ARM utilities

reference manual

ARM Evaluation System

Acorn OEM Products

ARM utilities

Part No 0448,005
Issue No 1.0
1 August 1986

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

- The manual is provided on an 'as is' basis except for warranties described in the software licence agreement if provided.
- The software and this manual are protected by Trade secret and Copyright laws.

The product described in this manual is subject to continuous developments and improvements. All particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

There are no warranties implied or expressed including but not limited to implied warranties or merchantability or fitness for purpose and all such warranties are expressly and specifically disclaimed.

In case of difficulty please contact your supplier. Every step is taken to ensure that the quality of software and documentation is as high as possible. However, it should be noted that software cannot be written to be completely free of errors. To help Acorn rectify future versions, suspected deficiencies in software and documentation, unless notified otherwise, should be notified in writing to the following address:

Customer Services Department,
Acorn Computers Limited,
645 Newmarket Road,
Cambridge
CB5 8PD

ISBN 1 85250 010

Published by:
Acorn Computers Limited, Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN, UK

Contents

1. Introduction

This document is a reference guide to the ARM firmware (the Executive ROM) and to the various utility programs supplied on disc.

Chapter 2 deals with the Executive ROM, and includes sections on the routines contained within the supervisor, the use of the SWI calls, and the organisation of memory. Chapter 3 explains the structure of Acorn Object Format files. The remaining chapters deal with the utility software: the linker in chapter 4, the debug utility in chapter 5, the floating point emulator in chapter 6 and a group of miscellaneous utilities in chapter 7.

1.1 Conventions used in this manual

Most ARM source code has its own interpretations of the punctuation symbols and special symbols which are available from the keyboard. These are:

```
! " # $ % & ^ @
( [ { } ] ) | : . , ;
+ - / * = < > ? _
```

This often makes it difficult for the user to determine precisely which characters on the printed page are explanatory or descriptive, and which (if any) are the ones which AAsm will accept as having the correct syntax. A typewriter-style typeface has been used to indicate both text which appears on the screen and text which can be typed on the keyboard (for example, AAsm source code). This is so that the position of relevant spaces is clearly indicated.

Both general and specific examples of syntax and screen output is given – there are occasions where the full syntax of an instruction and its accompanying screen appearance would obscure the specific points being made. It follows therefore that not all the examples given in the text can be used directly since they are incomplete.

Curly brackets { } enclose optional items in the syntax. For example, AAsm accepts a three field source line which may be expressed in the form:

Function keys (such as f1) and control keys (such as tab) often need to be pressed by themselves or in combination with the shift and ctrl keys. To indicate this these keys are printed in boxes. For example:

Press the RETURN key [RETURN]

Press the ESCAPE key [ESCAPE]

Press the DELETE key [DELETE]

Press the COPY key [COPY]

2. Operating system firmware

2.1 Introduction to the Executive ROM

This chapter describes a set of supervisor calls linking ARM machine code to the Acorn 6502 MOS as used in the BBC Microcomputer. For further information on Acorn MOS refer to the User Guide for the BBC Microcomputer, the DFS or ADFS manual and the Advanced User Guide. Programs should assume the use of ADFS or ANFS if they wish to do something of file system specific nature. Refer to the *ARM CPU Software Manual* for details of the machine instruction set.

The ARM Second Processor consists of an ARM CPU, memory, timing circuitry and Acorn's Tube interface to the BBC Microcomputer. The BBC Microcomputer serves as an IO processor; it handles all the input and output devices (keyboard, rs423, text and graphics displays, printer, disc drives and so on). The BBC Microcomputer requires additional software from the DNFS ROM to deal with the Tube. The ARM Second Processor contains a ROM called the Executive kernal which deals with the ARM end of the Tube. The ROM also contains a set of useful utilities in the supervisor.

To start the system, switch on both the BBC Microcomputer and the ARM Second Processor. For the BBC Microcomputer to recognise the Second Processor it must be reset or switched on after the Second Processor has been switched on. The following message, or something similar, should appear on your screen:

```
Acorn ARM Second Processor 4096K

Acorn ADFS

A*
```

The 'A*' will appear either as A* on a blue background in mode 7 or as a large legend in any of the other modes: it is the prompt from the supervisor.

2.2 The supervisor

When nothing else is using the system, the supervisor gives its A* prompt and its built-in commands can be used: anything it does not understand will be passed to the Acorn MOS CLI. The built in commands are:

- `BreakClr`
  Removes all breakpoints or just the one at the specified address. Puts the original contents back into the location.
- `BreakList`
  Lists the currently set breakpoints.
- `BreakSet`
  Set a breakpoint at an address.
- `Buff`
  Turn file buffering on
- `Continue`
  Start execution from breakpoint saved state. If there is a breakpoint at the continuation position, then a prompt will be given: reply Y if it is permissible to execute the instruction at a different address (that is, it does not refer to the pc).
- `Help`
  Generates reassuring message from supervisor: gives version number of the kernal (this manual refers to kernel version -.008). `help supervisor` gives the list of commands that the supervisor accepts.
- `InitStore`
  Fills all memory with `&EE000000` or the specified data.
- `Memory`
  Displays memory in ARM words from the address or register given to the next address or register given, a + meaning added to the first address. Default second address means 256 bytes displayed.
- `MemoryA`
  Display and alter memory in bytes or words, signalled by an optional B or b. Given just an address/register it enters an interactive mode: RETURN to go to next location, - to go back one location, hex digits to alter a location and proceed, anything else to exit. Alternatively give the data required on the command line as well.

- `MemoryI`
  Disassemble ARM instructions. Syntax as for memory. Given a limit it proceeds to the limit, otherwise it disassembles 24 instructions and waits for a key.

- `NoBuff`
  Turns file buffering off

- `Quit`
  Leave the supervisor by performing an `SWI Exit`. Any other supervisor will be returned to - or, of course, the supervisor itself.

- `ShowRegs`
  Displays the registers caught on one of the four traps (unknown instruction, address exception, data abort, address abort)

- `Transfer`
  Copies files from one file area to another.The syntax is `transfer` *argument*. Examples of arguments are:

`disc adfs fred`	copy fred from disc to adfs
`net adfs fred jim`	copy fred from net to adfs as jim
`disc adfs *`	all files in current disc directory
`disc adfs`	prompt for file names to be typed
`dir@$.1 dir@$.2 *`	all files in $.1 to $.2
`net`dir@&.1 adfs *`	all files in net directory &.1 to adfs

  The first two fields of the argument are simply interpreted by the command line interpreter or CLI, so they can cause quite a wide range of effects. The character @ in the first two fields of the argument will be replaced by a space character, the character ` in the first two fields of the argument represents a newline (multiple CLI commands) thus one can use transfer between net fileservers, `transfer fs@1.254 fs@0.126 *`. Transfer has special code in it, so that, although it changes file systems, this does not cause the exec file to be lost.

- `Go`
  Enter program at the address given. No address corresponds to `&1000`; R0 to R15 corresponds to the contents of the registers dumped on a trap. After the address a ; should precede the environment string. GO is, in fact, implemented at the Executive kernal level, so it still works from inside other systems. GOS enters the supervisor: the caller can be returned to if an Exit handler has been set using the Quit command.

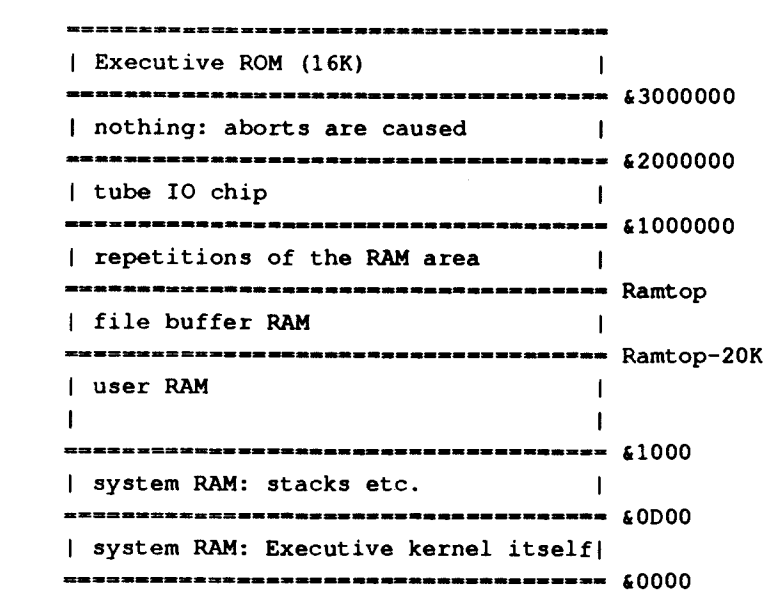
Star commands, for example `*CAT` may be used by typing them directly to the A* prompt. It is not necessary to type the `*`. Refer to the MOS, DFS, ADFS documentation for details of a particular ROM facility.

Programs from the current filing system are executed simply by typing their names. Refer to documentation on the particular program for the parameters it may require. Usually programs will respond to the parameter `-help`.

The A* prompt is created by redefining character 255. Spooling is disabled while this prompt is output using fx3 calls. The character 255 is left as a solid block. With the ARM Second Processor connected the entire character set has been exploded, see description of `fx20` in the User Guide.

With the Second Processor connected addresses are used to distinguish between the memories of the two machines. The ARM Second Processor can use addresses up to Ramtop (see section 3). The BBC Microcomputer uses addresses FFFxxxxx. The DFS filing system only has 18-bit addressing; any address greater than 2FFFF is assumed to be in the BBC Microcomputer: use of ADFS or the net filing systems avoids any problems of restricted addressing.

2.3 ARM Second Processor memory map



Programs should load in memory at &1000 or higher. If a program is to be run at an address outside available memory, the Executive will try wrapping it around in available memory: the TWIN program is supplied to run at &1D0000: in a 1/4 Mbyte machine it thinks it has been run at &10000, in a 1 Mbyte machine it thinks it has been run at &D0000. For a 4 Mbyte machine it could be moved to &3D0000, however, due to the hardware design of the 2 and 4 Mbyte machines this would cause it not to work on the 2 Mbyte machine, although it would be fine with the full 4 Mbytes. Programs should only use memory up to the Ramtop limit returned by the GetEnv call, on an otherwise empty machine the Excutive kernel uses the top 20K for file buffers.

2.4 Executive kernel

All BBC Microcomputer IO is accessed through SWI calls.

The following information documents all the Executive kernel calls that a program can make using the SWI instruction. A, X, Y represent the 8 bit 6502 registers referred to in the Acorn MOS documentation, C the 6502 carry flag and the ARM carry flag. R0 and so on are ARM registers. (string) is an indirect pointer to a string terminated by hex &00, &0A or &0D. R0b means lsb of R0 (only bottom 8 bits used as an input parameter, top 24 bits zeroed on result). R0 means all 32 bits of R0. All successful SWIs will clear the ARM V flag.

2.4.1 WriteC &00 (Acorn MOS OSWRCH)

Write R0b to the terminal output. For example:

```
MOV R0,#"H"
        SWI WriteC

In: R0b
Out:
```

2.4.2 WriteS &01

Write the bytes following the call to the terminal output. Terminates at first zero and starts execution at the next 32bit word. For example:

```
SWI WriteS
        = "Hello World",10,13,0
        ALIGN
        SWI WriteI+"K"

In:
Out:
```

2.4.3 Write0 &02

Write the bytes pointed to by register 0 to the terminal output. Terminates at first zero. R0 points to the byte after the zero on exit. For example:

```
ADD R0,PC,#data-.-8
          SWI Write0

In: R0
Out: R0 updated
```

2.4.4 NewLine &03 (Acorn MOS OSNEWL)

Write a line feed (&0A) followed by a carriage return (&0D) to the terminal output. For example:

```
SWI NewLine

In:
Out:
```

2.4.5 ReadC &04 (Acorn MOS OSRDCH)

Read a character and validity from the terminal input. C set if an unusual character (for example Escape) is read. For example:

```
SWI ReadC
          BCS Escape

In:
Out: R0 contains character; C contains validity
```

2.4.6 CLI &05 (Acorn MOS OSCLI)

Interpret a string as a command. The string is checked for HELP (or valid abbreviations) and an appropriate message issued. An additional command over those in the IO processor MOS is the GO command, see the data about the supervisor. For example:

```
SWI CLI

In: R0 pointing to string
Out: may not return if another
     program has been executed
```

2.4.7 Byte &06 (Acorn MOS OSBYTE)

Do an Acorn MOS OSBYTE call with A=R0b, X=R1b, Y=R2b, C=C. For calls with R0b less than 128 the R2 register is not required or altered. For example:

```
MOV R0,#5
        MOV R1,#4
        SWI Byte ;select network printer

In: R0, R1, R2
Out: R0, R1, R2, C
```

Note: because of the read only file buffering in the Executive the OSBYTE &7F call checks R1 WORD for being greater than 256 (range of fast handles is 257 to 511).

2.4.8 Word &07 (Acorn MOS OSWORD)

Do an Acorn MOS OSWORD call with A=R0b and a parameter block pointed to by R1. Note that call with R0b=0 (Acorn MOS RDLN) does nothing, the ReadLine call should be used instead. For example:

```
MOV R0,#1
        SUB R1,SP,#5
        SWI Word ;read time

In: R0, R1 pointing to parameter block
Out: parameter block updated.
```

2.4.9 File &08 (Acorn MOS OSFILE)

Do an Acorn MOS OSFILE call with A=R0b. Instead of a parameter block R1 to R5 contain the data (string pointer, load address, exec address, start address {*length*}, end address {*attributes*}). For example:

```
MOV R0,#5
        MOV R1,#ptr
        SWI File ;file info

In: R0, R1 pointing to string, R2, R3, R4, R5
Out: R2, R3, R4, R5 updated.
```

2.4.10 Args &09 (Acorn MOS OSARGS)

Do an Acorn MOS OSARGS call with A=R0b, file handle in R1, data value in R2. For example:

```
MOV R0,#0
        LDR R1,handle
        SWI Args ;read ptr to R2

In: R0, R1 (handle), R2
Out: R2 updated.
```

2.4.11 BGet &0A (Acorn MOS OSBGET)

Read the next byte from the file whose handle is in R1. Byte returned in R0, validity in C (set if end of file). The Executive does local buffering for Input Only and Output Only files by using handles in the range 256 to 511 - see section 2.4.14. For example:

```
LDR R1,handle
        SWI BGet
        BCS EndOfFile

In: R1 (handle)
Out: R0, C.
```

2.4.12 BPut &0B (Acorn MOS OSBPUT)

Write R0b to the file whose handle is in R1. The Executive does local buffering for Input Only and Output Only files by using handles in the range 256 to 511 - see section 2.4.14. For example:

```
MOV R0,#data
        LDR R1,handle
        SWI BPut

In: R0b, R1 (handle)
Out:
```

2.4.13 Multiple &0C (Acorn MOS OSGBPB)

Read and write multiple bytes from file whose handle is in R1. Control in R0b; addresses in R2 (data pointer), R3 (number of bytes), R4 (pointer in file, if required). C is set if the transfer could not be completed. Executive local buffering does not apply to the multiple call. For example:

```
MOV R0,#1
        LDR R1,handle
        MOV R2,#data
        MOV R3,#56
        MOV R4,#100
        SWI Multiple ; put 56 bytes to
                     ; file from 'data' at offset 100

In: R0b, R1 handle,
    R2 points to data,
    R3 number of bytes,
    R4 position

Out: R2, R3, R4 updated.
     C set if transfer past end of file.
```

2.4.14 Open &0D (Acorn MOS OSFIND)

Open and close a file. If `R0b=0` then the file whose handle is in `R1` is to be closed; if `R1b=0` then all files on the current filing system are closed. If `R0` is not zero a file, whose name `R1` is pointing at, is to be opened, {&40 for Input only, &80 for Output, &C0 for Update}; the file handle being returned in `R0`. The Executive does local buffering for BGet for Input Only and for BPut for Output Only files by using handles in the range 256 to 511: this feature can be ignored by masking out the extra bit in the handle, but you must not mix use of the masked and unmasked handles. Alternatively the command `NOBUFF` can be used at the supervisor prompt (NOT as a CLI call). The buffering produces the following times for BGet-ing a 64K file:

ANFS	ADFS	Executive+ANFS	Executive+ADFS
40sec	32sec	10sec	4sec

For example:

```
MOV R0,#&40
        MOV R1,NameAddress
        SWI Open

In: R0, R1 (handle/ pointer to name)
Out: R0 (handle if opened)
```

2.4.15 ReadLine &0E

Read a line of text from the terminal. R0 points to the buffer where the text will be placed; R1 contains the maximum possible length of the line; R2 contains the lowest character which will be placed in the buffer (excluding carriage return); R3 contains the highest character which will be placed in the buffer. C will be set if the buffer was terminated by Escape. The input may have been terminated by &0D or &0A. For example:

```
SUB R0,SP,#256
        MOV R1,#238
        MOV R2,#" "
        MOV R3,#255
        SWI ReadLine
        BCS Escape

In: R0, R1, R2, R3
Out: R1 length, C set if invalid.
```

2.4.16 Control &0F

Set the control programs for the exception handlers.

- R0 address of where to go when an error occurs (0 for no change)
- R1 address of a buffer for error status (0 for no change)

This will contain:

- [R1,#0] PC when error occurred, for example, just after the SWI which called Executive
- [R1,#4] cardinal: error number provided with the error
- R1+8 error string, terminated with a 0.

Continuing after errors is simple: just have a handler that reloads the PC with that in the block; it may like to set the V flag so that the retried code knows what has happened (normal SWIs clear it). R0 may be incorrect

R2 address of escape routine handler (0 for no change)
Entered with R11 bit 6 as escape status
R12 contains 0/-1 if not in/in the kernal presently
R11 and R12 may be altered. Return with MOV PC,R14
If R12 contains 1 on return then the Callback will be used

R3 address of event handler (0 for no change)
Entered with R0, R1 and R2 containing the A, X and Y parameters. R0, R1, R2, R11 and R12 may be altered. Return with MOV PC,R14
R12 contains 0/-1 if not in/in the kernal presently
R13 contains the IRQ handling routine stack. When you return to the system LDMFD R13!,{R0,R1,R2,R11,R12,PC}^ will be executed. If R12 contains 1 on return then the Callback will be used.

The handlers are initialised and do not normally need setting. Errors will cause the error message and number to be written to the terminal. Escape updates will be counted: the program will be terminated on the third. The default event handler does nothing. Addresses of 0 do not update the respective information field. All handlers are reset in the supervisor's (* prompt) or when a program is run using CLI. For example:

```
MOV R0,errorh
        SUB R1,SP,#256
        MOV R2,#0
        MOV R3,#0
        SWI Control ;control errors

In: R0, R1, R2, R3
Out: previous values
```

2.4.17 GetEnv &10

Read the program environment.

R0 address of the command string (0 terminated) which ran the program
R1 address of the permitted RAM limit for example &10000 for 64K machine
R2 address of 5 bytes - the time the program started running. For example:

```
SWI GetEnv
        SWI Writes ;write environment
                   ;string to terminal

In:
Out: R0, R1, R2
```

2.4.18 Exit &11

Leave the program and return to the supervisor * prompt. The supervisor will print the time for which the program ran. For example:

```
SWI Exit

In: irrelevant
Out: never returns
```

2.4.19 SetEnv &12

Set the program environment. This call should only be used to alter the overall environment for subprograms.

R0 address of exit routine for Exit above to go to (or 0 if no change)
R1 address of end of memory limit for GetEnv to read (or 0 if no change)
R2 address of the real end of memory (or 0 if no change)
R3 0 for no local buffering, 1 for local buffering (anything else no change)
R4 address of routine to handle undefined instructions (or 0 if no change)
R5 address of routine to handle prefetch abort (or 0 if no change)
R6 address of routine to handle data abort (or 0 if no change)
R7 address of routine to handle address exception (or 0 if no change).

Undefined, abort and exception handlers are initialised on using CLI to run a program. For example:

```
ADR R0,EXITROUT
          MOV  R1,#&10000
          MOV  R2,#0
          MOV  R3,#4
          MOV  R4,#0
          MOV  R5,#0
          MOV  R6,#0
          MOV  R7,#0
          SWI  SetEnv ;make a small machine

In: R0, R1, R2, R3, R4, R5, R6, R7
Out: previous values in R0, R1, R2, R3, R4, R5, R6, R7
```

2.4.20 IntOn &13

Enable interrupts.

2.4.21 IntOff &14

Disable interrupts.

2.4.22 CallBack &15

On an Escape Update or Event routine the user may modify his own flags but cannot call the system with an SWI call if it was in the kernal since the Executive is using the Tube Hardware and would thus become deadlocked. The solution is a CallBack on kernal Exit - as the thread of control leaves the kernel, instead of returning to the original caller the registers are saved in a save block and a different routine is entered; any necessary kernel calls can be made and then control can be resumed by reloading from the register save block.

`R0` sets the address of the register save block
`R1` sets the address of the callback routine

For example:

```
ADR   R0,savblock
          ADR   R1,nullroutine
          SWI   CallBack

nullroutine:
          ADDR  R0,savblock
          LDMIA R0,&FFFF^

In: R0, R1
Out: previous values in R0, R1
```

2.4.23 EnterSVC &16

Gives the caller SVC privilege mode. One should not then use `R13` or rely on `R14` being preserved across SWI calls. Exit back to user mode with `TEQP PC,#0`.

2.4.24 BreakPt &17

Cause a break point trap. See 1.4.25

2.4.25 BreakCtrl &18

When the BreakPt SWI is made all the user mode registers will be dumped in a block and execution continued at the Break control routine. The user can be continued with a `LDMIA` *block*`,&FFFF^`.

`R0` sets the address of the register save block; 0 for no change
`R1` sets the address of the control routine; 0 for no change

```
In: R0, R1
Out: previous values in R0, R1
```

2.4.26 UnusedSWI &19

SWIs of values 512 onwards can be used to add new SWI function calls.

`R0` sets the address of the unused SWI handler; 0 for no change.

The following code has been executed before the handler is reached:

```
SPSVC  * R13
TUBER  * R12
SVCWK0 * R11
SVC  STMFD SPSVC!,{TUBER,SVCWK0,SVCWK1}
     BIC   TUBER,R14,#CCMASK
     LDR   SVCWK0,[TUBER,#-4]
     BIC   SVCWK0,SVCWK0,#&FF000000
     CMP   SVCWK0,#512
     LDRCS PC,HISERV
```

The old address should be used to return to the user so as to process any outstanding CallBacks: Executive kernal does the following:

```
SLVK LDR    SVCWK0,DOCALL
     TEQ    SVCWK0,#1
     BEQ    SKCL
     LDMFD  SPSVC!,{TUBER,SVCWK0,SVCWK1}
     BICS   PC,R14,#&10000000 ;clear V flag
DOCALL & 0
SKCL MOV    TUBER,#0
     TEQP   PC,#&0C000003
     STR    TUBER,DOCALL
     LDR    TUBER,CALLBF
     STMIA  TUBER!,{R0,R1,R2,R3,R4,R5,R6,R7,R8,R9}
     LDMFD  SPSVC!,{R0,R1,R2}
            ;get TUBER,SVCWK0,SVCWK1 (10,11,12)
     STMIA  TUBER!,{R0,R1,R2,R13,R14}^
            ;write those 3 and user's 13,14
     BICS   R14,R14,#&10000000 ;clear V flag
     STMIA  TUBER!,{R14} ;user's PC
     LDR    PC,CALLAD

In: R0
Out: previous value in R0
```

2.4.27 SetCallBack &1B

This kernel call sets the internal call back flag: when the kernel is next exited a CallBack will occur (not when this call to the kernel exits).

```
In:
Out:
```

2.4.28 WriteI &100

Write the character contained in the bottom byte of the SWI call. For example:

```
SWI WriteI+" " ;write a space

In:
Out:
```

3. Acorn Object Format

This chapter defines a file format known as ARM Object Format, which is generated by language processors for the ARM system. The ARM linker processes files in this format and also generates its output in the same format. The term object file is used to denote a file in ARM Object Format.

3.1 Overall structure

An object file contains a number of separate but related pieces of data. In order to simplify access to these separate pieces, and to provide for a degree of extensibility, the object file format is itself layered on another format called Chunk File Format, which provides a simple an efficient means of accessing and updating distinct chunks of data within a single file. This is fully described elsewhere, but adequate information is included below for convenience.

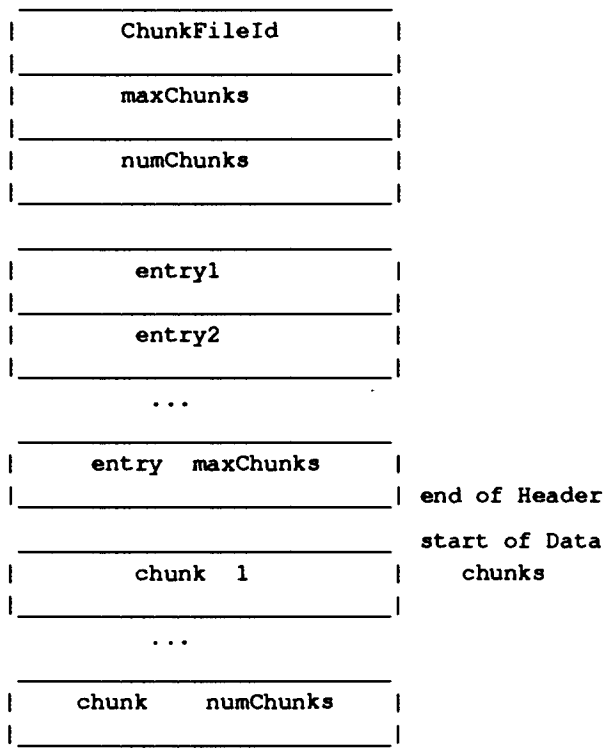
Each piece of an object file is stored in a separate, identifiable, chunk. The object file format defines five chunks: header, areas, identification, symbol table and string table.

The minimum size of a piece of data in both formats is four bytes or one word. It is easier to understand the format as a sequence of words than a sequence of bytes. Each word is actually stored in the file in Little-endian format; that is, the least significant byte of the word is stored first.

3.2 Chunk file format

Access to the individual chunks is made through a header at the start of the file. The header contains information on the number, size, location and identity of the chunks within the file. The size of the header may vary between different chunk files but is fixed for each file. However, not all entries in the header may be in use, thus allowing for a limited expansion of the number of chunks without a wholesale copy. A chunk file can be copied without knowledge of the contents of the individual chunks.

Graphically, the layout of a chunk file looks as follows:



The ChunkFileId is a single word field, which serves to identify the file as being in chunk file format. Its value is C3CBC6C5 hex. The maxChunks field defines the number of the entries in the header. This is fixed once the file is created. The numChunks defines how many chunks there are currently in the file, which can vary from 0 to maxChunks. The value of numChunks is redundant in the sense that it can be found by scanning the entries.

Each entry in the header comprises four words in the following order: chunkId, a two-word field; fileOffset a one-word field and size, a one-word field.

The chunkId field provides a conventional way of identifying what type of data a chunk contains. It is split into two parts. The first four characters (in the first word) contain a universally unique name allocated by a Acorn. The remaining four characters (in the second word) can be used to identify component chunks within this universal domain. In each part, the first character of the name is stored first in the file, and so on.

The fileOffset field defines the byte position within the file at which the chunk data begins, which must be at a byte position that is divisible by four. A fileOffset of zero specifies that this chunk entry is not in use.

The size field defines the exact size in bytes of the chunk, which need not be a multiple of four bytes.

3.3 Object file format

Each piece of the object file is stored in a separate, identifiable, chunk. The object file format defines five chunks: header, areas, identification, symbol table and string table, of which only header and areas must be present. The universal name for object file chunks is OBJ_, with the component names as follows:

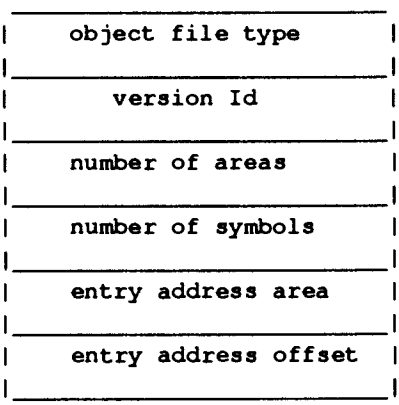
```
Header          OBJ_HEAD
Areas           OBJ_AREA
Identification  OBJ_IDFN
Symbol Table    OBJ_SYMT
String Table    OBJ_STRT
```

A typical object file will contain all five of the above chunks. A feature of the chunk file format is that they may appear in any order in the file.

A language translator or other system utility may add additional chunks to an object file, for example a language specific symbol table. Thus it is conventional to allow space in the chunk header for additional chunks; a maximum of eight is conventional.

3.4 The header chunk

The header chunk is logically in two parts. The first part is of fixed size and contains information on the contents and nature of the object file. The second part is variable in length (specified in the fixed part) and is a sequence of area declarations defining the code and data areas within the OBJ_AREA chunk. The fixed part of the header has the following format:



All fields occupy one word.

3.4.1 Object file type

An object file is identified by a type field, which is a convenience for system utilities which require a particular form of object file as input. The types which are recognised at present are as follows:

```
Relocatable      C5E2D080 hex
Image type 1     C5E2D081 hex
Image type 2     C5E2D083 hex
Image type 3     C5E2D087 hex
```

Most language translators generate relocatable object files.

3.4.2 Version Id

This word defines the version of the object format that was used to generate the object file. The current version number is 110 decimal.

3.4.3 Number of areas

The code and data of the object file is presented as a number of separate areas, in the OBJ_AREA chunk, each with a name and some attributes (see below). Each area is declared in the (variable-length) part of the header which immediately follows the fixed part. The value of this field in the fixed part defines the number of areas in the file and consequently the number of area declarations which follow the fixed part of the header.

3.4.4 Number of symbols

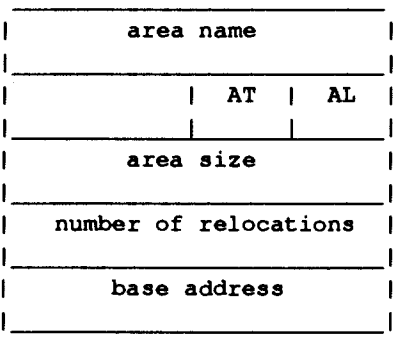
If the object files contains a symbol file chunk OBJ_SYMT, then this field defines the number of symbols in the symbol table.

3.4.5 Entry address area/entry address offset

One of the areas in the object file may be designated as containing the start address for any program which is linked to include this file. If so, the entry address is specified as *n, offset* pair, where *n* is in the range 1 to *number of areas*, specifying the n^{th} area declared in the area declarations part of the header. The entry address is then defined to be the base address of the area plus the offset. A value of 0 for *n* is taken as specifying no entry address.

3.5 Area declarations

The area declarations follow the fixed part of the header. Each declaration has the following form:



3.5.1 Area name

All names in an object file are encoded as offsets into the string table, which is stored in the OBJ_STRT chunk. This allows the variable-length characteristics of names to be factored out from primary data formats. Each area within an object file must be given a name which is unique among all the areas in that object file.

3.5.2 AL (area alignment)

When the area is included in a program image, it is aligned on the address boundary specified in this field, which is one byte in size. The allowable values for this field are between 2 and 12, which specify an alignment of two raised to the power AL. That is, between 4 and 4096 in powers of two. The standard value of AL is 2, specifying word alignment.

3.5.3 AT (area attributes)

As well as a name, an area can have a set of attributes. The linker groups areas first by attributes and then by name. The AT field is one byte in size and is interpreted as a sequence of bits, numbering the least significant bit as zero.

- Bit 0

  If this bit is set the area is absolute. The BaseAddress field of the area declaration gives the absolute address of the base of area. Otherwise it is relocatable.

- Bit 1

  If this bit is set, the area contains code, otherwise it contains data.

- Bit 2

  This bit specifies that the area is a common block definition. Common areas with the same name are overlaid on each other by the linker. The Size field of the declaration defines the size of the common block. All other references to this common block must specify a size which is smaller or equal to the definition size. In a link step there may be only one area of the given name with this bit set. If none of the common areas have this bit set, the actual size of the common area will be the largest size of the contribution common areas.

- Bit 3

  This bit defines the area as a common area. See the text on Bit 2 for further details on common areas.

- Bit 4

  This bit specifies that the area has no initialising data in this object file. In other words the area contents are completely missing from the OBJ_AREA chunk. This bit is typically used for large data areas where the initial values are unimportant.

- Bit 5

  This bit specifies that the area is read-only. The linker groups read only areas together so that they may be write protected at run time, hardware permitting. Code areas should ordinarily have this bit set.

- Bit 6

  This bit specifies that the area is position independent, which means that its base address may change at run time without any change being required to its contents. Such an area may only contain internal program-relative relocation and must make all external references through registers.

3.5.4 Area size

This field specifies the size of the area in bytes, which must be a multiple of four bytes. Unless the Not Initialised bit is set in the area attributes, there must be this number of bytes for this area in the OBJ_AREA chunk.

3.5.5 Number of relocations

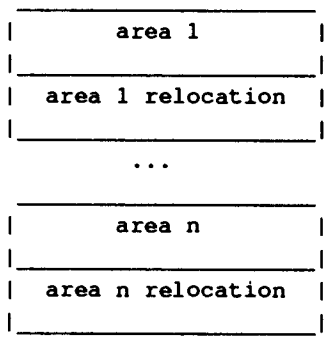
This specifies the number of relocation records which apply to this area.

3.5.6 Base address

This value is only meaningful if the area is absolute. It specifies the absolute address of the base (first byte) of the area.

3.6 The areas chunk

The areas chunk (OBJ_AREA) contains the actual areas (code and data) plus any associated relocation information. Its chunkId is OBJ_AREA. Graphically, the layout is as follows:



Both the area contents and the area relocation data must be aligned on a 4-byte boundary.

An area is simply a sequence of byte values, the order following that of the addressing rules of the ARM, that is the least significant byte of a word first. An area is followed by its associated relocation table (if any). An area is either completely initialised by the values from the file or not initialised at all (specified by bit 4 of the area attributes).

3.6.1 Relocation

If no relocation is specified, then the value of a byte/half-word/word in the preceding area is exactly that value that will appear in memory when the file is executed. Byte and half-word relocation is only possible by global constants. A field may be subject to more than one relocation.

Relocation can take two basic forms: additive and program-relative. Additive relocation specifies the modification of a byte/half-word/word, typically containing a data value (that is, constant or address). Program-relative relocation always specifies the modification of an instruction and involves the generation of a program counter relative displacement for either a branch instruction or a data transfer instruction using R15 as the base register.

- R (Relocation Type)
  This 1-bit field (bit 18) specifies either additive (0) or

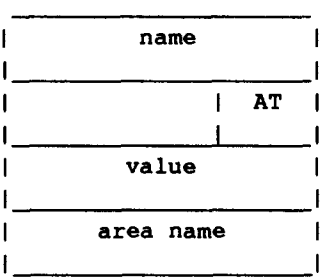
  program-relative (1) relocation.

- A (Additive Type)
  This 1-bit field (bit 19) is only interpreted if bit 18 is a zero. If the field is zero, it specifies additive internal relocation, implying that the base address of the area is added into the field to be relocated. If the field has the value one it specifies additive symbol relocation, implying that the value of the given symbol is added into the field to be relocated.

  Bits 20-32 of the second word in a relocation record are reserved.

3.7 The symbol table chunk (OBJ_SYMT)

The number of symbols field in the header defines how many entries there are in the symbol table. Each symbol table record has the following structure.



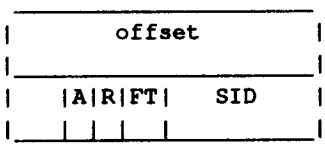
3.7.1 Name

This value is an index into the string table (in chunk OBJ_STRT) and thus locates the character string representing the symbol.

Additive relocation has two variants: internal and symbol. Internal relocation involves adding in the allocated base address of the preceding area to the field. Symbol relocation involves adding in the actual value of the symbol quoted.

Program-relative relocation always references a symbol, the value of which is used to compute the displacement to place within the instruction. The final displacement is derived from the address of the instruction being relocated, the value (address) of the symbol and any existing displacement in the instruction as found in the area. The fact that the PC is actually 8 bytes beyond the instruction is taken into account by the linker and should not be included by the creator of the object file.

An entry in the relocation looks as follows:



- Offset
  This field defines the byte offset within the preceding area of the field to be relocated.
- SID
  If a symbol is involved in the relocation, this field specifies the symbol-Id, which is the index within the symbol table (see below) of the symbol in question. The field is 16 bits in size.
- FT (Field Type)
  This 2-bit field (bits 16-17) specifies the size of the field to be relocated, as follows

```
00          byte
01          halfword
10          word
```

  The value 11 is illegal.

3.7.2 AT

This is a 4-bit field specifying the attributes of the symbol as follows:

- Bit 0
  If this bit is set, the symbol is defined in this object file. Otherwise, the symbol is an external reference to another object file. In this case bit 1 must be set as well.

- Bit 1
  If the symbol is defined (bit 0 set), this bit specifies that the symbol has global scope. That is, when attempting to resolve external references, the linker will match on symbols from other object files only if this bit is set. If the symbol is undefined (bit 0 unset) then this specifies an external reference to the symbol.

  Note that the combination, bit 0 set, bit 1 unset, specifies the definition of a local symbol. The linker will only match such symbols from references within the same object file.

- Bit 2
  The following attribute is only meaningful if the symbol is a defining occurrence (bit 0 set). If this bit is set it specifies that the symbol has an absolute value, for example, a constant. Otherwise its value is relative to the base address of the area defined by the area name field of the symbol table entry.

- Bit 3
  This bit is only meaningful if bit 1 is unset, that specifying an external reference. It states that the reference is case-insensitive. When attempting to resolve such an external reference, the linker will ignore case when performing the match.

3.7.3 Value

If the symbol is absolute (bit 2 of AT set), this field contains the value of the symbol. Otherwise, it is interpreted as an offset from the base address of the area defined by area name, which must be an area defined in this object file.

3.7.4 Area name

This field is only meaningful if bit 2 of AT is unset. It specifies an index into the string table which locates the character string representation of the area relative to which the symbol is defined.

3.8 The string table chunk (OBJ_STRT)

The string table chunk contains all the print names referred to within the areas and symbol table chunks. The separation is made principally to factor out the variable length characteristic of print names. Print names are stored in the string table and identified by an offset from the beginning of the table. Print names are stored as a sequence of ASCII characters terminated by a null (0) byte. The first character of each string must be aligned on a 4-byte boundary.

By convention, a value of 1 for an index into the string table is interpreted as defining the null name.

3.9 The identifiation chunk (OBJ_IDFN)

This chunk should contain an ASCII character string, terminated by a null (0) byte, providing information on the name and version of the language translator which generated the object file.

3.10 AOF-handling utilities

There are four utility programs which allow a programmer to investigate, merge and run AOF files. They are:

(1) Chunk file decoder `deccf`

(2) Object file decoder `decobj`

(3) Merge two chunk files `MergeCF`

(4) Run an AOF file `m2run`

These utilities can be found in chapter six, miscellaneous utilities.

4. The linker

Link processes files which are in Acorn object format (AOF) and also generates output in the same format. Link combines one or more AOF files into an absolute, executable image file, resolving external references and searching libraries for object files if necessary.

When generating an image, link produces an object file consisting of two areas, read only (data and code) and read-write (data).

4.1 Link areas

Areas with the same name and attributes are collected together and concatenated according to the order specified in the input list. These concatenated areas are then sorted alphabetically by name. See the ARM Object Format document specification for more details on areas.

The following are reserved symbols;

```
Image$$CodeBase,
Image$$CodeLimit,
Image$$DataBase,
Image$$DataLimit
```

If these symbols are referred to, they are set to the first location in the various areas as follows:

`Image$$CodeBase` : the read-only (code) area
`Image$$CodeLimit` : the first location beyond the read-only area
`Image$$DataBase` : the first location in the read-write (data) area
`Image$$DataLimit` : the first location beyond the read-write data

It is erroneous to define these symbols.

4.2 Keywords

The following describes the keywords, which control the operation of link.

- `-image`
  The argument to this keyword is the filename of the resulting object file.
- `-files`
  This argument is a list of object files to include in the link.
- `-via`
  This argument specifies a file from which a list of object files to link should be acquired. There should be one name per line in the file. This list is additional to and appended to that provided by the files argument.
- `-library`
  This argument specifies a list of object files libraries in which to search for unresolved external symbols. Libraries are searched as many times as necessary to resolve external symbols.
- `-use`
  This option specifies incremental linking with respect to one or more existing base images. For example, this can be used so that the resulting object may be dynamically loaded into an already executing program. The arguments are assumed to be previously linked image files whose symbol tables are taken as a basis on which to define new symbols. Only newly linked material will be entered into the new object file, but the symbol file will reflect every symbol defined before and after the incremental load. If any of the image files are marked as shareable, (see `-share`), any read-write data areas in these images are always duplicated in the new object file. This is typically used to allow code to be shared between a number of simultaneously active programs, each with a separate instance of the writeable data areas.

  Object filenames given with the `-files` and `-via` arguments may be postfixed by either `/l` or `/u`. This has the same meaning as if these files had been given as arguments with the `-library` or `-use` keywords.

- `-relocatable`
  This switch directs link to generate a relocatable object file as output. The object file retains the basic form of the component input files in that areas of the same name and attributes are coalesced into single areas. The only relocation that is resolved is program-relative relocation within the same area. The resulting object file may become part of the input files for another run of link.

- `-share`
  This switch directs link to generate a shareable image, which is a hybrid of an absolute image and a relocatable image. The code and read-only data areas are linked as per an absolute image, but any read-write areas are linked as per a relocatable image. This allows the read-write areas to be linked into an image which uses the code in the shared image, at a suitable address. This address may vary between different programs linking to the shared image.

- `-base`
  By default, link generates an absolute image with the read-only area starting at byte address 384K. This can be changed with this keyword, whose argument is interpreted as a decimal byte address, but can be postfixed by `k` to scale by 1024 and prefixed by `&` to specfify a hexadecimal value.

- `-pagealign`
  By default, the read-write area immediately follows the read-only area in the address space. That is ImageDataBase = ImageCodeLimit. If this switch is specified, the read-write data area is page aligned.

- `-keep`
  By default, non-external symbols are discarded from the symbol table of the resulting object file. This option forces their inclusion.

- `-adfs`
  This option specifies that the resulting file be suitable for direct execution by the ARM Executive ROM. In this case, the default base address is changed to 4K.

- `-chain`
  This keyword takes as argument a command that will be passed to the Executive ROM for execution, as the last action of link.

4.3 Examples

```
link -image Test aof.Test,aof.a,aof.b,aof.c -library $.lib.c
```

The object files of `Test,aof.a,aof.b` and `aof.c` are linked to form an absolute image in `Test`. The library `$.lib.c` is scanned to resolve any external references.

```
link -image Test -via aof.filelist -reloc
```

A relocatable object files is produced in `Test` by linking the list of object files in `aof.filelist`.

4.4 Diagnostics

Error messages are produced on the standard output.

4.5 Bugs

Shareable images are not implemented. Symbols that are multiply defined in one of the input files and an image give with the `-use` switch are reported as errors. This can be ignored.

5. Machine code debugger

The debugger is for debugging AOF files not containing unresolved references within the execution path. It works by loading itself where the program would normally load, and loading the program further up in store. The program can be run at this higher address without the control of the debugger (in order to achieve the same placement if it should be important) using the utility file `m2run`. The debugger takes control from the program by means of breakpoints, which cause a trap into the debugger. Thus the debugger cannot be used to stop a program which has gone wildly wrong and find out where it has gone.

The debugger is only for use in user mode, as its entry method following a breakpoint causes SVC mode to be entered temporarily, thus corrupting SVC register 14.

5.1 Starting the debugger

The debugger is started by typing:

`debug` *programname*

where *programname* is a linked AOF file. The debugger will respond with the prompt:

```
debug:
```

whenever it is ready for input.

5.2 Debug input conventions

The debugger deals with input of commands, numbers, register names and other miscellaneous items. The following terms will be used later in this manual, and so are defined now.

(1) A name is a sequence of letters, digits, dot and dollar beginning with a letter.

(2) A decimal number is a non-empty sequence of decimal digits.

(3) A hexadecimal number is an ampersand followed by a possibly empty sequence of hexadecimal digits.

(4) A register name is one of:-
R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, PC (=R15), SL (=R13), SP (=R12), FP (=R11) and SB (=R10).
All register names refer to the user bank of registers.

(5) Expressions are composed of numbers, names, register names, brackets and '+', '-', '/', '*', '^' and '.'. The symbols '+', '-', '/' and '*' have their usual meanings, '.' refers to the last location examined or deposited into, brackets may be used for clarity and forcing evaluation order, and ^ is a postfixed unary operator meaning 'the contents of' and delivers a word from a word-aligned address. In certain circumstances, a register name may be an expression on its own, such as when examining a register range. Otherwise, any register name used in an expression must be followed by ^ in order that the register contents be used. There is no meaning if the ^ is left out.

5.3 Debug commands

The debugger takes commands of the following forms:

name arguments

$arguments=arguments

5.3.1 Named commands

For commands which are names, only the first character is significant, and case is ignored, thus RUN, run and R are all equivalent.

(1) Run

syntax: `R` *rest-of-line*

The program is entered with *rest-of-line* as arguments which will be available to the argument decoder as if the program had been run from the supervisor command prompt. The debugger will not be entered again unless either a breakpoint is reached or a machine level trap of some sort is taken, such as an address exception.

(2) Single step

syntax: `S`

Execute the current instruction and re-enter the debugger after.

(3) Continue

syntax: `C`

Continue execution

The debugger re-enters the program. As with Run, the debugger can only regain control by means of a breakpoint or a machine level trap.

(4) Quit

syntax: `Q`

Leave the debugger and return to the supervisor.

(5) Breakpoint

syntax: `B S` *location*
`B D` *location*
`B D`
`B L`

Set or delete a breakpoint at the given location. The location may be an expression evaluating to a store address. A maximum of twenty breakpoints is permitted. Breakpoints work by replacing the existing instruction by an instruction which will cause a trap and enter the debugger. Breakpoints cannot be placed in ROM code. Deleting a breakpoint simply replaces the original instruction over the trap, and removes the breakpoint from the breakpoint table.

`B D` with no location specified will delete all breakpoints.

`B L` will list the addresses of all breakpoints.

(6) Unwind

syntax: `U`

Unwind the procedure call stack. This is only meaningful if the Acorn procedure calling standard has been used for procedure calling within the program being debugged. In this case it will give a list of procedure calls made starting with the most recent and ending at the mainline code.

(7) List

syntax: `L` *name*

List all symbols in the symbol table from the AOF which start with the given name. The output is not sorted, and is therefore in the order in which the symbols were encountered in the AOF.

(8) Examine

syntax: `E` {*$F*}
`E` *a1* {*$F*}
`E` *a1*:*a2* {*$F*}
`E` *a1*,*a3* {*$F*}

Examine locations, producing output in format *$F*, or in the default format if *$F* is omitted. *a1* is the first location examined; if omitted then the location after the last location examined/deposited into will be examined. Examination will continue until location *a2* or for *a3* locations, or if these are omitted only one location will be examined. The output format is precisely specified below.

Address	Value	
Base	Current base	Current base
Style	$Y	Current style
Length	4	Current length

(9) Deposit

syntax: `D` {*a1*}*value* {*$F*}

Deposit value in location *a1*, or if *a1* is omitted then in the next location, that is the one after the last to be examined/deposited into. *$F* specifies the amount of store to be updated (byte, half-word or word).

(10) Convert

Syntax = *a1* {*$F*}

Convert (display) *a1* in given format, or in the default format in none has been given.

(11) Format

Syntax: *$F*

Set default format for output.

The formats are:-
Base

`$D` decimal
`$X` hexadecimal
`$O` octal
$number base that number, in the range 2 - 36

Style

`$N` numeric
`$S` string
`$I` instruction (see note 1)
`$Y` symbolic name (see note 1)
`$P` as a PC part of a number (see note 1)
`$F` as a flags part of a number (see note 1)
`$R` as a floating point number if length = 4 or 8
(this is not yet implemented)

Note 1: these default to `$N` if the length is not a word.

Length

`$T` two words
`$W` word
`$H` half-word
`$B` byte

6. Floating point emulation

A general co-processor interface has been designed for ARM and future co-processors may have an unspecified number of registers, and a co-processor identity number in the range 1 - 15. Various new instructions will be added to the ARM to control these co-processors, and these will be of three types:

- CPDT – CoProcessor Data Transfer,
  transfer between a co-processor register and store
- CPRT – CoProcessor Register Transfer,
  transfer between a co-processor register and an ARM register
- CPDO – CoProcessor Data Operation,
  an operation within the co-processor.

The first such co-processor will be one that does IEEE floating point calculations but for ARM machines without this extra hardware, a floating point emulator has been provided which provides the functionality of a hardware processor with reduced performance. The emulator is entitled `fpe` and it should be placed in `$.library` in the filing system.

This chapter describes the instruction set extensions provided by this emulator for floating point calculations, and the syntax used by the assembler to represent these operations.

6.1 Programmer's model

The FP CoProcessor has eight work registers f0..f7, and a 32-bit status register. The work registers can each hold a single or double precision floating point number, although their internal format is not specified.

Single and double precision operations are provided. However, if the programmer were to load one size and then perform operations of another without converting explicitly between them, the result would be undefined.

There are separate instructions for loading and storing the condition code registers. Condition codes format:

```
 31  21  20  19  18  17  16 15  5   4   3   2   1   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
|   NAN INX ILL DVO OFL UFL|   NAN INX ILL DVO OFL UFL|
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
 <- - interrupt mask - - -> < - -cumulative flags- - >
```

NAN – not a number encountered in numeric application
INX – inexact result
ILL – illegal operation
DVO – divide by zero
OFL – overflow
UFL – underflow

For each unusual circumstance that could occur, there are two bits. One gets set if it ever occurs, the other indicates that its occurrence should cause an interrupt. An interrupted operation can probably not be successfully retried, because the execution of FPDO operations by the hardware FP unit may be asynchronous.

Notes

- Extra bits may be added to this mask in future, but currently the non-existent bits read as 0s, and efforts to write to them will have no effect.
- Exceptions are not guaranteed to occur directly after the operation that caused them, for a number of reasons: the hardware FP unit could be asynchronous; the FP numbers might not be represented in IEEE format in the FP registers, but in some unpacked format in order to improve performance. This format, for instance, might have greater range and precision than IEEE. So, a store operation could cause an overflow long after the construction of that value.
- References to registers 8-15 cause undefined values or traps (the result has not yet been defined).

6.2 Floating register directives

Assembler syntax:

```
f0      FN      0               ; behaves like RN
f1      FN      1
:       :       :
:       :       :
f15     FN      15
```

Floating register numbers and ARM register numbers must be distinguishable, since some of the MOV syntax below depends on this. A name can be given to the floating point status register thus:

```
FPPSW FN 15
```

There is also a directive that prevents the assembler from recognising floating point operations of any form:

```
NOFP
```

This may sometimes be useful for debugging.

6.3 Floating point constants

Assembler syntax:

```
DCF(D|S)        number <, number>*
LDF(D|S)        fn, =number
```

```
D -> double (64 bit), S-> single (32 bit).
```

In this syntax description, | means or, angle brackets mean optional round brackets denote grouping, * means repeat.

The DCF directive causes 32-bit or 64-bit IEEE floating point constants to be deposited in memory. An = constant generates a program counter relative load, use LTORG to actually generate the constant.

The syntax of numbers is:

```
number ::=  <+|->digits<exp>
            <+|->digits.<exp>
            <+|->digits.digits<exp>
            <+|->.digits<exp>

exp ::=     (e|E)<+|->digits

digits ::=  non-empty string of digits
```

6.4 Load and store instructions

Syntax:

```
(LD|ST)F<cond>(D|S) Fn, [Rn] <, off>
                                    [Rn <, off> ]
                                    [Rn,  off] !
```

These instructions load and store the floating registers. Addressing modes are very similar to `LDR` and `STR` instructions, except that the only possible offset form is an 8-bit word offset.

The 32-bit instruction takes the form:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
|Cond|110P|U/D|CLn|Wb|L/S|Rn|CRd|0001|8-bit offset|
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

P = 0, Wb = 0 -> post inc/dec, update Rn
    0, Wb = 1 -> DO NOT USE
    1, Wb = 0 -> pre inc/dec, do not update Rn
    1, Wb = 1 -> pre inc/dec, update Rn.
                 Do not use when Rn=15.
U/D = 0 -> subtract offset from Rn
      1 -> add offset to Rn
CLn = 0 -> S
      1 -> D
L/S = 0 -> STF
      1 -> LDF
```

6.5 Conversion instructions

Conversions between floating point and integer data take the form of a transfer between a FPCP register and an ARM one. The (D|S) refers to the floating point value, in all cases 32 bits are transferred between the processors.

Syntax:

```
FLT<cond>(D|S)      fn, rn    ; convert to FP
    TRN<cond>(D|S)      rn, fn    ; truncate towards zero
    FIX<cond>(D|S)      rn, fn    ; nearest integer
```

For the 32-bit layout, see the move instructions below.

6.6 Move instructions

Syntax:

```
MVF<cond>(D|S)    fn,  n       ; see note 1
    MVF<cond>(D|S)    fn, fn       ; see note 2
    MVF<cond>         rn, FPPSW
    MVF<cond>         FPPSW, rn
    MVF<cond>         fn, rn       ; see note 3
    MVF<cond>         rn, fn
    MVF<cond>D(1|2)   rn, fn
    MVF<cond>D(1|2)   fn, rn
```

Notes

- n in the range [0..15]. CPRO form.
- CPRO form.
- transfer single precision number.

Some of these are CPRT and some CPDO operations: the ones that mention an `rn` are CPRT. `MVFD1` and `MVFD2` allow the transfer of either the first or second 32-bit half of a double-precision FP number, this is because CPRT is only capable of accessing a single 32-bit ARM register in one instruction. When transferring to an FP register, a `MOVD2` should always follow a `MOVD1`, both mentioning the same FP register.

The bits for CPRT operations are:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
|Cond|1110|opcode|L/S|fn|rn| 0001 | 0001 | 0000 |
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

L/S = 1 -> the transfer is to an ARM register
0 -> the transfer is to an FP register

`rn` may not be 15 if L/S is 1 (that is, to ARM registers)

opcode L/S :=

```
000 0 -> FLTS
000 1 -> FIXS
001 1 -> TRNS
010 0 -> FLTD
010 1 -> FIXD
011 1 -> TRND
100 0 -> MVF FPPSW, rn     ; fn = 0
100 1 -> MVF rn, FPPSW     ; fn = 0
101 0 -> MVF fn, rn        ; transfer single precision
101 1 -> MVF rn, fn                              number
110 0 -> MVFD1 fn, rn
110 1 -> MVFD1 rn, fn
111 0 -> MVFD2 fn, rn
111 1 -> MVFD2 rn, fn
```

For an `FPPSW` transfer the `fn` field should be 0.

The bits for CPRO operations are shown in the section on Arithmetic operations.

6.7 Compare instructions

Syntax:

```
(CMF|CNF)<cond>(D|S) fn, <fm| m>
```

These instructions are CPRT format. The bits are:

a := 0 -> fm is a register
1 -> fm is a small constant, in the range [0.0 .. 15.0]

b := 0 -> single
1 -> double

c := 0 -> CMF (set N, Z, C, V from fn-fm)
1 -> CNF (set N, Z, C, V from fn+fm)

Following one of these instructions the conditions `EQ NE LT GT LE GE` work as you would expect. The assembler will recognise a small negative constant and invert the `CMF/CNF` as appropriate. Note that the unsigned comparisons will not necessarily work after this instruction.

6.8 Arithmetic instructions

Syntax:

```
op<cond>(D|S)     Fdst, Fs1, Fs
                  Fdst, Fs1,  n          ; n in [0..15]

op1<cond>(D|S)    Fdst, Fs2
                  Fdst,  n
```

```
op  ::= ADF SUF MUF DIF RSF RDF ASF POW RPW
op1 ::= SQT CVT ABS MVF MNF
        SIN COS TAN ASN ACS ATN LGN LOG EXP
```

In one instruction all operands are the same size, except `CVT` (convert single<->double), for which the `(D|S)` refers to the source.

The 32 bits are:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
|Cond | 1110 | abcd | Fs1 | Fdst | 0001 | efg0 | Fs2 |
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

a := 0 -> Fs2 is a register
1 -> Fs2 is a small constant [0.0 .. 15.0]
b := 0 -> single operation
1 -> double operation
ecdfg := (note the slightly unusual order)

```
00000 -> ADF add
00001 -> SUF subtract
00010 -> RSF reverse subtract
00011 -> ASF abs value of subtract
00100 -> MUF multiply
00101 -> DIF divide
00110 -> RDF reverse divide
00111       spare
01000 -> CVT convert D->S (if (D|S)), or S->D
01001 -> ABS absolute value
01010 -> SQT square root
01011 -> MVF move
01100 -> MNF negate
01101 -> LOG log to base 10
01110 -> LGN log to base e
01111 -> EXP e to the power of
10000 -> POW arg1 to the power of arg2
10001 -> RPW arg2 to the power of arg1
10xxx       spare
11000 -> SIN sine of number of radians
11001 -> COS cosine
11010 -> TAN tangent
11011 -> ASN arcsine, in radians
11100 -> ACS arccos
11101 -> ATN arctan
1111x       spare
```

leave Fs1=0 in monadic operations.
c = 0 -> dyadic operation
1 -> monadic operation

Notes

- Even with hardware support, loading ARM registers will always be slightly faster than loading FP registers. When simply moving floating point information around rather than performing operations on it, programmers and compiler writers are recommended to use the ARM registers rather than the FP ones. The best strategy for argument passing in compiled code is to pass pointer arguments in ARM registers, and return a floating point result in f0.
- The layout of IEEE numbers in ARM store is not quite to IEEE standard.
  The most significant word is first:
  that is, storing a double floating point number at x dumps:

  at x+4: 32 bits, low part of mantissa
  at x : 1 sign bit, 11 exp bits (subtract 1023), 20 mantissa bits.

  for example, the constant 1 in an assembler program is

  ```
  &          &3ff00000, &00000000
  ```

- If the second argument to `POW` (or the first to `RPW`) is an integral value in the range -50..50 then an integer power algorithm will be used.

6.9 The FPE and the Executive

FPE takes the form of a shell that runs under the Executive ROM.
Typing fpe after the ARM* prompt causes various things to happen:

- A title and version number will appear.
- A few thousand bytes of store at the top of memory is taken over by FPE, which has copied itself to its resident position.
- Also in high memory are some bytes which have the task of emulating the floating point registers, and some general floating point workspace. Floating point registers are initially zero, the status word is &002e0000 (that is, overflow, divide by zero, illegal operand and non-numbers will all cause exceptions).
- The ARM prompt logo has been replaced with a similar one containing an FP to denote that floating point instructions are now available.
- The illegal instruction trap vector has been replaced.

6.10 Notes for advanced users

The processor must be in user mode when emulated instructions are called.

Debug traps illegal instructions, and uses the instruction &ee000000 to implement single stepping and breakpoints. This works perfectly well with FPEbecause FPE installs itself directly in the ARM illegal intruction vector rather than through Executive's SetEnv. &ee000000 is still an illegal instruction. Any bit pattern not recognised as a floating point instruction is forwarded to the illegal instruction handler.

If a floating point exception happens ARM will print a message and then proceeds to perform an address error.

At location &FF4 is a pointer to the emulated FP registers. They are held in an unpacked format, each register takes 16 bytes:

x : high 32 bits of mantissa, left aligned, that is bit 31=1
there is an assumed decimal point between bits 30 and 31.
x+4 : low 32 bits of mantissa
x+8 : binary exponent as a 2's c number, that is no offset
x+12: = &80000000 (negative) or 0 (positive)

Zero is represented by a mantissa high word of 0, garbage sign and exponent.

Following the eight registers in this form, at offset 8*16, is the flags in one word.

The arithmetic takes place to the accuracy required by the IEEE specification. The IEEE concepts of infinity, non-numbers and denormalised numbers are not supported, and a request to load any of these into the processor will cause an exception. Overflows are not checked after every operation, and so a store can cause an overflow exception. Regardless of this, one should program on the assumption that store/load has no effect. If you use load/store floating to move arbitrary bit patterns around (as some FORTRAN programs do) then some bit patterns will become corrupted.

6.11 Floating point performance on ARM

Experimental Timings:

Instruction	usecs
MVFD f0, f0	19.2
LDFD f0, =x	23.3
ADFD f1, f0, f0	32.1
MULD f2, f1, f0	38..95
DIVD f2, f1, f0	39..125
SIND f1, f2	690
TAND f1, f2	1470
ATND f1, f2	780

These times assume N=2S, S=.15 usecs.

A block of adjacent floating point instructions goes faster than the sum of these times: subtract about 8 usecs for at each sucessive instruction.

7. Miscellaneous utilities

7.1 Filing system utilities

These are a set of programs which allow the ARM to exploit the ADFS filing system to the full.

7.1.1 List utility

The parameters are chosen from:

Syntax: `ls` *{-parameter} {-parameter}*

This utility provides a sorted list of selected files or complete directories. It can be used to produce well-ordered lists and is more versatile than the `*CAT` of the standard disc filing system. The parameters are:

(1) *filename/directoryname*
Gives a catalogues of selected files or directories. Wildcard characters ? and * may be used in the filename or directory name. In the absence of this parameter, the current directory is listed in a way which has been specified by the `full` and `bydate` switches (see below).

(2) `full` (minimum syntax `fu`)
This switch provides a comprehensive listing giving the dates in full, the length in decimal and the level of access permitted.

(3) `bydate` (minimum syntax `b`)
This switch allows the catalogue to be sorted into chronological order using the dates provided by the files.

For example:

```
ls util? -fu -b
```

will give a full date-ordered catalogue of the files `util1, util2, util3` and so on, from the current directory.

7.1.2 Disc use utility

Syntax: `du` *{-parameter} {-parameter}*

This utility gives information on the amount of space used on a section of disc or in a specific directory. This is a useful program to apply to the hierarchical filing system ADFS since it can give information on sections, giving the size of a named directory and also the size of the subdirectories available from that directory. The parameters are:

(1) *directoryname*
Scans the selected directory and all subdirectories and gives comprehensive size information on the files found.

(2) `total` (minimum syntax `tot`)
This switch inhibits much of the information given by `du` so that just the bytes used total is shown.

(3) `to` *filename* (minimum syntax `to`)
Directs the `du` output (comprehensive or totals) to a named file.

For example:

```
du gcallib -to temp
```

will give the size of files in the directory gcallib (and the size of any subdirectories which gcallib may possess) and place the results in a file called temp in the current directory.

7.1.3 File/directory delete utility

Syntax: `rm` *{-parameter} {-parameter}*

This utility complements the DELETE and DESTROY commands of the ADFS filing system and makes it possible to remove the entire contents of a hierarchical tree. `rm` should be used with the utmost care! Deleted files are irretrievable. The parameters are:

(1) *filename/directoryname*
The files or directories to be deleted. The default is the present directory.

(2) `recurse` (minimum syntax `r`)
This is a switch which causes `du` to check all subdirectories.

(3) `force` (minimum syntax `fo`)
This is a switch which causes file protection to be ignored, that is, the delete is forced. Since directories are usually locked, this switch will frequently need to be used.

(4) `confirm` (minimum syntax `c`)
This switch causes deletions to be interactively confirmed, that is, the delete will only be made when the user has typed `Y` to confirm that the file or directory should be wiped.

For example:

```
rm gcallib -fo -r -c
```

will ask for confirmation to erase the gcallib directory and all gcallib's subdirectories.

7.1.4 Grope utility

Grope is a powerful search utility capable of locating a sequence of characters within all files, specified files or a specified group of files. The conventions of this utility are similar to those of the find function of the ARM's editor TWIN. The sequence of characters to be searched for is known as the pattern. The search is line based and if the pattern crosses a line end it will not be found. Pattern length is limited to 1024 characters. When the pattern is found, the filename and the line number within that file will be printed to the screen. If grope is run as a task under TWIN, then the output will be directed to a TWIN file where it can be inspected at leisure.

Syntax: `grope` *{parameters}*

The parameters are:

{pattern} the pattern to search for
{`-nocase` } case insensitive search
{`-case` } case sensitive search, (the default)
{filenames} files to search in.

The patterns are constructed using a combination of alphanumeric and special characters to provide an intelligent degree of searching using sophisticated wildcards. The special characters are:

`.`
: matches any single character

`$`
: match the end of line character

`@`
: matches any identifier character A-Z, a-z, 0-9 and _

`#`
: matches any digit

`|c`
: make c a control character. If '@' <= c <= '_' matches CTRL c If c = '?' matches the delete character

`|!|c`

make c a character in the range 128-149

`|!c`

make c a character in the range 150-255

`\c`

match exactly this character, for example, \# matches a hash

`[abc]`

defines a set of characters, any one of which may be matched, for example, matches any one of a, b or c. Within the set delimited by square brackets only the following characters retain their special meanings: $ @ # \

`|`

within a set the sequence c1-c2 means match any single character between c1 and c2 (inclusive) in ASCII order, that is, a range of characters may be specified

`~c`

don't match c. c is an alphanumeric, special, control or literal character or a set of characters. Matches any character which is not c

`*c`

match as many contiguous occurrences of c as possible. (that is, zero or more occurrences). c is an alphanumeric, special, control or literal character or a set of characters, or a ~c character

`^c`

match the most possible contiguous occurrences of c. (that is, one or more). c is an alphanumeric, special, control or literal character or a set of characters, or a ~c character.

For example:

`grope LDMIB file1 .`
Look for the opcode LDMIB in file1.

`grope computer -nocase file2,file3`
Look for the word computer or Computer in file2 and file3.

`grope ^ $ file4`
Look for occurrences of trailing spaces in file4.

`grope -pathelp`
Ask for help on grope.

7.2 Program development utilities

7.2.1 Comparing files

The utility files `compare` and `compx` are for comparing a file produced by a program to see whether it differes from a file produced by a previous version of the program. `Compare` compares ASCII files, while `compx` compares binary files. The compare uses a sophisticated algorithm and will not stop when the two files fail to match. Match failures are reported, then an attempt is made to synchronise the two files for further comparison attempts.

ASCII files are compared using the syntax:

`compare` *filename1 filename2 {*`-to` *filename3}*

Binary files are compared using the syntax:

`compx` *filename1 filename2 {*`-to` *filename3}*

`Filename1` and `filename2` are the files to be compared. `Filename3`, which is optional, is a file into which the output generated by the compare may be placed.

7.3 Software clock

ARM software supports a clock so that files can be time-and-date stamped. The clock is not independent of the ARM's power supply, and will not in itself maintain time over a power break. The utility file `DATE` expects to be followed by parameters in the form:

DATE DD-MMM-YY HH:MM:SS

for example:

```
DATE 16-May-86 10:15:30
```

The system has been dated as the 16th May 1986, at a time of 10:15am (and 30s), and that information is now available to other utilities and programs. TWIN, for example, will start up with the current date and time shown on the status line. The date of the TWIN file will be updated when the file is saved. If the date has not been set, some programs will declare the date and time to be unset.

7.4 Memory test

A variety of tests are performed by the program `MemTest` to validate the address and data lines of the Random Access Memory (RAM). The utility reports the four phases of tests as it performs them, with dots indicating progress within each phase. In the example given below, the memory test has been run as a task from TWIN so it confined its test to the free memory under TWIN and went no further than &001D0000.

```
Memory tester for ARM 2nd processor
Testing up to address &001D0000
Phase one: incrementing pattern....
Phase two: TRUE hierarchy.....
Phase three: FALSE hierarchy....
Phase four: Cycling bits...
All seems OK
```

If any errors are found, the first ten are reported in detail, then the program continues the test, making a tally of the errors found. The total count of errors is given at the end of the test.

7.5 Column printing

The file `mc` is a utility for multi-column file printing. If an ordinary printout is required, the column width can be set to the paper width. However, multi-column printing is particularly suited to narrow listings and reduces the amount of paper needed for the print run. The program may be given useful parameters such as the width of the paper used, the number of columns required across the page, and the number of lines required down the page. Other parameters may also be given. The program, also has the capability of creating a postscript file for printers which can accept postscript code. Font size may be specified if the printer used is an Apple Laser Writer.

The program has a measure of intelligence in that it will attempt to construct a page with a satisfactory column layout if no specific requirements are passed to it.

Syntax: `mc` *{parameter list}*

The parameters consist of the following options:

(1) `From` *list of files to be printed*
A list of files may be provided; they will be printed one after another.

(2) `To` *destination*
The destination should be either `printer:` (outputs to the currently selected printer).

(3) `Title` *name*
The title to be printed on output pages. If no name is provided, the filename is used.

(4) `Lines` *n*
Lines to be printed per page. The default value is 65, unless the `laser` option is on, in which case the default value is computed according to the value of other parameters.

(5) `Width` *n*
Characters to be printed per output line. The default is 132, unless the `laser` option is on, in which case the default value is computed according to the value of other parameters.

(6) `Columns` *n*
Columns to be printed per page. If not used, the number of columns across the page will be computed.

(7) `Sep` *n*
Number of spaces to separate each column on the output page. The default is 2.

(8) `LN`
This is a switch. It causes source lines to be numbered.

(9) `Start` *n*
Number of first source line to be printed. The default is 1.

(10) `Stop` *n*
Number of last source line to be printed. The default value is infinity.

(11) `Truncate`
This is a switch. It causes lines which are too long to fit into the requested column size to be truncated. Otherwise they are wrapped round.

(12) `CRisNL`
This is a switch. It causes carriage returns in the input text to be treated as new lines.

(13) `Size` *n*
The size of output text, in units of 1/20 point. The figure is only meaningful if the output is to a laser printer. The default value is 120.

An example of the command could be:

```
mc -from fred joe -lines 58 -width 80 -sep 4
```

7.6 The command command utility

The `C` command is used to execute a command sequence with parameter substitution, and it requires that `$.execlib` and `$.tmp` to exist in the filing system.

Syntax: `C` *filename argument argument ...*

The *filename* refers to the command sequence file, which is searched for in the current directory or in `:0.$.execlib`. By renaming the C program as a different name, that name is used for the *filename*, for example, if `C` is duplicated as `f77` then the command `f77` will cause `:0.$.execlib.f77` to be obeyed.

C functions by copying the command sequence to the temporary file called `:0.$.tmp.exec`, obeying directives and performing parameter substitution in the process. If the current input is from an exec file, this is appended to the `tmp.exec` file so that nested command sequences can be used. The `tmp.exec` file is then executed by the command exec `:0.$.tmp.exec`.

The files in `:0.$.execlib` contain directives. These are:

- `.key` *keystring*

  Defines the *keystring* to be used to decode the parameters. The keystring is in BCPL format and `.key -help` will give the syntax. Only one .key directive is allowed: it must be present to allow any parameter subsititution referring to command line keys.

  Example: `.key from/a,to/a/k,opt/k`

- `.dot` *char*

  Subsequent directives introduced by different character.

  Example: `.dot +`

- `.bra` *char*

  Opening bracket for parameter substitution to be *char*.
  Initially <

- `.ket` *char*

  Closing bracket for parameter substitution to be *char*.
  Initially >

- `.dollar` *char*

  Set the character used to introduce parameter defaults to *char*.

  Initially $

- `.default` *key vl*

  Set the default value of keyword *key* to *vl*. Only one `.default` is allowed for each key, and it must be after the `.key` directive.

  Example: `.default to vdu:`

- `.concat` *char*

  Concatenate a non-directive line ending in *char* with the following line. There is no initial `concat` character.

- . *text comment*

  A dot and space precedes a comment line.

The value of a parameter may be inserted with a reference like `<key>` where the < and > are specified by `.bra` and `.ket` above. The value of a set switch `/S` parameter is its name. A default value for a key may be specified by `<key$default>` or even a default to another parameter `<key$<key2$default>` (note only one > is needed).

As well as the names specified in the `.key` directive, the following are also possible:

`=DATE` the date dd-mmm-yy

`=TIME` the time hh:mm:ss

`=DAYNO` the day of the month dd

`=MONTH` the month mmm

`=YEAR` the year 19yy

(or <unset> if the system time has not been set).

7.7 AOF handling utilities

7.7.1 Chunk file decoder

This utility displays textually the contents of a chunk file.

Syntax: `deccf` *-parameters {-parameters}*

The parameters are:

(1) *chunkfilename*
Chunk file to be decoded.

(2) `display` (Minimum syntax `d`)
The list of chunks to be displayed. (The default is none.)

(3) `to` (Minimum syntax `to`)
The decoded output, which defaults to standard output.

7.7.2 Object file decoder

This utility displays the various sections of the object file such as headers and area declarations. The utility can be used to check that the output of a program under development is creating files in the correct format.

Syntax: `decobj` *-parameters {-parameters}*

The parameters are:

(1) *objectfilename*
The object file to be decoded.

(2) `to` (Minimum syntax `to`)
The decoded output, which defaults to standard output.

The following keywords control which parts of the decoded object file are actually output. Since the default is for all parts, the keyword should be quoted as `-noKeyword` to suppress output.

(1) `header` (Minimum syntax `h`)
The object file header.

(2) `areadecls` (Minimum syntax `aread`)
The area declarations.

(3) `ident` (Minimum syntax `ide`)
The identification.

(4) `areas` (Minimum syntax `area`)
The area contents.

(5) `symbols` (Minimum syntax `s`)
The symbol table.

(6) `strings` (Minimum syntax `st`)
The string table.

7.7.3 Merging two chunk files

This utility allows chunk files to be manipulated.

Syntax: `MergeCF` *-parameters {-parameters}*

The parameters are:

(1) *chunkfilename*
The chunk file to be updated.

(2) `with` *secondfile* (Minimum syntax `w`)
The chunk file which is to be merged with *chunkfilename*.

(3) `replace` (Minimum syntax `r`)
Overwrite any existing chunks in *chunkfilename* which also occur in *secondfile*.

For example: `mergecf target -with update`

This would merge the chunks in update with those in target. Since the -Replace switch is not set, it is an error for any chunks with the same identity to appear in target and update.

7.7.4 m2run

OAF files may be run directly by the utility `m2run`, but only if the files themselves do not contain unresolved references to other AOF files. Normally `m2run` will be used in conjunction with the debugger to test and evaluate a single AOF file.

Parameters may be included in the command line after the filename and will be handed to the application being run on the assumption that the parameters are appropriate.

The syntax is:

`m2run` *filename -parameter {-parameter}*

8. Appendix A

8.1 Size of utility programs

Set machine time	Date	20K
List catalogue	LS	82K
Display usage of hierarchy	DU	82K
Delete hierarchy	RM	81K
Search file(s)	grope	86K
Text file compare	Compare	29K
Binary file compare	CompX	24K
Machine code debugger	Debug	117K
The linker	link	107K
Decode chunk file	deccf	75K
Decode object format file	decobj	93K
Merge chunk file	MergeCF	78K
Run object format file	m2run	38K
Multicolumn print	mc	35K
Floating point emulator	fpe	16K
Memory test	MemTest	1K
Command command programs	C	26K


Acorn
The choice of experience.