HighIntegritySystems

WITTENSTEIN

Interrupt Service Routines (ISRs)

Interrupts and ISRs

On a microprocessor system, an interrupt is a signal from hardware or software that needs immediate attention. The Task that is running must save its state on its stack and control passes to an Interrupt Service Routine (ISR) to handle the interrupt, shown in Figure 1. Once the ISR is completed the state that was saved on the stack is used to resume normal operations.

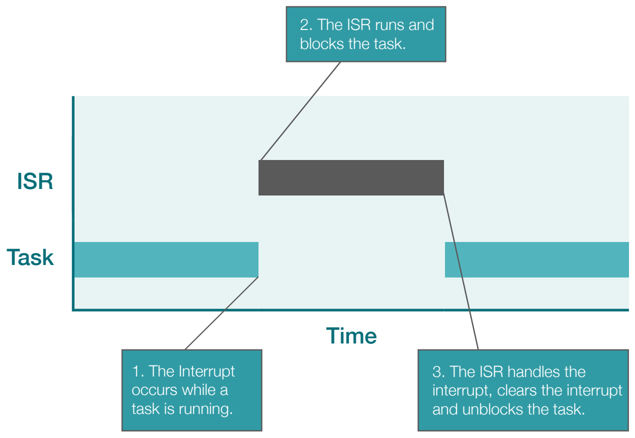
2. The ISR runs and blocks the task.
ISR
Task
Time
1. The Interrupt occurs while a task is running.
3. The ISR handles the interrupt, clears the interrupt and unblocks the task.

Figure 1. *The Interrupt Service Routine (ISR)*

How interrupts are handled is dependent on both the hardware and the Real-Time Operating System (RTOS). A microprocessor will provide a number of hardware interrupts to be used by system devices and peripherals, as well as a number of programmable software interrupts that the engineer can use. An RTOS typically requires a timer interrupt to maintain an internal time base.

The microprocessor includes different priorities for different types of interrupts. In some cases the microprocessor may support the nesting of interrupts and allow the RTOS to manage this.

An ISR (also called an interrupt handler) takes its priority from that of the interrupt. Priorities assigned to interrupts are entirely separate from priorities assigned to Tasks; ISRs will pre-empt any running Task.

SAFERTOS provides API functions that are designed to be safe to use inside an ISR.

ISRs and Tasks

When programming an ISR, it is important to remember that the lowest-priority ISR will always pre-empt the highest-priority Task. While an ISR is running, other Tasks and functions are delayed, potentially causing irregular processing of lower-priority Tasks. Therefore, it is advisable to keep the ISR as short as possible.

Ideally, the ISR should merely collect data and reset the interrupt source. Whenever possible, further processing should be deferred to Tasks, as illustrated in Figure 2. The elapsed time of an ISR should be short compared with the intervals between interrupts. This can avoid undesirable delays in processing of other functions.

An example of this is when data must be converted to another format, rather than stored in its original format. The conversion can be deferred to a Task and interrupts re-enabled immediately.

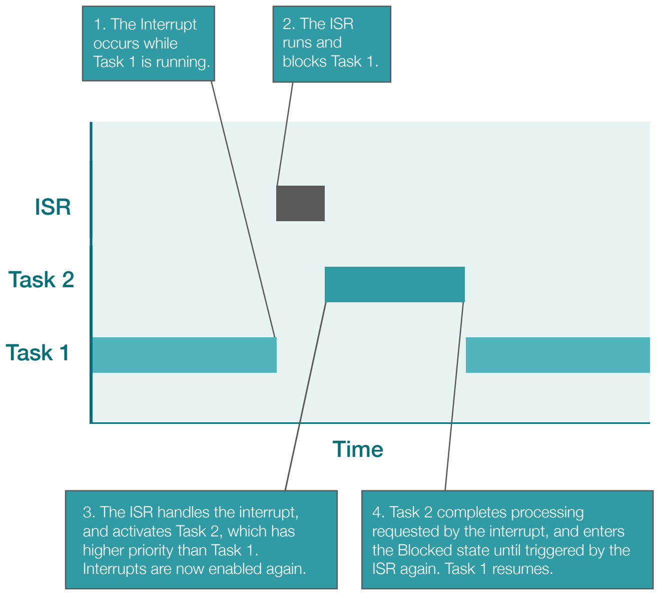
1. The Interrupt occurs while Task 1 is running.
2. The ISR runs and blocks Task 1.
ISR
Task 2
Task 1
Time
3. The ISR handles the interrupt, and activates Task 2, which has higher priority than Task 1. Interrupts are now enabled again.
4. Task 2 completes processing requested by the interrupt, and enters the Blocked state until triggered by the ISR again. Task 1 resumes.

Figure 2. *The Deferred Processing Model*

Critical Code Sections

Critical code sections are sections of code that must be executed as an atomic whole; they cannot be interrupted. RTOS macros are inserted into the code to indicate the beginning and end of the critical code sections.

In a simple, un-nested system the RTOS will turn off interrupts for the duration of the critical code sections, delaying the operation of the interrupt until after the critical code section has completed. This introduces Interrupt latency. The length of critical sections should be minimised, as interrupts usually need to be processed as quickly as possible.

In a nested system it is possible to set an interrupt priority threshold level. If a specific interrupt is assigned a priority level above the interrupt priority threshold level, the interrupt will not be disabled during critical code sections. This reduces the effect of interrupt latency, however an interrupt set above this level must not use any RTOS API calls. If the interrupt is assigned a priority level below the interrupt priority threshold level then it will be disabled during critical code sections, but interrupt friendly RTOS API calls can be used, and it could result in interrupt latency.

Setting an interrupt with a priority level above the interrupt priority threshold level is a technique used to achieve highly accurate, precision interrupt timing, and is frequently used in motor control applications.

Interrupt Processing

Microcontrollers use an interrupt controller to manage the way that asynchronous events are recognised and presented to the processing unit. These broadly take the form of a vectored interrupt model or a centralised interrupt handler model.

The centralised interrupt handler model works by the hardware monitoring or polling multiple interrupt sources and invoking a single interrupt handler within the software. The software is then responsible for determining the source of the interrupt and processing the event.

A vectored interrupt uses a table known as the “Vector Table” to go directly to the start of the ISR that handles the interrupt with that vector. The engineer is responsible for programming the ISR and ensuring it is correctly entered into the vector table. However, at run time much of the interrupt handling (such as saving registers) is directly controlled by the hardware.

Considerations with ISRs

Programming ISRs needs special care.

- Interrupts by their nature are asynchronous and can occur in any order.
- Errors can arise from processing that inadvertently depends on a state that may not exist at runtime.
- Interrupted Tasks must be resumed after the interrupt processing has completed, as if nothing has changed.
- Errors in ISRs can be difficult to debug and correct, especially as the effect may show up later in time than the execution of the error.
- The ISR must be able to complete, else the hardware may not be left in the correct state to receive another interrupt.

Typically an RTOS will have an alternative set of API calls that can be safely used from within an ISR. These ISR friendly API calls will not block or wait for events, semaphores or data on queues. Calling a standard API call that blocks/ waits for events from within an ISR will result in indeterminate behaviour and it is likely the application will lock up or crash.

RTOS and ISRs

Subject to hardware capabilities and programming choices by the engineer, on the ISR the RTOS can:

- Save the system context (flags, registers etc.);
- Disable all interrupts, unless specifically set by the engineer to allow other interrupts;
- Identify the interrupt that has occurred and take default actions, subject to the specific programming of the application;
- At the end of processing, restore the system context from the stack or variables;
- Re-enable all the blocked interrupts and resume execution of the interrupted Task.

WITTENSTEIN high **integrity** systems

Worldwide Sales and Support
Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@high**integrity**systems.com

WITTENSTEIN