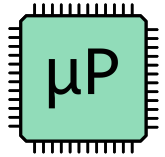
Toolchains and Startup

Lecture 3

Microprocessor-based Systems (E155)

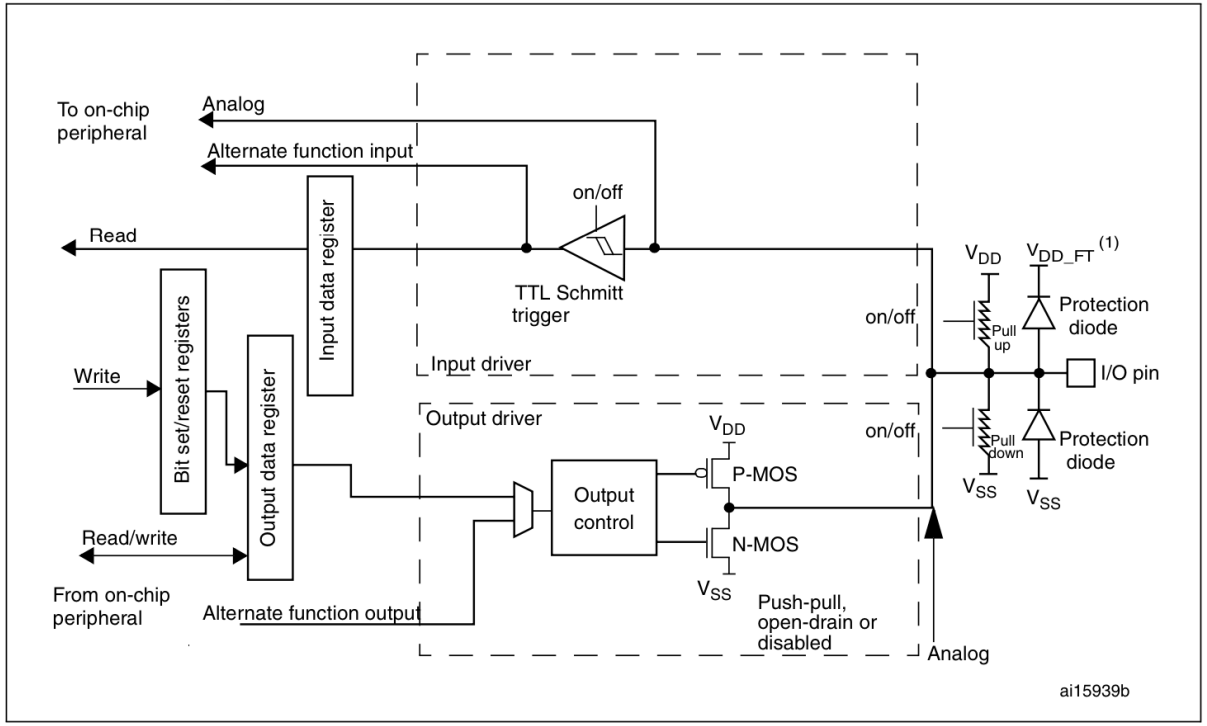
Prof. Josh Brake



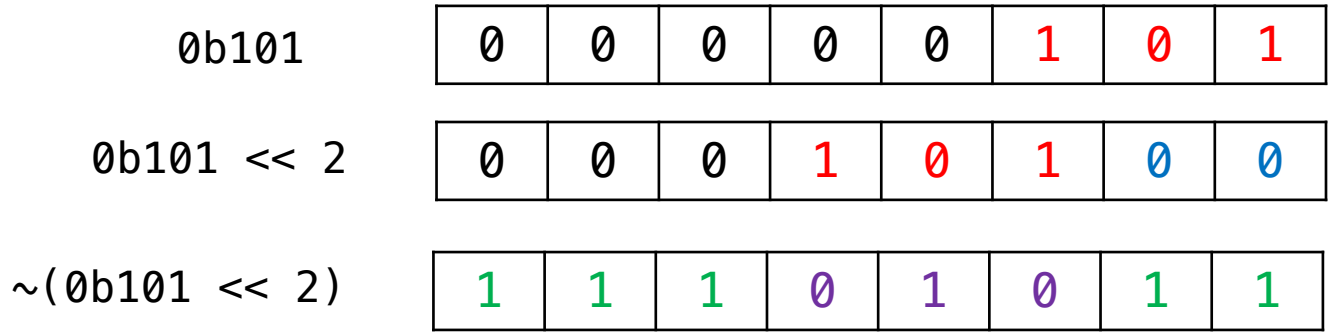
From last time…

- What are interrupts?
- Why would you want to read from output data register?
- Bit shifting
- Accessing struct values
- What are Schmitt Triggers?

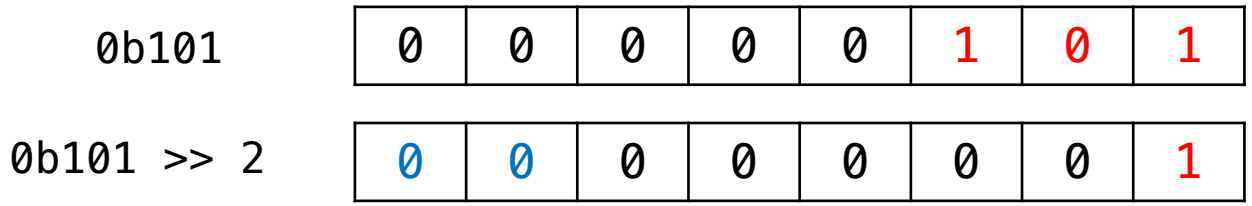
Figure 16. Basic structure of a five-volt tolerant I/O port bit



Bit shifting examples



What about shifting the other way?

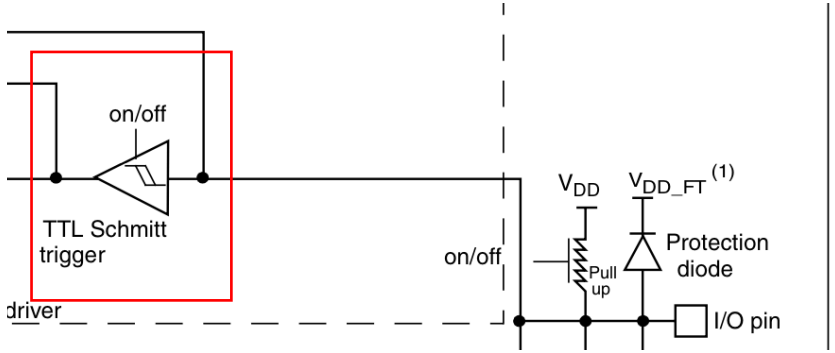


01 are shifted off the end

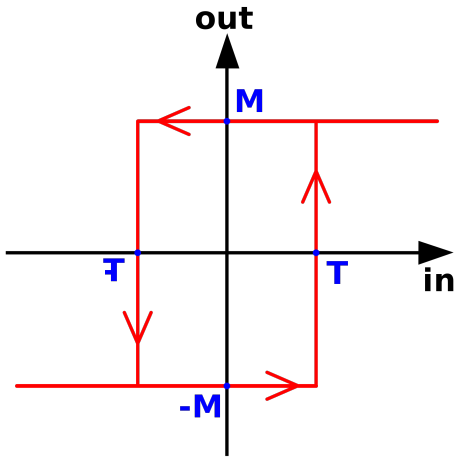
Struct value access review

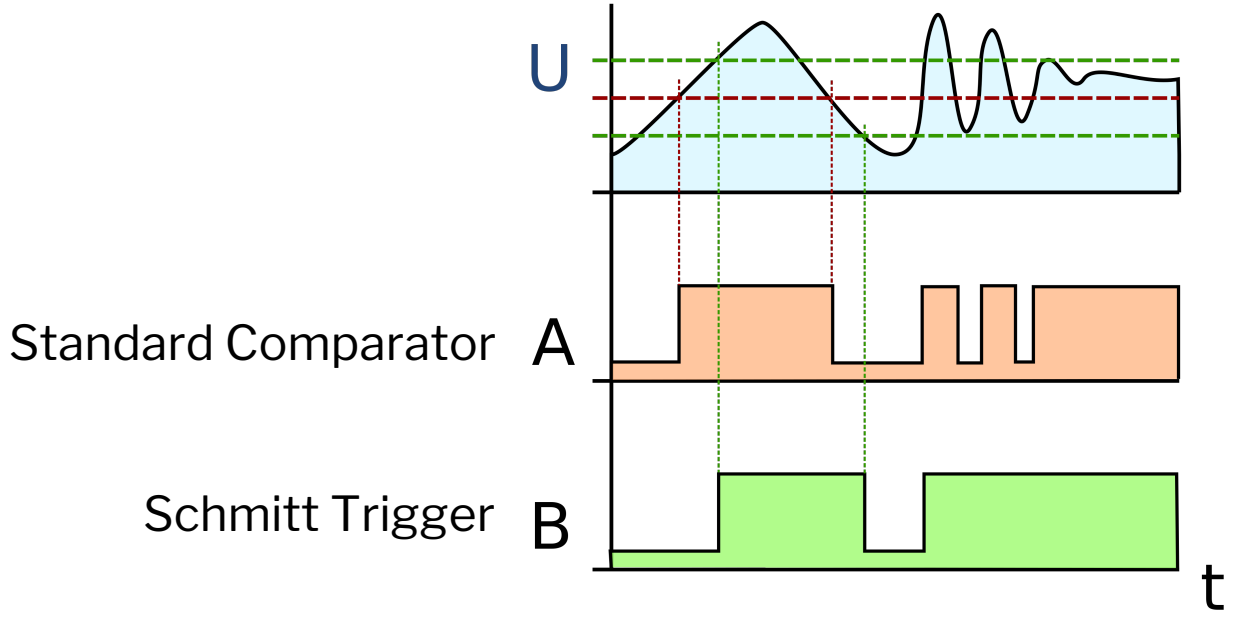
```
typedef struct {
char class_name[50];
int num_students;
} course_info;

// Option 1: Create variable (assigned to arbitrary memory location)
course_info microps;
microps.num_students = 23;

// Option 2: Create pointer to structure at specific memory address
microps_ptr = ((course_info *) 0x40000000);

// One way to access value (dereference pointer and assign member value)
(*microps_ptr).num_students = 23;

// Equivalent but cleaner and preferred way to access value using arrow operator.
microps_ptr->num_students = 23;
```

Schmitt Trigger



STM32F401RE Reference Manual p. 147





"Hysteresis sharp curve" by Alessio Damato CC BY-SA 3.0

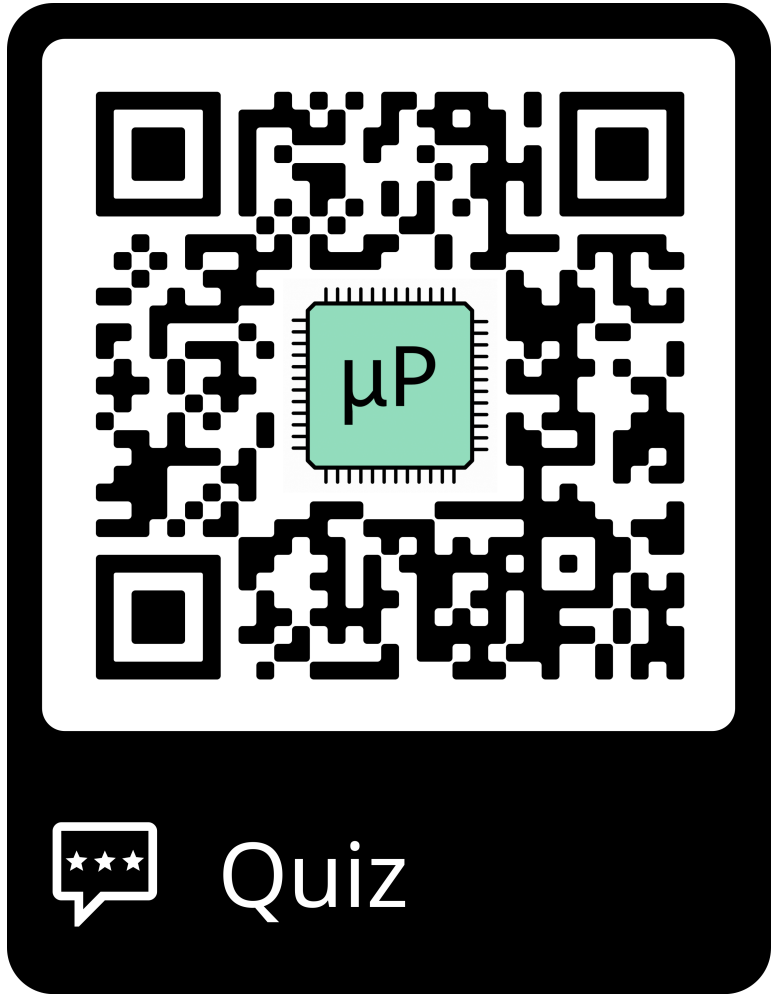
"Smitt (sic) hysteresis graph" by FDominec CC BY-SA 3.0

Main question for today

How do we go from high-level code (e.g., C) to machine code running on our processor?

Google Slides Quiz

- Go to slide corresponding to your breakout room
- Create a text box with your answers to the questions
- The high-tech smoke signal code
  - If you need help or have a question, change the background color of your slide to red
  - When you're done, change the background color of your slide to green
- Report out after ~5 min



Outline

- What is a toolchain?
- The GNU Compiler Collection (gcc) Overview
- The MCU boot process
- Using the GNU toolchain

What is a toolchain?

- A collection of tools which are used in succession to process an input (high-level code) into a binary file which contains machine-encoded instructions that can run on a processor.
- Some common toolchains for embedded development are
  - Keil Microcontroller Development Kit for ARM®
  - IAR Embedded Workbench for ARM®
  - The GNU Compiler Collection (gcc) toolchain
- We are using gcc in MicroPs this year
  - Free and open source
  - Popular with hobbyists and academics
  - Cross platform compatibility

Development workflow with GNU compiler collection (gcc)

gcc includes:

- Preprocessor
- Assembler
- Linker
- Disassembler
- Binary/Hex File Generation
- Standard libraries
- Debugger

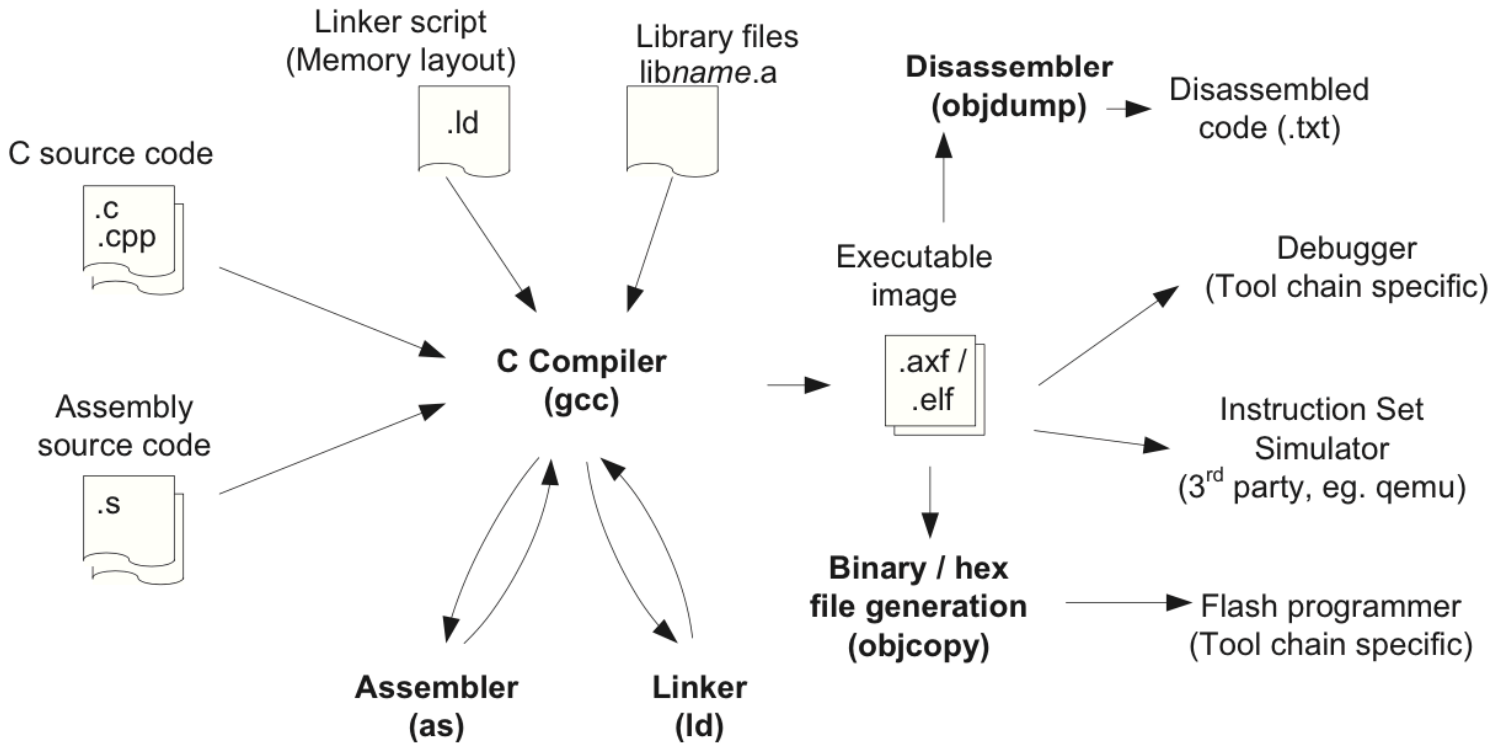


FIGURE 17.1

Typical program development flow

Note: need a separate flash programmer to load and debug executables. We're using openocd.

p. 563 - Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*, 3rd Ed.

Example application to binary flow

How do we get to main?

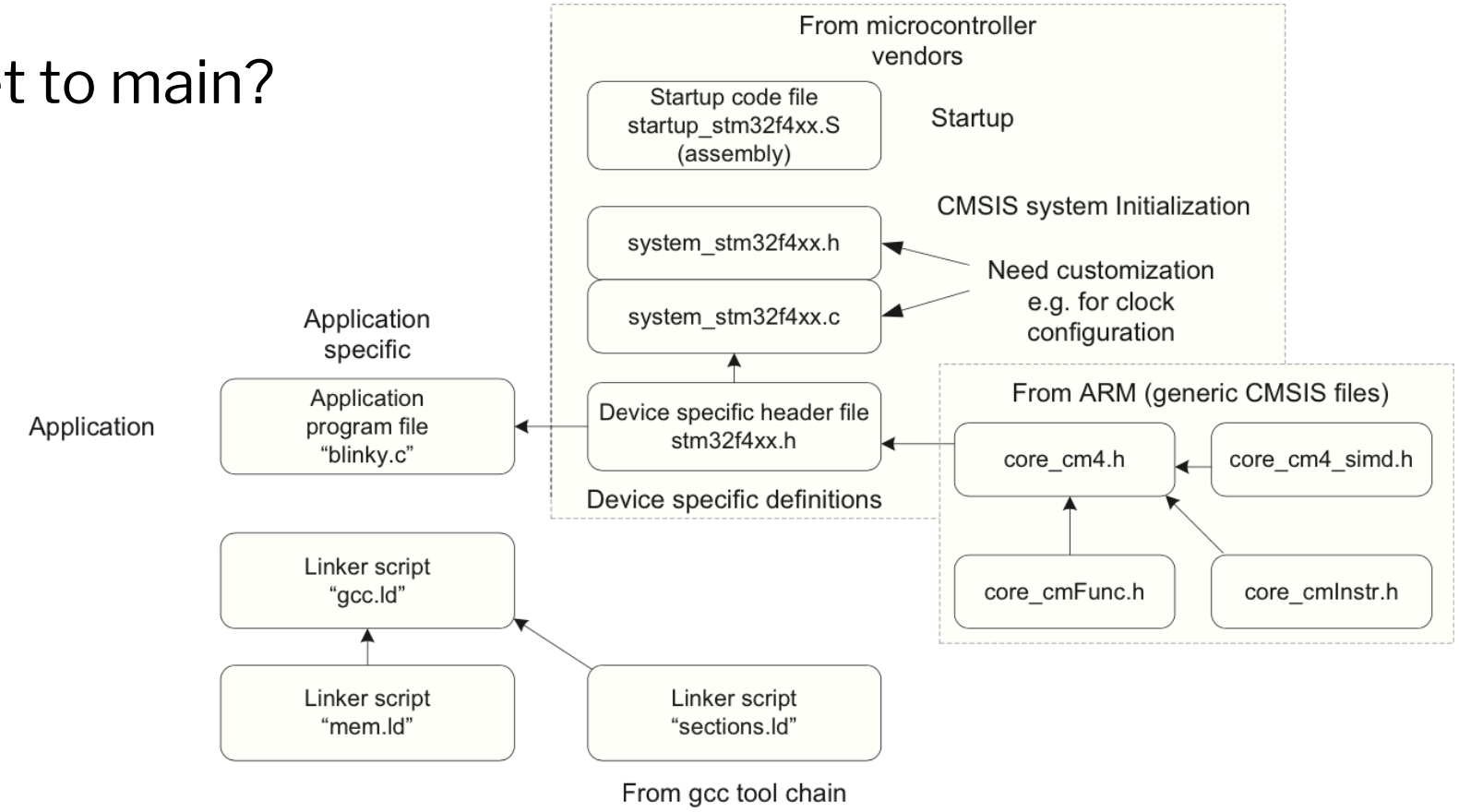


FIGURE 17.2

Example project with CMSIS-Core

p. 564 - Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*, 3rd Ed.

MCU startup procedure

- Well-defined order that must be followed in the vector table.
- This gets put into memory at address `0x00000000`
- First entry is stack pointer initialization
- Next is address of reset vector
- See Yiu chapter 7 for more details

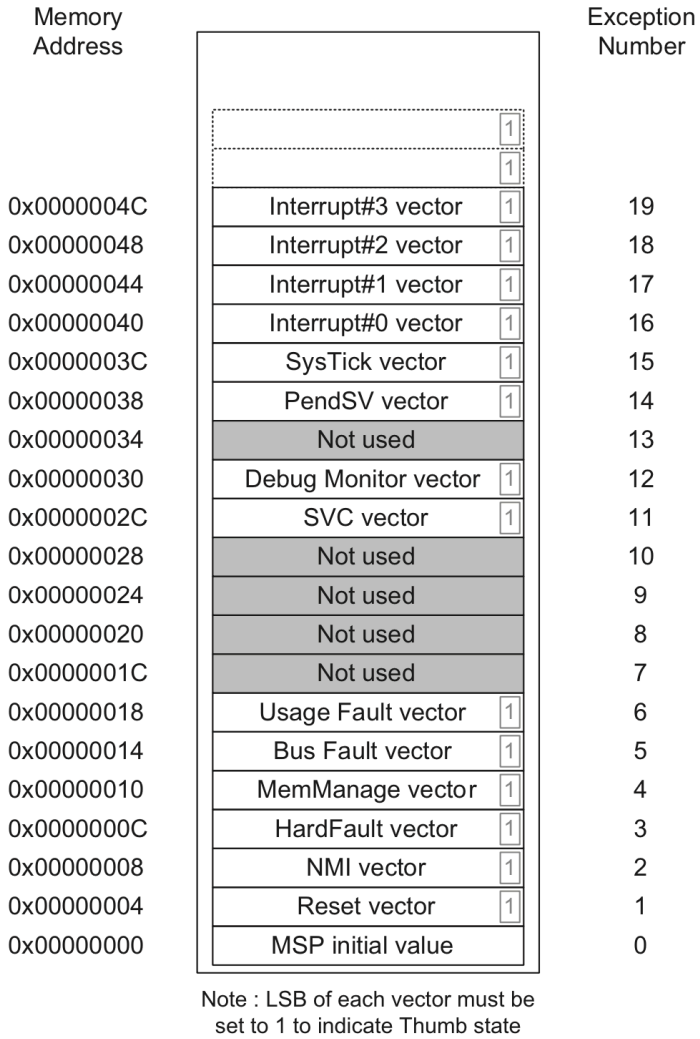


FIGURE 7.10

Vector table

p. 242 - Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*, 3rd Ed.

Reset handler/vector: configuration and globals

```
// core.s
// Simple STM32F401RE startup: vector table plus reset_handler
// Josh Brake
// jbrake@hmc.edu
// 6/17/20

////////////////////////////////////////////////////////////////////////////////
// Assembly configuration
////////////////////////////////////////////////////////////////////////////////

// Define chip attributes and assembly language used

.syntax unified // Unified syntax for ARM and THUMB instructions
.cpu cortex-m4 // STM32F401RE is a Cortex M4
.fpu softvfp // Use software floating-point operations
.thumb // Using thumb assembly

// Global memory locations. Declare as .global to make them available in
// other files.
.global vtable
.global reset_handler
```

Reset handler/vector: Vector table

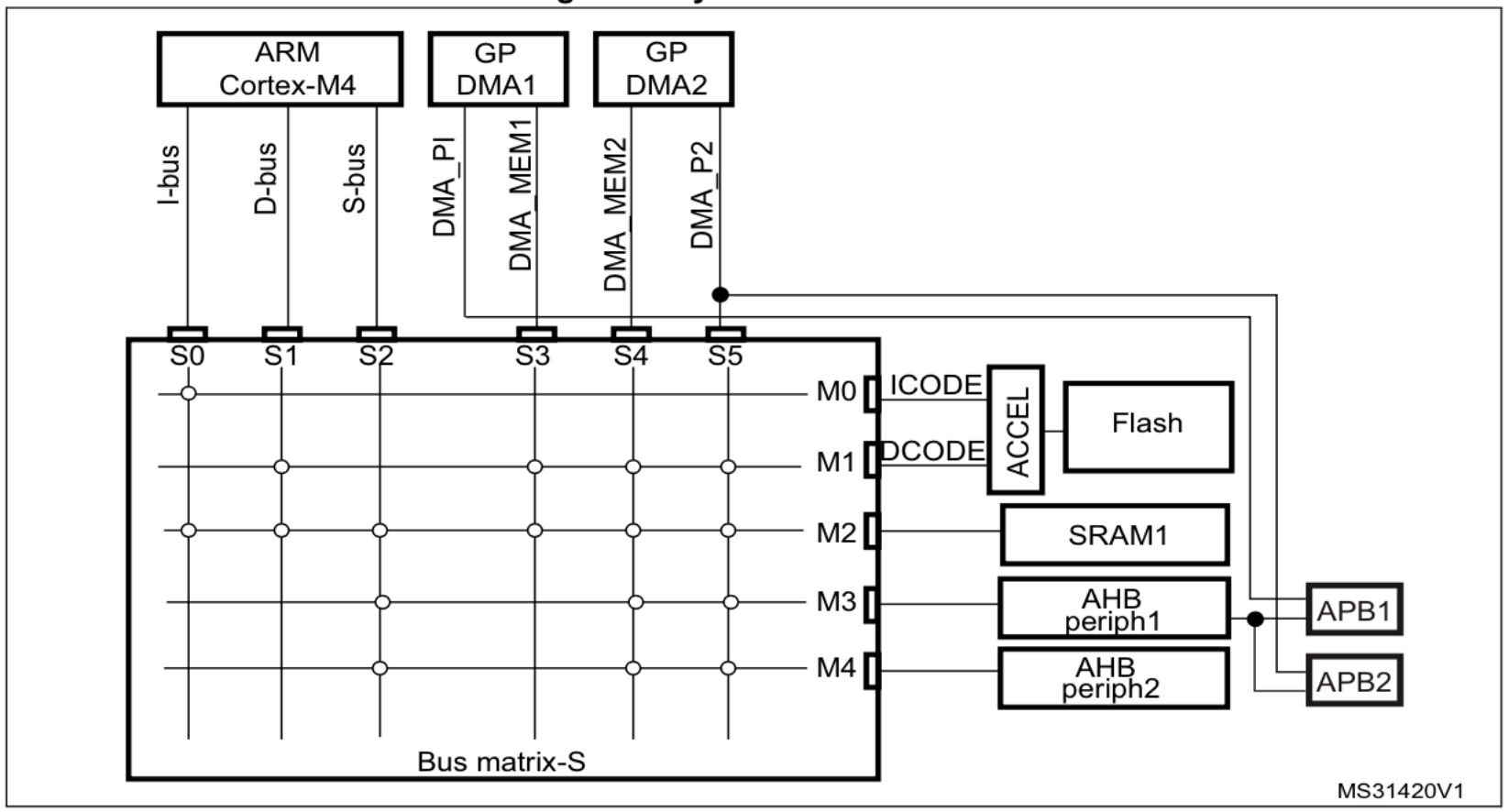
```
////////////////////////////////////////////////////////////////////////////////
// Vector table setup
////////////////////////////////////////////////////////////////////////////////

// The actual vector table.
// Includes only the size of the RAM and 'reset' handler for simplicity

// .type directive tells us what the label points to. Usually either %function or %object
.type vtable, %object

// Label for section
vtable:
// .word places value in 32-bit/4-byte location
.word _estack
.word reset_handler
// .size directive tells assembler how much space that symbol points to
// is using. The "." in the expression ".-<label>" means the current value of
// the location counter
.size vtable, .-vtable
```

Reset handler/vector: Reset handler function

```
////////////////////////////////////////////////////////////////////////////////
// reset_handler
////////////////////////////////////////////////////////////////////////////////

// Define reset_handler
.type reset_handler, %function
reset_handler:
// Set the stack pointer to the end of the stack
// '_estack' is defined in the linker script
LDR r0, =_estack
MOV sp, r0

// Set some dummy values. When we see these values in our debugger, we'll
// know that our program is loaded and working properly on the chip.
LDR r7, =0xDEADBEEF

BL main
BX lr
.size reset_handler, .-reset_handler
```

System architecture

Figure 1. System architecture



1. STM32F401xB/C: 128 KBytes / 256 KBytes Flash with 64 KBytes SRAM.
   STM32F401xD/E: 384 KBytes / 512KBytes Flash with 96 KBytes SRAM.

p. 36 STM32F401RE User Manual

Memory remapping and vector table relocation

- We have multiple spots to put our data
  - Main flash
  - System memory on flash
  - Embedded SRAM
- What are the tradeoffs?

Physical remap in STM32F401xB/C and STM32F401xD/E

Once the boot pins are selected, the application software can modify the memory accessible in the code area (in this way the code can be executed through the ICode bus in place of the System bus). This modification is performed by programming the *Section 7.2.1: SYSCFG memory remap register (SYSCFG_MEMRMP)* in the SYSCFG controller.

The following memories can thus be remapped:

- Main Flash memory
- System memory
- Embedded SRAM1

Table 3. Memory mapping vs. Boot mode/physical remap in STM32F401xB/C

Addresses	Boot/Remap in main Flash memory	Boot/Remap in embedded SRAM	Boot/Remap in System memory
0x2000 0000 - 0x2000 FFFF	SRAM1 (64 KB)	SRAM1 (64 KB)	SRAM1 (64 KB)
0x1FFF 0000 - 0x1FFF 77FF	System memory	System memory	System memory
0x0804 0000 - 0x1FFE FFFF	Reserved	Reserved	Reserved
0x0800 0000 - 0x0803 FFFF	Flash memory	Flash memory	Flash memory
0x0400 000 - 0x07FF FFFF	Reserved	Reserved	Reserved
0x0000 0000 - 0x0003 FFFF(1)	Flash (256 KB) Aliased	SRAM1 (64 KB) Aliased	System memory (30 KB) Aliased

1. Even when aliased in the boot memory space, the related memory is still accessible at its original memory space.

p. 42 STM32F401RE User Manual

Boot configuration

- This is how we control where the vector table is relocated on boot

2.4 Boot configuration

Due to its fixed memory map, the code area starts from address 0x0000 0000 (accessed through the ICode/DCode buses) while the data area (SRAM) starts from address 0x2000 0000 (accessed through the system bus). The Cortex®-M4 with FPU CPU always fetches the reset vector on the ICode bus, which implies to have the boot space available only in the code area (typically, Flash memory). STM32F4xx microcontrollers implement a special mechanism to be able to boot from other memories (like the internal SRAM).

In the STM32F4xx, three different boot modes can be selected through the BOOT[1:0] pins as shown in *Table 2*.

Table 2. Boot modes

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as the boot space
0	1	System memory	System memory is selected as the boot space
1	1	Embedded SRAM	Embedded SRAM is selected as the boot space

The values on the BOOT pins are latched on the 4th rising edge of SYSCLK after a reset. It is up to the user to set the BOOT1 and BOOT0 pins after reset to select the required boot mode.

BOOT0 is a dedicated pin while BOOT1 is shared with a GPIO pin. Once BOOT1 has been sampled, the corresponding GPIO pin is free and can be used for other purposes.

The BOOT pins are also resampled when the device exits the Standby mode. Consequently, they must be kept in the required Boot mode configuration when the device is in the Standby mode. After this startup delay is over, the CPU fetches the top-of-stack value from address 0x0000 0000, then starts code execution from the boot memory starting from 0x0000 0004.

p. 41 STM32F401RE User Manual

Example linker script

```
/* STM32F401RE_simple.ld */
/* Simple linker script for STM32F401RE */
/* Josh Brake */
/* jbrake@hmc.edu */
/* 6/17/20 */

/* Define the end of RAM and limit of stack memory */
/* STM32F401xD/E devices feature 96 KB of SRAM (User Manual RM0368 p.
40)*/
/* STM32F401RE has 512 KB of flash (User Manual RM0368 p. 45) */
/* 96 KB = 0x00018000 */
/* 512 KB = 0x00080000
/* RAM starts at address 0x20000000 */
/* Flash starts at 0x08000000 */
_estack = 0x20018000;

MEMORY
{
FLASH ( rx ) : ORIGIN = 0x08000000, LENGTH = 512K
RAM ( rxw ) : ORIGIN = 0x20000000, LENGTH = 96K
}
```

Output file

```
root@be959ab2072e:/usr/project/src# arm-none-eabi-nm test.elf
20018000 A _estack
08000018 T main
08000008 T reset_handler
08000000 T vtable
```

Label Legend

A: absolute
T: text/code

```
root@be959ab2072e:/usr/project/src# arm-none-eabi-objdump test.elf -x
SYMBOL TABLE:
08000000 l    d  .text 00000000 .text
00000000 l    d  .ARM.attributes 00000000 .ARM.attributes
00000000 l    d  .comment 00000000 .comment
00000000 l    d  .debug_info 00000000 .debug_info
00000000 l    d  .debug_abbrev 00000000 .debug_abbrev
00000000 l    d  .debug_aranges 00000000 .debug_aranges
00000000 l    d  .debug_line 00000000 .debug_line
00000000 l    d  .debug_str 00000000 .debug_str
00000000 l    d  .debug_frame 00000000 .debug_frame
00000000 l    df *ABS* 00000000 core.o
00000000 l    df *ABS* 00000000 main.c
08000008 g     F .text 0000000a reset_handler
08000018 g     F .text 0000000e main
20018000 g       *ABS* 00000000 _estack
08000000 g     O .text 00000008 vtable
```

Label legend

l/g: local or global
d: debug
f: file
F: function
O: object
<space>: normal symbol

Reminders/Questions

- Lab 1
  - Why use Docker?
- General Reminders
  - Sign up for checkoff time if you haven't already.
  - Send me a Slack message if you need a breadboard
  - Remember to submit lab writeups to Sakai before your checkoff (details on what should be included are in the syllabus and specific deliverables are in the report).
- Office hours

Summary

- GNU toolchain enables us to take our C-code and generate binary files which can be uploaded to our devices memory.
- At startup, the PC is initialized to `0x00000000` and begins executing instructions from there.
- The vector table is a well-defined set of specific memory addresses where certain values must be placed by our code. This is taken care of by device-specific startup code.
- Using the tools available in gcc we can investigate our generated binaries to debug any issues and the gnu debugger gdb allows us to step through our code and examine it as it runs.
- Next lecture: ARM Assembly!

Lecture Feedback

- What is the most important thing you learned in class today?
- What point was most unclear from lecture today?