# 7. Non-user modes

In the previous chapters, we have restricted ourselves to discussing the ARM while it is operating in user mode. For most purposes, this is all that is required. For example, large ARM programs such as the BBC BASIC interpreter manage to function entirely in user mode. There are times, however, when a program must execute in one of the other modes to work correctly. In this chapter, we discuss the characteristics of the non-user modes.

## 7.1 Extended programmer's model

### Register set

As described in Chapter Two, there are four modes in which the ARM may operate. The bottom two bits of R15 (called s1 and s0) determine the modes, as summarised below:

| **s1** | **s0** | **Mode** |
|---|---|---|
| 0 | 0 | 0 User (USR) |
| 0 | 1 | 1 Fast interrupt (FIQ) |
| 1 | 0 | 2 Interrupt (IRQ) |
| 1 | 1 | 3 Supervisor (SVC) |

When the ARM is in a non-user mode, its register set differs slightly from the user mode model. The numbering of the registers is identical, but some of the higher numbers refer to physically distinct registers in modes 1 to 3. The complete register model for all modes is shown overleaf. Each column shows the registers which are visible in mode 0, 1, 2 and 3 respectively.

The register names without a suffix refer to the user registers that we are used to dealing with. As the diagram shows, each of the non-user modes has at least two registers which are physically separate from the user mode ones. R14 is the link register, so all modes have their own link register, and R13 is traditionally used as the stack pointer, so each mode can have its own stack. FIQ mode has five additional private registers. These are provided so that important information may be stored in the processor for instant access when FIQ mode is entered.

In Acorn documentation, the term 'supervisor' modes is used to describe all non-user modes. We will adopt this convention for the rest of this chapter

Where the actual processor mode 3 is meant, the term SVC mode will be used.

Here is the extended programmer's model:

| USER | FIQ | IRQ | SVC |
|---|---|---|---|
| R0 | R0 | R0 | R0 |
| R1 | R1 | R1 | R1 |
| R7 | R7 | R7 | R7 |
| R8 | R8_FIQ | R8 | R8 |
| R9 | R9_FIQ | R9 | R9 |
| R10 | R10_FIQ | R10 | R10 |
| R11 | R11_FIQ | R11 | R11 |
| R12 | R12_FIQ | R12 | R12 |
| R13 | R13_FIQ | R13_IRQ | R13_SVC |
| R14 | R14_FIQ | R14_IRQ | R14_SVC |
| R15 | R15 | R15 | R15 |

## Instruction extensions

Although there are no instructions which can only be used in supervisor mode, the operation of some of the instructions already described in earlier chapters does alter slightly. These differences are covered for each instruction group below.

### Group one

Recall that in user mode, only the N, Z, V and C bits of the status register may be altered by the data manipulation instructions. The two interrupt mask bits, F and I, and the mode bits S0 and S1, may be read, but an attempt to alter them is ignored.

In supervisor mode, all eight bits of the status register may be changed. In fact, the very act of entering a supervisor mode may cause a change in the state of the four special bits. For example when a **SWI** instruction is used to call a supervisor mode routine, S0 and S1 are set to decimal 3, i.e. SVC mode, and IRQs are disabled by the I bit being set.

The easiest way to set the F, I, S0 and S1 bits to a required state is to use the **TEQP** instruction. Recall that the instruction:

performs an exclusive-OR of R15 (the PC and status flags) and the immediate value, but does not store the result anywhere. Because R15 is acting as a left-hand operand, only the PC bits (2 to 25) are used in the calculation, the status bits being set to 0. Furthermore, because the **P** option was specified after **TEQ**, the result of the exclusive-OR operation on bits 0, 1 and 26 to 31 of the operands is stored directly into the corresponding bits of R15. Thus the net result of the instruction is to store 1 bits in R15 where the **value** has one bits, and 0s where **value** was zero. You could view **TEQP** instruction as a special 'load status register' instruction of the form:

```
        LDSR    #value
```

As an example, suppose we are in SVC mode (S0 = S1 = 1) with interrupts disabled (I = 1), and want to move to FIQ mode (S0 = 1, S1 = 0) with both types of interrupts disabled (I = F = 1). The following instruction would achieve the desired result:

```
        TEQP    pc,#S0_bit + F_bit + I_bit
```

The following BASIC assignments would initialise the bit masks for the various status bits, such as **S0_bit**, used above:

```
    S0_bit = 1 << 1
    S1_bit = 1 << 2
    F_bit  = 1 << 26
    I_bit  = 1 << 27
    V_bit  = 1 << 28
    C_bit  = 1 << 29
    Z_bit  = 1 << 30
    N_bit  = 1 << 31
```

The **TEQP** instruction can only be used to store a given pattern of 1s and 0s into the status bits of R15. What we sometimes need is the ability to affect only some bits without altering others. This requires more work, as we have to read the current bits, then perform the desired operation. Suppose we want to disable both types of interrupts without altering the processor mode (i.e. F = I = 1, S0, S1 unchanged). Here is one way:

```
    MOV     temp,pc                     ;Load current flags
    ORR     temp,temp,#F_bit+I_bit      ;Set interrupt masks
    TEQP    temp,#0                     ;Move new flags into R15
```

This time, the current flags' states are loaded into a temporary register using the **MOV** instruction. Remember that to read the status part of R15, it must appear as a right-hand operand in a group one instruction. The **ORR** is used to set the I and F bits without altering the others. Finally, the **TEQP** sets the status bits from temp.

As a final example, suppose we want to return to user mode (S0 = S1 = 0) without altering the rest of the flags. We could use the **TST** instruction to clear S0 and S1, leaving the other flags unaltered:

```
    MOV     temp,#N_bit+V_bit+C_bit+Z_bit+I_bit+F_bit
    TSTP    temp,pc
```

The **MOV** loads temp with a mask of the bits that are to be unaltered, and the **TST** does an **AND** of this and the current status register, putting the result in R15, as the **P** option is specified.

## Group two

There is only one difference between using **LDR** and **STR** in supervisor mode and user mode. Recall that the post-indexed instructions of the form:

```
    LDR     <reg>,[<base>],<offset>
```

always use write-back, so there is no need to include a **!** at the end of the instruction to specify it. Well, the bit in the instruction code which would specify write-back is used for something else. It is ignored in user mode, but in supervisor mode it affects the state of a signal (called SPVMD, described below) which tells peripheral devices if the CPU is executing in supervisor or user mode.

Usually, when the ARM executes an **LDR** or **STR** in supervisor mode, the SPVMD signal tells the 'outside' world that this is a supervisor mode operation, so that devices like memory controllers can decide whether the requested data transfer is legal or not. If the **T** (for translate) option is given in an **STR**/**LDR** instruction, the SPVMD signal is set to indicate a user mode transfer, even if the CPU is really in supervisor mode. The **T** option comes after the optional byte-mode **B** flag, for example

```
    LDRBT   R0,[R1],#1
```

will load a byte into R0 addressed by R1, after which R1 will be incremented by one. Because **T** is present, the instruction will execute with **SPVMD** indicating user mode, even if the program is actually running in supervisor.

Note: The **T** option was included when it was envisaged that user and supervisor mode programs would have totally separate address spaces, with the former going through address translation and the latter not. As it turns out, the user address space enforced by the MEMC chip is actually a sub-set of the supervisor mode address space, so the **T** option is not usually needed. Remember also that its use is only valid with post-indexing, where the **!** option is not necessary.

## Group three

The **LDM** instruction provides the **^** option. If present, this specifies that if R15 is loaded by the instruction, the status bits will be overwritten. If the **^** is absent, the status bits of R15 will be unaffected by an **LDM**, even if the rest of R15 is loaded. In user mode, only N, Z, V and C may be overwritten; in supervisor mode, all the status bits are affected.

In supervisor mode, the **^** option is relevant even if R15 is not in the set of registers being loaded. In this situation, its presence indicates that the user-mode registers should be loaded, not the ones relevant to the current mode. So, in FIQ mode, for example, the instruction

```
LDMFD   sp,{R0-R14}^
```

will load the user-mode registers from memory. However, if R15 was in the list of registers to be loaded, the instruction would have its usual effect of loading the registers appropriate to FIQ mode set, and the **^** would indicate that the status bits of R15 are to be loaded.

For **STM**, there is no corresponding need for **^**, since all of R15 is always saved if specified in the register list. However, in supervisor mode, the presence of **^** in an **STM** is still relevant. Usually, an instruction like:

```
STMFD       sp,{R0-R15}
```

saves all of the registers which are visible in the current mode. For example, if the mode is SVC, then R0-R12, R13_SVC, R14_SVC and R15 are used. However, if **^** is specified, all registers saved are taken from the user bank, i.e. this instruction

```
STMFD   sp,{R0-R15}^
```

would cause R0-R15, all from the user bank of registers, to be saved.

Note that if write-back were specified in such an instruction, then the updated index register would be written back to the user bank instead of the appropriate supervisor mode bank. Therefore you should not specify write-back along with **^** when using the **STM** instruction from a supervisor mode.

## Group four

The only difference between using branches in user and supervisor mode is that, in **BL**, the link register (R14) appropriate to the current mode is used instead of the user R14. This is as expected, and does not require any special attention.

### Group five

When a SWI instruction is executed, the return address and flags are stored in R14_SVC. This means that when **SWI** is used from any mode other than SVC, no precautions are required to save R14 before the **SWI** is called. However, in SVC mode, executing a **SWI** will overwrite the current contents of R14_SVC. Therefore, this register should be preserved across calls to **SWI** if its contents are important.

To illustrate this, suppose a routine written to execute in user mode contains a call to the operating system's write character routine, but no other subroutine calls. It could be written thus:

```
;do some stuff
SWI    WriteC    ;Print char, R14_USR is preserved
;do some more stuff
MOV    pc,link   ;Return using R14
```

If the same routine is executed in SVC mode, the **SWI WriteC** would cause the return address in R14_SVC to be over written. The routine would have to be coded thus to work correctly:

```
STMFD  (sp)!,{link}  ;Save return address
;do some stuff
SWI    WriteC        ;Print char, R14_SVC is corrupt
;do some more stuff
LDMFD  (sp)!,{pc}    ;Return using stacked address
```

## Memory map

We have mentioned previously that some ARM systems are fitted with a memory controller (MEMC) which, amongst other things, translates addresses emanating from the CPU, performing a mapping from logical addresses to physical addresses. It also controls the access to other devices, for example ROM and I/O. MEMC controls the access of various parts of the memory map, restricting the operations which can be performed in user mode. The SPVMD signal produced by the ARM CPU tells the MEMC if the ARM is in supervisor mode or not. This enables MEMC to enforce a 'supervisor mode only' rule for certain locations.

Recall that the bottom 32M bytes of the address space is allocated to 'logical' RAM. This is divided into pages of between 8K and 32K bytes, up to 128 of them being present. Each page has a 'protection level' associated with it. There are four levels, 0 being the most accessible, and 3 being the most restricted. When the processor is in user mode, pages with protection level 0 may be read and written; pages at level 1 may be read only, and levels 2 and 3 are inaccessible. In supervisor mode, all levels may be read or written

MEMC, called OS mode. This allows read/write of levels 0 and 1 and reads-only of levels 2 and 3.)

The next 16M bytes of the memory map is set aside for physical RAM. This is only accessible in supervisor mode. The top 16M bytes is split between ROM and I/O. ROM may be read in any processor mode, but access to some I/O locations (e.g. those which control the behaviour of MEMC itself) is restricted to supervisor mode.

When an attempt is made to read or write an area of memory which is inaccessible in the current mode, an 'exception' occurs. This causes the processor to enter SVC mode and jump to a pre-defined location. There are various other ways in which user mode may be left, and these are all described below. Remember, though, that the memory scheme described in this section only refers to systems which use the Acorn MEMC, and might be different on your system.

# 7.2 Leaving user mode

There are several circumstances in which a program executing in user mode might enter one of the other modes. These can be divided roughly into two groups, exceptions and interrupts. An exception occurs because a program has tried to perform an operation which is illegal for the current mode. For example, it might attempt to access a protected memory location, or execute an illegal instruction.

Interrupts on the other hand, occur independently of the program's actions, and are initiated by some external device signalling the CPU. Interrupts are known as asynchronous events, because their timing has no relationship to what occurs in the program.

## The vectors

When an exception or interrupt occurs, the processor stops what it is doing and enters one of the non-user modes. It saves the current value of R15 in the appropriate link register (R14_FIQ, R14_IRQ or R14_SVC), and then jumps to one of the vector locations which starts at address &0000000. This location contains a **B**ranch instruction to the routine which will deal with the event.

There are eight vectors, corresponding to eight possible types of situation which cause the current operation to be abandoned. They are listed overleaf:

| Vector | Cause | I F | Mode |
|---|---|---|---|
| &0000000 | RESET | 1 1 | SVC |
| &0000004 | Undefined instruction | 1 X | SVC |
| &0000008 | Software interrupt (SWI) | 1 X | SVC |
| &000000C | Abort (prefetch) | 1 X | SVC |
| &0000010 | Abort (data) | 1 X | SVC |
| &0000014 | Address exception | 1 X | SVC |
| &0000018 | IRQ | 1 X | IRQ |
| &000001C | FIQ | 1 1 | FIQ |

The table shows the address of the vector, what causes the jump there, how the IRQ and FIQ disable flags are affected (X meaning it's unaffected), and what mode the processor enters. All events disable IRQs, and RESET and FIQ disable FIQs too. All events except the interrupts cause SVC mode to be entered.

Note that the FIQ vector is the last one, and the processor has no special use for the locations immediately following it. This means that the routine to handle a FIQ can be placed at location directly &1C, instead of a branch to it.

The following sections describe the interrupts and exceptions in detail. It is likely that most readers will only ever be interested in using the interrupt vectors, and possibly the `SWI` and undefined instruction ones. The rest are usually looked after by the operating system. However, fairly detailed descriptions of what happens when all the vectors are called are given. If nothing else, this may help you to understand the code that your system's OS uses to deal with them.

It is important to note that many of the routines entered through the vectors expect to return to the user program which was interrupted. To do this in a transparent way, all of the user's registers must be preserved. The PC and flags are automatically saved whenever a vector is called, so these can easily be restored. Additionally, all the supervisor modes have at least two private registers the contents or which are hidden from user programs. However, if the routine uses any registers which are not private to the appropriate mode, these must be saved and restored before the user program is restarted. If this is not done, programs will find register contents changing 'randomly' causing errors which are exceedingly difficult to track down.

# 7.3 RESET

The RESET signal is used to ensure that the whole system is in a wellsituations on most systems. Firstly, when power is first applied to the system, so-called power-on reset circuitry ensures that the appropriate levels are applied to the RESET signals of the integrated circuits in the computer. Secondly, there is usually a switch or button which may be used to RESET the system manually, should this be required.

On typical ARM systems, the MEMC chip, which contains the power-on reset circuitry, is used to control the resetting of the rest of the computer.

Upon receiving a RESET signal, the ARM immediately stops executing the current instruction. It then waits in an 'idle' state until the RESET signal is removed. When this happens, the following steps take place:

SVC mode is entered
R15 is saved in R14_SVC
The FIQ and IRQ disable bits are set
The PC is set to address &0000000

Although the program counter value when the RESET occurred is saved, it is not likely that an attempt will be made to return to the program that was executing. Amongst other reasons, the ARM may have been halfway through a long instruction (e.g. **`STM`** with many registers), so the affect of returning is unpredictable. However, the address and status bits could be printed by the operating system as part of 'debugging' information.

Likely actions that are taken on reset are initialisation of I/O devices, setting up of system memory locations, possibly ending with control being passed to some user mode program, e.g. BASIC.

# 7.4 Undefined instruction

Not all of the possible 32-bit opcodes that the ARM may encounter are defined. Those which are not defined to do anything are 'trapped' when the ARM attempts to decode them. When such an unrecognised instruction code is encountered, the following occurs:

SVC mode is entered
R15 is saved in R14_SVC
The IRQ disable bit is set
The PC is set to address &0000004

The program counter that is stored in R14_SVC holds the address of the instruction after the one which caused the trap. The usual action of the routine which handles this trap is to try to decode the instruction and point emulator interprets a range of floating point arithmetic instructions. Having done this, the emulator can jump back to the user's program using the PC saved in R15_SVC.

By trapping undefined instructions in this way, the ARM allows future expansions to the instruction set to be made in a transparent manner. For example, an assembler could generate the (currently unrecognised) machine codes for various operations. These would be interpreted in software using the undefined instruction trap for now, but when a new version of the ARM (or a co-processor) is available which recognises the instructions, they would be executed in hardware and the undefined instruction trap would not be triggered. The only difference the user would notice is a speed-up in his programs.

# 7.5 Software interrupt

This vector is used when a **SWI** instruction is executed. **SWI** is not really an exception, nor is it a proper interrupt since it is initiated synchronously by the program. It is, however, a very useful feature since it enables user programs to call routines, usually part of the operating system, which are executed in the privileged supervisor mode.

When a **SWI** is executed, the following happens:

SVC mode is entered
R15 is saved in R14_SVC
The IRQ disable bit is set
The PC is set to address &0000008

As with undefined instructions, the PC value stored in R14_SVC is one word after the **SWI** itself. The routine called through the **SWI** vector can examine the code held in the lower 24 bits of the **SWI** instruction and take the appropriate actions. Most systems have a well-defined set of operations which are accessible through various **SWI**s, and open-ended systems also allow for the user to add his or her own **SWI** handlers.

To return to the user's program, the **SWI** routine transfers R14_SVC into R15.

# 7.6 Aborts and virtual memory

An 'abort' is caused by an attempt to access a memory or I/O location which is out of bounds to the program that is currently executing. An abort is signalled by some device external to the ARM asserting a signal on the which controls all accesses to memory and I/O. Typical reasons for an abort occurring are attempts to:

write to a read-only logical RAM page
access physical RAM or I/O in user mode
access a supervisor mode-only logical page in user or OS mode
access an OS mode-only logical page in user mode
access a logical page which has no corresponding physical page

There are two types of abort, each with its own vector. The one which is used depends on what the ARM was trying to do when the illegal access took place. If it happened as the ARM was about to fetch a new instruction, it is known as a pre-fetch abort. If it occurred while the ARM was trying to load or store data in an **`LDR`**/**`STR`**/**`LDM`**/**`STM`** instruction, it is known as a data abort.

## Virtual memory

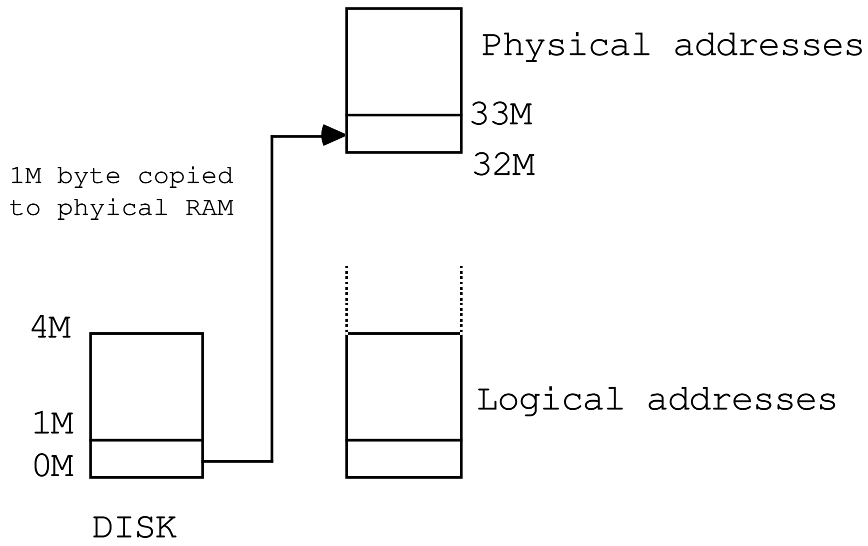
Except for programming errors, by far the most common cause of an abort is when the system is running what is known as virtual memory. When virtual memory is used, not all of the program is kept in physical RAM at once. A (possible majority) part of it is kept on a fast mass storage medium such as a Winchester disk.

Suppose a computer is fitted with 1M byte of RAM, and it is required that a program 'sees' a memory space of 4M bytes. This 4M bytes might be located in the first part of the logical memory map, from address &0000000 to &003FFFFF. On the Winchester disk, 4M bytes are set aside to represent the virtual address space of the program. Now as only 1M byte of RAM is available, only a quarter of this virtual address space can be physically stored in the computer's RAM. In the diagram overleaf, the first 1M byte of the disk area is loaded into physical RAM and mapped into the lowest megabyte of the logical address space.

As long as the program only accesses instructions and data which lie in the first megabyte of the logical address space, a mapping into physical RAM will be found by the MEMC and no problems will occur. Suppose, however, that the program attempts to access logical address 2,000,0000. There is no physical RAM which corresponds to this logical address, so MEMC will signal this fact to the processor using its ABORT line.

The abort handler program responds to an abort in the following way. First, it discovers what logical address the program was trying to access. It then allocates a page of physical RAM which can hold the page of virtual
memory corresponding to this address. The appropriate data is loaded in

from the disk, and the logical to physical address map in the MEMC adjusted so that when the processor next tries to access the location which caused the abort, the newly-loaded data will be accessed.




Now when a new page of virtual memory is loaded from disk, the previous contents of the area of physical memory used to store it must be discarded. This means that a range of addresses which used to be accessible will now cause an abort if an attempt is made to access them. Moreover, if that page contained data which has changed since it was first loaded from the disk, it must be re-written to the disk before the new page can be loaded. This ensures that the virtual memory on the disk is consistent with what is held in the RAM.

It is up to the software which deals with aborts to decide which page to discard when fetching a new page from the disk, and whether it needs to be written out before it is destroyed by the new data. (If the re-allocated page contain a part of the program, or read-only data, then it is not necessary to write it to the disk first, since the copy already stored there will be correct.) There are several algorithms which are used to decide the way in which pages are re-allocated in response to aborts (which are often called 'page faults'). For example, the so-called 'least recently used' algorithm will use the page which has not been accessed for the longest period of time, on the assumption that it is not likely to be required in the near future.

This may all seem incredibly slow and cumbersome, but in practice demand-paged virtual memory systems work well for the following reasons. Aborts are relatively infrequent as programs spend a lot of their time in small loops. ARM systems using MEMC have a fairly large page size (between 8K and 32K) so a program can spend a lot of its time in a single page without encountering 'missing' RAM. Additionally, virtual memory is often used on sharing the CPU for short time slots. While the relatively slow transfer of data between RAM and disk is taking place, another program can be using the CPU. This means that although one program might be held up waiting for a segment of its virtual address space to be loaded from disk, another program whose program and data are in physical RAM can proceed.

The subject of virtual memory is a complex one which is covered in a variety of text books. A good one is 'The Design of the Unix Operating System' by MJ Bach, published by Prentice-Hall.

## Pre-fetch aborts

When a pre-fetch abort occurs, the ARM completes the instruction(s) before the one which 'aborted'. When this instruction comes to be executed, it is ignored and the following takes place:

SVC mode is entered
R15 is saved in R14_SVC
The IRQ disable bit is set
The PC is set to address &000000C

The PC value saved is the one after the instruction which caused the abort. The routine which deals with the pre-fetch abort must perform some action, as outlined above, which will enable the instruction to be re-tried and this time succeed.

A simple single-tasking (one program at once) operating system running virtual memory might take the following steps on receiving a pre-fetch abort:

verify that it is a missing page problem (not access violation)
enable IRQs
find a suitable page of physical RAM
load the page corresponding to the required logical address
set-up MEMC to map the physical to logical page
re-try the instruction by jumping to R14_SVC minus 4

It is important to re-enable IRQs so that the normal servicing of interrupts is not disturbed while the new page is being loaded in. The third step may itself be quite involved, since a decision has to be made about which physical page is suitable for loading in the new section of the program and whether its current contents must be saved, as mentioned above.

## Data aborts

A data abort is a little more complex, since the ARM is halfway through the instruction by the time the abort occurs. If the instruction was an **LDR** or **STR**, it is abandoned and no registers are altered (in particular, the base register is unchanged, even if write-back was specified).

If the instruction was an **LDM** or **STM**, the instruction completes (though no registers are altered in an **LDM**), and the base register is updated if write-back was enabled. Note that if the base register was included in the list of registers to be loaded, it is not over-written as would normally be the case. The (possibly written-back) value is left intact. Then:

SVC mode is entered
R15 is saved in R14_SVC
The IRQ disable bit is set
The PC is set to address &0000010

This time, R14_SVC is set to the instruction two words after the aborted one, not one. The abort handler routine must take the following action. It must examine the aborted instruction to find out the type and address mode, and undo the affect of any adjustment of the base register due to write-back. It can derive the address of the data which caused the abort to occur from the base register, and perform a similar paging process to that described for pre-fetch aborts. It can then re-execute the instruction, by jumping to address R14_SVC minus 8 using an instruction such as:

```
SUB     pc,link,#8
```

The time taken to decode the instruction which caused the abort and perform the appropriate operations varies according to instruction type, number of registers (for **LDM**/**STM**), type of indexing (for **LDR**/**STR**) and whether write-back was enabled. Calculations made from typical abort-handler code result in times of between 20 s+41n-cycles for the best case and 121 s+36 n-cycles for the worst case. On an 8MHz ARM system, these translate into approximately 13.4us and 27.1us respectively.

# 7.7 Address exception

An address exception occurs if an attempt is made to access a location outside the range of the ARM's 26-bit address bus. This can be caused by the effective address (base plus any offset) of an **LDR**/**STR** instruction exceeding the value &3FFFFFF, or if the base register in an **LDM**/**STM** instruction contains a value greater than this.

Note that in the latter case, the exception will only occur if the base register is illegal when the instruction starts to execute. If it is legal for the first data transfer, and subsequently exceeds &3FFFFFF having been auto-incremented, no exception occurs. Instead the address 'wraps round' and subsequent loads or stores take place in the first few locations of the memory map.

Unlike the aborts described above, the address exception is detected internally by the ARM, and not by the assertion of a signal by some external device. Like data aborts, however, the incorrect instruction is abandoned or 'completed' as described above.

On detecting the address error, the ARM causes the following to take place:

SVC mode is entered
R15 is saved in R14_SVC
The IRQ disable bit is set
The PC is set to address &0000014

If it is required that the instruction be re-started after an address exception has been generated, the address in R14_SVC minus 4 can be used. However, there is usually not much point and the usual action is to enter a 'monitor' program which can be used to try to diagnose what went wrong.

# 7.8 IRQ

IRQ stands for Interrupt ReQuest, and is one of two interrupt inputs on the ARM. An external device signals to the ARM that it requires attention by asserting the IRQ line. At the end of every instruction's execution, the ARM checks for the presence of an IRQ. If an interrupt request has been made, and IRQs are enabled, the following happens:

IRQ mode is entered
R15 is saved in R14_IRQ
The IRQ disable bit is set
The PC is set to address &0000018

If IRQs are disabled, because the IRQ disable bit in the status register is set, the interrupt is ignored and the ARM continues its normal processing. Note that on initiating an IRQ routine, the ARM stes the IRQ, so further IRQs are disabled.

The routine handling the interrupt must take the appropriate action, which will result in the interrupting device removing its request. For example, a for reading. The IRQ routine, on discovering which device caused the interrupt, would read the data byte presented by the serial chip, and buffer it somewhere for later reading by a program. The action of reading a byte from the serial chip typically informs the device that its IRQ has been serviced and it 'drops' the interrupt request.

On the ARM, the interrupt disable bits and processor mode bits in the status register cannot be altered in user mode. This means that user mode programs typically execute with both types of interrupt enabled, so that the 'background' work of servicing interrupting devices can take place. However, it is sometimes desirable to disable all interrupts, and there is typically a **`SWI`** call provided by the operating system which allows the interrupt masks to be changed by user mode programs.

The following few paragraphs refer to the writing of both IRQ and FIQ routines.

Interrupt routines must be quick in execution, because while the interrupt is being serviced, the main program cannot progress. The Acorn IOC (I/O controller) chip provides some support for dealing with interrupts which makes their processing more efficient. For example, it provides several IRQ inputs so that many devices may share the ARM's single IRQ line. These inputs may be selectively enabled/disabled, and a register in the IOC may be read to find which devices at any time require servicing.

An interrupt routine usually has limitations imposed on what it can do. For example, it is undesirable for an interrupt handler to re-enable interrupts. If it does, another IRQ may come along which causes the handler to be called again, i.e. the routine is re-entered.

It is possible to write code which can cope with this, and such routines are known as re-entrant. Amongst other things, re-entrant routines must not use any absolute workspace, and must preserve the contents of all registers that they use (which all interrupt routines should do anyway). The first restriction means that the stack should be used for all workspace requirements of the routine, including saving the registers. This ensures that each re-entered version of the routine will have its own work area.

Unfortunately, it is common to find that operating system routines are impossible to write in a re-entrant fashion. This means that it is not possible to use many operating system routines from within interrupt service code. A common example is to find that a machine's output character routine is executed with interrupts enabled and is not re-entrant. (The reason is that in some circumstances, e.g. clearing the screen, the output routine might take several milliseconds, or even seconds, and it would be unwise to disable interrupts for this long.)

You should consult your system's documentation to find out the exact restrictions about using OS routines from within interrupt service code.

Interrupt routines should also be careful about not corrupting memory that might be used by the 'foreground' program. Using the stack for all workspace is one way to avoid this problem. However, this is not always possible (for example, if the IRQ routine has to access a buffer used by the foreground program). You should always endeavour to restrict the sharing of locations by interrupt and non-interrupt code to a bare minimum. It is very hard to track down bugs which are caused by locations being changed by something outside the control of the program under consideration.

To return to the user program, the IRQ routine subtracts 4 from the PC saved in R14_IRQ and places this in R15. Note that this saved version will have the IRQ disable bit clear, so as well as returning to the main program, the transfer causes IRQs to be re-enabled.

# 7.9 FIQ

FIQ stands for Fast Interrupt reQuest. A signal on the ARM's FIQ input causes FIQ mode to be entered, if enabled. As with IRQs, the ARM checks at the end of each instruction for a FIQ. If both types of interrupt occur at the same time, the FIQ is handled first. In this respect, it has a higher priority than IRQ. On responding to a FIQ, the ARM initiates the following actions:

FIQ mode is entered
R15 is saved in R14_FIQ
The FIQ and IRQ disable bits are set
The PC is set to address &000001C

Notice that a FIQ disables subsequent FIQs and IRQs, so that whereas a FIQ can interrupt an IRQ, the reverse is not true (unless the FIQ handler explicitly enables IRQs).

The term 'fast' for this type of interrupt is derived from a couple of its properties. First, FIQ mode has more 'private' registers than the other supervisor modes. This means that in order for a FIQ routine to do its job, it has to spend less time preserving any user registers that it uses than an IRQ routine would. Indeed, it is common for a FIQ routine not to use any user registers at all, the private ones being sufficient. Secondly, the FIQ vector was cleverly made the last one. This means that there is no need to have a branch instruction at address &000001C. Instead, the routine itself can start there, saving valuable microseconds (or fractions thereof).

To return to the user program, the FIQ routine subtracts 4 from the PC saved in R14_FIQ and places this in R15 (i.e. the PC). Note that this saved version will have the FIQ disable bit clear, so as well as returning to the interrupted program, the transfer causes FIQs to be re-enabled.