Freescale Semiconductor
Application Note

AN2891
Rev. 0, 11/2004

Handling Multiple Interrupts on the MAC7100 Microcontroller Family

Brian LaPonsey
32-Bit Embedded Controller Division
East Kilbride, Scotland

This application note discusses concepts and methods for servicing interrupt requests from multiple sources on the MAC7100 family of microcontrollers.

Table of Contents

1 Abstract

The concept of asynchronous interrupts is introduced briefly, followed by a discussion of mode switching and the banked register structure of the ARM7TDMI-S™ core. The normal interrupt request (IRQ) and fast interrupt request (FIQ) exception types are introduced, with details of how the core responds to an IRQ signal. The differences between FIQ and IRQ are discussed.

A method for interrupt handling using a vector service routine with a jump table is presented. This leads into a discussion of the features of the MAC7100 Interrupt Controller (INTC) module. Several of the INTC registers and their purposes are explained, with example strategies for classifying interrupt sources into IRQ or FIQ types. The concepts of priority selection and masking are covered in detail.

A simple vector service routine (VSR) written in C is presented as an introductory example, with a discussion

of the limitations involved in writing such a routine in a high-level language. The construction and use of a jump table is covered in greater detail than before, with an example. The problem of context saving is discussed, including the complications that arise when using non-standard C language to save the scratch registers. An equivalent VSR written in assembler is presented which works around these issues, with each assembler statement's purpose explained in detail.

The topic of nested interrupts is introduced with an explanation of what they are, why they are used, and the added complexities involved with their implementation. The additional machine state information that must be saved before enabling nested interrupts is discussed in detail. A reentrant vector service routine is presented with assembler source code and detailed explanatory notes.

Three example projects to illustrate the developed concepts are presented in increasing order of complexity. Detailed notes on the most critical source code files are included in Appendix A.

2 Objective

This application note will discuss the details of using the ARM7TDMI-S core architecture and instruction set in conjunction with the MAC7100 Interrupt Controller module to efficiently handle interrupt processing. Examples will be provided with increasing levels of complexity, explaining at each level the steps necessary to effectively manage the interrupt environment and processor context. The reader should gain an understanding of the fundamentals of handling multiply-nested interrupt requests effectively on the MAC7100 device family.

It is assumed that the reader will have some prior experience in embedded system programming, and will be familiar with the ARM v4T architecture. These topics will be reviewed briefly, but a more in-depth treatment is available from the reference materials listed in the last section.

Example software is written for the GNU compiler and assembler. These examples are used only to illustrate the concepts developed in the document, and are neither expected nor intended to be incorporated as-written into a commercial product. System designers are encouraged to evaluate these examples and modify them accordingly, in order to satisfy the requirements of a particular application.

3 Background

The MAC7100 and its successors provide a cost-effective solution for the embedded designer requiring 32-bit performance with the option of using the compact 16-bit Thumb instruction set.

An embedded controller such as the MAC7100 must be able to respond quickly to changing input signals from its environment. In a highly-integrated System-on-Chip design, many such signals can be generated asynchronously by peripheral modules such as communications controllers, analog inputs or digital I/O.

Rather than constantly polling all the active peripherals to see if any require service, it is far more efficient to have each peripheral notify the core when attention is required. Under these circumstances, the core needs the ability to set aside its current task and handle these events as they occur. An event such as this is called an interrupt, and on the MAC7100, interrupts can be generated by over 200 different sources.

The MAC7100 family integrates a hardware interrupt controller (INTC) module to allow the core to handle these events. The Interrupt Controller (INTC) Module supports 64 interrupt sources organized as

16 priority levels, and defines a unique vector number for each source. Automatic hardware-implemented masking relieves the core of some of the work needed to support priority-based nesting of interrupts.

4 ARM7TDMI-S Core

4.1 Processor Modes

The ARM7TDMI-S core of the MAC7100 family is an implementation of the ARM v4T architecture. For details of the ARM v4T, its programming model and instruction set, please refer to the *ARM Architecture Reference Manual*.

The core of the MAC7100 has seven operating modes (see Table 1), and its programming model changes depending on which operating mode the core is in at the time. Six of these are collectively known as privileged modes, and are intended for servicing exceptions and accessing protected system resources. The operating mode switches automatically into one of these privileged modes when an exception occurs.

The state of the lowest 5 bits in the Current Program Status Register (CPSR) define the mode (see Figure 1). The mode changes automatically in response to an exception signal, but it also can be altered under software control by manually modifying the contents of this field. There are only seven valid modes on the ARM v4T architecture, so care should be taken to use only the bit combinations shown in Table 1. Failure to do so will place the processor in an unrecoverable state.

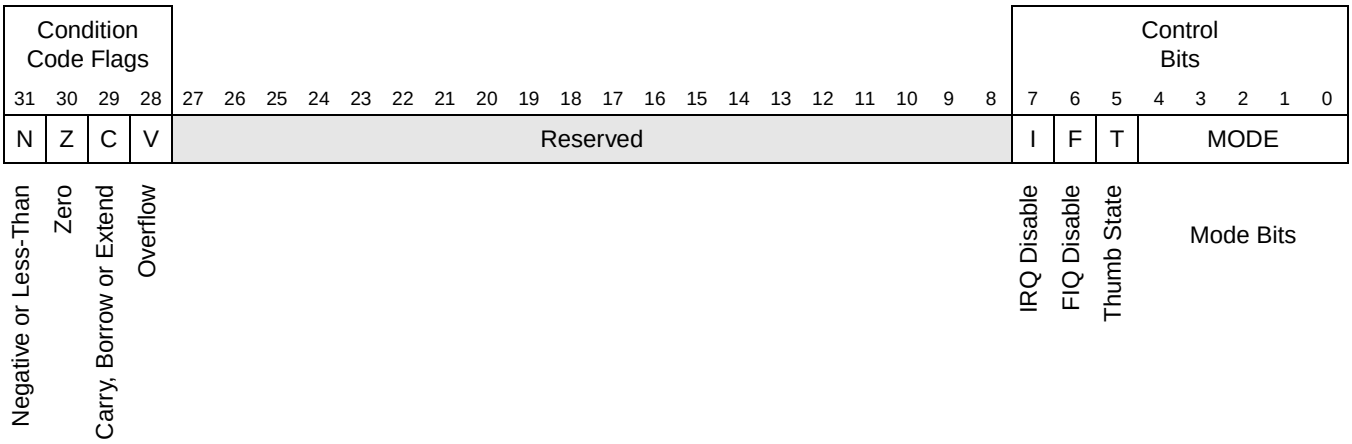


Figure 1. Current Program Status Register (CPSR)

Table 1. CPSR[MODE] Field Definitions

Mode	CPSR[4:0]	Description
User	10000	The least-privileged operating mode, useful when an operating system needs to restrict a running program's access to certain system resources. Many control and status registers on the MAC7100 family's peripherals are write-protected by default in user mode.
FIQ	10001	Fast Interrupt Request mode is used for time-critical events that must be handled as quickly as possible. High-speed communications reception or DMA access are examples of these.
IRQ	10010	Normal Interrupt Request mode is provided for general-purpose interrupt handling.
Supervisor	10011	Supervisor mode is intended as a protected mode for use by the operating system, and is the mode entered by the processor at reset.
Abort	10111	Abort mode is provided to implement virtual memory and protected memory schemes. This mode can be entered through a data abort (memory abort on data access) or prefetch abort (memory abort on instruction fetch).
Undef	11011	Undefined instruction mode is useful for software emulation of coprocessor functions.
System	11111	A privileged mode similar to Supervisor, but without the ability to access the banked registers. This provides an additional level of security when the operating system needs to perform system-related tasks, but does not want to incur the risk of accidentally corrupting the machine state preserved in the banked registers.

4.2 Banked Registers

The ARM7TDMI-S core implements 31 general-purpose registers, but only 16 of these are visible in User or System mode. The registers are organized into partially overlapping banks, and it depends on the operating mode which subset of these registers is accessible. Registers R0-R7 are always accessible, so these are said to be unbanked. If a program accesses an unbanked register, this access will always be to the same physical location no matter what the mode.

R8 to R14 are banked registers, implying that there is more than one occurrence of a register by that name. The actual physical location referred to by the name depends on the processor's operating mode at the time. Registers R8-R12 each have two banked versions, one for use in FIQ mode and the other for use in all other operating modes. To avoid confusion, the FIQ versions are usually referred to as R8_fiq - R12_fiq when it is necessary to distinguish them from the normal register set.

Registers R13 and R14 are also banked, and they each have six physical versions. One version is used for User and System mode, and the remaining five are reserved for use in each of the exception modes. To distinguish the versions from each other, it is customary to use the naming convention as described below, with the correct mode suffix corresponding to the exception type:

- usr (User and System mode)
- svc (Supervisor)
- abt (Abort)
- und (Undefined)
- irq (IRQ)
- fiq (FIQ)

Register R13 in all modes is normally used as a stack pointer, or SP. There are six banked versions of the SP, so it is important to initialize each one to a valid memory location at startup time. Failure to do so can result in some strange errors if exceptions occur, usually ending in a data abort when the banked SP tries to access memory from an undefined address.

Register R14 is used to hold subroutine return addresses, so it is sometimes called the link register (LR).

Register R15 is unbanked, and is also called the program counter. One unusual feature of the ARM architecture is that the program counter (PC) can be read and written just like any other general purpose register.

The CPSR (see Figure 1) is visible in all operating modes. Each exception mode has a Saved Program Status Register (SPSR) that is used to store the original CPSR contents at the time an exception occurs. User and System mode do not have access to a banked SPSR, because these modes cannot be entered by an exception.

Table 2 shows the entire register set, with each operating mode's programming model represented by a single vertical column.

Table 2. ARM7TDMI-S Core Programming Model

User	System	Supervisor	Abort	Undef	IRQ	FIQ
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq [1]
R9	R9	R9	R9	R9	R9	R9_fiq [1]
R10	R10	R10	R10	R10	R10	R10_fiq [1]
R11	R11	R11	R11	R11	R11	R11_fiq [1]
R12	R12	R12	R12	R12	R12	R12_fiq [1]
R13	R13	R13_svc [1]	R13_abt [1]	R13_und [1]	R13_irq [1]	R13_fiq [1]
R14	R14	R14_svc [1]	R14_abt [1]	R14_und [1]	R14_irq [1]	R14_fiq [1]
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc [1]	SPSR_abt [1]	SPSR_und [1]	SPSR_irq [1]	SPSR_fiq [1]

NOTES:
1. This is an alternative register specific to the exception mode, which replaces the normal register used by User or System mode.

5 Exceptions

An exception is a request for some action that is out of the ordinary, one that does not follow the normal program flow. Exceptions on the MAC7100 are caused by sources external to the core, and require a change in the normal sequence of execution. In essence, exception handling is a control mechanism

designed to deal with these situations. When viewed from this perspective, exceptions are not errors; they are requests for extra-ordinary service.

5.1 Exception Types

The ARM7TDMI-S core in the MAC7100 supports five types of exceptions:

- Fast interrupt requests (FIQ)
- Normal interrupt requests (IRQ)
- Memory aborts
- Attempted execution of undefined instructions
- Software interrupt instructions (SWI)

This application note will focus on IRQ and FIQ interrupt requests, because they are closely linked to the functioning of the MAC7100 Interrupt Controller module. Information about the other exception types and their uses can be found in the *ARM Architecture Reference Manual*, and from application notes available on the web and other sources.

5.2 IRQ: Normal Interrupt Requests

The ARM7TDMI-S core architecture provides two types of interrupt requests. The purpose of these is to allow an external peripheral to notify the core that it requires service. The first interrupt type is the Normal Interrupt Request, or IRQ. If an IRQ request is received by core, the current task will be suspended in order to service the request.

The aim of recognizing an interrupt is to allow the core to redirect the program flow to a routine that handles the situation, but this must be done in a manner that is transparent to the main program. In order to achieve this transparency, several items of the current context must be carefully saved. These items are referred to as the machine state, and will be restored after the interrupt handler completes. The first parts of the machine state that the core preserves in reaction to an exception are the PC and the CPSR.

In the case of a normal interrupt request, the banked "_irq" versions of the link register (R14_irq) and SPSR_irq are used for this purpose:

```
R14_irq         := PC+4             (save return link)
SPSR_irq        := CPSR             (save CPSR before exception)
```

The symbol **:=** means "is replaced by," and is used throughout this document. Note that the Link Register is replaced by PC+4, the address of the next instruction to be executed plus 4. This is a result of the ARM v4T architecture's 3-stage pipeline, and is explained more fully in the *ARM Architecture Reference Manual*.

Once the LR and CPSR are saved, the processor state can then safely be modified to facilitate interrupt handling. First, the processor is placed in ARM (32-bit instruction) state, and the mode is changed to enable the banked registers:

```
CPSR[4:0]       := 0b10010          (enter IRQ mode)
CPSR[5]         := 0                (put processor into ARM state)
```

The core then disables the recognition of further IRQ signals to avoid unintentional nesting, which would destroy the saved processor context. FIQs are left enabled during IRQ handling to allow nesting of FIQ over IRQ handlers:

```
CPSR[7]          := 1              (disable further IRQ requests)
```

Finally, the program flow is vectored to the memory address associated with IRQ exceptions. Execution continues at `0x0018` with the "_irq" banked register set enabled. [1]

```
PC               := 0x0018         (redirect flow to exception vector)
```

At this point, the program flow will enter a section of code to preserve any additional machine state necessary to restore the context later. The source of the interrupt must be identified, and whatever actions appropriate to handle it must be taken. After the IRQ handler has finished, it must restore the machine state and return control to the main program, which will continue undisturbed from the point at which it was interrupted.

Returning from an IRQ handler is slightly more problematic because of the banked registers and mode switching. The original, pre-exception CPSR must be restored from the SPSR_irq where it was saved. Doing this before restoring the PC would change the operating mode, making the saved PC in its banked link register (R14_irq) inaccessible.

If instead, the program counter is restored before the CPSR, this would return the program flow to its original point of execution with the CPSR in a corrupted state.

In short, both operations must take place, but neither can be done first. The solution is that they must take place simultaneously, and the ARM7TDMI-S core architecture has special instructions to accomplish this.

There are two ways to restore both the CPSR and PC simultaneously:

- use a data-processing instruction with the S bit set and the PC as the destination
- use a Load Multiple instruction that loads the PC, with the S bit set

The practice of using these special instructions to construct a working interrupt handler will be described in Section 8.2.

5.3 FIQ: Fast Interrupt Requests

The second type of interrupt signal available on the ARM v4T architecture is the Fast Interrupt Request, or FIQ. The FIQ works in much the same way as the IRQ, with the difference that in FIQ mode, there are five additional banked registers available for processing. See Table 2. These extra banked registers provide enough space so that a carefully-written FIQ handler can execute its assigned task entirely within the FIQ register set, R8_fiq - R14_fiq.

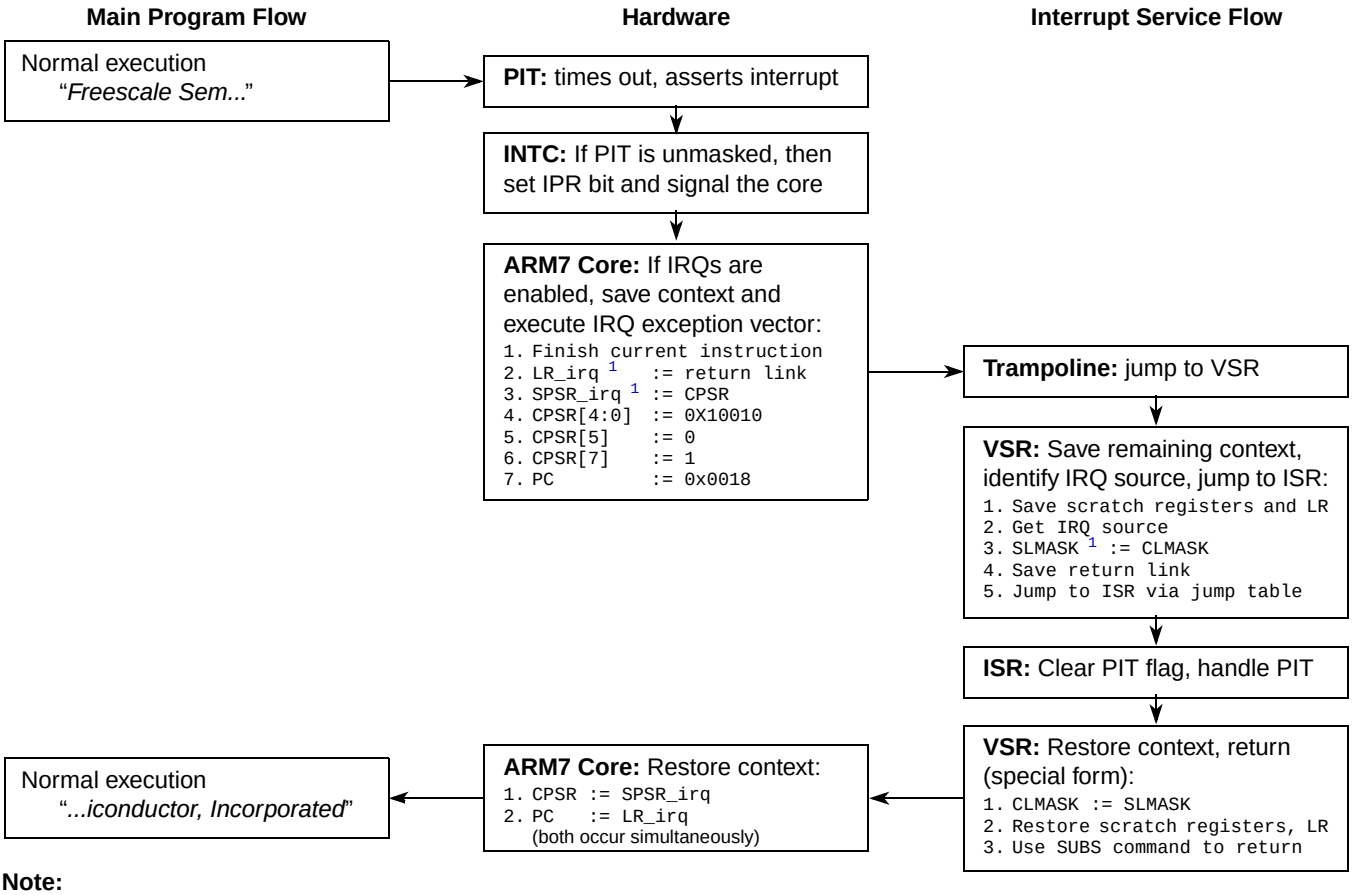
In this case, the FIQ handler would not disturb any part of the machine state except the FIQ state itself. Without the requirement to preserve any of the context, the FIQ handler can proceed with a substantially reduced CPU overhead, allowing FIQ exceptions to be serviced very quickly.

The ability of an FIQ routine to reliably begin execution within a well-defined time window is further enhanced by the fact that the recognition of a normal IRQ does not disable FIQ signals. An FIQ request always overrides all IRQ servicing, regardless of priority, ensuring that the FIQ receives immediate service.

1. This includes the R13_irq (stack pointer), so it is important to have initialized this banked register to a valid memory location in the system start-up routine.

6 Interrupt Handling

Figure 2 shows the sequence of events triggered when an IRQ exception interrupts a running program.



Note:

1. This object is being overwritten, and thus must be preserved to support nested interrupts.

Figure 2. Basic Interrupt Handling Process Flow

6.1 Trampoline

All of the steps prior to entering the interrupt handler are performed by the core automatically in response to the assertion of an IRQ signal. These actions are defined by the core architecture, and are taken independently of any software. On the ARM7TDMI-S, exceptions are always vectored to a specific location in a block of memory beginning at `0x0000` containing the exception vectors.

Table 3. ARM7TDMI-S Core Exception Table

Vector Address	Interrupt Source	Interrupt Type
0x0000	Reset	Reset
0x0004	Undef	Undefined instruction
0x0008	SWI	Software Interrupt
0x000C	Prefetch Abort	Abort
0x0010	Data Abort	Abort
0x0018	IRQ	Normal Interrupt
0x001C	FIQ	Fast Interrupt

As can be seen from Table 3, an IRQ exception will always be vectored to address `0x0018`. Whatever the contents of that word, they will be fetched and executed. It is up to the programmer to ensure that all of the exception vectors contain valid instructions.

A single word of memory isn't enough space to do any useful amount of processing, so typically an instruction is placed there to load the PC with another address, redirecting the flow elsewhere. This instruction is sometimes referred to as a trampoline.

It is common practice with ARM7 core-based systems for the trampoline to make an indirect jump using a short table of addresses. This table is customarily placed in memory immediately adjacent to the trampolines. [2] It contains the addresses of service routines that are designed to handle the exception vectors, so these are usually referred to as vector service routines, and the table as a VSR table.

6.2 Vector Service Routine

Each VSR, if implemented, must identify the cause of the exception and branch to an appropriate function to deal with the situation. Some extremely simple vector service routines are shown in the `__vector_service` section of Figure 3. These are merely trap functions that branch back to themselves. In other words, most of the VSRs in this example have not actually been implemented.

The only VSR that has been implemented in this example (other than the reset handler) is the one for IRQ exceptions. Notice that the `__VSR_table` in Figure 3 does not contain the address of the `IRQ_trap` function shown here. Instead, it contains the address of another function called `vsr_IRQ`. Assuming `vsr_IRQ` is a real vector service routine, its purpose will be to identify the cause of the interrupt and dispatch an appropriate interrupt service routine (ISR) to handle it.

In the simple situation where there is only one active interrupt source, the `vsr_IRQ` routine might also do the job of servicing the interrupt. There is no need to spend time identifying the source, if only one source is active. In this case, the VSR and ISR would be the same.

In a small application where there are a handful of interrupt sources, `vsr_IRQ` might be a test that looks sequentially at each source to see if it needs service. In larger systems like the MAC7100, there are far too many potential interrupt sources for this to be a viable strategy. A different technique must be employed, not only to simplify the code but also to ensure that each interrupt service routine can be dispatched in the same number of clock cycles as all the others.

2. On some systems, FIQ response is optimized by placing the entire FIQ handler at address `0x001C`. This removes the need for an intermediate vector service routine, and the main FIQ handler begins execution as soon as the FIQ signal is received. In this case, the table of vector addresses would come after the FIQ handler.

```
.code   32
        .section ".vectors","ax"
        .global __vectors

__vectors:                                  /* trampoline via table-jump to VSRs */
        ldr     pc, reset_addr              /* 0x0000 - reset vector */
        ldr     pc, UNDEF_addr              /* 0x0004 - undefined instr. vector */
        ldr     pc, SWI_addr                /* 0x0008 - sftw interrupt vector */
        ldr     pc, PABORT_addr             /* 0x000C - prefetch abort vector */
        ldr     pc, DABORT_addr             /* 0x0010 - data abort vector */
        .word   0x0                         /* 0x0014 - unused vector */
        ldr     pc, IRQ_addr                /* 0x0018 - IRQ vector */
        ldr     pc, FIQ_addr                /* 0x001C - FIQ vector */

        .global __VSR_table
__VSR_table:
reset_addr:     .word   reset               /* reset handler -- see crt0.S */
UNDEF_addr:     .word   UNDEF_trap          /* unimplemented -- see below */
SWI_addr:       .word   SWI_trap            /* unimplemented -- see below */
PABORT_addr:    .word   PABORT_trap         /* unimplemented -- see below */
DABORT_addr:    .word   DABORT_trap         /* unimplemented -- see below */
IRQ_addr:       .word   vsr_IRQ             /* the only real VSR - see crt0.S */
FIQ_addr:       .word   FIQ_trap            /* unimplemented -- see below */

        .global __vector_service
__vector_service:
UNDEF_trap:     b       UNDEF_trap          /* trap here on UNDEF exception */
SWI_trap:       b       SWI_trap            /* trap here on SWI exception */
PABORT_trap:    b       PABORT_trap         /* trap here on PABORT exception */
DABORT_trap:    b       DABORT_trap         /* trap here on DABORT exception */
IRQ_trap:       b       IRQ_trap            /* this trap is not used */
FIQ_trap:       b       FIQ_trap            /* trap here on FIQ exception */
```

Figure 3. Exception Vectors, VSR Table, and Trap Functions

One popular method for solving this is for `vsr_IRQ` to use a jump table. If the interrupt source can be resolved into a unique number, that number can be used as an index into a table of function pointers. This allows the VSR to calculate a jump to the correct ISR without the need to test each source individually. What is needed is an efficient way to get that source number, and this is one of the services provided by the MAC7100 Interrupt Controller (INTC).

Before discussing the design of a real VSR and the details of an ISR, it would be useful first to investigate the INTC module and some of the services it can provide to help implement these routines.

7 The MAC7100 Interrupt Controller Module

MAC7100 family of devices implement one Interrupt Controller Module. The purpose of the INTC is to organize, prioritize and control interrupt requests, and to increase overall system performance by reducing the time necessary to identify their sources. The INTC provides a number of hardware support services that help to implement advanced interrupt handling. See Chapter 10 of the *MAC7100 Microcontroller Family Reference Manual* (MAC7100RM).

7.1 INTC Features

The INTC supports 64 interrupt requests, maps them into 16 priority levels, and signals the ARM7TDMI-S core when a properly enabled, unmasked request is active. Each of the 64 sources has an interrupt control register (ICR*n*) that allows the designer to specify this mapping, linking the source number with the user-defined priority level.

The INTC also associates a unique vector number with each interrupt source. It provides the ability to mask individual interrupt sources according to the programmer's requirements. Hardware-assisted masking according to user-defined priority provides a substantial performance boost when nested interrupt servicing is part of the design.

Interrupt Force Registers provide the ability to generate interrupts within software for functional or debugging purposes, and Interrupt Pending Registers provide a map of active requests waiting to be processed.

7.2 The INTC Module Configuration Register (ICONFIG)

A highly-programmable module such as the INTC must be configured with the application's requirements in mind. The first step in this process usually involves the decision of how to classify the various priorities of interrupt into two main categories: IRQ and FIQ. This classification is made in the ICONFIG register. See Figure 4 and Section 10.5.1.4, "INTC Module Configuration Register (ICONFIG)" in the MAC7100RM.

Fast interrupts benefit from a total of seven banked registers, five more than the other exception modes. Fully utilizing these can substantially reduce the time it takes to respond to an interrupt request. The FIQ banked registers are not used by the main program, so they don't have to be saved and restored in an effort to preserve the context.

If the designer never expects to have more than one interrupt signal active at once, there is little reason to use the IRQ type. In this case, the ICONFIG[FIQDEF] field would be filled with zeroes. This maps all priority levels to the FIQ type, causing the INTC to always send an FIQ signal to the core when there is a pending request.

	7	6	5	4	3	2	1	0
R	0	0	EMASK	FIQDEF				
W								
Reset	1	0	1	0	0	0	0	0
Reg Addr	INTC Base + 0x001B							

Figure 4. INTC Interrupt Configuration Register (ICONFIG)

Another common situation is when there are many active interrupt sources, but one source sends requests at a much higher frequency than the others. In this case, it might make sense to program FIQDEF so that priorities 0-14 are mapped as IRQ, but priority 15 is an FIQ. Setting the high-frequency source to priority 15 allows the core to preempt any IRQ service routine to handle the FIQ request immediately.

FIQ requests are left enabled during IRQ handling. If the FIQ handler is carefully designed so that it uses only the seven IRQ-mode banked registers, then it can nest on top of an IRQ handler without ever needing to save or restore the context. This makes servicing the high-frequency FIQ source much more deterministic, because the worst-case latency does not have to include waiting for the IRQ handler to return.

The EMASK bit in the ICONFIG register enables a hardware-masking capability that allows interrupts to preempt other service routines of lower priority, even when the interrupt request is of the same type. This situation will be discussed in Section 9, Nested Interrupts. This allow interrupt nesting without using the FIQ type, leaving FIQ for other purposes. It is the responsibility of the system designer to ensure that interrupt-driven, safety-critical tasks are executed in a timely manner.

7.3 The IRQ Interrupt Acknowledge Register (IRQIACK)

The INTC associates a unique number with each source. For sources classified as IRQs, this number is available by reading the IRQ Interrupt Acknowledge Register (IRQIACK). As seen in Table 4, the interrupt sources are numbered from 0-63.

Table 4. INTC Interrupt Source-to-ICR*n* Assignments

Interrupt Source	Assigned ICR*n*	Interrupt Source	Assigned ICR*n*	Interrupt Source	Assigned ICR*n*	Interrupt Source	Assigned ICR*n*
eDMA0	0	eDMA Error	16	CAN_C Err./WU	32	eMIOS5	48
eDMA1	1	MCM SWT	17	CAN_D MB	33	eMIOS6	49
eDMA2	2	CRG	18	CAN_D MB14	34	eMIOS7	50
eDMA3	3	PIT1	19	CAN_D Err./WU	35	eMIOS8	51
eDMA4	4	PIT2	20	I^2C	36	eMIOS9	52
eDMA5	5	PIT3	21	DSPI_A	37	eMIOS10	53
eDMA6	6	PIT4 / RTI	22	DSPI_B	38	eMIOS11	54
eDMA7	7	VREG	23	eSCI_A	39	eMIOS12	55
eDMA8	8	CAN_A MB	24	eSCI_B	40	eMIOS13	56
eDMA9	9	CAN_A MB14	25	eSCI_C	41	eMIOS14	57
eDMA10	10	CAN_A Err./WU	26	eSCI_D	42	eMIOS15	58
eDMA11	11	CAN_B MB	27	eMIOS0	43	ATD_A, ATD_B	59
eDMA12	12	CAN_B MB14	28	eMIOS1	44	CFM	60
eDMA13	13	CAN_B Err./WU	29	eMIOS2	45	PIM	61
eDMA14	14	CAN_C MB	30	eMIOS3	46	$\overline{\text{IRQ}}$	62
eDMA15	15	CAN_C MB14	31	eMIOS4	47	$\overline{\text{XIRQ}}$	63

The source number is necessary for the VSR to quickly identify the source of the interrupt request, enabling it to compute an offset into the jump table that will redirect execution to the correct ISR (see Figure 2). The IAVECT field always contains a value between 63 and 127, so subtracting 64 from this value produces a source number between -1 and 63. A negative number indicates that the interrupt was spurious, i.e. there were no interrupts pending at the time the IRQIACK was read.

For the case in which several (or all) interrupt sources are classified as FIQ, there is an equivalent FIQIACK register that provides the same services as the IRQIACK.

When an interrupt request from one of these sources is asserted, reading the IRQIACK register (see Figure 5) will return the number of the source plus 64.

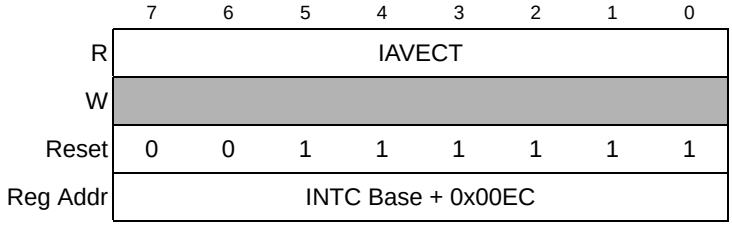


Figure 5. INTC IRQ Acknowledge Register (IRQIACK)

7.4 Setting the Priority of an Interrupt Source

The INTC provides the system designer with the ability to select a priority for each interrupt source. A bank of 64 Interrupt Control Registers (ICR*n*) is provided for this purpose. These 8-bit registers are individually assigned to each of the 64 interrupt sources as detailed in Table 4. The ICR*n*[LEVEL] field sets all priority levels to zero at reset, but this field can be overwritten to change the priority of a given source. See Figure 6.

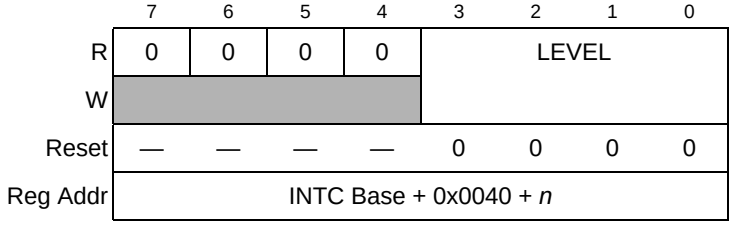


Figure 6. INTC Interrupt Control Registers (ICR*n*)

7.5 Unmasking an Interrupt Source

All interrupt sources are masked at reset. This means that if a peripheral module generates an interrupt request, the Interrupt Controller will not transmit the request to the ARM7 core. The masking state for each source is defined in the Interrupt Mask Registers (IMRH and IMRL). These 32-bit registers form a 64-bit map to allow the request for each source to be masked or unmasked, depending on the state of each bit.

To simplify the process of setting and clearing these bits, two additional registers are provided. The Set Interrupt Mask Register (SIMR) and the Clear Interrupt Mask Register (CIMR) provide a memory-mapped mechanism to set or clear a given bit in the IMR{H,L} registers. The data value written to the SIMR or CIMR causes the corresponding bit in the IMR{H,L} register to be set or cleared, respectively. For example, to clear the masking for Channel 0 of the eMIOS module, (source 43), one would write a value of 43 to the CIMR. This would cause IMR43 in the IMRH to be cleared.

8 Simple Interrupt Handlers

8.1 A Basic Vector Service Routine in C

A VSR would ordinarily be written in assembly language because of the greater control afforded by the assembler's precise syntax. Identifying the source and dispatching the ISR must be done as quickly and efficiently as possible, and assembler is usually the best tool to use to satisfy these requirements.

Some C compilers do however provide a limited amount of support for ARM7 core interrupt handling. If the application does not use nested interrupts, it is sometimes possible to use special compiler keywords to allow the source-identification and ISR-dispatch to be accomplished in a single C-source file. An example of this is shown in Figure 7.

```
#define INTC_CLMASK(*(volatile unsigned char *) 0xFC04801E ))
#define INTC_SLMASK(*(volatile unsigned char *) 0xFC04801F ))
#define INTC_IRQIACK(*(volatile unsigned char *) 0xFC0480EC ))

extern void (* jumptab[64])(void);
void vsr_IRQ (void) __attribute__ ((interrupt ("IRQ")));

void vsr_IRQ (void)
{
  int source = INTC_IRQIACK - 64;

  if (source >= 0)                  /* ignore spurious IRQs */
    jumptab[source]();              /* jump to correct ISR  */

  INTC_CLMASK = INTC_SLMASK;        /* restore CLMASK from SLMASK */
  return;
}
```

Figure 7. IRQ Vector Service Routine

Notice that the function prototype for `vsr_IRQ` uses some non-standard syntax. For the GCC compiler, the keywords:

```
__attribute__ ((interrupt ("IRQ")))
```

are used to warn the compiler that the `vsr_IRQ` routine is not a standard C function, and must be treated differently. The reason for this special treatment will be discussed in Section 8.2.

The `vsr_IRQ` has one local variable, a signed integer called `source`, that is used to receive the interrupt source number. It is signed because the `IRQIACK` register might contain a 63, and subtracting 64 from it could yield a negative number. It is an integer because the compiler will make more efficient use of a 32-bit quantity in the ARM7TDMI-S core 32-bit register set.

After `vsr_IRQ` identifies the source of the interrupt request, its remaining task is to dispatch a handler by calculating a jump to the correct function. If the jump table has been constructed correctly, the statement:

```
if (source >= 0)          /* ignore spurious IRQs */
   jumptab[source]();     /* jump to correct ISR  */
```

will accomplish exactly that. Control is passed to the function whose address is contained in the function pointer at that table entry. The jump table begins at `jumptab[0]`, so a negative source number (i.e. a spurious interrupt) is ignored and `vsr_IRQ` just returns, having done nothing. A partial jump table for the MAC7100 family is shown in Figure 8. [3]

```
#include "isr.h"                    /* function prototypes for ISRs */

void (* jumptab[64])(void) =
{
  &isr_eDMA0,                       /*  0 0x000 */
  &isr_eDMA1,                       /*  1 0x004 */
...
  &isr_XIRQ                         /* 63 0x0FC */
};
```

Figure 8. MAC7100 Jump Table (partial)

Figure 7 also shows the current level mask being restored from its saved location in the SLMASK register just before the VSR returns. If the EMASK bit in the ICONFIG register is left in its default state (set), then the value of CLMASK is automatically copied to SLMASK when the IRQIACK value is read. The current level mask must be restored to its former state, or else further interrupt signals of the same priority will be masked.

Although the EMASK bit can be cleared to avoid this, it is perhaps better to highlight the function of CLMASK in order to be aware of its capabilities. Hardware masking is one of the ways the MAC7100 Interrupt Controller helps to implement nested interrupts, which will be discussed in Section 9.

8.2 Special Considerations When Using C

Recall from Section 5.2 that when returning from an IRQ handler, the original CPSR and PC must be restored simultaneously from their saved locations in the banked register set, through the use of special machine instructions. These instructions are not normally generated by the compiler as return statements from an ordinary C function, so the compiler must be informed when this special treatment is necessary.

Another thing to consider is that compilers for ARM7TDMI-S core-based devices do not preserve every register used by a function. It is assumed that r0-r3 and r12 are scratch registers. These registers are not required by the ARM-Thumb Procedure Call Standard (ATPCS) to be saved on the stack, and the compiler considers their contents to be unchanging only between function calls, not across them.

This is fine when a compiler is producing code for a sequential flow of execution in which functions are called, and they return in a well-behaved and predictable fashion. But this is not the case when interrupts come into the picture. The difference with interrupts is that a vector service routine like the one in Figure 7 is never called. Because the compiler has no record of the VSR being called, it is unaware that the scratch registers are not safe storage locations.

3. Note that later variants in the MAC7100 family may contain a different number of interrupt sources, so the jump table would need to be modified in that case.

From the main program's point of view, the contents of any scratch registers used by `vsr_IRQ` might change unexpectedly at any time. That is why a VSR must save the scratch registers too, even though this is not required by the ATPCS, and it must restore them when it returns control to the main program.

Because of this, the `vsr_IRQ` routine must be identified to the compiler as being a special case. Many compiler vendors have implemented new syntax for making this distinction. Some use keywords like "`__attribute__`" or "`__irq`".

Others use pragmas such as:

```
#pragma TRAP_PROC
  ...
```

Some even wrap the function in pragmas to indicate the beginning and end:

```
#pragma interrupt
  ...
#pragma interrupt off
```

The point is that if you are going to write your interrupt handler entirely in C, you must be aware of the distinction between it and a normal C function, and you must know how to inform your compiler when the rules for procedure calls need to be modified. If you intend your source code to be portable between compiler vendors, this can cause real maintenance problems. This is why many embedded systems designers prefer to write these special routines in pure assembler instead of C.

8.3 A Simple VSR Written in Assembly Language

An example VSR written in assembler is shown in Figure 9. The VSR is entered with the processor in ARM state, in IRQ mode with the IRQ banked registers active. The scratch registers are saved in line 1, because it is guaranteed that the procedure being targeted by the table jump will not preserve them. R4_irq is also preserved in the same statement, to be used as non-volatile storage for the INTC base address.

The IRQ-mode Link Register is also preserved in the same statement in line 1. This is necessary because the table-jump requires a back link to return, but the LR_irq already contains the return address from the `vsr_IRQ` routine itself. If LR_irq is overwritten, `vsr_IRQ` cannot return to the main program, so a copy of the LR_irq must be saved and restored later.

The contents of the IRQIACK register are read in lines 2 and 3, leaving the base address of the INTC module in the non-volatile R4_irq to be used again after the table jump returns.

The `subs` instruction in line 4 subtracts 64 from the IRQIACK value to produce the interrupt source number. Setting the S-bit in this instruction causes the ARM7 core to write the condition code flags for later use (See Figure 1). The state of CPSR[N] allows the conditional execution of the next three instructions using the "execute-if-plus" encoding. [4]

Line 5 moves the address of the jump table into R3_irq. The Program Counter is saved into the Link Register in line 6 to create a return point for the jump. Note that the program counter actually corresponds to the currently executing instruction + 8 due to the three-stage pipeline of the ARM7 core, so saving the current PC creates a back link that will return to line 8 when the jump returns.

4. The order of the statements in this routine could be rearranged to avoid pipeline stalls. This additional level of optimization is beyond the scope of this document.

The next instruction in line 7 modifies the PC by loading the contents of the correct jump table entry, [5] calculated by multiplying the source number by four. This is what triggers the jump, but the statement is only executed on the condition that the result of the subtraction in line 4 was positive or zero. This is how the `vsr_IRQ` routine is able to ignore spurious interrupt requests.

```
      #define INTC_SLMASK  0x1F                    /* offsets from INTC base addr  */
      #define INTC_CLMASK  0x1E
      #define INTC_IRQIACK 0xEC

      vsr_IRQ:
            /* enter in ARM state, IRQ mode                                       */
            /* pre-IRQ CPSR saved in SPSR_irq, PC+4 saved in LR_irq               */

      1.    stmfd    sp!, {r0-r4, r12, lr}       /* save registers on IRQ stack  */
      2.    ldr      r4,  .__INTC_base           /* base addr of register space  */

            /* LR_irq, working and scratch registers are now saved on IRQ stack   */
            /* base address of INTC is in r4                                      */
            /* all other regs are non-volatile and will be preserved downstream   */

      3.    ldrb     r2,  [r4, #INTC_IRQIACK]    /* r2 := IRQIACK                */
      4.    subs     r2,  r2, #64                /* r2 := IRQ source number      */
      5.    movpl    r3,  #0x40000000            /* address of jump table        */
      6.    movpl    lr,  pc                     /* prepare link for branch      */
      7.    ldrpl    pc,  [r3, r2, lsl #2]       /* jump to correct ISR          */

                                                 /* r4 survives the table jump   */
      8.    ldrb     r2,  [r4, #INTC_SLMASK]     /* r2 := SLMASK (old CLMASK)    */
      9.    strb     r2,  [r4, #INTC_CLMASK]     /* restore old CLMASK from r2   */
      10.   ldmfd    sp!, {r0-r4, r12, lr}       /* restore registers from stack */
      11.   subs     pc,  lr, #4                 /* return from exception        */
```

Figure 9. A Simple (Non-Reentrant) Vector Service Routine

Notice that this example assumes that the jump table is located at `0x40000000`. This address was chosen because it is the bottom of internal RAM on the MAC7100 family, and also because it is a quick number to load into a register on the ARM7 core. In practice, the jump table could be located anywhere in memory on a four-byte boundary. The table doesn't have to be at `0x40000000`; it just makes things a little easier for this example.

5. It is of course up the programmer to ensure that this entry contains the address of the correct ISR.

8.4 The Interrupt Service Routine

An ISR is just a normal procedure called by the VSR, and it looks and works just like any other procedure that takes no arguments and returns no values. It does not require any special keywords to compile it, and the compiler is unaware that this particular ISR is executed in IRQ mode using the IRQ register bank.

The ISR will have a number of tasks to perform in order to service the interrupt. These operations will be specific to each application, so a discussion of those details will be postponed until the example projects are introduced. It is very important to note that whatever else the ISR does, it must not omit one very important task: When a peripheral module asserts an interrupt request, a flag bit will be set in that peripheral's status register. The request must be acknowledged and its associated flag cleared by the ISR.

The MAC7100 Interrupt Controller does not have the ability to acknowledge and clear interrupt flags at the peripheral module. The INTC can only identify the source; it can't clear any flags. The interrupt service routine must do that, or else the interrupt request will still be asserted when `vsr_IRQ` returns, and the program will enter an infinite loop.

8.5 Restore the Context and Return

After the table-jump returns, the VSR uses the contents of the saved level mask to restore the current level mask. This is necessary because hardware masking was enabled, and the mask would have prevented the IRQ signal from being generated again if CLMASK was not restored to its former state.

The r4, scratch and link registers are restored, and the final `subs` instruction restores the PC and CPSR simultaneously, returning control to the main program flow.

9 Nested Interrupts

9.1 Overview

The purpose of nesting interrupts is to reduce the latency for handling higher-priority interrupt requests. A simple vector service routine like the one detailed in Figure 9 does not re-enable IRQ requests before executing the table-jump. With IRQs disabled, the only advantage in prioritizing interrupt sources is that if two requests arrive at the same time, the higher-priority one will be serviced first. A high-priority IRQ occurring during a low-priority IRQ's service routine will have to wait until that ISR has finished.

With nested interrupts, a high-priority IRQ can suspend a low-priority ISR, allowing the higher-priority IRQ to be serviced without waiting. This can be a distinct advantage in a system where multiple sources of interrupt are active, and some of them require frequent and prompt servicing.

The availability of the FIQ exception type and its larger set of banked registers provides hardware support for a single level of nesting, and this is perhaps the simplest solution if only one additional nesting level is required. The machine state need not be saved, because a carefully constructed FIQ handler does not disturb it.

If further levels of nesting are needed, or if the programmer wants to reserve the FIQ for a special purpose, then the context must be saved in software. The MAC7100 INTC module has some features that will help (see Section 7), but the main responsibility for supporting a complex, nested-interrupt environment lies with the vector service routine.

In order to provide this type of functionality, the VSR must be rewritten so that it saves the entire machine state necessary for interrupt servicing. Once this has been accomplished, the VSR has become reentrant, meaning that it can be executed recursively without destroying the context. Only then can the VSR safely re-enable IRQ requests.

In this situation, coding the vector service routine in assembler becomes practically mandatory because of the additional complexities involved. There are several more items to be saved in order to preserve the complete machine state, and the core processor's operating mode must be also changed.

9.2 Saving the Machine State

The overall concept to achieving this end is simply stated: For every object containing a part of the machine state, that object must be preserved before interrupts are re-enabled if a higher-level interrupt might later change the object without saving its contents.

Objects satisfying this criteria fall into three categories:

- Scratch registers (r0-r3, r12)
- Banked registers (LR_irq, SPSR_irq)
- INTC module registers containing state information

The scratch registers have already been discussed, but interrupt nesting requires saving some additional context to allow the VSR to safely re-enable IRQ exceptions.

9.2.1 The Banked Link Register

In the previous examples, the IRQ exception caused the core to switch to IRQ mode, where it remained during interrupt handling until control was returned to the main program. The banked IRQ stack pointer and IRQ link register were used for stack manipulation and for setting return links from procedure calls. It is the LR_irq that must be preserved when IRQ handlers become reentrant.

Recall that the assertion of an IRQ request automatically and immediately overwrites the LR_irq. Consider also that the IRQ handler might itself need to call other procedures. If interrupt servicing is being done entirely in IRQ mode, the LR_irq would therefore be used for return links from these procedure calls. The LR_irq must be used in this case because procedure calls always require a back link, and the LR_irq is the only link register available in IRQ mode.

Ordinarily, the compiler saves the link register on the stack at the entry point of any C function. But if an ISR calls a procedure, and then a higher-priority IRQ exception is asserted before the LR_irq can be saved, the context will be destroyed.

This is why reentrant IRQ handlers must not re-enable IRQ exceptions when they are still in IRQ mode. [6] The IRQ-mode context must be saved, and the CPSR[Mode] bits must be changed to switch modes before IRQ exceptions can be re-enabled. Only then is it safe to consider servicing higher-priority IRQ requests. System mode is best suited for this purpose, because it allows privileged hardware access while protecting the banked registers from accidental corruption.

6. Or else you must ensure that all procedures after the trampoline do not ever use the link register, which is hardly an optimum solution considering that the VSR does exactly that.

9.2.2 The Banked SPSR_irq

The second item that must be preserved before it is safe to re-enable IRQs is the SPSR_irq. This banked register is provided in the ARM7 architecture to preserve the contents of the CPSR during IRQ handling, but there is only one SPSR_irq. If another IRQ event overwrites the SPSR_irq, the original CPSR saved by the lower-level interrupt will be destroyed. A reentrant IRQ handler must save a copy of SPSR_irq before other IRQ events can overwrite it.

9.2.3 The Contents of INTC[SLMASK]

Setting the EMASK bit in the INTC[ICONFIG] register enables a hardware-masking capability. Hardware masking allows the Interrupt Controller Module to prevent an interrupt request from reaching the core if an equal or higher-priority request is being serviced at the time. This prevents a situation where interrupts could continually preempt each other, resulting in a deadlock.

If masking is enabled, a current level mask value is defined that prevents interrupts with priorities equal to or less than this value from being recognized. The current level mask is stored in the INTC[CLMASK] register (see Figure 10). The value of CLMASK is 0x1F on reset, so all interrupts are unmasked here by default. [7]

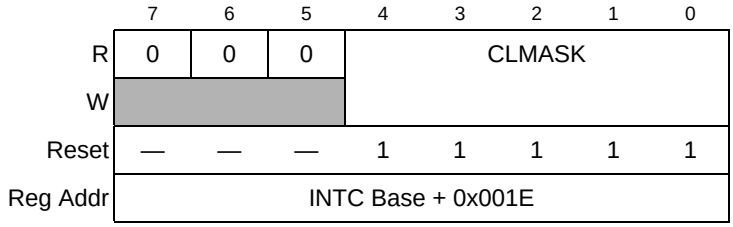


Figure 10. INTC Current Level Mask Register (CLMASK)

When a vector service routine reads the IRQIACK register to determine the source of an interrupt, the INTC does more than just return the source number. If the EMASK bit in the INTC[ICONFIG] register is set when the IRQIACK is read, the priority of the source is automatically copied into CLMASK, and the former contents of CLMASK are saved. A saved level mask register (SLMASK) is provided for this purpose (see Figure 11).

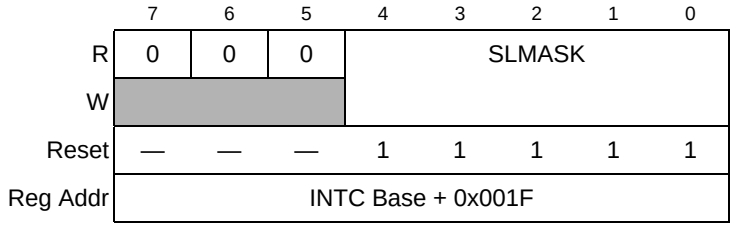


Figure 11. INTC Saved Level Mask Register (SLMASK)

7. This does not override the masking state enforced by the Interrupt Mask Registers, which mask all priorities at reset. The IMR{H,L} setting is not dynamic, and only changes when explicitly modified. The CLMASK is a dynamic setting, and changes automatically according to the most recent interrupt event that was active when IRQIACK was read. Both masks must be disabled for an interrupt signal to be transmitted to the core.

There is only a single SLMASK available, so if additional levels of nested interrupts are to be serviced, this machine-state information must be preserved. One way to do this is to store the contents of SLMASK into a non-volatile data register. This way, any further nested service routines will preserve the SLMASK, because the ATPCS requires functions to stack all non-volatile registers before overwriting them. This method also ensures that the value is readily available when the ISR returns, because the CLMASK must be restored from SLMASK at that time.

9.3 A Reentrant Vector Service Routine

Figure 12 shows a vector service routine that implements nested interrupts. Many of the statements in this listing will be familiar from the simple VSR shown in Figure 9, but it is somewhat more complicated.

As before, the VSR is entered in ARM state (32-bit instructions) and in IRQ mode, with the IRQ banked registers enabled. Several more registers are saved on the stack than previously, but a “store multiple with full descending stack” instruction (`stmfd`) handles this in a single statement. This requires more time to complete for the additional registers, but the ARM v4T architecture is optimized for this operation. Note that the LR_irq is one of the registers saved.

The SPSR_irq is stored into R6_irq, and the INTC[SLMASK] is stored into R5_irq. These are non-volatile data registers, ensuring that any functions using them later will preserve them first. The base address of the INTC module is also stored into R4_irq to be used later.

By the time processing reaches line 6, the VSR has saved all the necessary machine state information and is ready to switch modes. This is done by writing the pattern 0x1F to the CPSR control field (cpsr_c).

Ordinarily it would not be considered advisable to write an immediate value to the CPSR in this way, but here the entire state of the CPSR control bit field is already known. The Mode and IRQ Disable bits are the quantities being set, and Thumb mode is disabled during exception handling. It is also known that FIQ exceptions must be enabled at this point, because FIQs are only disabled during FIQ processing. [8] This VSR could not have been reached if FIQs were disabled, because IRQs are also disabled during FIQ processing.

Once the core has been placed into System mode, a higher-priority IRQ can safely preempt the current execution without destroying any context. The System-mode link register is saved on the stack in line 7 prior to modifying it in line 10. Calculating the source number and executing the table jump proceeds as in Figure 9, using the jump table at the bottom of RAM.

IRQ exceptions are disabled in line 13 after the jump returns, making it safe to restore the context that was saved earlier. At this point, the whole process is reversed. The machine is restored to its former state, and control passes back to the next lower level of interrupt servicing (or back to the main program) in line 19.

8. A significant assumption is being made here. If FIQs are not used in the application at all, or if for some unusual reason it is required to disable them during IRQ processing, then the immediate value written to the CPSR in Figure 12 lines 6 and 13 will necessarily be different.

```
        .global vsr_IRQ
vsr_IRQ:

        /* enter in ARM state, IRQ mode                                    */
        /* pre-IRQ CPSR saved in SPSR_irq, PC+4 saved in LR_irq            */
1.      stmfd     sp!, {r0-r6, r12, lr}    /* LR_irq + 7 regs onto IRQ stack */
2.      mrs       r6,  spsr                /* r6 := SPSR_irq                 */
3.      ldr       r4,  .__INTC_base        /* base address of INTC module    */
4.      ldrb      r5,  [r4,#INTC_SLMASK]   /* r5 := SLMASK                   */
5.      ldrb      r2,  [r4,#INTC_IRQIACK]  /* r2 := IRQIACK                  */

        /* LR_irq, working and scratch registers are now saved on IRQ stack */
        /* base addr of INTC, the SLMASK, and SPSR_irq are now in r4,r5,r6  */
        /* all other regs are non-volatile and will be preserved downstream */

        /* switch to system mode and re-enable IRQ exceptions              */
6.      msr       cpsr_c, #SYS_MODE       /* SYS mode, IRQ & FIQ enabled  */
7.      stmfd     sp!, {lr}               /* link register onto user stack */
8.      subs      r2,  r2, #64            /* r2 := IRQ source number      */
9.      movpl     r3,  #0x40000000        /* address of jump table        */
10.     movpl     lr,  pc                 /* prepare link for branch      */
11.     ldrpl     pc,  [r3, r2, lsl #2]   /* jump to correct ISR          */
12.     ldmfd     sp!, {lr}               /* restore LR_usr from user stack */

        /* switch back to IRQ mode, IRQs disabled, to restore machine state */
13.     msr       cpsr_c, #(IRQ_MODE|IRQ_DISABLE)
14.     ldrb      r2,  [r4, #INTC_SLMASK] /* r2 := SLMASK (the old CLMASK) */
15.     strb      r5,  [r4, #INTC_SLMASK] /* restore old SLMASK from r5   */
16.     strb      r2,  [r4, #INTC_CLMASK] /* restore old CLMASK from r2   */
17.     msr       spsr,  r6               /* restore old SPSR_irq from r6 */
18.     ldmfd     sp!, {r0-r6, r12, lr}   /* restore registers from stack */
19.     subs      pc,  lr,  #4            /* return from exception        */
```

Figure 12. A Reentrant Vector Service Routine

10 Example Projects

10.1 Overview

There are three example projects included with this application note -- a simple interrupt handling example, a more complicated version that uses nested normal interrupts, and a final example that adds a fast interrupt request. These examples are designed to be run on the MAC7100EVB, and are compiled with the GNU arm-elf-gcc cross compiler. All of the projects are based on an infinite loop main program that repeatedly copies a pseudo-random array of integers into RAM, and sorts them in place. This work has

absolutely no bearing on anything else, and is included for no reason other than to give the processor something to do in its spare time.

While this sorting is going on, interrupts are arriving from the MAC7111 peripheral modules and are being serviced using the concepts developed earlier.

The source code is nearly the same for all three projects, and is drawn from a /common folder by the Makefiles for each project. Small differences between each project are contained in separate versions of certain critical files. These are contained in private /src folders within each project. Any file in a private /src folder will be chosen preferentially over a file by the same name in the /common folder (see the VPATH setting in the Makefile for each project.) Explanatory notes for these files are contained in Appendix A.

10.2 Example 1: A Simple IRQ Handler

In the first example, the main program is being regularly interrupted by the PIT1 timer. Nested interrupts are not supported, because a non-reentrant vector service routine is used to dispatch a handler for the PIT1.

The PIT1 interrupt service routine produces a visible output on the LED array, combined with a text display on the SCI-B port that reports the LED position. You can see the text output by connecting Hyperterminal to the SCI-B port at 115200 baud, no parity, 8 data bits, 1 stop bit. The “LED_Blinker” ISR causes the LEDs to blink in a rotating pattern with three-on and one-off. The overall effect produced is a dark LED rotating among a field of bright ones.

A second, higher-priority interrupt is manually forced from within LED_Blinker. The service routine for the second interrupt simply reverses the bits of the LED state variable. This ISR (LED_Reverse) is not serviced immediately because interrupt nesting is not supported in this example, so the interrupt is still pending when the first ISR returns.

This change has no visible effect because the second interrupt request must wait for service until after the first ISR has returned. By that time, the LED state has already been written to the LED array.

10.3 Example 2: A Reentrant IRQ Handler

The second example illustrates a reentrant IRQ handler. The structure of the first project is maintained, with changes added to the vector service routine in crt0.S to implement interrupt nesting. The interrupt service routine is identical to the first example, and the second interrupt is forced manually just as before. Because the VSR is reentrant, the higher-priority interrupt is serviced immediately, as soon as it is forced. The result is that the bitwise reversal of the LED state variable now takes place within the first ISR, not after it returns.

The second interrupt is forced just before the LED state is written to the port, so the reversal becomes visible on the LED array. The blinking effect changes to become a single bright LED rotating within a field of dark ones, the opposite of the first example.

The hardware setup (sysinit.c) and the ISR code (isr.c) are identical to the first example, but the change is visible on the array because the second ISR was able to nest on top of the first. The only difference is that the second example has a more advanced vector service routine in crt0.S, one that supports nesting.

10.4 Example 3: Add an FIQ

The last example adds an FIQ to the previous projects, a PIT2 interrupt running with a 10 microsecond period. The setup for this is seen in the file sysinit.c, in the /src directory of the third example. The ISR for this interrupt drives a $1/16^{th}$ duty cycle pulse on PORTF[15]. This high-frequency signal completely overwhelms the slow $1/10^{th}$ second pulse from the other IRQs, causing one of the LEDs to appear to glow dimly instead of blinking as before.

The FIQ service routine is written in assembler, in the file crt0.S, also contained in the private /src directory. It executes immediately as soon as the FIQ signal is received by the core because it is located at 0x001C. There is no need for an intermediate vector service routine because there is only one FIQ source.

11 Conclusion and Further Reading

This application note has presented the interrupt handling capabilities of the MAC7100 family of microprocessors. The MAC7100 is based on the ARM7TDMI-S core, and much of the discussion has centered around the instruction set and programming model of that core. A number of the potential difficulties involved with interrupt servicing have been highlighted, and example solutions offered.

The Interrupt Controller Module of the MAC7100 provides many useful services to assist the programmer. Methods were presented for using the INTC module to classify multiple interrupt sources into types, organize them into a priority scheme, and selectively mask or unmask them.

The complexities involved with servicing multiply nested asynchronous interrupt signals has been presented with example projects.

Much of the information contained in this document has been gathered from sources available on the web. In particular, the Freescale and ARM web sites contain a great deal of useful information to help the system designer to implement the concepts developed in this paper.

For further information on the ARM core, see the following:

11.1 ARM Documentation

http://www.arm.com/documentation

ARM7TDMI-S (Rev.4) Technical Reference Manual, ARM DDI 0234A

ARM Architectural Reference Manual, ARM DDI 0100E

Exception Handling on the ARM, ARM DAI 0025E

Software Prioritization of Interrupts, ARM DAI 0030A

ARM-THUMB Procedure Call Standard, SWS ESPC 0002 A-05

11.2 ARM Technical Support FAQs

http://www.arm.com/support/Cores_Interrupts.html

11.3 MAC7100 Reference

For reference information about the MAC7100 family of microprocessors, see the Freescale web site: http://www.freescale.com.

MAC7100 Microcontroller Family Reference Manual, MAC7100RM

11.4 Other Reading

An excellent printed reference is available on the development of the ARM family, containing much background information on ARM processors in general:

Furber, *ARM system-on-chip architecture*, 2nd Edition, Pearson Education Limited, Edinburgh, 2000

For detailed information on the use of GNU tools for programming and debugging, consult the GCC and GDB manuals from the FSF:

Using the GNU Compiler Collection (GCC), Version 3.4.2, Free Software Foundation, Boston, MA 2004

Stallman, Pesch, Shebs, et al. *Debugging with GDB, The GNU Source-Level Debugger*, 9th Edition, Free Software Foundation, Boston, MA 2004

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A
Notes on The Example Source Code

A.1 vectors.S

The vectors.S file contains the trampolines, the VSR table, and the “trap” vector service routines as described in Figure 3. There is little more to discuss about this file, except to note that it must be located at the bottom of memory at `0x0000`. This location is not specified anywhere in the vectors.S file itself. Instead, a “.section” pseudo-op statement at the top of the file places the contents into a section called “.vectors”. The linker directives file “project.ld” determines the absolute address for each section, and it places the “.vectors” section into a one-kilobyte block of memory originating at `0x0000`.

The third example project has a slightly different vectors.S file than the other two, because it contains an FIQ service routine at `0x001C` instead of a trampoline:

```
                                /* 0x001C FIQ service routine */
ldr       r10, .__PIT_base      /* r10 := PIT base address    */
mov       r11, #0x4             /* TIF2 is bit 2              */
str       r11, [r10, #0x100]    /* clear the TIF2 flag        */

add       r8, r8, #0x01         /* increment r8               */
ldr       r10, .__portF_base    /* port F base address        */
ands      r8, r8, #0x0F         /* r8 <- (r8 modulo 16)       */

moveq     r11, #0x00
movne     r11, #0x01
strb      r11, [r10, #0x37]     /* write port F PINDATA[15]   */

subs      pc, lr, #4            /* return from FIQ            */
```

The first three statements of the ISR clear the PIT2 interrupt flag, as must be done to avoid an infinite interrupt loop. The next three statements increment a modulo-16 counter on R8_fiq. This banked register is not used by any other code in the project, so it can be considered non-volatile storage.

Note that if the data contained in R8_fiq was being used for a critical purpose, this would not be considered a good programming practice. Registers should never be used for long-term storage of important data. In this case, R8_fiq is just holding the state of a counter running at 100 kHz, and its value is only used to update the state of an LED.

Depending on the result of the `ands` statement (zero or non-zero) the R11_fiq register receives a value of zero or one, and this value is used to update the LED state. Port F[15] is written as a “1” most of the time, so the active-low LED is usually turned off. The LED appears to glow dimly because it receives a “0” every sixteenth time through the ISR.

A.2 crt0.S

The crt0.S file contains two vector service routines, one for the reset vector and one for the IRQ vector.

The reset VSR begins by initializing the exception environment. This is done by cycling through each operating mode and initializing the stack pointer for that mode. The locations and sizes for each stack are defined at the bottom of the file.

Note that the IRQ stack in the first example project is larger than the SYS stack. This is because interrupt servicing in the first example is done entirely in IRQ mode. There is no need to switch to System mode in this example because the VSR is not reentrant, so a larger SYS stack is not required. It is critical that the system designer monitors the stack requirements of the application and selects an appropriate size for each mode stack.

```
.global _reset
_reset:

/* initialize exception environments */
        mov     r0, #(IRQ_DISABLE|FIQ_DISABLE|FIQ_MODE)
        msr     cpsr, r0
        ldr     sp, .__FIQ_stack

        mov     r0, #(IRQ_DISABLE|FIQ_DISABLE|IRQ_MODE)
        msr     cpsr, r0
        ldr     sp, .__IRQ_stack

        mov     r0, #(IRQ_DISABLE|FIQ_DISABLE|ABT_MODE)
        msr     cpsr, r0
        ldr     sp, .__ABT_stack

        mov     r0, #(IRQ_DISABLE|FIQ_DISABLE|UND_MODE)
        msr     cpsr, r0
        ldr     sp, .__UND_stack

        mov     r0, #(IRQ_DISABLE|FIQ_DISABLE|SYS_MODE)
        msr     cpsr, r0
        ldr     sp, .__SYS_stack

        mov     r0, #(IRQ_DISABLE|FIQ_DISABLE|SVC_MODE)
        msr     cpsr, r0
        msr     spsr, r0            /* some LIBGCC1 funcs restore from SPSR */
        ldr     sp, .__SVC_stack
```

The next task of the reset handler is probably more familiar, because it is performed at startup by every compiled C program. Setting up the C runtime environment by copying initialized data and clearing the block start segment are actions required by ANSI C. These two tasks are somewhat simplified because the

data is moved four bytes at a time. This shortcut can be used in this case because the linker directives file aligns .data and .bss on four-byte boundaries.

Beneath the reset handler is the vector service routine for IRQ exceptions. The VSR for the first example project is the simplified, non-reentrant version as shown in Figure 9. The VSRs for examples 2 and 3 are the more advanced version shown in Figure 12.

A.3 sysinit.c

The system initialization routine is called by the reset handler, and is where all of the remaining hardware setup is performed. This is where the system clock is initialized to its final value, having run in its default state since power-on.

```
/* ----- initialize system clock to FSYS (see project.h) */
 errflag = mac7101_crg_init(FSYS);
```

The port pins are set as outputs to enable the LEDs…

```
/* initialize PORT module to GPIO outputs for LEDs */
  for (i=12; i<=15; i++)
    PORTF_CONFIG(i) = GPIO_MODE | DDR_OUTPUT;

 PORTF_PORTDATA  = 0xF000;             /* turn off all LEDs */
```

… and the serial communications interface (SCI) is initialized to allow RS-232 communication with the debugging monitor at 115200 baud.

```
/* initialize eSCI module */
  esci_mode = ESCI_CR12_RE              /* enable receiver */
            | ESCI_CR12_TE;             /* enable transmitter */

  errflag = mac7101_sci_init (
    FSYS,                               /* system clock (in MHz) */
    ESCI_BASE,                          /* eSCI chan (see project.h) */
    115200,                             /* baud rate */
    esci_mode);                         /* mode mask (see above) */
```

PIT1 is then set to produce interrupts every 100ms. [9] Note that the PIT3 interrupt doesn't need to be set up at the PIT module level. [10]

```
/* initialize PIT1 timer to produce interrupts */
  errflag = mac7101_pit_init(
    FSYS,                               /* system clock (in MHz) */
    1,                                  /* PIT channel (1..4) */
    (100*MSECONDS));                    /* period */
```

The Interrupt Controller's ICONFIG register is set up as described earlier, mapping all priorities to IRQs except priority 15, which is an FIQ.

9. The SCI and PIT setup are both done through calls to library functions. The source code for these functions is in the /lib folder parallel to the project directory.
10. There is no need to set up a PIT3 timer because its interrupt signal is being generated by INTC.

```
/* initialize interrupt controller */
/* configure levels 0-14 as IRQ, level 15 as FIQ, masking enabled */
  INTC_ICONFIG = INTC_CONFIG_EMASK
               | INTC_CONFIG_FIQDEF_15;

/* set PIT1 to a Level-7 interrupt and unmask it */
  INTC_ICR(PIT1_SRC) = 7;              /* make PIT1 a Level-7 int */
  INTC_CIMR = PIT1_SRC;                /* unmask PIT1 in CIMR */

/* set PIT3 to a higher priority than PIT1 and unmask it */
/* (unmasking is not actually necessary because PIT3 is being forced) */
  INTC_ICR(PIT3_SRC) = 8;              /* make PIT3 a Level-8 int */
  INTC_CIMR = PIT3_SRC;                /* unmask PIT3 in CIMR */
```

The address of the two PIT interrupt service routines are entered into the jump table with calls to the `install_isr` macro. (see “mac7111_intc.h” in the /lib folder).

```
/* enter the PIT1 ISR address in jump table */
install_isr(LED_Blinker, PIT1_SRC);

/* enter the PIT3 ISR address in jump table */
install_isr(LED_Reverse, PIT3_SRC);
```

IRQ exceptions are then enabled with an inline-assembler macro call. This macro is also defined in mac7111_intc.h, and illustrates the special syntax available from the GCC compiler.

```
/* read-modify-write the CPSR to enable IRQ requests */
enable_irq();
```

All compilers will have some means for setting or clearing a bit in the CPSR. The C language has no idea about the ARM7 core architecture, so this must be handled either by a call to a function in an assembler file, or by inserting assembler statements into the C file itself. Most compiler vendors will have some means of using inline assembler for this purpose.

In some cases, the assembler statements are injected verbatim directly into the compiler’s instruction stream output, and the avoidance of register conflicts is responsibility of the programmer. The GCC compiler has the ability to accept a more advanced syntax that informs the compiler which register is being used, so the compiler can avoid these conflicts. The `enable_irq` macro is an example of this. An explanation of the syntax used in this macro is available in *Using the GNU Compiler Collection (GCC).*

```
#define enable_irq()        \
    asm volatile (          \
        "mrs r3,cpsr;"      \
        "bic r3,r3,#0x80;"  \
        "msr cpsr,r3"       \
        :                   \
        :                   \
        : "r3"              \
        );
```

The method for enabling IRQ exceptions will vary widely according to the compiler being used. Consult the documentation from your compiler vendor for this information.

The sysinit.c file for Example 3 is slightly different from the first two examples, in that it includes code to initialize a PIT2 timer as a fast interrupt source, and to enable FIQ exceptions.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN2891
Rev. 0
11/2004