An **interrupt** is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution. This hardware event is called a **trigger**. The hardware event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer). When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag. A **thread** is defined as the path of action of software as it executes. The execution of the interrupt service routine is called a background thread. This thread is created by the hardware interrupt request and is killed when the interrupt service routine returns from interrupt (e.g., by executing a **BX LR**). A new thread is created for each interrupt request. It is important to consider each individual request as a separate thread because local variables and registers used in the interrupt service routine are unique and separate from one interrupt event to the next interrupt. In a **multi-threaded** system, we consider the threads as cooperating to perform an overall task. Consequently we will develop ways for the threads to communicate (e.g., FIFO) and to synchronize with each other. Most embedded systems have a single common overall goal. On the other hand, general-purpose computers can have multiple unrelated functions to perform. A **process** is also defined as the action of software as it executes. Processes do not necessarily cooperate towards a common shared goal. Threads share access to I/O devices, system resources, and global variables, while processes have separate global variables and system resources. Processes do not share I/O devices.

VIDEO 12.1A INTERRUPT BASICS

C12 1a Interrupts YouTube

DR. RAMESH YERRABALLI: So let's take a look interrupts are.

The definition of an interrupt is that it's

a hardware triggered software action.

DR JONATHAN VALVANO: Professor Yerraballi, what if I had two tasks

I wanted my software to

8:27 / 8:27 1.0x

DR. RAMESH YERRABALLI: Let's say we take a busy wait approach.

So there are two tasks that we want to perform.

We check for the condition for the first task.

Is it time to do the first task?

And the condition says, no, it's not time to do the first task.

So we repeat and check again.

So this says no.

So we keep checking.

And when the answer is yes, we perform the task, whatever

the task might be-- Task 1, let's say.

And then we go on, check the same thing for the second task--

whether the condition is satisfied for the second task.

If it's not satisfied, I go back, check again.

And then if it is satisfied, I will perform

the second task whatever the task might be.

And then I come down and perform other tasks.

This is the other things that I perform.

And when I'm done, I'm going to go back there and repeat this process.

Now, this is obviously very wasteful of resources.

Because if the first task is not ready but the second task is ready,

I can't go to the second task because I'm stuck in this wait state.

And the others only get done when both the tasks have being performed.

So as we said, one of the things about an Interrupt-based solution

is that we do not have to wait for something that has to be ready.

When the task is ready, it will interrupt us.

So here we have our Task 1.

And we perform the Task 1 when there is an interrupt.

So this is some Interrupt 1.

The interrupt causes the Task 1 to perform the task, Task1

and we return from interrupt.

So again, if the second task is ready-- the

Help

conditions for the second task are

ready--there'll be an Interrupt 2 that will trigger this task.

We perform the task, Task 2.

And we return from interrupt.

Now the good thing is, the main program, which is a loop, if you remember,

will perform some set up so that it enables the interrupts.

And it performs these other functions-- other interesting things,

which is the same as this.

And it continues to perform these other functions, which

may be of importance and which are predictable.

And when an interrupt occurs, we have a suspension.

We suspend.

We perform the task.

Let's say that this task is the one.

When the return from interrupt occurs, we resume where we left off.

This way, we are responsive to the interrupt

without having to continuously check for whether the conditions are

satisfied for the task.

So we saw the overall idea behind interrupts.

So let's delve into the details.

First, we'll look at the conditions that need

to be satisfied for an interrupt to occur.

This is usually part of our setup-- our setup ritual, if you will.

So the conditions that need to be satisfied,

the first condition is we have to arm the device-- the external device

or the internal device that is going to interrupt the microcontroller.

There is some flag associated with it that

says that this device is now able to interrupt our processor.

The second thing that we're going to do is

we're going to enable the device specifics.

There is a module within our microcontroller called the Nested

Vectored Interrupt Controller, which it's

kind of the module that controls all interrupts.

So we enable the NVIC specifically for that particular device-- device

specific.

Now the third thing we will do is there is

a global enable bit called the I bit.

This is a global interrupt enable.

We will make sure that this is equal to 0.

The I bit is used usually to turn all interrupts off or turn all interrupts

on.

It is the main interrupt flag.

The fourth thing that we have to do is the priority.

That is, devices can have associated priorities.

In other words, the interrupts can have specific priorities.

That is, if you want the device to interrupt the processor,

the device should have a priority higher than the task that

is currently being run.

If you have two devices, then we set their priorities

relatively so that we decide which device

can interrupt which device while it's being serviced.

The last thing that we want to look at is the trigger.

Unlike these steps-- unlike the Steps 1, 2, 3,

4 which are part of the setup ritual-- the trigger is an asynchronous event.

This is what causes the interrupt to occur.

So let's say our device were a switch-- if I

had a switch, a button if you will.

So when the switch is open, let's say there

is no trigger-- if that's what I'm looking for.

But once the switch is closed, that causes a trigger.

Which means that the device causes an interrupt, which

is registered by the module.

And it is reacted to.

So in our case, what we will see is there

will be a bit-- an RIS register.

And there'll to be some bit here that will

be enabled to tell us that the trigger has occurred.

DR JONATHAN VALVANO: What does RIS stand for?

DR. RAMESH YERRABALLI: Oh, yeah.

RIS stands for Raw Interrupt Status.

DR JONATHAN VALVANO: So these are where the trigger flags are?

DR. RAMESH YERRABALLI: These are where the trigger flags are stored.

And this is what the processor is checking when it knows to figure it out

when it has to cause an interrupt.

Help

There are no standard definitions for the terms mask, enable, and arm in the professional, Computer Science, or Computer Engineering communities. Nevertheless, in this class we will adhere to the following specific meanings. To **arm** a device means to allow the hardware trigger to interrupt. Conversely, to **disarm** a device means to shut off or disconnect the hardware trigger from the interrupts. Each potential interrupting trigger has a separate arm bit. One arms a trigger if one is interested in interrupts from this source. Conversely, one disarms a trigger if one is not interested in interrupts from this source. To **enable** means to allow interrupts at this time. Conversely, to **disable** means to postpone interrupts until a later time. On the ARMCortex-M processor there is one interrupt enable bit for the entire interrupt system. We disable interrupts if it is currently not convenient to accept interrupts. In particular, to disable interrupts we set the I bit in **PRIMASK**. In C, we enable and disable interrupts by calling the functions **`EnableInterrupts()`** and **`DisableInterrupts()`** respectively.

The software has dynamic control over some aspects of the interrupt request sequence. First, each potential interrupt trigger has a separate **arm** bit that the software can activate or deactivate. The software will set the arm bits for those devices from which it wishes to accept interrupts, and will deactivate the arm bits within those devices from which interrupts are not to be allowed. In other words it uses the arm bits to individually select which devices will and which devices will not request interrupts. For most devices there is a enable bit in the NVIC that must be set (periodic SysTick interrupts are an exception, having no NVIC enable). The third aspect that the software controls is the interrupt enable bit. Specifically, bit 0 of the special register **PRIMASK** is the interrupt mask bit, **I**. If this bit is 1 most interrupts and exceptions are not allowed, which we will define as **disabled**. If the bit is 0, then interrupts are allowed, which we will define as **enabled**. The fourth aspect is priority. The **BASEPRI** register prevents interrupts with lower priority interrupts, but allows higher priority interrupts. For example if the software sets the **BASEPRI** to 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. The software can also specify the priority level of each interrupt request. If **BASEPRI** is zero, then the priority feature is disabled and all interrupts are allowed. The fifth aspect is the external hardware trigger. One example of a hardware trigger is the **Count** flag in the **NVIC_ST_CTRL_R** register which is set periodically by SysTick. Another example of hardware triggers are bits in the **GPIO_PORTF_RIS_R** register that are set on rising or falling edges of digital input pins. Five conditions must be true for an interrupt to be generated:

1) device arm,
2) NVIC enable,
3) global enable,
4) interrupt priority level must be higher than current level executing, and
5) hardware event trigger.

For an interrupt to occur, these five conditions must be simultaneously true but can occur in any order.

An interrupt causes the following sequence of five events. First, the current instruction is finished. Second, the execution of the currently running program is suspended, pushing eight registers on the stack (**R0**, **R1**, **R2**, **R3**, **R12**, **LR**, **PC**, and **PSR** with the **R0** on top). If the floating point unit on the TM4C123 is active, an additional 18 words will be pushed on the stack representing the floating point state, making a total of 26 words. Third, the **LR** is set to a specific value signifying an interrupt service routine (ISR) is being run (bits [31:4] to 0xFFFFFFF, and bits [3:0] specify the type of interrupt return to perform). In our examples we will see LR is set to 0xFFFFFFF9. If the floating point registers were pushed, the LR will be 0xFFFFFFE9. Fourth, the **IPSR** is set to the interrupt number being processed. Lastly, the **PC** is loaded with the address of the ISR (vector).

1) Current instruction is finished,
2) Eight registers are pushed on the stack,
3) LR is set to 0xFFFFFFF9,
4) IPSR is set to the interrupt number,
5) PC is loaded with the interrupt vector

Help

These five steps, called a **context switch**, occur automatically in hardware as the context is switched from a foreground thread to a background thread. We can also have a context switch from a lower priority ISR to a higher priority ISR. Next, the software executes the ISR.

If a trigger flag is set, but the interrupts are disabled (I=1), the interrupt level is not high enough, or the flag is disarmed, the request is not dismissed. Rather the request is held **pending**, postponed until a later time, when the system deems it convenient to handle the requests. In other words, once the trigger flag is set, under most cases it remains set until the software clears it. The five necessary events (device arm, NVIC enable, global enable, level, and trigger) can occur in any order. For example, the software can set the I bit to prevent interrupts, run some code that needs to run to completion, and then clear the I bit. A trigger occurring while running with I=1 is postponed until the time the I bit is cleared again.

Clearing a trigger flag is called **acknowledgement**, which occurs only by specific software action. Each trigger flag has a specific action software must perform to clear that flag. We will pay special attention to these enable/disable software actions. The SysTick periodic interrupt will be the only example of an automatic acknowledgement. For SysTick, the periodic timer requests an interrupt, but the trigger flag will be automatically cleared when the ISR runs. For all the other trigger flags, the ISR must explicitly execute code that clears the flag.

The **interrupt service routine** (ISR) is the software module that is executed when the hardware requests an interrupt. There may be one large ISR that handles all requests (polled interrupts), or many small ISRs specific for each potential source of interrupt (vectored interrupts). The design of the interrupt service routine requires careful consideration of many factors. Except for the SysTick interrupt, the ISR software must explicitly clear the trigger flag that caused the interrupt (acknowledge). After the ISR provides the necessary service, it will execute **BX LR**. Because LR contains a special value (e.g., 0xFFFFFFF9), this instruction pops the 8 registers from the stack, which returns control to the main program. If the LR is 0xFFFFFFE9, then 26 registers (R0-R3,R12,LR,PC,PSW, and 18 floating point registers) will be popped by **BX LR**.
There are two stack pointers: PSP and MSP. The software in this class will exclusively use the MSP. It is imperative that the

ISR software balance the stack before exiting. Execution of the previous thread will then continue with the exact stack and register values that existed before the interrupt. Although interrupt handlers can create and use local variables, parameter passing between threads must be implemented using shared global memory variables. A private global variable can be used if an interrupt thread wishes to pass information to itself, e.g., from one interrupt instance to another. The execution of the main program is called the foreground thread, and the executions of the various interrupt service routines are called background threads.

An axiom with interrupt synchronization is that the ISR should execute as fast as possible. The interrupt should occur when it is time to perform a needed function, and the interrupt service routine should perform that function, and return right away. Placing backward branches (busy-wait loops, iterations) in the interrupt software should be avoided if possible. The percentage of time spent executing interrupt software should be small when compared to the time between interrupt triggers.