Chapter 8. Device Drivers

- Defining device drivers
- Discussing the difference between architecture-specific and board-specific drivers
- Providing several examples of different types of device drivers

Most embedded hardware requires some type of software initialization and management. The software that directly interfaces with and controls this hardware is called a device driver. All embedded systems that require software have, at the very least, device driver software in their system software layer. Device drivers are the software libraries that initialize the hardware and manage access to the hardware by higher layers of software. Device drivers are the liaison between the hardware and the operating system, middleware, and application layers. (See Figure 8-1.)

The reader must always check the details about the particular hardware if the hardware component is not 100% identical to what is currently supported by the embedded system. Never assume existing device drivers in the embedded system will be compatible for a particular hardware part—even if the hardware is the same type of hardware that the embedded device currently supports! So, it is very important when trying to understand device driver libraries that:

- Different types of hardware will have different device driver requirements that need to be met.
- Even the same type of hardware, such as Flash memory, that are created by different manufacturers can require substantially different device driver software libraries to support within the embedded device.
The types of hardware components needing the support of device drivers vary from board to board, but they can be categorized according to the von Neumann model approach introduced in Chapter 3 (see Figure 8-2). The von Neumann model can be used as a software model as well as a hardware model in determining what device drivers are required within a particular platform. Specifically, this can include drivers for the master processor architecture-specific functionality, memory and memory management drivers, bus initialization and transaction drivers, and I/O (input/output) initialization and control drivers (such as for networking, graphics, input devices, storage devices, or debugging I/O) both at the board and master CPU level.

Device drivers are typically considered either architecture-specific or generic. A device driver that is architecture-specific manages the hardware that is integrated into the master processor (the architecture). Examples of architecture-specific drivers that initialize and enable components within a master processor include on-chip memory, integrated memory managers (memory management units (MMUs)), and floating-point hardware. A device driver that is generic manages hardware that is located on the board and not integrated onto the master processor. In a generic driver, there are typically architecture-specific portions of source code, because the master processor is the central control unit and to gain access to anything on the board usually means going through the master processor. However, the generic driver also manages board hardware that is not specific to that particular processor, which means that a generic driver can be configured to run on a variety of
architectures that contain the related board hardware for which the driver is written. Generic drivers include code that initializes and manages access to the remaining major components of the board, including board buses (I2C, PCI, PCMCIA, etc.), off-chip memory (controllers, level 2+ cache, Flash, etc.), and off-chip I/O (Ethernet, RS-232, display, mouse, etc.).

Figure 8-3a. MPC860 Hardware Block Diagram.[2]. © Freescale Semiconductor, Inc. Used by permission.

Figure 8-3b. MPC860 Architecture-Specific Device Driver System Stack. © Freescale Semiconductor, Inc. Used by permission.

Figure 8-3a shows a hardware block diagram of an MPC860-based board and Figure 8-3b shows a
systems diagram that includes examples of MPC860 processor-specific device drivers, as well as
generic device drivers.

Regardless of the type of device driver or the hardware it manages, all device drivers are generally
made up of all or some combination of the following functions:

- Hardware Startup: initialization of the hardware upon PowerON or reset.
- Hardware Shutdown: configuring hardware into its PowerOFF state.
- Hardware Disable: allowing other software to disable hardware on-the-fly.
- Hardware Enable: allowing other software to enable hardware on-the-fly.
- Hardware Acquire: allowing other software to gain singular (locking) access to hardware.
- Hardware Release: allowing other software to free (unlock) hardware.
- Hardware Read: allowing other software to read data from hardware.
- Hardware Write: allowing other software to write data to hardware.
- Hardware Install: allowing other software to install new hardware on-the-fly.
- Hardware Uninstall: allowing other software to remove installed hardware on-the-fly.
- Hardware Mapping: allowing for address mapping to and from hardware storage devices when
  reading, writing, and/or deleting data.
- Hardware Unmapping: allowing for unmapping (removing) blocks of data from hardware storage
devices.

Of course, device drivers may have additional functions, but some or all of the functions shown
above are what device drivers inherently have in common. These functions are based upon the
software’s implicit perception of hardware, which is that hardware is in one of three states at any
given time—inactive, busy, or finished. Hardware in the inactive state is interpreted as being either
disconnected (thus the need for an install function), without power (hence the need for an
initialization routine), or disabled (thus the need for an enable routine). The busy and finished states
are active hardware states, as opposed to inactive; thus the need for uninstall, shutdown, and/or
disable functionality. Hardware that is in a busy state is actively processing some type of data and is
not idle, and thus may require some type of release mechanism. Hardware that is in the finished
state is in an idle state, which then allows for acquisition, read, or write requests, for example.

Again, device drivers may have all or some of these functions, and can integrate some of these
functions into single larger functions. Each of these driver functions typically has code that
interfaces directly to the hardware and code that interfaces to higher layers of software. In some
cases, the distinction between these layers is clear, while in other drivers, the code is tightly
integrated (see Figure 8-4).

On a final note, depending on the master processor, different types of software can execute in
different modes, the most common being supervisory and user modes. These modes essentially differ
in terms of what system components the software is allowed access to, with software running in
supervisory mode having more access (privileges) than software running in user mode. Device driver
code typically runs in supervisory mode.
The next several sections provide real-world examples of device drivers that demonstrate how device driver functions can be written and how they can work. By studying these examples, the reader should be able to look at any board and figure out relatively quickly what possible device drivers need to be included in that system, by examining the hardware and going through a checklist, using the von Neumann model as a tool for keeping track of the types of hardware that might require device drivers. While not discussed in this chapter, later chapters will describe how device drivers are integrated into more complex software systems.

**8.1 Example 1: Device Drivers for Interrupt Handling**

As discussed previously, interrupts are signals triggered by some event during the execution of an instruction stream by the master processor. What this means is that interrupts can be initiated asynchronously, for external hardware devices, resets, power failures, etc., or synchronously, for instruction-related activities such as system calls or illegal instructions. These signals cause the master processor to stop executing the current instruction stream and start the process of handling (processing) the interrupt.

The software that handles interrupts on the master processor and manages interrupt hardware mechanisms (i.e., the interrupt controller) consists of the device drivers for interrupt handling. At least four of the 10 functions from the list of device driver functionality introduced at the start of this chapter are supported by interrupt-handling device drivers, including:

- **Interrupt Handling Startup**: initialization of the interrupt hardware (interrupt controller, activating interrupts, etc.) upon PowerON or reset.
- **Interrupt Handling Shutdown**: configuring interrupt hardware (interrupt controller, deactivating interrupts, etc.) into its PowerOFF state.
- **Interrupt Handling Disable**: allowing other software to disable active interrupts on-the-fly (not allowed for non-maskable interrupts (NMIs), which are interrupts that cannot be disabled).
• Interrupt Handling Enable: allowing other software to enable inactive interrupts on-the-fly.

Plus one additional function unique to interrupt handling:

• Interrupt Handler Servicing: the interrupt handling code itself, which is executed after the interruption of the main execution stream (this can range in complexity from a simple non-nested routine to nested and/or reentrant routines).

How startup, shutdown, disable, enable, and service functions are implemented in software usually depends on the following criteria:

• The types, number, and priority levels of interrupts available (determined by the interrupt hardware mechanisms on-chip and on-board).
• How interrupts are triggered.
• The interrupt policies of components within the system that trigger interrupts, and the services provided by the master CPU processing the interrupts.

Note: The material in the following paragraphs is similar to material found in Section 4.2.3 on interrupts.

The three main types of interrupts are software, internal hardware, and external hardware. Software interrupts are explicitly triggered internally by some instruction within the current instruction stream being executed by the master processor. Internal hardware interrupts, on the other hand, are initiated by an event that is a result of a problem with the current instruction stream that is being executed by the master processor because of the features (or limitations) of the hardware, such as illegal math operations (overflow, divide-by-zero), debugging (single-stepping, breakpoints), and invalid instructions (opcodes). Interrupts that are raised (requested) by some internal event to the master processor (basically, software and internal hardware interrupts) are also commonly referred to as exceptions or traps. Exceptions are internally generated hardware interrupts triggered by errors that are detected by the master processor during software execution, such as invalid data or a divide by zero. How exceptions are prioritized and processed is determined by the architecture. Traps are software interrupts specifically generated by the software, via an exception instruction. Finally, external hardware interrupts are interrupts initiated by hardware other than the master CPU (board buses, I/O, etc.).

Title-2

For interrupts that are raised by external events, the master processor is either wired via an input pin(s) called an IRQ (Interrupt Request Level) pin or port, to outside intermediary hardware (e.g., interrupt controllers), or directly to other components on the board with dedicated interrupt ports, that signal the master CPU when they want to raise the interrupt. These types of interrupts are triggered in one of two ways: level-triggered or edge-triggered. A level-triggered interrupt is initiated when its IRQ signal is at a certain level (i.e., HIGH or LOW; see Figure 8-5a). These interrupts are processed when the CPU finds a request for a level-triggered interrupt when sampling its IRQ line, such as at the end of processing each instruction.
Edge-triggered interrupts are triggered when a change occurs on the IRQ line (from LOW to HIGH/rising edge of signal or from HIGH to LOW/falling edge of signal; see Figure 8-5b). Once triggered, these interrupts latch into the CPU until processed.

Both types of interrupts have their strengths and drawbacks. With a level-triggered interrupt, as shown in the example in Figure 8-6a, if the request is being processed and has not been disabled before the next sampling period, the CPU will try to service the same interrupt again. On the flip side, if the level-triggered interrupt were triggered and then disabled before the CPU’s sample period, the CPU would never note its existence and would therefore never process it. Edge-triggered interrupts could have problems if they share the same IRQ line, if they were triggered in the same manner at about the same time (say before the CPU could process the first interrupt), resulting in the CPU being able to detect only one of the interrupts (see Figure 8-6b).

Because of these drawbacks, level-triggered interrupts are generally recommended for interrupts that share IRQ lines, whereas edge-triggered interrupts are typically recommended for interrupt signals that are very short or very long.
At the point an IRQ of a master processor receives a signal that an interrupt has been raised, the interrupt is processed by the interrupt-handling mechanisms within the system. These mechanisms are made up of a combination of both hardware and software components. In terms of hardware, an interrupt controller can be integrated onto a board, or within a processor, to mediate interrupt transactions in conjunction with software. Architectures that include an interrupt controller within their interrupt-handling schemes include the 268/386 (x86) architectures, which use two PICs (Intel’s Programmable Interrupt Controller); MIPS32, which relies on an external interrupt controller; and the MPC860 (shown in Figure 8-7a), which integrates two interrupt controllers, one in the CPM and one in its SIU. For systems with no interrupt controller, such as the Mitsubishi M37267M8 TV microcontroller shown in Figure 8-7b, the interrupt request lines are connected directly to the master processor, and interrupt transactions are controlled via software and some internal circuitry, such as registers and/or counters.

Title-3

Interrupt acknowledgment (IACK) is typically handled by the master processor when an external device triggers an interrupt. Because IACK cycles are a function of the local bus, the IACK function of the master CPU depends on interrupt policies of system buses, as well as the interrupt policies of components within the system that trigger the interrupts. With respect to the external device triggering an interrupt, the interrupt scheme depends on whether that device can provide an interrupt vector (a place in memory that holds the address of an interrupt’s ISR (Interrupt Service Routine), the software that the master CPU executes after the triggering of an interrupt). For devices that cannot provide an interrupt vector, referred to as non-vectored interrupts, master processors implement an auto-vectored interrupt scheme in which one ISR is shared by the non-vectored interrupts; determining which specific interrupt to handle, interrupt acknowledgment, etc., are all handled by the ISR software.

Figure 8-7a. Motorola/Freescale MPC860 Interrupt Controllers.[4] © Freescale Semiconductor, Inc. Used by permission.
An interrupt-vectored scheme is implemented to support peripherals that can provide an interrupt vector over a bus and where acknowledgment is automatic. An IACK-related register on the master CPU informs the device requesting the interrupt to stop requesting interrupt service, and provides what the master processor needs to process the correct interrupt (such as the interrupt number and vector number). Based upon the activation of an external interrupt pin, an interrupt controller’s interrupt select register, a device’s interrupt select register, or some combination of the above, the master processor can determine which ISR to execute. After the ISR completes, the master processor resets the interrupt status by adjusting the bits in the processor’s status register or an interrupt mask in the external interrupt controller. The interrupt request and acknowledgment mechanisms are determined by the device requesting the interrupt (since it determines which interrupt service to trigger), the master processor, and the system bus protocols.

Keep in mind that this is a general introduction to interrupt handling, covering some of the key features found in a variety of schemes. The overall interrupt-handling scheme can vary widely from architecture to architecture. For example, PowerPC architectures implement an auto-vectored scheme, with no interrupt vector base register. The 68000 architecture supports both auto-vectored and interrupt-vectored schemes, whereas MIPS32 architectures have no IACK cycle and so the interrupt handler handles the triggered interrupts.

8.1.1 Interrupt Priorities
Because there are potentially multiple components on an embedded board that may need to request interrupts, the scheme that manages all of the different types of interrupts is priority-based. This means that all available interrupts within a processor have an associated interrupt level, which is the priority of that interrupt within the system. Typically, interrupts starting at level “1” are the highest priority within the system and incrementally from there (2, 3, 4, etc.) the priorities of the associated interrupts decrease. Interrupts with higher levels have precedence over any instruction stream being executed by the master processor, meaning that not only do interrupts have precedence over the main program, but higher priority interrupts have priority over interrupts with lower priorities as well. When an interrupt is triggered, lower priority interrupts are typically masked, meaning they are not allowed to trigger when the system is handling a higher-priority interrupt. The interrupt with the highest priority is usually called an NMI.

How the components are prioritized depends on the IRQ line they are connected to, in the case of external devices, or what has been assigned by the processor design. It is the master processor’s
internal design that determines the number of external interrupts available and the interrupt levels supported within an embedded system. In Figure 8-8a, the MPC860 CPM, SIU, and PowerPC Core all work together to implement interrupts on the MPC823 processor. The CPM allows for internal interrupts (two SCCs, two SMCs, SPI, I2C, PIP, general-purpose timers, two IDMA5s, SDMA, RISC Timer) and 12 external pins of port C, and it drives the interrupt levels on the SIU. The SIU receives interrupts from eight external pins (IRQ0–7) and eight internal sources, for a total of 16 sources of interrupts, one of which can be the CPM, and drives the IREQ input to the Core. When the IREQ pin is asserted, external interrupt processing begins. The priority levels are shown in Figure 8-8b.

In another processor, such as the 68000 (shown in Figures 8-9a and b), there are eight levels of interrupts (0–7), where interrupts at level 7 have the highest priority. The 68000 interrupt table (see Figure 8-9b) contains 256 32-bit vectors.

The M37267M8 architecture (shown in Figure 8-10a) allows for interrupts to be caused by 16 events (13 internal, two external, and one software), whose priorities and usages are summarized in Figure 8-10b.

Several different priority schemes are implemented in the various architectures. These schemes commonly fall under one of three models: the equal single level, where the latest interrupt to be triggered gets the CPU; the static multilevel, where priorities are assigned by a priority encoder, and the interrupt with the highest priority gets the CPU; and the dynamic multilevel, where a priority encoder assigns priorities and the priorities are reassigned when a new interrupt is triggered.

Figure 8-8a. Motorola/Freescale MPC860 Interrupt pins and table.[4] © Freescale Semiconductor, Inc. Used by permission.
Figure 8-8b. Motorola/Freescale MPC860 Interrupt Levels.[4] © Freescale Semiconductor, Inc. Used by permission.
Figure 8-9a. Motorola/Freescale 68000 IRQs. There are 3 IRQ pins: IPL0, IPL1, and IPL2.
<table>
<thead>
<tr>
<th>Vector Number(s)</th>
<th>Vector Offset (Hex)</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>Reset Initial Interrupt Stack Pointer</td>
</tr>
<tr>
<td>1</td>
<td>004</td>
<td>Reset initial Program Counter</td>
</tr>
<tr>
<td>2</td>
<td>008</td>
<td>Access Fault</td>
</tr>
<tr>
<td>3</td>
<td>00C</td>
<td>Address Error</td>
</tr>
<tr>
<td>4</td>
<td>010</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>5</td>
<td>014</td>
<td>Integer Divide by Zero</td>
</tr>
<tr>
<td>6</td>
<td>018</td>
<td>CHK, CHK2 instruction</td>
</tr>
<tr>
<td>7</td>
<td>01C</td>
<td>FTRAPc, TRAPc, TRAV instructions</td>
</tr>
<tr>
<td>8</td>
<td>020</td>
<td>Privilege Violation</td>
</tr>
<tr>
<td>9</td>
<td>024</td>
<td>Trace</td>
</tr>
<tr>
<td>10</td>
<td>028</td>
<td>Line 1010 Emulator (Unimplemented A-Line Opcode)</td>
</tr>
<tr>
<td>11</td>
<td>02C</td>
<td>Line 1111 Emulator (Unimplemented F-line Opcode)</td>
</tr>
<tr>
<td>12</td>
<td>030</td>
<td>(Unassigned, Reserved)</td>
</tr>
<tr>
<td>13</td>
<td>034</td>
<td>Coprocessor Protocol Violation</td>
</tr>
<tr>
<td>14</td>
<td>038</td>
<td>Format Error</td>
</tr>
<tr>
<td>15</td>
<td>03C</td>
<td>Uninitialized Interrupt</td>
</tr>
<tr>
<td>16-23</td>
<td>040-050</td>
<td>(Unassigned, Reserved)</td>
</tr>
<tr>
<td>24</td>
<td>060</td>
<td>Spurious Interrupt</td>
</tr>
<tr>
<td>25</td>
<td>064</td>
<td>Level 1 Interrupt Autovector</td>
</tr>
<tr>
<td>26</td>
<td>068</td>
<td>Level 2 Interrupt Autovector</td>
</tr>
<tr>
<td>27</td>
<td>06C</td>
<td>Level 3 Interrupt Autovector</td>
</tr>
<tr>
<td>28</td>
<td>070</td>
<td>Level 4 Interrupt Autovector</td>
</tr>
<tr>
<td>29</td>
<td>074</td>
<td>Level 5 Interrupt Autovector</td>
</tr>
<tr>
<td>30</td>
<td>078</td>
<td>Level 6 Interrupt Autovector</td>
</tr>
<tr>
<td>31</td>
<td>07C</td>
<td>Level 7 Interrupt Autovector</td>
</tr>
<tr>
<td>32-47</td>
<td>080-08C</td>
<td>TRAP #0 D 15 Instructor Vectors</td>
</tr>
<tr>
<td>48</td>
<td>0C0</td>
<td>FP Branch or Set on Unordered Condition</td>
</tr>
<tr>
<td>49</td>
<td>0C4</td>
<td>FP Inexact Result</td>
</tr>
<tr>
<td>50</td>
<td>0C8</td>
<td>FP Divide by Zero</td>
</tr>
<tr>
<td>51</td>
<td>0CC</td>
<td>FP Underflow</td>
</tr>
<tr>
<td>52</td>
<td>0D0</td>
<td>FP Operand Error</td>
</tr>
<tr>
<td>53</td>
<td>0D4</td>
<td>FP Overflow</td>
</tr>
<tr>
<td>54</td>
<td>0D8</td>
<td>FP Signaling NAN</td>
</tr>
<tr>
<td>55</td>
<td>0DC</td>
<td>FP Unimplemented Data Type (Defined for MC68040)</td>
</tr>
<tr>
<td>56</td>
<td>0E0</td>
<td>MMU Configuration Error</td>
</tr>
<tr>
<td>57</td>
<td>0E4</td>
<td>MMU Illegal Operation Error</td>
</tr>
<tr>
<td>58</td>
<td>0E8</td>
<td>MMU Access Level Violation Error</td>
</tr>
<tr>
<td>59-63</td>
<td>0ECD-0FC</td>
<td>(Unassigned, Reserved)</td>
</tr>
<tr>
<td>64-255</td>
<td>100D3FC</td>
<td>User Defined Vectors (192)</td>
</tr>
</tbody>
</table>

Figure 8-9b. Motorola/Freescale 68000 IRQs Interrupt Table.[6]

P4₁/MXG can be used as external interrupt pin INT2.

P4₄ can be used as external interrupt pin INT1.

Figure 8-10a. Mitsubishi M37267M8 8-bit TV Microcontroller Interrupts.[5]
8.1.2 Context Switching
After the hardware mechanisms have determined which interrupt to handle and have acknowledged the interrupt, the current instruction stream is halted and a context switch is performed, a process in which the master processor switches from executing the current instruction stream to another set of instructions. This alternate set of instructions being executed as the result of an interrupt is the ISR or interrupt handler. An ISR is simply a fast, short program that is executed when an interrupt is triggered. The specific ISR executed for a particular interrupt depends on whether a non-vectored or vectored scheme is in place. In the case of a non-vectored interrupt, a memory location contains the start of an ISR that the PC (program counter) or some similar mechanism branches to for all non-vectored interrupts. The ISR code then determines the source of the interrupt and provides the appropriate processing. In a vectored scheme, typically an interrupt vector table contains the address of the ISR.

The steps involved in an interrupt context switch include stopping the current program’s execution of instructions, saving the context information (registers, the PC, or similar mechanism that indicates where the processor should jump back to after executing the ISR) onto a stack, either dedicated or shared with other system software, and perhaps the disabling of other interrupts. After the master processor finishes executing the ISR, it context switches back to the original instruction stream that had been interrupted, using the context information as a guide.

Title-5

The interrupt services provided by device driver code, based upon the mechanisms discussed above, include enabling/disabling interrupts through an interrupt control register on the master CPU or the disabling of the interrupt controller, connecting the ISRs to the interrupt table, providing interrupt levels and vector numbers to peripherals, providing address and control data to corresponding
registers, etc. Additional services implemented in interrupt access drivers include the locking/unlocking of interrupts, and the implementation of the actual ISRs. The pseudocode in the following example shows interrupt handling initialization and access drivers that act as the basis of interrupt services (in the CPM and SIU) on the MPC860.

8.1.3 Interrupt Device Driver Pseudocode Examples
The following pseudocode examples demonstrate the implementation of various interrupt handling routines on the MPC860, specifically startup, shutdown, disable, enable, and interrupt servicing functions in reference to this architecture. These examples show how interrupt handling can be implemented on a more complex architecture like the MPC860, and this in turn can be used as a guide to understand how to write interrupt-handling drivers on other processors that are as complex or less complex than this one.

Interrupt Handling Startup (Initialization) MPC860

Overview of initializing interrupts on MPC860 (in both CPM and SIU)
1. Initializing CPM Interrupts in MPC860 Example
   1.1. Setting Interrupt Priorities via CICR.
   1.2. Setting individual enable bit for interrupts via CIMR.
   1.3. Initializing SIU Interrupts via SIU Mask Register including setting the SIU bit associated with the level that the CPM uses to assert an interrupt.
   1.4. Set Master Enable bit for all CPM interrupts.

2. Initializing SIU Interrupts on MPC860 Example
   2.1. Initializing the SIEL Register to select the edge-triggered or level-triggered interrupt handling for external interrupts and whether processor can exit/wakeup from low power mode.
   2.2. If not done, initializing SIU Interrupts via SIU Mask Register including setting the SIU bit associated with the level that the CPM uses to assert an interrupt.

** Enabling all interrupts via MPC860 "mtspr" instruction next step—see Interrupt Handling Enable

**

```
// Initializing CPM for interrupts - four-step process
// ***** step 1 *****
// initializing the 24-bit CICR (see Figure 8-11), setting priorities and the interrupt
// levels. Interrupt Request Level, or IRL[0:2] allows a user to program the priority
// request level of the CPM interrupt with any number from level 0 (highest priority)
// through level 7 (lowest priority).
```
### Initializing the 32-bit CIMR (see Figure 8-12), CIMR bits correspond to CMP Interrupt Sources indicated in CIPR (see Figure 8-11c), by setting the bits associated with the desired interrupt sources in the CIMR register (each bit corresponds to a CPM interrupt source).

#### CICR – CPM Interrupt Configuration Register

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SCdP</td>
<td>SCdP</td>
<td>SCdP</td>
<td>SCdP</td>
<td>SCaP</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>IRL0 _ IRL2</td>
<td>HP0 _ HP4</td>
<td>IEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8-11a. CICR Register**
Figure 8-11b. SCC Priorities. [2]

Figure 8-11c. CIPR Register. [2]

Figure 8-12. CIMR Register. [2]
// ***** step 3 *****
// Initializing the SIU Interrupt Mask Register (see Figure 8-13) including setting the SIU
// bit associated with the level that the CPM uses to assert an interrupt.
Figure 8-13. SIMASK Register.[2]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRM6</td>
<td>LVM6</td>
<td>IRM1</td>
<td>LVM1</td>
<td>IRM2</td>
<td>LVM2</td>
<td>IRM3</td>
<td>LVM3</td>
<td>IRM4</td>
<td>LVM4</td>
<td>IRM5</td>
<td>LVM5</td>
<td>IRM6</td>
<td>LVM6</td>
<td>IRM7</td>
<td>LVM7</td>
</tr>
</tbody>
</table>

16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Reserved

Figure 8-14. SIEL Register.[2]

```c
int SIMASK.IRM0 = 0x80000000; // enable external interrupt input level 0
int SIMASK.LVM0 = 0x40000000; // enable internal interrupt input level 0
int SIMASK.IRM1 = 0x20000000; // enable external interrupt input level 1
int SIMASK.LVM1 = 0x10000000; // enable internal interrupt input level 1
int SIMASK.IRM2 = 0x08000000; // enable external interrupt input level 2
int SIMASK.LVM2 = 0x04000000; // enable internal interrupt input level 2
int SIMASK.IRM3 = 0x02000000; // enable external interrupt input level 3
int SIMASK.LVM3 = 0x01000000; // enable internal interrupt input level 3
int SIMASK.IRM4 = 0x00800000; // enable external interrupt input level 4
int SIMASK.LVM4 = 0x00400000; // enable internal interrupt input level 4
int SIMASK.IRM5 = 0x00200000; // enable external interrupt input level 5
int SIMASK.LVM5 = 0x00100000; // enable internal interrupt input level 5
int SIMASK.IRM6 = 0x00080000; // enable external interrupt input level 6
int SIMASK.LVM6 = 0x00040000; // enable internal interrupt input level 6
int SIMASK.IRM7 = 0x00020000; // enable external interrupt input level 7
int SIMASK.LVM7 = 0x00010000; // enable internal interrupt input level 7
int RESERVED6 = 0x0000FFFF; // unused bits 16-31
```

// ***** step 4 *****

```c
// IEN bit 24 of CICR register - Master enable for CPM interrupts
int CICR.IEN = 0x00000080; // interrupts enabled IEN = 1
```

// Initializing SIU for interrupts - two-step process

// ***** step 1 *****

// Initializing the SIEL Register (see Figure 8-14) to select the edge-triggered (set to 1
// for falling edge indicating interrupt request) or level-triggered (set to 0 for a 0 logic
// level indicating interrupt request) interrupt handling for external interrupts (bits
// 0, 2, 4, 6, 8, 10, 12, 14) and whether processor can exit/wakeup from low power mode
// (bits 1, 3, 5, 7, 9, 11, 13, 15). Set to 0 is NO, set to 1 is Yes

Figure 8-14. SIEL Register.[2]
Interrupt Handling Shutdown on MPC860

There essentially is no shutdown process for interrupt handling on the MPC860, other than perhaps disabling interrupts during the process.

Interrupt Handling Disable on MPC860

// Essentially disabling all interrupts via IEN bit 24 of CICR - Master disable for CPM
// interrupts
CICR.IEN="CICR.IEN" AND "0"; // interrupts disabled IEN = 0

Interrupt Handling Enable on MPC860

// To disable specific interrupt means modifying the SIMASK, so disabling the external
// interrupt at level 7 (IRQ7) for example is done by clearing bit 14
SIMASK.IRM7="SIMASK.IRM7" AND "0"; // disable external interrupt input level 7
// disabling of all interrupts takes effect with the mtspr instruction.
mtspr 82,0; // disable interrupts via mtspr (move to special purpose register)
// instruction
Interrupt Handling Servicing on MPC860

In general, this ISR (and most ISRs) essentially disables interrupts first, saves the context information, processes the interrupt, restores the context information, and then enables interrupts.

InterruptServiceRoutineExample ()
{
    // disable interrupts
disableInterrupts();  // mtspr 82.0;
    // save registers
    saveState();
    // read which interrupt from S1 Vector Register (SIVEC)
    interruptCode = SIVEC.IC;

    // if IRQ 7 then execute
    if (interruptCode == IRQ7) {
        // if an IRQx is edge-triggered, then clear the service bit in the S1 Pending Register
        // by putting a "1"
        SIPEND.IRX = SIPEND.IRX or "1";
    }
    // main process

    // endif IRQ7

    // restore registers
    restoreState();
    // re-enable interrupts
    enableInterrupts();  // mtspr 80.0;
}

8.1.4 Interrupt Handling and Performance
The performance of an embedded design is affected by the latencies (delays) involved with the interrupt-handling scheme. The interrupt latency is essentially the time from when an interrupt is triggered until its ISR starts executing. The master CPU, under normal circumstances, accounts for a lot of overhead for the time it takes to process the interrupt request and acknowledge the interrupt, obtaining an interrupt vector (in a vectored scheme), and context switching to the ISR. In the case when a lower-priority interrupt is triggered during the processing of a higher priority interrupt, or a higher priority interrupt is triggered during the processing of a lower priority interrupt, the interrupt latency for the original lower priority interrupt increases to include the time in which the higher priority interrupt is handled (essentially how long the lower priority interrupt is
disabled). Figure 8-15 summarizes the variables that impact interrupt latency.

![Figure 8-15. Interrupt Latency.](image)

Within the ISR itself, additional overhead is caused by the context information being stored at the start of the ISR and retrieved at the end of the ISR. The time to context switch back to the original instruction stream that the CPU was executing before the interrupt was triggered also adds to the overall interrupt execution time. While the hardware aspects of interrupt handling (the context switching, processing interrupt requests, etc.) are beyond the software’s control, the overhead related to when the context information is saved, as well as how the ISR is written both in terms of the programming language used and the size, are under the software’s control. Smaller ISRs, or ISRs written in a lower-level language like assembly, as opposed to larger ISRs or ISRs written in higher-level languages like Java, or saving/retrieving less context information at the start and end of an ISR, can all decrease the interrupt handling execution time and increase performance.

© 2013 Elsevier, Inc. All rights reserved. Printed with permission from Newnes, a division of Elsevier. Copyright 2013. For more information on this title and other similar books, please visit [www.newnespress.com](http://www.newnespress.com).

If you liked this and would like to see a weekly collection of related products and features delivered directly to your inbox, [click here to sign up for the EDN on Systems Design newsletter](http://www.newnespress.com).