Lecture 02
Embedded Software

CE346 – Microcontroller System Design

Branden Ghena – Spring 2025

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

Administrivia

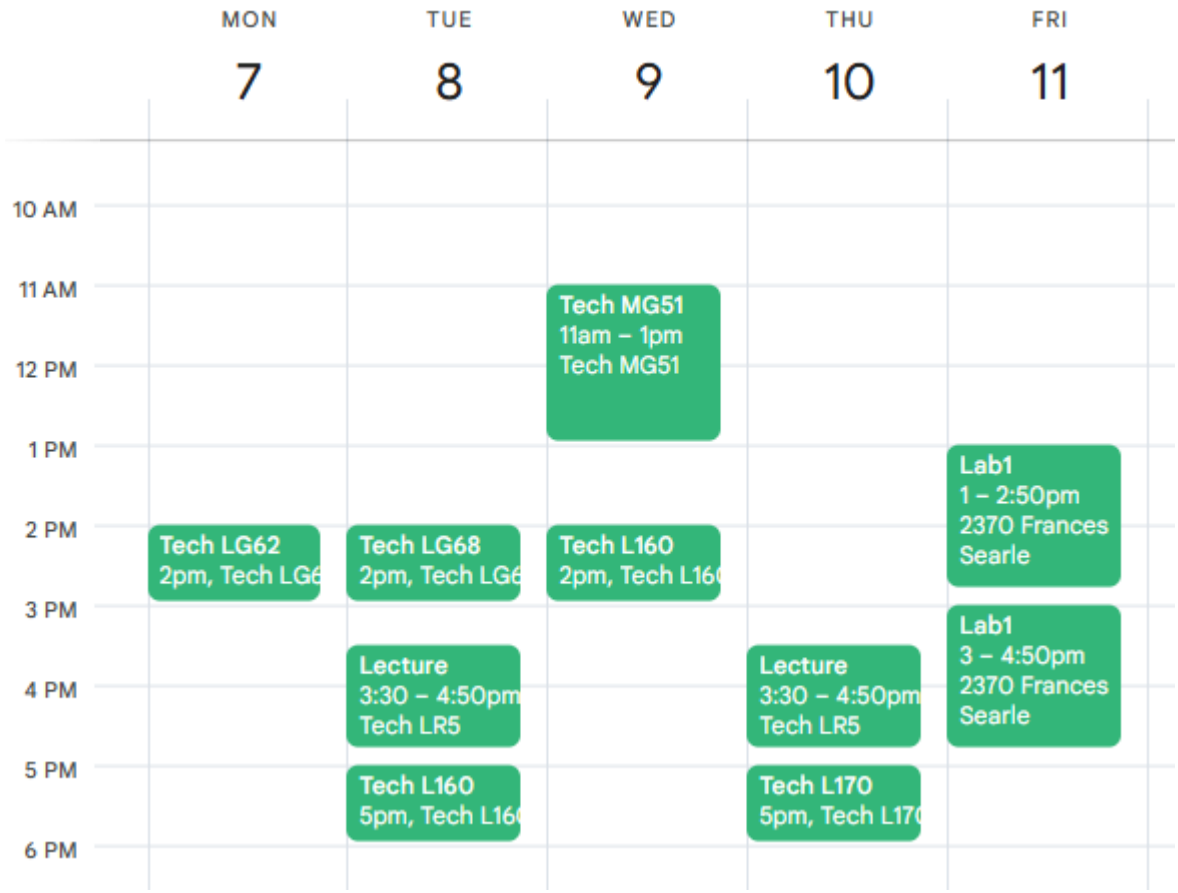
- Make sure you have your personal lab setup working
  - Ask in office hours or on Piazza if you run into issues

- Labs will start this Friday!!!
  - You MUST come to your scheduled lab session
    - If there's some known obligation and you give me a heads up, I can let you move to the other section
  - If you want to permanently change lab sessions (to the later lab at 3pm) let me know
  - I'll hand out Microbits as part of lab

Weekly Schedule

- Office Hours
  - Monday 2-3
  - Tuesday 2-3, 5-6 (joint)
  - Wednesday 11-1, 2-3
  - Thursday 5-6

- Checkoffs and debugging
  - But also ANY class question
  - These are for you to use



Today's Goals

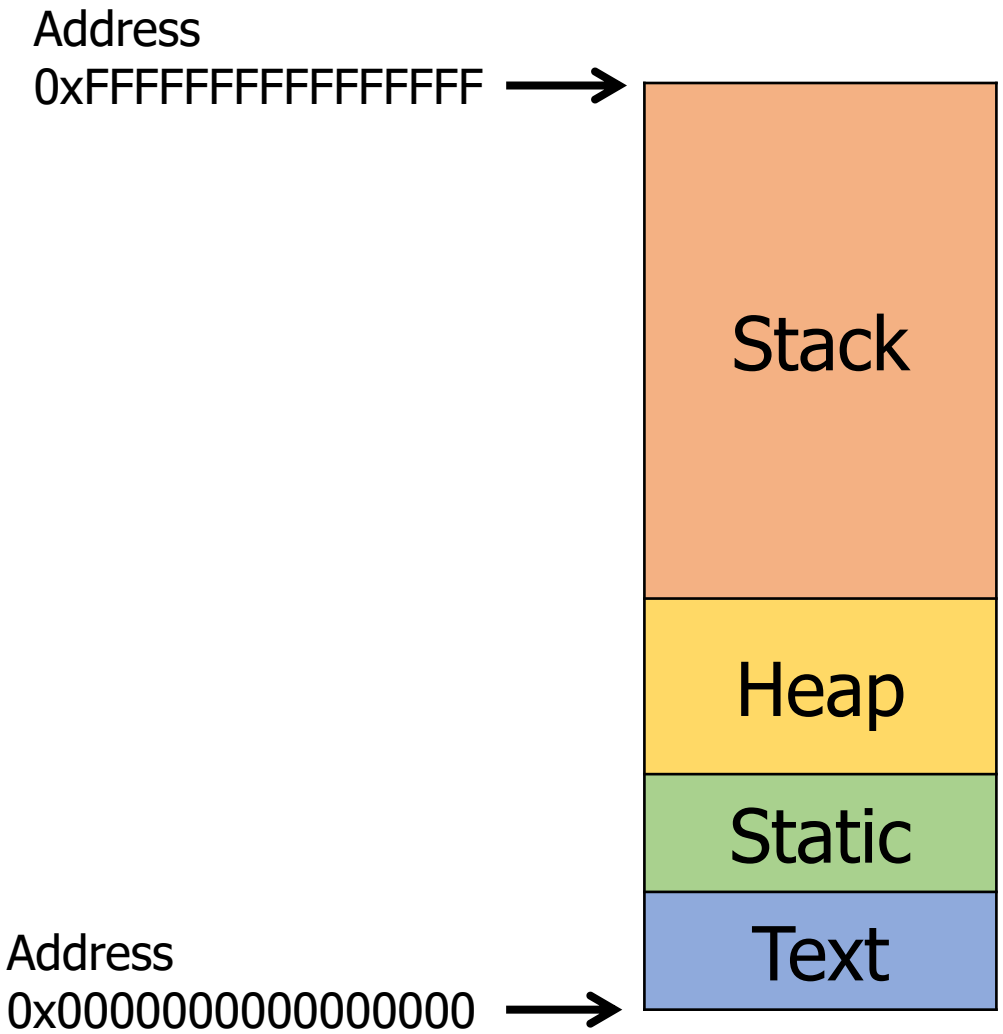
- Discuss challenges of embedded software
- Describe compilation and linking of embedded code
  - (Actually applies to all code, but you probably never learned much about linking before)
- Introduce Memory-Mapped I/O as a mechanism for communicating with peripherals
- Explore the microcontroller boot process

Outline

- **Embedded Software Overview**
- Embedded Toolchain
  - Lab Software Environment
- Memory-Mapped I/O
- 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
- Your code runs the entire system
  - Single application on the system that runs forever
- 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
  - In embedded, we often *encourage* global variables for large things
  - Fail at compile time rather than run-time

- Heap section might exist but should only be used cautiously
  - What do you do if the heap runs out?

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)
    - ~10 million instructions per second
  - 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 a big challenge
    - 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 can take up a lot of code space though ~100 KB
    - Some specialized versions of libraries take up less space

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

  - Microbit supports all of these, play around with them sometime:
    - https://makecode.microbit.org/
    - https://python.microbit.org/v/3
    - https://scratch.mit.edu/microbit

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 + Question

- Which program takes longer to run?

Program 1:

```
volatile float value = 1337.3235;
volatile float result = sqrt(value);
```

Program 2:

```
printf("Hello world!\n");
```

Break + Question

- Which program takes longer to run?

Program 1:

```
volatile float value = 1337.3235;
volatile float result = sqrt(value);
```

33 microseconds
~2112 instructions

Program 2:

```
printf("Hello world!\n");
```

3392 microseconds
~217088 instructions

Majority of the time isn't spent in instructions though

Printf takes about 100x longer!!

Outline

- Embedded Software Overview
- **Embedded Toolchain**
  - Lab Software Environment
- Memory-Mapped I/O
- 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?**
  - Standard layout for all processes
  - Virtual memory allows all applications to use the same memory addresses

- Embedded solution
  - Only run a single application
  - Provide our own standard layout: 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 Overview
- Embedded Toolchain
  - **Lab Software Environment**
- Memory-Mapped I/O
- 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

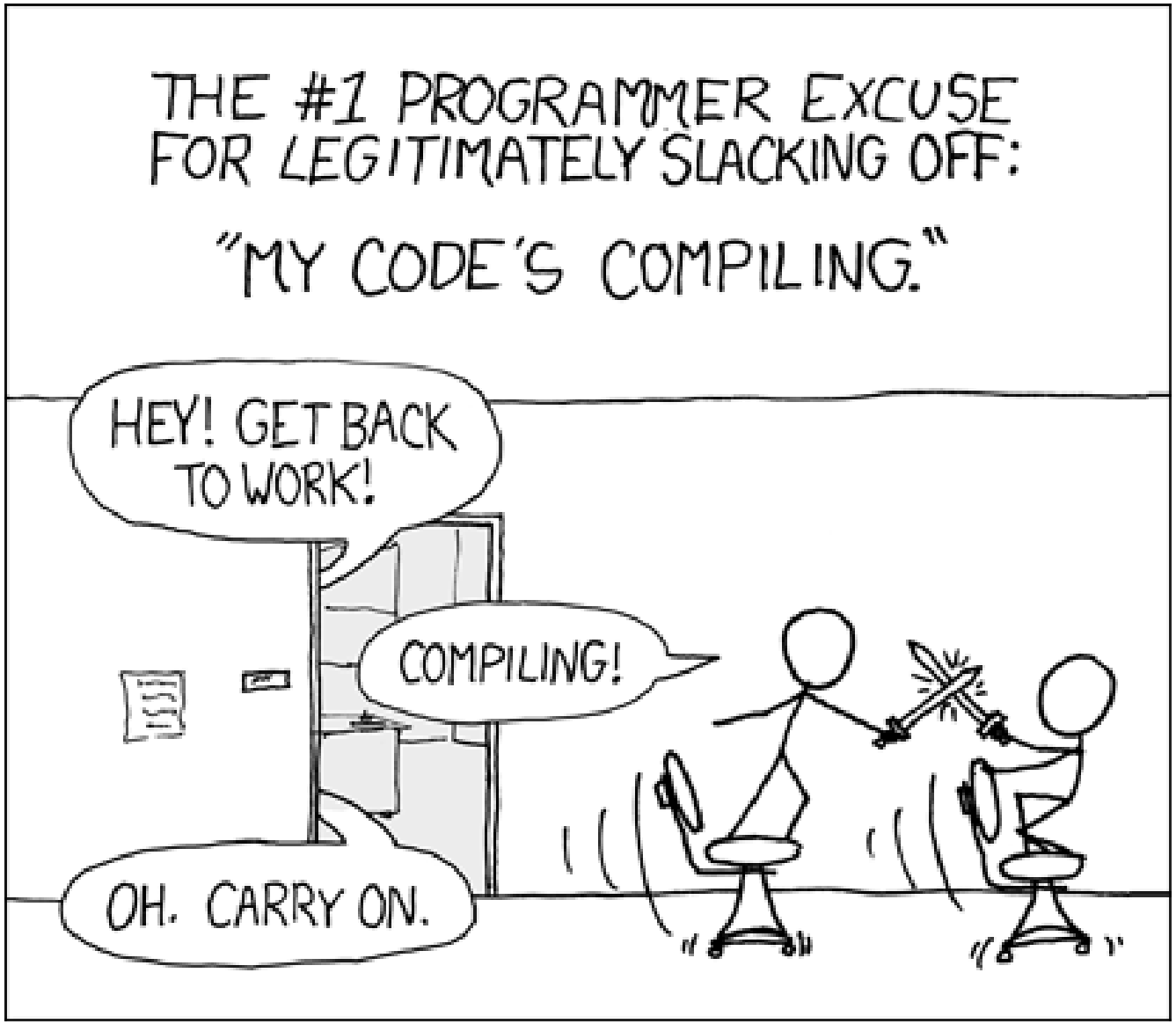
nRF52x-base

- 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 + xkcd



https://xkcd.com/303/

Outline

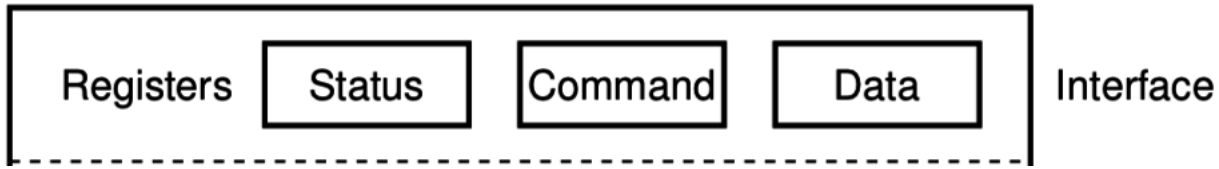
- Embedded Software Overview
- Embedded Toolchain
  - Lab Software Environment
- **Memory-Mapped I/O**
- Boot Process

How does a computer talk with peripherals?

- A peripheral is a hardware unit within a microcontroller
  - Sort of a "computer-within-the-computer"
  - Performs some kind of action given input, generates output

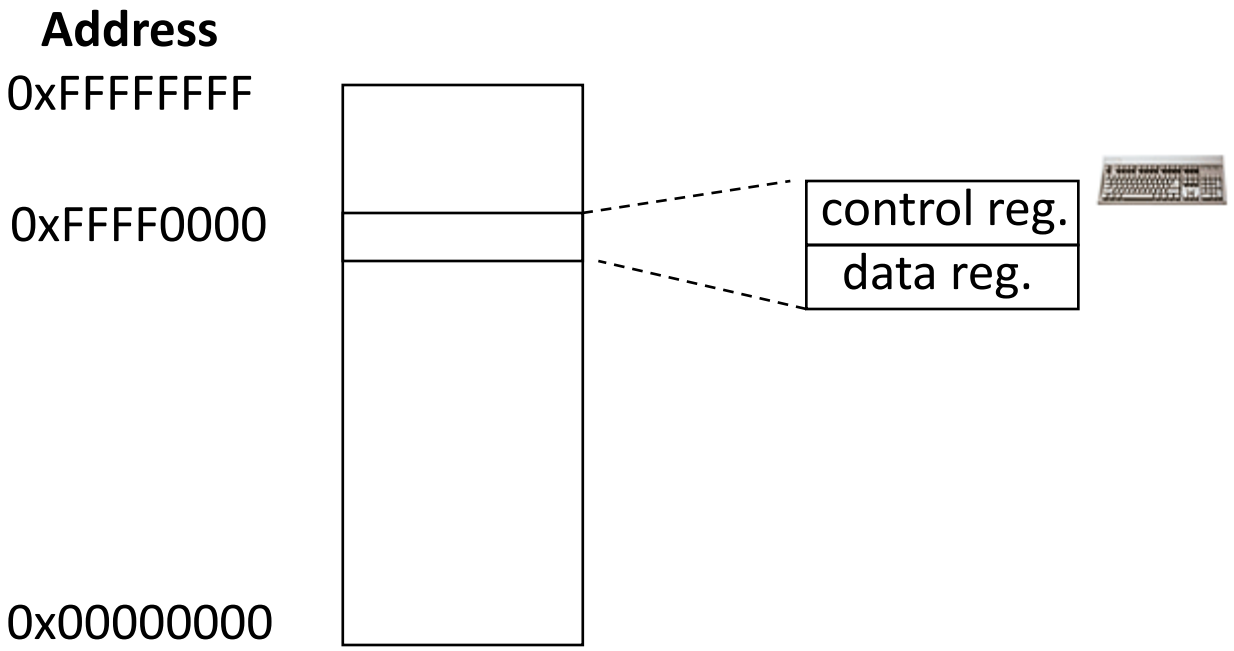
- We interact with a peripheral's interface
  - Called registers (actually are from EE perspective, but you can't use them)
  - Read/Write like they're data

- How do we read/write them?
  - Options:
    - Special assembly instructions
    - Treat like normal memory



Memory-mapped I/O (MMIO): treat devices like normal memory

- Certain physical addresses do not actually go to memory
- Instead they correspond to peripherals
  - And any instruction that accesses memory can access them too!
- Every microcontroller I've ever seen uses MMIO

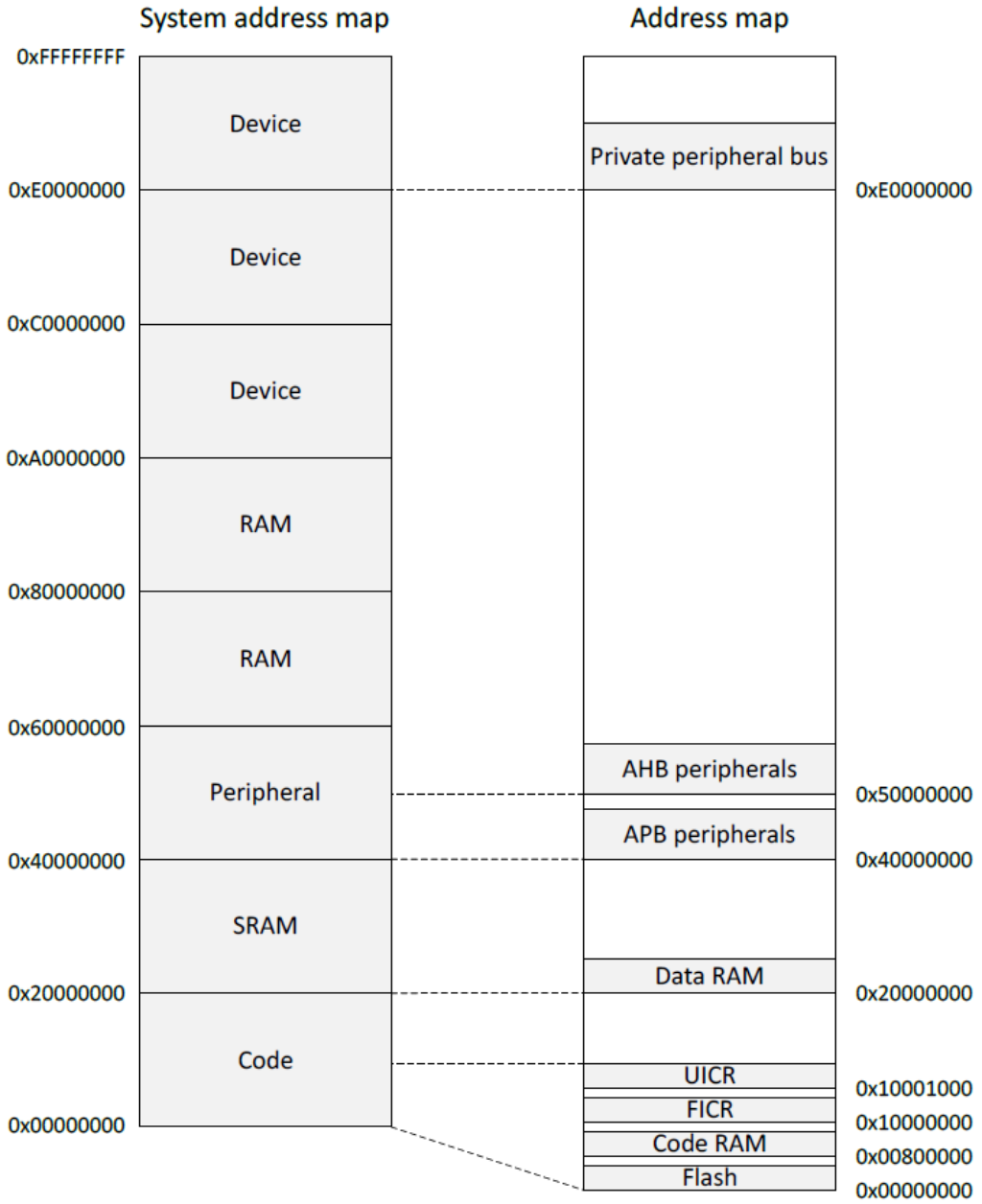


Memory map on nRF52833

- Flash 0x00000000
- SRAM 0x20000000

- APB peripherals 0x40000000
  - Everything but GPIO
- AHB peripherals 0x50000000
  - Just GPIO

- UICR – User Information Config
- FICR – Factory Information Config



Example nRF52 peripheral placement

- 0x1000 is plenty of space for each peripheral
  - 1024 registers, each 32 bits
  - No reason to pack them tighter than that

5	0x40005000	NFCT	NFCT	Near field communication tag
6	0x40006000	GPIOTE	GPIOTE	GPIO tasks and events
7	0x40007000	SAADC	SAADC	Analog to digital converter
8	0x40008000	TIMER	TIMER0	Timer 0
9	0x40009000	TIMER	TIMER1	Timer 1
10	0x4000A000	TIMER	TIMER2	Timer 2
11	0x4000B000	RTC	RTC0	Real-time counter 0
12	0x4000C000	TEMP	TEMP	Temperature sensor
13	0x4000D000	RNG	RNG	Random number generator
14	0x4000E000	ECB	ECB	AES electronic code book (ECB) mode block encryption
15	0x4000F000	AAR	AAR	Accelerated address resolver

Example register layout

PSEL.LED

Address offset: 0x51C

Pin select for LED signal

Bit number				31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
ID				C						BA	AAAA
Reset 0xFFFFFFFF				1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1111	1111
ID	R/W	Field	Value ID	Value		Description					
A	RW	PIN		[0..31]		Pin number					
B	RW	PORT		[0..1]		Port number					
C	RW	CONNECT				Connection					
			Disconnected	1		Disconnect					
			Connected	0		Connect					

- 32-bit value
  - Bits 0-4 are field A
  - Bit 5 is field B
  - Bit 31 is field C
  - Others are unused

- Each field has value ranges and descriptions of what it means

Registers can vary wildly in complexity

OUT

Address offset: 0x504

Write GPIO port

Bit number	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
ID	f e d c	b a Z Y	X W V U	T S R Q	P O N M	L K J I	H G F E	D C B A
Reset 0x00000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0

ID	R/W	Field	Value ID	Value	Description
A	RW	PIN0			Pin 0
			Low	0	Pin driver is low
			High	1	Pin driver is high
B	RW	PIN1			Pin 1
			Low	0	Pin driver is low
			High	1	Pin driver is high
C	RW	PIN2			Pin 2
			Low	0	Pin driver is low
			High	1	Pin driver is high
D	RW	PIN3			Pin 3
			Low	0	Pin driver is low
			High	1	Pin driver is high
E	RW	PIN4			Pin 4
			Low	0	Pin driver is low
			High	1	Pin driver is high
F	RW	PIN5			Pin 5
			Low	0	Pin driver is low
			High	1	Pin driver is high
G	RW	PIN6			Pin 6
			Low	0	Pin driver is low
			High	1	Pin driver is high
H	RW	PIN7			Pin 7
			Low	0	Pin driver is low
			High	1	Pin driver is high

TASKS_START

Address offset: 0x000

Start RTC COUNTER

Bit number	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
ID								A
Reset 0x00000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0

ID	R/W	Field	Value ID	Value	Description
A	W	TASKS_START			Start RTC COUNTER
			Trigger	1	Trigger task

Bit Masking

- How do you manipulate certain bits within a number?

- Use bit manipulation operations
  - ~, &, |, <<, >>

- Steps
  1. Create a “bit mask” which is a pattern to choose certain bits
  2. Use & or | to combine it with your number
  3. Optional: Use >> to move the bits to the least significant position

Bit mask values

- Selecting bits, use the AND operation
  - 1 means to select that bit
  - 0 means to not select that bit

Select bottom four bits:

```
num & 0x0F
```

- Writing bits
  - Writing a one, use the OR operation
    - 1 means to write a one to that position
    - 0 is unchanged

Set 6th bit to one:

```
num | (1 << 6)
num | (0b01000000)
```

  - Writing a zero, use the AND operation
    - 0 means to write a zero to that position
    - 1 is unchanged

Clear 6th bit to zero:

```
num & (~(1 << 6))
num & (~(0b01000000))
num & (0b10111111)
```

Example: selecting bits

- Select bits 2 and 3 from a number

Input: 0b01100100

Mask: 0b00001100

```
  0b01100100
& 0b00001100
------------
  0b00000100
```

Finally, shift right by two to get the values in the least significant position:

0b00000001

In C:

```
result = (input & 0x0C) >> 2;
```

Manipulating a register value

PSEL.LED

Address offset: 0x51C

Pin select for LED signal

Bit number				31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
ID				C						BA	AAAA
Reset 0xFFFFFFFF				1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 11	1111	1111
ID	R/W	Field	Value ID	Value		Description					
A	RW	PIN		[0..31]		Pin number					
B	RW	PORT		[0..1]		Port number					
C	RW	CONNECT				Connection					
			Disconnected	1		Disconnect					
			Connected	0		Connect					

- Check Port value

  value = (REG >> 5) & 1

  or (REG & (1 << 5)) >> 5

- Write port value

  REG |= (1 << 5) // set port to 0

  REG &= ~(1 << 5); // set port to 1

TEMP on nRF52833 example

- Internal temperature sensor
  - 0.25° C resolution
  - Range equivalent to microcontroller IC (-40° to 105° C)
  - Various configurations for the temperature conversion (ignoring)

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

Table 110: Instances

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

https://docs.nordicsemi.com/bundle/ps_nrf52833/page/temp.html

Interacting with the temperature peripheral

TASKS_START

Address offset: 0x000

Start temperature measurement

Bit number	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
ID								A
Reset 0x00000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 00	0000	0000

ID	R/W	Field	Value ID	Value	Description
A	W	TASKS_START			Start temperature measurement
			Trigger	1	Trigger task

EVENTS_DATARDY

Address offset: 0x100

Temperature measurement complete, data ready

Bit number	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
ID								A
Reset 0x00000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 00	0000	0000

ID	R/W	Field	Value ID	Value	Description
A	RW	EVENTS_DATARDY			Temperature measurement complete, data ready
			NotGenerated	0	Event not generated
			Generated	1	Event generated

Reading a temperature value

TEMP

Address offset: 0x508

Temperature in °C (0.25° steps)

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Reset 0x00000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ID	R/W	Field	Value ID	Value	Description
A	R	TEMP			Temperature in °C (0.25° steps) Result of temperature measurement. Die temperature in °C, 2's complement format, 0.25 °C steps. Decision point: DATARDY

- 32-bit value
- 2’s complement (i.e., signed)
- 0.25 °C steps (so 0 = 0°C, 4 = 1°C, etc.)

MMIO addresses for TEMP

- What addresses do we need? (ignore interrupts for now)
  - 0x4000C000 – TASKS_START
  - 0x4000C100 – EVENTS_DATARDY
  - 0x4000C508 - TEMP

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

Table 110: Instances

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

Accessing addresses in C

- What does this C code do?

```
*(uint32_t*)(0x4000C000) = 1;
```

Accessing addresses in C

- What does this C code do?

```
*(uint32_t*)(0x4000C000) = 1;
```

  - 0x4000C000 is cast to a uint32_t*
  - Then dereferenced
  - And we write 1 to it

  - "There are 32-bits of memory at 0x4000C000. Write a 1 there."

Example code

- To the terminal!

- Let’s write it from scratch

Example code (`temp_mmio` app)

```c
// loop forever
while (1) {

  // start a measurement
  *(uint32_t*)(0x4000C000) = 1;

  // wait until ready
  volatile uint32_t ready = *(uint32_t*)(0x4000C100);
  while (!ready) {
    ready = *(uint32_t*)(0x4000C100);
  }

  /* WARNING: we can't write the code this way!
   *  Without `volatile`, the compiler optimizes out the memory access
  while (!*(uint32_t*)(0x4000C100));
  */

  // read data and print it
  volatile int32_t value = *(int32_t*)(0x4000C508);
  float temperature = ((float)value)/4.0;
  printf("Temperature=%f degrees C\n", temperature);

  nrf_delay_ms(1000);
}
```

Using structs to manage MMIO access

- Writing simple C code and access peripherals is great!

- Problems:
  - Need to remember all these long addresses
  - Need to make sure compiler doesn’t stop us!

- Solution:
  - Wrap entire access in a struct!
  - Compilers turn it into the same thing in the end anyways

C structs

- Collection of variables placed together in memory

```
typedef struct {
    uint32_t variable_one;
    uint32_t variable_two;
    uint32_t array[2];
} example_struct_t;
```

- Placement rules - Variables are placed adjacent to each other in memory except:
  - Fields are always aligned to a multiple of their size
  - Padding added to the end to make the total size a multiple of the biggest member

  - Microcontrollers can usually ignore these: all registers are the same size!

Temperature peripheral MMIO struct

```
typedef struct {
    uint32_t TASKS_START;
    uint32_t TASKS_STOP;
    uint32_t _unused_A[62];
    uint32_t EVENTS_DATARDY;
    uint32_t _unused_B[0x204/4 - 1];
    uint32_t INTENSET;
    uint32_t INTENCLR;
    uint32_t _unused_C[(0x508 - 0x308)/4 - 1];
    uint32_t TEMP;
} temp_regs_t;

volatile temp_regs_t* TEMP_REGS = (temp_regs_t*)(0x4000C000);
```

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

With increasingly verbose ways to write the size of the "unused" space (any of these will do, but don't forget the -1)

Temperature peripheral MMIO struct

```
typedef struct {
    uint32_t TASKS_START;
    uint32_t TASKS_STOP;
    uint32_t _unused_A[62];
    uint32_t EVENTS_DATARDY;
    uint32_t _unused_B[0x204/4 - 1];
    uint32_t INTENSET;
    uint32_t INTENCLR;
    uint32_t _unused_C[(0x508 - 0x308)/4 - 1];
    uint32_t TEMP;
} temp_regs_t;

volatile temp_regs_t* TEMP_REGS = (temp_regs_t*)(0x4000C000);

// code to access
TEMP_REGS->TASKS_START = 1;
while (TEMP_REGS->EVENTS_DATARDY == 0);
float temperature = ((float)TEMP_REGS->TEMP)/4.0;
```

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

Using MMIO structs

- Note: structs still don’t get you individual bits, they only get you the 32-bit registers themselves

- You’ll need to do bit manipulations to get the read/write fields you want

- Bit fields are an option in C that can allow access to individual bits, but are generally not used
  - Implementation-specific details for how they actually work
  - What if you need to change multiple fields simultaneously?

Break + Question

- Are binaries portable to other microcontrollers?

Break + Question

- Are binaries portable to other microcontrollers?
  - Definitely not
  - 1) Each microcontroller has its own layout of Flash and RAM, so we might need to put our code in different locations
  - 2) Each microcontroller has its own MMIO addresses and devices
    And every device works at least *slightly* differently
  - Can *sometimes* get away with it for microcontrollers in the same family
    - I.e., class code might work on an nRF52840 instead of our nRF52833

Outline

- Embedded Software Overview
- Embedded Toolchain
  - Lab Software Environment
- Memory-Mapped I/O
- **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

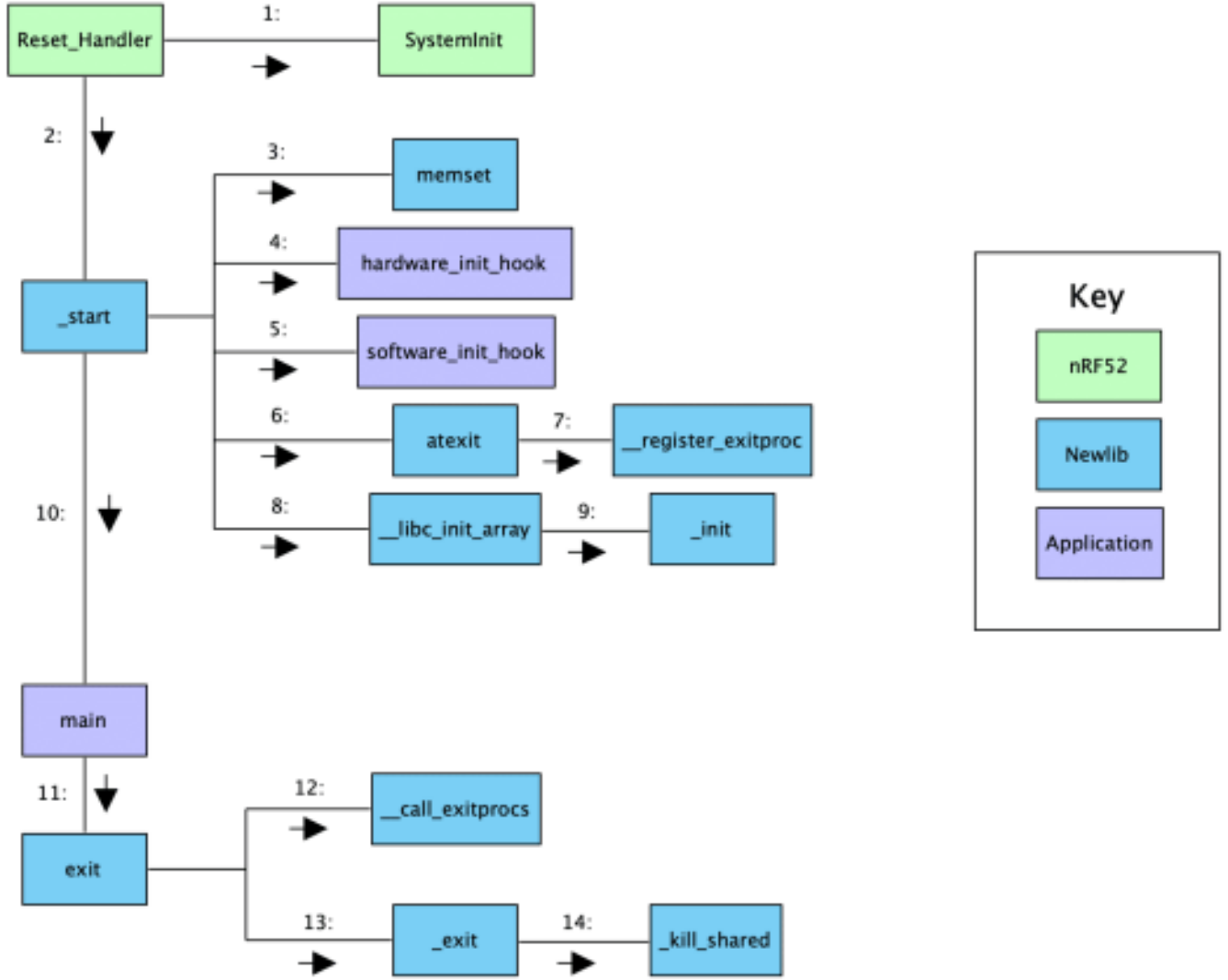
- _start is provided by newlib
  - An implementation of libc – the C standard library
  - Startup is a file usually named crt0

- 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 Overview
- Embedded Toolchain
  - Lab Software Environment
- Memory-Mapped I/O
- Boot Process