Application Note 30

Software Prioritization of Interrupts

Document Number: ARM DAI 0030A
Issued: March 1996

Proprietary Notice

Change Log

Issue	Date	By	Change
A	Oct 95	AP	Created

Table of Contents

Software Prioritization of Interrupts

1 Introduction

The ARM processor has two interrupt pins:

- IRQ for normal interrupts
- FIQ for fast interrupts

Application Note 25, *Exception Handling on the ARM* describe these pins in detail. However, many systems have more than two interrupt sources. In these systems interrupt sources are connected to an interrupt controller, which assert (set LOW) the appropriate pin (normally IRQ) on the ARM when one or more interrupts are detected.

In these multiple interrupt systems the current highest priority interrupt may be determined either by the hardware or the software. The interrupt handling software is simpler in the former case as it can read the current highest priority interrupt from the interrupt controller and then take the appropriate action. Application Note 25, *Exception Handling on the ARM,* contains an example of this.

Otherwise is left to the software to determine the current highest priority interrupt from a set of registers within the interrupt controller. An example of such a system is one based around *ARM's Reference Peripheral Specification (RPS)* (ARM DDI 0062). This application note describes some of the methods that can be used to carry out the prioritization of interrupts in software, using the RPS's interrupt controller. All relevant information from this specification is included within this application note, so a copy of the specification is not necessarily required.

2 The RPS Interrupt Controller

The interrupt controller defined by the RPS has the following registers (for both IRQ and FIQ unless otherwise stated):

Interrupt Raw Status

This register indicates which interrupt sources are currently active, prior to masking. Note that interrupt sources are active LOW and are inverted prior to being made available as the source register. Consequently a logic HIGH in a bit of the source register indicates that a particular interrupt source is active. This register is read-only.

Interrupt Enable

This register is used to mask the interrupt sources and defines which active sources generate an interrupt request to the processor. This register is read-only.

Interrupt Status

This register indicates which interrupt sources are currently active after masking is applied. A logic HIGH in a bit indicates that the appropriate interrupt is active and generates an interrupt in the processor. This register is read-only.

Interrupt Enable Set

This register is used to set bits in the interrupt enable register. When writing to this register, each data bit which is set high causes the corresponding bit in the interrupt enable register to be set. Data bits which are set LOW have no effect on the corresponding bit in the interrupt enable register. This register is write-only.

Interrupt Enable Clear

This register is used to clear bits in the interrupt enable register. Setting a data bit HIGH in this register causes the corresponding bit in the interrupt enable register to be cleared. Data bits which are set LOW have no effect on the corresponding bit in the interrupt enable register. This register is write-only.

Programmed IRQ Interrupt (IRQSoft)

A write to bit 1 of this register sets (if bit 1 is HIGH) or clears (if bit 1 is LOW) a programmed interrupt. This register is write-only. Its value may be determined by reading bit 1 of the IRQ raw status register. Bit 0 of this register is not used.

All of these registers are accessed by reading from an offset to a base address. This base address is not fixed and may be different on various systems. However the offset for any particular register from the base address is fixed, as shown in ***Table 1: Interrupt controller memory map***.

Address	Read Location	Write Location
IntBase + 0x00	IRQStatus	Reserved
IntBase + 0x04	IRQRawStatus	Reserved
IntBase + 0x08	IRQEnable	IRQEnableSet
IntBase + 0x0c		IRQEnableClear
IntBase + 0x10		IRQSoft
IntBase + 0x100	FIQStatus	Reserved
IntBase + 0x104	FIQRawStatus	Reserved
IntBase + 0x108	FIQEnable	FIQEnableSet
IntBase + 0x10c		FIQEnableClear

Table 1: Interrupt controller memory map

The RPS also defines some of the sources to the interrupt controller. The FIQ registers are one bit wide, located on bit 0, and the source is implementation specific. The IRQ registers have bits 1 to 5 defined, as shown in ***Table 2: Interrupt controller defined bits*** on page 4. Bit 0 is left available for the user (so that the FIQ source may also be routed to the IRQ controller in an identical bit position), as are bits 6 - 31.

.

Bit	IRQ Source
0	User defined
1	Programmed Interrupt
2	Comms Rx
3	Comms Tx
4	Timer 1
5	Timer 2

Table 2: Interrupt controller defined bits

The sample code given in this application note makes use of these registers. However, the principles illustrated may be adapted for another interrupt controller if the RPS is not being followed.

3 Simple Prioritization Example

The following sample code covers all the areas needed for prioritizing interrupts in software. It allows for six interrupt sources, each of which has a unique priority. The IRQ sources are assumed to match those given in ***Table 2: Interrupt controller defined bits***. The method of determining the current highest priority interrupt is kept simple to make the example as clear as possible.

3.1 Example description

This example first assigns the six priority levels to a single bit, so for example the priority 0 interrupt (the highest priority one—`B_PRI_0`) is assigned to the source mapped onto bit 5 of `IRQStatus`, and similarly for the other priority levels. If the priority levels change, these bit values need to be updated. The bit values for each priority are then used to calculate the value of the bit for each priority of interrupt. So, for example priority 0 interrupt is assigned to bit 5, which has a value of 1 shifted left by 5, (ie. 32), which is placed in `V_PRI_0`. These bit values are then used to calculate the combined bit values for all interrupts of lower (and equal) priority than a particular priority). Note that these combined values are stored in memory at the location `Lower_Priority`. The values stored in memory there need updating if the priorities change, as they are stored in bit order (0 to 5).

The actual code starts at the routine label `IRQHandler`. This first subtracts four from the link register and pushes it on the stack. This is because rather than using a SUBS `pc,lr,#4` instruction to return, the handler instead pops the return address directly off the stack into the program counter. Therefore the subtraction needs to be performed before the link register is pushed onto the stack. It is necessary to store the

link register in this way because once the initial processing is carried out, interrupts will be re-enabled. This means that another interrupt (of higher priority) could occur while the current one is still being handled, causing the link register to be overwritten.

It is also necessary to push the SPSR onto the stack before re-enabling interrupts, because this is also overwritten if another interrupt occurs while the current one is being processed. However the SPSR must be copied into an intermediate register and then pushed onto the stack. At the same time, the code also pushes some further working registers onto the stack.

At label `Detect_Highest` the code determines which is the highest priority bit set in the `IRQStatus` register. In this example, this is done in a very simple manner (other methods are examined later in this document). The code tests each bit, starting with the lowest priority bit, and if it is set, stores the corresponding bit number. Therefore when the code reaches the label `Disable_Lower`, the bit number for the highest priority interrupt is held in r11.

The code then loads in the `IRQEnable` register and the value stored in `Lower_Priority` for this particular bit (the value for the bits for lower priority interrupts than this one). These can then be logically ANDed together to give the value to be written to `IRQEnableClear` to mask out the lower priority interrupts that were previously enabled. This is necessary so that when a high priority source causes an interrupt while a lower one is being processed, only the interrupts of priorities between the two are disabled, and therefore re-enabled at the end of the processing of the higher priority one. Otherwise the higher priority one might re-enable all lower priority ones, meaning that a lower priority interrupt than the original might interrupt the original.

Once the `IRQEnable` register is updated in this manner, the IRQ disable bit in the CPSR can be cleared, so turning IRQs back on. When writing re-entrant interrupt handlers, the aim is to do the necessary minimum processing before turning IRQs back on. It is only when the IRQs are turned on again that the handler for the source of the current highest priority interrupt is called. This is done by directly loading the address of the handler routine for that bit from a table of handler addresses. This table must be updated if the names of any of these routines changes.

The example only gives the code for the handler `User_Interrupt`. Other handlers follow the same pattern:

1 All other working registers are pushed onto the stack.

2 The required operations needed to clear the cause of the interrupt are carried out.

3 Most of the working registers are popped off, except for r11, r12 and r14 which are still needed.

The final part of the handler is run with IRQs disabled, so that the SPSR can be restored from the stack safely:

4 The IRQ disable bit in the CPSR is set again.

5 The `IRQEnable` register then is updated, via the `IRQEnableSet` register, to unmask those interrupts disabled at `Disable_Lower`.

Note: in order to do this, r12 must have remained uncorrupted by the handler (or else pushed onto and popped off the stack).

6 Finally r11, r12 and the SPSR can be popped off the stack (SPSR via an intermediate register again) and then the original link register value popped off into the program counter (using ^ on the LDM instruction and so causing the required mode change by copying the SPSR into the CPSR).

3.2 Example code

```
    AREA      IRQHandler, CODE, READONLY

   ; Match interrupt priority to bit position
    ; ** This will need updating for each particular system **
B_PRI_0      EQU    5          ; Timer 2
B_PRI_1      EQU    2          ; Comms Rx
B_PRI_2      EQU    4          ; Timer 1
B_PRI_3      EQU    3          ; Comms Tx
B_PRI_4      EQU    0          ; User Defined
B_PRI_5      EQU    1          ; Programmed Interrupt

   ; match interrupt priority to bit value
   ; This is calculated using B_PRI_x.
V_PRI_0      EQU    1:LSL:B_PRI_0
V_PRI_1      EQU    1:LSL:B_PRI_1
V_PRI_2      EQU    1:LSL:B_PRI_2
V_PRI_3      EQU    1:LSL:B_PRI_3
V_PRI_4      EQU    1:LSL:B_PRI_4
V_PRI_5      EQU    1:LSL:B_PRI_5

   ; Calculate, for each priority of interrupt, which interrupts
   ; will need masking out when this particular priority occurs.
LOW_5        EQU    V_PRI_5
LOW_4        EQU    LOW_5 + V_PRI_4
LOW_3        EQU    LOW_4 + V_PRI_3
LOW_2        EQU    LOW_3 + V_PRI_2
LOW_1        EQU    LOW_2 + V_PRI_1
LOW_0        EQU    LOW_1 + V_PRI_0

   ; Create constants for accessing IRQ Interrupt Controller
IntBase             EQU       0x80000000         ; for example
IRQStatus           EQU       0x0
IRQRaw_Status       EQU       0x4
IRQEnable           EQU       0x8
IRQEnable_Set       EQU       0x8
IRQEnable_Clear     EQU       0xC
IRQSoft             EQU       0x10

   ; create IRQ disable bit constant
IRQDBit             EQU       0x80

   EXPORT    IRQHandler
   ; ***************************************
   ; * Start of actual code for IRQ Handler.*
   ; * This is the routine that should be   *
   ; * branched to from the IRQ vector.     *
   ; ***************************************
IRQHandler
   ; 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!, {r10,r11,r12,r14}    ; ... and stack that plus
                                        ; working registers too.

    ; now get the priority level of the highest priority active interrupt

Detect_Highest
    LDR       r14, =IntBase
    LDR       r10, [r14,#IRQStatus]

    ; test for each valid interrupt in turn, lowest -> highest priority]

    TST       r10, #V_PRI_5
    MOVNE     r11, #B_PRI_5     ; If Zero flag clear, interrupt active
    TST       r10, #V_PRI_4
    MOVNE     r11, #B_PRI_4     ; If Zero flag clear, interrupt active
    TST       r10, #V_PRI_3
    MOVNE     r11, #B_PRI_3     ; If Zero flag clear, interrupt active
    TST       r10, #V_PRI_2
    MOVNE     r11, #B_PRI_2     ; If Zero flag clear, interrupt active
    TST       r10, #V_PRI_1
    MOVNE     r11, #B_PRI_1     ; If Zero flag clear, interrupt active
    TST       r10, #V_PRI_0
    MOVNE     r11, #B_PRI_0     ; If Zero flag clear, interrupt active


    ; Thus at this point:
    ; - r14 contains address of IntBase
    ; - r11 contains the bit number of the highest priority interrupt
    ; - r10, r12 available for use

Disable_Lower
    LDR       r12, [r14, #IRQEnable]    ; Get currently enabled
                                        ; interrupts

    ADR       r10, Lower_Priority       ; Get address of lower
                                        ; priority values
    LDR       r10, [r10, r11, LSL #2]   ; Get value for lower
                                        ; priority interrupts

    ; Clear lower priority interrupts that are currently enabled.
    AND       r12, r12, r10
    STR       r12, [r14, #IRQEnable_Clear]

    ; now read-modify-write the CPSR to enable interrupts
    MRS       r14, CPSR                 ; read the status register
    BIC       r14, r14, #IRQDBit        ; clear the IRQ disable bit
    MSR       CPSR, r14                 ; write it back to
                                        ; re-enable interrupts

Goto_Handler
    ; jump to the correct handler
    LDR       PC, [PC, r11, 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
    ; ** This will thus need updating for each particular system **

    DCD       User_Interrupt     ; interrupt which maps to bit 0
    DCD       Prog_Interrupt     ; interrupt which maps to bit 1
    DCD       Rx_Interrupt       ; interrupt which maps to bit 2
    DCD       Tx_Interrupt       ; interrupt which maps to bit 3
    DCD       Timer1_Interrupt   ; interrupt which maps to bit 4
    DCD       Timer2_Interrupt   ; interrupt which maps to bit 5

    ; Store bit value for all lower priority interrupts.
    ; This is used when an interrupt occurs for a particular bit
    ; to disable lower priority ones, and is found by accessing
    ; Lower_Priority offset by (bit number) words.
    ; ** This will thus need updating for each particular system **
Lower_Priority
    DCD       LOW_4           ; bit 0 lower priority interrupt mask
    DCD       LOW_5           ; bit 1 lower priority interrupt mask
    DCD       LOW_1           ; bit 2 lower priority interrupt mask
    DCD       LOW_3           ; bit 3 lower priority interrupt mask
    DCD       LOW_2           ; bit 4 lower priority interrupt mask
    DCD       LOW_0           ; bit 5 lower priority interrupt mask


User_Interrupt
    STMFD     sp!, {r0 - r9}    ; save other registers (only those
                                ; needed!)
    ;
    ; Insert handler code here
    ; Remember that at some point the handler will need to “talk”
    ; directly to the interrupt source in order to clear the
    ; interrupt.
    ;
    LDMFD   r13!, {r0 - r10}        ; restore most of registers

    ; now read-modify-write the CPSR to disable interrupts
    MRS     r11, CPSR               ; read the status register
    ORR     r11, r11, #IRQDBit     ; set the IRQ Disable bit
    MSR    CPSR, r11              ; write it back to disable interrupts

    ; Now update IRQ Enable Register by re-enabling those lower
    ; priority interrupts disabled when this interrupt taken.
    ; Note that this relies on r12 still containing appropriate
    ; value. If this is not the case and the handler needs to use
    ; this register, then the value will need saving to and
    ; restoring from the stack during the handling of this particular
    ; interrupt.

    LDR       r11, =IntBase
```

```
    STR       r12, [r11, #IRQEnable_Set]

    ; Finally restore SPSR (can only be safely done with )
    ; interrupts disabled) and then return.

    LDMFD   sp!, {r11, r12, r14}   ; Restore rest of registers and
                                   ; get SPSR
    MSR     SPSR, r14              ; Restore status register from r14
    LDMFD   sp!, {pc}^             ; Return from handler using lr
                                   ; from stack, and
                                   ; restore spsr to cpsr

Prog_Interrupt
    ; as per User_Interrupt, but with own handler code.
Rx_Interrupt
    ; as per User_Interrupt, but with own handler code.
Tx_Interrupt
    ; as per User_Interrupt, but with own handler code.
Timer1_Interrupt
    ; as per User_Interrupt, but with own handler code.
Timer2_Interrupt
    ; as per User_Interrupt, but with own handler code.

    END
```

4 Other Methods of Determining the Highest Priority Interrupt

The method of determining the highest priority interrupt implemented in section 3.2 is not particularly sophisticated. As the number of valid interrupt sources increases, the number of instructions needed to find the highest priority interrupt also increases. However, the method is simple and does not obscure the other important aspects of the example. This also means that the time taken for interrupts to be re-enabled and for the correct handler for the particular interrupt to be reached is always constant, which is sometimes necessary. However, alternative methods may be used.

4.1 Terminating the search early

Rather than testing all of the possible interrupts from lowest to highest priority, it is possible to search down from the highest to lowest, branching out of the search once the highest priority active interrupt is identified. The branch could take place after setting the register that records the bit the interrupt is on, and could be targeted on `Disable_Lower`, meaning that only the test section has to change. However, this delays reaching the lowest priority interrupt, because each test now consists of three instructions rather than the two used in ***3 Simple Prioritization Example*** on page 4.

```
   TST       r10, #V_PRI_0
   MOVNE     r11, #B_PRI_0   ; If Zero flag clear, interrupt active.
   BNE       Disable_Lower   ; Skip rest of tests.
```

The test for the lowest priority interrupt could be replaced by simply setting the register (r11 in this case) as appropriate and falling through into `Disable_Lower`. If a test consisting of three instructions proves too slow (due to the time taken to detect the lowest priority interrupt) a slight variation on the above method could be implemented that branches to a further piece of code which sets the bit register and then branches onto `Disable_Lower`.

A further development of this method uses a BL instruction after each TST instruction. If the program counter's value is stored before the first TST instruction, when the program reaches `Disable_Lower` the difference between the stored program counter value and the current link register can be used to determine the highest priority interrupt in the following way. Another table in memory, `Table_B_PRI`, stores the bit value for each priority of interrupt (the table need only be a byte table as the values it stores lie between 0 and 31). The program counter value stored is the address of the first BL instruction. When `Disable_Lower` is reached, subtracting LR from this gives interrupt priority * 8 (as the difference between BLs is two instructions, or eight bytes). This value can then be divided by eight to give the offset which must be added to the address of Table_B_PRI in order to load in the appropriate bit number.

```
   ; Set up constants to obtain bit position given priority
Table_B_PRI
   DCB       B_PRI_0,B_PRI_1,B_PRI_2,B_PRI_2,B_PRI_4,B_PRI_5
   ALIGN
        :
        :
Detect_Highest
```

```
    ; now get the priority level of the highest priority
    ; active interrupt
    LDR       r14, =IntBase
    LDR       r10, [r14,#IRQStatus]
    MOV       r11, pc          ; pc points to first BLNE
    TST       r10, #V_PRI_0
    BLNE      Disable_Lower    ; Skip rest of tests.
    TST       r10, #V_PRI_1
    BLNE      Disable_Lower    ; Skip rest of tests.
     :
     :
    ; Thus at this point:
    ; - r14 contains link address from BL
    ; - r11 contains stored pc
    ; - r10, r12 available for use

Disable_Lower

    SUB    r11, r11, lr             ; r11 = (priority of interrupt) * 8
    LDR    r12, = Table_B_PRI       ; Get address of (byte) table
    LDRB   r11, [r12, r11, LSR #3]  ; Load bit number of the
                                    ; highest priority interrupt
    ADR    r10, Lower_Priority      ; Get address of lower
                                    ; priority values
    LDR    r10, [r10, r11, LSL #2]  ; Get value for lower priority
                                    ; interrupts

    LDR    r14, =IntBase            ; Get base address of interrupt
                                    ; controller
    ; now continue with rest of Disable_Lower as before
```

An alternative approach would be to branch directly to the required interrupt routine. However, each interrupt routine would need to include the interrupt masking and interrupt re-enable code that was included in `Disable_Lower` in ***3.2 Example code*** on page 7. It is also necessary for each routine to know which bit its interrupt is mapped onto. This can be done using another set of constants. Note, however, that this method will increase the code size of the application because it duplicates the masking and re-enabling code.

```
    ; Set up constants for bit in IRQStatus register that each
    ; interrupt is mapped onto.
User_Interrupt_Bit    EQU    0    ; interrupt which maps to bit 0
Prog_Interrupt_Bit    EQU    1    ; interrupt which maps to bit 1
Rx_Interrupt_Bit      EQU    2    ; interrupt which maps to bit 2
Tx_Interrupt_Bit      EQU    3    ; interrupt which maps to bit 3
Timer1_Interrupt_Bit  EQU    4    ; interrupt which maps to bit 4
Timer2_Interrupt_Bit  EQU    5    ; interrupt which maps to bit 5


Detect_Highest
    ; now get the priority level of the highest priority active
interrupt
    LDR       r14, =IntBase
```

```
    LDR       r10, [r14,#IRQStatus]

    ADR       r12, table_interrupts
    ; test for each valid interrupt in turn, highest -> lowest priority
    ; branch when found
    TST       r10, #V_PRI_0              ; Test priority 0 interrupt
    LDRNE     pc, [r12,#B_PRI_0,LSL #2]  ; Active if Zero flag clear
    TST       r10, #V_PRI_1              ; Test priority 1 interrupt
    LDRNE     pc, [r12,#B_PRI_1,LSL #2]  ; Active if Zero flag clear
    TST       r10, #V_PRI_2              ; Test priority 2 interrupt
    LDRNE     pc, [r12,#B_PRI_2,LSL #2]  ; Active if Zero flag clear
    TST       r10, #V_PRI_3              ; Test priority 3 interrupt
    LDRNE     pc, [r12,#B_PRI_3,LSL #2]  ; Active if Zero flag clear
    TST       r10, #V_PRI_4              ; Test priority 4 interrupt
    LDRNE     pc, [r12,#B_PRI_4,LSL #2]  ; Active if Zero flag clear
    TST       r10, #V_PRI_5              ; Test priority 5 interrupt
    LDRNE     pc, [r12,#B_PRI_5,LSL #2]  ; Active if Zero flag clear

    ; table of handler start addresses

table_interrupts
    DCD       User_Interrupt    ; interrupt which maps to bit 0
    DCD       Prog_Interrupt    ; interrupt which maps to bit 1
    DCD       Rx_Interrupt      ; interrupt which maps to bit 2
    DCD       Tx_Interrupt      ; interrupt which maps to bit 3
    DCD       Timer1_Interrupt  ; interrupt which maps to bit 4
    DCD       Timer2_Interrupt  ; interrupt which maps to bit 5

User_Interrupt
    ; Move bit number for this interrupt into r11.  Note that you would
    ; need to change this to appropriate constant for handlers for other
    ; sources.
    MOV       r11, #User_Interrupt_Bit

    ; Thus at this point:
    ; - r14 contains address of IntBase
    ; - r11 contains the bit number of the highest priority interrupt
    ; - r10, r12 available for use

    LDR r12, [r14, #IRQEnable] ; Get currently enabled interrupts
    ADR r10, Lower_Priority    ; Get address of lower priority values
    LDR r10, [r10, r11, LSL #2]; Get value for lower priority interrupts

    ; Clear lower priority interrupts that are currently enabled.
    AND r12, r12, r10
    STR r12, [r14, #IRQEnable_Clear]

    ; now read-modify-write the CPSR to enable interrupts
    MRS     r14, CPSR              ; read the status register
    BIC     r14, r14, #IRQDBit     ; clear the IRQ disable bit
    MSR     CPSR, r14              ;write it back to re-enable interrupts

    ; Now continue with main body of handler which remains as before
```

4.2 Priority groups

In some instances it is more useful to have groups of interrupts of the same priority than it is to grade interrupts individually by priority. For instance all interrupts caused by timers might have the same priority. In such a case, when several interrupts are active at one time within a group, the order of servicing is implementation-specific.

As with the previous examples, there are several ways of implementing this. The following example covers the important points.

For a particular system, the interrupt sources are grouped as follows (highest to lowest priority):

- Group 1 : Timer 1, Timer 2
- Group 2 : Comms Rx, Comms Tx
- Group 3 : User defined, Programmed Interrupt

The following program implements an interrupt handler for these groups. It does this by simply masking the `IRQStatus` register with each group mask, and storing the result. Once the result is non-zero, the following masks are ignored using conditional execution

The result from these mask instructions is then passed through `Find_LSB` which identifies the least significant bit in the result. The interrupt that maps onto this bit is handled during this iteration of the `IRQHandler`. The algorithm used relies on a binary search combined with a table lookup. The binary searches are used to reduce the table lookup index to four bits. The search proceeds down from identifying which halfword, byte or nibble contains the LSB. In each case, if it is the upper half, the lowest bit position of the upper half is added to the bit counter, and the upper half is shifted into the lower half. The search then continues by splitting the lower half into two in the same manner. Once there are only four bits left, the table lookup is used, and the value from the table to the bit counter. This then contains the bit position of the LSB of the currently active interrupts, and thus the bit position of the interrupt that this invocation of the handler will deal with.

The binary searches are actually only needed for interrupt sources mapped to more than four bits, with additional instructions required once interrupt sources are mapped on to more than four, eight and 16 bits. Thus, in this example, the halfword and byte identification parts of the search are commented out, as there are only six valid interrupts.

The code for `Disable_Lower` and the handlers themselves then remains the same as in the example in ***3.2 Example code*** on page 7.

```
    AREA      IRQHandler, CODE, READONLY

    ; Set up constants for bit in IRQStatus register that each ...
    ; ...interrupt is mapped onto.

User_Interrupt_Bit  EQU 0     ; interrupt which maps to bit 0
Prog_Interrupt_Bit  EQU 1     ; interrupt which maps to bit 1
Rx_Interrupt_Bit    EQU 2     ; interrupt which maps to bit 2
```

```
Tx_Interrupt_Bit        EQU  3 ; interrupt which maps to bit 3
Timer1_Interrupt_Bit    EQU  4 ; interrupt which maps to bit 4
Timer2_Interrupt_Bit    EQU  5 ; interrupt which maps to bit 5

   ; match interrupts to bit value
   ; This is calculated using bit value set above.
User_Interrupt_Value    EQU    1:LSL:User_Interrupt_Bit
Prog_Interrupt_Value    EQU    1:LSL:Prog_Interrupt_Bit
Rx_Interrupt_Value      EQU    1:LSL:Rx_Interrupt_Bit
Tx_Interrupt_Value      EQU    1:LSL:Tx_Interrupt_Bit
Timer1_Interrupt_Value  EQU    1:LSL:Timer1_Interrupt_Bit
Timer2_Interrupt_Value  EQU    1:LSL:Timer2_Interrupt_Bit

    ; Calculate masks for each priority group
Pri_Group_1  EQU    Timer1_Interrupt_Value + Timer2_Interrupt_Value
Pri_Group_2  EQU    Rx_Interrupt_Value + Tx_Interrupt_Value
Pri_Group_3  EQU    User_Interrupt_Value + Prog_Interrupt_Value

   ; Calculate, for each priority group, which interrupts will
   ; need masking out when an interrupt in this particular priority
   ; group occurs.
LOW_3        EQU    Pri_Group_3
LOW_2        EQU    LOW_3 + Pri_Group_2
LOW_1        EQU    LOW_2 + Pri_Group_1

User_Interrupt_Low            EQU     Low_3
Prog_Interrupt_Low            EQU     Low_3
Rx_Interrupt_Low              EQU     Low_2
Tx_Interrupt_Low              EQU     Low_2
Timer1_Interrupt_Low          EQU     Low_1
Timer2_Interrupt_Low          EQU     Low_1

    ; Create constants for accessing IRQ Interrupt Controller
IntBase             EQU       0x80000000         ; for example
IRQStatus           EQU       0x0
IRQRaw_Status       EQU       0x4
IRQEnable           EQU       0x8
IRQEnable_Set       EQU       0x8
IRQEnable_Clear     EQU       0xC
IRQSoft                       EQU      0x10

   ; create IRQ disable bit constant
IRQDBit      EQU    0x80

   EXPORT    IRQHandler
   ; ****************************************
   ; * Start of actual code for IRQ Handler.*
   ; * This is the routine that should be   *
   ; * branched to from the IRQ vector.     *
   ; ****************************************
IRQHandler
   ; 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!, {r10,r11,r12,r14}; ... and stack that plus
                                  ; working registers too.

    ; now get the priority group of the highest priority active
    ; interrupt
Detect_Highest
    LDR    r14, =IntBase
    LDR    r10, [r14,#IRQStatus]

    ; test for each valid interrupt in turn, lowest -> highest priority]
    AND    r11, r10, #Pri_Group_1
    ANDEQS r11, r10, #Pri_Group_2
    ANDEQS r11, r10, #Pri_Group_3

Find_LSB
    MOV    r12, #0              ; Initialise bit counter
                                ; The next three instructions
                                ; are only needed in cases where
                                ; there are more than 16 interrupts.
    MOVS   r10, r11, LSL #16    ; Find most significant half word
                                ; containing a nonzero bit (either
                                ; 0-15 or 16-31).
    ADDEQ  r12, r12, #16        ; If lsb is one of bits 16-31,
                                ; then increment the bit counter
    MOVEQ  r11, r11, LSR #16    ; and shift it into lower half of word.

    ; Now the top 16 bits of r11 are of no interest as the lsb will
    ; be located in bottom 16 bits.
    ; The next three instructions are only needed in cases where there
    ; are more than 8 interrupts.
    MOVS   r10, r11, LSL #24    ; Find most significant byte
                                ; containing a nonzero bit
                                ; (either 0-7 or 8-15).
    ADDEQ  r12, r12, #8         ; If lsb is one of bits 8-15,
                                ; then increment the bit counter...
    MOVEQ  r11, r11, LSR #8     ; and shift it into lower byte of word.

    ; Now the top 24 bits of r11 are of no interest as the lsb will be
    ; located in bottom 8 bits.

    MOVS   r10, r11, LSL #28    ; Find the most significant nibble
                                ; containing a nonzero bit(either
                                ; 0-3 or 4-7).
    ADDEQ  r12, r12, #4         ; If lsb is one of bits 4-7,
                                ; then increment the bit counter
    MOVEQ  r11, r11, LSR #4     ; and shift it into lower nibble
                                ; of word.

    ; Now the top 28 bits of r11 are of no interest as the lsb will be
    ; located in bottom 4 bits. So we can use lookup table to find which
    ; of those bits is the lsb set. Then we can add that to the
    ; current bit counter.
```

```
    AND    r10, r11, #0xf    ; Mask off bits 4-31.
    ADR    r11, LSB_Table    ; Generate address of the table.
    LDRB   r11, [r11, r10]   ; Load the bit value of lsb of bits 0-3
    ADD    r11, r11, r12     ; Add value to bit counter
    B      Disable_Lower LSB_Table
    ; The 0xff value should never be accessed as to reach this routine
    ; an interrupt must have occured and so one of the bits will be
    ; set.
    DCB    0xff, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0

    ; Thus at this point:
    ; - r14 contains address of IntBase
    ; - r11 contains the LSB from the highest priority active interrupts
    ; - r10, r12 available for use

Disable_Lower
    LDR    r12, [r14, #IRQEnable] ; Get currently enabled interrupts
    ADR    r10, Lower_Priority    ; Get address of lower priority values
    LDR    r10, [r10, r11, LSL #2]; Get value for lower priority
                                  ; interrupts

    ; Clear lower priority interrupts that are currently enabled.
    AND    r12, r12, r10
    STR    r12, [r14, #IRQEnable_Clear]

    ; now read-modify-write the CPSR to enable interrupts
    MRS     r14, CPSR           ; read the status register
    BIC     r14, r14, #IRQDBit  ; clear the IRQ disable bit
    MSR     CPSR, r14           ; write it back to re-enable interrupts

Goto_Handler
    ; jump to the correct handler
    LDR     PC, [PC, r11, 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    User_Interrupt       ; interrupt which maps to bit 0
    DCD    Prog_Interrupt       ; interrupt which maps to bit 1
    DCD    Rx_Interrupt         ; interrupt which maps to bit 2
    DCD    Tx_Interrupt         ; interrupt which maps to bit 3
    DCD    Timer1_Interrupt     ; interrupt which maps to bit 4
    DCD    Timer2_Interrupt     ; interrupt which maps to bit 5


    ; Store bit value for all lower priority interrupts. This is used
    ; when an interrupt occurs for a particular bit to disable lower
    ; priority ones, and is found by accessing Lower_Priority offset by
    ; (bit number) words.
Lower_Priority
    DCD    User_Interrupt_Low
    DCD    Prog_Interrupt_Low
    DCD    Rx_Interrupt_Low
```

```
    DCD    Tx_Interrupt_Low
    DCD    Timer1_Interrupt_Low
    DCD    Timer2_Interrupt_Low

User_Interrupt
           ; and other handlers as before

    END
```

The code given in `Find_LSB` could equally be replaced by other methods of handling the highest priority active interrupt(s). For instance, each mask instruction could be followed by a branch instruction in a similar way to the methods described in ***4.1 Terminating the search early*** on page 11. It would also be possible to have a branch to a handler for each priority of interrupt, which could then deal with the source as appropriate. In some cases it might not even be necessary for such a priority handler to know which source caused the interrupt.

5 Conclusion

This application note aims to illustrate the general principles needed for handling the prioritization of interrupts in software. The methods discussed for identifying which source actually generated an interrupt have been kept general so that they will work in any system, but it should be possible for them to be optimized for the particular system specification being used.

One particular optimization that may be possible is in the production of the addresses of the tables used in the examples. In the examples given here, a mixture of LDRs and ADRs have been used. Depending upon the amount of code in the actual handler produced for a system, it may be possible to replace all the table address generating LDRs with ADRs. See the *ARM Software Development Programming Techniques* guide (ARM DUI 0021) for further details of when to use ADRs as opposed to LDRs.

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/