About This Recreated Document

Title: Brazil Manual (for the Acorn ARM Second Processor)
Last remade: 1-Jun-2025
Repository: https://github.com/acheton1984/AcornDocsRemade

This is not quite the beautifully "remastered" document as seen elsewhere, but it is intended to probably similar to the original in an easy-to-read and searchable PDF format.

Reconstruction Notes

Source file “BrazilMan” from either:
https://arcarc.nl/archive/Acorn%20Source%20Code/ARM%20Co%20Pro%20-%20Brazil%20BASIC%20TWIN%20EDIT/
or
https://gtoal.com/acorn/arm/Kernel/

- Painfully recreated from the original GCAL document using the b-em emulator, a Domesday86 HD image, Panos, GCAL for Panos, ChatGPT to help correct the resulting PS file and ps2pdf.
- A title and remade document history page have been added, along with a link to this repository.

Disclaimer

---- original document follows ----

Brazil Manual

1. Contents

2. Introduction

This document 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 etc). The BBC Microcomputer requires additional software from the DNFS ROM to deal with the Tube. The ARM Second Processor contains a ROM to deal with its end of the Tube, this is the Brazil Kernel. The ROM also contains a set of useful utilities in the Brazil 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 1024K

Acorn DFS

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 Brazil Supervisor.

3. Brazil Supervisor

When nothing else is using the system, the Brazil 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 (i.e. it does not refer to the pc).

`Help` - generates reassuring message from Brazil Supervisor: gives version number of the Brazil Kernel (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 Brazil Supervisor by performing an `SWI Exit`. Any other Supervisor will be returned to - or, of course, the Brazil 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. Examples:

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

The first two arguments are simply `CLI`ed, so they can cause quite a wide range of effects. The character @ in the first two arguments will be replaced by " ", the character ‘ in the first two arguments 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 filesystems, 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 Brazil Kernel level, so it still works from inside other systems. GOS enters the

Brazil Supervisor: the caller can be returned to if he has set an Exit handler using the Quit command.

Acorn MOS CLI commands e.g. `CAT, KEY` or sideways ROM programs `TYPE` etc. may be used by typing them directly to the A* prompt. 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.

4. ARM Second Processor Memory Map

```
====================================
| Brazil Supervisor ROM (16K)      |
==================================== &3000000
| Nothing: Aborts are caused       |
==================================== &2000000
| Tube IO chip                     |
==================================== &1000000
| Repetitions of the RAM area      |
==================================== Ramtop e.g. &100000
| File buffer RAM                  |
==================================== Ramtop-20K
| User RAM                         |
|                                  |
==================================== &1000
| System RAM: Stacks etc.          |
==================================== &0D00
| System RAM: Brazil Kernel itself |
==================================== &0000
```

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

5. Brazil Kernel

All BBC Microcomputer IO is accessed through SWI calls.

The following information documents all the Brazil 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 etc 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.

(1) WriteC [&00] (Acorn MOS OSWRCH)

Write R0b to the terminal output.

Example:

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

```
In: R0b
Out:
```

(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.

Example:

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

```
In:
Out:
```

(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.

Example:

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

```
In: R0
Out: R0 updated
```

(4) NewLine [&03] (Acorn MOS OSNEWL)

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

Example:

```
SWI NewLine
```

```
In:
Out:
```

(5) ReadC [&04] (Acorn MOS OSRDCH)

Read a character and validity from the terminal input. C set if an unusual character (e.g. Escape) is read.

Example:

```
SWI ReadC
BCS Escape

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

(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 Brazil Supervisor.

Example:

```
SWI CLI

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

(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.

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 Brazil the `OSBYTE &7F` call checks `R1 WORD` for being greater than 256 (range of "fast" handles is 257 to 511).

(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.

Example:

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

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

(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}).

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.
```

(10) Args [&09] (Acorn MOS OSARGS)

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

Example:

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

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

(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). Brazil does local buffering for Input Only and Output Only files by using handles in the range 256 to 511 - see *Open*.

Example:

```
LDR R1,handle
SWI BGet
BCS EndOfFile

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

(12) BPut [&0B] (Acorn MOS OSBPUT)

Write `R0b` to the file whose handle is in `R1`. Brazil does local buffering for Input Only and Output Only files by using handles in the range 256 to 511 - see *Open*.

Example:

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

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

(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. Brazil local buffering does not apply to the Multiple call.

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.
```

(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`. Brazil 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 Brazil supervisor prompt (NOT as a CLI call). The buffering produces the following times for BGet-ing a 64K file:

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

Example:

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

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

(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.

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.
```

(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 occured e.g. just after the SWI which called Brazil
`[R1,#4]` cardinal: error number provided with the error.
`R1+8`error string, terminated with a 0
Continue 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 Kernel 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 Kernel 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 Brazil's Supervisor (* prompt) or when a program is run using CLI.

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
```

(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 e.g. &10000 for 64K machine.
`R2` address of 5 bytes - the time the program started running

Example:

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

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

(18) Exit [&11]

Leave the program and return to the Brazil Supervisor * prompt. Brazil will print the time for which the program ran.

Example:

```
        SWI Exit

In: irrelevant
Out: never returns
```

(19) SetEnv [&12]

Set the program environment. This call should only be used to alter the overall environment for sub-programs.

`R0` address of the exit routine for Exit above to go to (or 0 if no change)
`R1` address of the 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.

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
```

(20) IntOn [&13]

Enable interrupts.

(21) IntOff [&14]

Disable interrupts.

(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 Kernel since Brazil is using the Tube Hardware and would thus become deadlocked. The solution is a CallBack on Kernel 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

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
```

(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`.

(24) BreakPt [&17]

Cause a break point trap. See BreakCtrl (next).

(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
```

(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  RN R13
TUBER  RN R12
SVCWK0 RN R11
SVCWK1 RN R10
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: Brazil Kernel 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
```

(27) UpdateMEMC [&1A]

The *MEMC* chip is a Write Only Memory *WOM* device. Brazil kernel maintains a software copy of it's current state and provides a kernel call to update MEMC, which of course (when present) is only accessible by non-user modes of ARM. To allow the programming of individual bits the call takes a field and a mask. The new MEMC value is:

```
newMEMC := (oldMEMC AND NOT R1) OR (R0 AND R1)
R0 := oldMEMC

In: new bits in field R0, field mask R1
Out: previous state of MEMC register
```

(28) 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:
```

(29) Mouse [&1C]

This kernel call reads the mouse. Values are 0..65535 or 0..7 (bit 0 is right down, bit 1 is middle down, bit 2 is left down).

```
In:
Out: R0 (X), R1 (Y), R2 (buttons)
```

(30) WriteI [&100]

Write the character contained in the bottom byte of the SWI call.

Example:

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

In:
Out:
```