Interrupts

Prof. John McLeod

ECE9047/9407, Winter 2021

Microcontrollers often have to send and receive data from a wide variety of peripherals and communications networks. Each of these operates at its own speed, which is often quite a bit slower than the microcontroller CPU. As the number of peripherals and communications networks grows, sequentially check each for completion becomes unmanageable. This is where interrupts come in: an interrupt enabled processor can "mind its own business" until a piece of hardware tells it that it is ready.

Polling & Interrupts

As we have seen, programming a microcontroller to interact with peripherals presents a challenge for sequentially-minded programmers: things can happen in the peripheral that are outside the microcontroller's control. For example:

- The microcontroller can tell a peripheral to start some process (a timer counting down, and ADC converting some channel), but it doesn't necessarily know when that task will be completed.
- A microcontroller can connect to an input source or a communications network, but it doesn't necessarily know when input will be received.

Thus far we have used **polling** to check these things.

Definition: **Polling** refers to periodically checking some status register of a peripheral to see when it is ready.

Polling works fine, but it becomes more difficult to manage as more and more peripherals that can generate **events** are added — especially if some of these peripherals generate events over a much longer interval than others, and especially if dealing with the events of some peripherals takes much longer than others. A more powerful approach is to use **interrupts**.

Definition: An **interrupt** is a notification sent to the microcontroller by a peripheral that requests the main program be temporarily halted while that peripheral's event is handled.

All computer systems have interrupts, but in most computer programming (above the level of an operating system) they are rarely used. However interrupts are rather important in microcontroller systems. An contemporary analogy used to describe **polling** and **interrupts** is waiting for a pizza to be delivered.

- You can **poll** for the pizza by looking out your door every 5 minutes.
- You can go about your business and wait for the delivery person to **interrupt** you by ringing the door bell.

Here the door bell is a piece of hardware that tells you to "bookmark" your current task, [1] deal with the pizza delivery, then return to your normal business. Both polling and interrupts work for pizza delivery, but polling makes it difficult to do anything else until the pizza is delivered. [2] As we will see, interrupts are conceptually fairly simple. Unfortunately, because they are intimately related to the hardware of the microcontroller, actually *implementing* an interrupt is very device-specific.

[1] Literally bookmark it, if you were reading a book!

[2] Sometimes polling is necessary: if instead of a pizza, you were waiting for a package to be delivered by UPS, the interrupt would never come. Eventually when you do check the status of the delivery, you'll see the familiar and aggrevating "sorry we missed you" note.

Interrupt Service Routines & Vectors

How does an interrupt work? Like magic! At least, it may seem that way to those of you who only have experience in sequential programming. An interrupt is realized through a combination of hardware and software: an **interrupt service routine** (ISR) is the software you write to tell the microcontroller *how* to handle the interrupt, and the hardware peripheral tells the microcontroller *when* to handle the interrupt.

Definition: An **interrupt service routine** is the subroutine that is called whenever the interrupt occurs. You (the programmer) are responsible for writing this code, and it is written more-or-less like any normal subroutine.

The ISR is the implementation of what happens to resolve the interrupt (i.e., store an ADC conversion to memory, transmit data over the communications channel, light up the emergency LED, etc.). However, because it is called from hardware, there are a few differences between an ISR and a normal software subroutine.

- The ISR could be called at any time, from any place in the main program. Consequently, ISRs can *never* accept parameters. Any data from the main program that is required by the ISR must be a **global variable** (for higher level programming languages) or (equivalently) stored at in a predetermined register or memory address (for Assembly).
- As the ISR is called from hardware, the actual source code for the ISR must be at a predefined location in memory so the

hardware knows where to find it (i.e., it must be accessed by a **direct branch** to a *fixed memory address*).

To help the hardware find the ISR, the microcontroller has a **interrupt vector**.

Definition: The **interrupt vector** is a microprocessor-defined region of memory that contains the ISR addresses for each kind of interrupt.

When a particular interrupt is received by the microcontroller, it knows to look at the corresponding memory cell in the **interrupt vector** to find the address to branch to for the ISR. This is the device-specific nature of interrupt programming.

- The **interrupt vector** needs to always be defined across the same range of addresses in memory. These addresses are device-specific, you need to look them up in the programmer's manual.
- Each hardware peripheral on the microcontroller that is capable of generating an interrupt is assigned a unique, fixed, interrupt code. This code corresponds to the place in memory (the "row" of the **interrupt vector**) where the ISR address is stored. These codes are device-specific, you need to look them up in the programmer's manual.
- Because all of these are fixed in memory, the end user cannot define any new interrupts in software.

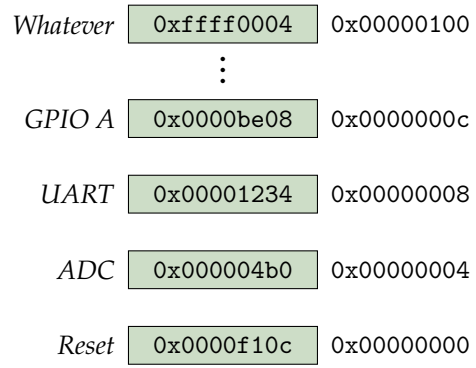

Figure 1: A representation of an **interrupt vector**. Each hardware peripheral capable of sending an interrupt (listed on the left) is assigned a fixed address in memory (on the right). The user-defined ISR is located at the user- or compiler-defined address stored in the memory cell (in the green box).

Interrupts & Context

A peripheral can trigger an interrupt at any arbitrary point in the main program. The "magic" part of interrupts is that the ISR is never *explicitly* called by software. As a programmer, all you need to do is:

- Write the microcontroller-specific code to enable interrupts and initialize the **interrupt vector**.
- Write the appropriate code for each ISR that will handle the interrupt.
- Write the main program.

You never explicitly call the ISR — assuming the **interrupt vector** is properly set up, the microcontroller will know how to branch to the ISR when an interrupt is triggered.
Because you have no control over *when* the main program will branch to the ISR, it is important to write the ISR code appropriately.

- If the ISR needs to use some registers for local variables, you should `push` the original contents of those registers to the stack at the start of the ISR, and `pop` the original contents from the stack at the end.
- Any data returned by the ISR should be stored at some predefined location in memory, or some particular register that

you are reserving for that purpose. You can't return data using the stack or some general register, as the main program does not know when to check for that return data after the ISR is complete.

As previously mentioned, the conventional protocol for writing a subroutine involves preserving the **state** of the system, by pushing general purpose registers and the link register to the stack when that subroutine is called, and by popping them back when the subroutine is complete. An ISR should do this as well, but go one step further: it should also preserve **context**.

Definition: The **context** of a program is the **state** of the general purpose registers, the stack, and the link register, as well as the current contents of the **program status register** (especially the comparison flags).

Context is important because an ISR may need to use comparison flags for conditional execution, but the ISR may have been called at a line the main code where comparison flags must be retained — as illustrated in Figure 2.
Normally the `cpsr` is "off limits" to software — you can't change the values in the `cpsr` the same way you could in a general-purpose register. Consequently, the way **context** is preserved in an ISR usually involves special commands, that are microcontroller- and language-dependent.

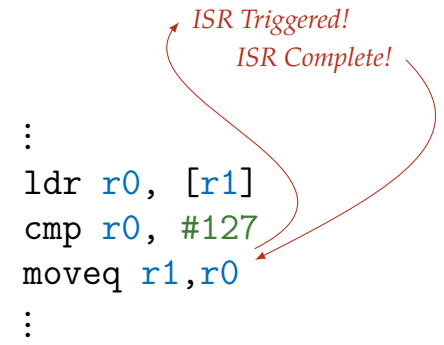

Figure 2: Example showing the importance of preserving **context**: Here the ISR is called between a comparison statement and a conditional execution statement. If the ISR itself uses any comparisons (as they often do), it is important that the original comparison flags are restored after the ISR is completed or the following conditional execution may behave unexpectedly.

Example: For an ARMv7-type microcontroller, the `cpsr` is automatically saved when an ISR is called, and `pc` is automatically written to `lr`. To restore context and exit the ISR, use the special command `subs pc, lr, #4` rather than the conventional `bx lr` that you would use for a normal subroutine.

- We need to subtract 4 from the address in `lr` because the ISR wasn't actually reached from an explicit branch instruction — so we need to "roll back" one step when returning to the main program.
- Furthermore, this particular instruction carries the hidden meaning to *also* restore the previously-saved state of the `cpsr`.

So when entering a ISR, all you need to do is push the appropriate registers to the stack to save the **state**. After the ISR is complete, pop those registers back off the stack, then exit the ISR with `subs pc, lr, #4` to return to the main program with the same **context**.

Interrupt Enable/Disable

As we have alluded to previously (in particular, when discussing the **control register** for the interval timer), interrupts can be enabled or disabled. As you may have guessed, most microcontrollers have interrupts disabled by default.

- So far, we have written a bunch of short programs and never once worried about interrupts.
- In fact, when saving the instructions from these programs to memory when the program was compiled, we actually wrote over the hard-coded memory address for the interrupt vector!
- Because the system was ignoring interrupts, this wasn't a problem.

For most microcontrollers, enabling interrupts is often a three step process.

1. A global flag in the **program status register** must be sent to enable (or disable) all interrupts, system-wide. As the **program status register** is protected from normal instructions, this requires special microcontroller- and language-dependent code.
2. Additional flags in memory, and/or the **interrupt vector** need to be configured to enable interrupts from individual peripherals.

3. Often an additional flag to enable interrupts must be set in that peripheral's **control register**.

So you need to tell the microcontroller that interrupts are allowed, tell the microcontroller what kind of interrupts to expect, and finally tell each of those devices that they are allowed to interrupt. If any one of these steps are omitted, an interrupt will not trigger an ISR.

Example: Many microcontrollers will automatically *disable* the global flag for interrupts when executing an ISR — i.e., they will not allow *nested* interrupts. The interrupt flag will be restored after the ISR is complete. This process occurs in the ARM; it cannot handle nested interrupts.

Interrupt-Driven Programming

For embedded systems that primarily are responsible for processing events generated by peripherals, using interrupts allows the main program to be deceptively — even confusingly — simple.

- Often the main program is just a dead loop, with the CPU in a low-power, idle state.

Those of you who are used to sequential programming and are uncomfortable with endless loops will no doubt be even *more* uncomfortable with an enedless loop that does nothing, but that is how many microcontrollers are coded. A whole bunch of ISRs for each kind of event were written, but they are never explicitly called in software.

- As each event triggers an interrupt, the appropriate ISR is called which deals with the interrupt, then returns to the main program's dead loop.

Conceptually, that is all there is to interrupts. As long as you accept that hardware can call a software subroutine, *interrupt-driven programming* is just a variant of *event-driven programming* where **waits** are never explicitly coded, as the hardware interrupt will trigger the **process** phase for each task.
Unfortunately, for various reasons, the ARMv7 has a rather complicated implementation of interrupts, as we will now discuss.

Interrupts & ARM Processors

Compared to many other microcontrollers, the ARM **interrupt vector** is relatively short. ARM actually calls it the **exception vector table** rather than **interrupt vector**, and it has the structure shown in Figure 3. Apparently, ARM reserves the word "interrupt" only for *peripheral-driven interrupts* (as discussed above) or *software-requested interrupts* (which we have not, and will not, talk about). Other kinds of *internal interrupts*, like how to handle restarting the microcontroller, or what to do if an instruction unexpectedly fails to execute are considered "exceptions". Furthermore, ARM systems store *instructions* rather than *addresses* in these vectors.

- The first row in the exception vector is what to do when the microcontroller is first powered on. Usually this row just directs the processor to branch to the start of the main program.
- The second row contains the instruction for what to do if it attempts to execute an an undefined opcode.
- The third row contains the instruction for handling software interrupts.
- The fourth row contains the instruction for what to do if an instruction unexpectedly aborts during execution.
- The fifth row contains the instruction for what to do if reading or writing data unexpectedly aborts.

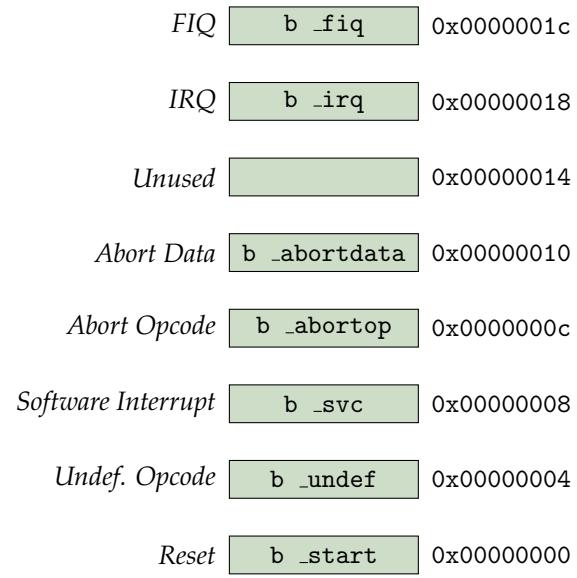

Figure 3: The ARM **exception vector table**. In contrast to the generic interrupt vector shown in Figure 1, the ARM exception vector table contains instructions rather than addresses. However as the instruction is typically a direct branch (as shown here), this has the same effect.

- The sixth row is undefined, for reasons best known to the mysterious and inscrutable ARM engineers.
- The seventh row is for normal peripheral-based interrupts, called "interrupt requests" (IRQs).
- The eighth row is for a special kind of prioritized "fast" peripheral-based interrupts, called "fast interrupt requests" (FIQs).

Here we will only discuss normal peripheral-based interrupts. The main difference between interrupts in an ARM compared to other microcontrollers is that *all* peripheral-based interrupts are handled by the same row in the **interrupt vector table**, which needs to call a special ISR that then figures out *which* peripheral-interrupt occurred. [3] Each peripheral that is capable of generating an interrupt has a unique IRQ code, these are written on the peripheral's window in the online simulator.

[3] In most other microcontrollers each interrupt-capable peripheral is associated with its own row in the table, which branches directly to that peripheral's ISR.

Example: The push buttons and the interval timers can generate interrupts. A push button-interrupt is distinguished from a timer-interrupt in software because the former generates an IRQ code #73, while the latter generates code #72. The switches can not generate interrupts — they do not have an IRQ code.

This mechanism is called the "generic interrupt controller" (GIC), and was implemented so that every ARM-based microcontroller has the same interrupt structure, even though different microcontrollers will have different peripherals.

ARMv7 Operating Modes

Before discussing how to configure the GIC on an ARM, we should first discuss the different **operating modes**. The current operating mode of the ARM is indicated by the five LSbs in the `cpsr`, the connection between these bit codes and the operating mode is shown in Table 1.

- By default the ARM starts in *supervisor* mode. As we have, thus far, never learned how to change the operating mode, all programs we have written in this course have been executed in *supervisor* mode.
- *User* mode is an operating mode with reduced access to Assembly language instructions. This is designed to run programs within a pre-written framework or operating system. The system can switch from *user* mode back to *supervisor* mode when a software interrupt is triggered.
- The *abort*, *underfined opcode*, and *system* modes all used to process non-peripheral- and non-software-type interrupts.
- The *IRQ* and *FIQ* modes are for processing regular and fast peripheral-type interrupts.

Bit Code	Mode
`0b10000`	*User*
`0b10001`	*FIQ*
`0b10010`	*IRQ*
`0b10011`	*Supervisor*
`0b10111`	*Abort*
`0b11011`	*Undefined Opcode*
`0b11111`	*System*

Table 1: Operating modes in the ARM®Cortex-A9.

In this course we will only consider *supervisor* and *IRQ* modes. In order to change the **operating mode**, we need to modify the `cpsr`. This can be done with a special mnemonic.

Mnemonic: The mnemonic `msr` can be used to move values between a general-purpose register and the `cpsr` or the "saved program status register" `spsr`.

A full `msr` instruction is written similarly to a `mov` instruction:

```
msr cpsr, r0   @move r0 into cpsr
msr r1, spsr   @move spsr into r1
```

As usual, condition codes can be appended to the mnemonic. However because the `cpsr` and `spsr` aren't really data registers, but rather a collection of **control bits**, they have the unique ability to allow only partial access by appending a **field code**.

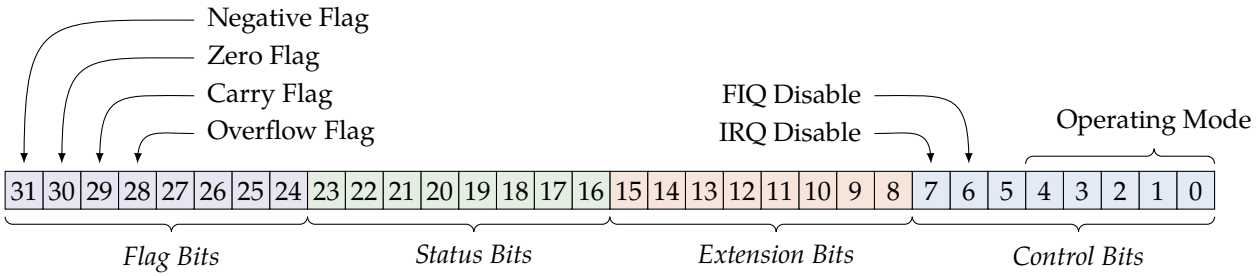

Figure 4: Structure of the ARMv7 `cpsr` and `spsr`, with some notable bits highlighted.

- Writing `cpsr_c` (or `spsr_c`) will only allow access to the *control bits*, which can set the operating mode and enable or disable interrupts. [4]
- Writing `cpsr_x` or `cpsr_s` (or `spsr_x`, `spsr_s`) will only allow access to the *extension* or *status bits*, respectively. These will not be discussed here.

[4] The remaining bit (bit 5) is used to control whether the ARMv7 or the Thumb instruction set is used for programming.

- Writing `cpsr_f` (or `spsr_f`) will only allow access to the *flag bits*, which include the conditional `NZCV` flags.

The structure of an ARMv7 `cpsr` (or `spsr`) is shown in Figure 4. The IRQ and FIQ bits are named "disable bits", because when they are set (i.e., bit 7 = 1) then interrupts (or fast interrupts) are disabled. These bits must be cleared to zero to enable that kind of interrupt. Therefore, to change to *user mode* and enable IRQ-type interrupts, but leave FIQ-type interrupts disabled, use the mnemonic:

```
msr cpsr_c, #0b01010000
```

This will change the mode and enable IRQ interrupts without changing any other bits in the `cpsr`. [5]
The **operating mode** will also change automatically when resolving any event that triggers something on the **exception vector table**. Therefore, all ISRs will be automatically executed in *IRQ mode*.

- As we will see later, using `msr` to manually change the operating mode is only necessary at the start of your program when you are configuring interrupts.

- There isn't much difference between running code in the various operating modes from a user's perspective, and certainly the programmer writes the code in largely the same way, but from a hardware perspective there is a difference.

The hardware implementation of the different operating modes is part of the reason why ARM microcontrollers are so efficient at executing comparatively complex programs.

[5] As you can see when you open the simulator, by default all of the *status* and *extension bits* are zeros. These will remain zeros unless you go well beyond the scope of the course in your ARMv7 programming. So if you don't care about preserving the conditional flags, you can just use `msr cpsr #0b01010000`.

Banked Registers

We previously mentioned that the ARM®Cortex-A9 has thirteen general-purpose registers (`r0` to `r12`), a stack pointer (`sp`), a link register (`lr`), a program counter (`pc`), and a current program status register (`cpsr`).

- We also recently mentioned the saved program status register (`spsr`), in fact this is what is used by the CPU to store the **context** when jumping to an ISR.

The CPU actually has a lot more registers than this: the extra ones are **banked** or *redundant* copies that are only available in the different **operating modes**.

- Each of the *user*, *supervisor*, *abort*, *undefined opcode*, *IRQ*, and *FIQ* modes has its own separate `spsr`, `sp`, and `lr` registers.
- The *FIQ* mode additionally has separate general-purpose registers `r8` to `r12`.
- The *system* mode allows access to all of the registers across every operating mode. The *system* operating mode is only accessed under very particular, and rather extreme, circumstances.

So, for example, on an ARMv7-system if you are coding an ISR that accesses another subroutine, you don't need to worry about accidentally overwriting the `lr` if the ISR was triggered during a subroutine call in the main program. The link register used by the

ISR is a *physically different register on the CPU chipset* than the one used in the main program. Similarly, because *IRQ* mode has its own stack pointer, the ISR stack could be completely distinct from the main program stack.

- This is one of the reasons why FIQs are "fast": as *FIQ* mode has some of its own general purpose registers, these can be used by the "fast-ISR" immediately. Whatever the main program may have stored in `r10` before the fast-ISR was called is still there, as the fast-ISR will use a physically different piece of hardware as `r10`.

In what follows we will discuss the technical details of how interrupts are configured on an ARMv7-type microcontroller. The above discussion of **banked registers** is really only important to the extent that it helps clarify how **context** is preserved when executing an ISR. For example, there is no need to push the contents of `lr` to the stack at the start of an ISR, because the ISR uses a physically different `lr` than the main program.

The ARMv7 Generic Interrupt Controller

The general approach to programming with interrupts is:

1. Write the ISR for each type of hardware interrupt, assigned to some label.

2. Set up the **interrupt vector** to identify the label of the ISR with the appropriate interrupt signal — recall, each interrupt signal has a different row in the interrupt vector.

3. Set all the appropriate control bits to enable interrupts. Recall that typically there is a global control bit that enables the CPU to receive interrupts, then peripheral-specific control bits that tell the CPU which interrupts to expect, and finally bits in the peripheral's **control register** that allow it to send interrupts.

The above presents a sensible order for implementing these steps. However in the actual source code, *sequentially* the **interrupt vector** typically comes first in the code, as it *must* be at a hardware-defined position in memory (in the case of the ARM, starting at `0x00000000`), then the interrupts are configured in the first part of the main program, and finally the ISR is written as a subroutine, typically near the end of the source code.
The above steps are still followed for an ARMv7-system, but because of the generic interrupt controller (GIC) there are a few extra steps. In an ARMv7-system, you should:

1. Write the ISR for each type of hardware interrupt, assigned to some label.

2. Write the **IRQ service routine** that handles all hardware interrupts. This should check the IRQ code that triggered the interrupt, then branch to the appropriate ISR.

3. Set up the **exception vector table**. This is generally always the same for every program. The 7th row of this table (the row for IRQs, refer back to Figure 3) should provide the branch instruction to the **IRQ service routine**.

4. Set all the appropriate control bits to enable interrupts. In addition to the ones described above, there are also several special registers in the GIC that need to be configured. Typically a special subroutine for configuring the GIC is written.

The process isn't really that complicated — at least conceptually — but there are a ton of special-purpose registers that need to be configured in the GIC, and each has its own long and very-similar-looking acronym as its name, so it is easy to get confused. To begin with, the **exception vector table** can be defined as follows (remember that an ARM interrupt vector requires an *instruction* rather than just an *address*, hence the vector is a list of direct branches):

```
.section .vectors, "ax"
  @reset vector
  b _start
  @undefined instruction vector
  b _service_und
  @software interrupt vector
  b _service_svc
  @aborted prefetch vector
```

```
  b _service_abt_inst
  @aborted data vector
  b _service_abt_data
  @unused vector
  .word 0
  @IRQ interrupt vector
  b _service_IRQ
  @FIQ interrupt vector
  b _service_FIQ
```

Here the directive `.vectors` implies that a sequential list of data is about to be given, and the code `"ax"` notifies the compiler that it is *allocatable* and *executable*. This code must be the first non-comment code in your file, so that it gets written to memory starting at `0x00000000`.

All of the service routines listed above must be written in your program. The reset vector typically branches to the main program, as that is what you generally want the microcontroller to do when it is first powered on. Many of the other possible exceptions will not — or at least *should not* — occur in your program, so those service routines can simply be dead loops. [6] For example:

```
/*Aborted data fetch service routine*/
_service_abt_data:
  @this should never happen and if it DOES
  @somehow happen, then we have BIG problems
  @so just loop until someone shows up to debug
  b _service_abt_data
```

[6] Or the routine could branch back to the start of the main program, but that is usually not recommended: restarting the main program doesn't wipe the contents of registers or RAM the way a full reset does, to the problem that led to an unexpected exception may still be present.

Additionally, there are a few other things need when setting up to enable interrupts in an ARMv7 system:

- You should to set the stack pointer `sp` for *IRQ mode*. As this is a **banked register**, it should point somewhere entirely different in memory than the regular `sp` for *supervisor mode* (or *user mode*). When the ARM is initially turned on, all `sp` in every mode are initialized to `0x00000000`: if they are left this way, then if the stack is used during an ISR it will overwrite the main program stack.
- You need to make sure all interrupts are disabled until the interrupt configuration is complete.

An example of these steps is shown below, this code would typically go at the start of the main program (usually, right after `_start:`), which is written immediately after setting up the **exception vector table** as described above.

```
_start:
@interrupts masked, MODE = IRQ
mov r1, #0b11010010
msr cpsr_c, r1 @change to IRQ mode
@set IRQ stack to A9 onchip memory
ldr sp, =0xfffffffc
@interrupts masked, MODE = SVC
mov r1, #0b11010011
msr cpsr_c, r1 @change to supervisor mode
@set main program stack
ldr sp, =0x3ffffffc
```

Notice how sp is set differently, and in immediate succession, after changing the modes. Remember that sp is a **banked register**, so a physically different piece of hardware is used in each mode. Consequently, the stack at 0xfffffffc is used for all ISRs, while the stack at 0x3ffffffc is used for the main program. As long as each stack does not get ridiculously large, they will not overlap. [7]
Now the program can branch to a subroutine to set up the GIC, then another subroutine to enable interrupts on all the desired peripherals, and finally the global interrupt flag in the cpsr can be turn on:

```
@configure the ARM GIC
bl config_gic

@configure interrupts on peripherals
bl config_peripheral

@enable IRQ in CPU
@interrupts unmasked, MODE = SVC
mov r0, #0b01010011
msr cpsr_c, r0

/*now rest of main program goes below*/
```

In hardware, an interrupt signal enters the GIC distributor, and gets routed to to the CPU through the GIC CPU interface. To set up the GIC (i.e., in config_gic:), both of these need to be configured.

[7] Of course those of you familiar with C programming will know that a **stack overflow** can always occur if you are not careful!

The GIC Distributor Interface

Are human beings innately good or evil? This is the eternal question that has been contemplated by many famous philosphers. I, for one, cannot say whether humanity has a general inclination towards good or evil, but I can say this:

- *The people who designed the ARM GIC distributor are unredeemably evil. Ask not what lies in their soul, for it is twisted and corrupted, and they seek to spread this corruption across the world in the form of the ARM GIC distributor interface.*

The GIC distributor is a cursed wasteland that we must traverse, so take a deep breath.

- The distributor is a very large structure, which starts at `0xfffed000`.
- The distributor has a lot of different registers that need to be configured, each of which has a long acronym.
- The basics aren't actually that difficult, but the as you may have guessed from my above theatrics, it does seem a bit unnecessarily obtuse.

There are six groups of registers in the GIC distributor. As mentioned above, every hardware peripheral has an IRQ code x. [8] To enable interrupts for a peripheral with a given IRQ code, a number of bits and bytes in these registers needs to be set.

[8] The push buttons are IRQ #73, one of the interval timers is IRQ #72, one of the GPIO parallel ports is IRQ #11, etc.

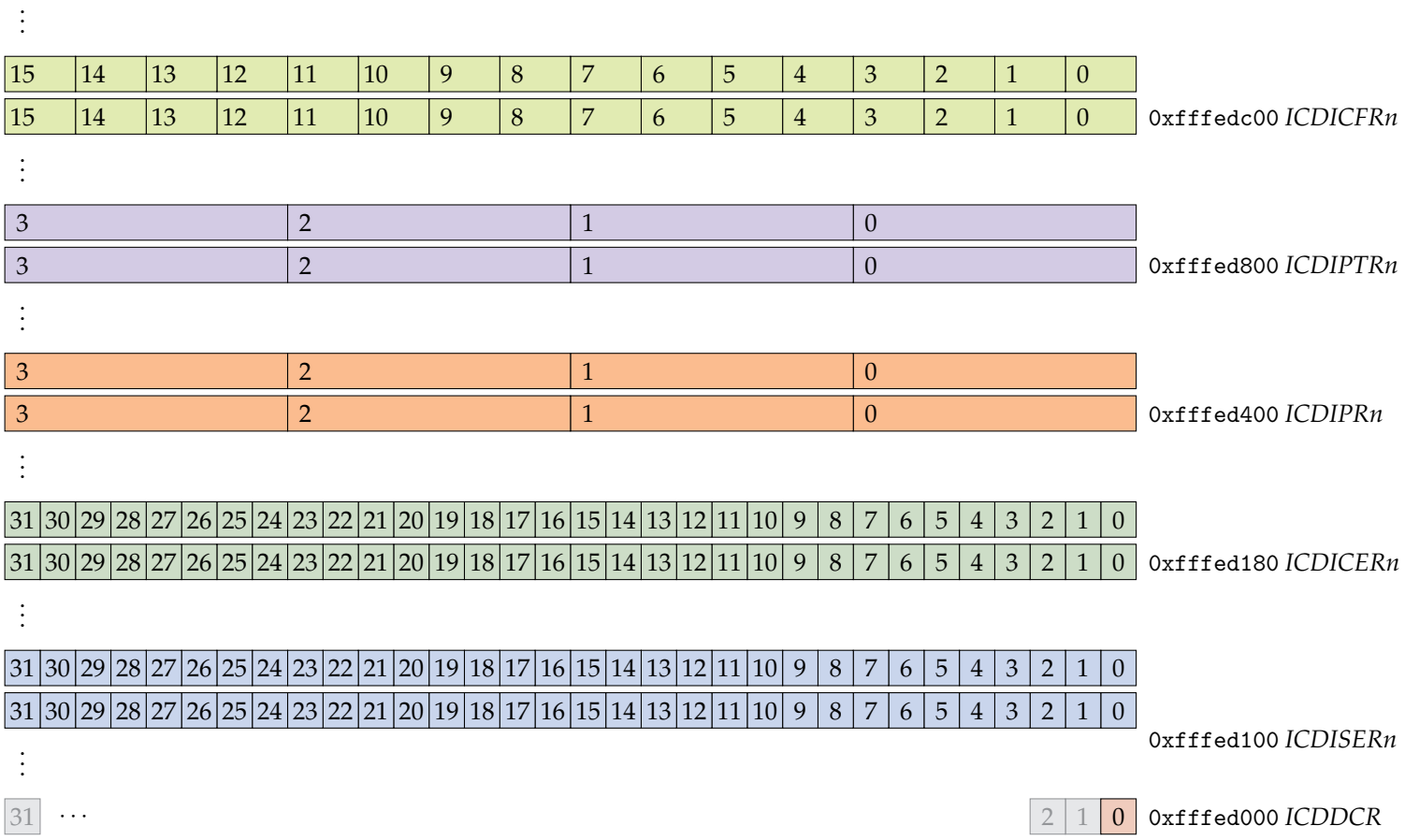

Figure 5: Structure of the GIC distributor registers. Every register with "*n*" at the end of its name is repeated until there is enough space for all IRQ peripherals.

A simplified schematic of the GIC distributor is shown in Figure 5. Many of the registers in the GIC distributor are repeated multiple times to ensure there are enough control bits or control bytes for all IRQ peripherals. This leads to three different formulae for determining which bits to set for a given IRQ x in a given GIC distributor register n.

- If each IRQ peripheral has a **single bit** in that set of registers, then: [9]

$$\begin{aligned} \text{Register } n &= \texttt{[base]} + 4 \times \texttt{int}\left(\frac{x}{32}\right), \\ \text{Bit } b &= x \pmod{32}. \end{aligned} \tag{1}$$

- If each IRQ peripheral has a **pair of bits** in that set of registers, then:

$$\begin{aligned} \text{Register } n &= \texttt{[base]} + 4 \times \texttt{int}\left(\frac{x}{16}\right), \\ \text{Bits } b[1,0] &= [1,0] + 2 \times (x \pmod{16}). \end{aligned} \tag{2}$$

- If each IRQ peripheral has a **byte** in that set of registers, then:

$$\begin{aligned} \text{Register } n &= \texttt{[base]} + 4 \times \texttt{int}\left(\frac{x}{4}\right), \\ \text{Bits } b[7..0] &= [7..0] + 8 \times (x \pmod{4}). \end{aligned} \tag{3}$$

So, with that in mind, to configure the distributor follow these steps for each IRQ peripheral.

1. Enable that peripheral in the appropriate *interrupt set-enable register* (*ICDISERn*). These registers have the base address `0xfffed100`. Each IRQ x has a **single bit** in this set of registers, so find the appropriate register and bit using Formula 1. These registers allow read and write, but *only* recognize a write of 1, which enables that peripheral.

[9] Here $x \pmod{32}$ means to divide x by 32 and only keep the remainder.

2. You can set interrupts with different **priority** if you want. If two interrupts occur simultaneously, the one with higher **priority** is guaranteed to be resolved first. Setting a high priority is done by providing a lower value to the appropriate *interrupt priority register* (*ICDIPRn*), so the highest priority corresponds to a value of zero. These registers have the base address `0xfffed400`. Each IRQ x has a **byte** in this set of registers, so find the appropriate register and bits using Equation 3. If you don't care about prioritizing different interrupts you can skip this step. [10]

3. An ARM®Cortex-A9 microprocessor has at least 2 CPU cores, and can have up to 8. You need to specify which of these cores should be targeted for resolving the interrupt by providing a bit pattern to the appropriate *interrupt processor target registers* (*ICDIPTRn*). These registers have the base address `0xfffed800`. Each IRQ x has a **byte** in this set of registers, so find the appropriate register and bits using Equation 3. The byte should be considered a bit pattern: setting a 1 in bit b indicates that processor b can handle the interrupt. Therefore writing `0b00000011` to the appropriate byte in the *ICDIPTRn* will allow both processors 0 and 1 to handle the interrupt.

4. The interrupt signal can be specified as edge- or level-sensitive by configuring the appropriate bit in the *interrupt configuration register* (*ICDICFRn*). These registers have the base address `0xfffedc00`. Each IRQ x has a **pair of bits** in this register, so use Equation 2 to find the appropriate register and bits. For whatever strange reason, only the MSb of this pair can be

[10] As a default the priority values are zero — meaning they all interrupts are equally ranked as the highest priority.

used. Setting it to 1 makes the interrupt signal edge-sensitive, while setting it to 0 makes the interrupt signal level-sensitive.

5. Finally, after all peripherals are configured, the GIC distributor itself should enabled by writing 1 to the *distributor control register* (*ICDDCR*) at address `0xfffed000`. Only the LSb of this register is used to enable/disable the GIC distributor.

Unless you are writing a program that uses a vast number of interrupts, in which scheduling and priority is very important, it is likely you will configure each interrupt simply by enabling the appropriate *ICDISERn*, and allow the interrupt to use either of the two cores by writing `0b00000011` to the appropriate *ICDIPTRn*. You may, however, want to consider whether to set the interrupt signal to edge- or level-sensitive.

- If the event that triggers an interrupt could be continuous (like holding down a button), setting that signal to edge-sensitive can prevent continuous interrupts from being called (unless that is what you want, of course).

- If a level-sensitive interrupt event occurs while a previous interrupt ISR is still being resolved, the level-sensitive interrupt can "disappear" (or *deasserted*) if the event is released before the current ISR is resolved. An edge-sensitive interrupt does not deassert, and remains in the queue until it is resolved by the appropriate ISR.

- If, for example, the user presses a button *very* quickly while the microcontroller is handling a complicated UART-driven

interrupt, if the button is configured as a level-sensitive interrupt then the interrupt signal could be gone by the time the microcontroller returns to regular operation.

As an aside, if you want to disable a previously-enabled peripheral at some point in your program, there is an extra hoop to jump through. As mentioned above, the "set-enable" *ICDISERn* register only accepts a write of 1 to enable the peripheral. It is not possible to clear this bit by any direct `str` operation. Instead:

6. To disable a peripheral you have to write to a *interrupt clear-enable register* (*ICDICERn*), at base address `0xfffed180`. Writing a 1 to the appropriate bit (again, found by Equation 1) will turn off interrupts for that IRQ. [11]

No doubt you are all 100% crystal clear on what to do. But just in case, let's look at a quick example.

Example: You need a timer that is interrupt-enabled for your program. As you know, there are two *interval timers* in the ARM microcontroller. But you can't use those — they are haunted. After searching through the ancient scrolls of documentation, you find there are some extra timers on the development board, called *HPS Timers*. [12] One of these is IRQ #199 — a most auspicious number! How can we configure the GIC distributor for this peripheral?

- First we need to enable this IRQ with the appropriate *ICDISERn* register. The enable is a single bit — it will be

[11] Somehow having separate enable/disable registers provides extra security for the interrupt interface. I don't really understand why, and I don't care to find out.

[12] This is true, but as far as I can tell they are not implemented in the simulator, so it would be rather hard to test your code.

the 199th bit after `0xfffed100`. Using Equation 1 gives us:

$$n = \texttt{0xfffed100} + 4 \times \texttt{int}\left(\frac{199}{32}\right) = \texttt{0xfffed118}$$
$$b = 199 \pmod{32} = 7.$$

Therefore we need to set bit 7 at address `0xfffed118`. We should use a bit mask to avoid interfering with any other peripherals.

```
@ base address of ICDISERn
ldr r0, =0xfffed100
@ load current value with offset
ldr r1, [r0, #6]
@ apply mask to set bit 7
mov r2, #1
orr r1, r2, lsl #7
@ write back to ICDISERn
str r1, [r0, #6]
```

- We can skip setting the priority, but now we will set target CPU core 0 as the target CPU for this IRQ. The *ICDIPTRn* registers use a byte for each peripheral, so we need to fing the 199th byte after `0xfffed800`. Using Equation 3 gives us:

$$n = \texttt{0xfffed800} + 4 \times \texttt{int}\left(\frac{199}{4}\right) = \texttt{0xfffed8c4}$$
$$b = 8 \times (199 \pmod{4}) = 24.$$

Therefore we need to set bits 24 to 31 at address `0xfffed8c4`.

```
@ base address of ICDIPTRn
ldr r0, =0xfffed800
@ load current value with offset
ldr r1, [r0, #196]
@ apply mask to set last bits
mov r2, #1
orr r1, r2, lsl #24
@ write back to ICDIPTRn
str r1, [r0, #196]
```

Actually, because memory is byte-addressible, we could directly write to that byte: `strb r2, [r0]` where the address in `r0` is `0xfffed800` $+199_{10}$ = `0xfffed8c7`.

- I don't care whether the timer is level- or edge-sensitive, so I won't configure the *ICDICFRn*. By default, it will be level-sensitive. Now, assuming we are done configuring each peripheral in the GIC distributor, we can enable the distributor by writing 1 to `0xfffed000`.

```
@ base address of ICDDCR
ldr r0, =0xfffed000
@enable distributor
mov r1, #1
str r1, [r0]
```

The GIC CPU Interface

Setting up the GIC distributor was a major headache, no? Fortunately we are *almost* done — now we just have to configure the GIC CPU interface. Fortunately, this is a less-complicated structure and only needs to be configured once for all interrupts.

- The GIC CPU interface is important because not only does it have to be configured to enable interrupts, it also has to be checked when an interrupt is triggered to figure out which ISR should be called.

The structure of the GIC CPU interface is shown in Figure 6, the base address is `0xfffec100`. There are only 4 registers that apply for all peripherals, so there is no need for fancy formulas to calculate addresses based on IRQ numbers.

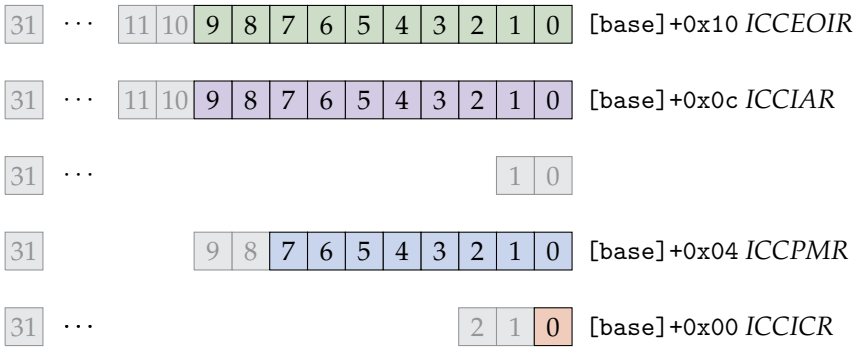

Figure 6: Structure of the GIC CPU interface registers. The base address is `0xfffec100`. For some reason there is an unused register at `0xfffec108`.

Configuring the GIC CPU interface is, fortunately, reasonably simple and can probably be done exactly the same way for every program you would want to write.

1. The *interrupt priority mask register* (*ICCPMR*) must be set. This is a global control used for testing interrupts: only interrupts with higher priority (meaning a *lower value* in the *ICDIPRn* register) than the mask given here will ever get called. Unless you are debugging, you probably want all interrupts to get called, so write `0xff` to fill the lowest 8 bits (the only ones that are used) with 1s.

2. The *CPU interface control register* (*ICCICR*) must be set to enable the GIC CPU interface. Just write 1 to the LSb in this register.

That's all we need to do to turn on the CPU interface!

```
@base address of CPU interface
ldr r0, =0xfffec100
@ make sure all priorities are enabled
ldr r1, =0xff
str r1, [r0, #4]
@ enable CPU interface
mov r1, #1
str r1, [r0]
```

As mentioned above, you are not done with the GIC CPU interface yet. In the **exception vector table** you defined a IRQ interrupt vector (in my example code on page 21 I called it `_service_IRQ`). This subroutine will be called whenever an IRQ peripheral is triggered. We need to use the GIC CPU interface to figure out *which* actual hardware peripheral triggered the interrupt.

- The IRQ number of the peripheral that triggered the interrupt is stored in the *interrupt acknowledge register* (*ICCIAR*). The IRQ interrupt vector subroutine should read from this register to determine which ISR to call to resolve the interrupt.

- After the ISR is complete, you need to tell the GIC CPU interface that the interrupt has been resolved. This is done by writing the same IRQ number to the *end of interrupt register* (*ICCEOIR*).

An example of an IRQ interrupt vector subroutine is given below. This program only expects interrupts from an interval timer or the push buttons. Note the use of `subs pc, lr, #4` to return from this subroutine — as discussed above, this is the correct way to exit *IRQ mode*, restore the `cpsr`, and return to the main program after resolving an interrupt.

```
_service_IRQ:
  @ push context
  push {r0, r1}  @etc...

  @ CPU interface address
  ldr r0, =0xfffec100

  @ which IRQ?
  ldr r1, [r0, #12]

  @ interval timer?
IRQ_is_timer:
```

```
  cmp r1, #72
  bne IRQ_is_push
  beq timer_ISR

  @ push button?
IRQ_is_push:
  cmp r1, #73
  bne IRQ_unknown
  beq push_button_ISR

  @something is wrong
IRQ_unknown:
  b IRQ_unknown

@ branch back here when done ISR
exit_IRQ:
  @ write end of interrupt
  str r1, [r0, #16]

  @restore context
  pop {r0, r1} @etc...

  @back to main program
  subs pc, lr, #4
```

Enabling Interrupts on the Peripherals

So we now know how to set up the **exception vector table**, create separate stacks for ISRs and the main program, enable interrupts and change the operating mode in the `cpsr`, and configure the GIC. Are we done yet?
Not quite: we have set up the ARM®Cortex-A9 to *recieve* interrupts, but we still need to tell each piece of hardware that it is allowed to *transmit* an interrupt. Generally this is done by configuring a **control register**.

- For some peripherals, such as the *interval timers* or the *JTAG UART*, we already identified one or more bits in the **control register** as being used to enable interrupts.
- For other peripherals, mainly those based on GPIO ports, an additional **control register** at `[base]+0x08` provides interrupt enable flags. For a GPIO port, each bit enables/disables interrupts for the corresponding pin. [13]
- The push buttons are similar to a GPIO port: bits 0 to 3 at `[base]+0x08` enable (set to 1) or disable (set to 0) interrupts from the corresponding push button.
- For any other interrupt-enabled peripheral not discussed here, you may need to look in a user's manual to find how to enable/disable interrupts. [14]

[13] Recall that the GPIO **data register** is at `[base]`, and the input/output direction **control register** is at `[base]+0x04`.

[14] Note that several peripherals on the DE1-SoC board are not capable of generating interrupts, as they are not mapped to a IRQ number. These include the switches and the ADC — hardware peripherals which may be capable of generating interrupt in other microcontroller systems.

ARMv7 Interrupts: Summary and the Good News

I want you to write an interrupt-enabled program for the online simulator for lab 2. Ouch! But the **good news** is that a sample program is provided that uses push button interrupts to write to the 7-segment display. [15] Really you just need to modify that program, and if you qualitatively understand the above steps it should be pretty easy.

- This program is provided by Altera, who make the Cyclone V chipset that includes the ARM®Cortex-A9 core.
- They are clever enough that they wrote a generalized subroutine to configure the GIC distributor interface, so all you need to do is make sure the appropriate IRQ number is passed to that subroutine and it will handle all the messy address and bit calculations.
- Since the lab only *requires* you to have push-button interrupts enabled (you have the option to include timer interrupts if you are ambitious), and these interrupts are already used in the sample program, you can just rewrite the ISR (called `KEY_ISR`) to as necessary, and of course rewrite the main program.

Just in case you feel the urge to write an interrupt-enabled program from scratch, the basic program flow is summarized in Figure 7.

[15] Source code is provided in Assembly (`Push_button_example.s`) and in C (`Push_button_example.c`), it is available in the *Laboratory* section of *Course Content*. You should be able to copy+paste the code into the simulator and try it out.

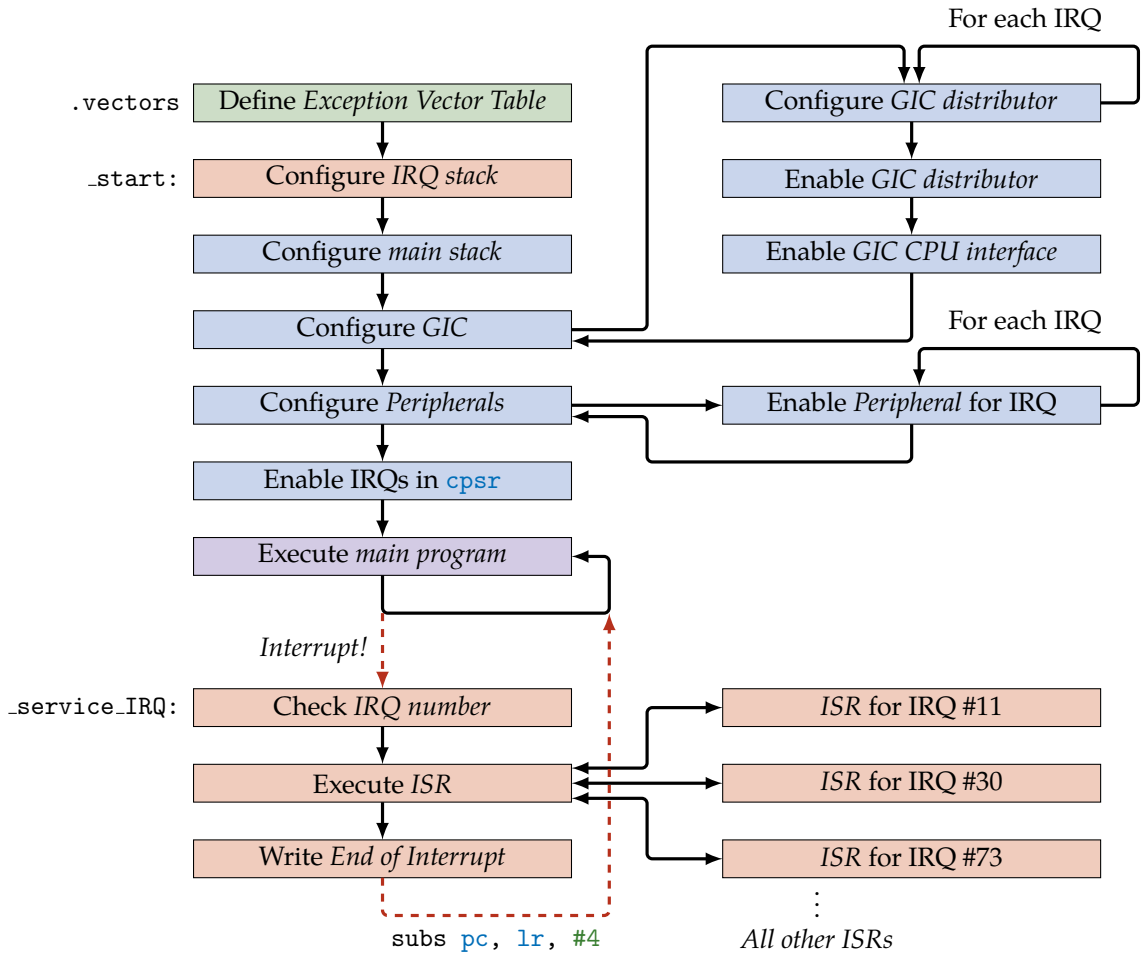

Figure 7: Program flow for enabling interrupts in an ARMv7-type CPU. Everything in blue is done in *supervisor mode*, everything in red is done in *IRQ mode*. Code in green are compiler-only instructions (i.e., to set the *exception vector table*), and the main program (in purple) could be executed in *supervisor mode* or *user mode*, as desired. All solid arrows indicate sequential flow or a branch to a normal subroutine. The red dashed arrows indicate a hardware-triggered branch according to the *exception vector table*, and an automatic change in the operating mode (to or from *IRQ mode*, as appropriate).