Freescale Semiconductor

Freescale Semiconductor, Inc.

Application Note AN001A

Handling 32 and more interrupts
with the M•CORE architecture

Version A

FINAL

Chris Joffrain
Applications Manager
M•CORE Technology Center

© Freescale Semiconductor, Inc., 2004. All rights reserved.

For More Information On This
Go to: www.freescale

freescale™
semiconductor

Table of contents

1 Introduction 3
2 Overview 3
3 Terms, Acronyms 3
4 Interrupt handling 4
4.1 Auto-vectored interrupts 4
4.2 Vectored interrupts 4
5 General implementation (auto-vectored, nested) 4
6 What does the M•CORE architecture provide to handle interrupts 6
7 Initialization 8
8 Hardware Latency 8
9 Software examples 9
10 Auto-vectored, nested interrupts 9
11 Auto-vectored, non-nested interrupts 12
12 Vectored, nested interrupts 14
13 Vectored, non nested interrupts 16
14 Fast vectored, non nested interrupts 17
15 Summary for 32 interrupt requests 18
16 Serving more than 32 interrupt requests 19
17 Summary for up to 256 interrupts 20
18 Conclusion 20
19 Appendix 1: Sample code for Autovectored, Nested Interrupts 21
20 Appendix 2: Sample code for Auto-Vectored, Non-Nested Interrupts 23
21 Appendix 3: Sample code for Vectored, Nested Interrupts 25
22 Appendix 4: Sample code for Vectored, Non-Nested Interrupts 27
23 Appendix 5: Sample code for Vectored, Non-Nested Fast Interrupts 28

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

1 Introduction

The purpose of this document is to describe how to handle 32 and more interrupts with the M•CORE architecture and meet the requirements of speed of execution with the lowest possible overhead. It describes how to implement a software prioritization of interrupts and how vectored interrupts can be used to reduce the response time. Some examples are given of how to take advantage of the various features that have been implemented to make the M•CORE architecture the preferred solution for low power, real-time applications.

2 Overview

Many embedded applications have a large number of interrupt-driven activities. New processors are now expected to handle more than a hundred different sources of interrupt. At the maximum load on the application, there may be up to 30,000 interruptions per second, leaving no more than 30 microseconds to process each of them. There are a number of low-level actions that take place within a microcontroller to respond to different kinds of interrupts. These tasks, some of which being executed in hardware and some being executed in software, introduce what is called 'overhead'. It is imperative that applications reduce this overhead and respond to interrupt requests in the most effective way. The first part of this document reviews these tasks. The second part describes how the M•CORE architecture approaches these tasks. The third part reviews different approaches to respond to interrupt requests.

3 Terms, Acronyms

Context — Refers to the processor environment (memory, registers) available to a user program at a given time.

Alternate register file — Refers to a set of registers that are switched in place of the normal set of registers seen by the user program. This usually happens when the processor changes its flow of execution because of an interrupt or exception.

Shadow register(s) — Refers to registers that automatically receive copies of the Program Counter and Status registers during interrupt and exception processing in order to preserve their original context.

Envelope — Refers to the part of software executed between the processor's detection of an interrupt request and the first instruction servicing the interrupt request. The envelope is defined to contain any jump instruction to access a handler and any interrupt recognition, prioritization, arbitration and routing to a proper handler. The envelope may also contains any software necessary to mask and re-enable lower priority interrupts.

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Handler — Refers to the software that is executed to service a specific interrupt. It also contains the necessary actions to make sure that any resource (register, memory) used by it is properly saved unless this resource is dedicated to this handler.

4 Interrupt handling

There are several ways to handle interrupts. Each offers a different compromise between flexibility and execution time overhead. The M·CORE architecture offers four different ways to handle interrupts: auto-vectored nested interrupts, auto-vectored non-nested interrupts, vectored nested interrupts and vectored non-nested interrupts.

4.1 Auto-vectored interrupts

Auto-vectored nested interrupts activate the standard 'fast interrupt' or 'normal interrupt' inputs to the processor and are serviced by branching to a predetermined address in the vector table, the address for fast or normal interrupts services. The interrupts are nested, which means they can be interrupted by other interrupts with a higher priority. When the interrupts are non-nested, that means they will not be interrupted by other requests of any priority level. The software does not require any priority masking or unmasking and can be significantly smaller and faster.

4.2 Vectored interrupts

Vectored interrupts activate standard 'fast interrupt' or 'normal interrupt' inputs to the processor but also provide a vector number with their request for service. This vector number is read by the processor and used as an index to the main interrupt vector table to access the address of the associated handler. The tasks executed by the envelope are then reduced and incorporated into the handler itself. Nesting of vectored interrupts will require tasks such as context saving and re-enabling of interrupts. If it is acceptable to not interrupt current services, then the overhead can be reduced to a very minimum of only few instructions and clock cycles.

FINAL

5 General implementation (auto-vectored, nested)

Handling auto-vectored and nested interrupts in an application involves a hardware implementation and a software implementation. The hardware implementation is what the processor and system architecture is capable of doing, by design, to assist any application responding to interrupts. The software implementation is what the designer must provide, specific to the application and interrupt sources, to properly handle the interrupts for this application, especially when software prioritization is required.

The initial phase, not included in the overhead calculation, occurs at the initialization stage: the set up of all necessary registers, pointers, memory areas and other parameters that are required to handle interrupts in the application. This is a software component.

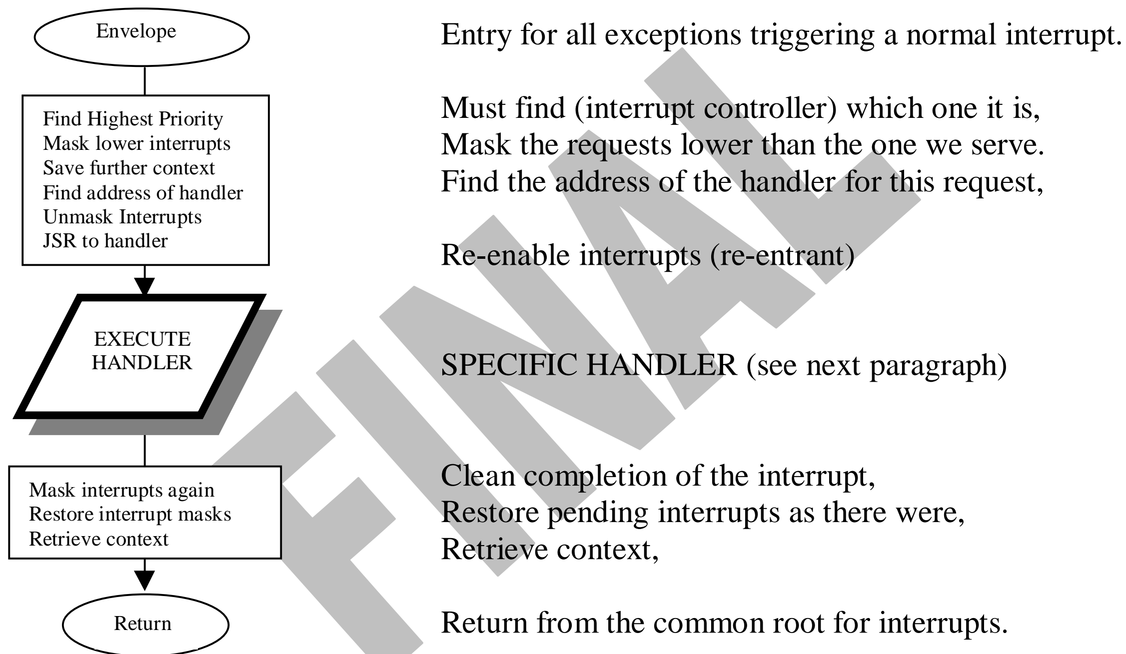
The first phase is executed each time an interrupt occurs. This phase consists of the recognition of the interrupt (i.e. synchronizing the system, waiting for the current instruction

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

to terminate, and transitioning to exception processing.) The time required for the current instruction to terminate before the processor can respond to the interrupt is called ‘latency’. It varies, depending on the number of clock cycles required to execute each of the architecture’s instructions and the ability of the architecture to abort multi-cycles instructions. This latency is determined by the design of the architecture. This is a hardware dependent component.

The second phase is the envelope for the interrupt handler. It deals with what is needed by the handler but not provided by the architecture in hardware. It may perform the selection of the proper handler according to the interrupt activated. The envelope is executed in two parts. The first part occurs at the beginning of the interrupt servicing and identifies the selected interrupt, saving some of the original context if necessary. The second part occurs at the end of the interrupt servicing and restores the conditions that were in effect before the interrupt occurred. Although essential to the correct working of the application, this phase has no real value for the response to the interrupt. It adds overhead, and as a result should be kept minimal.



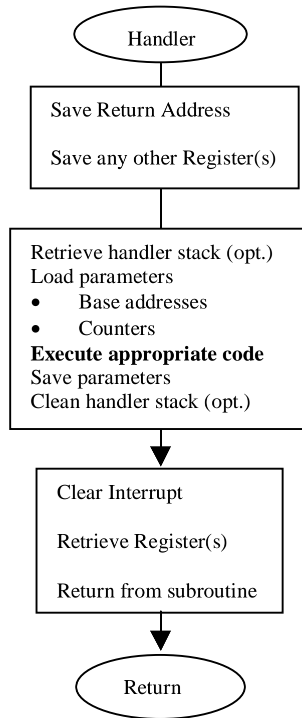
The lesser software context switching is required, the smaller this overhead will be for the application and the faster the interrupt servicing will be.

The third and final phase is the interrupt handler itself. It is actually called from the envelope and will return to the envelope to guarantee a proper system context saving and restoring. It may still contain some supplemental context saving tasks as well as the acknowledgment of the interrupt. In some application the global re-enabling of interrupts may be left to the handler, especially when some handlers are made non-interruptible and some others are made interruptible.

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com



Entry to the specific handler.

Save return address for re-entrance.
Responsible for preserving the registers that will be destroyed by the handler.

Retrieve the parameters specific to this handler.

Execute any task necessary for this handler.

Save the parameters for this handler.

Clear the interrupt that came here.
Retrieve any context saved upon entry.

Retrieve Return Address (re-entrance).

Exit handler and return to the envelope .

6 What does the M•CORE architecture provide to handle interrupts

Several features have been designed into the M·CORE architecture to efficiently support interrupt handling. Details of the M·CORE architecture can be found in the reference manual MCORERM/AD.

Interrupt inputs

There are two interrupt inputs, the normal interrupt input and the fast interrupt input. They operate in a very similar manner except that fast interrupts supersede and can interrupt normal interrupts. They have their dedicated system shadow registers (FPC / FPSR and EPC / EPSR).

Vectored / Auto-vectored acknowledge

All system exceptions, fast interrupt input and the normal interrupt input have a dedicated vectored address in the interrupt table. If the hardware causing an interrupt request can provide an interrupt vector number to the processor, this vector number is taken in place of an auto-vectored normal interrupt. Control is passed directly to the associated handler, saving the overhead time of identifying the interrupt source and retrieving the handler address. Up to 96 vectored handlers can be pointed to with the current architecture and provision has been made for expanding this number for future applications.

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Interrupt latency control bit

Some instructions take several clock cycles to complete. Interrupt requests occurring during the execution of such instructions may be affected by the hardware latency. The M•CORE architecture provides a mechanism to avoid this problem. A latency control bit in the status register (IC bit) allows the processor, when set, to abort multiple cycles instructions in favor of the interrupt request. Upon return from the interrupt, the instruction that was aborted is re-executed again.

Alternate register file

The M•CORE architecture has a full alternate register file as defined earlier. A response to an interrupt request can take advantage of this alternate register file and operate on these alternate registers, leaving the user's registers unchanged and still having not to save a user's context (environment). It can be automatically selected upon detection of the interrupt or exception to provide fast context switching. This alternate register file can also be selected by the application via a control bit (AF bit) in the Processor Status Register (PSR).

Machine state shadow registers

Fast and normal interrupt services have their dedicated system shadow registers. When responding to a fast interrupt request, the program counter (PC) is saved into the fast interrupt PC shadow register (FPC) and the system status register (PSR) is saved into the fast interrupt status shadow register (FPSR). When responding to a normal interrupt request or an exception, the program counter (PC) is saved into the exception PC shadow register (EPC) and the system status register (PSR) is saved into the exception status shadow register (EPSR).

STM / LDM and STQ / LDQ instructions

Handling of multiple interrupts requires saving some information (context) related to the current program that is suspended and restoring this context when returning control back to the program. These operations of context switching (saving and restoring) is facilitated in the M•CORE architecture by the use of four instructions: Load Multiple registers (LDM), Store Multiple registers (STM), Load registers Quadrant (LDQ) and Store registers Quadrant (STQ). They provide a fast way to save and retrieve the selected registers to memory (or stack) in a very few clock cycles.

Find First One instruction

This instruction has been designed to assist in finding the position of the most significant bit 1 in a word. One application of this instruction when handling interrupts is to find the next highest priority to serve from a list of pending interrupts found in a 32-bit register of the interrupt controller. This instruction executes in 1 clock cycle whether the bit found is in the first or last position. This is a powerful instruction which saves the many instructions and clock cycles of a full algorithm prioritization.

Freescale Semiconductor, Inc.
FINAL

For More Information On This Product,
Go to: www.freescale.com

Barrel shifter

The barrel shifter is used to perform the 'Find First One' instruction and many others also used for interrupt handling (mask processing) such as Bit Generate (BGENI), Bit Mask (BMASKI) as well as usual logical and arithmetic shift and rotate instructions.

Supervisor scratch registers

Five scratch registers are provided to hold temporary data for the application without having to resort to moving data to external memory with the associated drawbacks (multiple cycles access memory, dedicated memory locations).

7 Initialization

Initialization is briefly described here. It performs the loading of the vector base register pointing to the exception vector table. Loading of the system stack pointer may be part of this task but is also necessary for the other tasks in the system and is not considered specific to the processing of interrupts in this document. Other tasks performed during the initialization phase must setup the interrupt controller, the masks to enable or disable the interrupts and possibly execute some initialization of peripherals associated with these interrupts.

8 Hardware Latency

The first action that takes place, when an interrupt is generated, is a hardware process that consists of several actions:

- Synchronization of the interrupt request with the system.
- Waiting for the current instruction to terminate.
- Automatic saving of the program counter and the status register.
- Identification of the interrupt or exception and fetching of the associated vector.
- Selection of the Alternate Register File if requested (low order bit in the fetched vector).

The synchronization of the interrupt request happens with the first clock edge arriving after the required setup time. This takes a maximum of one clock cycle.

Most of M•CORE architecture's instructions execute in one clock cycle. The longest execution time is for the DIVU or DIVS instruction and can be up to 37 clock cycles. However, there is a specific feature that limits this waiting time to a maximum of three clock cycles. This is called the 'interrupt control' and is accessible via an interrupt control bit (IC bit) in the status register. Setting the IC bit to one allows multi-cycle instructions to be aborted before completion and thus limit the interrupt latency to a maximum of 3 clock cycles from the time the interrupt is received. Once interrupt processing is initiated, whether the instruction is aborted or allowed to complete, an additional 3 clock cycles are required by the M•CORE architecture to transition to the interrupt service routine and be ready to execute the first instruction.

FINAL
Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

9 Software examples

New embedded applications are expected to handle at least 32 and up to 200 interrupt sources. Not many processor derivatives have that many interrupt inputs built in the silicon and software prioritization is required. Different techniques are also described here that reduce or eliminate the software prioritization with the help of special hardware features.

10 Auto-vectored, nested interrupts

This is the most flexible configuration. It also has the largest overhead. Auto-vectoring implicitly uses the Fast Interrupt vector or the Normal Interrupt vector. To handle multiple fast or normal interrupts, the auto-vectoring function requires a software prioritization of the interrupts. Nesting (capability of being interrupted) of interrupt services requires the saving and restoring of the context along with the masking and unmasking of other interrupts.

The first software part that is executed in response to a global interrupt request is called the envelope. The envelope deals with context switching and directing the processor to the execution of the handler associated with the specific interrupt that has triggered the current response. The envelope may also be responsible for handling higher and lower priorities masking and unmasking. In order to minimize or even eliminate the overhead of saving the context of the task that is interrupted, the M•CORE architecture can switch to an alternate register file automatically by way of special encoding of each interrupt entry in the vector table. This means that the 16 user registers, the status register and the program counter of the current context will be kept intact during the processing of the interrupt. If the least significant bit in the interrupt vector is set, the processor automatically switches to the alternate register file upon entry in the interrupt service routine and switches back upon exit from the service routine.

The second software part is called the handler. It is specific to the interrupt request being serviced and is often dependent of the hardware being used to process the service.

Detect the highest priority to serve:

This segment of code handles the registers of the interrupt controller designed in the MMC2001 derivative. Software prioritization is performed here using a ‘Find First One’ instruction and a “Reverse Subtract’ instruction to generate the bit number of the highest interrupt request found in the Normal Pending Interrupt Register.

```
lrw    r5, [INT_CTLR]       // get INT CTRL address                      2 cyc
ld.w   r6, (r5, INT_NIPND)  // read pending list                         2 cyc
ff1    r6                   // get position of first bit 1 from left     1 cyc
rsubi  r6, 31               // scale it as bit number                    1 cyc
mtcr   r6, cr10             // temporary save it to SS4                  2 cyc
```

Generate masks for higher and lower priorities:

In this segment, it is necessary to mask the interrupt inputs with a lower priority level than the one being selected. It is also necessary to leave enabled the interrupt inputs with a higher priority level than the one being selected. This is what is being done here, using

For More Information On This Product,
Go to: www.freescale.com
Freescale Semiconductor, Inc.
FINAL

r4 as mask for the lower interrupts and r6 as mask for the higher interrupts. Register r5 holds the base address for the interrupt controller.

```
bgenr  r7, r6              // set this bit, need all those below         1 cyc
lsli   r7, 1               //                                            1 cyc
subi   r7, 1               // all below are set to one here              1 cyc
mov    r4, r7              // save copy of temp 'all lower'              1 cyc
ld.w   r6, (r5, INT_NIER)  // read mask in register                      2 cyc
and    r4, r6              // r4 = only lower currently enabled          1 cyc
andn   r6, r7              // r6 = only higher currently permitted       1 cyc
st.w   r6, (r5, INT_NIER)  // write back the higher ones                 2 cyc
```

Save minimum context (r4, r5, r6, r7):

To support re-entrance of the code (nested interrupts), it is necessary to save EPC and EPSR before they are destroyed by another coming interrupt. This is done here along with the saving of two other parameters that will be used upon returning from the handler. A 'Store Quadrant' (STQ) instruction is used to reduce the execution time. The context is saved to the stack, pointed to by r0.

```
mfcr   r7, cr4             // get context EPC                            2 cyc
mfcr   r6, cr2             // get context EPSR                           2 cyc
subi   r0, 16              // make room in stack (4 registers)           1 cyc
stq    r4-r7, (r0)         // save with mask & controller address        5 cyc
```

Prepare to jump to appropriate handler:

By design of the interrupt controller, interrupt requests activate the INT (or FINT) input to the processor. It is necessary to perform a software prioritization and discriminate which of the 32 interrupts has been granted service and which handler must be executed for this interrupt. This is done by using the interrupt number as an index to a table of addresses pointing to the 32 possible handlers.

```
lrw    r7, [it_tbl_add]    // point to the table of 32 handler addresses 2 cyc
mfcr   r6, cr10            // retrieve interrupt number                  2 cyc
ixw    r7, r6              // offset to vector table                     1 cyc
ld.w   r7, (r7,0)          // and get routine address                    2 cyc
```

Re-enable interrupts to allow nesting:

When the nesting of interrupts is required (handlers with long execution time), it is necessary to re-enable interrupt in the status register by setting the Interrupt Enable bit (IE). This is done after proper saving of the important registers (context saving).

```
mfcr   r6, cr0             // get PSR                                    2 cyc
bseti  r6, PSR_IE          // enable flag                                1 cyc
mtcr   r6, cr0             // write back                                 2 cyc
```

Go to handler:

Context has been saved, interrupts have been masked and unmasked accordingly, interrupts have been re-enabled. It is time to go to the selected handler. Address of this handler has been calculated before re-enabling the interrupts. Now, just execute a jump to subroutine so that the return address will be stored in r15 to be saved for re-entrance capability.

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

```
jsr    r7                  // NOW JUMP TO HANDLER                      2 cyc
```

Return from handler:

This is where the processor returns when the handler has been executed and completed. The tasks to be performed hereinafter are related to restoring the original context and returning to the program that had been suspended to respond to the interrupt.

Mask interrupts for safe context restoring:

Before any context or any interrupt mask can be restored it is necessary to disable all interrupts and operate in a stable configuration.

```
mfcr   r6, cr0             // read PSR                                 2 cyc
bclri  r6, PSR_IE          // mask normal interrupts                   1 cyc
mtcr   r6, cr0             // write back                               2 cyc
```

Restore context:

Context consists of two pointers and two critical registers that hold EPC and EPSR. They are restored using a ‘Load Quadrant’ instruction (LDQ) which loads four registers in 5 clock cycles. Then EPC and EPSR are loaded with their original contents.

```
ldq    r4-r7, (r0)         // retrieve saved context                   5 cyc
addi   r0, 16              // clean stack (4 registers)                1 cyc
mtcr   r6, cr2             // restore EPSR (re-entrance)               2 cyc
mtcr   r7, cr4             // restore EPC                              2 cyc
```

Re-enable lower interrupts:

At this point, it is necessary to restore the interrupt mask the way it was at the time of the interruption. The lower priority interrupts that were disabled must be restored. However, higher priority interrupts may have changed (they were enabled as higher priority). It is then necessary to take the current mask in effect for the higher priorities and keep them when writing back the mask for the lower priority interrupts. This is done by reading the current mask (which should have all lower interrupts disabled) and restore the old mask for these lower interrupts.

```
ld.w   r6, (r5, INT_NIER)  // read mask, highers may have changed      2 cyc
or     r6, r4              // add the lower we had disabled            1 cyc
st.w   r4, (r5, INT_NIER)  // write back, current higher, previous lower  2 cyc
```

Return from interrupt (restore PC and PSR):

Here, everything has been restored to its original state. Interrupts are still disabled and EPC and EPSR registers hold the original PC and PSR that were suspended. Executing a ‘Return From Exception’ will reload the PC and PSR system registers with their original values and the suspended program will resume. Interrupts will be re-enabled automatically when reloading PSR if they were enabled in the original program.

```
rte                        // return from exception (use EPC, EPSR)    3 cyc
```


For More Information On This Product,
Go to: www.freescale.com
Freescale Semiconductor, Inc.
FINAL

Freescale Semiconductor, Inc.

Typical handler base code:

There are many possible forms of interrupt handlers. They depend on the hardware involved and the specific task to perform. There are three basic actions that may be required as a minimum, the saving of the return address, the clearing of the associated interrupt request and the restoring of the return address. They are briefly described hereinafter.

Handler entry:

When entering the handler, it is necessary to save the return address. The nesting of other interrupts might destroy this return address during a jsr or a bsr instruction. This is done here by saving the return address to the stack (using r0 as stack pointer).

```
subi   r0, 4         // make room for 1 register = 4 bytes                1 cyc
st.w   r15, (r0,0)   // like saving done by JSR                           2 cyc
```

Exit handler:

When all tasks have been executed for the specific interrupt, the handler returns to the envelope for proper restoring of context and interrupt masks. Return is accomplished by retrieving the return address from the stack and executing the equivalent of a ‘return from subroutine’.

```
ld.w   r15, (r0, 0)        // retrieve return address                     2 cyc
addi   r0, 4               // clean stack (4 bytes)                       1 cyc
jmp    r15                 // and return to envelope                      2 cyc
```

Acknowledgment and clearing of the associated interrupt request:

Here is an example of code that can be written to clear the interrupt request that has caused the program to come here.

```
lrw    r4, [INT_CTLR]      // retrieve interrupt controller base address
ld.w   r5, (r4, INT_NIPND) // get interrupt pending register
ff1    r5                  // get bit position
rsubi  r5, 31              // and scale it
bgenr  r5, r5              // create a clearing mask
ld.w   r6, (r4, INT_SRC)   // read the source
andn   r6, r5              // clear the bit
st.w   r6, (r4, INT_SRC)   // and write back
```

FINAL

This code is handler dependent and may not be immune to register changes, especially when nesting of interrupts is permitted. It is the responsibility of the developer to guarantee a stable operation by properly masking and unmasking interrupts when necessary.

11 Auto-vectored, non-nested interrupts

This configuration reduces the overhead at the expense of potential longer ‘waiting time’ for some interrupts. Auto-vectoring implicitly uses the Fast Interrupt vector or the Normal Interrupt vector. To handle multiple fast or normal interrupts, auto-vectoring function requires

For More Information On This Product,
Go to: www.freescale.com

a software prioritization of the interrupts. Not nesting interrupts (making them non-interruptible) saves the bytes and clock cycles associated with saving and restoring the context as well as masking and unmasking of the other interrupts.

Detect the highest priority to serve:

This segment of code handles the registers of the interrupt controller designed in the MMC2001 derivative. Software prioritization is performed here using a 'Find First One' instruction and a "Reverse Subtract' instruction to generate the bit number of the highest interrupt request found in the Normal Pending Interrupt Register.

```
lrw    r5, [INT_CTLR]      // get INT CTRL address and keep in context    2 cyc
ld.w   r6, (r5, INT_NIPND) // read pending list                           2 cyc
ff1    r6                  // get position of first bit 1 from left       1 cyc
rsubi  r6, 31              // scale it as bit number                      1 cyc
```

Save minimum context (EPC and EPSR):

Although interrupts are defined as non-nested, normal interrupts can still be interrupted by exceptions. Exceptions have their dedicated handlers and can be written to take care of context saving. To support limited re-entrance of the code (nested exceptions), it is necessary to save EPC and EPSR before the processing of an exception destroys them. They are saved to scratch registers that will not be over-written since there is no re-entrance permitted.

```
mfcr   r7, cr4             // get context EPC                             2 cyc
mtcr   r7, cr6             // save context EPC                            2 cyc
mfcr   r7, cr2             // get context EPSR                            2 cyc
mtcr   r7, cr7             // save context EPSR                           2 cyc
```

Prepare to jump to appropriate handler:

Because of a software prioritization, all interrupt requests activate the same INT input to the processor. It is necessary to discriminate which of the 32 interrupts has been granted service and which handler must be executed for this interrupt. This is done by using the interrupt number as an index to a table of addresses pointing to the 32 possible handlers.

```
lrw    r7, [it_tbl_add]    // point to table of 32 handler addresses      2 cyc
ixw    r7, r6              // r6 holds the offset to vector table         1 cyc
ld.w   r7, (r7,0)          // and get routine address                     2 cyc
```

Go to handler:

Context has been saved, interrupts are kept disabled. It is time to go to the selected handler. Address of this handler has been acquired. Now, just execute a jump to subroutine so that the return address will be stored in r15 to be saved for re-entrance capability.

```
jsr    r7                  // NOW JUMP TO HANDLER                         2 cyc
```

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Return from handler:
This is where the processor returns when the handler has been executed and completed. The tasks to be performed here are related to restoring the original context and returning to the program that had been suspended to respond to the interrupt.

Restore context:
Context consists of two critical registers that hold EPC and EPSR. They are restored from the two scratch registers that held them during the execution of the handler. Then EPC and EPSR are loaded with their original contents.

```
mfcr   r7, cr6             // get EPC                                  2 cyc
mtcr   r7, cr4             // restore context EPC                      2 cyc
mfcr   r7, cr7             // get EPSR                                 2 cyc
mtcr   r7, cr2             // restore context EPSR                     2 cyc
```

Return from interrupt (restore PC and PSR):
Here, everything has been restored to its original state. Interrupts are still disabled and EPC and EPSR registers hold the original PC and PSR that were suspended. Executing a 'Return From Exception' will reload the PC and PSR system registers with their original values and the suspended program will resume. Interrupts will be re-enabled automatically when reloading PSR if they were enabled in the original program.

```
rte                        // return from exception (use EPC, EPSR)    3 cyc
```

12 Vectored, nested interrupts

This is a very flexible configuration that does not have the overhead required by the auto-vectoring feature. The handler vector number is provided by the interrupt mechanism, allowing the processor to pass control directly to the associated handler. This saves the bytes and clock cycles of the tasks associated with software prioritization. Context saving and restoring tasks are still required for the nesting of multiple interrupts. These tasks will be duplicated in each handler.

Take the interrupt number:
Vectored interrupts have the vector number provided to the processor. The number is used as an index to the interrupt vector table to get the address of the handler to execute. This means that the handler is immediately accessed and its number is known within this handler.

```
movi   r6, INT_nmb         // knows directly its interrupt number      1 cyc
```

Generate masks for higher and lower priorities:
In this segment, it is necessary to mask the interrupt inputs with a lower priority level than the one being selected. It is also necessary to leave enabled, the interrupt inputs with a higher priority level than the one being selected. This is what is being done here, keeping in r4 the mask for the lower interrupts and in r6 the mask for the higher interrupts. Register r5 holds the base address for the interrupt controller.

For More Information On This Product,
Go to: www.freescale.com
Freescale Semiconductor, Inc.
FINAL

```
bgenr  r7, r6              // set this bit, need all those below         1 cyc
lsli   r7, 1               //                                            1 cyc
subi   r7, 1               // all below are set to one here              1 cyc
mov    r4, r7              // save copy of temp 'all lower'              1 cyc
ld.w   r6, (r5, INT_NIER)  // read mask in register                      2 cyc
and    r4, r6              // r4 = only lower currently enabled          1 cyc
andn   r6, r7              // r6 = only higher currently permitted       1 cyc
st.w   r6, (r5, INT_NIER)  // write back the higher ones                 2 cyc
```

Save minimum context (r4, r5, r6, r7):

To support re-entrance of the code (nested interrupts), it is necessary to save EPC and EPSR before they are destroyed by another interrupt. This is done here along with the saving of two other parameters that will be used upon returning from the handler. A 'Store Quadrant' (STQ) instruction is used here to reduce the execution time. The context is saved to the stack, pointed to by r0.

```
mfcr   r7, cr4             // get context EPC                            2 cyc
mfcr   r6, cr2             // get context EPSR                           2 cyc
subi   r0, 16              // make room in stack (4 registers)           1 cyc
stq    r4-r7, (r0)         // save with mask & controller address        5 cyc
```

Re-enable interrupts to allow nesting:

When the nesting of interrupts is required (handlers with long execution time), it is necessary to re-enable interrupt in the status register by setting the Interrupt Enable bit (IE). This is done after proper saving of the important registers (context saving).

```
mfcr   r6, cr0             // get PSR                                    2 cyc
bseti  r6, PSR_IE          // enable flag                                1 cyc
mtcr   r6, cr0             // write back                                 2 cyc

// already in handler

// **********************************
// here, code for a specific handler
// **********************************
```

Mask interrupts for safe context restoring:

Before any context or any interrupt mask can be restored it is necessary to disable all interrupts and operate in a stable configuration.

```
mfcr   r6, cr0             // read PSR                                   2 cyc
bclri  r6, PSR_IE          // mask normal interrupts                     1 cyc
mtcr   r6, cr0             // write back                                 2 cyc
```

Restore context:

Context consists of two pointers and two critical registers that hold EPC and EPSR. They are restored using a 'Load Quadrant' instruction (LDQ) which loads four registers in 5 clock cycles. Then EPC and EPSR are loaded with their original contents.

```
ldq    r4-r7, (r0)         // retrieve saved context                     5 cyc
addi   r0, 16              // clean stack (4 registers)                  1 cyc
mtcr   r6, cr2             // restore EPSR (re-entrance)                 2 cyc
mtcr   r7, cr4             // restore EPC                                2 cyc
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

FINAL

Re-enable lower interrupts:

At this point, it is necessary to restore the interrupt mask the way it was at the time of the interruption. The lower priority interrupts that were disabled must be restored. However, higher priority interrupts may have changed (they were enabled as higher priority). It is then necessary to take the current mask in effect for the higher priorities and keep them when writing back the mask for the lower priority interrupts. This is done by reading the current mask (which should have all lower interrupts disabled) and restore the old mask for these lower interrupts.

```
ld.w   r6, (r5, INT_NIER)  // read mask, highers may have changed         2 cyc
or     r6, r4              // add the lower we had disabled                1 cyc
st.w   r4, (r5, INT_NIER)  // write back, current higher, previous lower  2 cyc
```

Return from interrupt (restore PC and PSR):

Here, everything has been restored to its original state. Interrupts are still disabled and EPC and EPSR registers hold the original PC and PSR that were suspended. Executing a 'Return From Exception' will reload the PC and PSR system registers with their original values and the suspended program will resume. Interrupts will be re-enabled automatically when reloading PSR if they were enabled in the original program.

```
rte                        // return from exception (use EPC, EPSR)       3 cyc
```

13 Vectored, non nested interrupts

This is the most time-effective configuration for normal interrupts. It has the smallest overhead at the expense of a larger 'waiting time' for some interrupts. However, the gain in execution time offered by this configuration will, in many cases, compensate the 'waiting time'. In short, the interrupt will have been identified, processed and completed in the time spent saving and restoring the context.

Save minimum context (EPC and EPSR):

Although interrupts are defined as non-nested, normal interrupts can still be interrupted by exceptions. Exceptions have their dedicated handlers and can be written to take care of context saving. To support limited re-entrance of the code (nested exceptions), it is necessary to save EPC and EPSR before the processing of an exception destroys them. They are saved to scratch registers that will not be over-written since there is no re-entrance permitted.

```
mfcr   r7, cr4             // get context EPC                              2 cyc
mtcr   r7, cr6             // save context EPC                             2 cyc
mfcr   r7, cr2             // get context EPSR                             2 cyc
mtcr   r7, cr7             // save context EPSR                            2 cyc

// already in appropriate handler

// ***********************************
// here, code for a specific handler
// ***********************************
```


For More Information On This Product,
Go to: www.freescale.com
Freescale Semiconductor, Inc.
FINAL

Restore context:

Context consists of two critical registers that hold EPC and EPSR. They are restored from the two scratch registers that held them during the execution of the handler. Then EPC and EPSR are loaded with their original contents.

```
mfcr   r7, cr6             // get EPC                               2 cyc
mtcr   r7, cr4             // restore context EPC                   2 cyc
mfcr   r7, cr7             // get EPSR                              2 cyc
mtcr   r7, cr2             // restore context EPSR                  2 cyc
```

Return from interrupt (restore PC and PSR):

Here, everything has been restored to its original state. Interrupts are still disabled and EPC and EPSR registers hold the original PC and PSR that were suspended. Executing a 'Return From Exception' will reload the PC and PSR system registers with their original values and the suspended program will resume. Interrupts will be re-enabled automatically when reloading PSR if they were enabled in the original program.

```
rte                        // return from exception (use EPC, EPSR)      3 cyc
```

14 Fast vectored, non nested interrupts

It has virtually no overhead. The only penalty is the latency that may affect other pending interrupts. This solution however should be implemented when the handler is known to be very short and when the overhead of implementing a re-entrance solution is larger than the interrupt processing itself. The vectored capability allows the interrupt request to go directly to the execution of the handler. The non-nested feature saves the many instructions required to save and restore the context. The use of the fast interrupt input guarantees that potential exceptions will not overwrite FPC and FPSR, thus saving the few remaining instructions in a context saving task.

```
already in appropriate handler when entering the interrupt service

// **********************************
// here, code for a specific handler
// **********************************
```

Return from interrupt (restore PC and PSR):

Here, interrupts are still disabled and FPC and FPSR registers still hold the original PC and PSR that were suspended. Executing a 'Return From Fast Interrupt' will reload the PC and PSR system registers with their original values and the suspended program will resume. Interrupts will be re-enabled automatically when reloading PSR if they were enabled in the original program.

```
rfi                        // return from exception (use FPC, FPSR)      3 cyc
```

FINAL
Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

15 Summary for 32 interrupt requests

The following table gives a close estimate of the number of bytes and clock cycles used in the different interrupt handling techniques.

tasks	Auto-Vectored Nested	Auto-Vectored Non-Nested	Vectored INT Nested	Vectored INT Non-Nested	Vectored FINT Non-Nested	notes
H/W latency	(6-9 cyc)	(6-9 cyc)	(6-9 cyc)	(6-9 cyc)	(6-9 cyc)	synchronization & wait for instr. to complete
S/W prioritization of interrupt sources	20 (17 cyc)	16 (18 cyc)	2 (1 cyc)	-	-	identify highest priority and handler
nesting for pre-emptive interrupt handling	50 (45 cyc)	16 (16 cyc)	50 (45 cyc)	16 (16 cyc)	-	save / restore context and interrupts
handler overhead	10 (8 cyc)	10 (8 cyc)	-	-	-	specific save / restore context for handler
others	2 (3 cyc)	2 (3 cyc)	2 (3 cyc)	2 (3 cyc)	2 (3 cyc)	basic tasks
TOTAL bytes (cycles)	82 (82 cyc)	44 (49 cyc)	54 (58 cyc)	18 (28 cyc)	2 (9-12 cyc)	

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

16 Serving more than 32 interrupt requests

The software examples that have been given here are designed to handle the 32 interrupts lines of the MMC2001's on-chip interrupt controller, some of them being connected to internal resources. There are applications that will require a higher number of interrupts. The extra overhead for the M·CORE architecture to support more than 32 interrupts, say 64, 96, 128 or 256 interrupts can be evaluated for the various types of interrupt handling models that have been discussed.

Auto-vectored and nested interrupts
This model requires a software implementation of the request prioritization. It requires additional code to scan the various 32-bit registers holding the status bits and to modify the registers holding the mask patterns for each block of interrupt lines. This is hardware dependent and the code required depends on how the information is made available to the user. By keeping track of the 32-bit block scanned and the number of inputs checked during the Find-First-One instruction, the interrupt request number is easily calculated as an offset to the interrupt jump table.

Auto-vectored and non-nested interrupts
Since this model does not nest interrupts, the problem of handling the interrupt masks does not exist. Limited additional code is required here to handle interrupt request prioritization by looking at the various registers holding the status of the interrupt requests. This is hardware dependent and the code required depends on how the information is made available to the user.

Vectored and nested interrupts
The model uses a vectored mode, removing the need for interrupt request prioritization. The additional code required is reduced since there is no need for scanning multiple registers in order to find where the highest requesting input is. Some additional code is still required to handle the multiple interrupt masks to support nesting capability.

Vectored and non-nested interrupts
This model too uses a vectored mode, removing the need for interrupt request prioritization. The additional code required is greatly reduced since there is no need for scanning multiple registers in order to find where the highest requesting input is and there is no need either to handle multiple mask registers.

Fast vectored non-nested interrupts
This model, due to a very reduced software implementation, requires no additional code when increasing the number of interrupt inputs to 64, 96, 128 or even 256. The vectoring mechanism takes care of the interrupt request prioritization and the selection of the appropriate interrupt vector in the interrupt vector table. With this model, the envelope and the handler are combined into one software module.

FINAL
Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

17 Summary for up to 256 interrupts

The following table gives an estimate of the number of bytes and clock cycles required in implementing up to 256 interrupts using the different interrupt handling techniques.

Number of Interrupts	Auto-Vectored Nested	Auto-Vectored Non-Nested	Vectored INT Nested	Vectored INT Non-Nested	Vectored FINT Non-Nested	notes
1 to 32	82 (82 cyc)	44 (49 cyc)	54 (58 cyc)	18 (28 cyc)	2 (9-12 cyc)	
up to 64	92 (101 cyc)	51 (58 cyc)	60 (69 cyc)	18 (28 cyc)	2 (9-12 cyc)	estimated
up to 96	97 (107 cyc)	56 (64 cyc)	60 (69 cyc)	18 (28 cyc)	2 (9-12 cyc)	estimated
up 128	102 (113 cyc)	61 (70 cyc)	60 (69 cyc)	18 (28 cyc)	2 (9-12 cyc)	estimated
up to 256	122 (136 cyc)	81 (93 cyc)	60 (69 cyc)	18 (28 cyc)	2 (9-12 cyc)	estimated

18 Conclusion

The M•CORE architecture offers a wide range of solutions to handle interrupt requests. This is due to its interrupt mechanism, its instruction set and its register set model. Expanding to more than 32 interrupt sources can easily be done at almost no expense in bytes or clock cycles. The M•CORE architecture is perfectly suited for real-time applications requiring very fast response time, very small overhead with the capability to handle more than a hundred interrupts at rates above 10,000 interrupt requests per second.

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

19 Appendix 1: Sample code for Autovectored, Nested Interrupts

```
       IRQ_Envelope

; Constants for accessing IRQ Interrupt Controller (MMC2001)
INT_NIER      EQU    0x4           ; Normal Interrupt Enable Register
INT_SRC       EQU    0x0           ; Interrupt Source Register
INT_NIPND     EQU    0xC           ; Normal Interrupt Pending Register
; IRQ disable bit constants
PSR_IE        EQU    0x40
PSR_IC        EQU    0x7


; ********************************************************************************
; * Start of actual code for Normal Interrupt. This is the routine that should    *
; * be branched to from the INT vector.The module is re-entrant.                 *
; * INT only uses r1-r7.      r0 is Stack, r15 holds return address              *
; ********************************************************************************


// Detect
; when entering, registers r4-r7 can be used at will (EABI specifications)
       LRW    r5, [INT_CTLR]      // get INT CTRL address and keep in context
       LD.W   r6, (r5, INT_NIPND) // read pending list
       FF1    r6                  // get position of first bit 1 from left
       rsubi  r6, 31              // scale it as bit nmb (b30 was 1, now 30)
       mtcr   r6, cr10            // temporary save it to SS4

// mask
       BGENR  r7, r6              // set this bit, need all those below
       LSLI   r7, 1               // mult by 2
       SUBI   r7, 1               // and adjust
       MOV    r4, r7              // save copy of temp 'all lower'
       LD.W   r6, (r5, INT_NIER)  // read mask in register
       AND    r4, r6              // r4 = only lower currently enabled
       ANDN   r6, r7              // r6 = only higher currently permitted
       ST.W   r6, (r5, INT_NIER)  // write back the higher ones

// save context (r4, r5, r6, r7)
       MFCR   r7, cr4             // get context EPC
       MFCR   r6, cr2             // get context EPSR
       SUBI   r0, 16              // make room in stack (4 registers)
       STQ    r4-r7, (r0)         // save with mask & controller address

// prepare to jump to appropriate handler
       LRW    r7, [it_tbl_add]
       MFCR   r6, cr10            // retrieve interrupt number
       IXW    r7, r6              // offset to vector table
       LD.W   r7, (r7,0)          // and get routine address

// re-enable interrupts to allow nesting
       MFCR   r6, cr0             // get PSR
       BSETI  r6, PSR_IE          // enable flag
       MTCR   r6, cr0             // write back

// go to handler,
       JSR    r7                  // NOW JUMP TO HANDLER


// mask interrupts for safe context restoring
       MFCR   r6, cr0             // read PSR
       BCLRI  r6, PSR_IE          // mask normal interrupts
       MTCR   r6, cr0             // write back

// restore context
       LDQ    r4-r7, (r0)         // retrieve saved context
       ADDI   r0, 16              // clean stack (4 registers)
       MTCR   r6, cr2             // restore EPSR (re-entrance)
       MTCR   r7, cr4             // restore EPC
```

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

```
// re-enable lower interrupts
       LD.W   r6, (r5, INT_NIER)  // read mask, highers may have changed
       OR     r6, r4              // add the lower we had disabled
       ST.W   r4, (r5, INT_NIER)  // write back, current higher and old lowers

// return from interrupt (restore PC and PSR)
       RTE                        // return from exception (use PC,EPSR)


       .align 2
it_tbl_add:   .long  int_table            // address of first (for a LRW instr).
INT_CTLR:     .long  0x10000000           // address of interrupt controller

// table of handler start addresses
int_table:
itbit00:      .long  hdlr_loop            // lowest priority
// ...........................
itbit29:      .long  Interrupt_3
itbit30:      .long  Interrupt_2
itbit31:      .long  Interrupt_1          // highest priority

// Example
Interrupt_1:
       SUBI   r0, 4         // make room for 1 register = 4 bytes
       ST.W   r15, (r0,0)   // like saving done by JSR

// **********************************
// here, code for a specific handler
// **********************************

// exit handler
       LD.W   r15, (r0, 0)        // retrieve return address
       ADDI   r0, 4               // clean stack (4 bytes)
       JMP    r15                 // and return to envelope

Interrupt_2:
Interrupt_3:
; as per Interrupt_1, but with own handler code.



// example how to clear pending interrupt
       LRW    r4, [INT_CTLR]      //
       LD.W   r5, (r4, INT_NIPND) // get pending register
       FF1    r5                  // get position of bit
       RSUBI  r5, 31              // and scale it
       BGENR  r5, r5              //
       LD.W   r6, (r4, INT_SRC)   // read the source
       ANDN   r6, r5              // clear the bit
       ST.W   r6, (r4, INT_SRC)   // write back



END
```

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

20 Appendix 2: Sample code for Auto-Vectored, Non-Nested Interrupts

```
       IRQ_Envelope

; Constants for accessing IRQ Interrupt Controller (MMC2001)
INT_NIER      EQU    0x4           ; Normal Interrupt Enable Register
INT_SRC       EQU    0x0           ; Interrupt Source Register
INT_NIPND     EQU    0xC           ; Normal Interrupt Pending Register
; IRQ disable bit constants
PSR_IE        EQU    0x40
PSR_IC        EQU    0x7


; ***************************************************************************
; * Start of actual code for Normal Interrupt. This is the routine that should     *
; * be branched to from the INT vector.The module is re-entrant.                   *
; * INT only uses r1-r7.      r0 is Stack, r15 holds return address               *
; ***************************************************************************


// Detect
; when entering, registers r4-r7 can be used at will (EABI specifications)
       LRW    r5, [INT_CTLR]       // get INT CTRL address and keep in context
       LD.W   r6, (r5, INT_NIPND) // read pending list
       FF1    r6                   // get position of first bit 1 from left
       rsubi  r6, 31               // scale it as bit nmb (b30 was 1, now 30)

// save context (r4, r5, r6, r7)
       MFCR   r7, cr4              // get context EPC
       MTCR   r7, cr6              // save EPC (only once)
       MFCR   r7, cr2              // get context EPSR
       MTCR   r7, cr7              // save EPSR (only once)

// prepare to jump to appropriate handler
       LRW    r7, [it_tbl_add]
       IXW    r7, r6               // offset to vector table
       LD.W   r7, (r7,0)           // and get routine address

// go to handler,
       JSR    r7                   // NOW JUMP TO HANDLER


// restore context
       MFCR   r7, cr6              // get EPC
       MTCR   r7, cr4              // restore context EPC
       MFCR   r7, cr7              // get EPSR
       MTCR   r7, cr2              // restore context EPSR

// return from interrupt (restore PC and PSR)
       RTE                         // return from exception (use PC,EPSR)
```

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

```
       .align 2
it_tbl_add:   .long  int_table           // address of first (for a LRW instr).
INT_CTLR:     .long  0x10000000          // address of interrupt controller

// table of handler start addresses
int_table:
itbit00:      .long  hdlr_loop           // lowest priority
// ...........................
itbit29:      .long  Interrupt_3
itbit30:      .long  Interrupt_2
itbit31:      .long  Interrupt_1         // highest priority

// Example
Interrupt_1:
       SUBI   r0, 4         // make room for 1 register = 4 bytes
       ST.W   r15, (r0,0)   // like saving done by JSR

// **********************************
// here, code for a specific handler
// **********************************

// exit handler
       LD.W   r15, (r0, 0)         // retrieve return address
       ADDI   r0, 4                // clean stack (4 bytes)
       JMP    r15                  // and return to envelope

Interrupt_2:
Interrupt_3:
; as per Interrupt_1, but with own handler code.



// example how to clear pending interrupt
       LRW    r4, [INT_CTLR]       //
       LD.W   r5, (r4, INT_NIPND) // get pending register
       FF1    r5                   // get position of bit
       RSUBI  r5, 31               // and scale it
       BGENR  r5, r5               //
       LD.W   r6, (r4, INT_SRC)    // read the source
       ANDN   r6, r5               // clear the bit
       ST.W   r6, (r4, INT_SRC)    // write back



END
```

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

21 Appendix 3: Sample code for Vectored, Nested Interrupts

```
       IRQ_Envelope

; Constants for accessing IRQ Interrupt Controller (MMC2001)
INT_NIER      EQU    0x4          ; Normal Interrupt Enable Register
INT_SRC       EQU    0x0          ; Interrupt Source Register
INT_NIPND     EQU    0xC          ; Normal Interrupt Pending Register
; IRQ disable bit constants
PSR_IE        EQU    0x40
PSR_IC        EQU    0x7

; ********************************************************************************
; * INT only uses r1-r7.      r0 is Stack, r15 holds return address             *
; ********************************************************************************

// Detect
; when entering, registers r4-r7 can be used at will (EABI specifications)
       MOVI   r6, INT_nmb          // get interrupt number

// mask
       BGENR  r7, r6               // set this bit, need all those below
       LSLI   r7, 1                // mult by 2
       SUBI   r7, 1                // and adjust
       MOV    r4, r7               // save copy of temp 'all lower'
       LD.W   r6, (r5, INT_NIER)   // read mask in register
       AND    r4, r6               // r4 = only lower currently enabled
       ANDN   r6, r7               // r6 = only higher currently permitted
       ST.W   r6, (r5, INT_NIER)   // write back the higher ones

// save context (r4, r5, r6, r7)
       MFCR   r7, cr4              // get context EPC
       MFCR   r6, cr2              // get context EPSR
       SUBI   r0, 16               // make room in stack (4 registers)
       STQ    r4-r7, (r0)          // save with mask & controller address

// re-enable interrupts to allow nesting
       MFCR   r6, cr0              // get PSR
       BSETI  r6, PSR_IE           // enable flag
       MTCR   r6, cr0              // write back

// already in handler

// **********************************
// here, code for a specific handler
// **********************************

// mask interrupts for safe context restoring
       MFCR   r6, cr0              // read PSR
       BCLRI  r6, PSR_IE           // mask normal interrupts
       MTCR   r6, cr0              // write back

// restore context
       LDQ    r4-r7, (r0)          // retrieve saved context
       ADDI   r0, 16               // clean stack (4 registers)
       MTCR   r6, cr2              // restore EPSR (re-entrance)
       MTCR   r7, cr4              // restore EPC

// re-enable lower interrupts
       LD.W   r6, (r5, INT_NIER)   // read mask, highers may have changed
       OR     r6, r4               // add the lower we had disabled
       ST.W   r4, (r5, INT_NIER)   // write back, current higher and old lowers

// return from interrupt (restore PC and PSR)
       RTE                         // return from exception (use PC,EPSR)
```

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

```
        .align 2
it_tbl_add:    .long  int_table            // address of first (for a LRW instr).
INT_CTLR:      .long  0x10000000           // address of interrupt controller



Interrupt_2:
Interrupt_3:
; as per Interrupt_1, but with own handler code.




// example how to clear pending interrupt
        LRW    r4, [INT_CTLR]       //
        BSETI  r5, INT_nmb          // set bit for this interrupt number
        LD.W   r6, (r4, INT_SRC)    // read the source
        ANDN   r6, r5               // clear the bit
        ST.W   r6, (r4, INT_SRC)    // write back


END
```

Freescale Semiconductor, Inc.
FINAL

For More Information On This Product,
Go to: www.freescale.com

22 Appendix 4: Sample code for Vectored, Non-Nested Interrupts

```
       IRQ_Envelope

; Constants for accessing IRQ Interrupt Controller (MMC2001)
INT_NIER      EQU    0x4           ; Normal Interrupt Enable Register
INT_SRC       EQU    0x0           ; Interrupt Source Register
INT_NIPND     EQU    0xC           ; Normal Interrupt Pending Register
; IRQ disable bit constants
PSR_IE        EQU    0x40
PSR_IC        EQU    0x7


; ***************************************************************************
; * INT only uses r1-r7.      r0 is Stack, r15 holds return address        *
; ***************************************************************************


// save context
       MFCR   r7, cr4              // get context EPC
       MTCR   r7, cr6              // save EPC (only once)
       MFCR   r7, cr2              // get context EPSR
       MTCR   r7, cr7              // save EPSR (only once)

// already in appropriate handler

// **********************************
// here, code for a specific handler
// **********************************

// restore context
       MFCR   r7, cr6              // get EPC
       MTCR   r7, cr4              // restore context EPC
       MFCR   r7, cr7              // get EPSR
       MTCR   r7, cr2              // restore context EPSR
//
// return from interrupt (restore PC and PSR)
       RTE                         // return from exception (use PC,EPSR)


       .align 2
it_tbl_add:   .long  int_table            // address of first (for a LRW instr).
INT_CTLR:     .long  0x10000000           // address of interrupt controller



Interrupt_2:
Interrupt_3:
; as per Interrupt_1, but with own handler code.



// example how to clear pending interrupt
       LRW    r4, [INT_CTLR]       //
       BSETI  r5, INT_nmb          // set bit for this interrupt number
       LD.W   r6, (r4, INT_SRC)    // read the source
       ANDN   r6, r5               // clear the bit
       ST.W   r6, (r4, INT_SRC)    // write back



END
```

FINAL

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

23 Appendix 5: Sample code for Vectored, Non-Nested Fast Interrupts

```
       IRQ_Envelope

; Constants for accessing IRQ Interrupt Controller (MMC2001)
INT_SRC       EQU    0x0          ; Interrupt Source Register
INT_FIPND     EQU    0x10         ; Fast Interrupt Pending Register


; ***********************************************************************************
; * r0 is Stack, r15 holds return address                                           *
; ***********************************************************************************


// already in appropriate handler

// **********************************
// here, code for a specific handler
// **********************************


// return from interrupt (restore PC and PSR)
       RFI                        // return from exception (use FPC,FPSR)




Interrupt_2:
Interrupt_3:
; as per Interrupt_1, but with own handler code.



// example how to clear pending interrupt
       LRW    r4, [INT_CTLR]      //
       BSETI  r5, INT_nmb         // set bit for this interrupt number
       LD.W   r6, (r4, INT_SRC)   // read the source
       ANDN   r6, r5              // clear the bit
       ST.W   r6, (r4, INT_SRC)   // write back



END
```

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.
Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

FINAL

Freescale Semiconductor, Inc.