Interrupts and Exceptions

The Univac 1103, in the early 1960s, is credited as the first computer to fully implement interrupts.

Interrupt and exception processing may be regarded as the core mechanism driving all modern operating systems.

Many real-time operating systems provide wrapper functions to handle exceptions and interrupts, in order to shield the application programmer from the low-level details.

FreeRTOS provides minimal shielding, which makes it good for teaching students to be embedded *systems* programmers rather than just embedded *applications* programmers.

Definitions

An *exception* is any event that disrupts the normal execution of the processor and forces the processor into execution of special instructions (usually in a *privileged* state).

Exceptions are a “hardware thing.”

The CPU normally just repeats a fetch-execute cycle.

At the end of each cycle, if an exception flag is set, then the processor saves the address of the next instruction that it intends to fetch, *vectors* to the exception handler, and begins executing code there.

Exceptions and Interrupts

There are two types of exception:

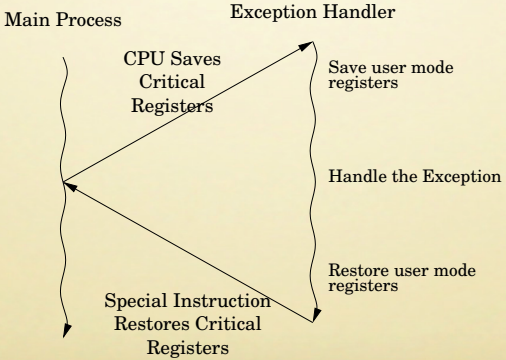
synchronous Exceptions that are caused by events internal to the CPU, such as executing a syscall (software interrupt) instruction, attempting to divide by zero, or attempting to access an instruction or data that is not aligned properly.

asynchronous Exceptions that are caused by events external to the CPU, such as devices signaling that they need attention. These exceptions are often referred to as *interrupts,* to distinguish them from synchronous exceptions.

Uses

Exceptions are used for:

- managing internal errors and special conditions,
- hardware concurrency, and
- management of Operating System service requests.



External Interrupts

Most processors have a single interrupt input for all external interrupts.

Each device has a flag (or flags) that can be read to determine whether or not the device is signaling an interrupt.

On very small systems, the interrupt handler can simply check (or *poll*) each device, in order of *priority*, to see which one is signal-ling the interrupt.

However, if there are many devices, then polling takes too long.

There are two types of hardware device that can help:

- Priority Interrupt Controller (PIC)
- Vectored Interrupt Controller (VIC)

Priority Interrupt Controller (PIC)

- The priority interrupt controller has multiple input pins.
- Each device may be assigned a unique pin, or some devices may share a pin (the interrupts from the devices are ORed together).
- Each pin has a priority.
- The ISR checks the PIC to get the highest priority pin that is signaling an interrupt.
- The ISR polls the set of devices connected to the indicated pin.

Typically, the PIC will have between eight and sixteen inputs, and the multiple PICS can be chained together in various ways.

The IBM PC had an eight-input PIC. The IBM PC AT had two of those PICS, connected to provide 15 priority levels.

Vectored Interrupt Controller (VIC)

- The controller typically has an input pin for each device.
- Each pin has a priority, and the address (the vector) of the ISR is also stored in the VIC.
- When an interrupt comes in, the VIC sends the interrupt *and* the address of the ISR (the vector) to the CPU.
- The hardware automatically jumps to the ISR, without having to determine the source in software.

Some VICs may require the ISR to read the vector from the VIC and then jump to the vector.

Some VICs may require that each device ISR be at a specific location in memory.

Most modern systems (even very small ones) have a VIC.

Cortex M3 NVIC

Accessing the Cortex-M3 NVIC registers using CMSIS

General Exception Classes

Most modern systems have four types of exception:

Asynchronous maskable: external interrupts that can be disabled/enabled by software (writing an enable bit somewhere)

Asynchronous non-maskable: external exceptions that cannot be disabled (e.g. hardware reset). Some processors have a separate NMI input pin.

Synchronous precise: When the exception occurs, the exception handler can determine the exact address of the instruction that caused the exception.

Synchronous imprecise: The exception handler cannot determine the exact instruction that caused the exception. This may happen due to pipelining and out-of-order execution.

Priority

Although it could vary, exception classes usually have the following priority:

- Asynchronous non-maskable (highest)
- Synchronous precise
- Synchronous imprecise
- Asynchronous maskable (lowest)

Higher priority exceptions can usually interrupt processing of lower priority exceptions.

All exceptions have priority over application (and even OS) code, including tasks, semaphores, queues, etc.

General Exception Processing – Hardware

In general, the CPU must take certain steps when an exception occurs:

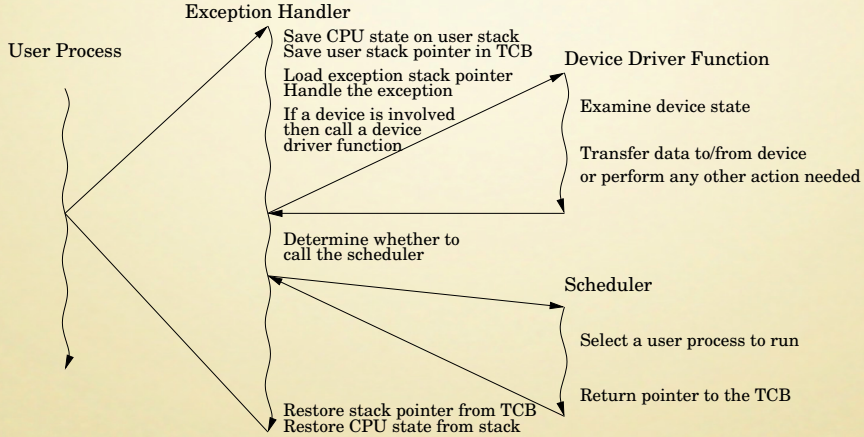
- Save the most critical current processor state, so that it can be restored later
- Load the address of the exception or interrupt handler into the program counter (jump to the handler)
- The handler should end with a special instruction that restores the critical processor state, to resume executing the user task.

General Exception Processing – Software

The exception handling software (middle item in previous list) must:

- Save additional registers/system state on the user stack
- Switch to the exception handler stack (Save original stack pointer!)
- Ensure that exceptions with lower or equal priority are masked
- Perform minimum amount of work to handle the exception (a dedicated task may be used to finish processing later)
- Possibly call the scheduler to choose a new “running” task
- Load the stack pointer for the running task
- Restore the remaining registers from the stack
- Execute special instruction to return to user mode and pick up where the user code was interrupted

General Exception Processing – Hardware and Software



Installing Exception Handlers

Most embedded system development environments provide some easy way for the programmer to define exception handlers.

The executable image includes startup code which, among other things, installs the exception handlers.

Typically, the system also provides a *default* or *general* exception handler for any exceptions that the programmer has not explicitly provided.

On GCC based development environments, the normal method for specifying an exception handler is to use either the *interrupt pragma* or the *interrupt attribute*

Exception handlers can be written in assembly, C, or C++. If written in assembly, there should be a C prototype somewhere, so that the C compiler knows about the handler.

Interrupt Pragma

The syntax for the GCC interrupt pragma in C is:

```
1 #pragma INTERRUPT(func)
```

This simply tells the compiler that `func` will be used as an exception handler.

If `func` is written in C, then the compiler will generate function entry and exit code that *may* work properly for an exception handler.

Depending on the system, it may be up to the programmer to set up the VIC to vector to the interrupt function.

Function Attributes

With GCC, attributes can be specified on a function prototype, or on a function definition.

Function attributes for GCC are processor dependent. Documentation can be found here.

The following function prototype on a MIPS processor:

```
1 void __attribute__((interrupt(IPL1AUTO),vector(VECTOR_NUMBER))) func( void );
```

specifies that `func` is an exception handler, using the MIPS VIC, it can be given a default priority, and it should be connected to the specified exception vector.

The vector number will also be system dependent. On the PIC32 with MPLAB, the vectors can be found in
`...xc32/v2.30/pic32mx/include/proc/<PIC_VERSION>.h`

Saving and Restoring Processor State

The tricky bit of writing an exception handler is getting the entry and exit code to work.

The entry code must save the *Processor State Information* somewhere in memory. Usually, the "somewhere" is on the stack of the currently running task or exception handler.

The goal is to save the minimum amount of state (registers) that can allow the exception handler to do its job, and then restore the state so that the interrupted code can continue to operate as if nothing ever happened.

Examples in `FreeRTOS/Source/portable/MPLAB/PIC32MX/ISR_Support.h` and `FreeRTOS/Source/portable/GCC/ARM7_AT91FR40008/portmacro.h`

Nested Exceptions

Since exceptions have priority levels, an exception can occur in the middle of handling an exception.

How many levels can occur?

Does each exception type or level have its own stack?

ALL PROCESSOR DEPENDENT!

One way to deal with nested exceptions is with *exception frames*. Similar to the stack frame created when calling a function/subroutine. Basically, when an exception occurs, a new stack area is created for processing that exception.

ESR vs ISR

Exceptions are generated within the CPU, so there is usually more information easily accessible to an ESR than to an ISR.

ISRs can mask interrupts in various ways so that no additional interrupts can occur. Many exceptions cannot be masked.

Interrupt handlers must run as quickly as possible, to avoid missing interrupts, and to ensure that timing constraints are met. Exception handlers do not usually have such stringent requirements.

Interrupt Response Time

The time taken to process an interrupt:

$$T_D = T_B + T_C$$

where T_B is the *interrupt latency*, and T_C is the *interrupt processing time*.

T_C can be estimated by counting instructions

T_B is more difficult:

- Best case, it is the time taken to vector to the ISR and save the critical state information.
- What if a higher priority interrupt is being executed when the interrupt occurs?
- What if application code has disabled interrupts temporarily to make sure it gets through a critical section without an ISR causing a race condition?

Interrupt Response Time – Part 2

This is why we want ISR to be as short and fast as possible. Also why we want *all* critical sections to be as short and fast as possible.

Daemon Tasks can help keep the ISR short. ISR just sends message or uses semaphore, mutex, etc to wake up a daemon task. The daemon task actually does the processing, and interrupts do not need to be masked while it does so (except possibly for short critical sections).

$$T_D = T_B + T_C + T_E + T_F$$

where T_E is the *scheduling delay*, and T_F is the *processing time in the daemon task*.

Exam Question: Pros/cons of two-part exception processing?

Spurious Interrupts

Spurious interrupts can waste enormous amounts of CPU time, and make timing guarantees worthless.

- Edge triggered interrupt with slow rising/falling edge and insufficient hysteresis (add a Schmidt trigger).
- “Open” interrupt line and induced current (tie it to power/ground and disable that particular interrupt).
- Digital glitch (re-design the device generating the interrupt, or add debounce hardware).

If you can’t get rid of all spurious interrupts, just process them as quickly as possible.