Microprocessors

Fall 2020

5. Embedded C and Toolchain

Tentative Weekly Schedule

- Week x1 - Introduction to Course
- Week x2 - Architecture
- Week x3 - Assembly Language Introduction
- Week x4 - Assembly Language Usage, Memory and Faults
- **Week x5 - Embedded C and Toolchain**
- Week x6 - Interrupts
- Week x7 - Timers
- Week x8 - Modulation
- Week x9 - Serial Communications I
- Week xA - Serial Communications II
- Week xB - Analog Interfacing
- Week xC - DMA
- Week xD - RTOS
- Week xE - Wireless Communications

Review: Memory alignment

- Cortex-M0+ is really picky about memory alignment
  - only 32-bit aligned (**word aligned**) memory accesses are supported
- We need to make sure compiler aligns our code, using `.align` directive.

```
    b .
    nop
    nop

.align
myarray:
    .word 0x1234
    .word 424
    .word 0b0010101011
    .word 12
    .word 42
    .word 0x3FFFFFFF
```

w/o .align

```
 8000080:   e7fe        b.n 8000080 <main+0x28>
 8000082:   46c0        nop         ; (mov r8, r8)
 8000084:   46c0        nop         ; (mov r8, r8)

08000086 <myarray>:
 8000086:   1234        .short  0x1234
 8000088:   01a80000    .word   0x01a80000
 800008c:   00ab0000    .word   0x00ab0000
 8000090:   000c0000    .word   0x000c0000
 8000094:   002a0000    .word   0x002a0000
 8000098:   ffff0000    .word   0xffff0000
 800009c:   00003fff    .word   0x00003fff
```

with .align

```
 8000080:   e7fe        b.n 8000080 <main+0x28>
 8000082:   46c0        nop         ; (mov r8, r8)
 8000084:   46c0        nop         ; (mov r8, r8)
 8000086:   46c0        nop         ; (mov r8, r8)

08000088 <myarray>:
 8000088:   00001234    .word   0x00001234
 800008c:   000001a8    .word   0x000001a8
 8000090:   000000ab    .word   0x000000ab
 8000094:   0000000c    .word   0x0000000c
 8000098:   0000002a    .word   0x0000002a
 800009c:   3fffffff    .word   0x3fffffff
```


Review: Memory layout - Sections

- The program memory is divided into areas for organization.
  - **text, data, bss, heap** and **stack**.
- **text** segment holds the code and usually found in ROM. It is read-only memory (when program is executing) and code can be executed from here.
- **data, bss, heap** and **stack** sections holds the variables, and temporary variables and they are located in RAM.

```
.section .vectors /* place following into .vectors section */
.section .text    /* place following into .text section */
.section .data    /* place following into .data section */
```

- In C these happen automatically, in assembly we need to place things.
- We will see these sections in detail next week.

Review: Section placements

- The program memory is divided into areas for organization.
  - **text, data, bss, heap** and **stack**.
- **How does the compiler know where things go?**

```
MEMORY
{
 ROM  (rx) : ORIGIN = 0x08000000, LENGTH = 64K
 RAM (rwx) : ORIGIN = 0x20000000, LENGTH =  8K
}

ENTRY(Reset_Handler)

/* end of RAM */
_estack = ORIGIN(RAM) + LENGTH(RAM);
```

Linker Script

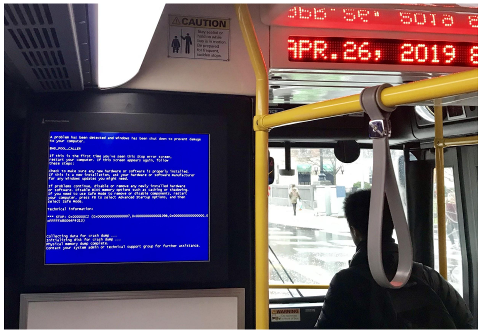
```
SECTIONS
{
  .text :
  {
    KEEP(*(.vectors))
    *(.text*)
  } >ROM

  .data :
  {
    *(.data*)
  } >RAM AT> ROM

  .bss :
  {
    *(.bss*)
  } >RAM
}
```


Review: Faults

- In a processor, if a program goes wrong and the processor detects a **fault**, then a **fault exception** occurs.
- On Cortex M0+ processors, there is only one exception type that handles faults - **HardFault** exception.
- Sometimes there is none available. We will see what to do in those situations.



Embedded C

- Embedded C is **a set of language extensions** for the C programming language by the C Standards Committee to address commonality issues that exist between C extensions for different embedded systems.
- A Technical Report is available in links.
- Mostly the same except Embedded C includes extra features over C, such as **fixed point types**, **multiple memory areas**, and **I/O register mapping**.

Basic C program structure

```c
#include "stm32g0xx.h"
#define LEDDELAY    1600000

int main(void);
void delay(volatile uint32_t);

int main(void) {
    RCC->IOPENR |= (1U << 2);
    GPIOC->MODER &= ~(3U << 2*6);
    GPIOC->MODER |= (1U << 2*6);
    GPIOC->ODR |= (1U << 6);
    while(1) {
        delay(LEDDELAY);
        GPIOC->ODR ^= (1U << 6);
    }
    return 0;
}

void delay(volatile uint32_t s) {
    for(; s>0; s--);
}
```

- Register addresses for specific microcontroller
- Preprocessor directives
- Function declarations
- Bitwise operations
- Infinitive loop
- Function definitions

Bitwise masking operations

```c
/* clear bit 0 of reg A */
A &= 0xFFFE;
/* mask bit 0 of reg A */
A &= 0x0001;
/* set bit 0 of reg A */
A |= ~0xFFFE;
/* clear bit 5 of reg A */
A &= 0xFFDF;
/* mask bit 5 of reg A */
A &= ~0xFFDF;
/* mask bit 5 of reg A */
A &= (1 << 5);
/* clear bit 5 of reg A */
A &= ~(1 << 5);
```


volatile keyword

- When the compiler is asked to optimize code, it will try to remove and dead code - meaning code that does not lead to anywhere.
- This is a problem with embedded systems since we need to write to registers or even execute busy loops.
- For example: `GPIOD->ODR = 0x01;` The compile will think you are writing to some address, but never use that value, so will remove this part. (Similar in busy loops)
- **volatile** keyword is used to tell compilers not to do optimization on that variable, and even read from memory.

weak symbol

- In C, declaring two functions with same name is automatic error
  - This is related to **function signature**.
    - In C the signature of a function is just **function name**.
    - In C++, the signature of the function is the **function name** with **the number of parameters** with **type of parameters** (thus overloading...)
- To make sure to give necessary function calls (mostly for interrupts) and compile basic code, vendors will include a generic function definition with weak keyword. **`__attribute__ ((weak))`** before function name
- If the user wants to add his/her own function that will replace it, compiler(linking stage) will choose the **strong** one.

inline keyword

- In C, declaring a function as inline, we can tell the compiler not generate function properties, and directly try to insert the contents.
  - Purpose is to save from overhead from calling functions
  - just a suggestion, not a requirement
  - **`inline`** before function definition
- Inline semantics will be different for different standards.
  - static / extern keywords need to be prefixed
  - e.g **`static inline int max(..`**
  - e.g **`extern inline int max(..`**

```c
inline int max(int a, int b) {
  if (a > b) return a;
  else return b;
}
```

Address definitions

Since peripherals are memory-mapped I/O, we need a reliable way of accessing these peripherals

#define is used to define the addresses

```
#define RCC_IOPENR_ADDR 0x5000140
#define RCC_IOPRST_ADDR 0x5000124
```

If needed a custom pointer can be defined to point to a given address.

```
volatile uint32_t* rcc_iopenr_data = (uint32_t *)RCC_IOPENR_ADDR;
```


Peripheral Registers

- A typedef struct is used to create register list.
- This struct instance is then associated with the peripheral base address.
- All other peripheral registers can be accessed with arrow operator (**->**)

```
typedef struct {
  volatile uint32_t MODER;
  volatile uint32_t OTYPER;
  volatile uint32_t OSPEEDR;
  ...
} GPIO_TypeDef;
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
```


Remark: Parentheses

- Do not rely on **operator precedence** even if you know it to be correct.
- Always use **parentheses** to separate operations - both arithmetic and logical
- It is easy to forget / mistake / oversee these precedences.

```
int y = a + ++b * c % d;
if ( (y < 10) || (y > 15) ) {
  // do something
}
```


Remark: Language Standards

- Do not rely on default compiler standard to compile your code.
- Always define the standard that you use when compiling.
- Compilers change but standards don't change
- For **C** - C99, **C11**, C17
- For **C++** - C++03, C++11, **C++14**, C++17, C++20

Remark: Understandable names

- Do not have variable or function names like **a, b, c, ,d, asdf** etc.
- It is not readable, and you need to follow the variable to understand what it is.
- Instead give definitive names like, **counter**, **motorControlReady**, **timeoutSteps**, **is_motor_ready()** etc.
- Also keep a convention. For example:
  - For functions use _ instead of space.
  - For variables use capital letters to merge words.
  - This is up to you, choose, and be consistent.

Remark: Keywords usage - static, const

- Understand how variables are declared and used.
- Do not unnecessarily make variables global if they are only used in a single function.
- Utilize **static** keyword to declare all functions and variables that do not need to be visible outside of the code in which they are declared
- Utilize **const** keyword whenever appropriate.
  - if a variable will not be changed after initialization, use it.
  - if a call-by-reference function parameter that should not be modified.

```
void cbr(int const * p)
```


Remark: Keywords usage - volatile

- Utilize **volatile** keyword to avoid mostly optimization generated problems
  - to declare a global variable accessible by any interrupt service routine
  - to declare a global variable accessible by multiple threads
  - to declare a pointer to a memory-mapped I/O peripheral register (`GPIOC_ODR`)
  - to declare a delay loop counter

```
volatile uint32_t* IOENR  = (uint32_t *)(IOENR_ADDR);
void delay(volatile uint32_t s) {
    for(; s>0; s--);
}
```


Remark: Comments

- Always explain what the block does in clear and complete sentences.
- Avoid explaining the obvious.

```
x << 2; // shift x left by 2 bits.
```

- Give proper references if needed in the comment about the algorithm / theorem
- Use automatic documentation tools like Doxygen to generate documentation.
- Utilize **TODO:**, **WARNING:**, and **NOTE:** comment markers to highlight the important bits. Some IDEs automatically show these markers as a general list.

Remark: Code structure

- Make sure to be consistent about **whitespace** usage.

```
for(int i =0; i<10; ++i ){          // ugly
for (int i = 0; i < 10; ++i) {    // better
```

- Although C/C++ does not enforce indentation, always use **proper indentation**.
  - Easier to read, and understand nested structures.
  - **spaces / tabs** are a big debate.
    - Choose one, stick with it.
    - Usually tabs are tricky to handle, so I'd suggest go with spaces, always.

Remark: Equivalence Tests

- When a variable is tested against a constant, placing constant to the left will avoid any possible errors.

```
if (a == 5) {}
if (a = 5) {}
if (5 == a) {}
if (5 = a) {}
```

Which one will produce an error, if any?

Memory for Program

- Eight possible types of information
  - code
  - read-only static data
  - writable static data
  - initialized
  - zero-initialized
  - uninitialized
  - heap
  - stack

```
int a = 21;
char b[8] = {0};
static int c = 14;
char d[4];
const unsigned int e = 1337;

void fun() {
    static int g = 3;
    static int h = 0;
    int j = 0;
    ++g + j + ++h;
}

int main(void) {
    int f;
    int g = 4;
    char h[10];
    fun();
    for(;;);
    return 0;
}
```


General placement principle

- **How long does the data need to exist?**
  - Reuse memory if possible
- **Statically allocated**
  - exists from **program start to end**
  - each variable has its own fixed location
  - space is not reused
- **Automatically allocated**
  - exists from **function start to end**
  - space can be reused
- **Dynamically allocation**
  - exists from **explicit allocation to explicit deallocation**
  - space can be reused

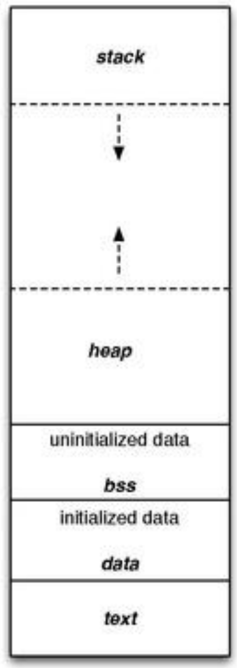

```
int a = 21;
char b[8] = {0};
static int c = 14;
char d[4];
const unsigned int e = 1337;

void fun() {
    static int g = 3;
    static int h = 0;
    int j = 0;
    ++g + j + ++h;
}

int main(void) {
    int f;
    int g = 4;
    char h[10];
    fun();
    for(;;);
    return 0;
}
```

Memory Layout for Program

- **text** segment holds the code and usually found in ROM. It is read-only memory (when program is executing) and code can be executed from here
- **data** segment holds the initialized code and usually found in RAM. It is read-write memory
- **bss** segment holds the uninitialized and zero-initialized code (all initialized to zero)
- **heap** segment holds the dynamically allocated memory segments (**malloc**). Grows towards larger addresses
- **stack** segment holds the local variables and registers for functions as temporary storage area. Grows towards smaller addresses



Where the variables will be placed

- a
- b
- c
- d
- e
- fun::g
- fun::h
- fun::j
- main::f
- main::k
- main::s

```c
int a = 21;
char b[8] = {0};
static int c = 14;
char d[4];
const unsigned int e = 1337;

void fun() {
    static int g = 3;
    static int h = 0;
    int j = 0;
    ++g + j + ++h;
}

int main(void) {
    int f;
    int k = 4;
    char s[10];
    fun();
    for(;;);
    return 0;
}
```


Where the variables will be placed

- **text** segment
  - e - 4 bytes
- **data** segment:
  - a, c, g - 12 bytes
- **bss** segment
  - b, d, h - 16 bytes
- **stack** segment:
  - j (within fun function)
  - f, k, s (within main function)


```c
int a = 21;
char b[8] = {0};
static int c = 14;
char d[4];
const unsigned int e = 1337;

void fun() {
    static int g = 3;
    static int h = 0;
    int j = 0;
    ++g + j + ++h;
}

int main(void) {
    int f;
    int k = 4;
    char s[10];
    fun();
    for(;;);
    return 0;
}
```


Where the variables will be placed

We can also see the locations from the generated `*.list` (`*.lst`) file

```
SYMBOL TABLE:
08000000 l    d  .text  00000000 .text
20000000 l    d  .data  00000000 .data
2000000c l    d  .bss   00000000 .bss
00000000 l    d  .comment   00000000 .comment
00000000 l    d  .ARM.attributes    00000000 .ARM.attributes
00000000 l    df *ABS*  00000000 main.c
20000004 l     O .data  00000004 c
20000008 l     O .data  00000004 g.4160
20000018 l       .bss   00000004 h.4161
20000000 g       .data  00000000 __data_start__
2000000c g     O .bss   00000008 b
080000cc g       .text  00000000 __etext
2000000c g       .bss   00000000 __bss_start__
08000070 g     F .text  0000000e Reset_Handler
08000000 g     O .text  00000010 vector_table
08000084 g     F .text  00000034 fun
2000000c g       .data  00000000 __data_end__
2000001c g       .bss   00000000 __bss_end__
0800007e g     F .text  00000006 Default_Handler
080000b8 g     F .text  00000010 main
20000014 g     O .bss   00000004 d
08000010 g     F .text  00000060 _init_data
20002000 g       .text  00000000 _estack
20000000 g     O .data  00000004 a
080000c8 g     O .text  00000004 e
```


```c
int a = 21;
char b[8] = {0};
static int c = 14;
char d[4];
const unsigned int e = 1337;

void fun() {
    static int g = 3;
    static int h = 0;
    int j = 0;
    ++g + j + ++h;
}

int main(void) {
    int f;
    int k = 4;
    char s[10];
    fun();
    for(;;);
    return 0;
}
```

Activation Record

- Activation records are located on the stack
  - Calling a function creates an activation record
  - Returning from a function deletes the activation record
- Automatic variables and housekeeping information are stored in a function's activation record
- Not all fields may be present for each activation record

Lower address		(Free stack space)	
	Activation record for current function	Local storage	<- Stack ptr
		Return address	
		Arguments	
	Activation record for caller function	Local storage	
		Return address	
		Arguments	
	Activation record for caller's caller function	Local storage	
		Return address	
		Arguments	
Higher address	Activation record for caller's caller's caller function	Local storage	
		Return address	
		Arguments	

C Run-Time Start-Up

After reset, Microcontroller must

- Initialize hardware
  - Stack pointer
  - Peripherals
  - Clock, etc.
- Initialize C / C++ run-time environment
  - Set up heap memory
  - Initialize variables
    - Copy **data** section from the end of ROM to RAM
    - Initialize **bss** section to 0

```c
void Reset_Handler(void) {
    system_init();
    _init_data();
    main();
    for(;;);
}

void _init_data(void) {
    extern unsigned long __etext;
    extern unsigned long __sdata, __edata;
    extern unsigned long __sbss, __ebss;

    unsigned long *src = &__etext;
    unsigned long *dst = &__sdata;

    /* ROM has data at end of text. copy it */
    while (dst < &__edata)
        *dst++ = *src++;

    /* zero bss */
    for (dst = &__sbss; dst< &__ebss; dst++)
        *dst = 0;
}
```


Toolchain

- One of the design challenges is **restricted development environments**.
- Usually, code is designed / developed / compiled in one machine (PC), and run on another machine (MC) called **cross-compiling**
- Compiling the code is usually not enough, and requires an integration of many tools and architecture specific libraries called a **toolchain**
- As a definition, we can state that a **toolchain** is a compilation of programming tools to create software applications.

GNU Toolchain

- GNU GCC Toolchain is a popular toolchain produced by GNU Project.
- ARM is maintaining a GNU toolchain targeted at embedded ARM processors.
- ARM GNU Embedded toolchain includes
  - **gcc** - GNU Compiler Collection
  - **binutils** - a suite of tools including linker, assembler and other tools
  - **gdb** - GNU debugger, a code debugging tool
  - **newlib** - C library for embedded systems (such as stdlib.h, math.h, stdio.h, time.h, ...)

GNU Compiler Collection (gcc)

The **GNU Compiler Collection** (**gcc**) is a collection of open source tools for generating machine code.

- It supports multiple languages such as C, C++ and Fortran
- It has compilers for each of these languages, **gcc** for **C** and **g++** for **C++**
- Supports multiple architectures.
- Prefix of commands reflects the type of the prebuilt toolchain.

GNU Binutils

The **GNU Binutils are a collection of sofware tools.**

- **ld** - the GNU linker
- **as** - the GNU assembler
- **nm** - List symbols from object files
- **objcopy** - Copies and translates object file
- **objdump** - Displays information of object files
- **readelf** - Displays information of any ELF format object file
- **size** - Lists the section sizes of an object file

GNU ARM Toolchain prefixes

Add the prefix **`arm-none-eabi-`** to commands to execute the arm equivalent of the command.

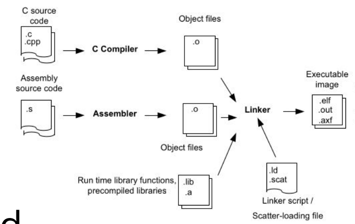
Tools	Generic command name	Command name in GNU Tools for ARM® embedded processors
C Compiler	gcc	arm-none-eabi-gcc
Assembler	as	arm-none-eabi-as
Linker	ld	arm-none-eabi-ld
Binary file generation tool	objcopy	arm-none-eabi-objcopy
Disassembler	objdump	arm-none-eabi-objdump

Application Binary Interface (abi-, eabi-)

- Defines rules which allow separately developed functions to work together
- **ARM Architecture Procedure Call Standard**
  - Which registers must be saved and restored
  - How to call procedures
  - How to return from procedures
- C Library ABI
  - C Library functions
- Run-Time ABI
  - Run-time helper functions
    - 32/32 integer division
    - memory copying
    - floating-point operations
    - data type conversions

Compiling

- Compiling a code means translating the high-level language into machine specific code.
- for GNU GCC and C code, this can be achieved using gcc command (after installation)




```
gcc -o main main.c
```

- `gcc` is the compiler that is invoked
- `-o main` is the flag that we pass to the compiler to define the output executable name.
  - there are various flags for different purposes.
- `main.c` is the source file that we are compiling

Compiler Stages - Preprocessor

- C preprocessor (cpp) is a *macro* processor that is used automatically by the C compiler to transform defined *macros*, **which are brief abbreviations for longer constructs**
- expand **#**'s. (`#include, #define, #if, #ifdef` ..)
- **parenthesis are important**
  - when macros are transformed, **operator precedence** will come into play. e.g. **`COUNT % 15`** will yield **3** vs. **18**
- for GNU GCC: `-E` flag `gcc -E main.c`

```
#include <stdio.h>
#define DELAY 10000
#define COUNT (15 + 18)
#ifndef DEBUG
   #define printf
#endif
```

What do you think will happen when you don't have **`DEBUG`** on your program?

Compiler Stages - Core Compiler

- translates C code from preprocessor to **machine-specific assembly code (.s file)**.
- it is also possible to translate directly into **relocatable binary machine code (.o file)**
- multi-level optimization
- allocates variable uses to registers
- For GNU GCC: `-S` flag

```
gcc -o main.s -S main.c
```

```
    .syntax unified
    .arm
    .fpu softvfp
    .type   main, %function
main:
    @ Function supports interworking.
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1,
uses_anonymous_args = 0
    push    {fp, lr}
    add fp, sp, #4
    ldr r2, .L3
    ldr r3, .L3
    ldr r3, [r3, #52]
    orr r3, r3, #4
    str r3, [r2, #52]
    ldr r2, .L3+4
    ldr r3, .L3+4
```


Compiler Stages - Assembler

- creates **relocatable binary machine code** (object / .o file)
- Mostly a simple **one-to-one mapping** of assembly code
- For GNU GCC: `-c` flag

```
gcc -o main.o -c main.c
```

```
00000000 <main>:
   0:   b580        push    {r7, lr}
   2:   af00        add r7, sp, #0
   4:   4b11        ldr r3, [pc, #68]   ; (4c <main+0x4c>)
   6:   6b5a        ldr r2, [r3, #52]   ; 0x34
   8:   4b10        ldr r3, [pc, #64]   ; (4c <main+0x4c>)
   a:   2104        movs    r1, #4
   c:   430a        orrs    r2, r1
   e:   635a        str r2, [r3, #52]   ; 0x34
  10:   4b0f        ldr r3, [pc, #60]   ; (50 <main+0x50>)
  12:   681a        ldr r2, [r3, #0]
  14:   4b0e        ldr r3, [pc, #56]   ; (50 <main+0x50>)
  16:   490f        ldr r1, [pc, #60]   ; (54 <main+0x54>)
  18:   400a        ands    r2, r1
  1a:   601a        str r2, [r3, #0]
  1c:   4b0c        ldr r3, [pc, #48]   ; (50 <main+0x50>)
  1e:   681a        ldr r2, [r3, #0]
  20:   4b0b        ldr r3, [pc, #44]   ; (50 <main+0x50>)
  22:   2180        movs    r1, #128    ; 0x80
  24:   0149        lsls    r1, r1, #5
  26:   430a        orrs    r2, r1
  28:   601a        str r2, [r3, #0]
```

```
stm32g0/blinky/Debug$ file main.o
main.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV),
with debug_info, not stripped
```

Compiler Stages - Linker

- creates executable image from **one or more object** (.o) files and **libraries** (.a or .so files - **static** or **dynamic**)
- requires each of the source files to be compiled separately
- requires a linker script to know where to place the code

```
gcc main.o banana.o -lm -lc -o banana
```


Linker Map File (Linker Script)

- Contains extensive information on functions and variables
  - value, type, size, object
- Cross references between sections
- Memory map of image
- Sizes of image components
- Summary of memory requirements

```
MEMORY
{
 ROM  (rx) : ORIGIN = 0x08000000, LENGTH = 64K
 RAM (rwx) : ORIGIN = 0x20000000, LENGTH =  8K
}

ENTRY(Reset_Handler)

/* end of RAM */
_estack = ORIGIN(RAM) + LENGTH(RAM);
```

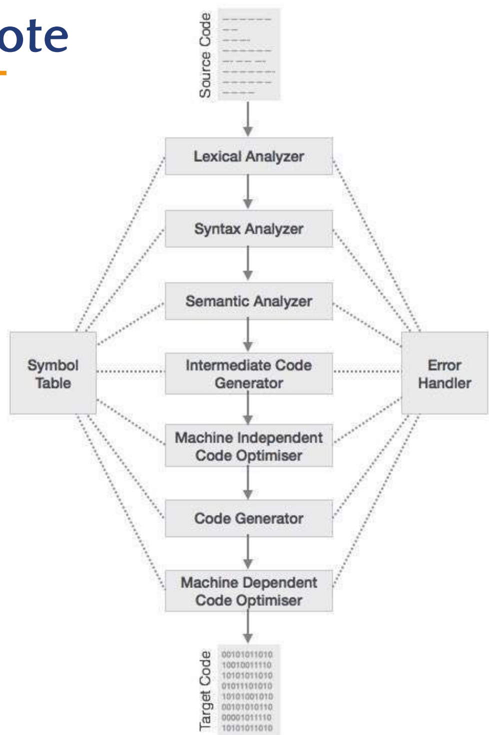
```
SECTIONS
{
  .text :
  {
    KEEP(*(.vectors))
    *(.text*)
  } >ROM

  .data :
  {
    *(.data*)
  } >RAM AT> ROM

  .bss :
  {
    *(.bss*)
  } >RAM AT> ROM
}
```


Compiler Stages - A note

- It is actually a little more complex than that
- But, we don't really care that much about how it is generated
- We mostly care about general steps to **help us understand and fix problems** if / when we face any



Code Optimizations

- Compiler and the rest of the toolchain try to optimize code
  - simplifying operations
  - removing dead code
  - using registers
- These optimizations often get in the way of understanding what the code does
- Compiler optimization levels for GNU GCC
  - `-O0, -O1, -O2, -O3, -Os`

Processor selection

- We talked about the possibility of compiling the code for a different machine (architecture) - **cross-compiling**
- Specific toolchains will include all the necessary header files libraries for target architecture.
- For GNU GCC `-mcpu` flag. `-mcpu=cortex-m0plus`

```
arm-none-eabi-gcc
  -mcpu=cortex-m0plus -mthumb
  -mfloat-abi=soft
```


Warnings / Errors

- Compilers add options to produce warnings when things are looking out of specs.
- Some of these can be overlooked, but some are not.
- For GNU GCC: Warnings can be enabled with `-W` flag

```
uint8_t i = 0;
for(;;)
    if (++i > 255)
        GPIOC->ODR ^= (1U << 6);
```

```
main.c: In function 'main':
main.c:34:15: warning: comparison is always false due to limited
range of data type [-Wtype-limit]
        if (++i > 255) {
                ^
```


Warnings / Errors

- Individual warnings can be enabled, but there are flags that will enabled a batch of them
  - **`-Wall`** for most common warnings
  - **`-Wextra`** for more extra warnings
  - **`-Wpedantic`** for strict ISO C standard
- All warnings can be converted to automatic errors with **`-Werror`** flag.

Overall as a good practice, enable *at least* **all** and **extra** warnings, and make sure your code **does not produce any warnings.**

Directories, Symbols

- Additional directories can be included to be looked for header files with **`-I`** flag.
  `gcc -I../include`
  - Two dots symbolizes parent directory
  - One dot symbolizes current directory
- Additional directories can be included to be looked for libraries when linking with **`-L`** flag. `gcc -L../libs`
- Additional symbols can be defined with **`-D`** flag
  `gcc -DDEBUG`

Additional flags

There are a lot of other helpful flags that we can use. Some of the useful ones include

`-fno-common` give error on duplicate global variable names in different files

`-Wsign-compare` warning on unsigned/signed comparison

`-Wconversion` warning on inherent unsigned/signed conversion

`-ffunction-sections -fdata-sections` place things in separate names in sections

Basic one file + linker project

- We can simply call gcc with specific processor options and linker script on our source file.
- We do not need to separate compiling and linking stages, and can merge them together to single stage.

```
arm-none-eabi-gcc -o target.elf
-mcpu=cortex-m0plus -mthumb -Tstm32g0.ld
-nostdlib target.c
```

Not that bad, but what if we had 3 files and wanted to use a bunch of flags?

Example - blinky project - 3 files

`blinky.c`, `system_stm32g0xx.c`, `startup_stm32g031k8tx.s`

```
arm-none-eabi-gcc -DSTM32G031xx  -mcpu=cortex-m0plus -mthumb  -O0  -std=gnu11  -g -gdwarf-2  -MMD
-MP -MF"Debug/main.d" -MT"Debug/main.d" -fno-common -Wall  -Wextra  -pedantic
-Wmissing-include-dirs -Wsign-compare -Wcast-align -Wconversion  -fsingle-precision-constant
-fomit-frame-pointer  -ffunction-sections -fdata-sections --specs=nano.specs -I../include -c main.c
-o Debug/main.o
```

```
arm-none-eabi-gcc -DSTM32G031xx  -mcpu=cortex-m0plus -mthumb  -O0  -std=gnu11  -g -gdwarf-2  -MMD
-MP -MF"Debug/main.d" -MT"Debug/main.d" -fno-common -Wall  -Wextra  -pedantic
-Wmissing-include-dirs -Wsign-compare -Wcast-align -Wconversion  -fsingle-precision-constant
-fomit-frame-pointer  -ffunction-sections -fdata-sections --specs=nano.specs -I../include -c main.c
-o Debug/main.o
```

```
arm-none-eabi-gcc -DSTM32G031xx  -mcpu=cortex-m0plus -mthumb  -O0  -std=gnu11  -g -gdwarf-2  -MMD
-MP -MF"Debug/system_stm32g0xx.d" -MT"Debug/system_stm32g0xx.d" -fno-common -Wall  -Wextra
-pedantic -Wmissing-include-dirs -Wsign-compare -Wcast-align -Wconversion
-fsingle-precision-constant -fomit-frame-pointer  -ffunction-sections -fdata-sections
--specs=nano.specs -I../include -c ../include/system_stm32g0xx.c -o Debug/system_stm32g0xx.o
```

Linking stage

```
arm-none-eabi-gcc Debug/main.o Debug/system_stm32g0xx.o Debug/startup_stm32g031k8tx.o
-mcpu=cortex-m0plus -mthumb  --specs=nano.specs -Wl,--gc-sections  -Wl,-Map=Debug/blinky.map
-Wl,--cref  -T../linker/STM32G031K8Tx_FLASH.ld   -lc -o Debug/blinky.elf
```


Automating tasks: make utility

- Well, memorizing all these is hard, and typing them is cumbersome. Thankfully there is a way to automate all these tasks with a utility: **make**
- **make** utility automatically determines which pieces of a large program need to be recompiled, and issues commands to re-compile them.
- All the steps can be included and organized in a file called **`makefile`** by defining a set of rules and giving them order.
- Usually IDEs auto generate these makefiles based on your configuration in the background and execute them when you hit compile button.

This week

- Boards should be shipped
- Read Chapter 16 from Yiu
- General reading from the links about toolchain
- Project 1 due on 7th week
- Lab 2 is due on Monday night

Links

- GNU GCC - http://gcc.gnu.org/
- `arm-none-eabi-gcc` - https://launchpad.net/gcc-arm-embedded
- Operator Precendece - https://en.cppreference.com/w/c/language/operator_precedence
- GCC Optimizations - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- Linker manual - https://sourceware.org/binutils/docs/ld/
- Linker script output section - https://sourceware.org/binutils/docs/ld/Output-Section-Attributes.html#Output-Section-Attributes
- GNU Debugger - https://sourceware.org/gdb/current/onlinedocs/gdb
- GNU ARM Toolchain - https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm
- Target processor names - https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/ARM-Options.html#index-mcpu-2
- What belongs to header file - https://www.embedded.com/what-belongs-in-a-header-file/
- Embedded C Technical Report - http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1169.pdf