Application Note

AN2220/D
Rev. 0, 12/2001

Vectored Interrupt Handling on the M·Core MMC2107

by Brian LaPonsey
M-Core Applications Group
Freescale, East Kilbride

Freescale Semiconductor, Inc.

Abstract

This application note addresses the need for a more in-depth discussion of vectored interrupt handling on the M·Core MMC2107 microprocessor. Current documentation is sparse, with few detailed examples. Metrowerks provides two different, but closely related tool chains for M·Core, one with the classic CodeWarrior compiler/debugger integrated development environment (IDE), and one that incorporates the tools previously offered by Hiware. Code for both of the Metrowerks M·Core IDEs is demonstrated in two projects using a high-level programming language, and the main differences between the versions are highlighted.

The target audience is the advanced engineering student, or the system engineer who is new to the MMC2107 device and who may be unfamiliar with the use of a high-level language for programming embedded applications. The intent is to give these programmers a head start by offering sample solutions to some of the more common obstacles. These examples should be used as a starting point from which the engineer can progress independently. They are not meant to be a "manual of best practice," so much as an example of "what worked in this case," and the engineer is encouraged to accept, reject, or adapt these examples as appropriate. The practice of hiding key parameters behind multiple references and redefinitions has been avoided. The goal has been to expose the workings of the application with transparent code, not to produce production-quality reusable modules.

The introduction speaks briefly about the history and purpose of the M·Core architecture, and presents the MMC2107 integrated processor.

The second section provides an overview of the concept of interrupts, and defines some terminology. Some of the basic goals of interrupt processing are discussed.

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

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

The third section introduces some of the important features of the M·Core M210 central processing unit, and illustrates how these allow the core to optimize interrupt handling for real-time control.

The fourth section highlights the specific features of the MMC2107 that allow it to make full use of the M·Core architecture. The advantages of vectoring over autovectoring are discussed, and the fast and normal interrupt types are introduced.

Sections 5 and 6 describe in detail the sequence of events that occurs when an interrupt request is made to the MMC2107. The Interrupt Controller Module is introduced, and the construction of the vector table is explained. A hypothetical interrupt sequence is analyzed.

Sections 7 and 8 discuss further optimization methods and develop the use of the alternate register file. Some strategies are suggested on how to make use of the fast interrupt type, and how to gain the most benefit from it. A potential pitfall for high-level language programmers is highlighted, and a method for avoiding this problem is discussed.

Section 9 provides a background of the tools used in the sample application project.

The appendices contain a glossary and explanatory notes on the CodeWarrior projects that make use of the concepts developed. The example employs multiple types and sources of interrupt, and the alternate register file. The M·Core instruction set architecture is tailored to support high-level languages, so the example uses C language whenever possible.

I. Introduction

It is often said that “form follows function,” and in the microprocessor industry, this adage certainly applies. Microprocessors are designed and built to fulfill the expected requirements of typical systems, and the system designer will select a processor that best fits the special needs of the application. Speed is always a consideration, but in many circumstances, speed may not be the only, or even the most important issue.

One industry requirement that has regularly appeared over the past decade is for good real-time performance with low power consumption for extended battery life in portable applications. In 1993, Freescale began research into a new microprocessor architecture that would provide compact code, good power efficiency and excellent real-time response for these portable embedded systems. The M·Core M200 RISC processor core was the result of this research. In addition to its efficiency, the M·Core also uses a fixed-length 16-bit instruction set to optimize code density, and has impressive interrupt handling

Freescale Semiconductor, Inc.

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

capabilities that provide a deterministic interrupt response even at modest clock frequencies.

Since the M·Core architecture's public introduction in 1997, there have been several microprocessors based on the M200 core, but only one was targeted toward the general-purpose embedded controller market. The MMC2001, a 34MHz, 32-bit ROM device, does not implement the entire M200 feature set, and this prevents programmers from fully optimizing its interrupt response behavior. Although an excellent choice for many portable applications, the MMC2001 is not always an ideal solution for real-time control.

In July 2000, Freescale introduced the first member of a new family of general-purpose 32-bit microcontrollers based on the M·Core M210 processor core.[1] The MMC2107 is the first general-purpose M·Core device to fully utilize the M210's extensive interrupt-processing capabilities. This application note will investigate the MMC2107's enhanced functionality.

II. Interrupt Processing and Latency

An interrupt[2] is the redirection of a program's normal flow of execution triggered by an asynchronous external event. When an interrupt occurs, the core has to branch to an *interrupt service routine*, a special set of instructions written specifically to deal with this event. In real-time embedded applications, the way a processor reacts to these events is critical to the system's performance. Computer-controlled machine tools, data acquisition equipment, robotic devices, and communication networks are types of systems that may require interrupt servicing.

When an external event occurs that requires interrupt service, a signal must be sent to the processor indicating that this event has taken place. This signal is called an *interrupt request.* The time elapsed between the moment the external event occurs and the moment its *interrupt service routine* (ISR) begins to execute is called the *latency*. The latency will vary depending on other tasks that the processor is performing when the interrupt request arrives. It is always possible to improve a system's latency by increasing the clock frequency, but this has the unwanted side effects of increasing the power drain and electromagnetic interference (EMI). This is acceptable if power consumption

1. The M210 is essentially just the M200 core with the addition of bus arbitration.

2. Much of the documentation for the M·Core family uses the term *exceptions* as well as the term *interrupts*. In many cases these words appear to be interchangeable, but interrupts are actually a subset of exceptions. An interrupt is specifically an intentional user-defined exception, from a source such as a timer module or an A/D converter. Exceptions also include other unexpected events like a divide-by-zero, or a misaligned memory access error. These internally-generated system exceptions also need service routines, and have their own vector table entries pre-assigned to them in positions 0-31 (see Table 1).

Freescale Semiconductor, Inc.

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

and EMI are not critical issues, but for many applications this is not always a good compromise.

A more useful measure of latency is in processor clock cycles. This is independent of clock frequency, allowing the system designer to compare different processors without regard to speed of operation. The designer can calculate the lowest acceptable clock frequency based on the maximum acceptable response time, thereby meeting the latency requirement while minimizing power consumption and EMI.

In real-time systems, it is not simply the best-case latency that determines the system performance. Critical tasks often must be guaranteed to execute within a bounded time interval, so the worst-case latency is just as important. If the difference between the best-case and worst-case is relatively small, then the system will service the critical interrupt within a narrowly defined and reliable time window. Systems like these are said to be *deterministic*, and they are highly valued in real-time applications for their ability to react consistently and predictably to asynchronous events.

When an interrupt request is received, it needs to be *recognized*, *identified* and *serviced*. Recognition means that the core becomes aware that an interrupt has occurred and suspends the current process. Identification means that the core uniquely associates the interrupt request with a particular source. Servicing the interrupt begins when the program flow branches to the beginning of the ISR for that interrupt source.

Interrupt requests cannot always be recognized, because there are times when the core is preoccupied with other tasks. For example, it would not be reasonable to expect any system to recognize interrupts during a power-up initialization sequence. Another situation requiring interrupts to be disabled would be during a critical operation, one which the programmer has deemed to be more important than any other possible interrupt request. An example of this might be a two-way communication system.   If a data transmission is interrupted, the information can probably be sent again later. Interrupt a data reception, and its content may be lost forever.

All processors capable of dealing with interrupts must therefore have a way of enabling and disabling interrupts from within the application. More advanced devices will also have a system for choosing the priority of different interrupt sources, so that higher priority interrupts will be serviced preferentially over lower priority ones.

Even when interrupts are enabled, the processor still may not be able to respond instantly to an interrupt request. Most microprocessor instruction sets contain commands requiring more than one clock cycle to execute. If an interrupt request occurs in the middle of the execution of a multi-clock instruction, the request usually has to wait until the instruction is finished before it can be recognized.

Freescale Semiconductor, Inc.

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

Freescale Semiconductor, Inc.

If there is more than one source of interrupt, an important design decision to be considered is whether some high-priority interrupts will be able to pre-empt the execution of other interrupt service routines. The ability of a higher-priority ISR to interrupt a lower-priority one is key to the design of deterministic systems. The lower-priority ISR resumes execution when the higher priority ISR is complete. This is called *nesting*, and often requires a carefully constructed scheme of prioritization and masking.

Once an interrupt request is recognized, the core must identify the source of the request. Somewhere in memory are the instructions of an ISR written to service that interrupt source. The processor must somehow associate the interrupt request with the address of that code, and then branch to that location and begin execution. One way to do this is to arrange for each interrupt source to provide the core with a *vector number* along with the request. The core can then use that number to look up the ISR address from a dedicated block of memory called a *vector table*.

If no vector number is available, or if there are more potential sources of interrupt than there are entries available in the vector table, an *interrupt handler* must be written to poll the available sources to find out which one was responsible for the request. The address of this handler is then programmed into the vector table. Once the handler identifies the interrupt source, it either branches to an appropriate section of code within the handler, or calls a separate function to process the interrupt. These look-up, address fetch and branch operations all take time, and careful optimization of these steps can significantly improve the system latency.

After the interrupt has been serviced, the program's normal flow of execution must be able to resume normally from the point the interrupt was recognized. The only way to accomplish this is to save the machine state that existed just before the interrupt occurred, and restore it after the ISR completes. This machine state is often called the *context*, and it must be restored to its former state to avoid corrupting the execution of the processor's other tasks. A complete context save includes the program counter (PC) and processor status registers (PSR), any data registers the ISR uses, the stack pointer, and often other special-purpose registers depending on the particular processor architecture.[3] Any of these that might be altered by the interrupt service routine must be restored to their original state before the main application resumes processing. If interrupt nesting is used, then several levels of context may need to be stored, and the time it takes to do this can affect the system latency.

3. See M·Core Applications Binary Interface Standards Manual, Section 2.2.1 "Register Assignments"

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

Freescale Semiconductor, Inc.

III. Interrupt Processing on M·Core

The M·Core architecture was designed from the outset for excellent real-time response in embedded applications, and many features were added to make this possible. Since the power-efficient M2xx cores are not intended to run at high frequencies, they need other ways to improve their interrupt latency.

Deterministic interrupt response was accomplished by using an instruction-restart model for processing interrupts. If an interrupt request occurs while the core is processing a multi-clock instruction, an optional setting allows that instruction to be aborted and restarted later, after the interrupt has been serviced. This provides a best-case latency of 5 clocks, and a worst-case latency of 9 clocks before an interrupt service routine begins to execute. These performance figures represent the capability of the hardware. Software methods for realizing this performance level will be discussed later.

It is futile to recognize an interrupt request quickly if it takes a long time to save the context. To help resolve this problem, the M·Core architecture defines two types of interrupt – *normal* and *fast*. This is more than just a simple two-level priority definition. Normal and fast correspond to the two available interrupt request signal paths into the core, and these different signals are used to determine how the context is saved. Instead of saving the PC and PSR on the stack, the M210 saves them in dedicated 32-bit shadow registers provided just for that purpose. Moreover, there are two sets of these registers – one set for normal interrupts (EPC and EPSR) and one for fast interrupts (FPC and FPSR). There is no need to execute instructions to save the PC or PSR, because the core saves them automatically as part of the sequence triggered by the interrupt request.

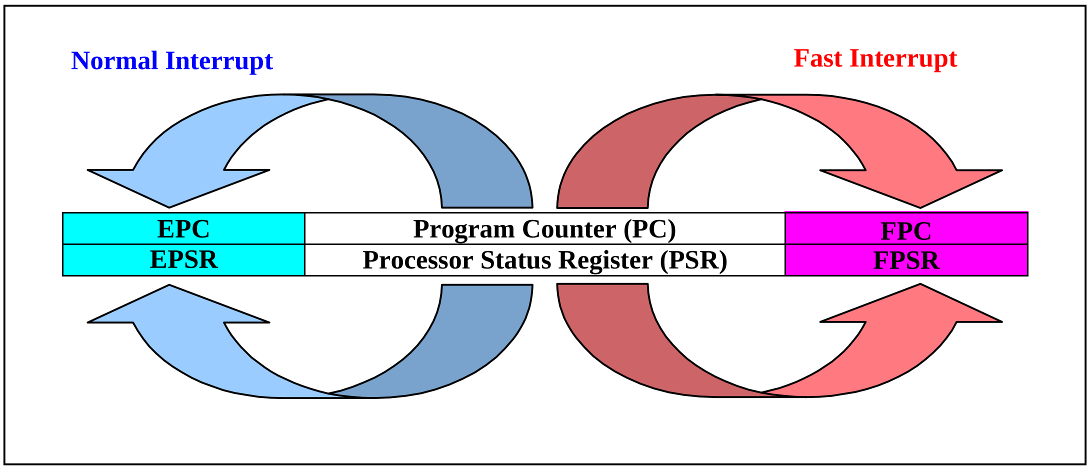

Figure 1. Context Save for Normal and Fast Interrupts

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

Freescale Semiconductor, Inc.

The M210 programming model also offers two complete sets, or “files”, of general purpose data registers – 16 regular registers and 16 alternates (Figure 2). Using the alternate register file provides a significant speed advantage when processing time-critical interrupts, because it allows a high-priority interrupt to use an independent set of registers, thereby eliminating the need to save the register context.

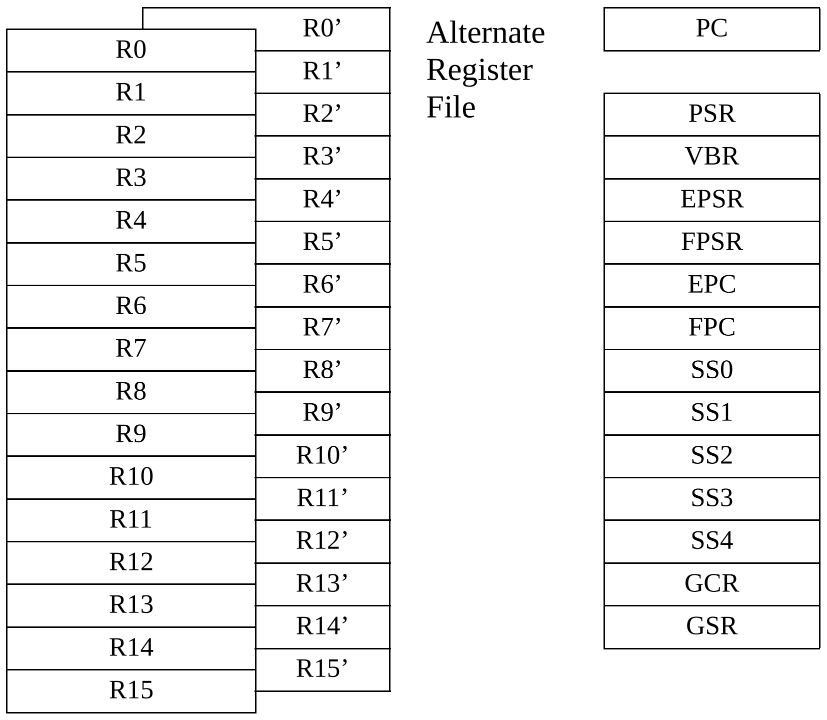

Figure 2. M210 Supervisor programming model

The state of the AF (Alternate File) bit in the PSR determines which register file is active. In practice, the PSR(AF) bit does not usually need to be explicitly set or cleared by the application. An example will be provided later explaining how and when PSR(AF) bit is set, and some potential problems with using the alternate file will also be discussed.

Freescale Semiconductor, Inc.

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

IV. MMC2107 Improves on MMC2001

All M·Core-based architectures dedicate a 512-byte contiguous block of memory to be used as a vector table to hold the addresses of all exception handlers and user-defined interrupt service routines. No M·Core device actually has enough interrupt sources to fill the entire table, and there are many locations that are reserved for future expansion. Table 1 shows the assignments for the memory locations within the M210 vector table. The table entries shaded in blue are for the addresses of user-defined ISRs.

Table 1. Vector Table Assignments

Vector Number(s)	Offset (Hex)	Assignment
0	000	Reset
1	004	Misaligned access
2	008	Access error
3	00C	Divide by zero
4	010	Illegal instruction
5	014	Privilege violation
6	018	Trace exception
7	01C	Breakpoint exception
8	020	Unrecoverable error
9	024	Soft reset
10	028	INT autovector
11	02C	FINT autovector
12	030	Hardware accelerator
13	034	(Reserved)
14	038	
15	03C	
16–19	040–04C	TRAP #0–3 instruction vectors
20–31	050–07C	(Reserved)
32–127	080–1FC	Assigned to vectored interrupt controller

Freescale Semiconductor, Inc.

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

Freescale Semiconductor, Inc.

All M·Core-based processors have a special-purpose 32-bit *vector base register* that the core uses to define the location of the vector table. The VBR is always cleared to zero on reset, but can be rewritten later to contain any address divisible by 1024. This means that after reset, the application program can relocate the vector table to any place on a 1024-byte boundary within the memory map.

The peripheral modules on the MMC2001 do not have the ability to provide a vector number with their interrupt requests. Since the peripherals can't generate their own vector numbers, the core must provide a vector for them, so this method is called *autovectoring* in the M·Core documentation. Even though there are 96 additional vectors available for servicing user-defined interrupts, the MMC2001 can only use the INT (normal) and FINT (fast) autovectors. This restriction is not a problem as long as the application has only one or two interrupt sources. Under these conditions, two request lines and a two-entry vector table are sufficient.

As soon as a third interrupt source appears, an intermediate *interrupt handler* has to be written to poll the interrupt pending registers to identify the active source(s), select the source having the highest priority, and then branch to the correct routine to service it. This is where the autovectoring approach becomes a less attractive option, because of the additional processing needed between the time the interrupt request occurs and the time it gets serviced.

The MMC2107 has improved this situation by allowing the peripheral modules generating interrupt requests to also provide the core with a vector number. The MMC2107 still supports autovectoring, but there is no longer any need to use this method because all interrupt sources can now have their own unique vector number.

Vectoring on the MMC2107 uses 32 separate priorities for normal interrupts, and the application can assign one of these priorities to any interrupt source. A complete vector table may be constructed based on these 32 priorities. With vectored interrupts, the core is now able to redirect the flow of execution to an ISR without the need for an intermediate handler to identify the interrupt's source or priority. This significantly reduces the overhead required before the application can begin processing an interrupt request.

It is also possible to assign any of the 32 priorities to be implemented as a fast interrupt type, which gives it precedence over all normal interrupts regardless of priority. Because fast interrupts use a different set of shadow registers to save the processor context, a single level of nesting can be implemented without saving the PC or PSR on the stack. If this nested, fast interrupt is also configured to use the alternate register set, then its ISR will benefit from the smallest interrupt latency times even if other normal ISRs are being processed concurrently.

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

V. The Vectored Interrupt Sequence on the MMC2107

There are nine hardware modules on the MMC2107 capable of generating 40 different user-defined interrupts, as shown in Table 2. The interrupt controller module[4] provides an interface to the processor core interrupt logic for these 40 interrupt sources.

When a module generates an interrupt, it supplies a unique signal to the interrupt controller. The interrupt controller uses this signal to identify the source, and then associates the source with a 5-bit priority level that the programmer has chosen for that source. The interrupt controller module then generates a request to the core consisting of a 7-bit vector number and an interrupt request signal. This vector number is constructed from the 5-bit priority level of the source and two additional bits to identify the type of exception that has occurred – internal, normal, or fast.

The core itself knows nothing of the actual interrupt source – all the core knows is that an exception has occurred, the type of exception, and its priority (0-31). That is all the core needs to know, because the vector number contains all the information necessary to calculate an offset into the vector table. The table entry at this position is the address of the ISR for the source having a priority corresponding to this position. Since the vector number is based on the priority of the interrupt source and not on the source itself, the vector table must be constructed according to priority, not according to source. Higher priority interrupts will therefore have the addresses of their service routines at higher positions in the vector table.

4. See MMC2107 Technical Data – Section 7, “Interrupt Controller Module” for an in-depth description of the interrupt controller module.

Freescale Semiconductor, Inc.

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

Table 2. MMC2107 Interrupt Source Assignments

Source	Module	Flag	Description
0	QADC	PF1	Queue 1 conversion pause
1		CF1	Queue 1 conversion complete
2		PF2	Queue 2 conversion pause
3		CF2	Queue 2 conversion complete
4	SPI	MODF	Mode fault
5		SPIF	Transfer complete
6	SCI1	TDRE	Transmit data register empty
7		TC	Transmit complete
8		RDRF	Receive data register full
9		OR	Receiver overrun
10		IDLE	Receiver line idle
11	SCI2	TDRE	Transmit data register empty
12		TC	Transmit complete
13		RDRF	Receive data register full
14		OR	Receiver overrun
15		IDLE	Receiver line idle
16	TIM1	C0F	Timer channel 0
17		C1F	Timer channel 1
18		C2F	Timer channel 2
19		C3F	Timer channel 3
20		TOF	Timer overflow
21		PAIF	Pulse accumulator input
22		PAOVF	Pulse accumulator overflow

Freescale Semiconductor, Inc.

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

Table 2. MMC2107 Interrupt Source Assignments

Source	Module	Flag	Description
23	TIM2	C0F	Timer channel 0
24		C1F	Timer channel 1
25		C2F	Timer channel 2
26		C3F	Timer channel 3
27		TOF	Timer overflow
28		PAIF	Pulse accumulator input
29		PAOVF	Pulse accumulator overflow
30	PIT1	PIF	PIT interrupt flag
31	PIT2	PIF	PIT interrupt flag
32	EPORT	EPF0	Edge port flag 0
33		EPF1	Edge port flag 1
34		EPF2	Edge port flag 2
35		EPF3	Edge port flag 3
36		EPF4	Edge port flag 4
37		EPF5	Edge port flag 5
38		EPF6	Edge port flag 6
39		EPF7	Edge port flag 7

The programmer decides which priorities are assigned to which interrupt sources, and this assignment is made in the Priority Level Select Registers (PLSR) in the interrupt controller module. The MMC2107 has 40 unique user-defined interrupt sources and 32 possible priorities to assign to them. Naturally, there are 40 PLSRs corresponding to the 40 different interrupt sources, and each PLSR can be programmed with a priority value from 0-31. The interrupt controller module uses this information to associate a priority with each incoming interrupt request, and it is this priority that the core uses to find the address of each interrupt service routine.

Freescale Semiconductor, Inc.

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

VI. An Example MMC2107 Interrupt Sequence

For example, consider an Edge Port[5] interrupt. The MMC2107 Edge Port module provides eight external pins that can be configured to sense an external pulse and generate an interrupt when that pulse occurs. This is useful for interfacing with keypads, limit switches, motion sensors, or any other device that can produce a voltage transition edge.

Table 2 shows that Edge Port Flag 5 (EPF5) is interrupt source number 37. If the programmer wanted to give that interrupt source a priority of 13, he would therefore program PLSR37 in the interrupt controller module with a value of 13.

The programmer then has to decide whether this should be a normal or fast interrupt type. This decision will determine whether interrupts from this source will use the normal or fast shadow registers to save the PC and PSR (see Figure 1), and also whether this source will be able to interrupt the execution of normal interrupt service routines already in progress.

Assume that the programmer wants this to be a normal interrupt, because it is not time-critical to this system. He would then set the 13th bit in the Normal Interrupt Enable Register (NIER). This defines a priority-13 interrupt to be a normal type.

Now, whenever Edge Port 5 (source 37) generates an interrupt request, the interrupt controller associates priority 13 with it, and sends a normal interrupt request signal and a vector number to the core.

The normal interrupt request tells the core to stop doing whatever it is doing, save the PC and PSR on the EPC and EPSR shadow registers, and then clear the Exception Enable bit in the PSR. This last step is taken because, if another normal interrupt request is recognized now, the core will be unable to save the PC and PSR again, and the processor context will be lost. With PSR(EE) cleared, any further normal exception requests will trigger an unrecoverable error system exception. If the programmer has written a proper handler for this situation, he'll receive a diagnostic report that a fatal error has occurred. The program will need to be modified to either avoid this error or take some corrective action if it happens again.

The core then sets the PSR(S) bit to enter Supervisor mode, and clears the PSR Trace Mode (TM) and Interrupt Enable (IE) bits to disable Trace Mode and any further normal interrupts. The vector table is located in a real, physical memory location, not on a virtual page, so the PSR(TC) bit is also cleared to prevent the address from being translated by an external memory management unit.

5. See MMC2107 Technical Data – Section 12, "Edge Port Module".

Freescale Semiconductor, Inc.

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

The core then extracts the priority information from the vector number. Based on that priority (13), it fetches an address from the 13th 4-byte entry in the vectored interrupt section of the vector table and copies that address into the program counter.

Table 1 shows that the vectored interrupt section begins at an offset of 32 words (128 bytes) from the start of the vector table. The start of the table is defined by the contents of the vector base register, so the core would read the VBR, add an offset of (32 + 13) * 4 to it, and use the contents of that memory location as the address of the EPF5 interrupt service routine. Hopefully the programmer has had the foresight to place a useful address in this vector table entry. If not, things are going to go very wrong, very quickly, because the processor will jump to whatever address it finds there and begin executing instructions at that address, whether or not they are valid.

With the Edge Port 5 ISR address loaded into the program counter, the core fetches, decodes, and executes the first instruction of the ISR. In other words, it begins servicing the normal interrupt. Note that the ability to recognize fast interrupts has not been affected by any of these operations. The PSR(FE) bit is still set, so the core can still recognize a fast interrupt request from some other source, and the PC and PSR will still be saved because the FPC and FPSR shadow registers are still available for that purpose.

In practice, the whole process can take as few as 5 clock cycles from the time the interrupt request is recognized to the time the ISR begins to execute. This latency will be longer if a multi-clock instruction is being executed when the interrupt request occurs. The longest multi-clock instruction is the signed divide, which can take an additional 37 clock cycles to complete. Setting the Interrupt Control bit in the PSR can reduce this extra latency to 4 additional clocks. If the PSR(IC) bit is set when the interrupt request is received, the core will abort the multi-clock instruction, recognize and service the interrupt, and then restart the original instruction afterwards. Like so many other options on this device, PSR(IC) provides the flexibility to tailor the behavior of the processor to suit the requirements of the system.

Freescale Semiconductor, Inc.

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

VII. Further Optimizations – The Alternate Register File

After the core fetches the address of the EPF5 ISR from the vector table and puts it into the program counter, it also uses that address to do one other very important thing – the least significant bit of the EPF5 ISR's address is copied into the Alternate File bit of the PSR. The state of the PSR(AF) bit is what enables the 16 alternate registers. These are the registers which should be used if it is necessary provide the minimum latency for the most time-critical interrupt service routine. Because the EPF5 interrupt isn't very important in this hypothetical application, there is no need for this extra level of optimization.

The PSR(AF) bit is cleared on reset and is seldom set in normal operations, so the alternate registers are usually left disabled when an interrupt is serviced. Remember that the M·Core architecture has a fixed-length, 16-bit instruction set. Since all instructions are two bytes long, their addresses in the executable section of the program code will always be even numbers. Since all the least significant bits in all the ISR addresses contained in the vector table are always zero, these bits would essentially be wasted space. M·Core's designers decided instead to use these bits as flags to selectively enable the alternate registers for each individual ISR.

A vector table is constructed in Flash or RAM by copying the addresses of all the interrupt service routines into it, either when the Flash is programmed, or during the RAM initialization sequence. Since these ISR addresses are always even by default, the only way a vector table entry can become odd is if the compiler/linker forces it to be odd by explicitly setting that entry's least significant bit before the table is written. This can't occur by accident, so the alternate file remains disabled unless intentionally enabled by the program.

The alternate file should be used only for the most time-critical interrupt service routines. When nesting is used, the alternate file should only be used with the fast interrupt type to provide the minimum latency. The Edge Port Flag 5 interrupt was not time-critical in this example, so there was no need to use the alternate register file.

Freescale Semiconductor, Inc.

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

VIII. Unintended Consequences

Whenever a C function is called, the compiler preserves the register context by saving onto the stack any non-volatile registers that the function uses. These will be popped off the stack after the function returns, restoring the context to its former state. Some of the function's arguments may also be passed on the stack, depending on how many there are.

From a C compiler's point of view, an interrupt service routine is similar to a function that is never called, takes no arguments and returns no values. The return statement is different, but the compiler still generates the code to save the register context before it uses any registers in the ISR. This is where a potential problem arises, one which needs to be avoided when activating the alternate registers for a fast interrupt service routine.

On the M·Core processor, certain registers are bound to a particular purpose because specific instructions use them.[6] In particular, the R0 register is used as a stack pointer, because the M·Core instruction set contains some special commands that depend on having R0 point to the next available stack location.

If the alternate register file is active, then the alternate register R0' will be used as a stack pointer instead of the normal R0. But R0' is not connected to R0 to allow it to constantly mirror the contents of the regular stack pointer. The two are not linked together in any way. Unless R0' is initialized in the startup sequence, it will probably not even contain the address of a valid memory location. Even if R0' is initialized at startup, there is no way to ensure that R0' still matches R0 when an ISR begins to execute. In short, if the alternate file is enabled, the stack cannot be used safely without first checking that R0' has the same contents as R0.

This is the crux of the problem. When a C compiler generates code to save the register context on the stack, this code forms the first executable statements of the ISR. This preamble code runs before the ISR has a chance to do anything else. By the time R0' can be validated, the damage is already done. This is why M·Core compilers need to provide the option of turning off the generation of preamble code to build a stack frame.

The whole point of using the alternate registers is to avoid having to use the stack to save the register context in the first place. The stack should not be used when the alternate register file is enabled.

6. See M·Core Applications Binary Interface Standards Manual, Section 2.2.1 "Register Assignments"

Freescale Semiconductor, Inc.

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

IX. Putting It All Together – A Sample Application

The Hardware Platform

Freescale provides two hardware development tools for the MMC2107, a mid-range EVB2107 evaluation board, and a more expensive CMB2107 Controller and Memory Board. On both the CMB/EVB2107 boards, there are four LEDs available for user control.[7] There are sample applications available that blink these LEDs, but this project will do something a little more involved – it will make the LEDs fade in and out by varying their apparent brightness.

If a light flashes rapidly enough, the human brain will sense a continuous light source that appears to become brighter or dimmer as the duty cycle changes. Varying the average power by changing the duty cycle is called *pulse-width modulation*, and the technique is often used to control the speed and torque of electrical motors. It is also a good way to save energy, because it allows the average current through a coil (or an LED, in this case) to be limited without using a resistance.

The MMC2107 Timer Module[8]

Many microprocessor designs have integrated modules for measuring time intervals and generating waveforms. There are two general-purpose, four-channel timers on the MMC2107. Although these timers can implement simple pulse-width modulation (PWM), they are not specially designed for this purpose. In particular, the MMC2107 timers are not capable of generating *buffered* PWM waveforms. This means that the duty cycle must not be changed too suddenly, or the waveform will be inconsistent and the LEDs will flicker.

Another design constraint of this specific application is that the LEDs are not electrically connected to the output channels of the timers on the CMB/EVB2107 boards. Instead of using the timer output pins to drive waveforms into the LEDs, the timers will generate interrupts. These interrupts will be used to turn the LEDs on and off. This is a more processor-intensive way to accomplish PWM, but it works on the CMB/EVB2107 without having to change the circuit.

7. See MMCCMB2107 Controller and Memory Board (CMB2107) User’s Manual
8. See MMC2107 Technical Data – Section 15, “Timer Modules (TIM1 and TIM2)”.

Freescale Semiconductor, Inc.

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

The MMC2107 PIT Module[9]

Once the LEDs are flashing with a given duty cycle, the duty cycle will be slowly changed to produce a fading effect. The MMC2107's *Programmable Interrupt Timer* (PIT) will be used for this purpose. The PIT is a countdown timer driven from a scalable input clock, and can be set to generate an interrupt when its counter underflows. There are two PITs on the MMC2107, and both will be used for this application. Each time a PIT interrupt occurs, a pointer will be advanced through a look-up table of duty cycle values. By choosing these values carefully, the fading effect will be produced.

Metrowerks CodeWarrior

The Metrowerks CodeWarrior for M·Core R2.5 integrated development environment (IDE) was used to develop this application. The R2.5 version of CodeWarrior provides the user with a choice between the traditional CodeWarrior R2.0 IDE (compiler / linker / debugger / project manager) and an IDE that uses a similar set of tools developed by Hiware. Since it is impossible to predict which CodeWarrior version will be in use by the reader, two separate projects are included with this applications note. The syntax is very similar (most of the source code is common to both versions) and there are only a few things that need to be changed.

The most significant difference is in the files that instruct the linker where in memory to locate the compiled objects. In the CodeWarrior environment, these instructions are contained in a *linker command file.* In the Hiware tool set, this is done in a *linker parameter file.* Although the format of these files is different, the purpose is the same.

When programming an application that is closely tied to the processor hardware, it is often necessary to issue commands directly to the compiler to configure the way it produces executable code. These commands are called *pragmas,* and their syntax is compiler-specific. Where pragmas have been used, a conditional directive instructs the compiler to choose between the two versions.

In both the Metrowerks and Hiware M·Core compilers, stack frame generation is enabled by default, and the option to turn it off is selected by a pragma. The pragma syntax is different for the two compilers, but they have the same effect.

9. See MMC2107 Technical Data – Section 14, "Programmable Interrupt Timer Modules (PIT1 and PIT2)".

Freescale Semiconductor, Inc.

The Hiware and CodeWarrior assemblers also use different syntax. Assembly language is required in cases where there is no appropriate C-language command to accomplish what is desired at a hardware level. An example of this is when the program must directly read or modify a hardware register on the MMC2107. The C programming language does not contain any statements to do this, so M·Core assembly language is used instead.

There are two ways to incorporate assembly language statements into a C language program – in-line, or as a separate module – and examples are provided for each method. Every compiler handles assembly language a little differently, so this is another case where two different versions of the code are necessary.

Once CodeWarrior is installed on the host computer, is should be possible to double-click on either of the project (*.mcp) files to launch each CodeWarrior project with its associated IDE.

Note that at the time of this writing, the Hiware version of the project does not support the MetroTRK target resident kernel. Because of this, some type of background debug mode interface (e.g. Freescale EBDI[10]) is necessary to run the Hiware version. This project was initially developed using an Abatron[11] BDI2000.

10. Enhanced Background Debug Interface, Freescale part number MMC14EBDI02

11. For information on the Abatron BDI2000, see http://www.abatron.ch

Freescale Semiconductor, Inc.

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

Freescale Semiconductor, Inc.

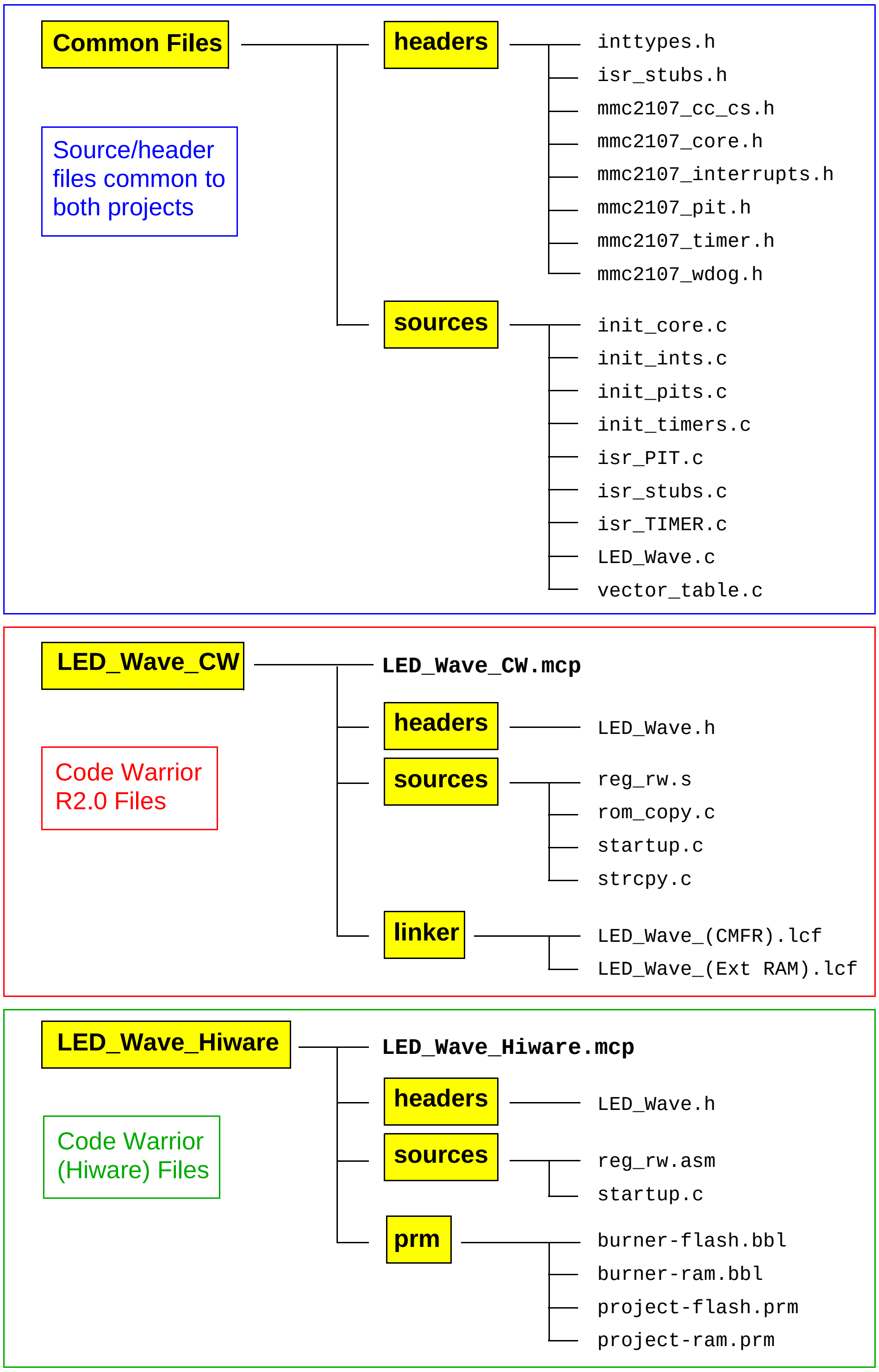

Figure 3. Project Directory Tree

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

Appendix A: Notes on the LED_Wave project

The Main Program

The LED_Wave main program is a simple interrupt-driven application, meaning that it stays in a low-power mode and waits for interrupts to occur.

File: **LED_Wave.c**

```
void main (void)
{
  WriteVBR(vectors);
  init_core();
  init_pits();
  init_ints();
  init_timers();

  EnableExsAllInts;

  for(;;) {
    Doze_Mode
  }
}
```

The main() function points the VBR to the vector table, makes some one-time calls to initialize the peripheral modules, enables interrupts, and goes to sleep. The rest of the application's working parts are in the initialization functions and the interrupt service routines.

"`EnableExsAllInts`" and "`Doze_Mode`" are substitution macros for inline assembly-language routines. Once the project is compiled, CodeWarrior will navigate to the definitions of these macro names by using a right-mouse click. The assembly code could have been written inline with the main program code, but this would have required two different versions because the assembly syntax is slightly different between the two compilers.

"`WriteVBR(vectors)`" is a function call to an assembler subroutine, one of a complete set of register read/write routines that are present in two files called ***reg_rw.s*** (CodeWarrior) and ***reg_rw.asm*** (Hiware).

Freescale Semiconductor, Inc.

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

Initializing the Vector Table

File: **vector_table.c**

A vector table is a block of memory containing pointers which are the physical addresses of interrupt service routines. From the C language's standpoint, ISRs are functions which take and return no arguments, therefore the vector table is defined that way:

```
void (*const vectors[128])(void) = { . . . }
```

It is an array of 128 constant pointers to functions taking no parameters and returning no parameters. These are defined as constant pointers because they will not change after the program is running. Vectors usually go into non-volatile Flash memory, so it is entirely accurate to tell the linker that they are constants.

Also in vector_table.c is a macro definition:

```
#define set_low_bit(addr) ((void(*const)(void)) ((uint8_t *) (addr) + 1))
```

"`set_low_bit`" is a macro used to convert a vector table entry into an odd number by setting its least significant bit. This is how the alternate registers are enabled for a given interrupt service routine. Since the vector table is an array of function pointers, it is not legal to "add a one" to them. The address first has to be typecast into a "pointer to unsigned char" before it can be incremented. (Note that ANSI C9x portable type definitions have been used throughout.) Once the address has been changed, it is typecast back into a function pointer.

Further down the vector table is an example of how this macro is used:

```
set_low_bit(&isr_TIM1C0F),
```

The linker stores where it has relocated the `isr_TIM1C0F` module, so the module's name is used to reference its address, and the macro sets the least significant bit. Now the service routine for the Timer 1, Channel 0 Flag interrupt will use the alternate registers.

Note that although only six positions in the table are used for the interrupt service routines, the remaining positions have been filled with the addresses of stub ISRs. This is for debugging, as can be seen from the definition of one of these routines:

Freescale Semiconductor, Inc.

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

File: **isr_stubs.c**

```
/*  Misaligned Access Exception Handler  */
#ifdef HIWARE
#pragma NO_RETURN
#pragma NO_FRAME
#else
#pragma naked on
#endif

void misaligned_access(void)
{
    for(;;) {BreakPoint}
}

#ifndef HIWARE
#pragma naked reset
#endif
```

This translates into just two lines of assembly code – a breakpoint instruction, and an unconditional branch back to that breakpoint. All of the conditional directives are there to tell the two different compilers not to waste memory space building a stack frame or executing a return statement. If the program gets here, it will be trapped. If memory space is at a premium, these stubs could be combined into a single "spurious interrupt" handler. It is easier when debugging to keep them as separate routines, because then the information about where the program is trapped appears in the debugging window.

Initializing the Core and Peripherals

File: **init_core.c**
File: **init_pits.c**
File: **init_ints.c**
File: **init_timers.c**

These initialization functions are all pretty straightforward, with only a few points worth mentioning. When the interrupt control module is initialized in the `init_ints()` function, notice that four of the six TIMER interrupts are of the fast interrupt type. This allows them to interrupt the other normal interrupt service routines. Also notice that even though these Priority 0..3 TIMER interrupts appear to be of lower priority than the rest, they are in fact higher priority because they are fast interrupts.

There is also a point of interest in the TIMER initialization function. Although there are only four LEDs to flash, six timer channels are actually used. The third channel of both timers is set up to perform a special task – the Timer Counter Reset Enable (TCRE) bits are set in the Timer Status and Control Registers, causing each Channel 3 output-compare event to reset its main timer counter. This allows the implementation of variable-frequency PWM with 16-bit resolution.

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

Freescale Semiconductor, Inc.

The Interrupt Service Routines

File: **isr_PIT.c**
File: **isr_TIMER.c**

These are the routines which handle the switching of the LEDs. The PIT ISR sets a global flag which signals the other ISRs to change the duty cycle of the LEDs.

The isr_TIMxC3F routines perform two functions. Most of the time, they turn on two of the four LEDs. Occasionally they have to respond to a flag which tells them to change the duty cycle, causing the apparent brightness of the LEDs to change. In this application, each routine clears this flag after it finished setting the new duty cycle, so two separate flags were required. If only one of the routines cleared the flag, only one flag and one PIT interrupt would be necessary.

The remaining four isr_TIMxCxF routines each toggle one of the four LEDs. These fast interrupt service routines also use the alternate registers, as can be seen from the compiler directives which surround the code. These ISRs can pre-empt the others, they do not use the stack, they store the PC and PSR in the fast shadow registers, and the compiler translates their return statement differently.[12]

```
void isr_TIM1C0F (void)
{
  LED_State ^= bit16;             /* toggle LED0 */
  *CMB2107_LED_addr = LED_State;
  TIM1FLG1_reg = 0x01;            /* clear TIMER1 channel 0 flag */
}
```

The Project Header File

File: **LED_Wave.h**

There are two versions of this file, one in the “CW” (CodeWarrior) subdirectory, and one in the “HW” (Hiware) subdirectory. They are identical except for one line:

```
#define HIWARE
```

in the Hiware version, and

```
/* #define HIWARE */
```

in the CodeWarrior version.

12. It is also possible to combine the first two statements of this ISR into one, and eliminate the LED_State variable, but some compilers produce unexpected results when you do this.

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

Freescale Semiconductor, Inc.

A number of things get either initialized or declared in this file. This initialization happens only once, when the file is #included from the main program. This selectivity is handled by the GLOBALS flag, which is defined only once, in the main program.

It is important to define NULL as a pointer to a type void, because NULL is used as a filler for the unused vector table entries. Since the vector table is an array of pointers, NULL needs to be a pointer as well. This is pretty standard, but some libraries redefine NULL to be zero.

```
#undef NULL
#define NULL ((void *) 0)
```

The lookup table for duty cycles is self-explanatory. Notice that 0% and 100% duty cycles are avoided (100% would be a value of 1024), and there are no sudden large changes, because the MMC2107 timers cannot produce true buffered PWM.

```
  const uint16_t oc_lookup[oc_tabsize] =  {
    5,    5,    5,    5,    5,
    5,    5,    5,    5,    5,
    5,    5,    5,    5,    5,
    5,    5,   10,   15,   20,
   30,   50,   90,  130,  170,
  210,  250,  290,  330,  370,
  410,  450,  490,  530,  570,
  610,  650,  690,  730,  770,
  810,  850,  890,  930,  970,
 1010, 1010, 1000,  990,  970,
  950,  930,  910,  880,  840,
  800,  760,  720,  680,  640,
  600,  560,  520,  480,  440,
  400,  360,  320,  280,  240,
  200,  160,  120,   80,   40,
   20,   15,   10,    5,    5
   };
```

There is one especially important part of LED_Wave.h which is used only by the Hiware linker, and it identifies the vector table as a block belonging to its own special memory section. The reason for this is explained in the next section on linker instructions.

```
#ifdef HIWARE
#pragma CONST_SECTION Exception_Table
#endif
```

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

Freescale Semiconductor, Inc.

Freescale Semiconductor, Inc.

Instructing the Linker

Instructions must be provided to the linker to specify where the executable code is to be located in memory. Because Hiware and CodeWarrior use different linkers, it is not surprising that they use different linker instructions. In the CodeWarrior version, these instructions are contained in a *linker command file.* Hiware refers to them as *linker parameter files,* but their purpose is the same.

File: **LED_Wave_(CMFR).lcf**
File: **LED_Wave_(Ext RAM).lcf**

These two linker command files contain statements relating to the Executable and Linking Format (ELF) output of the CodeWarrior linker.[13] Most of the statements in this file are explained in the documentation or in the comments in the linker command files themselves, but a few things should be explained because they are essential.

These two files differ by the value of one number on a single line – the first line of the MEMORY section that defines the location of the vector table. In the “CMFR” version, the linker is told to locate the vectors in the internal Flash array of the MMC2107. In the “Ext RAM” version, the vectors will reside in external RAM on the EVB2107 development board. Because the remaining memory sections are located relative to the vectors, the entire program image will reside either in Flash or external RAM depending on the value of that single address. This allows the selection of either internal or external program execution simply by choosing the build target containing the correct linker command file.

The vector table is a data block that is not accessed or referenced by any executable statement in the program. An optimizing linker will recognize the fact that this data is apparently unused and will omit this section in an effort to reduce code size, a process known as *dead-stripping.* The CodeWarrior linker’s FORCE_ACTIVE command is one way to inform the linker that the objects within the brackets must be left alone and not dead-stripped:

```
FORCE_ACTIVE {                 # prevent dead-stripping of vector table
  vectors
}
```

Since interrupt service routines are never actually called, they would also be at risk from dead-stripping were it not for the fact that they are referenced in the vector table. As long as the vector table is forced active, the ISRs are safe and will be included by the linker.

Notice also how the vector table data block is singled out from the rest of the program by specifying just the read-only data (.rodata) generated from the particular file containing the vector table:

13. A guide to the ELF Linker and Command Language is included in Chapter 9 of the document *CodeWarrior IDE – Targeting Freescale M·Core Embedded Systems*

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

```
.VECTOR_TABLE: {

  . = ALIGN(0x400);        # vector table must reside on a 1KB boundary
  vector_table.c (.rodata)

} > .vectors
```

This only works if the following CodeWarrior IDE option is selected:

- Mcore Base Project Settings
  - Target Settings Panel
    - Code Generation
      - Make Constant Data Read-Only

An initialized array like the vector table would ordinarily be placed in the ".data" section, but the vectors need to be read-only because they are constants. Selecting this option causes all constant data to be placed into the ".rodata" section, which the linker then locates at an address specified in the MEMORY segment at the beginning of the linker command file.

***File:* project-flash.prm**
***File:* project-ram.prm**

The Hiware linker instructions are somewhat less complex than CodeWarrior's linker command files because the Hiware SmartLinker doesn't need as much information. The `Exception_Table` section that was defined in the Hiware version of the **LED_Wave.h** header file is referenced here, and it is easy to see where the vectors are going to reside. The section labelled `CMB2107_Pseudo_ROM` mirrors the part of the internal Flash array on the MMC2107 that would have been used for constant data and program storage. The ability to locate the entire program image into RAM allows the designer to develop the application without repeatedly erasing and reprogramming the Flash array. When the application is tested and stable, it can be located in Flash by selecting the Flash build target.

Freescale Semiconductor, Inc.

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

```
NAMES
END

SECTIONS
  MMC2107_Vector_ROM = READ_ONLY  0x00000000 TO 0x000001FF;
  MMC2107_OnChip_ROM = READ_ONLY  0x00000200 TO 0x0001FFFF;
  MMC2107_OnChip_RAM = READ_WRITE 0x00801000 TO 0x00801FFF;

  CMB2107_Vector_ROM = READ_ONLY  0x80000000 TO 0x800001FF;
  CMB2107_Extern_ROM = READ_ONLY  0x80000200 TO 0x801FFFFF;

  CMB2107_Vector_RAM = READ_ONLY  0x81000000 TO 0x810001FF;
  CMB2107_Pseudo_ROM = READ_ONLY  0x81000200 TO 0x8101FFFF;
  CMB2107_Extern_RAM = READ_WRITE 0x81020000 TO 0x811FFFFF;
END

PLACEMENT
  Exception_Table INTO CMB2107_Vector_RAM;
  DEFAULT_ROM     INTO CMB2107_Pseudo_ROM;
  DEFAULT_RAM     INTO CMB2107_Extern_RAM;
END

STACKSIZE  0x400
```

Freescale Semiconductor, Inc.

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

Appendix B: Glossary

Autovector	There are only two autovectors defined in the M210 core architecture. These are the /INT and /FINT Autovectors, and are located at addresses `[VBR]+0x28` and `[VBR]+0x2C`, where `[VBR]` is the address contained by the Vector Base Register. If the AE bit in the Interrupt Control Register is set, these two autovectors contain the addresses of the two interrupt handlers that must recognize, identify, prioritize and act upon every interrupt request.
EBDI	Enhanced Background Debug Interface. A Freescale development tool that translates debugging commands from a host computer into machine commands that a target system can execute. The EBDI exercises the target MCU and performs debugging functions through the On-Chip Emulation (OnCE) connector or the Background Debug Mode (BDM) connector.
Fast Interrupt	A fast interrupt is one that has been initialized to use the fast interrupt circuitry of the M210 core. A fast interrupt is initialized when the bit is set in the Fast Interrupt Enable Register (FIER) corresponding to the priority of the interrupt source. For example, EPORT[6] is interrupt source 38. If PLSR38 is initialized to 5, then EPORT[6] will be a priority-5 normal interrupt. But if the FIE5 bit in the FIER is set, then all priority 5 interrupts will be processed as fast interrupts, and will have the ability to preempt the processing of all normal interrupt service routines.
Interrupt Request	An interrupt request is a voltage transition on an interrupt request line into the core. Before the core can recognize the interrupt request, it must synchronize the request with the processor clock. If the MMC2107 is in Stop mode, there is no clock available, so the part must wake up before processing the request. This is possible because the request propagates from the interrupt stimulus, through the interrupt controller, and into the M210 core through purely combinational logic. When a peripheral device asserts an interrupt request, this assertion must be recognized and acted upon before it can be serviced. An interrupt request is just that – a request. The request might be serviced immediately or not, depending on whether it is normal or fast, whether interrupts are enabled at the moment the assertion occurs, or whether interrupts of equivalent or lesser priority have been masked.

Freescale Semiconductor, Inc.

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

Interrupt Service Routine	An ISR is an executable code segment containing the program instructions that are associated with a particular interrupt source.
Mask	To disable all interrupt sources of a given priority or less. Masking is useful only if the program employs nesting of multiple interrupts of the same type (normal or fast), because fast interrupts on the M·Core are capable of nesting over normal interrupts.
Nesting	Nesting is the recognition of one interrupt request from within another interrupt's service routine. The MMC2107 supports single-level nesting automatically through the use of fast and normal interrupts, which have separate fast and normal shadow registers to save the program counter and processor status register. Nesting of multiple interrupts of the same type can be accomplished by saving the complete context within an ISR, masking interrupts of lesser or equal priority, and then re-enabling interrupts of that type by setting the PSR(EE) or PSR(FE) bit.
Normal Interrupt	A normal interrupt is one that has been initialized to use the normal interrupt circuitry of the M210 core. A normal interrupt is initialized when the bit is set in the Normal Interrupt Enable Register (NIER) corresponding to the priority of the interrupt source. For example, the PIT2 module is interrupt source 31. If PLSR31 is initialized to 3 and the NIER(NIE3) bit is set, then PIT2 interrupts will be processed as normal, priority level 3.
Priority	The value between 0-31 entered into the Priority Level Select Register (PLSR) for a given interrupt source. The position in the vector table of a given ISR's address entry. The higher in the vector table, the greater the priority. If more than one interrupt is pending at a given time, the one with greater priority will be serviced first.
Reset Vector	The reset vector is the first entry of the vector table. On reset, the program counter will be loaded with the value contained at this location, causing the contents of memory at an address equal to this value to be fetched and executed. This location may be in internal or external memory, depending on the boot configuration.

Freescale Semiconductor, Inc.

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

Vector	An individual entry in the vector table. There are 15 internal system exception vectors, and a choice of two autovectors or 64 user-defined interrupt vectors implemented in the MMC2107 architecture. (There is room in the table for 96 interrupt vectors, but MMC2107 only implements 64 of them.) The system exception vectors begin at the address contained by the Vector Base Register (VBR). The VBR is cleared on reset, so the reset vector must always reside at 0x0. This address can be in external or internal memory, depending on the choice of external or internal boot selection at reset. The 64 user-defined vectors assigned to the vectored interrupt controller occupy a 256-byte contiguous block of memory beginning at `[VBR]+128`. If the AE bit in the Interrupt Control Register is clear, the core will use these 64 vectors as a table from which it will fetch the addresses of any interrupt service routines. The address will be fetched from a location in the table corresponding to the priority of the interrupt source.
Vector Table	In M·Core processors, this is a contiguous 512-byte memory segment beginning at an address equal to the contents of the VBR. The vector table contains the addresses of all the system exception handlers and user-defined interrupt service routines. The vector table must reside on a 1024-byte boundary, because the VBR's lowest 10 bits are hard-wired to zero.

Freescale Semiconductor, Inc.

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

[redacted]

Freescale Semiconductor, Inc.

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

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.