Application Note 25

Exception Handling on the ARM (including Thumb-aware Processors)

Document Number: ARM DAI 0025E
Issued: September 1996

ARM
Advanced RISC Machines

Proprietary Notice

Change Log

Issue	Date	By	Change
A	Oct 95	AP	Created
B	Mar 96	AP, JU, EH	Major edits and additions
C	Apr 96	EH	Corrections to ARM code listing in Section 12.2
D	Sept 96	JU	Minor correction to code listing in Section 5.5
E	Sept 96	JU	Minor edits

Table of Contents

1 Overview

During the normal flow of execution through a user program, the program counter increases sequentially through the address space, with branches to nearby labels or branch-with-links to subroutines.

Exceptions occur when this normal flow of execution is diverted, to allow the processor to handle events generated by internal or external sources. Examples of such events are:

- externally generated interrupts
- an attempt by the processor to execute an undefined instruction

It is necessary to preserve the previous processor status when handling such exceptions, so that execution of the original user program can resume once the appropriate exception routine has been completed.

The ARM recognises seven different types of exception, as shown below:

Exception	Description
Reset	Occurs when the CPU reset pin is asserted. This exception is only expected to occur for signalling power-up, or for resetting as if the CPU has just powered up. It can therefore be useful for producing soft resets.
Undefined Instruction	Occurs if neither the CPU nor any attached coprocessor recognises the currently executing instruction.
Software Interrupt (SWI)	This is a user-defined synchronous interrupt instruction, which allows a program running in User mode to request privileged operations which need to be run in Supervisor mode.
Prefetch Abort	Occurs when the CPU attempts to execute an instruction which has prefetched from an *illegal* address, ie. an address that the memory management subsystem has determined as inaccessible to the CPU in its current mode.
Data Abort	Occurs when a data transfer instruction attempts to load or store data at an illegal address.
IRQ	Occurs when the CPU's external interrupt request pin is asserted (LOW) and the I bit in the CPSR is clear.
FIQ	Occurs when the CPU's external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear.

Table 1: Exception types

1.1 The vector table

Exception handling is controlled by a *vector table.* This is a reserved area of 32 bytes at the bottom of the memory map with one word of space allocated to each exception type (plus one word currently reserved for handling address exceptions when the processor is configured for a 26-bit address space). This is not enough space to contain the full code for a handler, so the vector entry for each exception type typically contains a branch or load PC instruction to continue execution with the appropriate handler.

1.2 Use of modes and registers by exceptions

As a rule, an application runs in *User mode*, but the servicing of exceptions requires privileged (ie. non-User mode) operation. An exception changes the processor mode, and this in turn means that each exception handler has access to a certain subset of the banked registers:

- its own r13 or *Stack Pointer* (SP_<mode>)
- its own r14 or *Link Register* (LR_<mode>)
- its own *Saved Program Status Register* (SPSR_<mode>)

and, in the case of an FIQ, five more general-purpose registers (r8_FIQ to r12_FIQ)

Each exception handler must ensure that other registers are restored to their original contents upon exit. This can be done by storing the contents of any registers the handler needs to use onto its stack and restoring them before returning.

Note *You must ensure that the required stacks have been set up. If you are using Demon or ARMulator, this is done for you.*

1.3 Exception priorities

Several exceptions can occur simultaneously, and they are serviced in a fixed order of priority. Each exception is handled in turn before execution of the user program continues. However, it is not possible for all exceptions to occur concurrently. For instance, the undefined instruction and SWI exceptions are mutually exclusive as they both correspond to particular decodings of the current instruction.

Table 2: The exception vectors on page 4 shows the exceptions, their corresponding processor modes and handling priorities.

Placing the Data Abort exception above the FIQ exception in the priority list ensures that the Data Abort is actually registered before the FIQ is handled. The Data Abort handler is entered, but control is then passed immediately to the FIQ handler. Once the FIQ has been handled, control returns to the Data Abort Handler. This means that the data transfer error does not escape detection as it would if the FIQ were handled first.

Vector Address	Exception Type	Exception Mode	Priority (1=High, 6=Low)
0x0	Reset	svc	1
0x4	Undefined Instruction	undef	6
0x8	Software Interrupt (SWI)	svc	6
0xC	Prefetch Abort	abort	5
0x10	Data Abort	abort	2
0x14	*Reserved*	*Not applicable*	*Not applicable*
0x18	Interrupt (IRQ)	irq	4
0x1C	Fast Interrupt (FIQ)	fiq	3

Table 2: The exception vectors

2 Entering and Leaving an Exception

2.1 The processor's response to an exception

When an exception is generated, the processor:

1. copies the *Current Program Status Register* (*CPSR*) into the *Saved Program Status Register* (*SPSR*) for the mode in which the exception is to be handled
   This saves the current mode, interrupt mask and condition flags.
2. sets the appropriate CPSR mode bits:
   a) to change to the appropriate mode, also mapping in the appropriate banked registers for that mode.
   b) to disable interrupts.
   IRQs are disabled when any other type of exception occurs, and FIQs are also disabled when a FIQ occurs.
3. stores the *return address* (PC – 4) in LR_<mode>
4. sets the PC to the appropriate vector address
   This forces the branch to the appropriate exception handler.

Note *If the application is being run on a Thumb-aware processor, the processor's response is slightly different. See* ***12 Additional Considerations on Thumb-Aware Processors*** on page 28 *for more details.*

2.2 Returning from an exception handler

To return execution to the place where the exception occurred, the handler must:

- restore the CPSR from SPSR_<mode>.
- restore the PC using the return address stored in LR_<mode>

It carries out these two operations by performing a data processing instruction with the S flag set , and the PC as destination register. Each exception type requires the use of a different instruction—see ***3 The Return Address and Return Instruction*** on page 6 for details.

This method also applies to the Load Multiple instruction (using the ^ qualifier). So if a handler stores:

- all the working registers used when the handler is invoked
- the link register, modified to produce the same effect as the data processing instructions given in ***3 The Return Address and Return Instruction***

onto, say, a full descending stack, the exception handler may restore the registers and return in one instruction:

```
LDMFD sp!,{r0-r12,pc}^
```

3 The Return Address and Return Instruction

The actual value in the PC which causes a return from a handler depends on the exception type. Because of the way in which the ARM loads its instructions, when an exception is taken:

- the PC may or may not be updated to the next instruction to be fetched
- the return address may not necessarily be the next instruction pointed to by the PC

When loading the instructions needed for the execution of a program, the ARM uses a pipeline with a fetch, a decode and an execute stage. There will be one instruction in each stage of the pipeline at any time. The PC points to the instruction currently being *fetched*. Since each instruction is one word long, the instruction being decoded is at address (PC – 4) and the instruction being executed is at (PC – 8).

Note *See **12.1 The return address** on page 29 for details of the return address on Thumb-aware processors when an exception occurs in Thumb state.*

3.1 Returning from SWI and undefined instruction

The SWI and Undefined Instruction exceptions are generated by the instruction itself, so the PC is not updated when the exception is taken. Therefore storing (PC – 4) in LR_<mode> makes LR_<mode> point to the next instruction to be executed. Restoring the PC from the LR with

```
MOVS pc, lr
```

returns control from the handler

3.2 Returning from FIQ and IRQ

After executing each instruction, the CPU checks to see whether the interrupt pins are LOW and whether the interrupt disable bits in the CPSR are clear. As a result IRQ or FIQ exceptions are only generated after the PC has been updated, and consequently storing (PC – 4) in LR_<mode> causes LR_<mode> to point two instructions beyond where the exception occurred. When the handler has finished, execution must continue from the instruction prior to the one pointed to by LR_<mode>. The address to continue from is one word (or four bytes) less than that in LR_<mode>, so the return instruction is:

```
SUBS pc, lr, #4
```

3.3 Returning from prefetch abort

If the CPU attempts to fetch an instruction from an illegal address the instruction is flagged as invalid. Instructions already in the pipeline continue to execute until the invalid instruction is reached, at which point a prefetch abort is generated.

The handler gets the MMU to load the appropriate virtual memory locations into physical memory. Having done this, it must return to the offending address and reload the instruction, which should now load and execute correctly.

Because the PC will not have been updated at the time the prefetch abort was issued, LR_ABORT will point to the instruction following the one that caused the exception. The handler must therefore return to LR_ABORT – 4 using:

```
SUBS pc, lr, #4
```

3.4 Returning from data abort

When a load or store instruction tries to access memory, the PC has been updated. A stored value of (PC – 4) in LR_ABORT points to the second instruction beyond the address where the exception was generated. Once the MMU has loaded the appropriate address into physical memory, the handler should return to the original, aborted instruction so that a second attempt can be made to execute it. The return address is therefore two words (or eight bytes) less than that in LR_ABORT, making the return instruction:

```
SUBS pc, lr, #8
```

4 Installing an Exception Handler

Any new exception handler must be installed in the vector table. Once installation is complete, the new handler executes whenever the corresponding exception occurs.

This installation can take one of the following two forms.

Branch instruction

This is the simplest method of reaching the exception handler. Each entry in the vector table contains a branch to the required handler routine. However, this method does have a limitation: because the branch instruction only has a range of 32 MB, with some memory organizations the branch will be unable to reach the handler routine.

Load PC instruction

With this method, the PC is forced directly to the handler's address by:

1. storing the address of the handler in a suitable memory location (within 4 KB of the vector address)
2. placing an instruction in the vector which loads the PC with the contents of the chosen memory location

4.1 Installing the handlers at reset

If your application is standalone (ie. does not rely on the debugger or debug monitor to start program execution), it is possible to load the vector table directly from your assembler reset (or startup) code. If your ROM is at location 0x0 in memory, you can simply have a branch statement for each vector at the start of your code. This also could include the FIQ handler if it is running directly from 0x1c. See ***6 Interrupt Handlers*** on page 19.

In this case, the following section of code would set up the vectors if located in ROM at address zero. Note that you could substitute branch statements for the loads.

```
Vector_Init_Block
               LDR   PC, Reset_Addr
               LDR   PC, Undefined_Addr
               LDR   PC, SWI_Addr
               LDR   PC, Prefetch_Addr
               LDR   PC, Abort_Addr
               NOP                     ;Reserved vector
               LDR   PC, IRQ_Addr
               LDR   PC, FIQ_Addr

Reset_Addr     DCD   Start_Boot
Undefined_Addr DCD   Undefined_Handler
SWI_Addr       DCD   SWI_Handler
Prefetch_Addr  DCD   Prefetch_Handler
Abort_Addr     DCD   Abort_Handler
               DCD   0                 ;Reserved vector
IRQ_Addr       DCD   IRQ_Handler
FIQ_Addr       DCD   FIQ_Increment
```

If there is RAM at location zero, the vectors (plus the FIQ handler if required) have to be copied down from an area in ROM into the RAM. In this case you must use Load PC instructions, and copy the storage locations, in order to make the code relocatable.

The following piece of code copies down the vectors given above to the vector table in RAM:

```
MOV   R8, #0
ADR   R9, Vector_Init_Block
LDMIA R9!, {R0-R7}                  ;Copy the vectors (8 words)
STMIA R8!, {R0-R7}
LDMIA R9!, {R0-R7}                  ;Copy the DCD'ed addresses
STMIA R8!, {R0-R7}                  ;(8 words again)
```

For further information, refer to the chapter on *Writing Code for ROM* in the *ARM Software Development Toolkit Programming Techniques Manual* (ARM DUI 0021).

4.2 Installing the Handlers from C

Sometimes during development work, it is necessary to install exception handlers into the vectors directly from the main application. As a result the required instruction encoding must be written to the appropriate vector address. This can be done for both the branch and the load pc method of reaching the handler.

Branch method

The required instruction can be constructed as follows:

1. Take the address of the exception handler.
2. Subtract the address of the corresponding vector.
3. Subtract 0x8 to allow for the pipeline.
4. Shift the result to the right by two to give a word offset, rather than a byte offset.
5. Test that the top eight bits of this are clear, to ensure that the result is only 24 bits long (as the offset for the branch is limited to this).
6. Logically OR this with 0xea000000 (the opcode for the BAL instruction) to produce the value to be placed in the vector.

A C function which implements this algorithm is provided below. This takes the following arguments:

- the address of the handler
- the address of the vector in which the handler is to be to installed

The function installs the handler and returns the original contents of the vector. This result might be used to create a chain of handlers for a particular exception: see ***11 Chaining Exception Handlers*** on page 27 for further details.

```
unsigned Install_Handler (unsigned routine, unsigned *vector)
/* Updates contents of 'vector' to contain branch instruction */
/* to reach 'routine' from 'vector'. Function return value is */
/* original contents of 'vector'.*/
/* NB: 'Routine' must be within range of 32Mbytes from 'vector'.*/
{   unsigned vec, oldvec;
    vec = ((routine - (unsigned)vector - 0x8)>>2);
    if (vec & 0xff000000)
    {
         printf ("Installation of Handler failed");
         exit (1);
    }
    vec = 0xea000000 | vec;
    oldvec = *vector;
    *vector = vec;
    return (oldvec);
}
```

Code which calls this to install an IRQ handler might be:

```
unsigned *irqvec = (unsigned *)0x18;
Install_Handler ((unsigned)IRQHandler, irqvec);
```

In this case the returned, original contents of the IRQ vector are discarded.

Load PC method

The required instruction can be constructed as follows:

1. Take the address of the exception handler.
2. Subtract the address of the corresponding vector.
3. Subtract 0x8 to allow for the pipeline.
4. Logically OR this with 0xe59ff000 (the opcode for LDR PC,[PC,#offset] instruction) to produce the value to be placed in the vector.
5. Put the address of the handler into the storage location.

The following C routine implements this:

```
unsigned Install_Handler (unsigned *location, unsigned *vector)
/* Updates contents of 'vector' to contain LDR pc, [pc, #offset] */
/* instruction to cause long branch to address in ‘location’. */
/* Function return value is original contents of 'vector'.*/
{      unsigned vec, oldvec;
       vec = ((unsigned)location - (unsigned)vector-0x8) | 0xe59ff000
       oldvec = *vector;
       *vector = vec;
       return (oldvec);
}
```

Code which calls this to install an IRQ handler might be:

```
unsigned *irqvec = (unsigned *)0x18;
unsigned *irqaddr = (unsigned *)0x38; /* For example */
*irqaddr = (unsigned)IRQHandler;
Install_Handler (irqaddr,irqvec);
```

Again in this example the returned, original contents of the IRQ vector are discarded, but they could be used to create a chain of handlers: see ***11 Chaining Exception Handlers*** on page 27.

5 SWI Handlers

When the SWI handler is entered, it must establish which SWI is being called. This information is usually stored in bits 0–23 of the instruction itself, as shown in ***Figure 1: ARM SWI instruction***.

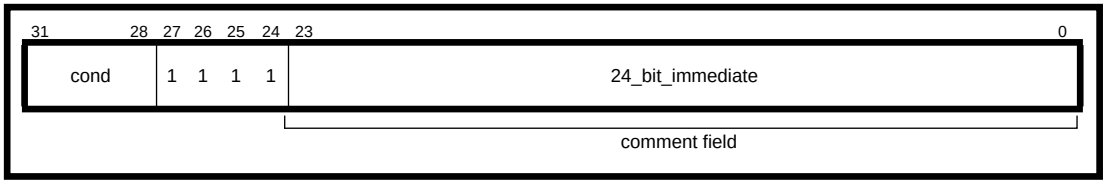

Figure 1: ARM SWI instruction

The top-level SWI handler will typically access the link register and load the SWI instruction from memory, and therefore has to be written in assembly language. (The individual routines that implement each SWI can be written in C if required.)

The handler must first load the SWI instruction that caused the exception into a register. At this point, LR_SVC holds the address of the instruction that follows the SWI instruction, so the SWI is loaded into the register (in this case r0) using

```
LDR r0, [lr,#-4]
```

The handler can then examine the comment field bits, to determine the required operation. The SWI number is extracted by clearing the top eight bits of the opcode:

```
BIC r0, r0, #0xff000000
```

These instructions can be put together to form a top level SWI handler:

```
   AREA TopLevelSwi, CODE, READONLY ; name this block of code
   EXPORT  SWI_Handler
SWI_Handler

   STMFD sp!,{r0-r12,lr}   ; store registers
   LDR   r0,[lr,#-4]       ; Calculate address of SWI
                           ; instruction and load it into r0
   BIC   r0,r0,#0xff000000 ; Mask off top 8 bits of
                           ; instruction to give SWI number
   ;
   ; Use value in r0 to determine which SWI routine to execute
   ;
   LDMFD sp!, {r0-r12,pc}^ ; Restore registers and return

   END                     ; mark end of this file
```

Note *See* ***12.2 Determining the processor state*** on page 30 *for an example of a handler which deals with both ARM and Thumb state SWI instructions.*

5.1 SWI handlers in assembly language

The easiest way to call the handler for the requested SWI is to make use of a jump table. If r0 contains the SWI number, the following code could be inserted into the top level handler given above, following on from the BIC instruction:

```
   ADR r2, SWIJumpTable
   LDR pc, [r2,r0,LSL #2]
SWIJumpTable
   DCD SWInum0
   DCD SWInum1
   ;
   ; DCD for each of other SWI routines
   ;
SWInum0                 ; SWI number 0 code
   B EndofSWI
SWInum1                 ; SWI number 1 code
   B EndofSWI
   ;
   ; Rest of SWI handling code
   ;
EndofSWI
   ; Return execution to top level SWI handler
   ; so as to restore registers and go back to user program
```

5.2 SWI handlers in C and assembly language

Once the top header, written in assembly language, has decoded which SWI is being called, the rest of the SWI handler can be written in C if required. The top-level assembly language code uses a BL instruction to the appropriate C function. Since the SWI number is loaded into r0 by the assembler routine, this is passed to the C function as the first parameter (in accordance with the ARM Procedure Call Standard). The function can then use this value in, for example, a `switch()` statement.

The following line can therefore be added to routine SWI_Handler:

```
   BL    C_SWI_Handler      ; Call C routine to handle the SWI
```

and the C function can be implemented as:

```
void C_SWI_handler (unsigned number)
{  switch (number)
   {     case 0 : /* SWI number 0 code */
                  break;
         case 1 : /* SWI number 1 code */
                  break;
          :
          :
         default : /* Unknown SWI - report error */
   }
}
```

The code implementing each SWI must be kept as short as possible and in particular should not call routines from the C library, because these can make many nested procedure calls which can exhaust the stack space and cause the application to crash. (Supervisor mode typically only has a small amount of stack space available to it.)

It is also possible to pass values in and out of such a handler written in C, provided that the top level handler passes the stack pointer value into the C function as the second parameter (in r1):

```
MOV   r1, sp            ; second parameter to C routine...
                        ; ...is pointer to register values.
BL    C_SWI_Handler     ; Call C routine to handle the SWI
```

and the C function is updated to access it:

```
void C_SWI_handler (unsigned number, unsigned *reg)
```

Therefore the C function can access the values contained in the registers at the time the SWI instruction was encountered in the main application code (see ***Figure 2: Accessing the supervisor stack***, below). It can read from them:

```
value_in_reg_0 = reg [0];
value_in_reg_1 = reg [1];
 :
 :
value_in_reg_12 = reg [12];
```

and also write back to the them:

```
reg [0] = updated_value_0;
reg [1] = updated_value_1;
 :
 :
reg [12] = updated_value_12;
```

causing the updated value to be written into the appropriate stack position, to be restored into the register by the top level handler.

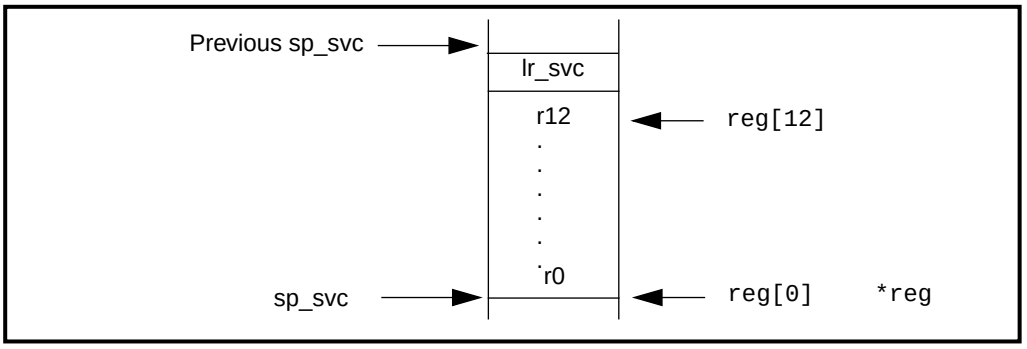

Figure 2: Accessing the supervisor stack

5.3 Using SWIs in Supervisor mode

When an SWI instruction is executed, Supervisor mode is entered as part of the exception handling process, the CPSR is copied into spsr_svc and the return address is stored in lr_svc, as described in ***2.1 The processor's response to an exception*** on page 5. As a result, if the processor is already in Supervisor mode, the LR and SPSR will be corrupted, so take precautions to ensure that the original values of the LR and SPSR are not lost.

This can happen if the handling of a particular SWI number causes a further SWI to be called. Where this occurs, it is possible to modify the SWI handler routine previously given to store not only lr onto the stack but also the spsr. This ensures that each invocation of the handler saves the necessary information to return to the instruction following the SWI that invoked it. The following code will do this:

```
SWI_Handler
   STMFD sp!,{r0-r12,lr}    ; store registers
   LDR   r0,[lr,#-4]        ; Calculate address of SWI
                            ; instruction...
                            ; ...and load it into r0
   BIC   r0,r0,#0xff000000  ; Mask off top 8 bits of
                            ; instruction to give SWI number
   MOV   r1, sp             ; second parameter to C routine...
                            ; ...is pointer to register values.
   MRS   r2, spsr           ; Move the spsr into a gp register.
   STMFD sp!, {r2}          ; Store spsr onto stack. This is
                            ; only really needed in case of
                            ; nested SWIs.
   BL    C_SWI_Handler      ; Call C routine to handle the SWI
   LDMFD sp!, {r2}          ; Restore spsr from stack into r2...
   MSR   spsr, r2           ; ... and restore it into spsr
   LDMFD sp!, {r0-r12,pc}^  ; Restore registers and return
   END                      ; mark end of this file
```

A further problem can be encountered if the second level of the handler is written in C. By default, the C compiler does not take into account that an inline SWI may overwrite the contents of lr. It is therefore necessary to instruct the C compiler to allow for this using the `-fz` compiler option. So if our C function was in the module `c_swi_handle.c`, the following command should be used to produce the object code file:

```
armcc -c -fz c_swi_handle.c
```

The `-fz` option can also be used with tcc.

Note *On processors which implement ARM Architecture 4 or 4T, it is also possible to make use of System mode to avoid this problem. See* ***13 System Mode*** on page 32 *for details.*

5.4 Calling SWIs from an application

The easiest way to call SWIs from your application code is to set up any required register values and call the relevant SWI in assembler language. For example:

```
MOV   r0, #65     ; load r0 with the value 65
SWI   0x0         ; Call SWI 0x0 with parameter value in r0
```

The SWI can be conditionally executed, as can all ARM instructions.

It is slightly more complicated to do this in C as it is necessary to map a call to a function onto each SWI called by the application code using the `__swi` compiler directive. This allows a SWI to be compiled in-line, without additional calling overhead, provided that:

- its arguments (if any) are passed in r0–r3 only
- its results (if any) are returned in r0–r3 only

The parameters are passed to the SWI as if the SWI were a real function call. However if there are between two and four return values, the compiler must be instructed that the return values are being returned in a structure, and the directive `__value_in_regs` used. This is because a struct valued function is usually treated as if it were a void function whose first argument is the address where the result structure should be placed. See the *ARM Software Development Toolkit Reference Manual* (ARM DUI 0020) for further details.

Example

This example shows a SWI handler which provides SWI numbers 0x0 and 0x1. SWI 0x0 takes four integer parameters and returns a single result, whereas SWI 0x1 takes a single parameter and returns four results. To declare these use:

```
struct four
{  int a, b, c d;
};

__swi (0x0) int calc_one (int,int,int,int);
__swi (0x1) __value_in_regs struct four calc_four (int);
```

They might then be called using:

```
void func (void)
{  struct four result;
   int single, res1, res2, res3, res4;
   single = calc_one (val1, val2, val3, val4);
   result = calc_four (val5);
   res1 = result.a;
   res2 = result.b;
   res3 = result.c;
   res4 = result.d;
}
```

Note `__swi` *can also be used with tcc.*

5.5 Calling SWIs dynamically from an application

In some circumstances it may be necessary to call a SWI whose number is not known until runtime. This situation might occur, for example, when there are a number of related operations that can be performed on an object, and each operation has its own SWI. In such a case the methods described above are not appropriate.

There are several ways of dealing with this. For example:

- constructing the SWI instruction from the SWI number, storing it somewhere and then executing it
- using a “generic” SWI which takes, as an extra argument, a code for the actual operation to be performed on its arguments. This generic SWI would then decode the operation and perform it.

The second mechanism can be implemented in assembler by passing the required operation number in a register, typically r0 or r12. The SWI handler can then be rewritten to act on the value in the appropriate register. Because some value has to be passed to the SWI in the comment field, it would be possible for a combination of these two methods to be used. For instance, an operating system might make use of only a single SWI instruction and employ a register to pass the number of the required operation. This would then leave the rest of the SWI space available for application-specific SWIs. This method can also be used if the overhead of extracting the SWI number from the instruction is too great in a particular application.

A mechanism is included in the compiler to support the use of r12 to pass the value of the required operation. Under the ARM Procedure Call Standard, r12 is the ip register that only has a dedicated role during function call. At other times it may be used as a scratch register, as it is in the code fragment below. The arguments to the generic SWI are passed in registers r0–r3 and values optionally returned in r0–r3 as described earlier. The operation number passed in r12 could be—but need not be—the number of the SWI to be called by the generic SWI.

Below is an C fragment that uses a generic, or *indirect* SWI:

```
__swi_indirect(0x80)
    unsigned SWI_ManipulateObject(unsigned operationNumber,
                      unsigned object,unsigned parameter);

unsigned DoSelectedManipulation(unsigned object,
                      unsigned parameter, unsigned operation)
{ return SWI_ManipulateObject(operation, object, parameter);
}
```

This produces the following code:

```
EXPORT DoSelectedManipulation
DoSelectedManipulation
  0x000000:  e1a0c002  .... : MOV      r12,r2
  0x000004:  ef000080  .... : SWI      0x80
  0x000008:  e1a0f00e  .... : MOV      pc,r14
```

It would also be possible to pass the SWI number in r0 from C using the `__swi` mechanism. For instance, if SWI 0x0 is used as the generic SWI and operation 0 was a character read and operation 1 a character write, the following could be set up:

```
__swi (0) char __ReadCharacter (unsigned op);
__swi (0) void __WriteCharacter (unsigned op, char c);
```

These could then be used in a more friendly fashion by defining the following:

```
#define ReadCharacter () __ReadCharacter (0);
#define WriteCharacter (c) __WriteCharacter (1, c);
```

However, using r0 in this way means that only three registers are available for passing parameters to the SWI. Normally, if more parameters need to be passed to a subroutine in addition to r0–r3, this can be done using the stack. However, stacked parameters are not easily accessible to a SWI handler, because they will typically exist on the User mode stack rather than the Supervisor stack employed by the SWI handler.

Note *In release 2.0 of the ARM Software Development Toolkit, tcc uses r3 rather than r12 to pass the value of the operation required when* `__swi_indirect` *is used. Take care when writing SWI handlers that make use of the register value in this way if you write SWIs that can be generated from both ARM and Thumb code. This situation may change in future releases of the toolkit. See section* ***12.2 Determining the processor state*** on page 30 *for further details.*

6 Interrupt Handlers

The ARM processor has two levels of external interrupt, FIQ and IRQ, both of which are level-sensitive active LOW signals into the core. For an interrupt to be taken, the relevant input must be LOW and the disable bit in the CPSR must be clear.

FIQs have higher priority than IRQs in two ways:

1. FIQs are serviced first when multiple interrupts occur.
2. Servicing a FIQ causes IRQs to be disabled, preventing them from being serviced until after the FIQ handler has re-enabled them (usually by restoring the CPSR from the SPSR at the end of the handler).

The FIQ vector is the last entry in the vector table (at address 0x1C) so that the FIQ handler can be placed directly at the vector location and run sequentially from that address. This removes the need for a branch and its associated delays, and also means that if the system has a cache, the vector table and FIQ handler may all be locked down in one block within it. This is important because FIQs are designed to service interrupts as quickly as possible. The five extra FIQ mode banked registers enable status to be held between calls to the handler, again increasing execution speed.

Note *An interrupt handler should contain code to clear the source of the interrupt.*

6.1 Interrupt handlers in C

You can set up C functions as simple interrupt handlers by using the special function declaration keyword `__irq`. This keyword:

- preserves all registers (excluding the floating-point registers) used by the function. Without `__irq`, the ARM Procedure Call Standard allows certain registers to be corrupted by a function call.
- exits the function by setting the PC to (LR – 4) and restoring the CPSR to its original value.

Note *C interrupt handlers cannot be produced in this way using tcc. The __irq directive is ignored by tcc.*

The example handler below reads a byte from location 0x80000000 and writes it to location 0x80000004:

```
void __irq IRQHandler (void)
{    volatile char *base = (char *) 0x80000000;
     *(base+4) = *base;
}
```

This produces the following code:

```
  EXPORT IRQHandler
IRQHandler
  0x000000: e92d0003 ..-. : STMDB       r13!,{r0,r1}
  0x000004: e3a00102 .... : MOV         r0,#0x80000000
  0x000008: e5d01000 .... : LDRB        r1,[r0,#0]
  0x00000c: e5c01004 .... : STRB        r1,[r0,#4]
  0x000010: e8bd0003 .... : LDMIA       r13!,{r0,r1}
```

```
  0x000014: e25ef004 ..^. : SUBS        pc,r14,#4
```

Compare this to the result of not using `__irq`:

```
  EXPORT IRQHandler
IRQHandler
  0x000000: e3a00102 .... : MOV         r0,#0x80000000
  0x000004: e5d01000 .... : LDRB        r1,[r0,#0]
  0x000008: e5c01004 .... : STRB        r1,[r0,#4]
  0x00000c: e1a0f00e .... : MOV         pc,r14
```

However `__irq` is only suitable for producing single-level C interrupt handlers. Functions declared in this manner cannot call subroutines written in C, because these could corrupt registers that have not been preserved by the top-level routine.

6.2 Interrupt handlers in assembly language

Interrupt handlers are often written in assembly language to ensure that they execute quickly. Below are some examples.

Single-channel DMA transfer

The following code is an interrupt handler which performs interrupt driven IO to memory transfers (soft DMA). The code is especially useful as a FIQ handler. It uses the banked FIQ registers to maintain state between interrupts, so this code is best situated at location 0x1c.

R8	points to the base address of the IO device that data is read from
IOData	is the offset from the base address to the 32-bit data register that is read and reading this register disables the interrupt
R9	points to the memory location to where that data is being transferred
R10	points to the last address to transfer to

The entire sequence for handling a normal transfer is four instructions; code situated after the conditional return is used to signal that the transfer is complete.

```
LDR    r11, [r8, #IOData] ; load port data from the IO device
STR    r11, [r9], #4      ; store it to memory: update the pointer
CMP    r9, r10            ; reached the end ?
SUBLES pc, lr, #4         ; no, so return
; Insert transfer complete code here
```

Byte transfers can be made by replacing the load instructions with load byte instructions, and transfers from memory to an IO device are made by swapping the addressing modes between the load instruction and the store instruction.

Dual-channel DMA transfer

This example is similar to the one in ***Single-channel DMA transfer*** on page 20, except that here two channels are being handled (which may be the input and output side of the same channel). Again, this code is especially useful as a FIQ handler, because it uses the banked FIQ registers to maintain state between interrupts. For this reason it is best situated at location 0x1c.

r8	points to the base address of the IO device that data is read from
IOStat	is the offset from the base address to a register that indicates which of two ports caused the interrupt
IOPort1Active	is a bit mask indicating if the first port caused the interrupt (otherwise it is assumed that the second port caused the interrupt)
IOPort, IOPort2	are offsets to the two data registers to be read. Reading a data register disables the interrupt for the corresping port.
r9	points to the memory location that data from the first port is being transferred to
r10	points to the memory location that data from the second port is being transferred to
r11 and r12	point to the last address to transfer to (r11 for the first port, r12 for the second)

The entire sequence to handle a normal transfer comprises nine instructions; code situated after the conditional return is used to signal that the transfer is complete.

```
LDR         r13, [r8, #IOStat]   ; load status register to
                                 ; find which port caused the
TST         r13, #IOPort1Active  ; interrupt ?
LDREQ       r13, [r8, #IOPort1]  ; load port 1 data
LDRNE       r13, [r8, #IOPort2]  ; load port 2 data
STREQ       r13, [r9], #4        ; store to buffer 1
STRNE       r13, [r10], #4       ; store to buffer 2
CMP         r9, r11              ; reached the end?
CMPLE       r10, r12             ; on either channel?
SUBLES      pc, lr, #4           ; return
; Insert transfer complete code here
```

Byte transfers can be made by replacing the load instructions with load byte instructions, and transfers from memory to an IO device are made by swapping the addressing modes between the conditional load instructions and the conditional store instructions.

Interrupt prioritization

This code dispatches up to 32 interrupt sources to their appropriate handler routines. Since it is designed for use with the normal interrupt vector (IRQ), it should be branched to from location 0x18.

External hardware is used to prioritize the interrupt and present the high-priority active interrupt in an IO register.

IntBase	holds base address of the interrupt controller
IntLevel	holds the offset of the register containing the highest-priority active interrupt
r13	is assumed to point to a small, full-descending stack

Interrupts are enabled after ten instructions (including the branch to this code).

The specific handler for each interrupt is entered after a further two instructions (with all registers preserved on the stack).

In addition, the last three instuctions of each handler are then executed with interrupts turned off again, so that the SPSR can be safely recovered from the stack.

Note *Application Note 30,* Software Prioritization of Interrupts *describes multiple source prioritization of interrupts using software, as opposed to using hardware as described here.*

```
   ; first save the critical state
   SUB    lr, lr, #4           ; adjust the return address
                               ; before we save it.
   STMFD  sp!, {lr}            ; stack return address
   MRS    r14, SPSR            ; get the SPSR ...
   STMFD  sp!, {r12, r14}      ; ... and stack that plus a
                               ; working register too.

   ; now get the priority level of the highest prioity active interrupt
   MOV    r12, #IntBase        ; get the interrupt controller's
                               ; base address
   LDR    r12, [r12, #IntLevel]; get the interrupt level (0 to 31)

; now read-modify-write the CPSR to enable interrupts
   MRS    r14, CPSR            ; read the status register
   BIC    r14, r14, #0x80      ; clear the I bit
                               ; (use 0x40 for the F bit)
   MSR    CPSR, r14            ; write it back to re-enable interrupts

   ; jump to the correct handler
   LDR    PC, [PC, r12, LSL #2]; and jump to the correct handler
                               ; PC base address points to this
                               ; instruction + 8
   NOP                         ; pad so the PC indexes this table

   ; table of handler start addresses
   DCD    Priority0Handler
   DCD    Priority1Handler
   DCD    Priority2Handler
        ........

Priority0Handler           .
   STMFD  sp!, {r0 - r11}      ; save other working registers
   ; insert handler code here
   ........
   LDMFD  sp!, {r0 - r11}      ; restore working registers (not r12).

   ; now read-modify-write the CPSR to disable interrupts
   MRS    r12, CPSR            ; read the status register
   ORR    r12, r12, #0x80      ; set the I bit
                               ; (use 0x40 for the F bit)
   MSR    CPSR, r12            ; write it back to disable interrupts

   ; now that interrupt disabled, can safely restore SPSR then return

   LDMFD  sp!, {r12, r14}      ; restore r12 and get SPSR
   MSR    SPSR, r14            ; restore status register from r14
   LDMFD  sp!, {PC}^           ; return from handler
Priority1Handler
   .....
```

Context switch

This code performs a context switch on the User mode process. The code is based around a list of pointers to *Process Control Blocks* (*PCBs*) of processes that are ready to run. The pointer to the PCB of the next process to run is pointed to by r12, and the end of the list has a zero pointer. Register 13 is a pointer to the PCB, and is preserved between timeslices (so that on entry it points to the PCB of the currently running process).

The code assumes the layout of the PCBs shown below in ***Figure 3: PCB layout***.

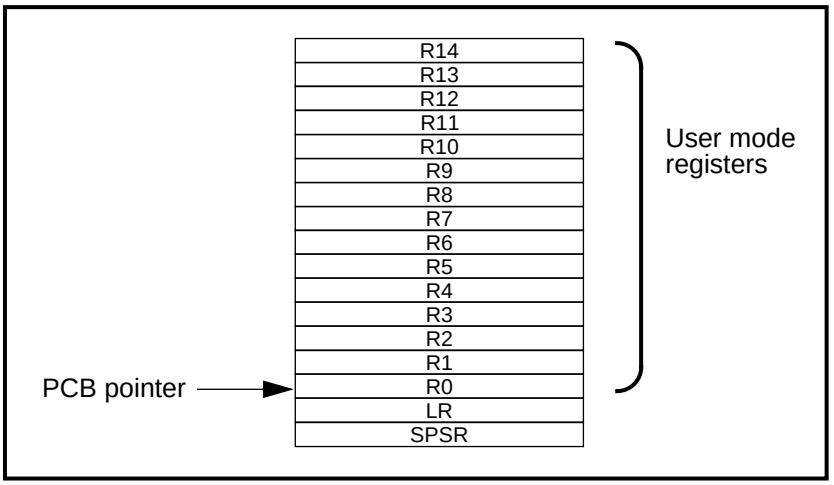

Figure 3: PCB layout

```
STMIA     r13, {r0 - r14}^      ; dump user registers above r13
MSR       r0, SPSR              ; pick up the user status
STMDB     r13, {r0, LR}         ; and dump with return address below
LDR       r13, [r12], #4        ; load next process info pointer
CMP       r13, #0               ; if it is zero, it is invalid
LDMNEDB   r13, {r0, LR}         ; pick up status and return address
MRSNE     SPSR, r0              ; restore the status
LDMNEIA   r13, {r0 - r14}^      ; get the rest of the registers
SUBNES    pc, r14               ; and return and restore CPSR
; insert “no next process code” here
```

7 Reset Handlers

The operations carried out by the Reset handler depend on the system for which the software is being developed. For example, it may:

- set up exception vectors—see ***4 Installing an Exception Handler*** on page 8 for details
- initialise stacks and registers
- initialise the memory system (if using an MMU)
- initialise any critical I/O devices
- enable interrupts
- change processor mode and/or state
- initialise variables required by C
- call the main application

For further information, refer to the chapter on Writing Code for ROM in the *ARM Software Development Toolkit Programming Techniques* guide (ARM DUI 0021).

8 Undefined Instruction Handlers

Instructions that are not recognised by the CPU are offered to any coprocessors attached to the system. If the instruction remains unrecognised, an undefined instruction exception is generated. It could be the case that the instruction is intended for a coprocessor, but that the relevant coprocessor—for example a Floating Point Accelerator—is not attached to the system. However, a software emulator for such a coprocessor might be available.

Such an emulator should:

1. attach itself to the undefined instruction vector, storing the old contents
2. examine the undefined instruction to see if it should be emulated

   This is similar way to the way in which a SWI handler extracts the number of a SWI, but rather than extracting the bottom 24 bits, the emulator must extract bits 27– 24, which determine whether the instruction is a coprocessor operation:

   - If bits 27–24 = b1110 or b110x, the instruction is a coprocessor instruction.
   - If bits 8–11 show that this coprocessor emulator should handle the instruction, the emulator should process the instruction and return to the user program.
   - Otherwise the emulator should pass the exception onto the original handler (or the next emulator in the chain) using the vector stored when the emulator was installed.

Once a chain of emulators is exhausted, no further processing of the instruction can take place, so the undefined instruction handler should report an error and quit. See ***11 Chaining Exception Handlers*** on page 27 for more information.

9 Prefetch Abort Handler

If the system contains no MMU, the Prefetch Abort handler can simply report the error and quit. Otherwise the address that caused the abort needs to be restored into physical memory. LR_ABORT points to the instruction at the address following the one that caused the abort, so the address that needs to be restored is at LR_ABORT – 4. The virtual memory fault for that address can be dealt with and the instruction fetch re-attempted. The handler should therefore return to the same instruction rather than the following one.

10 Data Abort Handler

If there is no MMU, the data abort handler should simply report the error and quit. If there is an MMU, the handler should deal with the virtual memory fault.

The instruction which caused the abort is at LR_ABORT – 8 (since LR_ABORT points two instructions beyond the instruction that caused the abort).

Three types of instruction can cause this abort:

1 Single Register Load or Store

If the abort has taken place on an ARM6 processor, the following holds true:

a) If the CPU is in early abort mode and writeback was requested, the address register will not have been updated.

b) If the CPU is in late abort mode and writeback was requested, the address register will have been updated. The change will need to be undone.

If the abort takes place on an ARM7 (or later) processor the address register will have been updated and the change will need to be undone.

2 Swap

There is no address register update involved with this instruction.

3 Load / Store Multiple

If writeback is enabled, the base register will have been updated as if the whole transfer had taken place. (In the case of an `LDM` with the base register in the register list, the processor replaces the overwritten value with the modified base value so that recovery is possible.) The original base address can then be recalculated using the number of registers involved.

In each case, the MMU can load the required virtual memory into physical memory: the MMU's *Fault Address Register* (*FAR*) contains the address which caused the abort. Once this is done, the handler can return and try to execute the instruction again.

11 Chaining Exception Handlers

Under some circumstances, it may be necessary to have more than one handler accessible for an exception type. Examples of this are:

- Undefined Instructions

  Your core application may have a default undefined instruction handler that simply reports an error and quits in some system-dependent manner. However, the system may have the option of having a hardware coprocessor attached which extends the instruction set (by providing, for example, floating-point operations). If the hardware coprocessor does not exist on a particular example of the system, a software emulation of it can be installed on top of the default undefined instruction handler in order to trap the coprocessor instructions.

- SWI usage by a Debug Monitor / RTOS

  It may be necessary for a debug monitor or RTOS to use some of the SWI instruction space for carrying out certain privileged operations. For instance the debug monitor, Demon, reserves SWIs in the range 0–255 for its own use in order to implement semihosted operations such as printing to the debugger screen.
  The SWI handler which deals with this might be installed as part of the debug monitor / RTOS's startup routine. However, the application itself might also need to make use of SWIs, in which case it must install its own handler, but cannot overwrite the original handler as the functions provided by that would then be lost.

In such cases you should store the original vector contents, so that if the latest handler installed in the vector cannot handle the cause of the exception, the exception can be passed onto the other handler. This means that it is possible to build up a chain of exception handlers. In effect the startup code of the Debug Monitor / RTOS hardwires the vector table instruction as described in ***4.1 Installing the handlers at reset*** on page 8. The application then needs to update the vector contents and store the old contents as described in ***4.2 Installing the Handlers from C*** on page 9.

Both routines given in ***4.2 Installing the Handlers from C*** return the old contents of the vector automatically. This value can be decoded to give:

- the offset for a branch instruction
  This can be used to calculate the location of the original handler and allow a new branch instruction to be constructed and stored at a suitable place in memory. If the replacement handler fails to handle the exception, it can branch to the constructed branch instruction, which in turn will branch to the original handler.

- the location used to store the address of the original handler
  If the application handler failed to handle the exception, it would then need to load the PC from that location.

In most cases, such calculations may not be necessary, as information on the debug monitor / RTOS's handlers should be available to the application writer. If so, the instructions required to chain in the next handler can be hardcoded into the application. The last section of the application's handler must check that the cause of

the exception has been handled. If it has, the handler can return to the application, If not, it will need to call the next handler in the chain.

Note *When chaining in a handler before a debug monitor handler, you must remove the chain when the monitor is removed from the system, then directly install the application handler.*

12 Additional Considerations on Thumb-Aware Processors

Note *This section only applies to processors that implement ARM Architecture 4T.*

This section describes the additional considerations which you must take into account when writing exception handlers suitable for use on Thumb-aware processors.

Thumb-aware processors use the same basic exception handling mechanism as non Thumb-aware processors, where an exception causes the next instruction to be fetched from the appropriate vector table entry.

The same vector table is used for both Thumb state and ARM state exceptions. An initial step must be added at the top of the exception handling procedure described in ***2.1 The processor's response to an exception*** on page 5. The procedure now reads:

1. Check the processor's state. If it is operating in Thumb state, switch to ARM state.
2. Copy the CPSR into SPSR_<mode>.
3. Set the CPSR mode bits.
4. Store the return address in LR_<mode>.

   See ***12.1 The return address*** on page 29 for further details.
5. Set the PC to the appropriate vector address.

The switch from Thumb state to ARM state in step 1 ensures that the ARM instruction installed at the appropriate vector (either a branch or a PC-relative load) is correctly fetched, decoded and executed. Execution then moves to a top-level veneer, also written in ARM code, which saves the processor status and any registers. The programmer then has two choices.

- Write the whole exception handler in ARM code.
- Make the top-level veneer store any necessary status, and then perform a BX (branch and exchange) to a Thumb code routine which handles the exception.

  Such a routine needs to return to an ARM code veneer in order to return from the exception, since the Thumb instruction set does not have the instructions required for restoring the CPSR from the SPSR.

This second strategy is shown in ***Figure 4: Handling an exception in Thumb state*** on page 29. See Application Note 27, *ARM/Thumb Interworking* for details of how to combine ARM and Thumb code in this manner.

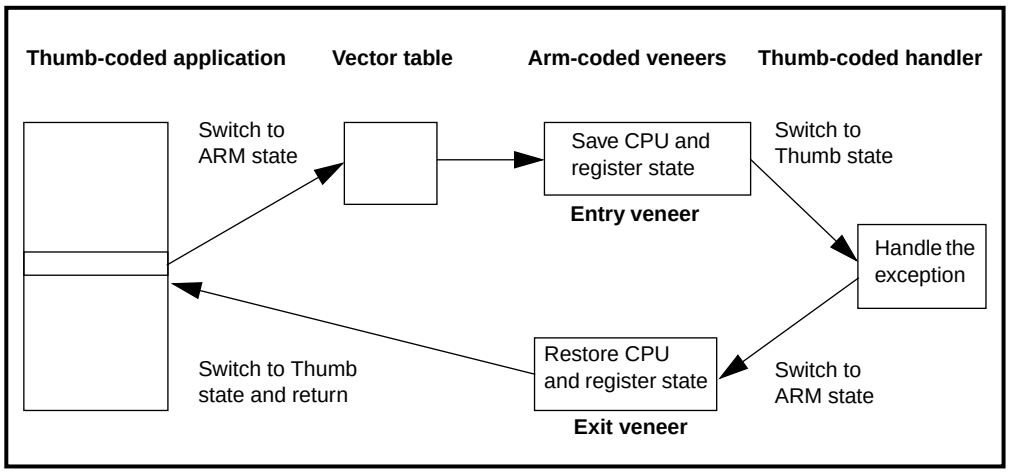

Figure 4: Handling an exception in Thumb state

12.1 The return address

If an exception occurs when in ARM state, the value stored in LR_<mode> is (PC – 4) as described in ***3 The Return Address and Return Instruction*** on page 6. However, if the exception occurs when in Thumb state, the processor automatically stores a different value for each of the exception types. This adjustment is required because Thumb instructions take up only a halfword, rather than the full word that ARM instructions occupy.

If this correction were not made by the processor, the handler would have to determine the original state of the processor, and use a different instruction to return to Thumb code rather than ARM code. By making this adjustment, however, the processor allows the handler to have a single return instruction which will return correctly, regardless of the processor's state (ARM or Thumb) at the time the exception occurred.

There follows a summary of the values to which the processor sets LR_<mode> if an exception occurs when the processor is in Thumb state.

SWI and Undefined Instruction handlers

The handler's return instruction (`MOVS pc,lr`) must reset the PC to the address of the next instruction to execute. This is at (PC – 2), so the value stored by the processor in LR_<mode> is (PC – 2).

FIQ and IRQ handlers

The handler's return instruction (`SUBS pc,lr,#4`) must reset the PC to the address of the next instruction to execute. Because the PC is updated before the exception is taken, the next instruction is at (PC – 4). The value stored by the processor in LR_<mode> is therefore PC.

Prefetch abort handlers

The handler's return instruction (SUBS `pc,lr,#4`) must reset the PC to the address of the aborted instruction. Because the PC is not updated before the exception is taken, the aborted instruction is at (PC – 4). The value stored by the processor in LR_<mode> is therefore PC.

Data abort handlers

The handler's return instruction (SUBS `pc,lr,#8`) must reset the PC to the address of the aborted instruction. Because the PC is updated before the exception is taken, the aborted instruction is at (PC – 6). The value stored by the processor in LR_<mode> is therefore (PC + 2).

12.2 Determining the processor state

In some circumstances, a handler needs to determine what state the processor was in when an exception occurred. This can be done by examining the T-bit within the SPSR. This might be needed, for example, in a SWI handler. Both the ARM and Thumb instruction sets contain SWI instructions. Handling ARM SWIs has already been discussed—see ***5 SWI Handlers*** on page 12.

When handling a Thumb SWI instruction, three extra things need to be taken into account:

- The address of the instruction is at (LR – 2), rather than (LR – 4).
- A halfword load rather than a word load is required to load the instruction.
- There are only eight bits available for the SWI number instead of the ARM's 24 bits.

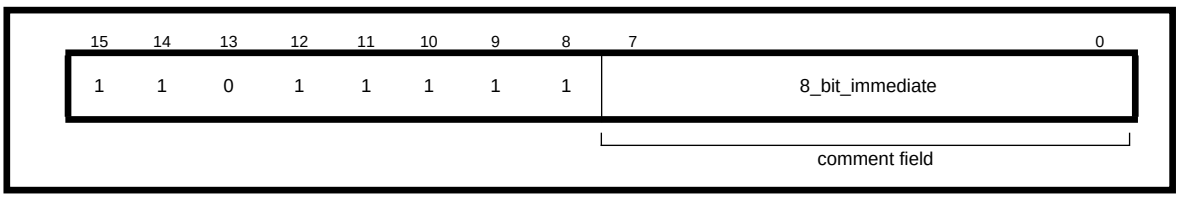

Figure 5: Thumb SWI instruction

Example

The following ARM code example handles a SWI from both sources:

```
T_bit  EQU             0x20     ; Thumb bit of CPSR/SPSR, ie bit 5.
       :
       :
SWIHandler
       STMFD  sp!, {r0-r12,lr};  ; Store the registers
       MRS    r0, spsr           ; move SPSR into general purpose
                                 ; register
       TST    r0, #T_bit         ; Test if bit 5 is set.
       LDRNEH r0,[lr,#-2]        ; T_bit set so load halfword (Thumb)
       BICNE  r0,r0,#0xff00      ; and clear top 8 bits of halfword
                                 ; (LDRH clears top 16 bits of word)
       LDREQ  r0,[lr,#-4]        ; T_bit clear so load word (ARM)
       BICEQ  r0,r0,#0xff000000  ; and clear top 8 bits of word

       ADR    r1, switable       ; Load address of the jump table
       LDR    pc, [r1,r0,LSL#2]  ; Jump to the appropriate routine
switable
       DCD    do_swi_1
       DCD    do_swi_2
       :
       :
do_swi_1
       ; handle the SWI
       LDMFD sp!, {r0-r12,pc}^ ; Restore the registers and return.
do_swi_2
       :
```

Notes

1. Each of the `do_swi_x` routines could carry out a switch to Thumb state and back again to reduce code density if required.
2. The jump table could be replaced by a call to a C function containing a `switch()` statement to implement the SWIs.
3. It would be possible for a SWI number to be handled differently depending upon which state it was called from.
4. The range of SWI numbers accessible form Thumb state can be increased by calling SWIs dynamically as described in ***5 SWI Handlers*** on page 12.

13 System Mode

Note *This section only applies to processors that implement ARM Architectures 4 and 4T.*

The ARM Architecture defines a User mode which has15 general-purpose registers, a PC and a CPSR. In addition to this mode there are five privileged processor modes, each of which have an SPSR and a number of registers that replace some of the 15 User mode general-purpose registers.

When a processor exception occurs, the current program counter is copied into the exception mode's r14 (lr), and the CPSR is copied in to the exception mode's SPSR. The CPSR is then altered in an exception-dependent way, and the PC is set to an exception-defined address to start the exception handler.

The ARM subroutine call instruction (BL) copies the return address into r14 before changing the PC, so the subroutine return instruction moves r14 to the PC (MOV PC, LR).

Together these actions imply that ARM modes which handle exceptions must ensure that they do not cause the same type of exceptions if they call subroutines, because if the exception occurs just after a BL, the subroutine return address will be overwritten with the exception return address.

In earlier versions of the ARM architecture this problem has been solved by either carefully avoiding subroutine calls in exception code, or changing from the privileged mode to User mode. The first solution is often too restrictive, and the second solution means the task may not have the privileged access it needs to run correctly.

ARM Architecture 4 introduces a new processor mode, called *System* mode, to help avoid this problem. System mode is a privileged processor mode that shares the User mode registers. Privileged mode tasks can run in this mode, and exceptions no longer overwrite the link register.

Note *System mode cannot be entered via an exception. An exception handler must cause it to be entered by modifying the CPSR.*

ENGLAND
Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
England
Telephone:+44 1223 400400
Facsimile:+44 1223 400410
Email:info@armltd.co.uk

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone:+49 (0) 89 608 75545
Facsimile:+49 (0) 89 608 75599
Email:info@armltd.co.uk

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone:+81 44 850 1301
Facsimile:+81 44 850 1308
Email:info@armltd.co.uk

USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone:+1 408 399 5199
Facsimile:+1 408 399 8854
Email:info@arm.com

World Wide Web Address: http://www.arm.com/