Chapter 10: Exceptions and Interrupts

10.1 Introduction

Exceptions and interrupts are part of a mechanism provided by the majority of embedded processor architectures to allow for the disruption of the processor's normal execution path. This disruption can be triggered either intentionally by application software or by an error, unusual condition, or some unplanned external event.

Many real-time operating systems provide wrapper functions to handle exceptions and interrupts in order to shield the embedded systems programmer from the low-level details. This application-programming layer allows the programmer to focus on high-level exception processing rather than on the necessary, but tedious, prologue and epilogue system-level processing for that exception. This isolation, however, can create misunderstanding and become an obstacle when the programmer is transformed from an embedded *applications* programmer into an embedded *systems* programmer.

Understanding the inner workings of the processor exception facility aids the programmer in making better decisions about when to best use this powerful mechanism, as well as in designing software that handles exceptions correctly. The aim of this chapter is to arm the programmer with this knowledge.

This chapter focuses on:

- the definitions of exception and interrupt,
- the applications of exceptions and interrupts,
- a closer look at exceptions and interrupts in terms of hardware support, classifications, priorities, and causes of spurious interrupts, and
- a detailed discussion on how to handle exceptions and interrupts.

10.2 What are Exceptions and Interrupts?

An *exception* is any event that disrupts the normal execution of the processor and forces the processor into execution of special instructions in a privileged state. Exceptions can be classified into two categories: synchronous exceptions and asynchronous exceptions.

Exceptions raised by internal events, such as events generated by the execution of processor instructions, are called *synchronous exceptions.* Examples of synchronous exceptions include the following:

- On some processor architectures, the read and the write operations must start at an even memory address for certain data sizes. Read or write operations that begin at an odd memory address cause a memory access error event and raise an exception (called an *alignment exception*).
- An arithmetic operation that results in a division by zero raises an exception.

Exceptions raised by external events, which are events that do not relate to the execution of processor instructions, are called *asynchronous exception* s. In general, these external events are associated with hardware signals. The sources of these hardware signals are typically external hardware devices. Examples of asynchronous exceptions include the following:

- Pushing the reset button on the embedded board triggers an asynchronous exception (called the *system reset exception*).
- The communications processor module that has become an integral part of many embedded designs is another example of an external device that can raise asynchronous exceptions when it receives data packets.

An *interrupt*, sometimes called an *external interrupt*, is an asynchronous exception triggered by an event that an external hardware device generates. Interrupts are one class of exception. What differentiates interrupts from other types of exceptions, or more precisely what differentiates synchronous exceptions from asynchronous exceptions, is the source of the event. The event source for a synchronous exception is internally generated from the processor due to the execution of some instruction. On the other hand, the event source for an asynchronous exception is an external hardware device.

Because the term *interrupt* has been used extensively in other texts, therefore, the text that follows uses *exceptions* to mean *synchronous exceptions* and *interrupts* to mean *asynchronous exceptions.* The book uses *general exceptions* to mean both. The term *interrupts* and *external interrupts* are used interchangeably throughout the text.

Exceptions and interrupts are the necessary evils that exist in the majority of embedded systems. This facility, specific to the processor architecture, if misused, can become the source of troubled designs. While exceptions and interrupts introduce challenging design complications and impose strict coding requirements, they are nearly indispensable in embedded applications. The following sections describe the most common and important uses of these mechanisms.

10.3 Applications of Exceptions and Interrupts

From an application's perspective, exceptions and external interrupts provide a facility for embedded hardware (either internal or external to the processor) to gain the attention of application code. Interrupts are a means of communicating between the hardware and an application currently running on an embedded processor.

In general, exceptions and interrupts help the embedded engineer in three areas:

- internal errors and special conditions management,
- hardware concurrency, and
- service requests management.

10.3.1 Internal Errors and Special Conditions Management

Handling and appropriately recovering from a wide range of errors without coming to a halt is often necessary in the application areas in which embedded systems are typically employed.

Exceptions are either error conditions or special conditions that the processor detects while executing instructions. Error conditions can occur for a variety of reasons. The embedded system might be implementing an algorithm, for example, to calculate heat exchange or velocity for a cruise control. If some unanticipated condition occurs that causes a division by zero, over-flow, or other math error, the application must be warned. In this case, the execution of the task performing the calculation halts, and a special exception service routine begins. This process gives the application an opportunity to evaluate and appropriately handle the error. Other types of errors include memory read or write failures (a common symptom of a stray pointer), or attempts to access floating-point hardware when not installed.

Many processor architectures have two modes of execution: normal and privileged. Some instructions, called *privileged instructions*, are allowed to execute only when the processor is in the privileged execution mode. An exception is raised when a privileged instruction is issued while the processor is in normal execution mode.

Special conditions are exceptions that are generated by special instructions, such as the TRAP instruction on the Motorola 68K processor family. These instructions allow a program to force the processor to move into privileged execution mode, consequently gaining access to a privileged

instruction set. For example, the instruction used to disable external interrupts must be issued in privileged mode.

Another example of a special condition is the trace exception generated by the break point feature available on many processor architectures. The debugger agent, a special software program running on the embedded device, handles this exception, which makes using a host debugger to perform software break point and code stepping possible.

Although not all microcontrollers or embedded processors define the same types of exceptions or handle them in the same way, an exception facility is available and can assist the embedded systems engineer design a controlled response to these internal errors and special conditions.

10.3.2 Hardware Concurrency and Service Request Management

The ability to perform different types of work simultaneously is important in embedded systems. Many external hardware devices can perform device-specific operations in parallel to the core processor. These devices require minimum intervention from the core processor. The key to concurrency is knowing when the device has completed the work previously issued so that additional jobs can be given. External interrupts are used to achieve this goal.

For example, an embedded application running on a core processor issues work commands to a device. The embedded application continues execution, performing other functions while the device tries to complete the work issued. After the work is complete, the device triggers an external interrupt to the core processor, which indicates that the device is now ready to accept more commands. This method of hardware concurrency and use of external interrupts is common in embedded design.

Another use of external interrupts is to provide a communication mechanism to signal or alert an embedded processor that an external hardware device is requesting service. For example, an initialized programmable interval timer chip communicates with the embedded processor through an interrupt when a preprogrammed time interval has expired. (Chapter 11 discusses programmable interval timers in detail.) Similarly, the network interface device uses an interrupt to indicate the arrival of packets after the received packets have been stored into memory.

The capabilities of exceptions and their close cousins, external interrupts, empower embedded designs. Applying the general exception facility to an embedded design, however, requires properly handling general exceptions according to the source and associated cause of each particular general exception in question. The following section provides the needed background knowledge.

10.4 A Closer Look at Exceptions and Interrupts

General exceptions have classifications and are prioritized based on the classifications. It is possible there exists another level of priorities, imposed and enforced by the interrupt hardware, among the external interrupts. Understanding the hardware sources that can trigger general exceptions, the hardware that implements the transfer of control, and the mechanisms for determining where control vectors reside are all critical to properly installing general exception handlers and to writing correct general exception handlers.

10.4.1 Programmable Interrupt Controllers and External Interrupts

Most embedded designs have more than one source of external interrupts, and these multiple external interrupt sources are prioritized. To understand how this process is handled, a clear understanding of the concept of a *programmable interrupt controller* (PIC) is required.

The PIC is implementation-dependent. It can appear in a variety of forms and is sometimes given different names, however, all serve the same purpose and provide two main functionalities:

- Prioritizing multiple interrupt sources so that at any time the highest priority interrupt is presented to the core CPU for processing.
- Offloading the core CPU with the processing required to determine an interrupt's exact source.

The PIC has a set of interrupt request lines. An external source generates interrupts by asserting a physical signal on the interrupt request line. Each interrupt request line has a priority assigned to it. Figure 10.1 illustrates a PIC used in conjunction with four interrupt sources. Each interrupt source connects to one distinct interrupt request line: the airbag deployment sensor, the break deployment sensor, the fuel-level sensor detecting the amount of gasoline in the system, and a real-time clock.

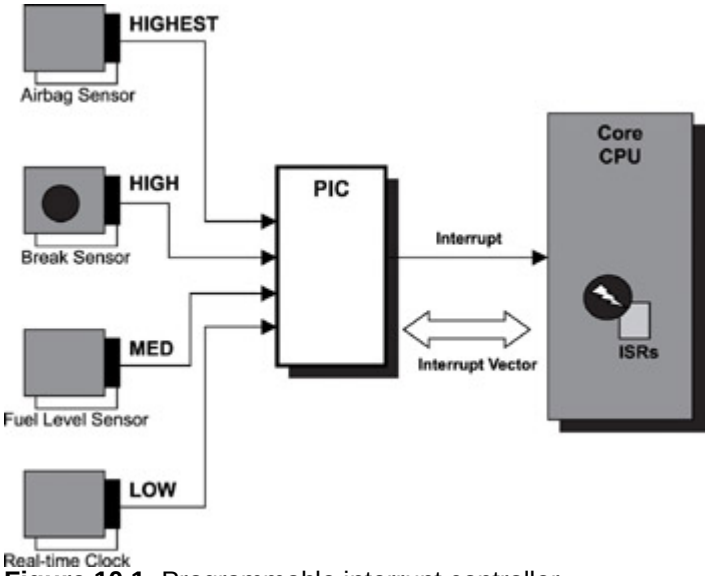


Figure 10.1: Programmable interrupt controller.

Figure 10.1 translates into an interrupt table that captures this information more concisely. The *interrupt table* lists all available interrupts in the embedded system. In addition, several other properties help define the dynamic characteristics of the interrupt source. Table 10.1 is an example of an interrupt table for the hypothetical example shown in Figure 10.1. The information in the table illustrates all of the sources of external interrupts that the embedded system must handle.

Why is it important to know this information? Understanding the priorities of the interrupt sources enables the embedded systems programmer to better understand the concept of *nested interrupts.* The term refers to the ability of a higher priority interrupt source to preempt the processing of a lower priority interrupt. It is easy to see how low-priority interrupt sources are affected by higher priority interrupts and their execution times and frequency if this interrupt table is ordered by overall system priority. This information aids the embedded systems programmer in designing and implementing better ISRs that allow for nested interrupts.

The maximum frequency column of the interrupt table specifies the process time constraint placed on all ISRs that have the smallest impact on the overall system.

Table 10.1: Interrupt table.

Source	Priority	Vector Address	IRQ	Max Freq.	Description
Airbag	Highest	14h	8	N/A	Deploys airbag

Table 10.1: Interrupt table.

Source	Priority	Vector Address	IRQ	Max Freq.	Description
Sensor					
Break Sensor	High	18h	7	N/A	Deploys the breaking system
Fuel Level Sensor	Med	1Bh	6	20Hz	Detects the level of gasoline
Real-Time Clock	Low	1Dh	5	100Hz	Clock runs at 10ms ticks

The vector address column specifies where in memory the ISR must be installed. The processor automatically fetches the instruction from one of these known addresses based on the interrupt number, which is specified in the IRQ column. This instruction begins the interrupt-specific service routine. In this example, the interrupt table contains a vector address column, but these values are dependent on processor and hardware design. In some designs, a column of indexes is applied to a formula used to calculate an actual vector address. In other designs, the processor uses a more complex formulation to obtain a vector address before fetching the instructions. Consult the hardware manual for specific details. Later sections of this chapter discuss the interrupt service routine in detail. In general, the vector table also covers the service routines for synchronous exceptions. The service routines are also called *vectors* in short.

10.4.2 Classification of General Exceptions

Although not all embedded processors implement exceptions in the same manner, most of the more recent processors have these types of exceptions:

- asynchronous-non-maskable,
- asynchronous-maskable,
- synchronous-precise, and
- synchronous-imprecise.

Asynchronous exceptions are classified into maskable and non-maskable exceptions. External interrupts are asynchronous exceptions. Asynchronous exceptions that can be blocked or enabled by software are called *maskable exceptions.* Similarly, asynchronous exceptions that cannot be blocked by software are called *non-maskable exceptions.* Non-maskable exceptions are always acknowledged by the processor and processed immediately. Hardware-reset exceptions are always non-maskable exceptions. Many embedded processors have a dedicated non-maskable interrupt (NMI) request line. Any device connected to the NMI request line is allowed to generate an NMI.

External interrupts, with the exception of NMIs, are the only asynchronous exceptions that can be disabled by software.

Synchronous exceptions can be classified into precise and imprecise exceptions. With *precise exception* s, the processor's program counter points to the exact instruction that caused the exception, which is the *offending instruction*, and the processor knows where to resume execution upon return from the exception. With modern architectures that incorporate instruction and data pipelining, exceptions are raised to the processor in the order of written instruction, not in the order of execution. In particular, the architecture ensures that the instructions that follow the offending instruction and that were started in the instruction pipeline during the exception do not affect the CPU state. This chapter is concerned with precise exceptions.

Silicon vendors employ a number of advanced techniques (such as predictive instruction and data loading, instruction and data pipelining, and caching mechanisms) to streamline overall execution in order to increase chip performance. For example, the processor can do floating point and integer memory operations out of order with the non-sequential memory access mode. If an embedded processor implements heavy pipelining or pre-fetch algorithms, it can often be impossible to determine the exact instruction and associated data that caused an exception. This issue indicates an *imprecise exception*. Consequently, when some exceptions do occur, the reported program counter does not point to the offending instruction, which makes the program counter meaningless to the exception handler.

Why is it important to know this information? Knowing the type of exception for which an exception handler is written helps the programmer determine how the system is to recover from the exception, if the exception is at all recoverable.

10.4.3 General Exception Priorities

All processors handle exceptions in a defined order. Although not every silicon vendor uses the exact same order of exception processing, generally exceptions are handled according to these priorities, as shown in Table 10.2.

Table 10.2: Exception priorities.

Highest	Asynchronous	Non-maskable
	Synchronous	Precise
	Synchronous	Imprecise
Lowest	Asynchronous	Maskable

The highest priority level of exceptions is usually reserved for system resets, other significant events, or errors that warrant the overall system to reset. In many cases, hardware implementations for this exception also cause much, if not all, of the surrounding hardware to reset to a known state and condition. For this reason, this exception is treated as the highest level.

The next two priority levels reflect a set of errors and special execution conditions internal to the processor. A synchronous exception is generated and acknowledged only at certain states of the internal processor cycle. The sources of these errors are rooted in either the instructions or data that is passed along to be processed.

Typically, the lowest priority is an asynchronous exception external to the core processor. External interrupts (except NMIs) are the only exceptions that can be disabled by software.

From an application point of view, all exceptions have processing priority over operating system objects, including tasks, queues, and semaphores. Figure 10.2 illustrates a general priority framework observed in most embedded computing architectures.

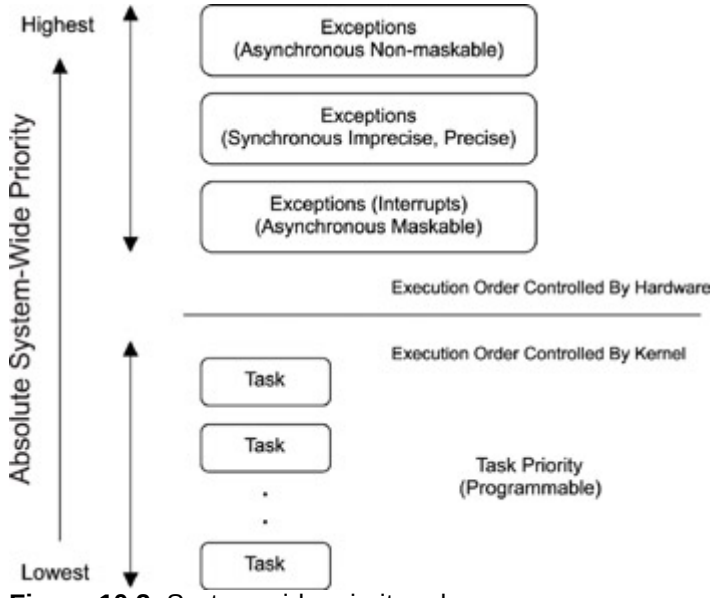


Figure 10.2: System-wide priority scheme.

10.5 Processing General Exceptions

Having introduced the fundamentals of exceptions and external interrupts, it is time to discuss processing exceptions and external interrupts. The overall exception handling mechanism is similar to the mechanism for interrupt handling. In a simplified view, the processor takes the following steps when an exception or an external interrupt is raised:

- Save the current processor state information.
- Load the exception or interrupt handling function into the program counter.
- Transfer control to the handler function and begin execution.
- Restore the processor state information after the handler function completes.
- Return from the exception or interrupt and resume previous execution.

A typical handler function does the following:

- Switch to an exception frame or an interrupt stack.
- Save additional processor state information.
- Mask the current interrupt level but allow higher priority interrupts to occur.
- Perform a minimum amount of work so that a dedicated task can complete the main processing.

10.5.1 Installing Exception Handlers

Exception service routines (ESRs) and interrupt service routines (ISRs) must be installed into the system before exceptions and interrupts can be handled. The installation of an ESR or ISR requires knowledge of the exception and interrupt table (called the *general exception table*).

The general exception table, as exemplified in Table 10.1, has a vector address column, which is sometimes also called the *vector table*. Each vector address points to the beginning of an ESR or ISR. Installing an ESR or ISR requires replacing the appropriate vector table entry with the address of the desired ESR or ISR.

The embedded system startup code typically installs the ESRs at the time of system initialization. Hardware device drivers typically install the appropriate ISRs at the time of driver initialization.

If either an exception or an interrupt occurs when no associated handler function is installed, the system suffers a system fault and may halt. To prevent this problem, it is common for an embedded RTOS to install default handler functions (i.e., functions that perform small amounts of work to ensure the proper reception of and the proper return from exceptions) into the vector table for every possible exception and interrupt in the system. Many RTOSes provide a mechanism that the embedded systems programmer can use to overwrite the default handler function with his or her own or to allow the programmer to insert further processing in addition to the default actions. If allowed, the embedded systems programmer can code specific actions before and after the default action is completed.

In this book, the general term *service routine* means either an ESR or an ISR when the distinction is not important.

10.5.2 Saving Processor States

When an exception or interrupt comes into context and before invoking the service routine, the processor must perform a set of operations to ensure a proper return of program execution after the service routine is complete. Just as tasks save information in task control blocks, exception and interrupt service routines also need to store blocks of information, called *processor state information*, somewhere in memory. The processor typically saves a minimum amount of its state information, including the status register (SR) that contains the current processor execution status bits and the program counter (PC) that contains the returning address, which is the instruction to resume execution after the exception. The ESR or the ISR, however, must do more to preserve more complete state information in order to properly resume the program execution that the exception preempted. A later section discusses this issue in more detail.

So, whose stack is used during the exception and interrupt processing?

Stacks are used for the storage requirement of saving processor state information. In an embedded operating system environment, a *stack* is a statically reserved block of memory and an active dynamic pointer called a *stack pointer,* as shown in Figure 10.3. In some embedded architectures, such as Motorola's 68000 microprocessors, two separate stacks-the *user stack* (USP) and the *supervisor stack* (SSP)-are used. The USP is used when the processor executes in non-privileged mode. The SSP is used when the processor executes in privileged mode.

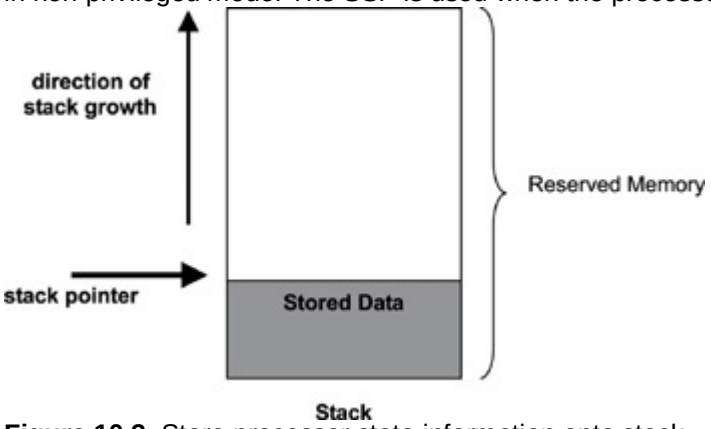


Figure 10.3: Store processor state information onto stack.

Section 10.3.1, 'Internal Errors and Special Conditions Management' on page 145, discusses processor execution modes. On this type of architecture, the processor consciously selects SSP to store its state information during general exception handling. While some architectures offer special support for stack switching, the balance of this chapter assumes a simple environment with just one run-time stack.

As data is saved on the stack, the stack pointer is incremented to reflect the number of bytes copied onto the stack. This process is often called *pushing values on the stack*. When values are copied off the stack, the stack pointer is decremented by the equivalent number of bytes copied from the stack. This process is called *popping values off the stack*. The stack pointer always points to the first valid location in order to store data onto the stack. For purposes of this book, the stack grows up; however, a stack can grow in the opposite direction. Note that a typical stack does not store identifiers for the contents. Stack users are required to push and pop items onto and off the stack in a symmetric order. If this rule is not followed during exception or interrupt processing, unintended results are likely to occur.

As Chapter 5 discusses, in an embedded operating system environment, all task objects have a task control block (TCB). During task creation, a block of memory is reserved as a stack for task use, as shown in Figure 10.4. High-level programming languages, such as C and C++, typically use the stack space as the primary vehicle to pass variables between functions and objects of the language.

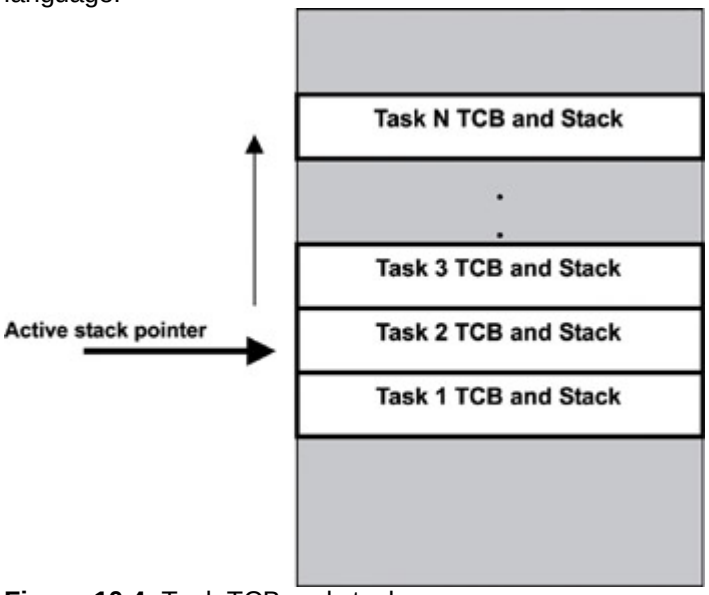


Figure 10.4: Task TCB and stack.

The active stack pointer (SP) is reinitialized to that of the active task each time a task context switch occurs. The underlying real-time kernel performs this work. As mentioned earlier, the processor uses whichever stack the SP points to for storing its minimum state information before invoking the exception handler.

Although not all embedded architectures implement exception or interrupt processing in the same way, the general idea of sizing and reserving exception stack space is the same. In many cases, when general exceptions occur and a task is running, the task's stack is used to handle the exception or interrupt. If a lower priority ESR or ISR is running at the time of exception or interrupt, whichever stack the ESR or ISR is using is also the stack used to handle the new exception or interrupt. This default approach on stack usage can be problematic with nested exceptions or interrupts, which are discussed in detail shortly.

10.5.3 Loading and Invoking Exception Handlers

As discussed earlier, some differences exist between an ESR and an ISR in the precursory work the processor performs. This issue is caused by the fact that an external interrupt is the only exception type that can be disabled by software. In many embedded processor architectures, external interrupts can be disabled or enabled through a processor control register. This control register directly controls the operation of the PIC and determines which interrupts the PIC raises

to the processor. In these architectures, all external interrupts are raised to the PIC. The PIC filters interrupts according to the setting of the control register and determines the necessary action. This book assumes this architecture model in the following discussions.

Formally speaking, an interrupt can be disabled, active, or pending. A *disabled interrupt* is also called a *masked interrupt*. The PIC ignores a disabled interrupt. A *pending interrupt* is an unacknowledged interrupt, which occurs when the processor is currently processing a higher priority interrupt. The pending interrupt is acknowledged and processed after all higher priority interrupts that were pending have been processed. An *active interrupt* is the one that the processor is acknowledging and processing. Being aware of the existence of a pending interrupt and raising this interrupt to the processor at the appropriate time is accomplished through hardware and is outside the concern of an embedded systems developer.

For synchronous exceptions, the processor first determines which exception has occurred and then calculates the correct index into the vector table to retrieve the ESR. This calculation is dependent on implementation. When an asynchronous exception occurs, an extra step is involved. The PIC must determine if the interrupt has been disabled (or masked). If so, the PIC ignores the interrupt and the processor execution state is not affected. If the interrupt is not masked, the PIC raises the interrupt to the processor and the processor calculates the interrupt vector address and then loads the exception vector for execution, as shown in Figure 10.5.

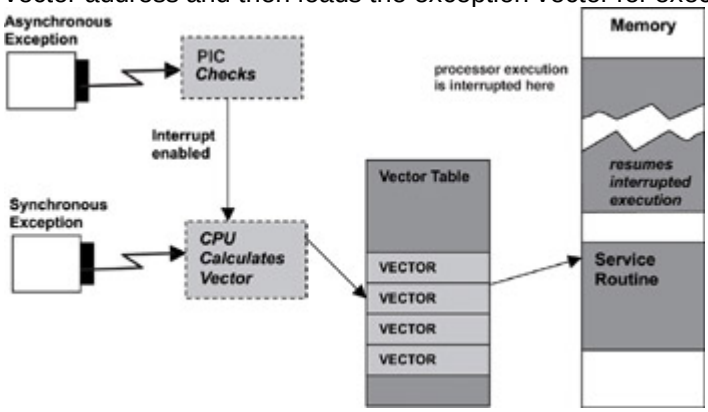


Figure 10.5: Loading exception vector.

Some silicon vendors implement the table lookup in hardware, while others rely on software approaches. Regardless, the mechanisms are the same. When an exception occurs, a value or index is calculated for the table. The content of the table at this index or offset reflects the address of a service routine. The program counter is initialized with this vector address, and execution begins at this location. Before examining the general approach to an exception handler, let's first examine nested interrupts and their effect on the stack.

10.5.4 Nested Exceptions and Stack Overflow

Nested exceptions refer to the ability for higher priority exceptions to preempt the processing of lower priority exceptions. Much like a context switch for tasks when a higher priority one becomes ready, the lower priority exception is preempted, which allows the higher priority ESR to execute. When the higher priority service routine is complete, the earlier running service routine returns to execution. Figure 10.6 illustrates this process.

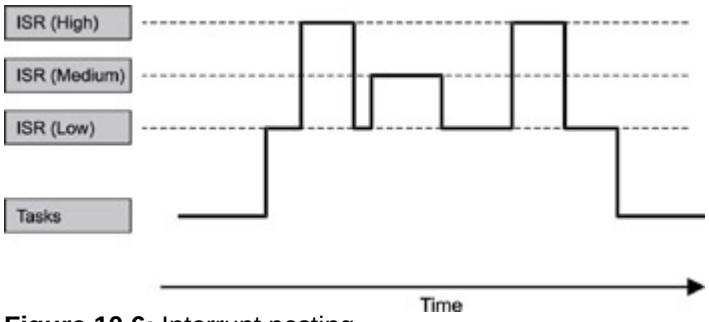


Figure 10.6: Interrupt nesting.

The task block in the diagram in this example shows a group of tasks executing. A low-priority interrupt then becomes active, and the associated service routine comes into context. While this service routine is running, a high-priority interrupt becomes active, and the lower priority service routine is preempted. The high-priority service routine runs to completion, and control returns to the low-priority service routine. Before the low-priority service routine completes, another interrupt becomes active. As before, the low-priority service routine is preempted to allow the medium-priority service routine to complete. Again, before the low-priority routine can finish, another high-priority interrupt becomes active and runs to completion. The low-priority service routine is finally able to run to completion. At that point, the previously running task can resume execution.

When interrupts can nest, the application stack must be large enough to accommodate the maximum requirements for the application's own nested function invocation, as well as the maximum exception or interrupt nesting possible, if the application executes with interrupts enabled. This issue is exactly where the effects of interrupt nesting on the application stack are most commonly observed.

As exemplified in Figure 10.4, *N* tasks have been created, each with its own TCB and statically allocated stack. Assuming the stack of the executing task is used for exceptions, a sample scenario, as shown in Figure 10.7, might look as follows:

1. Task 2 is currently running.
2. A low-priority interrupt is received.
3. Task 2 is preempted while exception processing starts for a low-priority interrupt.
4. The stack grows to handle exception processing storage needs.
5. A medium-priority interrupt is received before exception processing is complete.
6. The stack grows again to handle medium-priority interrupt processing storage requirements.
7. A high-priority interrupt is received before execution processing of the medium interrupt is complete.
8. The stack grows to handle high-priority interrupt processing storage needs.

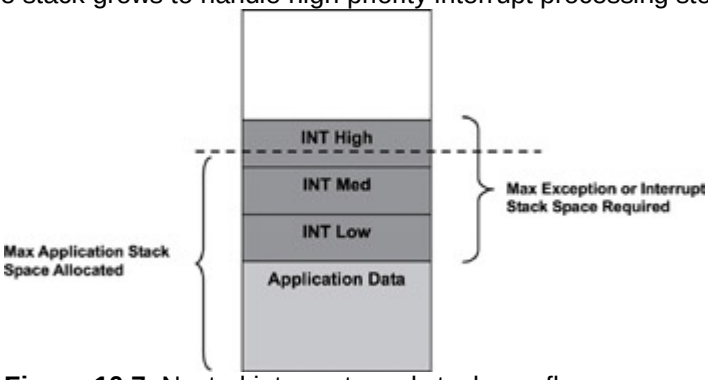


Figure 10.7: Nested interrupts and stack overflow.

In each case of exception processing, the size of the stack grows as has been discussed. Note that without a MMU, no bounds checking is performed when using a stack as a storage medium. As depicted in this example, the sum of the application stack space requirement and the exception stack space requirement is less than the actual stack space allocated by Task 2. Consequently, when data is copied onto the stack past the statically defined limits in this example, Task 3's TCB is corrupted, which is a *stack overflow*. Unfortunately, the corrupted TCB is not likely to be noticed until Task 3 is scheduled to run. These types of errors can be very hard to detect. They are a function of the combination of the running task and the exact frequency, timing, and sequence of interrupts or exceptions presented to the operating environment. This situation often gives a user or testing team the sense of a sporadic or flaky system. Sometimes, dependably recreating errors is almost impossible.

Two solutions to the problem are available: increasing the application's stack size to accommodate all possibilities and the deepest levels of exception and interrupt nesting, or having the ESR or ISR switch to its own exception stack, called an *exception frame*.

The maximum exception stack size is a direct function of the number of exceptions, the number of external devices connected to each distinct IRQ line, and the priority levels supported by the PIC. The simple solution is having the application allocate a large enough stack space to accommodate the worst case, which is if the lowest priority exception handler executes and is preempted by all higher priority exceptions or interrupts. A better approach, however, is using an independent exception frame inside the ESR or the ISR. This approach requires far less total memory than increasing every task stack by the necessary amount.

10.5.5 Exception Handlers

After control is transferred to the exception handler, the ESR or the ISR performs the actual work of exception processing. Usually the exception handler has two parts. The first part executes in the exception or interrupt context. The second half executes in a task context.

Exception Frames

The *exception frame* is also called the *interrupt stack* in the context of asynchronous exceptions.

Two main reasons exist for needing an exception frame. One reason is to handle nested exceptions. The other reason is that, as embedded architecture becomes more complex, the ESR or ISR consequently increases in complexity. Commonly, exception handlers are written in both machine assembly language and in a high-level programming language, such as C or C++. As mentioned earlier, the portion of the ESR or ISR written in C or C++ requires a stack to which to pass function parameters during invocation. This fact is also true if the ESR or ISR were to invoke a library function written in a high-level language.

The common approach to the exception frame is for the ESR or the ISR to allocate a block of memory, either statically or dynamically, before installing itself into the system. The exception handler then saves the current stack pointer into temporary memory storage, reinitializes the stack pointer to this private stack, and begins processing. This is depicted in Figure 10.8.

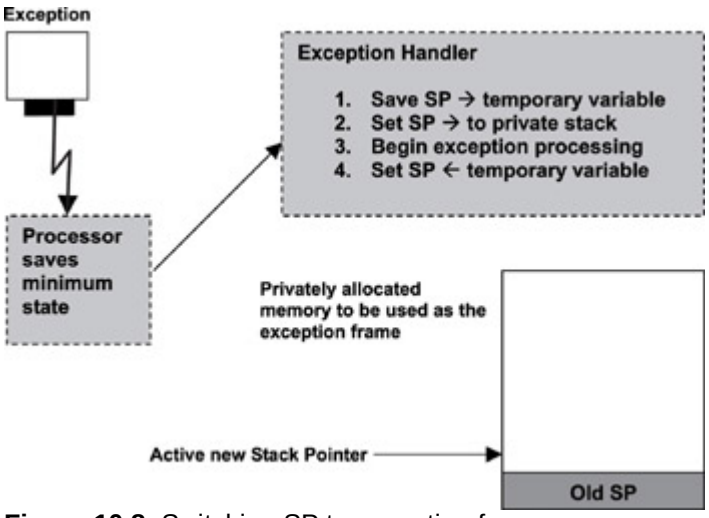


Figure 10.8: Switching SP to exception frame.

The exception handler can perform more housekeeping work, such as storing additional processor state information, onto this stack.

Differences between ESR and ISR

One difference between an ESR and an ISR is in the additional processor state information saved.

The three ways of masking interrupts are:

- Disable the device so that it cannot assert additional interrupts. Interrupts at all levels can still occur.
- Mask the interrupts of equal or lower priority levels, while allowing higher priority interrupts to occur. The device can continue to generate interrupts, but the processor ignores them.
- Disable the global system-wide interrupt request line to the processor (the line between the PIC and the core processor), as exemplified in Figure 10.1. Interrupts of any priority level do not reach the processor. This step is equivalent to masking interrupts of the highest priority level.

An ISR would typically deploy one of these three methods to disable interrupts for one or all of these reasons:

- the ISR tries to reduce the total number of interrupts raised by the device,
- the ISR is non-reentrant, and
- the ISR needs to perform some atomic operations.

Some processor architectures keep the information on which interrupts or interrupt levels are disabled inside the system status register. Other processor architectures use an interrupt mask register (IMR). Therefore, an ISR needs to save the current IMR onto the stack and disable interrupts according to its own requirements by setting new mask values into the IMR. The IMR only applies to maskable asynchronous exceptions and, therefore, is not saved by synchronous exception routines.

One other related difference between an ESR and an ISR is that an exception handler in many cases cannot prevent other exceptions from occurring, while an ISR can prevent interrupts of the same or lower priority from occurring.

Exception Timing

Discussions about the ESR or ISR, however, often mention keeping the ESR or ISR short. How so and how short should it be? To answer this question, let's focus the discussion on the external interrupts and the ISR.

It is the hardware designer's job to use the proper interrupt priority at the PIC level, but it is the ISR programmer's responsibility to know the timing requirements of each device when an ISR runs with either the same level or all interrupts disabled.

The embedded systems programmer, when designing and implementing an ISR, should be aware of the interrupt frequency of each device that can assert an interrupt. Table 10.1 contains a column called Maximum Frequency, which indicates how often a device can assert an interrupt when the device operates at maximum capacity. The allowed duration for an ISR to execute with interrupts disabled without affecting the system can be inferred from Table 10.1.

Without going into detail, an ISR, when executing with interrupts disabled, can cause the system to miss interrupts if the ISR takes too long. *Interrupt miss* is the situation in which an interrupt is asserted but the processor could not record the occurrence due to some busy condition. The interrupt service routine, therefore, is not invoked for that particular interrupt occurrence. This issue is typically true for a device that uses the edge-triggering mechanism to assert interrupts. The edge-triggering mechanism is discussed in 'The Nature of Spurious Interrupts' on page 163, section 10.6.

The RTOS kernel scheduler cannot run when an ISR disables all system interrupts while it runs. As indicated earlier, interrupt processing has higher priority than task processing. Therefore, real-time tasks that have stringent deadlines can also be affected by a poorly designed ISR.

Figure 10.9 illustrates a number of concepts as they relate to a single interrupt. In Figure 10.9, the value of T_A is based on the device interrupt frequency.

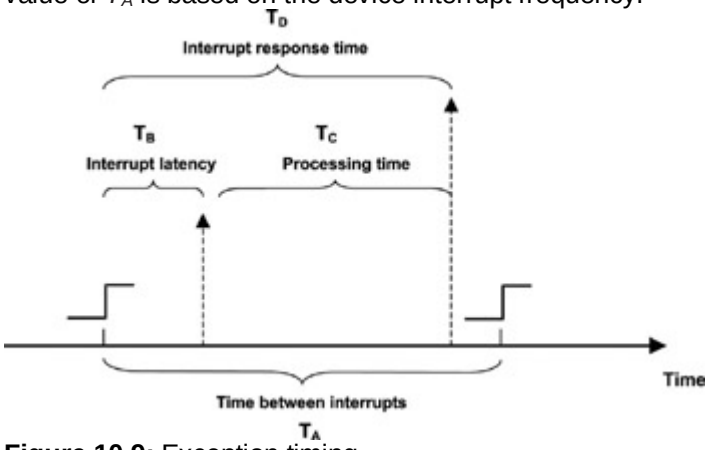


Figure 10.9: Exception timing.

The interrupt latency, T_B, refers to the interval between the time when the interrupt is raised and the time when the ISR begins to execute. Interrupt latency is attributed to:

- The amount of time it takes the processor to acknowledge the interrupt and perform the initial housekeeping work.
- A higher priority interrupt is active at the time.
- The interrupt is disabled and then later re-enabled by software.

The first case is always a contributing factor to interrupt latency. As can be seen, interrupt latency can be unbounded. Therefore, the response time can also be unbounded. The interrupt latency is

outside the control of the ISR. The processing time T_C, however, is determined by how the ISR is implemented.

The interrupt response time is $T_D = T_B + T_C$.

It is possible for the entire processing to be done within the context of the interrupt, that is, with interrupts disabled. Notice, however, that the processing time for a higher priority interrupt is a source of interrupt latency for the lower priority interrupt. Another approach is to have one section of ISR running in the context of the interrupt and another section running in the context of a task. The first section of the ISR code services the device so that the service request is acknowledged and the device is put into a known operational state so it can resume operation. This portion of the ISR packages the device service request and sends it to the remaining section of the ISR that executes within the context of a task. This latter part of the ISR is typically implemented as a dedicated daemon task.

There are two main reasons to partition the ISR into two pieces. One is to reduce the processing time within the interrupt context. The other is a bit more complex in that the architecture treats the interrupt as having higher priority than a running task, but in practice that might not be the case. For example, if the device that controls the blinking of an LED reports a failure, it is definitely lower in priority than a task that must send a communication reply to maintain its connection with the peer. If the ISR for this particular interrupt were partitioned into two sections, the daemon task that continues the LED interrupt processing can have a lower task priority than the other task. This factor allows the other higher priority task to complete with limited impact. Figure 10.10 illustrates this concept.

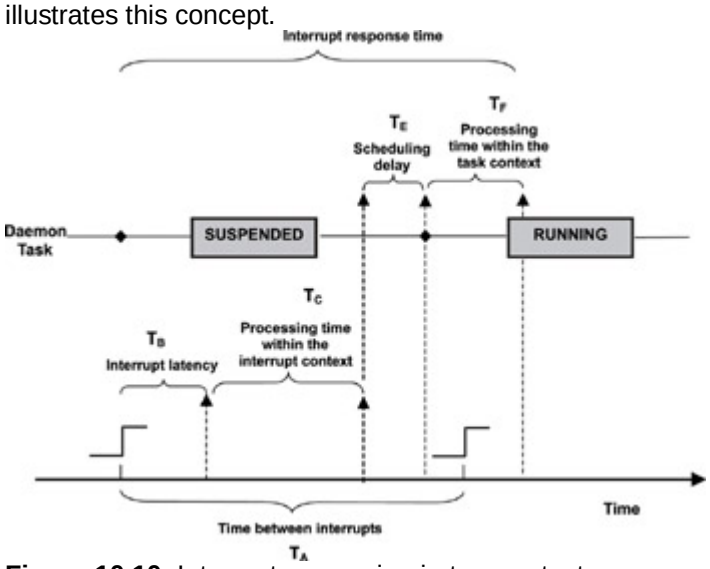


Figure 10.10: Interrupt processing in two contexts.

The benefits to this concept are the following:

- Lower priority interrupts can be handled with less priority than more critical tasks running in the system.
- This approach reduces the chance of missing interrupts.
- This approach affords more concurrency because devices are being serviced minimally so that they can continue operations while their previous requests are accumulated without loss to the extent allowed by the system.

On the other hand, the interrupt response time increases, because now the interrupt response time is $T_D = T_B + T_C + T_E + T_F$. The increase in response time is attributed to the scheduling delay, and the daemon task might have to yield to higher priority tasks.

The scheduling delay happens when other higher priority tasks are either running or are scheduled to run. The scheduling delay also includes the amount of time needed to perform a context switch after the daemon task is moved from the ready queue to the run queue.

In conclusion, the duration of the ISR running in the context of the interrupt depends on the number of interrupts and the frequency of each interrupt source existing in the system. Although general approaches to designing an ISR exist, no one solution exists to implement an ISR so that it works in all embedded designs. Rather the embedded systems developer must design an ISR according to the considerations discussed in this section.

General Guides

On architectures where interrupt nesting is allowed:

- An ISR should disable interrupts of the same level if the ISR is non-reentrant.
- An ISR should mask all interrupts if it needs to execute a sequence of code as one atomic operation.
- An ISR should avoid calling non-reentrant functions. Some standard library functions are non-reentrant, such as many implementations of `malloc` and `printf`. Because interrupts can occur in the middle of task execution and because tasks might be in the midst of the "`malloc`" function call, the resulting behavior can be catastrophic if the ISR calls this same non-reentrant function.
- An ISR must never make any blocking or suspend calls. Making such a call might halt the entire system.

If an ISR is partitioned into two sections with one section being a daemon task, the daemon task does not have a high priority by default. The priority should be set with respect to the rest of the system.

10.6 The Nature of Spurious Interrupts

A *spurious interrupt* is a signal of very short duration on one of the interrupt input lines, and it is likely caused by a signal glitch.

An external device uses a triggering mechanism to raise interrupts to the core processor. Two types of triggering mechanisms are *level triggering* and *edge triggering*. Figure 10.11 illustrates the variants of edge triggers (rising edge or falling edge). This kind of triggering is typically used with a digital signal.

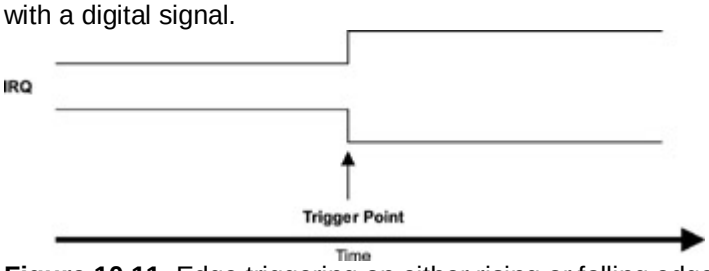


Figure 10.11: Edge triggering on either rising or falling edge.

In contrast, level triggering is commonly used in conjunction with an analog signal. Figure 10.12 illustrates how level triggering might be implemented in a design. It is important to note that when using level triggering, the PIC or microcontroller silicon typically defines the trigger threshold value.

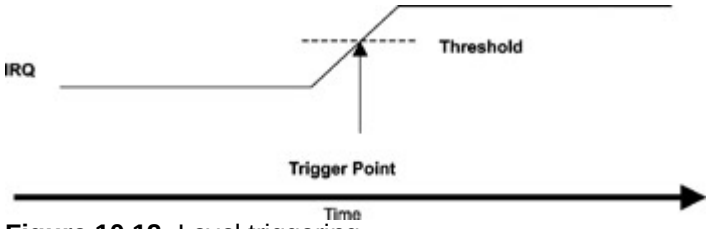


Figure 10.12: Level triggering.

How do spurious interrupts occur? In real-world situations, digital and analog signals are not as clean as portrayed here. The environment, types of sensors or transducers, and the method in which wiring is laid out in an embedded design all have a considerable effect on how clean the signal might appear. For example, a digital signal from a switch might require debouncing, or an analog signal might need filtering. Figure 10.13 provides a good illustration of how both digital and analog signals can really look. While electronic methods for debouncing and filtering fall beyond the realm of this book, it is important nonetheless to understand that input signals, whether for interrupts or other inputs, might not be as clean as a developer might envision them. These signals, therefore, can represent a potential source for sporadic behavior.

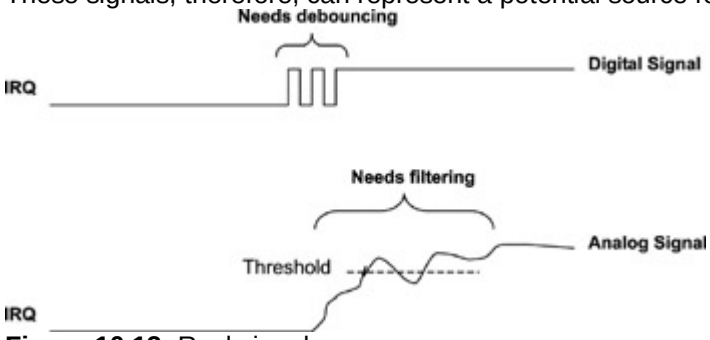


Figure 10.13: Real signals.

As can be seen, one reason for the occurrence of spurious interrupts is unstableness of the interrupt signal. Spurious interrupts can be caused when the processor detects errors while processing an interrupt request. The embedded systems programmer must be aware of spurious interrupts and know that spurious interrupts can occur and that this type of interrupt must be handled as any other type of interrupts. The default action from the kernel is usually sufficient.

10.7 Points to Remember

Some points to remember include the following:

- Exceptions are classified into synchronous and asynchronous exceptions.
- Exceptions are prioritized.
- External interrupts belong to the category of asynchronous exceptions.
- External interrupts are the only exceptions that can be disabled by software.
- Exceptions can be nested.
- Using a dedicated exception frame is one solution to solving the stack overflow problem that nested exceptions cause.
- Exception processing should consider the overall timing requirements of the system devices and tasks.
- Spurious interrupts can occur and should be handled as any other interrupts.