
`sp`	`r12`	Stack pointer
`sl`	`r13`	Stack limit
`lr`	`r14`	Link register
`pc`	`r15`	Program counter (without the flags)

Two special characters that may be used in expressions are period (`.`) and tilde (~). These stand for addresses used in recent commands, saving you the need to type the whole expression again. Period stands for the last address that was examined or altered in an `E` or `D` command. Say you examine the contents of addresses in the range `&24` to `&38`. Then using `.` in an expression will yield the address `&38`.

`.` is set to the start address when a file is loaded, and to the program counter when control returns to Dbug (after a Single Step, for example).

Tilde, when used in an `E` or `D` command, yields the first address used in the previous command of the same type. Using the example above, ~ would evaluate to `&24`.

Symbols used in expressions evaluate to the address defined for that symbol. Any symbol which is in Dbug's symbol tables may be used. You can load separate symbol tables, from AOF files other than the one given on the command line. This can be useful in certain advanced applications.

A symbol which is the name of a procedure evaluates to the entry address of that procedure. This is useful when, say, you want to set a breakpoint at some offset from a procedure's entry point. If a symbol is the name of a piece of static data, the resultant address is the first location occupied by that data. Some symbols are 'constants' which do not refer to particular objects, but give information about the object file. For example, the symbol `Image$$CodeBase` evaluates to the lowest address used by the code area of the file. You can obtain a list of symbols that the debugger knows about using the `L` command.

When giving the name of a symbol which begins with a `$` sign, you should enclose it between vertical bars, eg `|$entry|`. This prevents confusion with the format strings which also begin with `$`, and which are described below.

If you suffix a symbol name with `{`*n*`}`, the symbol will only be searched for in symbol table *n*. This can be useful to ensure that you are using the right symbol when you have more than one table loaded.

Operators

The debugger understands seven different operators and allows the use of brackets. Four of the operators are the usual arithmetic ones:

*	Multiply
/	Divide
+	Add
-	Subtract

Multiply and divide have a higher precedence than add and subtract, so an expression such as `1+2*3` would evaluate to `1+(2*3)=7` not `(1+2)*3=9`. You can use brackets to over-ride this, eg `(5-2)*(3+7)`.

Unary plus and minus are allowed. The former has no effect on the expression; the latter subtracts the value of the item following it from zero. The unary operators have a higher precedence that the binary ones above, so `-1+1` evaluates to 0, not -2.

The final 'arithmetic' operator is `^`. This is a unary postfix operator, which means it takes one operand, which it follows. Its meaning is 'contents of', much the same as the Pascal operator of the same name. The operand is either an address, in which case the contents of the word at that address are fetched, or a register, in which case the register's contents are fetched. `^` has a higher precedence than unary plus and minus, so `-12^` means minus the contents of address 12, not the contents of address minus 12.

In addition, there are some relational and logical operators. The former perform unsigned comparisons on integers. They are:

=	Equal
<>	Not equal
#	Not equal
>=	Greater than or equal
<	Less than
<=	Less than or equal
>	Greater than

All of these return either `TRUE` or `FALSE` when displayed, which are represented as 1 and 0 internally. They may be combined with the logical operators:

`AND`	Logical AND
`OR`	Logical OR
`NOT`	Logical NOT

which obey the usual truth-tables. Note that these are *only* logical operations, not bitwise ones. Examples of expressions are:

`main`	The address give by the symbol `main`
`.+&100`	The last address in a `D` or `E` command plus `&100`
`_iob^`	The contents of the data at the address `_iob`
`(arr2-arr1)/4`	The difference in two symbols' values divided by 4
`.=main`	The . symbol has the same value as `main` NOT `main<printf`

Note: it may appear that Dbug can handle both signed and unsigned values. However, internally it treats all operands as unsigned. The fact that expressions such as `-2-4` produce the expected result is due to the fact that two's complement and unsigned arithmetic are the same thing on the ARM. Thus you should bear in mind that a number such as `-1` (`&FFFFFFFF`) is always treated as a very large positive number, never a small negative one. This is illustrated by the expression `-1 > 0` yielding `TRUE`.

FORMATS

Dbug is very flexible in the way in which it can display values. There are three main attributes that data may take. These are its 'style', its 'base' and its 'size'. The first attribute determines whether a value will be shown as a number, as a symbolic name, as an instruction and so on. The base attribute refers to numbers, and determines which base they will be displayed in. Any base between 2 and 36 may be used. Finally, the size relates to the width of objects used in certain commands.

Most commands have default attributes, but these may be over-ridden. An attribute begins with the character `$`. This is followed by a single letter, or a number in the range 2 to 36 when an arbitrary base is specified.

Style formats

Below is a list of the styles available.

`$c` Character. The value is shown as a sequence of between one and four characters. ASCII values in the range 32 to 126 are displayed directly. All other values are shown in the form `\`*num*, where *num* is

the numeric representation of the character code. This uses the current numeric base (which defaults to hex), but may be over-ridden by specifying a base attribute too.

`$i` Instruction. This is the default style for many instructions. The value is interpreted as a four-byte (one word) ARM op-code, and decoded appropriately. The instruction is displayed in standard ARM format. Registers in the special group listed above are displayed by name, rather than number. Numeric values, eg immediate operands, shift counts and immediate offsets, are displayed in the current numeric base.

Destinations of branches are displayed symbolically if possible. If not, the destination is given as an absolute address. If this can't be calculated (because there is no particular address associated with the op-code), a byte offset from the instruction is given.

`$s` String. This treats the value as a series of characters, and prints them in `$c` format. The string is terminated by the byte `&00`.

`$y` Symbol. In this format, Dbug tries to display the value as a symbol whose value is found in the symbol table. If an exact match is found, ie there is a symbol whose value is exactly equal to the expression, then the symbol's name is displayed. If not, the symbol with the nearest value less than the expression is used, with an offset added in the current base. Because there may be more than one symbol with the same value defined (particularly in the case of constant symbols), you may not always see the expected name displayed. If the value is less than any symbol, the numeric version is used.

`$p` PC part. This format displays its operand in the current numeric base. However, before doing that, it bitwise ANDs the value with `&3FFFFFFC`. This effectively masks out the bits which correspond to the flags in the ARM's R15 register. The main use of this format is when you are examining the contents of R15 (PC) or R14 (LR) and want to see only the address held there, not the status bits too.

`$f` Flags part. This complements the previous format. It uses only the bits of the operand which correspond to the flags and mode bits in the ARM's R15, and prints a textual interpretation of them. If any of the status bits NZCVIF are set, then the appropriate letter is displayed. This is followed by one of the strings `User`, `FIQ`, `IRQ` or `SVC`, according to bits 0 and 1 of the value.

`$n` Numeric. Values are displayed as numbers when this style is specified. The current base is used, unless a base attribute is also specified in the command. The base defaults to hexadecimal when Dbug starts up. Values are always displayed as unsigned in this style. See `$z` below for an alternative.

`$z` Signed numeric. This style also displays values as numbers, but if the top bit (determined by the size of the operand) is set, then it prefixes the value by a minus sign and then displays the absolute value. This applies to all bases, not just decimal, so you can see things like `-&1` for a signed, hexadecimal value.

`$r` Real. This style allows the display of IEEE floating point values. The precision used is determined by the size attribute: one, two or three words for single, double and extended precision respectively.

`$q` Packed real. This format interprets the value as a three-word packed floating point number.

`$v` FP flags. This format interprets the value as a set of flags whose bits correspond to those in the FPU processor status register. When you examine the FPPSW, this format will be used by default. It is also useful for examing memory locations which are supposed to contain dumped versions of the FPPSW.

Base formats

Below is a list of the bases that can be used.

`$d` Decimal. A decimal number is just a sequence of digits in the range 0 to 9.

`$x` Hexadecimal, prefixed by the `&` character. Letters are printed in upper case.

`$o` Octal (base 8). Numbers are printed in the form `8_digits`. Octal digits are in the range 0 through 7.

`$n` Base *n*. This may be in the range 2 to 36. The letters A through Z are used as highest digits in the bases 11 through 36 respectively. Numbers are printed in the form `n_digits`, except for bases 10 and 16, which are printed as described above.

Size formats

Usually operands are treated as being four-byte quantities for display purposes. However, for particular commands it might be more useful to treat them differently. The size attribute allows this.

`$e` Three word. This size is used for extended and packed precision real numbers.

`$t` Two word. This size is used for double precision real numbers.

`$w` Word. This is the usual four-byte integer interpretation of operands, or single precision reals.

`$h` Half-word. This attribute gives a 16-bit (two-byte) interpretation of operands.

`$b` Byte. When this attribute is set, operands are treated as individual bytes.

Setting default formats

As mentioned above, most commands use a set of default formats for their output. Unless a note to the contrary is made, you can assume that the commands described below produce their output using the current default attributes. When Dbug starts, these are set to: `$I $X $W`.

You can alter the default formats using the format type as a command. For example, the command:

```
$N
```

causes the default style to become Numeric instead of Instruction. Thereafter, commands which usually produce assembly listings (eg the Examine command) will display numbers instead. Similarly you could issue the command:

```
$H
```

to set the default size to half-word.

DATA COMMANDS

This section lists the commands which are concerned with the examination and altering of memory locations and registers. It also covers the commands which produce useful information, such as =, which displays the result of an expression, and L, which lists symbols.

D – Deposit values

This command stores a given value in a memory location or register. The syntax is:

```
D addr val [format]
```

The first argument, *addr*, is an expression giving the address (or register name or number) of the first byte to be altered. An address may be numeric, or *@name*, where *name* is the name of a user variable. See the section *Variables and macros* below for details on user variable.

The *val* argument is an expression giving the value to be stored there. The only format allowed (if one is given) is a size attribute. This determines how many bytes will be affected by the command.

If the size is omitted, the the current size is usually used. Note however that certain types of operand (eg machine registers) have their own 'natural' sizes which over-ride the current setting. In particular, for ARM register and the FPPSW the size defaults to `$w`; for FP registers it is `$e`.

Examples are:

`D r0 123`	Store the value 123 in R0, zeroing the top three bytes
`D r0 123 $b`	Store 12 in R0, leaving the top three bytes unaltered
`D mystr &00434241`	Store the characters A, B, C, NULL at mystr
`D FPPSW &17`	Store INX,OFL,DVZ,IVO flags as set
`D @ADDR main^`	Set a user variable - see *Variables and macros*
`D F1 21`	Set the contents of an FPU register
`D .+4 (-1)`	Store the value -1 at the word after the last one Note the brackets around -1 to avoid ambiguity

E - Examine memory or registers

Examine complements the Desposit command described above. It allows you to see the contents of a series of memory locations or registers. The syntax of the command is:

`E [`*`address-range`*`] [`*`format`*`]`

The *`address-range`*, if specified, can take one of two forms. If it is omitted altogether, it means 'the single location after the previous one examined'. This allows you to type a sequence of `E` command, with each displaying successive locations.

An *`address-range`* of the form *`start`*`:`*`end`* gives the first and last address (inclusive) of the block to be examined. Both values can be expressions. The output of the command displays values at successive locations until the *end* location would be exceeded by the next line. The address of each location exceeds the one before by the size attribute being used.

If you give the address range in the form *`start`*`,`*`count`*, then *`count`* lines of output are produced, starting at address *`start`*. In fact, it is equivalent to

start:*start+count*size-1*, where *size* is the size attribute used during the command.

At the end of every E command, Dbug sets the special symbols . and ~, as described in the section 'Operands' above. It also sets an internal 'next location' value, equal to .+*size*. This is used as the location for the next E command if you do not specify any addresses.

The output of the E command is a series of lines with the following format:

label: *contents*

The *label* is the current location, displayed using the $Y format. Thus, if possible Dbug displays the location symbolically. If it can't, it displays it in the numeric base used during the command. The *contents* are displayed in the style format used during the command. Unless this is over-ridden, it will be the default of $i for word-length operands and $n for other lengths. Default formats for ARM registers, FPU registers and FPPSW are $n, $r and $v respectively.

Examples of use of the E command are:

E 0,4	Display the contents of words at &0, &4, &8, &12
E 0,4 $d	As above, but all offsets etc. are in decimal
E 0,4 $n	Display the contents of the 4 words numerically (hex)
E 0,5 $n $d	Display the contents of the 4 words in decimal
E	(After the previous one) display the contents of the word at address 16
E 0,4 $b	Display the contents of the bytes at &0, &1, &2, &3
E main:err	Display the words at main to err as instructions
E r0,16	Display all of the user registers' contents
E lr $f	Display the flags part of R14
E fppsw	Display FP flags
E f0	Display F0 as a real
E f1 $n	Display F0 as three words

F - display the registers

This command displays the contents of all the ARM's 27 registers. The 16 user/all-mode registers are displayed, followed by the two SVC mode registers, the two IRQ mode registers, and the seven FIQ mode registers.

You can also use `E R0,16 $n` to display just the user mode registers.

= - display an expression

This is a general-purpose display command. It has the syntax:

`= [`*`expression`*`] [`*`format`*`]`

The *`expression`* can involve any of the operand and operator elements described above. If it is omitted, nothing is displayed. The *`format`* may be used to over-ride the current defaults. As the start-up style is Instruction, you will usually use `$n` in this command to display the results numerically, or issue a `$n` command to set the default style to numeric.

Examples of the use of = are:

```
= main $n          Print the value of the symbol main
= 1234 $n $2       Print 1234 in binary
= main^            Display the instruction at main
= Image$$DataLimit-Image$$DataBase $n
= pc^^             Display the instruction addressed by the PC
```

FILE AND SYMBOL TABLE COMMANDS

This section covers the commands relating to whole AOF files and the symbol table part of AOF files. The only AOF-related command is `G`, to load a file. It is described below.

G - get an AOF file

This command has the form:

```
G file
```

and is used to load the code, data and symbol table chunks of an AOF file into memory. The code and data parts are loaded at the locations appropriate to the base address used when the file was linked (and given by the symbols `Image$$CodeBase` and `Image$$DataBase`). The symbol table is loaded into the free space between the end of the Dbug program and the memory limit available to it. It adds to (rather than replaces) any symbol tables already loaded.

A use of this command is when you have started using Dbug without specifying a filename on the command line, and then want to load one. An example is:

```
G o.test
```

L - list symbols

This command lists symbols from the current table(s). Its syntax is:

```
L [pattern] [format]
```

The default format is `$n`. You can over-ride this to print the symbols' values in other bases, or perhaps as characters. You can't over-ride the size, which is always word.

If you specify a *pattern*, only those symbols which start with the characters in the pattern (and with the same letter case) will be displayed. Examples are:

`L _`	All symbols that start with an underscore
`L $2`	All symbols, in binary
`L $8 C$$`	All symbols that start with `C$$`, in octal

X - handle symbol tables

There are three forms of this command, dependent on the letter that follows it. They are:

`X G` *file*	Get the symbol table from the AOF file given
`X D` *file*	Dispose of the symbol table obtained from the named file
`X L`	List the symbol tables

The memory space between the top of the Dbug program and the limit specified by the `-Limit` command line parameter is used to hold symbol tables. As mentioned earlier, when an AOF file is loaded, its symbol table is automatically loaded too. You can see this by using the `X L` command, which produces output of the form:

```
Table 1 test
```

where `test` is the filename.

There is a small chance that if you load many symbol tables, the space used to store them will eventually become full. To avoid this, you can delete memory symbol tables that are no longer required using the `X D` command.

Finally, `X G` is used to load a new symbol table from the AOF file given.

PROGRAM EXECUTION COMMANDS

Clearly there is more to debugging a program than examining memory locations and displaying expressions. This section describes an important set of commands concerned with controlling the execution of the debugged program.

R - run the program

This command starts execution of the program. It has the form:

```
R [arguments]
```

You do not have to specify an address. Dbug starts execution from the entry point of the program, given by the symbol `Image$$CodeBase`.

The arguments, if present, are passed directly to the program. They can then be read using whichever method the program normally uses (eg via `*argv[]` and `argc` in C).

If you just Run a program, it will execute until it either terminates normally, or some exception such as illegal address or undefined instruction occurs. At this point, control will return to Dbug. It is usual to set breakpoints before executing a program, so that it is interrupted at a well defined place, enabling the stack and variables etc. to be examined.

B - handle breakpoints

Breakpoints are set, cleared and listed using this command. A breakpoint marks an instruction which, when executed, will cause control to return to the debugger instead of allowing the program to continue normally. The three forms of the command are:

```
B S address ["commands"]
B D [address]
B L
```

The first form Sets a breakpoint at the *address* given. This can be a general expression yielding a suitable word-aligned address. The string, if present, is a list of commands to be executed when the breakpoint is encountered. For example, you might use:

```
B S getstr "F"
```

to display all the registers when the breakpoint at the entry of `getstr` is executed. Multiple commands may be separated using semicolons (`;`).

You can also set conditional break points. The first 'command' in the string should be an expression which yields either `TRUE` or `FALSE`. Whenever the

breakpoint is encountered, the expression is evaluated. Execution only terminates if it yields TRUE or an error. An example is:

```
B S process "count^=10;e array,5"
```

The second form of the command is used to Delete a breakpoint. If the *address* is given, only the breakpoint there is deleted, otherwise all breakpoints are removed, with a prompt for confirmation before each one is removed.

The last form Lists the current breakpoints. An example of the output from the command is:

```
Breakpoints
code main
code getstr+&48
```

S - single step

Instead of running the program at full speed until a breakpoint is met, you may want to step through the code at a more sedate pace, examining the state of execution as you go. The S command allows you to do this. It has the form:

```
S [count]
```

count is the number of instructions you want to execute. The default is one. At the end of each Step command, the next instruction to be executed is displayed. Note that if you use, say:

```
S 10
```

to step through ten instructions, the command still only displays one instruction - the next one.

T - trace

The T command allows you to Trace program execution. It is similar to executing an infinite succession of S commands, but rather faster. As with Step, each instruction is displayed just before it is executed.

Note that like R, the only way to stop a Trace is to execute a breakpoint. You should therefore be careful not to start tracing a section of code with a long, uninterrupted loop, unless you are prepared for a long wait.

C - continue execution

Having interrupted the program with a breakpoint, you can use any of the commands that Dbug provides to examine and perhaps alter the execution state. Eventually you will want to resume execution. You do this using the C command. It continues execution from the next instruction, which will be just after a breakpoint unless you have single-stepped on.

U - unwind the stack

This command Unwinds the stack, displaying the contents of the frame pointer (R10) and location of the procedure call which created the stack frame. The syntax of the command is:

```
U [C|P] [F] [count]
```

If the *count* is omitted, the stack is unwound until the stack base is reached. The stack base is the highest memory location occupied by the (descending) stack, and marks the start of procedure stack frames. If you include the *count*, only that number of stack frames are shown.

If you give the C option, then the unwind continues from the previous U command. This would had to have been one which specified a count for the C to be useful. Specifying P causes the stack to be unwound from the same level as the previous U command.

The F option causes all stacked registers to be displayed, in addition to the information mentioned above. On entry, a procedure saves the pc, lr, ip and fp

registers, in addition to any registers that it uses and needs to preserve under the Acorn procedure calling protocol. For each register that was saved, its name and contents are displayed, using the `$Y` format.

> Note: this command relies on the stack being organised according to the Acorn procedure calling standard for its operation. Programs which do not follow this standard cannot have their stack checked using the Unwind command.

J - jump to a procedure

A - set procedure arguments

It is sometimes useful to be able to call a procedure in the debugged program independently of the normal flow of control of the program. For example, you might want to test a procedure from the debugger before incorporating it into the program. Alternatively, you might have a procedure written specifically to be used from the debugger, for example one which prints out complex data structures which would be tedious to examine directly from Dbug.

The `J` command can be used to call a procedure in the manner described above. It has the syntax:

```
J address
```

The *`address`* is the entry point of the required procedure, and so is usually just a symbol, eg:

```
J disptree
```

You can set up the first four parameters of the called procedure using the `A` command. It has the syntax:

```
A arg value
```

The *`arg`* is in the range 0 to 3. The *`value`* can be any expression appropriate to the argument that is being initialised. For example, suppose the `disptree`

procedure from the previous example takes a pointer to a data structure which is held in the static variable `treeroot`, you might use this before calling it:

```
A 0 treeroot^
```

On return from the procedure, the result (ie the value in R0) is displayed, as in:

```
Function result     1E
```

> Note: Using the `J` command is potentially dangerous. For example, if the called procedure assumes some facts about its run-time environment which aren't true, it might execute instructions (eg by calling a library procedure) which result in incorrect operation. It should be classed as a 'use with caution' command.

VARIABLES AND MACROS

Dbug provides some facilities for making debugging more convenient. The first of these is variables. A variable is a named object where you can store data to be used later. Variables obviate the need to make notes of important addresses etc - they can just be stored for later access.

The second device is macros. A macro is a named 'string' which contains a list of commands that can be executed. Storing common command sequences in macros can save a lot of typing.

Variable and macro names are separate from each other and from symbol names. There is therefore no need to worry about clashes of names.

V - create a variable

This command has the form:

```
V @name [size]
```

The *name* part is a sequence of one or more alphanumeric characters, the first one being a letter or underscore. The *size* is the number of bytes required for

the variable, with four being the default. It can be arbitrarily large, so you can use variables to store strings. The contents of the variable are not initialised at all.

Variables may be used in the Examine and Deposit commands' address part. For example:

```
E @addr
```

would show the contents of the variable called `addr`, and:

```
D @addr 12 $b
```

would set the first byte of that variable's value. Variables can also be used as the operand of the `^` operator, as in:

```
D @cnt @cnt^+1
```

to increment a count.

M - Macros

The `M` command is used to create, list and execute macros. To create a macro, a command of the form

`M C` *`name`* `"`*`commands`*`"`

is used. The *`name`* obeys the same rules as those for variables (but doesn't have a leading `@`). The *`commands`* part within the quotes is a sequence of Dbug command, separated by semicolons. An example is:

```
M C A "E @here^,20;D @here @here^+80"
```

This sets the macro called `A` to be an `E` command, which happens to use the contents of a variable as one of the operands. The second part of the command increments the variable by the number of bytes examined.

It is a good idea to keep macro names short, as they are supposed to save typing. You can execute the macro using:

```
M X A
```

This will execute the Examine and Deposit commands exactly as if they had been typed at the keyboard.

Finally, you can use:

```
M L A
```

to list the contents of the macro. If you don't give a name in the `M L` command, all macros are listed.

MISCELLANEOUS COMMANDS

This sections describes those commands which don't come under any of the previous headings and which, in general, are fairly esoteric.

K - set MEMC domain

This command sets the 'domain' for use by the Examine command. A domain is a property of the memory mapping performed by the MEMC chip in the Archimedes workstation, and is not currently used under Arthur. The syntax of the command is:

```
K domain
```

where *domain* is a number between 0 and 15. This command is best ignored at the moment.

P - set register save address

This command has the syntax:

```
P [addr]
```

It sets the address where Dbug believes the registers have been saved. This allows, for example, the examination of the saved register set of a thread which is not the current one. The default value may be restored using the P command on its own.

As with K, this command has little application under Arthur.

Q - quit from Dbug

This command quits Dbug. It does not verify your decision to exit the program, so you should be careful not to type it accidentally.

*** - issue an OS command**

As with most interactive programs running on the Archimedes personal workstation, Dbug will treat a line starting with a * character as a string to be passed directly to the operating system. This makes the full power of the Arthur command line interpreter available to the user.

This chapter describes not a complete program, but a BASIC library file, which can be used to advantage by any other BBC BASIC program. The library is called `BASIClib.Shell`. It contains the following function and procedure definitions:

`PROCAssemble_Shell`	Assemble machine code for the Shell
`PROCShell(command$)`	Call the Shell code with a string
`FNShell_String_UC(c$)`	Convert string to upper case
`FNShell_Array(c$(),c$)`	Split string into words

There are really two aspects to the Shell library. The first is concerned with calling applications from within BASIC, and enabling BASIC to continue when the application completes. This ability enables you to use BASIC as a command language (such as the C or Bourne shell under Unix). You can call compilers, linkers, editors and other such programs from BASIC, and use the full power of the language to change the way in which such programs are called based on variables such as user input, the results of previous compilations etc.

The second use of the Shell library is in decoding parameters from the BASIC command line. There is no built-in command in BASIC for reading command line parameters (such as `argv` and `argc` in C). The Shell library provides a way of accessing the BASIC command line, and splitting this up into 'words'.

THE SHELL FACILITY

We will describe the use of the Shell procedures first. To see why these procedures are required, you have to understand what happens when you start an application running using a command such as `*link` or `*f77`. Applications running under Arthur are allowed to use memory from the lower limit of `&8000` up to a variable upper limit, called HIMEM, whose value depends on the memory size of the machine and on how much memory has been configured for other resources such as the screen and modules.

Most RAM-based applications load at `&8000`, have a stack growing down from HIMEM, and use the area in between for dynamic (heap) storage. Programs written in C (eg the Diff and Common utilites described in the

previous chapter) are examples of such applications, as are the compilers and linkers that run under Arthur.

When an application terminates, it executes a SWI called OS_Exit. This causes control to return to the most recent program that set up an 'exit handler'. This is usually the operating system, so applications return to the Arthur * prompt when they finish.

Consider now what happens when you call an application from a BASIC program using the * command facility (or equivalently the `OSCLI` statement). The application loads at `&8000`. This is a disaster for a start. ROM BASIC's workspace starts there - about 4K of internal variables, followed by the BASIC program and its variable. RAM BASIC actually executes there, so would be over-written by the new application. Then, when the application terminates, it returns to the OS, so even if BASIC was left intact, control wouldn't return to the BASIC program after an application terminates.

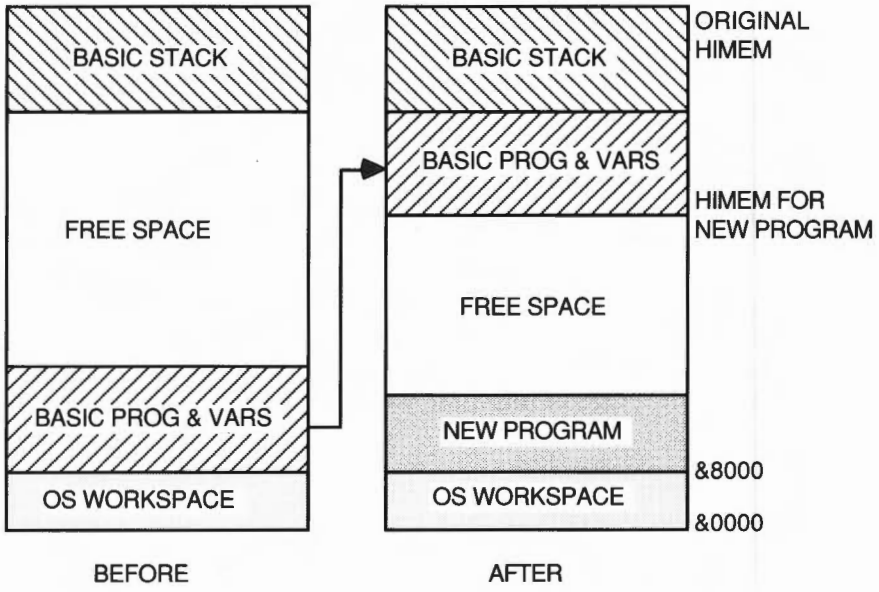
The upshot of this, and the Shell library's raison d'etre, is that you can't call applications from BASIC using the normal methods. Note that we are talking only of applications here. It is of course perfectly possible to call other types of command, eg module commands, transients etc. These return using a simple `MOV PC,R14`-type instruction, and don't interfere with application workspace.

To enable applications to be run from BASIC, the Shell library does the following. First, the program using the library must call `PROCAssemble_Shell`. This assembles the machine code which is subsequently called by `PROCShell`, and which is the key to the whole thing. When you subsequently execute a statement such as:

```
PROCShell("f77 -link fred")
```

the command string is placed into a buffer, and the machine code assembled previously is called, with the address of the buffer as a parameter.

The Shell machine code ensures that, before the operating system is called to execute the command given, the BASIC workspace is moved to a safe place. It also sets up an exit handler, so that when the application terminates, control passes back to BASIC (or more accurately, the Shell machine code, then BASIC). The phrase 'a safe place' above means HIMEM. All of the memory used by BASIC from `&8000` to `END` (which gives the highest address used by BASIC's variables) is copied up towards HIMEM. The operating system's value for HIMEM is then altered to be just below the relocated BASIC storage. This is shown diagrammatically below:



When the application terminates, the new exit handler copies the program down again and restores HIMEM, so that the BASIC interpreter knows nothing of what has happened. Note that any OS_CLI command may be called by using `PROCShell`, but for commands such as `*CAT` and `*MODULES`, it will involve a lot of work which could be avoided using `*` or `OSCLI`.

READING COMMAND PARAMETERS

It is quite likely that a BASIC program which uses PROCShell will itself be called as an OS_CLI command. As you may know, the Arthur Alias$@RunType facility lets you execute BASIC programs directly from the command line. For example, if you type the command:

```
MakeC
```

and `MakeC` is a file with type FFB (BASIC), the actual command executed by the operating system will be:

```
BASIC -quit "MakeC"
```

This causes BASIC to start-up, load the program (which may be text or tokenised), execute it, then exit back to the OS. Using this facility, you can write utilities which act as though they are written in machine code or a compiled language, but with the convenience of using an interpreted language.

It is useful if you can pass parameters to the BASIC program in the usual way, by appending them to the command name. As mentioned above, BASIC has no built-in way of reading such parameters. However, `PROCAssemble_Shell` sets up a string variable `Shell_Env$` which contains the text of the command following the program name. For example, if you type the command:

```
f77 -opt XO myprog
```

then after the (BASIC) program `f77` calls `PROCAssemble_Shell`, the string `5Shell_Env$` will be set to `-opt XO myprog`.

This is a useful first step. However, it is even more useful if the command parameters can be manipulated as an array of words instead of a single string. This is where `FNShell_Array` comes in. It is called with two parameters. The first is a string array, and the second is a string of space-separated command words (eg `Shell_Env$`). The function splits the string into its constituent words and assigns the elements of the array to them. It returns the number of words read.

If we use the value of `Shell_Env$` used in the example above, then the call `FNShell_Array(a$(),ShellEnv$)` will return 3 and elements 0, 1 and 2 of `a$()` will contain `-opt`, `XO`, and `myprog`.

The final function is useful when you examine the parameters for options or 'switches'. By convention these begin with a minus sign, and may be upper or lower case. The function `FNShell_String_UC` takes a string parameter and returns the same string, but with all the letters in the string forced to their upper case versions.

All programs should allow the `-help` switch as the first parameter, so a typical line near the start of a BASIC Shell program might be:

```
IF FNShell_String_UC(a$(0))="-HELP" THEN ...
```

This assumes that the parameters have already been split into words.

USING THE SHELL LIBRARY

Now you understand what the Shell library does, we give some examples of its use. Programs which use the library should start by loading it using a `LIBRARY` statement.

Next, the program should call `PROCAssemble_Shell`. This both assembles the machine code used by `PROCShell` and sets up `Shell_Env$`. If the program expects arguments on the command line, it could use `FNShell_Array` to read them. It should at least check for `-help`.

Putting these first few actions togther, we have:

```
1000 REM >filecat
1010 LIBRARY "$.BASICLib.Shell"
1020 PROCAssemble_Shell
1030 DIM argv$(20)
1040 argc=FNShell_Array(argv$(),Shell_Env$)
1050 IF FNShell_String_UC(argv$(0))="-HELP" THEN
1060   PRINT "Filecat concatenates zero or more files"
1070   PRINT "Syntax: Filecat [file]... [-to file]"
1080   END
1090 ENDIF
```

This is followed by the body of the program. In the current example, no further use is made of the procedures and functions in the Shell library. However, the program is listed below as it does show how `argc` and `argv$()` are used. As you can see from the syntax description printed when `-help` is specified, the command is followed by a list of filenames, optionally followed by a destination specification. If no source files are given, the keyboard is used; if no destination is mentioned, the screen is used.

```
1100 IF argc >= 2 AND FNShell_String_UC(argv$(argc-2)) = "-TO" THEN
1110   dest$ = argv$(argc-1)
1120   argc -= 2
1130 ELSE
1140   dest$ = "rawvdu:"
1150 ENDIF
1160
1170 IF argc = 0 THEN
1180   argc = 1
1190   argv$(0) = "rawkbd:"
1200 ENDIF
1210
1220 out = OPENOUTdest$
1230
1240 FOR i% = 0 TO argc-1
1250   in = OPENINargv$(i%)
1260   IF in = 0 CLOSE#out:ERROR 99,"Filecat: file "+argv$(i%)+"not found"
```

```
1270   WHILE NOT EOFin : BPUT#out,BGET#in : ENDWHILE
1280   CLOSE#in
1290 NEXT i%
1300
1310 CLOSE#out
1320 END 1
```

The next example uses PROCShell to call an application from BASIC. In particular, it calls the Fortran 77 compiler front-end, and optionally the code generator. It also interacts with the user to obtain certain parameters. It is a good example of a BASIC program which uses PROCShell to provide a more powerful command script than would be available using a simple *EXEC file (which, for example, couldn't use an interactive user input).

```
1000 REM  > $.Library.F77
1010 LIBRARY"$.TEMP.SHELL_LIB"
1020 PROCAssemble_Shell
1030 IFFNShell_String_UC(Shell_Env$)="-HELP"THEN
1040 PRINT"F77 command command 1.00: -help"'
1050 PRINT"Use: f77 <program name> [-opt <options>]"
1060 PRINT"Program will accept just f77 and prompt for parameters."
1070 END
1080 ENDIF
1090 A$=Shell_Env$
1100 IFINSTR(A$," ") A$=LEFT$(A$,INSTR(A$," ")-1)
1110 B%=INSTR(FNShell_String_UC(Shell_Env$),"-OPT")
1120 B$=""
1130 IFB%>2 B$=MID$(Shell_Env$,B%+5)
1140 IFA$="" THEN INPUT"Program source file name, options: "A$,B$
1150 PROCShell("f77fe src."+A$" -to fcode."+A$+" -opt +"B$)
1160 PRINT"Run the cg? [y/Y]:";
1170 R$=GET$
1180 PRINTR$
1190 IFR$<>"Y"IFR$<>"y" END
1200 INPUT"New map size: "A
1210 IFA=0 A+40
1220 PROCShell("f77cg fcode."+A$+" -to aof."+A$+" -opt m"+STR$A+"+"+B$)
1230 PRINT"link it? [y/Y]:";
```

```
1240 R$=GET$
1250 PRINTR$
1260 IFR$<>"Y"IFR$<>"y" END
1270 PROCShell("link aof."+A$+ -library $.arm.fortran.xlib_f77 -image
"+A$+" -adfs")
1280 PRINT"run it? [y/Y]:";
1290 R$=GET$
1300 PRINTR$
1310 IFR$<>"Y"IFR$<>"y" END
1320 PROCShell (A$)
```

This chapter describes a utility which will not be used frequently, but is very useful when it is required. Memtest checks the application workspace (from `&8000` to HIMEM) in a variety of ways. The checks performed detect memory errors due to faults such as: address lines stuck at '1' or '0', address lines shorted, and data lines stuck at one level or shorted together.

Memtest actually performs four separate tests on memory, designed to detect different types of memory fault. These are as follows.

Phase one: incrementing pattern

In this test, the application memory is filled with a pattern of words, such that the content of each word is equal to that of the previous word plus a fixed number. Once filled, the memory is checked to ensure that the pattern read back is the same as the one stored there initially.

This test is performed four times, with different start values and increments. It is likely to fail if there are any problems on the data bus, eg data lines stuck or shorted, or faulty memory locations.

Phase two: TRUE hierarchy

This test gives the address lines of the machine a good work-out. It works as follows. There are several passes 1, 2, 3... 31. For each pass n, the application space is filled such that if address line n is high for the word being addressed, the value `&FFFFFFFF` is stored there, otherwise 0 is stored. As above, once the pattern has been written, it is checked for validity.

Because memory is filled a word at a time in this test, address lines A0 and A1 are always zero. Thus on the first pass (testing A1), a zero is written to every location. On the second pass (testing A2), alternate words are set to 0 (even word addresses) and `&FFFFFFFF` (odd words). On the third pass, the pattern is 0, 0, `&FFFFFFFF`, `&FFFFFFFF` and so on.

Phase three: FALSE hierarchy

This is exactly the same as phase two, except that the opposite values are stored in the memory locations. Thus if the address line for pass n is high, a 0 is stored in the word; if it is low, `&FFFFFFFF` is stored.

Phase four: Cycling bits

This phase is another exhaustive test of the data bus. It works by storing a pattern of words, each with a single bit set in memory. Each word has bit (n+1) MOD 32 set, where n is the bit that was set in the previous word.

There are 32 passes. On the first pass, the first word to be tested has bit 0 set, the second word has bit 1 set, and so on. The cycle repeats, so that the 32nd word also has bit 0 set, the 33rd word has bit 1 set, and so on. On the second pass, the first word has bit 1 set, the second one has bit 2 set. On the last pass, the first word has bit 31 set, the second word has bit 0 set. This is summarised in the diagram below:

```
Addr   Pass 1       Pass 2                 Pass 31      Pass 32
00     &00000001    &00000002     .....    &40000000    &80000000
04     &00000002    &00000004     .....    &80000000    &00000001
08     &00000004    &00000008              &00000001    &00000002
0C     &00000008    &00000010              &00000002    &00000004
...

1C     &80000000    &00000001              &20000000    &40000000
20     &00000001    &00000002              &40000000    &80000000
24     &00000002    &00000004              &80000000    &00000001
28     &00000004    &00000008              &00000001    &00000002
```

ERROR MESSAGES

If an error is detected during any of the phases of the memory test, a message of the following form is printed:

```
Phase 1 fail at &xxxxxxxx with &xxxxxxxx instead of &xxxxxxx
```

where xxxxxxxx stands for a 32-bit hexadecimal number.

If no errors were detected, the message:

```
PASSED.......
Press SPACE to continue.
```

is printed. When you press space, control returned to the OS, and the VDU is disabled.

This appendix briefly discusses the three file formats mentioned in this manual. They are Acorn Object Format (AOF), Acorn Image Format (AIF) and Acorn Library Format (ALF). This data is given for information only. It is not detailed enough for, say, a compiler writer to be able to produce a correctly formatted AOF file. If you need this level of detailed information, contact Acorn directly.

CHUNK FILES

AOF and library files are held in a format known as 'chunk file format'. Chunk files allow a single file to hold multiple objects which can be treated as separate entities. At the start of a chunk file is a header. This identifies the file as being in chunk file format, and contains information about the rest of the file.

The first word of a chunk file is the special value `&C3CBC6C5`. (Or, if it's easier to remember, the string 'EFKC' with the top bits set.) Unless a file starts with these four bytes, it is not a chunk file.

The next two words in the file give the maximum number of chunks it may contain (maxChunks) and the current number of chunks (numChunks), respectively. MaxChunks is fixed when the file is created; numChunks can vary throughout the life of the file.

The next 4*maxChunks words are entries for each chunk. An entry comprises a chunk id (two words), a file offset for the chunk (one word, a multiple of four bytes), and the length of the chunk (one word). The format of the chunk id is a four character type name followed by a four character component name. Examples of type names are 'OBJ_' for AOF files and 'LIB_' for library files. Examples of component names are 'HEAD' and 'SYMT', which are both component chunks of an AOF file.

If the file offset for a chunk is zero, then that entry is not being used. There should be maxChunks - numChunks entries for which this is true.

Immediately following the chunk entries are the chunks themselves. These are of arbitrary format, defined independently of chunk files. The contents of the chunks in library and AOF files are described below.

AOF FILES

AOF files are produced by compilers and also by assemblers when assembling a module for linking into another program. An AOF file is a chunk file.

AOF files are defined to have a least five chunks. The names of these chunks are stored in the chunk file header. They are:

OBJ_HEAD	The header
OBJ_AREA	The areas information
OBJ_IDFN	The indentification part
OBJ_SYMT	The symbol table
OBJ_STRT	The string table

The prefix OBJ_ identifies the chunks as being part of an AOF file. The next four characters identify the particular sections of the AOF file which are present in most AOF files. In fact, only the header and areas chunks are compulsory, and there may be others (eg debugging information) for use by special tools.

The header

The header contains information about the rest of the file. It is in two parts; the first six words are fixed and are always present. Following the fixed part is a variable-length part which describes the contents of the areas chunk.

The fixed part of the header contains: the object file type, the version of the object format, the number of areas, the number of symbols, and entry address information.

The areas information contains one entry for each area in the areas chunk. Typically there are two areas, one for data and one for code. The area information entry contains: the name of the area, the alignment of the area, its attributes (whether it's code or data, absolute or position independent, etc), the size of the area, the number of relocations it contains, and the base address for the area (for absolute areas).

Area name strings are held in the string table chunk, OBJ_STRT, as is all textual information in the AOF file. Strings are referred to by their offsets from the start of the string table. Typical names for areas produced by a compiler (the C compiler, in fact) are: C$$code, C$$data and C$$debug. The last refers to an area holding debugging information for the ASD program.

The areas chunk

This chunk contains the code or data referred to in the header chunk. It consists of one or more areas, each followed by its associated relocation information. Relocation information is present to allow the area to be moved from the absolute area for which it was compiled, when it is being linked with other files. It consists of an offset into the area of the data to be relocated, a symbol to be used to in the relocation (if required), the size of item to be relocated (byte, half-word or word), and the method to use to perform the relocation.

The Linker uses the relocation table to 'patch' the area so that the data or instructions it contains are consistent with the position it occupies in the final linked ouput. It might also use the symbol associated with the relocation to insert the address of an external routine or data object into the file.

The symbol table chunk

This chunk contains information about the symbols referenced in the relocation information. Additionally, symbols which are being exported from an area, eg library routines, are defined here. The number of entries is given in the header chunk.

Each symbol entry includes a reference to: the symbol name, its attributes (local/global, constant/offset, definition/reference etc.), its value and the area with which it is associated if its value is an offset from an area base address.

As with all AOF file strings, the name is actually stored as an offset into the string table chunk. This enables the name to be stored as a fixed length (one word) item in the symbol table entry.

The string table chunk

This chunk contains the textual names of all of the identifiers in the file. In particular the area names are stored here, as are the names of the symbols in the symbol table chunk. An entry in the string table consists of a word-aligned text string, terminated by a zero byte.

> Note: identifiers in the symbolic debugger information area produced by compilers have their text stored in the debug area itself, not in the string table chunk.

The identification chunk

This chunk contains a text string, terminated by a zero byte, identifying the language and version number which produced the AOF file. This chunk is not required.

AIF FILES

An image format file is produced by the Linker as a result of resolving the external references in one or more object files and libraries. Although an AIF file is not a chunk file, it is divided into several parts. These are described briefly below:

Header

The header is 32 words (128 bytes) long. The first four words are branch instructions into the following pieces of code:

```
000000      BL decompressCode
000004      BL selfRelocCode
000008      BL zeroInitCode
00000C      BL imageEntryPoint
```

The addresses on the left are offsets from the start of the file. Only the last branch, to the start of the program proper is required. The other three may be replaced by no-op instructions (BLNV) if the relevant routine is not present.

An AIF file is always entered at its first location, so before the program starts to execute, the following three things may happen. First, the file is 'decompressed'. This applies to files which have been processed by the Squeeze program. This program compacts the image by encoding four-byte words into one or two-byte versions, and adds a translation table and code to perform the decompression to the end of the file. Squeeze then sets the first instruction to branch to the decompression code.

Once the program has been expanded, it may relocate itself. This branch is filled in by the Linker when it is given the `-relocate` option. It appends a table of relocation offsets and a small routine to perform the relocation on to the end of the image. When this is called, it uses the load address (accessible through the link register R14), the address the image was linked for (available in the header) and the relocation table to patch up the words which contain position dependent offsets.

Note that this relocation is less versatile than the static relocation performed by the Linker because it can only deal with whole-word quantities and symbol-relative relocation is not possible. However, because of the position-independent manner in which most data accesses are performed in ARM code, even large programs require only a few relocation table entries (eg less than 50 for a 64K program), so the overhead of using the `-r` link option is quite small.

Next, the zero initialise code is called. This is used to store zeros after the read/write area (see below) which should be so initialised before the program runs. The code to perform this and the parameters necessary to initialise the correct area are also stored in the AIF header. Note that not all AIF files will use this entry; some have the data area already initialised on loading.

Finally, once the other three routines have executed (or been have skipped) the branch to the start of the program is made.

> Note: The description of the AIF format given above and completed below refers to the final image in memory, after the decompression and relocation routines have executed. The image on disc (and when

first loaded) will be substantially different before these routines have run.

The rest of the header

Immediately after the four branches, the header contains the following information:

SWI Exit	Just in case the main program returns
Read-only size	Number of bytes in program area, inc. the header
Read/write size	Number of bytes in the data area
Debug size	Number of bytes in the optional debug area
Zero-init size	Number of bytes in the zero-intialised area
Debug type	0 for none, 1 for Dbug, 2 for ASD
Image base	Base address used during linking
Reserved	Five 0 words
Zero-init code	16 words long

The other areas

Following the header are the read-only (program code) and read/write (static data) areas. The header is counted as part of the read-only code. The read/write area includes the zero-intialised area (if present), whose size is given in the header. You should note that if this is used, the area of memory occupied by the program may be considerably larger than its length on the disc implies. This is also true of compressed files.

If there is any Arthur Symbolic Debugger information in the image, it is stored at the start of the zero-initialised area. Therefore the debugger has to move it before the zero-initialise code is executed.

ALF FILES

Library files are chunk files. There are at least three chunks, and may be many more. The two compulsory chunks are identified as:

LIB_TIME	Time and date of last library modification
LIB_VRSN	Library file format version number
LIB_DIRY	Directory chunk

The first chunk is eight bytes long and holds an encoded form of the time and date the library file last modified (by Libfile, for example). The second chunk is four bytes long and contains the library format version number in binary. This is currently 1.

The third chunk is a directory of all the AOF files which make up the library. Each file has its name, length and date/time-stamp stored.

In addition to these two chunks, there is one chunk for every AOF file in the library. These have chunk identifiers:

LIB_DATA

The format contents of an AOF chunk is exactly the same as a complete AOF file. Thus you can regard a library as a chunk file of chunk files.

Finally, the library may contain two chunks created by Objlib command. These are called:

OFL_SYMT	External symbol table chunk
OFL_TIME	Symbol table date/time

The first one contains a list of the external symbols which are defined in the various modules that make up the library. These can be viewed using the Objlib `-l` option. The second chunk contains the time and date when the table was last updated. This allows Objlib to check that the table is not out of date with respect to the rest of the library.

INTRODUCTION

In this appendix, we describe the ARM Procedure Calling Standard. This standard has been laid down for the benefit of software developers who want to be able to call 'external' routines. Example uses are assembler programs which use routines from the common library and C language programs which call hand-written assembler routines.

The standard covers several areas - register allocation, stack discipline and backtracing, procedure entry and exit conditions etc. What it doesn't define is data representation. Certain data types have 'obvious' representation on the ARM. Integers, for example, are most usefully one-word quantities. However, the Standard only concerns itself with how a given number of one-word parameters are passed to a routine, and how the result (if any) is returned. You should consult the manual for each language to see how they use these rules to implement their required parameter-passing mechanisms. The ANSI C language should be regarded as the model for both data-type representation and implementation of the Standard. It is therefore used in some of the examples given in this appendix.

Note that the standard only refers to *external* calls. Procedure calls within a language may use any scheme that is convenient; it is only at the point of call and return when interfacing to an external procedure that a language has to obey the rules described here.

REGISTER ALLOCATION

The 16 ARM registers and eight floating point registers are allocated specific names and uses under the Standard. Some of these uses are optional, for example registers used for stack backtracing and stack overflow checking may be 'undedicated' if those facilities are not required. However, both these features are supported by the common library and by the code generated by the C compiler. It is therefore recommended that all programs conform.

The table below gives the symbolic names assigned to the registers, with a brief description of the register's use.

R0	a1	Argument 1/integer result
R1	a2	Argument 2
R2	a3	Argument 3
R3	a4	Argument 4
R4	v1	Register variable
R5	v2	Register variable
R6	v3	Register variable
R7	v4	Register variable
R8	v5	Register variable
R9	v6	Register variable
R10	fp	Frame pointer. Used for stack backtrace
R11	ip	Used as temporary workspace
R12	sp	Stack pointer to full, descending stack
R13	sl	Stack limit. Used for stack overflow checking
R14	lr	Link register. Used for procedure return
R15	pc	Program counter
F0	f0	Floating point result
F1	f1	Floating point work register
F2	f2	Floating point work register
F3	f3	Floating point work register
F4	f4	Floating point register variable
F5	f5	Floating point register variable
F6	f6	Floating point register variable
F7	f7	Floating point register variable

PROCEDURE ENTRY

In this section we describe the exact contents of the registers at the point when control is passed to a procedure, ie immediately after a `BL` instruction to the procedure's entry point. We also describe typical entry code to perform stack checking and argument stacking.

Arguments

Arguments are passed in the usual C way of pushing them in reverse order on a descending stack. This ensures that they appear in ascending order in memory when read left to right. (Note though that this doesn't imply anything about the order in which the arguments are evaluated, only that they are pushed right to left.)

For efficiency, the first four words of the arguments are passed in the argument registers `a1-a4`. Access to registers is much quicker than access to memory locations, and as operands in ARM instructions have to be in registers anyway, it makes sense to ensure that they are already in place.

Suppose a procedure is called with two integer arguments, as in:

```
proc(i1,i2);
```

where `i1` and `i2` are available in registers. The code to call `proc` looks, in theory, like this:

```
STMFD    sp!,{i2}      ;Push arg2
STMFD    sp!,{i1}      ;Push arg1
LDMFD    sp!,{a1-a2}   ;Load up to first four words in a1-a4
BL       proc          ;Call routine
```

Of course, in practice a decent compiler (or hand coder) would load the arguments into `a1` and `a2` directly from the other registers.

The theoretical pushing and pulling occurs in practice when multi-word arguments are passed. Consider passing three floating point expressions (eg C `doubles`). These occupy two ARM words each. A typical calling sequence would be:

```
...                         ;Obtain third arg in f0
STFD    f0,[sp, #-8]!       ;Save it on the FD stack
...                         ;Obtain second arg in f0
STFD    f0,[sp, #-8]!       ;Save it
...                         ;Obtain first arg in f0
STFD    f0,[sp, #-8]!       ;Save it
LDMFD   sp!,{a1-a4}         ;Load first four words in a1-a4
BL      proc                ;Call routine
ADD     sp,sp,#8            ;Discard the remaining two words
```

At the point when the `proc` is called, `a1-a4` hold the four words which constitute the first two parameters, and the last parameter is at the top of the stack. It is very likely that the called routine will immediately save `a1-a4` back on to the stack so that the arguments can be transferred into FP registers. It might seem more sensible to pass floating point arguments in the FP registers in the first place. Unfortunately, this does not tie in very well with the C way of doing things. Anyway, you could imagine cases where the called routine could use the contents of the argument registers directly (for example, where they contain the first four words of a structure type).

Statistically, procedures are very likely to have four or fewer arguments, so in effect parameters usually passed in registers under the Standard. If a procedure is passed, say, two one-word parameters, `a3` and `a4` will be undefined on entry and the stack will not hold any further parameter words.

Register variables

The registers named `v1-v6` and `f4-f7` above are allocated for register variables. On entry to a procedure, these registers are not defined to contain any particular values, but must be preserved. Thus if a procedure uses `v1` and `v2` as working registers, it must store their values on the stack before they are altered for the first time, and load them back after they have been used for

the last time. It is usual for these actions to occur at the procedure entry and exit points respectively.

The stacking of register variables is usually combined with the creation of the stack backtrace structure into a single `STM` instruction. This is illustrated below when the backtrace structure is described.

Frame pointer

This register is used to point to a 'backtrace structure', or contains 0 if that mechanism is not supported. The backtrace structure is effectively the values of some stacked registers, including the PC. The FP register itself points to the stacked copy of the PC. Through this, the backtrace software can access information relating to what other registers are stacked. This information is used, for example, by the default C signal handler, and the Dbug program. We describe the backtrace structure in more detail below.

C also uses `fp` as the base from which to access stacked arguments when transferring them to and from registers.

IP register

This is undefined on procedure entry. It is used as temporary workspace. In particular, it is used to hold the entry value of the stack pointer while the backtrace structure is pushed on to the stack, so that the SP value appears in the correct place in that structure. See below for details. Other than that, it is free for use within the procedure to hold temporary results etc.

Stack pointer

On entry to a procedure, this register points to the most recently pushed word of a full, descending stack. The stack is used to hold arguments, local variables and the backtrace information.

As noted above, up to four argument words are passed in registers. The remaining words are located on the stack, with `sp` holding the address of the fifth word.

It is the responsibilty of the calling code to remove any arguments that it pushed on the stack by adjusting `sp`. This is illustrated by the second example in the section *Arguments* above.

Stack limit

The `sl` register contains a value which the stack pointer should not descend below. In fact `sl` contains an address 512 bytes above the lower end of the available stack.At the moment of procedure entry, `sp+256>=sl`. That is, at least 256 bytes are available on the stack, which should be enough for the procedure to check for stack overflow and allocate a new segment if necessary.

The way to check for stack on entry to a procedure overflow is as follows:

```
CMP       sp,sl
BLLT      |x$stack_overflow|
```

If the comparison fails, `sp>=sl` and there are at least 512 bytes available on the stack. This means that as long as the procedure uses 256 or fewer bytes of stack, it can call any other procedure without further checking.

If the test succeeds, that is `sp<sl`, there are between 256 and 508 bytes left on the stack, and the call to the stack overflow routine `x$stack_overflow` is required to allocate a new stack segment. This is the name of the common library routine to perform this action; other systems may use other routines. If `x$stack_overflow` can't allocate a new stack segment, a C 'signal' is raised.

Procedures which use more than 256 bytes of stack must perform slightly more checking. In particular, they must compare `sl` with the value of `sp` minus the amount of space used, eg:

```
SUB    ip,sp,#256          ;Uses 256 bytes of stack workspace
CMP    ip,sl
BLLT   |x$stack_overflow|
```

This shows another use of the `ip` register.

Link register

When a procedure is entered, `lr` hold the return address and status register. Its contents are restored into R15 (the PC) in order to effect the return to the caller. Note that both the PC *and* flags part must be transferred; a procedure call is defined to preserve the status register.

As the `lr` is saved as part of the backtrace structure on the stack, procedure return is usually combined with the restoration of the other saved registers, using an instruction such as:

```
LDMDB       fp,{fp,sp,pc}^
```

The reasons for using `fp` as the base register in a 'decrement before' instruction are discussed in the section *The stack backtrace structure* below. Note the `^` to ensure that the flags part of R15 is also loaded.

Program counter

R15, the PC, is, of course, wholly dedicated on the ARM and it always fulfills the same function. Bits 2..23 contain the word address of the next instruction to be fetched; bits 0..1 and 24..31 contain the processor mode, interrupt mask and status flags.

On execution of the first instruction of a procedure it contains an address eight greater than that of the instruction (due to pipelining), and the flags part is the same as the return flags in `lr`. Note that it is not sufficient for a procedure to preserve the flags in R15 at entry; it must actually restore them from the contents of `lr`.

PROCEDURE EXIT

To return to the caller, a procedure must load any result in `a1` or `f0` as appropriate, then restore various registers. In particular, `fp`, `sp`, `sl`, `v1-v6` and `f4-f7` must be restored to their entry values. The return link which was in `lr` on entry should be placed in the PC.

All other registers may contain any value. In particular, `ip`, `lr`, `a2-a4` and `f1-f3` are not defined on return.

THE STACK BACKTRACE STRUCTURE

We have mentioned this several times already, and describe it in detail in this section. The structure is simply a set of four or more registers saved on the stack, with `fp` pointing at the one occupying the highest location. (If there is no backtrace structure, `fp` contains 0.) The stack looks likes this:

```
fp - 0->     | save mask pointer (pc) |
fp - 4->     | return link value (lr) |
fp - 8->     | return sp value   (ip) |
fp -12->     | return fp value   (fp) |
           [| saved v6 value    (v6) |]
           [| saved v5 value    (v5) |]
           [| saved v4 value    (v4) |]
           [| saved v3 value    (v3) |]
           [| saved v2 value    (v2) |]
           [| saved v1 value    (v1) |]
           [| saved a4 value    (a4) |]
           [| saved a3 value    (a3) |]
           [| saved a2 value    (a2) |]
           [| saved a1 value    (a1) |]
           [| saved f7 value    (f7) |] three words
           [| saved f6 value    (f6) |] three words
           [| saved f5 value    (f5) |] three words
           [| saved f4 value    (f4) |] three words
```

Stack entries in square brackets are the optional ones. The first four words are always saved. The register names in brackets refer to the register which was pushed to make that entry. Note that the return `sp` value is saved by pushing `ip`.

The *save mask pointer* pointed to by `fp`, when ANDed with 0x03fffffc holds the address plus 12 of a word called the *return data save instruction*. This is the `STM` instruction which saved all of the registers onto the stack in the first place. It has the form:

```
STMFD      sp!,{[a1], ...,[a4],[v1], ...,[v6],fp,ip,lr,pc}
```

Again square brackets denote optional registers.

The value of `fp` pushed points to a similar backtrace structure (or is 0), and so on.

The value of `ip` pushed contains the value of `sp` on entry, so that `sp` can be restored in the same instruction as the rest of the registers.

The value of `lr` pushed holds the return address and flags.

The value of `pc` pushed is 12 bytes beyond the `STMFD` itself, due to pipelining. The status bits are also written, which is why the word at `[fp]` has to be masked before it can be used to ascertain the address of the instruction.

The `STM` is followed immediately by between zero and four `STFE` instructions. These perform the optional pushes of `f4-f7`. The instructions must be of the form and in the order shown below:

```
STFE     f7, [sp, #-12]   ;11101101 01101100 01110001 00001100
STFE     f6, [sp, #-12]   ;11101101 01101100 01100001 00001100
STFE     f5, [sp, #-12]   ;11101101 01101100 01010001 00001100
STFE     f4, [sp, #-12]   ;11101101 01101100 01000001 00001100
```

Any deviation from this ordering terminates the sequence. By examining the the `STM` and instructions immediately after it, the backtrace code can determine which registers have been stacked and print their values accordingly.

Having seen the conditions at entry and exit of a procedure, and the form of the stack backtrace structure, we can give some examples of typical entry and exit code sequences. Note that these aren't mandatory. Any sequence that produces the desired effects is allowed.

Take the simplest case of a function which uses no (preserved) registers or additional stack and which returns an integer result. This could use the following code:

```
MOV         ip,sp                   ;Save entry sp in ip
STMFD       sp!,{fp, ip, lr, pc}    ;Save compulsory backtrace
stuff
SUB         fp, ip, #4              ;fp points at saved pc
...
ADD         a1,a2,a3 LSL #2         ;Form a result in a1
LDMDB       fp, {fp, sp, pc}^       ;Restore and return
```

You can see now why an `LDMDB` is used to return. Since `fp`, which is used as the base register, points at the saved `pc` value, the registers pulled are the three below that on the stack (as the instruction Decrements Before pulling the first one). Thus the PC and flags are loaded from the saved `lr`, the `sp` is loaded from the saved `ip` (which was the entry `sp`) and `fp` is loaded from the entry `fp`.

Now consider the case where it is desirable to have all of the parameters on the stack as, the procedure assumes that that they occupy contiguous locations in memory (as C's `printf` does). Also, say the routine uses the stack for workspace and uses `v1` and `v2`. Its entry and exit code might look like this:

```
MOV     ip,sp                       ;Save entry sp
STMFD   sp!,{a1-a4}                 ;Make args. contiguous on stack
STMFD   sp!,{v1,v2,fp,ip,lr,pc}     ;Save backtrack info and reg. vars.
SUB     fp,ip,#20                   ;fp points at saved pc
CMP     sp,sl                       ;Check for stack space
BLLT    |x$stack_overflow|          ;Deal with overflow
...
```

```
LDMDB   fp,{v1,v2,fp,ip,lr,pc}^   ;Restore and return
```

Note the calculation to derive the new value of `fp` had to take into account the four words pushed before the backtrace structure was set-up.