Lecture 5 — Interrupt Implementation

Jeff Zarnett & Mike Cooper-Stachowsky
jzarnett@uwaterloo.ca, mstachowsky@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 31, 2024

Credit: Based on material from Allyson Giannikouris

More About Interrupts

The previous topic discussed interrupts, both hardware and software.

We also learned about how to deal with them on the Cortex M4.

But to truly understand them, we need to look deeper into hardware.

Key Idea: No Polling

Avoid polling as much as possible: it's a waste of time!

Instead: CPU can do whatever else is ready…
And get alerted when something happens.

Sleeping

The CPU can be in a “sleeping” mode, but we’re ignoring that for this course.

Triggering an Interrupt

What can cause an interrupt?

- Clock signals
- Timers
- Peripherals (UART, SPI, I2C, GPIO, ADC...)

Interrupt Time

We covered 3 of the 4 items here last time:

1. Interrupt the CPU
2. Save the state of the current task (PC, SP, R0-R3, LR)
3. Handle the interrupt
4. Resume execution (unless it's hard fault or similar)

But we didn't cover how the hardware interrupt works!

How to get the CPU's Attention

It's a physical voltage signal (the power of electricity!).

How to get the CPU's Attention

Each interrupt has a dedicated wire – we call them lines.

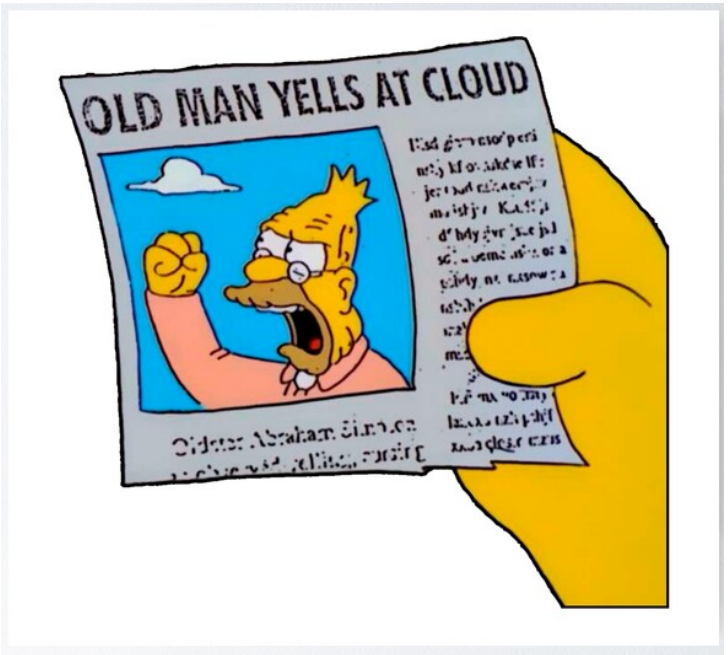
The line indicates the interrupt is ready.

The CPU has various interrupt registers to see if the interrupt is ready.

Signalling the processor is called an interrupt request (IRQ).

IRQ Configuration

Configuring the interrupts is a bit of a headache and at one time involved setting physical jumpers on cards to configure them.



But the idea is still a table of interrupts…

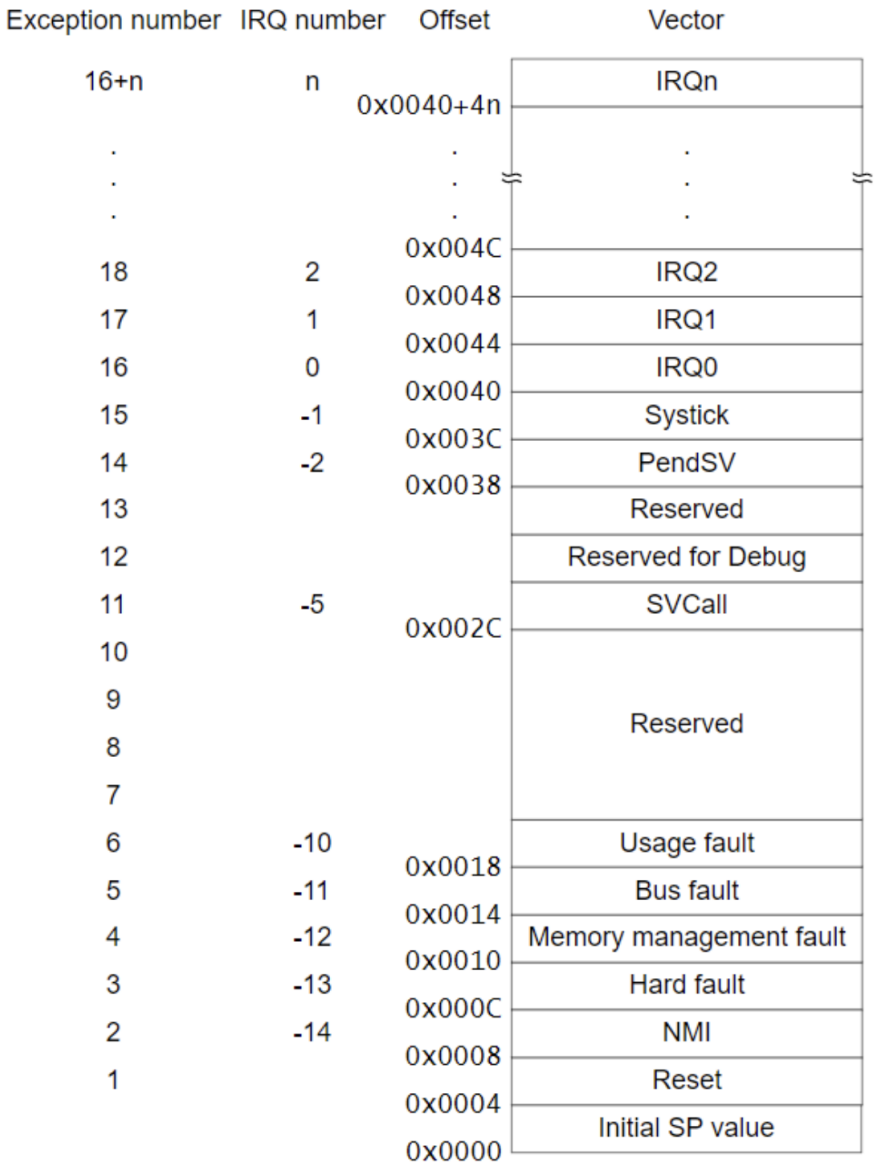
Interrupt Vector Table

We use an array of function pointers!

Each interrupt references a particular address in the array.

Cortex M's vector table starts at `0x0`.

Interrupt Vector Table



That's a Relief

Vector tables are configured in the setup code.

Good news: we almost never have to write it ourselves.

Pre-made code names the interrupt handler functions…
And the names are used to write your IRQ functions.

Example Config

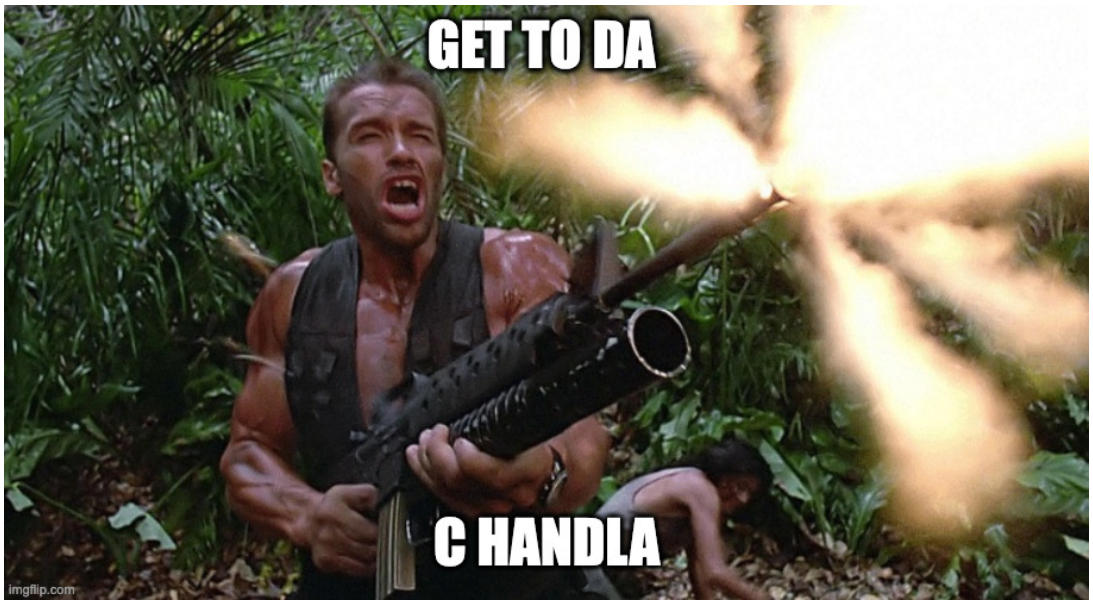
Setting up the vector table

```
    .section  .isr_vector,"a",%progbits
  .type  g_pfnVectors, %object
  .size  g_pfnVectors, .-g_pfnVectors

g_pfnVectors:
  .word  _estack
  .word  Reset_Handler
  .word  NMI_Handler
  .word  HardFault_Handler
  .word  MemManage_Handler
  .word  BusFault_Handler
  .word  UsageFault_Handler
  .word  0
  .word  0
  .word  0
  .word  0
  .word  SVC_Handler
  .word  DebugMon_Handler
  .word  0
  .word  PendSV_Handler
  .word  SysTick_Handler
```

A sample, pre-made Interrupt Service Routine (ISR)

```
.weak      NMI_Handler
.thumb_set NMI_Handler,Default_Handler

.weak      HardFault_Handler
.thumb_set HardFault_Handler,Default_Handler

.weak      MemManage_Handler
.thumb_set MemManage_Handler,Default_Handler

.weak      BusFault_Handler
.thumb_set BusFault_Handler,Default_Handler

.weak      UsageFault_Handler
.thumb_set UsageFault_Handler,Default_Handler

.weak      SVC_Handler
.thumb_set SVC_Handler,Default_Handler
```

Once Activated



This doesn’t apply to SVC, though.

Once Activated

The code below runs when the SysTick interrupt happens, only if enabled.

```
void SysTick_Handler() {
  // Do the thing
}
```

Is it magical auto-generated design? Sure is.

Gotta Go Fast

Four main reasons to keep Interrupt Subroutines (ISRs) short:

1. Want to return to normal operation quickly
2. Minimize the chance of other interrupts
3. Shorter functions = less code = less risk of bug
4. C is hard enough; Assembly is much harder

So Call Me Maybe?

It’s better to avoid calling functions in ISRs.

We’ll learn more about why this is complex in a future topic.

Functions called must be reentrant.

Non-Reentrant Code

Any function that accesses a global or static variable is non-reentrant.

```
int tmp;
void swap( int *x, int *y ) {
  tmp = *x;
  *x = *y;
  *y = tmp;
}
```

How would you fix this to make it reentrant?

All Done

To return from an interrupt there's a special instruction `rti`.
You cannot just do a regular `return` statement.

Let's think for a minute – why not?

Usually the IDE helps you out here.

Stop Talking While I'm Interrupting

We know a few ways of handling interrupts during another interrupt:

Ignore it, do it sequentially, switch to the new one?

But how to choose… by priority perhaps?

Interrupts are assigned a priority number; lower number is higher priority.

At least in the Cortex M4; other systems are different.

Stop Talking While I’m Interrupting

Some interrupts are always at a high priority and we can’t change that.
    Others are under our control!

We aren’t going to handle nested interrupts in our OS (too hard).

Maskable Interrupts

Some interrupts are maskable: can be ignored.

IRQs fall in this category. Usually indicate non-critical things.

Non-Maskable Interrupts

Other interrupts are non-maskable (NMI).

Cannot be turned off or ignored; for critical things.

These force the CPU to halt execution!

This is It

In the Cortex M4 there's only one NMI.

It's the one that gets called when other interrupt handlers fault.

It's higher priority than anything other than RESET.

Fault Priority Levels

Type	Exception Number	IRQ Number	Exception Type	Priority	Handled Using
processor core exceptions or internal interrupts	1	n/a	Reset	-3	
	2	-14	NMI	-2	
	3	-13	HardFault	-1	Fault handlers
	4	-12	MemManage	Configurable	
	5	-11	BusFault		
	6	-10	UsageFault		
	11	-5	SVCall		System handlers
	14	-2	PendSV		
	15	-1	SYSTic		
device-specific exceptions or external interrupts	16	0	IRQ		ISRs
	17	1			
	...	...			

IRQ States

Pending: The conditions for the interrupt are met, but ISR has not run.

Active: The CPU is running the ISR.

Pending and Active: The next one is ready but we aren't done with the last!

Inactive: Neither pending nor active.

Pending and Active

Pending and active can happen if we're not keeping up with the speed of the device somehow, or the device isn't keeping up with the CPU.

Sometimes it's also about luck and timing; events can happen very close together.

Global Interrupt Enable

There is a single setting to turn on/off all interrupts.

When “off”, if triggered, they go to the pending state.

When interrupts re-enabled, all pending ones evaluated by priority.

This does not turn off NMI.

Is this Easy or Hard?

When a hardware interrupt occurs, though, what does the ISR do?

Simple ISRs are easier to write but they can only do so much.

What if the interrupt handler just needs to tell a user program it can proceed?

Don’t forget to Like and Subscribe

This takes us down an interesting path…

Tasks (user processes) have state, more than just the current registers.

Try Again Later

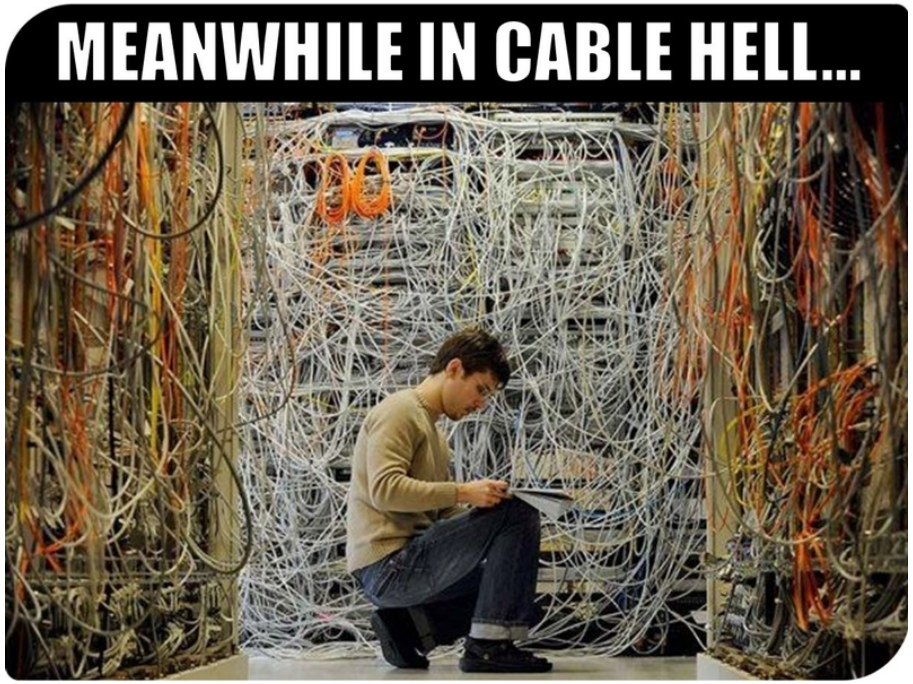
We'll have to return to the subject of how tasks (processes and threads) can be made to wait when they're not able to continue.

Just to be clear, what we're talking about is context-switching and multi-threading, but we're not ready for that yet.

It will come up soon and will be important in the labs.

On to Hardware Devices Then

Before we're ready to go on, we need to spend some more time on hardware.



Our next topic will focus on communicating with hardware devices!