Lecture 03
Embedded Software

CE346 – Microprocessor System Design

Branden Ghena – Spring 2021

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

Northwestern

Updates

- Hardware for anyone who filled out the form by class Wednesday
  - Should be here today? Maybe Monday.

- All orders have been placed
  - Except for a few new students

Today’s Goals

- Discuss challenges of embedded software

- Describe compilation and linking of embedded code

- Explore the microcontroller boot process

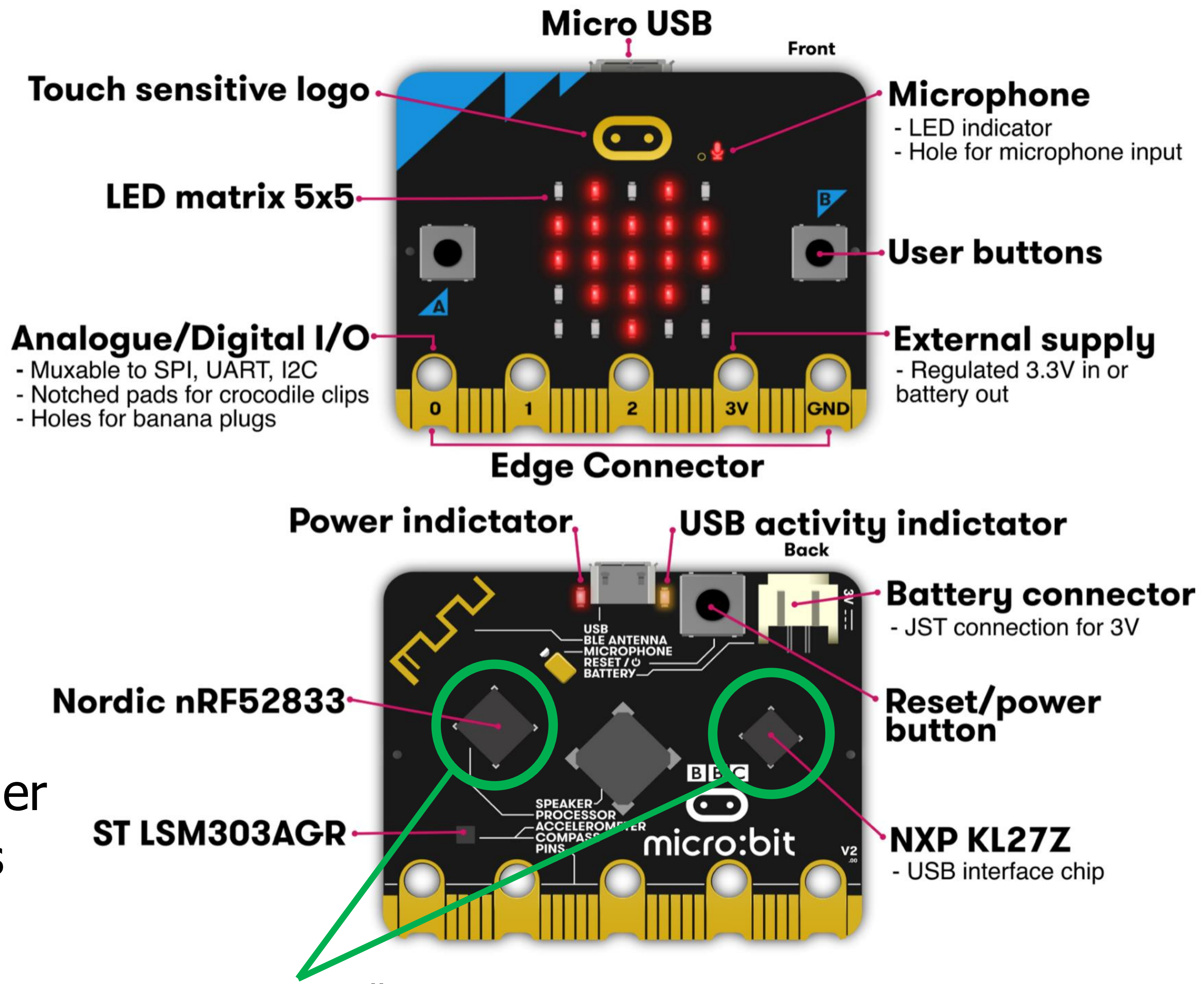
Outline

- **Microbit microcontroller**
- Embedded Software
- Embedded Toolchain
- Lab Software Environment
- Boot Process

Micro:bit v2

- Circuit board
  - Entire thing
  - a.k.a "Dev Board"
  - a.k.a PCB
    (Printed Circuit Board)

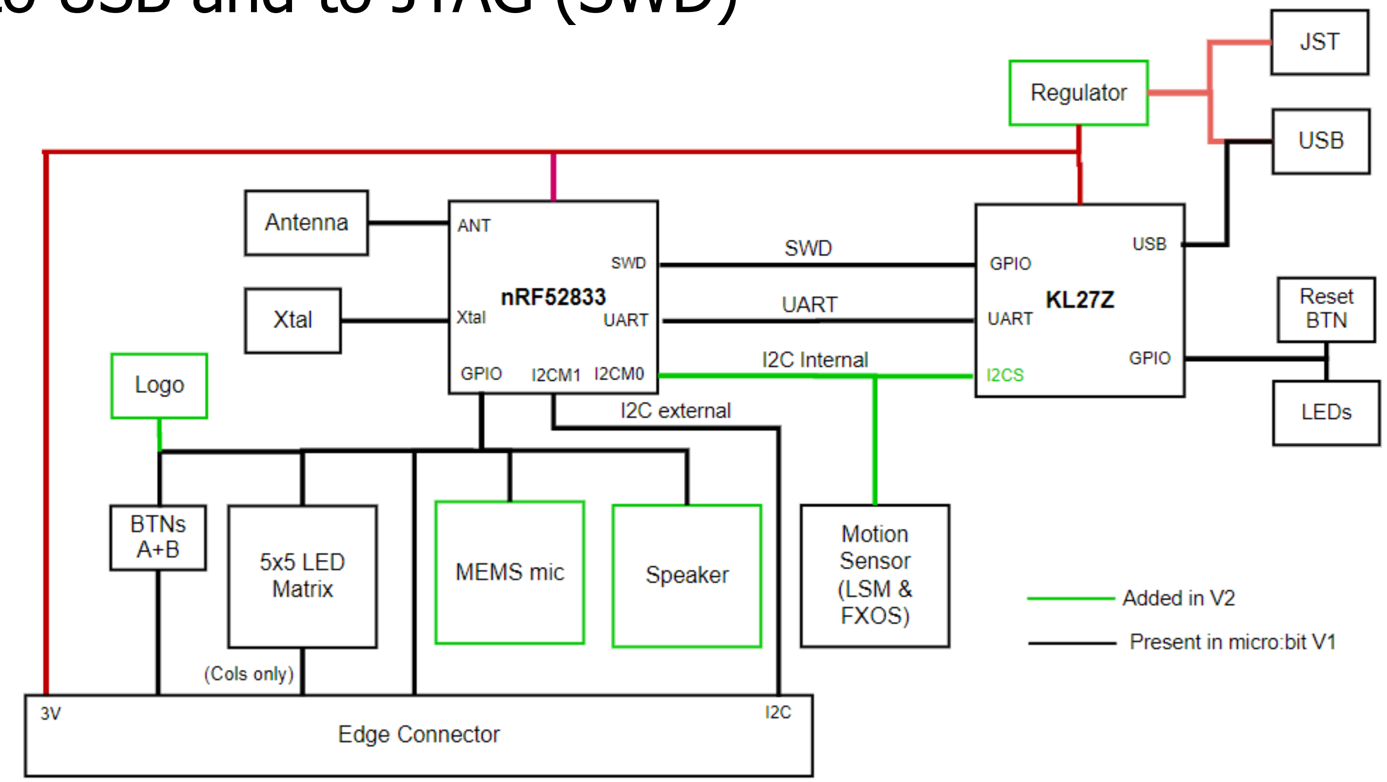
- Microcontroller
  - The computer on it
  - Microbit has two
    - One as a programmer
    - One for applications



Microcontrollers

KL27Z – Programmer on the Micro:bit v2

- 32-bit ARM Cortex-M0+ microcontroller
  - 48 MHz core, 16 KB RAM, 256 KB Flash
- Acts as a programming interface to nRF52833
  - Connects to USB and to JTAG (SWD)



nRF52833 – Microcontroller on the Micro:bit v2

- 32-bit ARM Cortex-M4F microcontroller
  - 64 MHz core, 128 KB RAM, 512 KB Flash
  - Floating point support
  - 2.4 GHz Radio: Bluetooth Low energy / 802.15.4
  - Various peripherals
    - ADC, PWM
    - I2C, UART, SPI, USB
    - RNG, 32-bit Timers, Watchdog, Temperature
    - Up to 42 I/O pins

To the datasheet!

- nRF52833 Product Specification
  - Online: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fps_nrf52833%2Fkeyfeatures_html5.html
  - PDF: https://infocenter.nordicsemi.com/pdf/nRF52833_PS_v1.3.pdf

Outline

- Microbit microcontroller
- **Embedded Software**
- Embedded Toolchain
- Lab Software Environment
- Boot Process

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

- 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

- 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
  - Controlling hardware usually doesn’t have a lot of code complexity
    - Quickly gets to the “waiting on hardware” part

- 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

Rarer programming languages for embedded

- C++
  - Similar to C but with better library support
  - Libraries take up a lot of code space though ~100 KB

- Rust
  - Modern language with safety and reliability guarantees
  - Relatively new to the embedded space
    - And a high learning curve

- Python, Javascript, etc.
  - Mostly toy languages
  - Fine for simple things but incapable of complex operations

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 clocks can have a large impact

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

Outline

- Microbit microcontroller
- Embedded Software
- **Embedded Toolchain**
- Lab Software Environment
- Boot Process

Embedded compilation steps

- Same first steps as any system

1. Compiler
    - Turn C code into assembly
    - Optimize code (often for 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 is named: ARCH-VENDOR-(OS-)-ABI-gcc
  - arm-none-eabi-gcc
    - ARM architecture
    - No vendor
    - No OS
    - Embedded Application Binary Interface
  - Others: arm-none-linux-gnueabi, i686-unknown-linux-gnu

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 traditional computers handle this?**

Informing linker of system memory

- Linker actually places code and variables in memory
  - It needs to know where to place things
- **How do traditional computers handle this?**
  - 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

Example

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

Outline

- Microbit microcontroller
- Embedded Software
- Embedded Toolchain
- **Lab Software Environment**
- 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 plus some extensions made by my research group
  - It’ll be a few weeks until that matters for the most part
  - We’ll start off by writing low-level drivers ourselves

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://infocenter.nordicsemi.com/topic/sdk_nrf5_v16.0.0/index.html
  - Warning: search doesn't really work

- Most useful link is probably to the list of data structures
  - https://infocenter.nordicsemi.com/topic/sdk_nrf5_v16.0.0/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

Outline

- Microbit microcontroller
- Embedded Software
- Embedded Toolchain
- Lab Software Environment
- **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

- ARM: valid instruction pointer is at address 4 in Flash
    - Could point to RAM, usually to Flash though
    - 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
    - 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

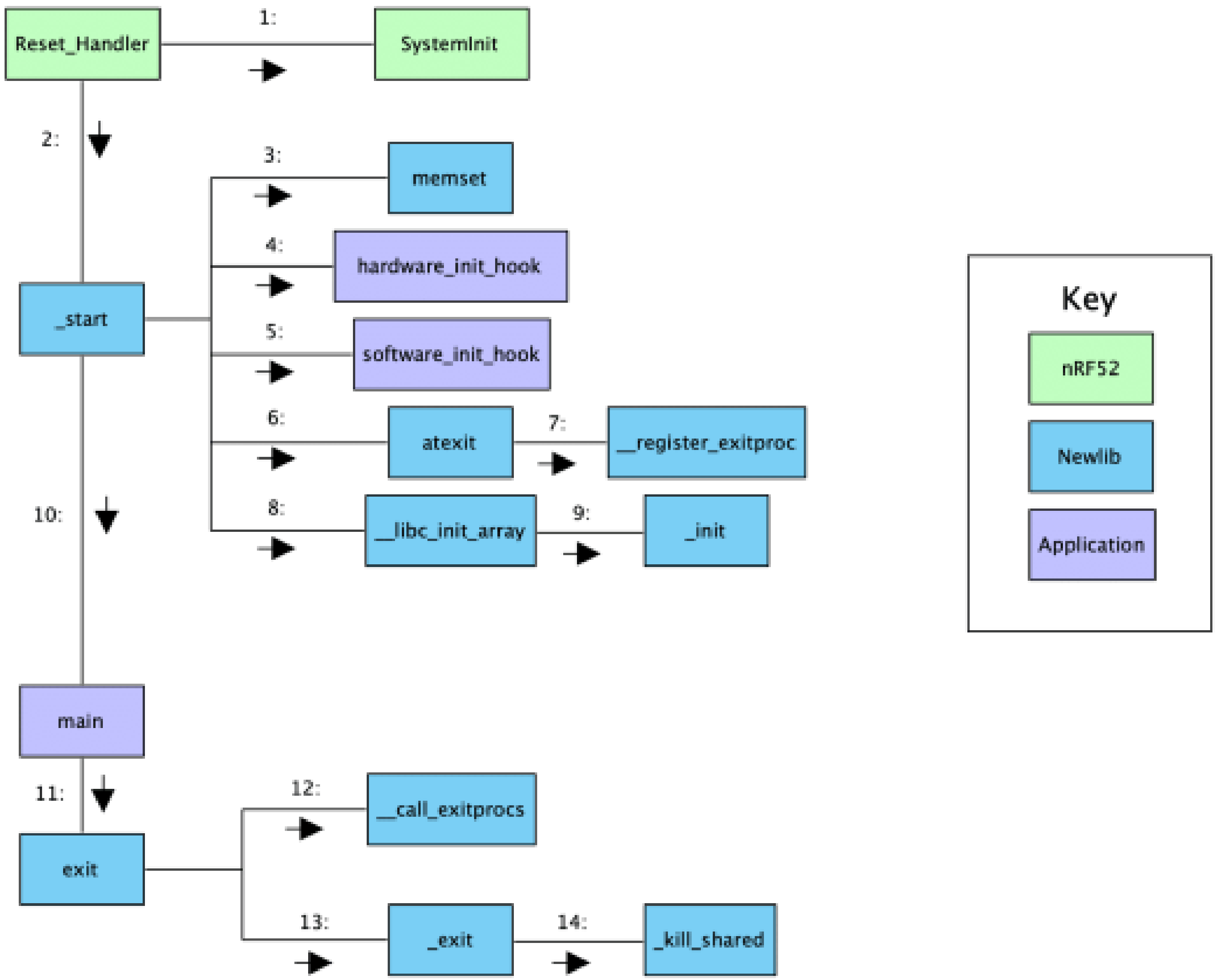
- _start is provided by newlib
  - An implementation of libc – the C standard library
  - Startup is a file usually named crt0

- 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

- Microbit microcontroller
- Embedded Software
- Embedded Toolchain
- Lab Software Environment
- Boot Process