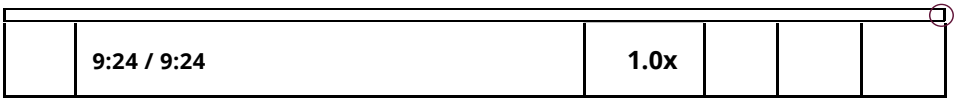
VIDEO 12.1B INTERRUPT PROCESSING - THE CONTEXT SWITCH

C12 1b Interrupt Processing YouTube



DR. JONATHAN VALVANO: Next, let's look at the context switch.

The trigger has happened, and the context switch

is the process by switching from the main program

into the interrupt service routine.

So we're executing here in the main program.

The trigger happens, and we're going to switch over

to the interrupt service routine.

DR. RAMESH YERRABALLI: So can you show me

the details of the program that's currently running?

Maybe an instruction or something that gets interrupted?

DR. JONATHAN VALVANO: Sure.

The main program might be reading and it might be storing.

So the program counter might be pointing to an instruction as it executes.

DR. RAMESH YERRABALLI: So this is the assembly code of the C code

that we may have written, and this is what

is being run right now?

DR. JONATHAN VALVANO: Yeah.

So we're in the main program running something.

Program counter points to where we are.

And now the trigger has happened.

The first thing that will happen is-- assuming the priority is high enough,

and it's armed and enabled, and the I bit is all 0,

like we did before-- the first thing that will happen

is it will finish the instruction.

Instructions take a finite amount of time to execute.

So this instruction that I'm executing will finish.

The second thing that has to happen is we have to suspend.

Suspend means make a record of where we were.

And that is going to be used, or recorded, onto our stack.

And the stack pointer points to the top of the stack.

And when the interrupt happens, we are going to push R0, R1, R2, R3.

pushes R12, R14-- which is the link register--

R15, which is the program counter.

And it will push the program status word.

And the order it does it is a program status word is here.

Program counter, link register, R12 a lot of registers-- R3, R2, R1, R0,

such that the new stack pointer is pointing way up here,

and eight registers have been pushed on the stack.

DR. RAMESH YERRABALLI: So why are we only pushing these registers?

And what about those other registers, R4 through R11?

DR. JONATHAN VALVANO: These eight registers

are actually not pushed on the stack.

And so when the programmer writes their interrupt service routine,

we will not use R4 through R11.

They're not used in the interrupt service routine.

DR. RAMESH YERRABALLI: So there's no need to save them.

DR. JONATHAN VALVANO: Exactly.

The third step is to set the link register to a special code.

And this special code is 0xFFFFFFF9.

You notice the link register is odd, and it's got this weird thing.

This pattern means I'm running in an interrupt service routine.

DR. RAMESH YERRABALLI: And we will see why this special pattern is required

when we see the details of how we return from an interrupt

to the main program that was interrupted.

DR. JONATHAN VALVANO: Exactly.

The IPSR register is going to be set to the interrupt number.

There are hundreds of interrupts.

Help

And so this register will contain the interrupt number

of the device that's currently executing in the ISR.

And finally, the program counter is loaded

with the address of the interrupt service routine.

And this is called the vector.

DR. RAMESH YERRABALLI: So with this five step process,

we suspended the current of program, which is a main--

and we call this a main thread-- and we were able to transfer control

to the interrupt service routine thread.

So we have two threads-- the main thread,

and the interrupt service routine thread.

And the control now is with the interrupt service routine thread.

DR. JONATHAN VALVANO: You want to see how we get back?

DR. RAMESH YERRABALLI: Yeah.

Let's look at it.

DR. JONATHAN VALVANO: Right here at the end of the interrupt service routine

is in instruction call BX LR.

And that is to attempt to put the link register back into the program counter.

But because the link register has this special number,

instead of storing this in the program counter

and jumping off into nowhere land, what it will do is it will then pop.

This BX LR instruction will then pop these eight registers off of the stack,

including the program counter.

And so this instruction here will return back to where we interrupted from.

DR. RAMESH YERRABALLI: Which will bring the program

counter to the next instruction, because that's what was saved on the stack.

DR. JONATHAN VALVANO: Perfect.

DR. RAMESH YERRABALLI: So interrupts are a powerful mechanism.

But with power such as this comes responsibility.

That is, in your interrupt service routine,

you have to be careful to do certain things and not do certain other things.

DR. JONATHAN VALVANO: So let's begin with good practices.

One of the important things to do in the interrupt service routine

is to acknowledge.

Acknowledging is to clear the trigger flag.

Remember, the trigger flag caused the interrupt to happen.

And the software should clear it.

DR. RAMESH YERRABALLI: So not clearing the trigger

has the potential to make the interrupt be considered unserviced.

And therefore, the interrupt continues to be triggered again and again,

so you get stuck in your interrupt service routine.

DR. JONATHAN VALVANO: Yeah, we call that a crash.

That's not good.

So we will acknowledge.

The second thing we're going to do, think about,

when we write interrupt service routines

is to make them very, very short.

In a short interrupt service routine, we'll

guarantee that all the interrupts get service.

Well, while we're talking about what to do, let's talk about what not to do.

I think good interrupt service routines do not have delay loops.

This means that I don't think that once you get an interrupt service

routine that you should wait, or loop, over and over and over again.

The second thing has to do with how often we trigger an interrupt.

So let's do a profile.

In other words, if the interrupts were to occur at this rate,

it will be important for us not to have the interrupt service routine take

Help

Performance measures: latency and bandwidth. For an input device, the **interface latency** is the time between when new input is available, and the time when the software reads the input data. We can also define **device latency** as the response time of the external I/O device. For example, if we request that a certain sector be read from a disk, then the device latency is the time it take to find the correct track and spin the disk (seek) so the proper sector is positioned under the read head. For an output device, the interface latency is the time between when the output device is idle, and the time when the software writes new data. A **real-time** system is one that can guarantee a worst case interface latency. **Bandwidth** is defined as the amount of data/sec being processed.

Many factors should be considered when deciding the most appropriate mechanism to synchronize hardware and software. One should not always use busy wait because one is too lazy to implement the complexities of interrupts. On the other hand, one should not always use interrupts because they are fun and exciting. Busy-wait synchronization is appropriate when the I/O timing is predictable and when the I/O structure is simple and fixed. Busy wait should be used for dedicated single thread systems where there is nothing else to do while the I/O is busy. Interrupt synchronization is appropriate when the I/O timing is variable, and when the I/O structure is complex. In particular, interrupts are efficient when there are I/O devices with different speeds. Interrupts allow for quick response times to important events. In particular, using interrupts is one mechanism to design real-time systems, where the interface latency must be short and bounded. **Bounded** means it is always less than a specified value. **Short** means the specified value is acceptable to our consumers.

Interrupts can also be used for infrequent but critical events like power failure, memory faults, and machine errors. Periodic interrupts will be useful for real-time clocks, data acquisition systems, and control systems. For extremely high bandwidth and low latency interfaces, direct memory access (DMA) should be used. DMA is described in Embedded Systems: Real-Time Operating Systems for ARM® Cortex™-M Microcontrollers . This example uses a timer and DMA to input data from an input port and store it in RAM http://users.ece.utexas.edu/~valvano/arm/DMATimer_4F120.zip (http://users.ece.utexas.edu/%7Evalvano/arm/DMATimer_4F120.zip) and is beyond the scope of this introductory class.

An **atomic** operation is a sequence that once started will always finish, and cannot be interrupted. All instructions on the ARM®Cortex™-M processor are atomic except store and load multiple, **`STM LDM PUSH POP`**. If we wish to make a section of code atomic, we can run that code with I=1. In this way, interrupts will not be able to break apart the sequence. Again, requested interrupts that are triggered while I=1 are not dismissed, but simply postponed until I=0. In particular, to implement an atomic operation we will 1) save the current value of the **PRIMASK**, 2) disable interrupts, 3) execute the operation that needs to run atomically, and 4) restore the **PRIMASK** back to its previous value.

CHECKPOINT 12.1

What five conditions must be true for an interrupt to occur?

Hide Answer

Trigger flag set by hardware; the device is armed by software; the device is enabled for interrupts in the NVIC; the processor is enabled for interrupts (PRIMASK I bit is clear); the interrupt level must be less than the BASEPRI. The order of these conditions does not matter.

CHECKPOINT 12.2

How do you enable interrupts?

Hide Answer

The processor is enabled for interrupts by clearing the I bit in the PRIMASK. Execute
CPSIE I

Help

CHECKPOINT 12.3

What are the steps that occur when an interrupt is processed?

Hide Answer

Instruction is finished; registers R0-R3, R12, LR, PC, and PSR are pushed; LR is set to 0xFFFFFFF9; IPSR is set to the interrupt number being processed; PC is set with interrupt vector address. The last three steps can occur in any order.

As you develop experience using interrupts, you will come to notice a few common aspects that most computers share. The following paragraphs outline three essential mechanisms that are needed to utilize interrupts. Although every computer that uses interrupts includes all three mechanisms, how the mechanisms operate will vary from one computer to another.

All interrupting systems must have the **ability for the hardware to request action from computer**. In general, the interrupt requests can be generated using a separate connection to the processor for each device. The TM4C microcontrollers use separate connections to request interrupts.

All interrupting systems must have the **ability for the computer to determine the source**. A vectored interrupt system employs separate connections for each device so that the computer can give automatic resolution. You can recognize a vectored system because each device has a separate interrupt vector address. With a polled interrupt system, the interrupt software must poll each device, looking for the device that requested the interrupt. Most interrupts on the TM4C microcontrollers are vectored, but there are some triggers that share the same vector. For these interrupts the ISR

must poll to see which trigger caused the interrupt. For example, all input pins on one GPIO port can trigger an interrupt, but the trigger flags share the same vector. So if multiple pins on one GPIO port are armed, the shared ISR must poll to determine which one(s) requested service.

The third necessary component of the interface is the **ability for the computer to acknowledge the interrupt.** Normally there is a trigger flag in the interface that is set on the busy to ready state transition. In essence, this trigger flag is the cause of the interrupt. Acknowledging the interrupt involves clearing this flag. It is important to shut off the request, so that the computer will not mistakenly request a second (and inappropriate) interrupt service for the same condition. Except for periodic SysTick interrupts, TM4C microcontrollers use software acknowledge. So when designing an interrupting interface, it will be important to know exactly what hardware condition sets the trigger flag (and request an interrupt) and how the software will clear it (acknowledge) in the ISR.

Common Error: The system will crash if the interrupt service routine doesn't either acknowledge or disarm the device requesting the interrupt.

Common Error: The ISR software should not disable interrupts at the beginning nor should it reenable interrupts at the end. Which interrupts are allowed to run is automatically controlled by the priority set in the NVIC.

Help