Lecture 03
Embedded Software

CE346 – Microcontroller System Design

Branden Ghena – Fall 2024

Some slides borrowed from:
Josiah Hester (Northwestern), Prabal Dutta (UC Berkeley)

Northwestern

Administrivia

- Make sure you have your personal lab setup working
  - Ask in office hours or on Piazza if you run into issues
- Office hours delayed until later this week
  - I have time after class today if people have questions
- Labs will start this Friday!!!
  - You MUST come to your scheduled lab session
  - Not really enough room for students to move sections
    - If there's some known obligation and you give me a heads up, I could approve a few per week

Today's Goals

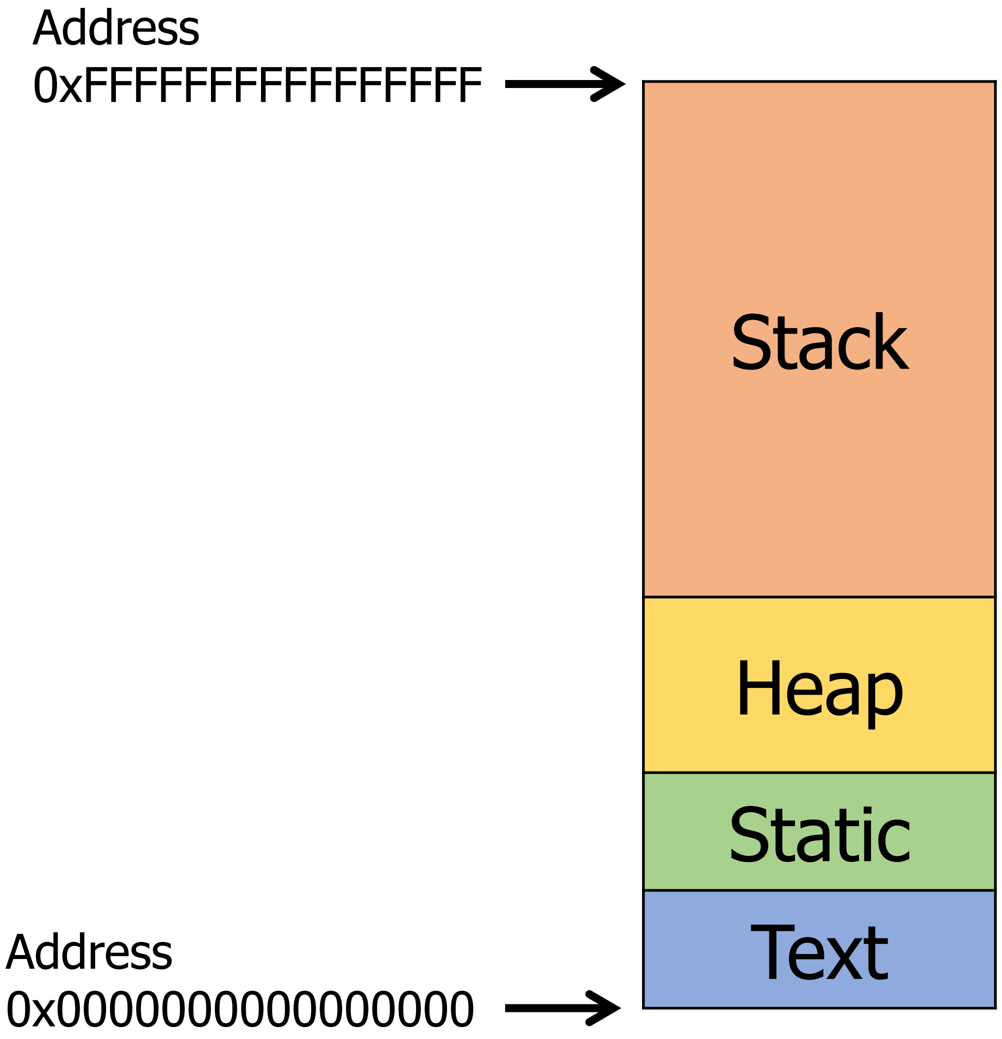
- Discuss challenges of embedded software
- Describe compilation and linking of embedded code
  - Actually applies to all code, but you probably never learned much about linking before
- Introduce new software pattern: interrupts
- Explore the microcontroller boot process

Outline

- **Embedded Software**
- Embedded Toolchain
- Lab Software Environment
- Interrupts
- Boot Process

Review: C memory layout

- Stack Section
    - Local variables
    - Function arguments
- Heap Section
    - Memory granted through `malloc()`
- Static Section (a.k.a. Data Section)
    - Global variables
    - Static function variables
- Text Section (a.k.a Code Section)
    - Program code



Assumptions of embedded programs

- Expect limitations
  - Very little memory
  - Very little computational power
  - Very little energy

- Don't expect a lot of support
  - Likely no operating system
  - Might not even have error reporting capabilities

- Moral: think differently about your programs

Ramifications of limited memory

- Stack and Data sections are limited
  - Be careful about too much recursion
  - Be careful about large local variables

  - Large data structures defined globally are preferred
    - Fail at compile time
    - In embedded, we often *encourage* global variables for large things

- Heap section is likely non-existent
  - **Why?**

Ramifications of limited memory

- Stack and Data sections are limited
  - Be careful about too much recursion
  - Be careful about large local variables

  - Large data structures defined globally are preferred
    - Fail at compile time
    - In embedded, we often *encourage* global variables for large things

- Heap section is likely non-existent
  - **Why?**
    - Malloc could run out of memory at runtime

Avoiding dynamic memory

- Malloc is ***scary*** in an embedded context
- What if there's no more memory available?
  - Traditional computer
    - Swap memory to disk
    - Worst case: wait for a process to end (or kill one)

  - Embedded computer
    - There's likely only a single application
    - And it's the one asking for more memory
    - So it's not giving anything back anytime soon

- This is unlikely to happen at boot
  - Instead it'll happen hours or days into running as memory is slowly exhausted…

Limitations on processing power

- Typically not all that important
  - Code still runs pretty fast
    - 10 MHz -> 100 ns per cycle (i.e. ~100 ns per instruction)
  - Controlling hardware usually doesn't have a lot of code complexity
    - Quickly gets to the "waiting on hardware" part (apps are I/O bound)

- Problems
  - Machine learning
    - Learning on the device is neigh impossible
    - Memory limitations make it hard to fit weights anyways
  - Cryptography
    - Public key encryption takes seconds to minutes

Common programming languages for embedded

- C
  - For all the reasons that you assume
  - Easy to map variables to memory usage and code to instructions
- Assembly
  - Not entirely uncommon, but rarer than you might guess
  - C code optimized by a modern compiler is likely faster
  - Notable uses:
    - Cryptography to create deterministic algorithms
    - Operating Systems to handle process swaps
- C++
  - Similar to C but with better library support
  - Libraries take up a lot of code space though ~100 KB

Rarer programming languages for embedded

- Rust
  - Modern language with safety and reliability guarantees
  - Increasingly relevant in the embedded space
    - But with a high learning curve

- Python, Javascript, etc.
  - Mostly toy languages
  - Fine for simple things but incapable of complex operations
    - Especially low-level things like managing memory

What’s missing from programming languages?

- The embedded domain has several requirements that other domains do not

- What is missing from programming languages that it wants?
  - Sense of time
  - Sense of energy

Programming languages have no sense of time

- Imagine a system that needs to send messages to a motor every 10 milliseconds
  - Write a function that definitely completes within 10 milliseconds

- Accounting for timing when programming is very challenging
  - We can profile code and determine timing at runtime
  - If we know many details of hardware, instructions can give timing
    - Unless the code interacts with external devices

Determining energy use is rather complicated

- Software might
  - Start executing a loop
  - Turn on/off an LED
  - Send messages over a wired bus to another device

- Determining energy these operations take is really difficult
  - Even with many details of the hardware
  - Different choices of processor clocks can have a large impact

  - Often profiled at runtime after writing the code
    - Iterative write-test-modify cycle

Break + Say hi to your neighbors

- Things to share
  - Name

  - Major

  - One of the following
    - Favorite Candy
    - Favorite Pokemon
    - Favorite Emoji

Break + Say hi to your neighbors

- Things to share
  - Name -Branden

  - Major -Electrical and Computer Engineering, and Computer Science

  - One of the following
    - Favorite Candy - Twix
    - Favorite Pokemon - Eevee
    - Favorite Emoji - 🍢

Outline

- Embedded Software
- **Embedded Toolchain**
- Lab Software Environment
- Interrupts
- Boot Process

Embedded compilation steps

- Same first steps as any system

1. Compiler
    - Turn C code into assembly
    - Optimize code (often for code size instead of speed)

Cross compilers compile for different architectures

- The compiler we'll be using is a cross compiler
  - Run on one architecture but compile code for another
    - Example: runs on x86-64 but compiles armv7e-m

- GCC naming scheme: ARCH-VENDOR-(OS-)-ABI-gcc
  - arm-none-eabi-gcc
    - ARM architecture
    - No vendor
    - No OS
    - Embedded Application Binary Interface
  - Others: arm-none-linux-gnueabi-gcc, i686-pc-windows-msvc-gcc

Embedded compilation steps

- Same first steps as any system

1. Compiler
    - Turn C code into assembly
    - Optimize code (often for size instead of speed)

2. Linker
    - Combine multiple C files together
    - Resolve dependencies
        - Point function calls at correct place
        - Connect creation and uses of global variables

Informing linker of system memory

- Linker actually places code and variables in memory
  - It needs to know where to place things

- **How do x86-64 compilers know which addresses to use?**

Informing linker of system memory

- Linker actually places code and variables in memory
  - It needs to know where to place things

- **How do x86-64 compilers know which addresses to use?**
  - Virtual memory allows all applications to use the same memory addresses

- Embedded solution
  - Only run a single application
  - Provide an LD file
    - Specifies memory layout for a certain system
    - Places sections of code in different places in memory

Anatomy of an LD file

- nRF52833: 512 KB Flash, 128 KB SRAM
- First, LD file defines memory regions

```
MEMORY {
  FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x80000
  RAM (rwx) :  ORIGIN = 0x20000000, LENGTH = 0x20000
}
```

- A neat thing about microcontrollers: pointers have meaning
  - Just printing the value of a pointer can tell you if it's in Flash or RAM

Anatomy of an LD file

- It then places sections of code into those memory regions

```
.text : {
    KEEP(*(.Vectors))
    *(.text*)
    *(.rodata*)
    . = ALIGN(4);
} > FLASH
__etext = .;
```

```
.data : AT (__etext) {
        __data_start__ = .;
        *(.data*)
        __data_end__ = .;
} > RAM

.bss :    {
        . = ALIGN(4);
        __bss_start__ = .;
        *(.bss*)
        . = ALIGN(4);
        __bss_end__ = .;
} > RAM
```

Sections of code

- Where do these sections come from?
- Most are generated by the compiler
  - .text, .rodata, .data, .bss
  - You need to be deep in the docs to figure out how the esoteric ones work

- Some are generated by the programmer
  - Allows you to place certain data items in a specific way

```
__attribute__((section(".foo")))
int test[10] = {0,0,0,0,0,0,0,0,0,0};
```

Embedded compilation steps

- Same first steps as any system

1. Compiler
   - Turn C code into assembly
   - Optimize code (often for size instead of speed)

2. Linker
   - Combine multiple C files together
   - Resolve dependencies
     - Point function calls at correct place
     - Connect creation and uses of global variables

- Output: a binary (or hex) file

Loading the hex file onto a board

- This is a use case for JTAG
  - You provide it a hex file which specifies addresses and values
  - It writes those into Flash on the microcontroller

- The LD file already specified addresses
  - So passing around hex files is enough to load an application

  - But a hex file for one microcontroller won’t work on another with a different memory layout

Example

- Demonstrated in the blink application in lab repo
  - https://github.com/nu-ce346/nu-microbit-base/tree/main/software/apps/blink

Outline

- Embedded Software
- Embedded Toolchain
- **Lab Software Environment**
- Interrupts
- Boot Process

Embedded environments

- There are a multitude of embedded software systems
  - Every microcontroller vendor has their own
  - Popular platforms like Arduino

- We’re using the Nordic software development libraries plus some extensions made by my research group
  - It’ll be a week until that matters for the most part
  - We’ll start off by writing low-level drivers ourselves without libraries

Software Development Kit (SDK)

- Libraries provided by Nordic for using their microcontrollers
  - Actually incredibly well documented! (relatively)
  - Various peripherals and library tools

- SDK documentation
  - https://docs.nordicsemi.com/bundle/sdk_nrf5_v16.0.0/page/index.html
  - Warning: search doesn't really work

- Possibly more useful: the list of data structures
  - Search that page for whatever "thing" you're working with
  - https://docs.nordicsemi.com/bundle/sdk_nrf5_v16.0.0/page/annotated.html

nRF52x-base

- Wrapper built around the SDK by Lab11
  - Branden Ghena, Brad Campbell (UVA), Neal Jackson, a few others
  - Allows everything to be used with Makefiles and command line
  - https://github.com/lab11/nrf52x-base

- We include it as a submodule
  - It has a copy of the SDK code and softdevice binaries
  - It has a whole Makefile system to include to proper C and H files
  - We include a Board file that specifies our specific board's needs and capabilities

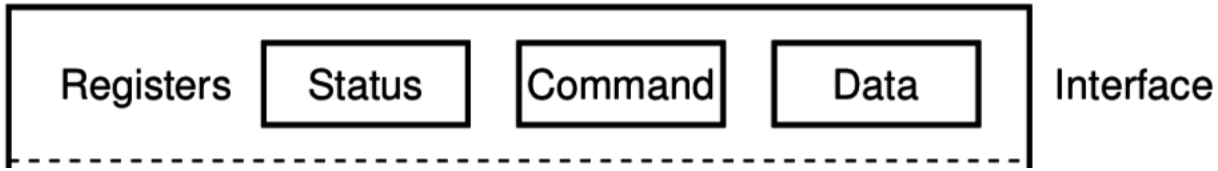
- Go to repo to explain

Break

Outline

- Embedded Software
- Embedded Toolchain
- Lab Software Environment
- **Interrupts**
- Boot Process

What do interactions with devices look like?



1. while STATUS==BUSY; Wait
   - (Need to make sure device is ready for a command)
2. Write value(s) to DATA
3. Write command(s) to COMMAND
4. while STATUS==BUSY; Wait
   - (Need to make sure device has completed the request)
5. Read value(s) from Data

This is the "polling" model of I/O.

"Poll" the peripheral in software repeatedly to see if it's ready yet.

Waiting can be a waste of CPU time

1. **while STATUS==BUSY; Wait**
    - **(Need to make sure device is ready for a command)**
2. Write value(s) to DATA
3. Write command(s) to COMMAND
4. **while STATUS==BUSY; Wait**
    - **(Need to make sure device has completed the request)**
5. Read value(s) from Data

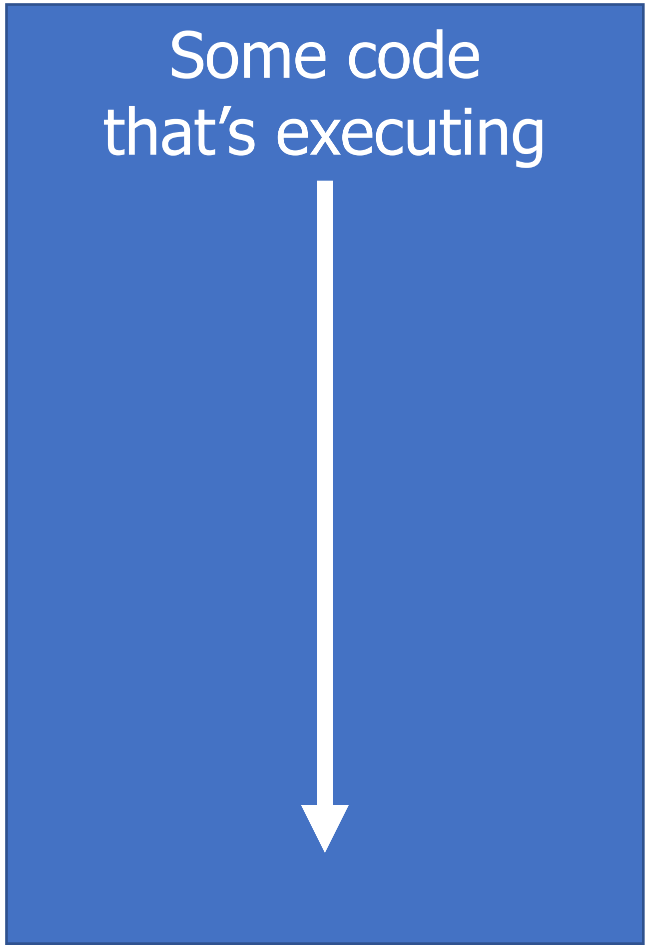
- Problem: imagine a keyboard device
    - CPU could be waiting for minutes before data arrives
    - Need a way to notify CPU when an event occurs
        - Interrupts!

Interrupts

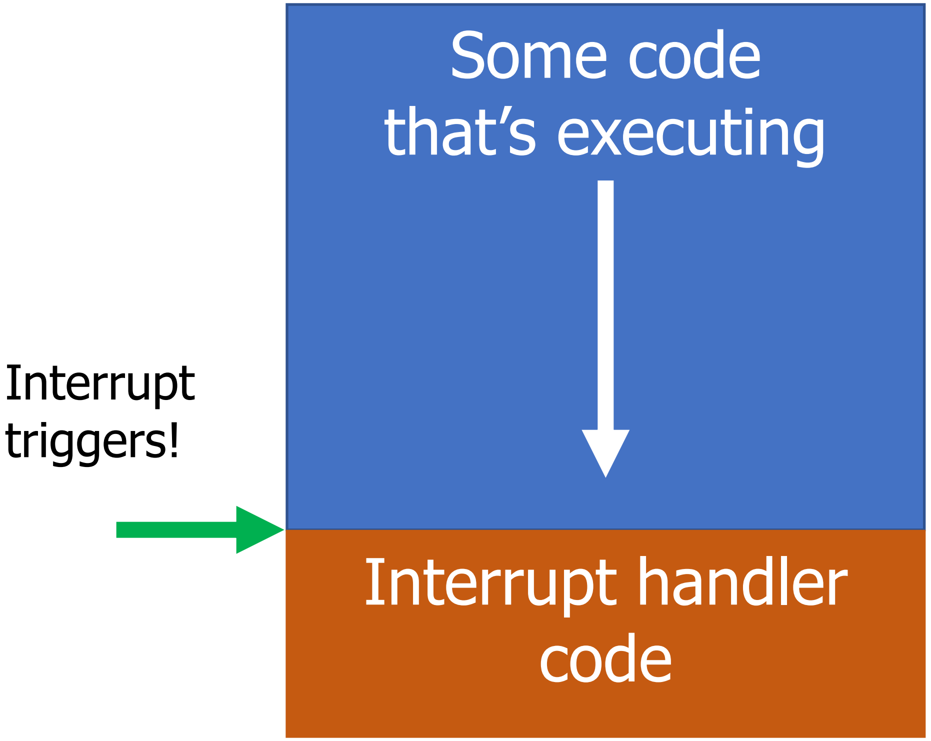
- What is an interrupt?
  - Some event which causes the processor to stop normal execution
  - The processor instead jumps to a software “handler” for that event
    - Then returns back to what it was doing afterwards

- What causes interrupts?
  - Hardware exceptions
    - Divide by zero, Undefined Instruction, Memory bus error
  - Software
    - Syscall, Software Interrupt (SWI)
  - External hardware
    - Input pin, Timer, various “Data Ready”

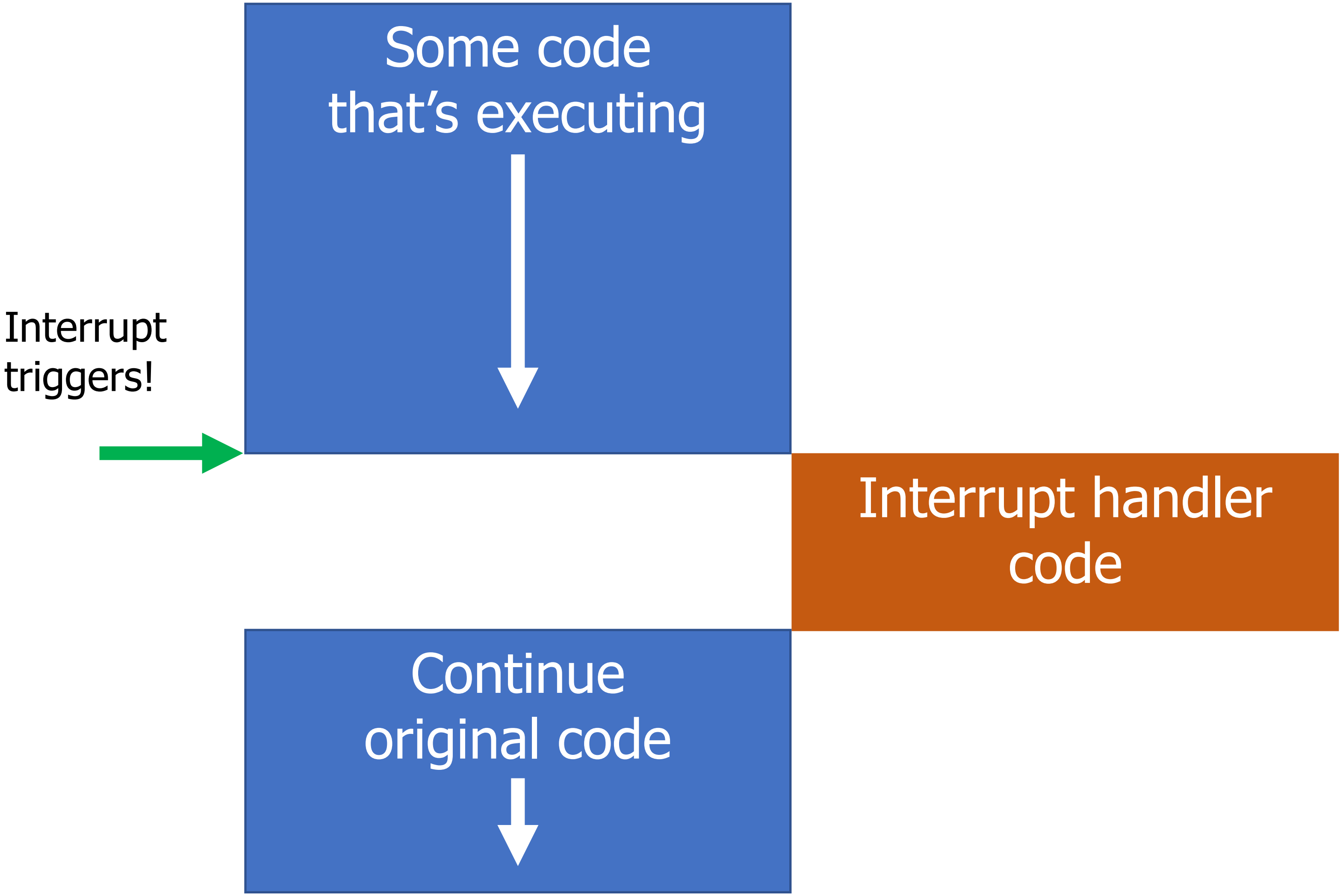
Interrupts, visually



Interrupts, visually



Interrupts, visually



ARM Nested Vectored Interrupt Controller (NVIC)

Interrupts can preempt other interrupts!

Jump directly to the interrupt handler

Handles interrupt entry and exit
- Stacking
- Unstacking
- Priorities

- Manages interrupt requests (IRQ)
  - Stores all caller-saved registers on the stack
    - So the handler code doesn't overwrite them
  - Moves execution to proper handler, a.k.a. Interrupt Service Routine (ISR)
  - Restores registers after handler returns and moves execution back

ARM Vector table

- List of function pointers to handler for each interrupt/exception

- First 15 are architecture-specific exceptions

- After that are microcontroller interrupt signals

Table 7.1 List of System Exceptions

Exception Number	Exception Type	Priority	Description
1	Reset	–3 (Highest)	Reset
2	NMI	–2	Nonmaskable interrupt (external NMI input)
3	Hard fault	–1	All fault conditions if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; Memory Protection Unit (MPU) violation or access to illegal locations
5	Bus fault	Programmable	Bus error; occurs when Advanced High-Performance Bus (AHB) interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access)
6	Usage fault	Programmable	Exceptions resulting from program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	—
11	SVC	Programmable	Supervisor Call
12	Debug monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	—
14	PendSV	Programmable	Pendable Service Call
15	SYSTICK	Programmable	System Tick Timer

Table 7.2 List of External Interrupts

Exception Number	Exception Type	Priority
16	External Interrupt #0	Programmable
17	External Interrupt #1	Programmable
…	…	…
255	External Interrupt #239	Programmable

Vector table in software

- Placed in its own section
  - LD file puts it first in Flash
- Reset_Handler determines where software starts executing
- After that are all exception and interrupt handlers
  - All function pointers to some C code somewhere

```
    .section .isr_vector
    .align 2
    .globl __isr_vector
__isr_vector:
    .long   __StackTop                  /* Top of Stack */
    .long   Reset_Handler
    .long   NMI_Handler
    .long   HardFault_Handler
    .long   MemoryManagement_Handler
    .long   BusFault_Handler
    .long   UsageFault_Handler
    .long   0                           /*Reserved */
    .long   0                           /*Reserved */
    .long   0                           /*Reserved */
    .long   0                           /*Reserved */
    .long   SVC_Handler
    .long   DebugMon_Handler
    .long   0                           /*Reserved */
    .long   PendSV_Handler
    .long   SysTick_Handler

  /* External Interrupts */
    .long   POWER_CLOCK_IRQHandler
    .long   RADIO_IRQHandler
    .long   UARTE0_UART0_IRQHandler
    .long   SPIM0_SPIS0_TWIM0_TWIS0_SPI0_TWI0_IRQHandler
    .long   SPIM1_SPIS1_TWIM1_TWIS1_SPI1_TWI1_IRQHandler
    .long   NFCT_IRQHandler
    .long   GPIOTE_IRQHandler
    .long   SAADC_IRQHandler
```

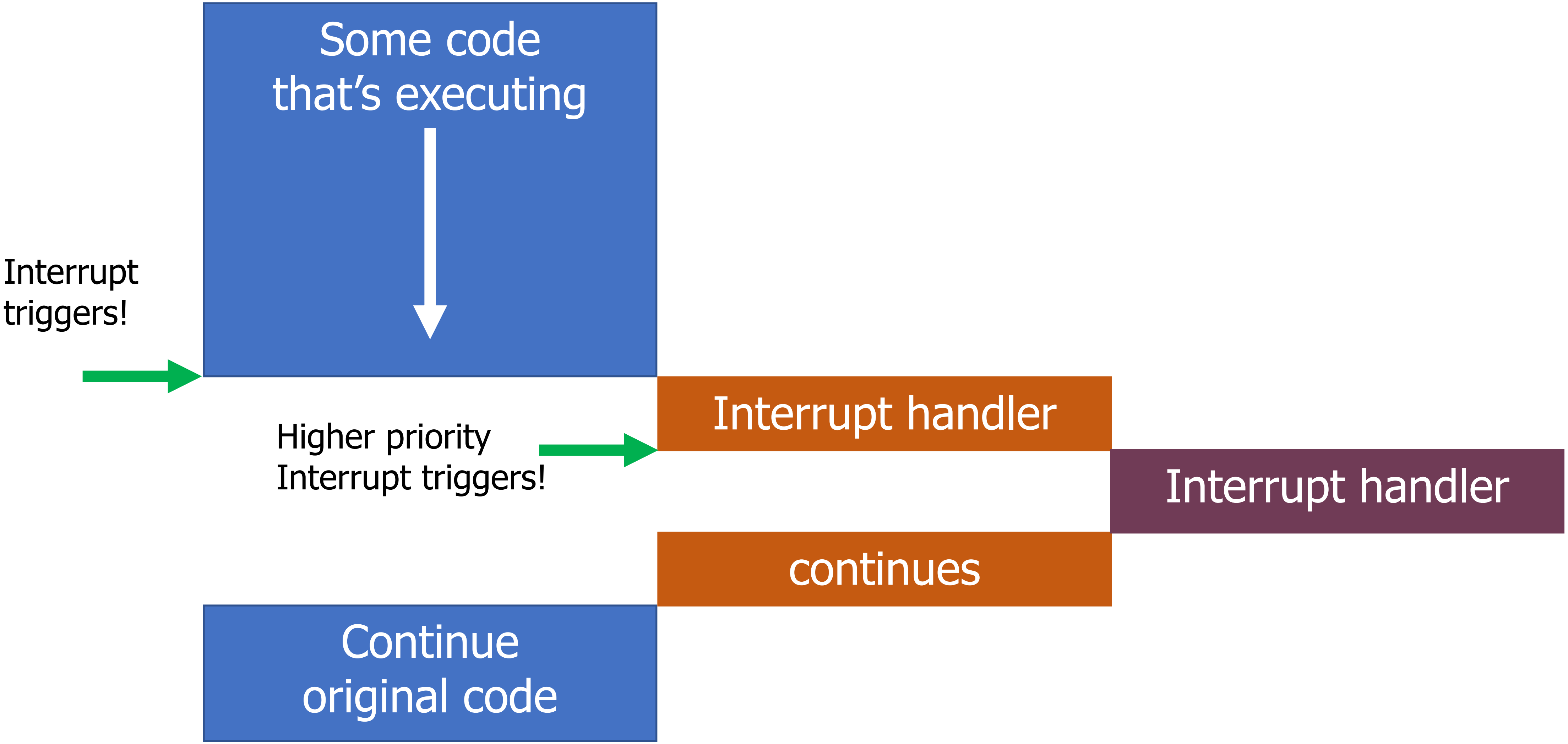
NVIC functionality

- NVIC functions
  - `NVIC_EnableIRQ(number)`
  - `NVIC_DisableIRQ(number)`
  - `NVIC_SetPriority(number, priority)`
    - Technically 256 priorities
    - Only 8 are implemented

- Must enable interrupts in two places!
  - Enabling interrupt in the peripheral will generate the signal
  - Enabling interrupt in the NVIC will cause signal to jump to handler

- Priority determines which interrupt goes first
  - And determines how interrupts are nested

Nested interrupts, visually



Break + Open Question

- When should a system use polling versus interrupts?

Break + Open Question

- When should a system use polling versus interrupts?

- Polling
  - Great if the device is going to respond immediately (like 1 cycle)
  - Important if we need to respond very quick (less than a microsecond)

- Interrupts
  - Great if we'll need to wait a long time for status to change
  - Still responds pretty quickly, but not *immediately*
    - Needs to context switch from running code to interrupt handler

Outline

- Embedded Software
- Embedded Toolchain
- Lab Software Environment
- Interrupts
- **Boot Process**

How does a microcontroller *start* running code?

- Power comes on
- Microcontroller needs to start executing assembly code

- You expect your main() function to run
  - But a few things need to happen first

Step 0: set a stack pointer

- Assembly code might need to write data to the stack
  - Might call functions that need to stack registers

- ARM: Valid address for the stack pointer is at address 0 in Flash
  - Needs to point to somewhere in RAM

  - Hardware loads it into the Stack Pointer when it powers on

Step 1: set the program counter (PC)

- a.k.a. the Instruction Pointer (IP) in x86 land

- 32-bit ARM: valid instruction pointer is at address 4 in Flash
  - Could point to RAM, usually to Flash though
  - In interrupt terms: this is the “Reset Handler”!

  - Automatically loaded into the PC after the SP is loaded
    - Again, hardware does this

Step 2: “reset handler” prepares memory

- Code that handles system resets
  - Either reset button or power-on reset
  - Address was loaded into PC in Step 1

- Reset handler code:
  - Loads initial values of .data section from Flash into RAM
  - Loads zeros as values of .bss section in RAM
  - Calls SystemInit
    - Starts correct clocks for the system
    - Handles various hardware configurations/errata
  - Calls _start

nu-microbit-base/software/nrf52x-base/sdk/nrf5_sdk_16.0.0/modules/nrfx/mdk/gcc_startup_nrf52833.S

nu-microbit-base/software/nrf52x-base/sdk/nrf5_sdk_16.0.0/modules/nrfx/mdk/system_nrf52.c

Step 3: set up C runtime

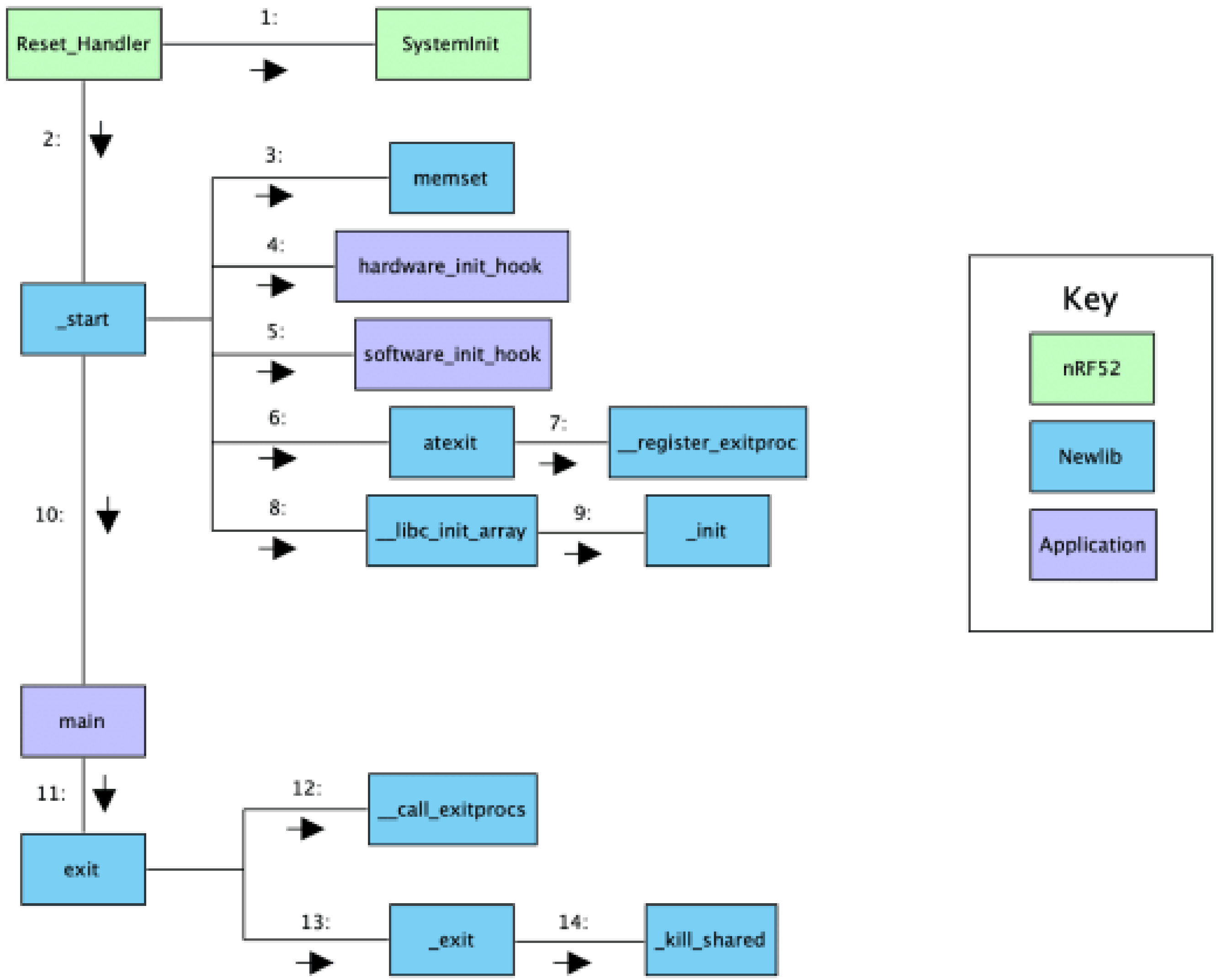
- _start is provided by newlib
  - An implementation of libc – the C standard library
  - Startup is a file usually named crt0

- Does more setup, almost none of which is relevant for our system
  - Probably is this code that actually zeros out .bss
  - Sets argc and argv to 0
  - Calls main() !!!

https://sourceware.org/git/gitweb.cgi?p=newlib-cygwin.git;a=blob_plain;f=libgloss/arm/crt0.S;hb=HEAD

Online writeup with way more details and a diagram

- Relevant guide!!
  - https://embeddedartistry.com/blog/2019/04/17/exploring-startup-implementations-newlib-arm/
  - Covers the nRF52!



Outline

- Embedded Software
- Embedded Toolchain
- Lab Software Environment
- Interrupts
- Boot Process