Foundations of Embedded Systems

Physical Constraints, Data Uncertainty, Low-Level C, RISC-V, and Open-Source FPGA Tools

Draft Version of Lent 2023

Phillip **Stanley-Marbell**

UNIVERSITY OF CAMBRIDGE

4
C Programming for Embedded Systems

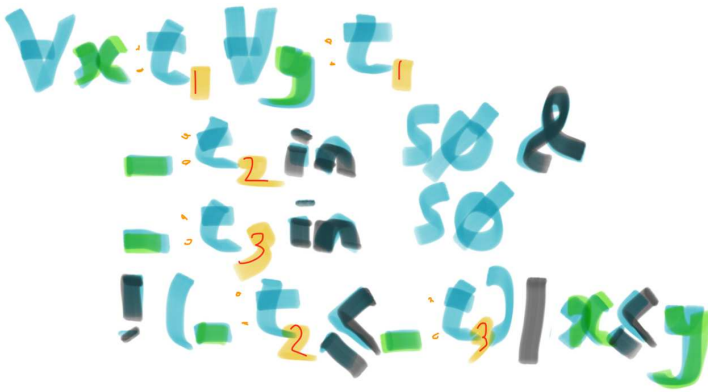


Figure 4.1: **The computational problem specification of sorting** partly expressed in the Sal *set assembly language*.

Use the following formula to turn off the rightmost 1-bit in a word x, producing 0 if none (e.g., **0101 1000** $\longrightarrow$ **0101 0000**):

$$x\&(x-1)$$

—Henry S. Warren Jr., *Hacker's Delight*, Addison-Wesley, 2003.

The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry.

—Henry Petroski.

One of the main causes of the fall of the Roman Empire was that, lacking zero, they had no way to indicate successful termination of their C programs.

—Robert Firth.

Table 4.1: **Concepts.**

Concept	Section
Firmware	§4
Libraries and linking	§4.3,4.19
C pointers	§4.5
C **`struct`**s	§4.6
C arrays	§4.7
Bitwise arithmetic	§4.8
`volatile`	§4.9
Executables	§4.10
Object files	§4.10
The linker, `ld`	§4.10
Linker script files	§4.11
Preprocessor	§4.12
`#include`	§4.12
`crt0.o`	§4.13.2
Memory map	§4.13
Stack	§4.13
Heap	§4.13
Debug information	§4.14,4.16
The function **`main`**	§4.2.14.15.1
The utility `objdump`	§4.15.1,4.16
The utility `objcopy`	§4.15.1,4.16
Memory layout	§4.16
Memory-mapped I/O	§4.17
Makefiles	§4.20

COMPUTATIONAL PROBLEMS relate a set of inputs to a set of outputs. Efficient *algorithms* and *heuristics* define concrete mechanisms by which to solve computational problems and to do so either exactly or approximately.

Programs implement algorithms and heuristics and specify a concrete sequence of operations in the language of some *machine model*. In most computing systems, the machine model you need to understand is usually an abstract machine[1] exposed to you by the operating system.

[1] Engler, Kaashoek, and O'Toole 1995.

In embedded systems, there is typically no abstraction provided to you by an operating system: quite often, there is no operating system at all and you are either writing the *firmware* (software running directly over the processor), or you might be the one responsible for implementing the operating system. The machine model that you must become familiar with in embedded systems is therefore the model defined by the combination of the hardware architecture of the microcontroller or embedded processor in the system and the compiler for the programming language you will be using.

Platforms such as the Arduino (and its variants and derivatives), Raspberry Pi, and the BBC micro:bit are wonderful educational tools for school children and amateur adults who want to be able to read from a sensor or control an actuator. They are great tools for those who do not need to understand the fundamentals of what is going on under the hood. But experience with platforms such as Arduino, Raspberry Pi, or the BBC micro:bit, though fun and easy, leaves you with none of the foundations you need to design the hardware or firmware for a new embedded system from scratch. These are skills which you would need if you wanted to, e.g., implement new instrumentation for an autonomous ground or aerial vehicle, or if you wanted to design a custom embedded system to go into a surgical implant.

This chapter introduces the inner workings of C programs for embedded systems compiled with the GCC compiler and tools from the Binutils suite of binary manipulation tools. The ideas introduced in this chapter also apply to other compilers such as Clang and to compilers in embedded system IDEs such as those from IAR, TI's Code Composer Studio, or Analog Devices' CrossCore tools. The target processor architecture we will use in the examples below will be the Hitachi SuperH RISC (Hitachi SH) architecture and we will use the open-source Sunflower embedded system emulator[2,3] to allow us to interactively explore the execution of programs. The concepts and examples introduced in this Chapter apply largely unchanged to other target processor architectures such as the ARM Cortex-Mo+ in the hardware platform used in this course and to modern architectures such as RISC-V which we will see in Chapter 5.

[2] Stanley-Marbell and Hsiao 2001.
[3] Stanley-Marbell and Marculescu 2007.

4.1 Intended Learning Outcomes

By the end of this chapter, you should be able to:

1. Write C programs that use the basic facilities of the language (e.g., arrays, **`struct`**s, pointers), and language facilities needed in embedded systems programming (e.g., memory-mapped I/O and the **`static`** and **`volatile`** type modifiers).

2. Describe the role of the C preprocessor and explain what happens when a C program includes a header file, e.g., by the directive **`#include <stdio.h>`**.

3. List the steps and tools involved in converting a C-language program into a sequence of instructions for execution on a processor.

4. List the steps involved in executing a binary on a processor.

5. Differentiate between binary file formats such as the Motorola S-record format, ELF, and `a.out` formats.

6. Explain the special role of the function **`main`** in C programs.

7. Design linker script/command files to control where parts of *object files* are placed in the address space of the final *binary* generated by the linker.

8. Explain the difference between *linker script files* and *map files*, explain the contents of a program's map file, and determine what object files have been linked into a binary based on the contents of the map file.

9. Create programs with a mixture of C code and assembly language code.

10. Design C programs which access specific memory-mapped registers, given the corresponding memory addresses defined in a data sheet.

4.1.1 Learning outcomes pre-assessment

Complete the following quiz to evaluate your prior knowledge of the material for this chapter.

4.1.2 Things to think about

Complete the following thinking exercise to stimulate your thoughts about the contents of this chapter before proceeding with the material.

4.1.3 Things to look out for

Concepts people sometimes get confused by in this chapter include:

1. What linker script files do.
2. What interrupts and exceptions are.
3. How interrupt handlers work.
4. How the Sunflower embedded system emulator works.
5. The difference between linker script files and map files.
6. What the statement **`#include`** does in C.
7. The role of the function called **`main`** in a C program.
8. The different roles of the S-record/SREC, ELF, and a.out formats and how all of those relate to linker script files, map files, and so on.
9. What *labels* in assembly language do.
10. The difference between a *symbol* and a *variable*.
11. What is in a `.o` file.

4.1.4 The muddiest point

As you go through the material in this chapter, think about the following two questions and note your responses for yourself or using the annotation tools of the online version of the chapter. You will have the opportunity to submit your responses to these questions at the end of the chapter:

1. What is least clear to you in this chapter? (You can simply list the section numbers or write a few words.)
2. What is most clear to you in this chapter? (You can simply list the section numbers or write a few words.)

4.1.5 Quiz

In the next five minutes, *carefully and sequentially* go through the questions below and mark those which you think you can explain to someone in less than 15 seconds. In particular, make sure you answer Question 50, Question 51, and Question 52.

At the end of the five minutes, pick one of the questions that does not have a simple "yes"/"no" answer. Tell the person sitting closest to you which question it is and wait for them to read that question. Next, explain your answer to them in less than 30 seconds. We will then go round the class and each person will state the question number for one question to which they did *not* know the answer (pick the question whose answer you are most interested in knowing the answer to).

1. What is an algorithm?
2. What is a heuristic?
3. What are the main stages of a processor's pipeline?
4. What is the *instruction set architecture* (ISA) of a processor?
5. What are common instructions you would expect to find in the ISA of any processor?
6. How would you define the term *compiler*?
7. How would you define the term *assembler*?
8. How would you define the term *linker*?
9. What does **`#include <stdio.h>`** in a C program do?
10. Is **`#include`** part of the C language syntax?
11. What about **`main`**?
12. Is **`main`** a reserved keyword in C?
13. What is in an executable?
14. What are the parts of an executable?
15. Does everything in an executable get loaded into the processor to run?
16. What is a binary?
17. What is an object file?
18. How is an object file different from a binary?

19. What are some of the different executable and object file formats?

20. What is the ELF file format?

21. What is the COFF file format?

22. Can you create an object file from a C program without a function called **`main`**?

23. Can you (under normal circumstances) create an executable without a function called **`main`**?

24. What is the difference between an executable and an object file?

25. What does a compiler do to convert a C program into an executable?

26. What is a linker script file?

27. Are linker scripts meant for embedded systems?

28. When you compile using gcc without the **`-c`** flag, does it also use a linker script? If yes what linker script does it use?

29. Does the content of the linker script affect the output of the **`.text`** section?

30. What is the memory map of a program?

31. What is a Von Neumann (Princeton) architecture?

32. What is a Harvard architecture?

33. If a processor uses a Harvard architecture, how does that affect the memory map and linker script files?

34. What is a cache in a processor?

35. Are caches common in microcontrollers? If yes, why? If no, why not?

36. In the context of a program's address space, what is the heap?

37. In the context of a program's address space, what is the stack?

38. What determines how large the stack is?

39. What goes on the stack?

40. Who determines the size of the stack?

41. What goes on the heap?

42. Who determines the size of the heap?

43. What tools can you use to convert between different object file formats?

44. What is a `Makefile`?

45. What is the difference between `make`, `gmake`, `cmake`, and `mk`?

46. What is a `configure` script?

47. What is the *memory map* of an object file or binary?

48. How can you control the memory map of a binary?

49. What command-line tools can you use to inspect the memory map of a binary?

50. Given just a pen and piece of paper, a disassembled program binary, and sufficient time, could you determine exactly what the result of the program will be?

51. What constitutes the "result" of executing a given program binary? In terms of state of a processor, how would you quantify this?

52. Given a program binary (e.g., given an S-record file), could you implement a program that runs on your PC, using any programming language of your choice, which will determine precisely the result of running the S-record file on the processor for which it was compiled? What is such a program called?

53. Would a program such as the one in the question above always produce the same output? Why?

54. What is the length (in characters) of the shortest C program that you can run from a command shell?

55. Is the program below a compilable C program?

```
??=include <stdio.h>
int main(void)??<printf("Hello, world??/n");??>
```

56. Is the program below a valid C program?

```
#include <stdio.h>

int
hello(void) {
    printf("Hello, World.\n");

    return 0;
}
```

57. Could the program above be run from a command shell?

58. If, on an embedded system, your program used **`printf`** to print out a string, what is the complete sequence of steps by which the string reached the terminal / console output?

59. What is the *signature* of a function?

60. What is an interrupt?

61. What is an exception?

62. Are interrupts different from exceptions?

63. What happens when a processor incurs an interrupt or exception?

64. What is an exception handler? What is an interrupt handler?

65. Will the following C program ever terminate? What could cause the program to terminate?

```c
#include "print.h"

/*      8 GPRs + PR     */
unsigned char   REGSAVESTACK[36];
static void     hdlr_install(void);
volatile int    gFlag = 1;

int
main(void) {
    hdlr_install();

    print("\n\n\nStarting...\n\n\n");
    while (gFlag) {
    }
    print("\n\n\nExiting, bye!\n\n\n");

    return 0;
}

void
intr_hdlr(void) {
    gFlag = 0;

    return;
}

void
hdlr_install(void) {
    extern  unsigned char   vec_stub_begin, vec_stub_end;
    unsigned char *         dstptr = (unsigned char *)0x8000600;
    unsigned char *         srcptr = &vec_stub_begin;

    /*  Copy the vector instructions to vector base */
    while (srcptr < &vec_stub_end) {
        *dstptr++ = *srcptr++;
    }

```

```
38      return;
39  }
```

66. In the example above, why is the variable **gFlag** defined as **volatile int gFlag**? What could happen if it was just defined as **int gFlag**?

67. What do you think the following sequence of Hitachi SH assembly instructions does?

```
1       MOVL    r15,    @-r0
2       MOVL    r14,    @-r0
3       MOVL    r13,    @-r0
4       MOVL    r12,    @-r0
5       MOVL    r11,    @-r0
6       MOVL    r10,    @-r0
7       MOVL    r9,     @-r0
8       MOVL    r8,     @-r0
```

68. What do you think the following sequence of Hitachi SH assembly instructions does?

```
1       MOVL    @r0+,   r8
2       MOVL    @r0+,   r9
3       MOVL    @r0+,   r10
4       MOVL    @r0+,   r11
5       MOVL    @r0+,   r12
6       MOVL    @r0+,   r13
7       MOVL    @r0+,   r14
8       MOVL    @r0+,   r15
```

69. What does the following function do?

```
1   unsigned long
2   devexcp_getintevt(void) {
3       return  *(0xFFFFFFD8);
4   }
```

70. Is **sizeof** a library function or a C language operator?

71. Is **strlen** a library function or a C language operator?

72. What is the value of **sizeof("hello")**

73. What is the value of **strlen("hello")**?

74. What does the program `objdump` do?

75. What does the program `objcopy` do?

4.2 *A Quick Introduction to C and Compiling C Programs from the Command Line*

C is a small language[4]. C is straightforward to learn if you know what to prioritize, if you know what information about C matters, and what details of the language can be left to later.

You will first need to know how to compile C programs. On most computing systems[5], you can access a C compiler as the program `cc` from the command line. In some systems, this will be an alias for `gcc`, on others it will be an alias for the `clang` compiler, and so on.

There are three main ways in which you will invoke the C compiler. The first method will be to pass it the flag **`-c`** and a C program. We will cover the other two invocation methods later.

C programs that are meant to be executed (e.g., from the command shell) are different from C programs that are intended for other purposes: An empty program is a valid C program but can only be compiled to an *object file*, not to an *executable*. In the examples below, `shell prompt%` indicates the command line prompt; on your system, it will be something different. Try this out:

```
shell prompt%   echo '' > empty.c
shell prompt%   cat empty.c
shell prompt%   cc -c empty.c
shell prompt%   ls
empty.c empty.o
```

The **`-c`** flag tells the C compiler to compile your C source file into what is called an *object file*. Object files are compiled source code that has not yet been *linked* to create an *executable*. They are not executables *per se* and cannot be run from the command line.

If you want to compile a C program to yield an executable binary, pass it to a C compiler without the **`-c`** flag. In the example below, we also specify the **`-o`** flag to tell the compiler where to store the executable[6]: Try this out:

```
shell prompt%   echo '' > empty.c
shell prompt%   cat empty.c
shell prompt%   cc -o empty empty.c
Undefined symbols for architecture x86_64:
  "_main", referenced from:
     implicit entry/start for main executable
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
shell prompt%   ls
empty.c
```

Clearly, this time, for the same file, the C compiler, `cc`, did not accept the input file `empty.c` even though it generated an object file when given the same file with the **`-c`** flag.

[4] You can find a complete BNF grammar for ISO C in Appendix B of Harbison and Steele.

[5] We will assume a Unix-based system such as Linux, FreeBSD, OpenBSD, or macOS.

[6] Otherwise the compiler places the executable in the default file `a.out`.

Exercise Why do you think the empty C program compiles when compiled with flags **`-c`** to generate an *object file*, but not when compiled to generate an *executable*? The error message should give you a clue. We will see more about this later in this chapter.

4.2.1 The function **`main`**

By convention, every C program that will be run by the command line (i.e., by a shell) needs to have a function called **`main`**. There is some leeway in the convention of what the *signature* of the function **`main`** should look like. The signature of a function is the function's name, its return type, and its argument types (regardless of their names). We will discuss function signatures again later in this Chapter. Two common forms for the function main are:

```c
/*
 *  A function called 'main' taking no argument (the keyword/type 'void')
 *  and returning an integer (the keyword/type 'int').
 */
int
main(void)
{
    return 0;
}
```

and

```c
/*
 *  A function called 'main' taking two arguments. The first
 *  argument, 'argc' is a scalar of type 'int'. The second
 *  argument is an array in which each element has type 'char *'.
 *  The function returns an integer (the keyword/type 'int').
 */
int
main(int argc, char *argv[])
{
    return 0;
}
```

The type specifier **`int`** at the beginning of the program is a C language built-in datatype. In the examples above, it specifies the type of values that the function **`main`** can return. What follows (**`main(int argc, char *argv[])`**) says that the function **`main`** takes two arguments, the first of which is of type **`int`**, and the second of which is of type **`char *`** (and because that second argument ends in **`[]`**, it is an array). This part of the function definition is called the function's signature. Next, the curly braces (**`{}`**) define the *body* or *implementation* of the function.

There is nothing special about the argument variable names **`argc`** and **`argv`**: We could have used any other names in defining the function arguments as long as we are consistent in using those names in the body of the function too. In other words, the signature of a function is about its types and the names of the arguments do not matter (as long as you are consistent between

the function's signature and its body or implementation). That is all you need to write a C program that can be run from the command line.

4.2.2 *A larger example*

The program in the example above does not do much. What if we wanted to create a slightly more sophisticated program, to take all its command-line arguments and to print out the one of its arguments that is a palindrome? That is, we want the program `palindrome` that behaves like:

```
shell prompt%   cc -o palindrome palindrome.c
shell prompt%   ./palindrome the quick brown fox jumped over the madam on the way to the ball

    madam
```

In the example above, to invoke the compiled binary `palindrome`, we typed `./palindrome`. The command shell treats executables specially (to help prevent you from shooting yourself in the foot) and you therefore have to give the explicit path to a binary if it is not already in your shell's *search path*. For a file in the current directory, the explicit path is given by prefixing the file name with "**`./`**".

Back to the task of printing out which of the arguments is a palindrome, we would need to know how to make the program know what its arguments are. That is where the second convention of the form of the function **`main`** comes in. If the signature of your **`main`** function is of the form **`int main(int argc, char *argv[])`**, the shell or whatever program that runs it will pass it the arguments with which the program was invoked, as an array of strings. Using an **`exec`** system call[7], the shell (or equivalent program) will request your program to be executed by the operating system and will set the argument **`argc`** to the number of arguments with which your program was called and will set the array **`argv`** to the strings. One aspect of this convention worth noting is that the list of arguments passed in via **`argv`** (and whose number is in the integer variable **`argc`**) will start with the name of the program itself. So in the case of **`palindrome`**, when invoked from the shell, the string of its first argument will be **`"./palindrome"`**.

[7] You can find a brief synopsis of the **`exec`** system call for POSIX-compliant systems here.

Strings in C are arrays of elements of type **`char`**. In C, you can also refer to any item in memory using a *pointer* and you will often refer to strings as pointers to **`char`** items (i.e., as **`char *`** items). We will learn more about pointers in Section 4.5. Here is a first part of the implementation of `palindrome.c`:

```
#include <stdio.h>
#include <stdbool.h>

bool isPalindrome(char *  string);

int
```

```
main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++)
    {
        if (isPalindrome(argv[i]))
        {
            /*
             *  See the man page for printf() (or Harbison and Steele,
             *  fifth edition, page 387) to find out more about its syntax.
             */
            printf("\n\t%s\n\n", argv[i]);
        }
    }
}
```

The snippet above introduces the **for** construct for looping, the **if** statement for conditional branching, and the **printf** library routine for formatted output. The syntax of **for** and **if** and the behavior of the library routine **printf** you can find in a language reference for the C language. A very good one is Harbison and Steele.[8]

[8] Steele and Harbison 2007.

The snippet above is missing the function **isPalindrome**. If we know the length of a string is n, then a simple way to check if a string is a palindrome is to check if the ith character is equal to the $n - i - 1$th character. The full program is thus:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

bool isPalindrome(char *  s);

int
main(int argc, char *  argv[])
{
    for (int i = 0; i < argc; i++)
    {
        if (isPalindrome(argv[i]))
        {
            /*
             *  See the man page for printf() (or Harbison and Steele,
             *  fifth edition, page 387) to find out more about its syntax.
             */
            printf("\n\t%s\n\n", argv[i]);
        }
    }
}

bool
isPalindrome(char *  s)
{
    int check = 0;

    /*
     *  See the man page for strlen() (or Harbison and Steele,
     *  fifth edition, page 351) to find out more about its syntax.
```

```
     */
    int n = strlen(s);
    for (int i = 0; i < n; i++)
    {
        if (s[i] == s[n - i - 1])
        {
            check++;
        }
    }

    return (check == n);
}
```

Running this program, we get the desired result:

```
shell prompt%   ./palindrome the quick brown fox jumped over the madam on the way
     to the ball

    madam
```

4.3 *Linking*

A linker combines one or more object files into an executable that can be loaded by an operating system. Individual object files may contain references to functions in other object files and one of the tasks of the linker is to make sure that all the cross references between the object files it is linking together are met.

In simplest use, you can provide the linker with a complete list of object files (i.e., `.o` files) to link together to generate an executable. Because multiple object files might together implement some single piece of functionality, you can group together multiple object files into what is known as an *archive*, using the `ar` command. Archives have the file suffix `.a`. Archives are also usually referred to as *libraries*. There are at least two variants of libraries: static libraries and dynamic libraries (also referred to as *shared objects*). The following discussion refers primarily to static libraries.

In common use, the linker takes two kinds of command line flags. The flag `-l` directs the linker to link in an archive (i.e., a library). By convention, when you specify the linker flag `-lABC`, the linker will try to find a file called `libABC.a` and will link the objects in that library with the other objects you have specified such as other libraries or your own object file. The linker will by default search through a number of standard locations in your filesystem for the libraries such as `libABC.a` in the above example. You can explicitly direct the linker to the location of a library specified by the `-l` (lowercase) flag, by specifying a *library search path* using the `-L` (uppercase) flag.

4.4 The Basic C Datatypes

The basic numeric datatypes in the C language are **char**, **short**, **int**, **long**, **long long**, **float**, and **double**.

The sizes of the basic datatypes (on most systems) are 8 bits for **char**, 16 bits for **short**, 32 bits for **int**, 32-bits for **float** and 64 bits for **double**. This is not the always the case on all embedded platforms. On the TI C55x family of DSPs, **float** and **double** are both 32-bit, and **int** is 16 bits.[9] As a result of these variations across platforms, it is better to use the data type aliases defined in the header file **stdint.h**. These define types such as **uint32_t** (always refers to a 32-bit unsigned integer), **int64_t** (a 64-bit signed integer), and so on. The section titled *"Very difficult computers"* on page 187 of the fifth edition of Harbison and Steele[10] gives a few other examples of architectures that are challenging to map C programs to.

[9] Texas Instruments Inc. 2007.

[10] Steele and Harbison 2007.

As we saw before, there is no string datatype *per se* in C: strings are represented in C programs using arrays of **char**. The design of the C programming language chose to allow arrays to be used interchangeably with references to the first elements of the array: these references to memory locations (whether these memory locations contain an array or another datatype) are referred to as pointers.

4.5 Pointers in C

The C programming language allows programs to access arbitrary memory addresses. A pointer in C is a program variable that refers to a specific location in the memory of the computing system. Because pointers allow a program to refer to any memory location, they are used in a broad variety of contexts. However, because pointers allow a program to refer to any memory location, they are also easy to abuse.

C programs declare a variable as a pointer by prefacing the variable with an asterisk (*). So, for example, the following program declares two variables **value** and **pointer**, and uses accesses to the variable **pointer**, which is setup to refer to the memory occupied by the variable **value** using the reference operator, **&**, to change the value of the variable **value**:

```
#include <stdio.h>

/*
 *  For uint32_t
 */
#include <stdint.h>

int
main(int argc, char *argv[])
{
```

```
    uint32_t    value = 100;
    uint32_t *  pointer = &value;

    printf("Value is [%d]\n", value);
    *pointer = 20;
    printf("Value is [%d]\n", value);
}
```

4.6 *Aggregate Data Types and* **`struct`***s in C*

Other than arrays, which are collections of data items of the same type, the C programming language also allows programs to create collections of data items of differing data types. A canonical example is representing information about a person (name, age, gender). The C programming language provides the construct of a **`struct`** to group such data items together.

There are a few different ways to define **`struct`**s in C. We will stick with just one of the ways of defining **`struct`**s. The program below defines a **`struct`** for the personal information example, takes its command line arguments (assuming the arguments are in the order *name*, *age*, *gender*), and populates the fields of the **`struct`** with the information provided on the command line:

```
/*
 *      For the uint64_t
 */
#include <stdint.h>

/*
 *  For the function prototype of printf()
 */
#include <stdio.h>

/*
 *  For the function prototype of strcmp()
 */
#include <string.h>

/*
 *  For the function prototype of strtoul()
 */
#include <stdlib.h>

/*
 *  This defines an "enumeration type" or ‘enum‘.
 *  Enumeration types are useful when you want to
 *  define variables which will take on a small set
 *  of values and where it is is useful to name those
 *  values.
 */
typedef enum
{
    kGenderFemale,
    kGenderMale,
```

```
} Gender;

typedef struct
{
    char *      name;
    uint64_t    age;
    Gender      gender;
} PersonInfo;

int
main(int argc, char *argv[])
{
    PersonInfo  p;
    char        tmp;
    char *      ep = &tmp;

    p.name = argv[1];

    /*
     *  See the man page for strtoul() (or Harbison and Steele,
     *  fifth edition, page 412) to find out more about its syntax.
     */
    p.age = strtoul(argv[2], &ep, 0);

    /*
     *  See the man page for strcmp() (or Harbison and Steele,
     *  fifth edition, page 349) to find out more about its syntax.
     */
    if (!strcmp(argv[3], "female"))
    {
        p.gender = kGenderFemale;
    }
    else
    {
        p.gender = kGenderMale;
    }

    /*
     *  We choose to just print out the struct we just filled:
     */
    printf("Name: [%s], age: [%lu], gender: [%s]\n",
            p.name, p.age,
            (p.gender == kGenderFemale ? "female" : "male"));
}
```

4.7 *Arrays in C*

Arrays in C are collections of data elements of the same type. The example below defines a variable **`numbers`** which is an array of 10 integers:

```
int
main(void)
{
    int numbers[10];

    return 0;
```

```
}
```

Arrays in C are *zero-indexed*: the first element in the array **numbers** is **numbers[0]** and the last (tenth) element in the array is **numbers[9]**. You can initialize the elements of an array when you declare it, and you need not specify initializers for all elements. The elements for which you specify no initializers will have undefined values (may be zero if you are lucky) if the array is defined within a function (i.e., is on the *stack*[11]) and will be initialized with zeros if the array is global. The example below declares a global array of **int**s (**globalNumbers**) and a function-local array of **int**s (**localNumbers**) and initializes the first, second, and last elements in each array:

[11] We will discuss the stack and heap in more detail in Section 4.13.

```
#include <stdio.h>

typedef enum
{
    kArraySize  = 10,
} Sizes;

int globalNumbers[kArraySize] = {
                    [0] =   1123,
                    [1] =   33,
                    [9] =   33,
};

int
main(void)
{
    int localNumbers[kArraySize] = {
                    [0] =   1123,
                    [1] =   33,
                    [9] =   33,
    };

    for (int i = 0; i < kArraySize; i++)
    {
        printf("globalNumbers[%d] = [%d]\n", i, globalNumbers[i]);
        printf("localNumbers[%d] = [%d]\n", i, localNumbers[i]);
    }

    return 0;
}
```

You can define multi-dimensional arrays using additional sets of square braces, such as **int localNumbers2D[kArraySize][kArraySize]**. When defining multi-dimensional arrays with dynamic memory allocation (Section 4.13), you will need to allocate memory for each of the additional dimensions. A common mistake is to mistakenly allocate a single-dimensional array with as many elements as are desired in the multi-dimensional array.

4.8 Bitwise Arithmetic in C

In programming embedded systems, you will often need to extract parts of the bit-level representation of a variable or to combine bit values from different variables into a single word. For example, many sensors provide their readings as individual 8-bit words which you then need to combine to get, e.g., a 10-, 12-, 16-, or 20-bit (or larger) word. For example, the following snippet of code constructs a 10-bit value, to be stored in a 16-bit variable, **`combined`**. The upper 2 bits of the 10-bit value will be taken from the lower 2 bits of **`data[1]`** and the lower 8 bits of the 10-bit value will be taken from **`data[0]`**:

```
1    uint16_t    combined = ((data[1] & 0x3) << 8) | (data[0]);
```

In the example above, the operation **`data[1] & 0x3`** performs a bitwise AND (the operator **`&`**) of the value of **`data[1]`** with the constant **`0x3`**, zeroing out the upper 6 bits but preserving the lower two bits. Because they are used to set or clear parts of a binary word, constants such as **`0x3`** used in this way are called *bit masks*. The expression **`(data[1] & 0x3) « 8)`** combines the masking operation with a left shift (the operator «), by 8 bits, of the result of the masking operation. Finally, the expression **`((data[1] & 0x3) « 8) | (data[0])`** combines the mask and shift with a bitwise OR (the operator **`|`**) to concatenate the contents of **`data[0]`** with the lower two bits of **`data[1]`**. The lower ten bits of the 16-bit variable **`combined`** are therefore from **`data[0]`** and **`data[1]`** and the upper six bits are zero.

4.9 The Keyword `volatile`

When a C compiler processes programs you provide to it, it will often perform many transformations of your program, such as simplifying arithmetic expressions in your program whose outcome can be determined before your program runs, and much more. These transformations are often referred to as *optimizations*, though they may have little to do with mathematical optimization[12] and there is no guarantee their results will be optimal in any sense. As part of these transformations, a compiler may infer that parts of your program compute values that are never used and it may therefore "simplify" your program by omitting instructions for those unused parts of the program.

[12] Bertsekas, Nedi, Ozdaglar, et al. 2003; Boyd and Vandenberghe 2004; Papadimitriou and Steiglitz 1998.

One program transformation that could be troublesome in embedded systems programming is the omission of reads from memory whose result are apparently never used. In the example in Question 65 of Section 4.1.5 above, a compile-time analysis of the program would incorrectly indicate that the variable **`gFlag`** can never be **`0`**. Because an *interrupt* may trigger the execution

of the function **intr_hdlr()**, **gFlag** may indeed get set to **0**. The C language provides the keyword **volatile** as a way for programs to indicate to the compiler that a variable's value may change even if analysis of the code indicates that it should not. Compilers ensure that reads from variables declared with the type modifier **volatile** are always performed.

Applications executing in the absence of an operating system are often constructed as a main event loop, with interrupts handled asynchronously by an *interrupt handler*. In such a situation, you need to ensure that data structures which are modified by interrupt handlers asynchronously do not adversely affect the execution of the main event loop. In such situations, always declare variables to be used to exchange information between the main event loop and the interrupt handler as **volatile**. This ensures that the C compiler will generate code that ensures that variable updates always occur, even when the compiler thinks such updates can be optimized away. If the main event loop is something like the following:

```
int flag;

flag = 0;
while (flag)
{
    print("hello");
}
```

then the compiler might think that since the variable **flag** is never updated in the body of the loop and it can decide not to generate code for the **while** loop in its *dead code elimination* transformation. If the variable **flag** is modified by an interrupt handler, this will however be an incorrect transformation to make. To tell a C compiler that a variable might be changed asynchronously, such a variable must be marked as **volatile**. For example, the following is a corrected implementation of the above:

```
volatile int    flag;

flag = 0;
while (flag)
{
    print("hello");
}
```

4.10 Object Files, Executables, and their File Formats

An object file contains the instructions (code) and data of a given compiled source file. A single object file will not contain all the instructions and data that are needed in an executable, such as the instructions of functions from any statically-linked libraries that it invokes. Object files are created by the compiler, which takes the input source file, converts that to assembly language for a given target processor, and these assembly instructions are then

converted by the assembler into object code. Typically, this distribution of work between what is usually referred to as the *compiler* (converts, e.g., C to assembler) and the *assembler* (converts assembler into object code) is transparent to the user. And some compiler implementations might fuse the implementations of these two components into the same program.

An executable file is created from one or more object files which are combined using the *linker*. Again, even though the process of linking together one or more object files is logically distinct from compilation to assembly or generation of object files from assembly, the process of linking also happens transparently behind the scenes if you are compiling a single source file to an executable.

In many compiler implementations, the command you invoke (e.g., `gcc`) is often what is called the *compiler driver program* (not to be confused with device drivers). This *front-end* or *driver* program invokes the necessary tools to perform the compilation, assembly, and linking. To see these intermediate steps happening, you can invoke the compiler driver with an appropriate flag to request it to be verbose. For `sh-elf-gcc`, this is the `-v` flag. We can see more details when we invoke `sh-elf-gcc` with the **`-v`** flag:

```
shell prompt% sh-elf-gcc -v simple.c -o simple

Using built-in specs.
COLLECT_GCC=sh-elf-gcc
COLLECT_LTO_WRAPPER=/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/libexec/gcc/sh-elf/8.2.0/lto-wrapper
Target: sh-elf
Configured with: ../configure --disable-docs --target=sh-elf --prefix=/usr/local/src/git/sunflower-simulator-github-clone/tools/superH --disable-libssp --with-gnu-as --with-gnu-ld --with-newlib --enable-languages=c --disable-multilib --disable-libssp --disable-libstdcxx-pch --disable-libmudflap --disable-libgomp --with-headers=/usr/local/src/git/sunflower-simulator-github-clone/tools/source/newlib-2.5.0.20170922/newlib/libc/include --with-gmp=/opt/local --with-mpfr=/opt/local --with-mpc=/opt/local --with-libiconv-prefix=/opt/local -v
Thread model: single
gcc version 8.2.0 (GCC)
COLLECT_GCC_OPTIONS='-v' '-o' 'simple'
 /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/libexec/gcc/sh-elf/8.2.0/cc1 -quiet -v simple.c -quiet -dumpbase simple.c -auxbase simple -version -o /tmp/ccpqQ3U8.s
GNU C17 (GCC) version 8.2.0 (sh-elf)
    compiled by GNU C version 5.4.0 20160609, GMP version 6.1.0, MPFR version 3.1.4, MPC version 1.0.3, isl version none
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/include
 /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/include-fixed
 /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/../../../../sh-elf/sys-include
 /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/../../../../sh-elf/include
End of search list.
GNU C17 (GCC) version 8.2.0 (sh-elf)
```

```
    compiled by GNU C version 5.4.0 20160609, GMP version 6.1.0, MPFR version 3.1.4, MPC version 1.0.3, isl version none
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: 3645f49a73a1199f14dd83119fb2d30f
COLLECT_GCC_OPTIONS='-v' '-o' 'simple'
 /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/../../../../sh-elf/bin/as -v -big -o /tmp/ccEwASy4.o /tmp/ccpqQ3U8.s
GNU assembler version 2.29.1 (sh-elf) using BFD version (GNU Binutils) 2.29.1
COMPILER_PATH=/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/libexec/gcc/sh-elf/8.2.0/:/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/libexec/gcc/sh-elf/8.2.0/:/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/libexec/gcc/sh-elf/:/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/:/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/:/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/../../../../sh-elf/bin/
LIBRARY_PATH=/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/:/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/../../../../sh-elf/lib/
COLLECT_GCC_OPTIONS='-v' '-o' 'simple'
 /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/libexec/gcc/sh-elf/8.2.0/collect2 -m shelf -o simple /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/crt1.o /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/crti.o /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/crtbegin.o -L/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0 -L/usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/../../../../sh-elf/lib /tmp/ccEwASy4.o -lgcc -lgcc-Os-4-200 -lc -lgcc -lgcc-Os-4-200 /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/crtend.o /usr/local/src/git/sunflower-simulator-github-clone/tools/superH/lib/gcc/sh-elf/8.2.0/crtn.o
COLLECT_GCC_OPTIONS='-v' '-o' 'simple'
```

In the GCC open source tools, the assembler and linker have traditionally been part of a separate open-source project called *Binutils*.

Executables are, as their names suggest, files that can be executed (e.g., from the command line). As such, they need to obey certain conventions so that the executor (typically the shell or some other program that invokes an **`exec`** system call) can execute them. For example, where in the stream of bytes in an executable do instructions start? Where are constants stored? How can constants be distinguished from instructions? and so on. On POSIX-compliant operating systems, these conventions are defined by the **`exec`** system call.

Every executable has at least three main sections: the **`.text`**, **`.data`**, and **`.bss`** sections. These sections are sometimes also referred to as the **`text`**, **`data`**, and **`bss`** sections or the **`TEXT`**, **`DATA`**, and **`BSS`** sections. The **`.text`** section contains executable instructions, the **`.data`** section contains initialized data (e.g., global constants), and the **`.bss`** section contain uninitialized data[13]. The basic task of a program or operating system that loads and runs an executable is to load all three sections into the memory of a processor, (optionally) zero out the **`.bss`** section, and then jump to the beginning of the **`.text`** section and start executing the instructions encountered there.

[13] BSS is a historical term that stands for *block started by symbol*.

There are several file formats for object files and executables. These file formats define how, e.g., the **`.text`**, **`.data`**, and **`.bss`** sections can be found

in an object file. They typically also contain additional information such as information needed by debuggers. Examples of some object file formats include the ELF format, the COFF format, and the a.out format.

In addition to these object file formats, there are several (simpler) file formats used in embedded systems to represent sequences of bytes that can be loaded directly into a processor's memory for execution. These formats include the *Motorola S-record format* (SREC), and the *Hex* format. S-record files are human-readable text files. Each line in an S-record file begins with an **S** (hence the name) followed by a number of characters indicating the type of information encoded in the line, the memory address to which any subsequent data field should be loaded, and a data field, all encoded as plaintext[14]. As a result of this simple line-based text structure, S-record files can be decoded incrementally by a very small program (e.g., a *bootloader*) running on a microcontroller.

[14] You can find a succinct illustration of the S-record format here.

4.11 Linker Command / Script Files

Most linkers allow you to specify what is usually referred to as a *linker script file* or *linker command file*. The linker script file provides the linker with instructions on how to layout the **.text**, **.data**, and **.bss** sections in memory (e.g., at what memory addresses and how any alignment on word boundaries should be enforced). The listing below shows a basic linker script file:

```
OUTPUT_FORMAT("elf32-littlearm")
SECTIONS
{
    .text . :
    {
            _text = . ;
            *(.text)
            *(.strings)
            _etext = . ;
    }
    .data . :
    {
            _data = . ;
            *(.data)
            _edata = . ;
    }
    .bss . :
    {
            _bss = . ;
            *(.bss)
            *(COMMON)
            _ebss = . ;
            _end = . ;
    }
}
```

This linker script instructs the linker to place the **.text**, **.data**, and **.bss**

sections in the binary in that order. The linker script also instructs the linker to insert the symbols **_text**, **_etext**, **_data**, **_edata**, **_bss**, and **_ebss** at the start and end of the **.text**, **.data**, and **.bss** sections, respectively. These symbols are useful because they can be referred to by a C program linked using this linker script file, thereby allowing a program to introspect its own memory layout. Such a capability is invaluable for programs running directly over the processor without an operating system and is also invaluable in implementing embedded operating systems of your own.

On line 6, the linker script instructs the linker to place the symbol **_text** at the start the start of the **.text** section[15] so that the address of the symbol (which can be taken by a C program using the construct **&text**) will indicate the start of the **.text** section. Similarly, on line 9, the linker script instructs the linker to place the symbol **_etext** at the end of the **.text** section. In both cases, the linker script gives the location implicitly as "**.**"[16].

For example, a C program can take the address of the symbol **bss** to obtain the memory address of the start of the BSS memory section. Because the BSS memory section holds uninitialized data, it is often desired to zero it out at firmware initialization. Being able to know the start and end of the BSS memory section makes this possible. The following routine, taken from an implementation of an embedded operating system, zeroes out the BSS memory section:

[15] That is, before all the symbols destined for the **.text** section, indicated on line 7 with ***(.text)**.

[16] ".", in the world of Unix (and computing in general), is often used as a shorthand to indicate "here".

```
1  void
2  bssinit(void)
3  {
4      long *dst;
5
6      for (dst=&bss; dst<&ebss; dst++)
7      {
8          *dst = 0;
9      }
10
11     return;
12 }
```

The example above illustrates one important peculiarity of referring to symbols defined by the linker or assembler, from C: to reference symbols defined in the assembler or linker from C, those symbols must have been defined with a leading underscore and you reference them by removing one leading underscore. Thus, in the example above, the symbol **_ebss** defined in the linker script file is referenced in C code as **ebss**. We will discuss linking to assembly sources and symbol referencing between C and assembler again in Section 4.19.

4.12 What `#include` Does: the Preprocessor

The **`#include`** statement is not a part of the C language. Rather, it is what is known as a *preprocessor directive*. In traditional C compilers, before the compiler received code to be compiled, that code would have been processed by a separate program, the C preprocessor (`cpp`). The C preprocessor recognizes a number of directives such as **`#include`**, **`#define`**, **`#ifdef`**, and so on. The task performed by the C preprocessor is to perform *textual replacement* of these preprocessor directives: a **`#include`** statement is replaced with the text contained in the file referenced by the **`#include`** statement, a **`#define`** statement causes the preprocessor to textually substitute occurrences of the string defined with the string's definition given in the second part of the **`#define`** statement, and so on.

It is accepted convention within the circles of people who created the C programming language, those who have worked closely with the language's creators, and the like, that **`#include`** statements should only be used to include header files (`.h` files). Similarly, it is convention that header files should in turn only contain data structure definitions and function prototypes and should never contain the definitions of function bodies or complete programs. Many of the tools and libraries that process C programs implicitly assume this convention is kept.

4.13 Memory Map, Stack, Heap, and Dynamic Memory Allocation

During program execution, data in a program (e.g., variables, arrays, structures) reside either in the processor's registers (if they fit within the processor's *word size* and if there are unoccupied registers available) or in memory (RAM).

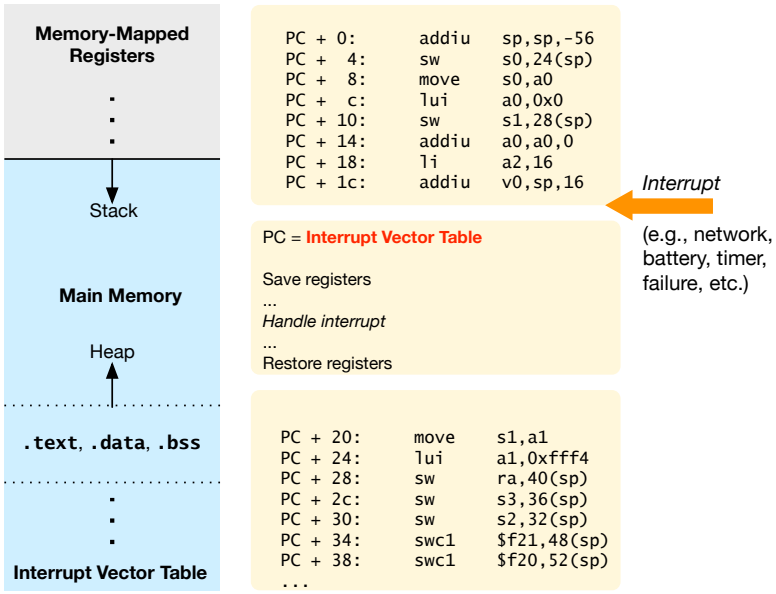
Most microcontrollers used in embedded systems have some amount of RAM integrated into the processor. Usually, this on-chip RAM is static random-access memory (SRAM). For example, the KL03 microcontroller used in this course has 2 kB of SRAM and the KL05 microcontroller used in this course has 4 kB of SRAM. The KL03 and KL05 also have 32 kB of non-volatile Flash memory which we will use primarily for code storage. In this course, we will refer to this non-volatile flash memory simply as "Flash" and will refer to the volatile SRAM simply as "RAM" or "memory".

Processors and microcontrollers typically reserve regions of memory for a number of distinct purposes. We will refer to the logical way in which the set of memory addresses that are available given the amount of memory in a processor, as the *memory map*.

The memory map of a program indicates where the program is located in memory when loaded and how the different regions of memory accessible

by a program are laid out. Typically, this is just an extension of the layout indicated by the linker script file implicitly used by the linker or explicitly specified by a programmer in compiling a program.

Figure 4.2 shows a generic memory map. Figure 4.3 illustrates the memory map of the Hitachi SH version of the architecture modeled by the Sunflower emulator. The base of the address space is at memory address **0x0800 0000**. The region of memory beginning at address **0x0800 0600** contains the *interrupt vector base*. On the occurrence of an *interrupt* (a hardware-generated exceptional condition) or *exception* (a software-generated exceptional condition), the processor starts fetching instructions at this address, and code in this region of memory is executed. Code at the interrupt vector base address must perform necessary saving of register state, determine the actual cause of the exceptional condition (i.e., the type of interrupt or exception raised) and call the appropriate routines to handle the condition. The type of interrupt or exception is determined by reading the **EXCP_INTEVT** or **EXCP_EXPEVT** *memory mapped registers* in the Hitachi SH architecture, respectively.



(a) Example memory map.

(b) Example flow of instruction execution in the presence of interrupts from modeled peripherals.

Figure 4.2: A generic memory map.

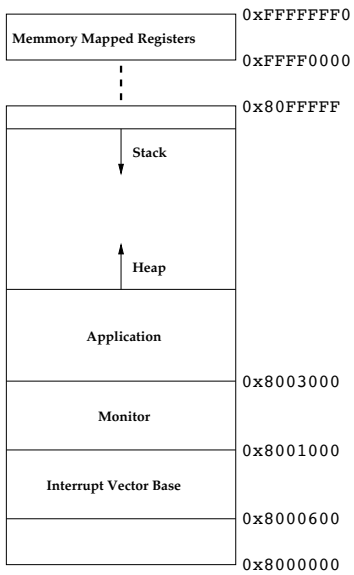


Figure 4.3: Example memory map: Memory map of the Hitachi SH machine model in the Sunflower simulator.

Memory-mapped registers in the Hitachi SH are mapped to a separate region of memory starting at **0xFFFF 0000** and ending at **0xFFFF FFF0**. The manner in which such memory mapped registers are accessed is described in Section 4.17.

The region of memory from **0x0800 1000** upwards is used for application memory. In Figure 4.3, the region of memory above **0x0800 3000** is used

to hold general application code, followed by the application heap (growing upwards from the end of the application code) and the stack growing downwards from the top of memory. The region of memory occupied by an application, is further broken down into regions for code (**`text`**), initialized data (**`data`**) and uninitialized data or **`bss`**.

4.13.1 The stack and the heap

During execution, in addition to its machine registers, a processor executing a program will use memory[17] to hold data that could not fit into registers[18].

When the data stored in memory are inherently temporary in nature, it is useful to have a mechanism to easily decide where in memory to store this data. One example of inherently temporary data are local variables in a function: Once the function returns, they are no longer valid. The standard mechanism for handling such temporary memory is for the processor to treat part of memory as a *stack*. However, unlike a stack in the colloquial sense, a program's stack grows *downwards*, from higher memory addresses to lower memory addresses. The convention is to use the highest address in the region of memory a program can access[19] as the starting point of the stack and to grow the stack downwards from there.

For data that is long-living and which is not limited in scope to a specific function, it is useful to have another region of memory for storing data. The standard mechanism is to reserve a region of memory for storing long-lived data. This region of memory is called the *heap*. By convention, the heap starts in memory right after the end of the **`.text`**, **`.data`**, and **`.bss`** sections and grows upwards towards the stack. Unlike the stack, the heap is not necessarily organized temporally and it might have holes.

Figure 4.3 shows an example address space, with the stack growing downwards from the address **`0x80FFFFF`** and the heap growing upwards from the address just above the end of the region of executable code in memory.

On each C function call, the code generated by most compilers will allocate a new region of the stack for local variables, growing the stack further downwards. Programs that have a very long chain of function calls (e.g., as a result of recursion) can therefore in principle have a stack that overflows into the heap or into the application code region in Figure 4.3.

[17] In the case of the KL03 and KL05, SRAM.

[18] Every processor has a limited number of registers, typically between 16 and 32.

[19] That is, the program's *address space*.

4.13.2 Dynamic memory allocation with **`malloc`**, **`calloc`**, *and* **`alloca`**

You can statically-allocate long-lived data structures by defining them as global variables. To dynamically allocate non-temporary data structures, you use the **`malloc`** and **`calloc`** C library calls. You can find the details of the syntax of **`malloc`** and **`calloc`** in Section 16.1 of the fifth edition of Harbison

and Steele.[20]

In working with the KL03 and KL05, we will almost never dynamically allocate memory. In many deeply-embedded systems, you will have so little memory to work with that you will rarely (if ever) resort to using **`malloc`** and **`calloc`**.

Finally, you can dynamically allocate memory on the stack, in the same way **`malloc`** and **`calloc`** allow you to allocate memory on the heap. To do this, you use the C library function **`alloca`**.

The program's dynamically allocated memory will typically grow upwards from the top of the **`.bss`** section, and the program's stack will typically grow downwards from an address setup either by `crt0.o`[21], or from an address explicitly set by the executable (or by the operating system) at the point at which the program begins executing.

[20] Steele and Harbison 2007.

[21] The *C RunTime 0* which is low-level initialization code that sets up the stack and calls **`main`**.

4.14 *Debugging Information*

In addition to executable code, an object file or an executable file may contain additional information to assist debuggers. There are several standard formats for such debugging information, including the DWARF debugging format[22] and the STABS debugging format.[23]

[22] Eager et al. 2007.

[23] Menapace, Kingdon, and MacKenzie 1992.

4.15 *Anatomy of Four C Programs*

To see what goes into running a C program on an embedded system, we will use the Sunflower embedded system emulator[24,25] and its associated tools. Sunflower is a program, analogous to VirtualBox or VMWare, which models an entire embedded system platform (see Question 50, Question 51, and Question 52 in the quiz of Section 4.1.5 above). Sunflower implements, in software, a complete processor, memory, peripherals such as communication interfaces and sensors, the (analog) physical signals to be sensed by sensors, the power dissipation of computation and its dependence on supply voltage, the interdependence between maximum clock frequency and supply voltage, the power dissipation of communication interfaces, the electrochemical properties of batteries, and more. You can compile any C/C++ program and load the binary into Sunflower for execution. Sunflower implements processors for multiple processor architectures. In the examples below, we will use its Hitachi SH processor support.

[24] Stanley-Marbell and Hsiao 2001.

[25] Stanley-Marbell and Marculescu 2007.

4.15.1 *Program 1: A simple C program with a function named* **`main`**

The directory `benchmarks/source/SimpleMainExample/` in the Sunflower GitHub repository contains the source code and additional files for a simple example. The file `simple.c` contains a simple C program:

```
int
main(void)
{
    return 0;
}
```

The function `main` in C is special. It is the default entry point at which code execution begins when a program is loaded by the shell or by an **`exec()`** system call. However, when your program runs directly over the bare hardware, there is no shell or operating system. The *symbol*[26] **`main`** is also special because programs containing a function called **`main`** are treated differently by the C compiler and linker.

When we run `make`[27] in the directory `benchmarks/source/SimpleMainExample/`, the following is executed:

```
sh-elf-gcc -E init.S > init.i; sh-elf-as init.i -o init.o
sh-elf-gcc -gstabs3 -O0  -c simple.c
sh-elf-ld -Ttext 0x08004000 -TsuperH.ld -Map simple.map init.o simple.o -o simple
sh-elf-objcopy -O srec simple simple.sr
```

The file `benchmarks/source/SimpleMainExample/init.S` contains assembly language code which initializes the processor's status register and sets up the processor's stack pointer register. When C programs are compiled to machine code, those variables are either assigned to machine registers, assigned to offsets from the top of the stack, or assigned to other locations in the program's memory map as introduced in Section 4.13.1 and detailed further in Section 4.16 below. The low-level initialization code in `init.S` therefore needs to setup the processor's stack pointer to point to a valid region of memory. The command `sh-elf-gcc -E init.S > init.i` invokes the C preprocessor to process the **`#include`** preprocessor directive in the file `init.S`, and places the preprocessed output in the file `init.i`. The file `init.i` is then assembled (converted into an object file) by the assembler (`sh-elf-as`). Similarly, the command `sh-elf-gcc` compiles `simple.c` into an object file. In the penultimate step, the linker, `sh-elf-ld`, links the object files `init.o` and `simple.o`. The directive `-Ttext 0x08004000` instructs the linker to setup the binary to be loaded at address **`0x0800 4000`** and the directive `-TsuperH.ld` instructs the linker that the objects in the binary should be laid out in memory according to the rules defined in the linker script file `superH.ld`. The directive `-Map simple.map` tells the linker to output a "map" of the linked binary's address space into the file `simple.map`. In the final step, the program `sh-elf-objcopy` converts the executable (linked) binary in `simple`, which happens to be in the ELF file format, into the Motorola S-record format, placing the result in the file `simple.sr`.

We can *disassemble* the binary in the file `simple` to see what instructions and data it contains and at what memory addresses those instructions and data will be loaded when the binary is loaded into a processor's memory:

[26] The term *symbol* is typically used to refer to any identifier or label in a program.

[27] If running this in your clone of your fork of the Sunflower repository on the coursework server, and you have not gone through the process of configuring and compiling your complete Sunflower repository, you can to use the command `make TREEROOT=/usr/local/src/git/sunflower` to use the pre-installed *Sunflower elf-format compiler tools* on the coursework server.

```
% sh-elf-objdump -d simple

simple:     file format coff-sh


Disassembly of section .text:

08004000 <_text>:
 8004000:   c9 00           and #0,r0
 8004002:   40 0e           ldc r0,sr
 8004004:   df 02           mov.l   8004010 <stack_addr>,r15    ! c001000
 8004006:   d0 03           mov.l   8004014 <start_addr>,r0 ! 800401c <_main>
 8004008:   40 0b           jsr @r0
 800400a:   00 09           nop
 800400c:   e4 01           mov #1,r4
 800400e:   c3 22           trapa   #34

08004010 <stack_addr>:
 8004010:   0c 00           .word 0x0c00
 8004012:   10 00           mov.l   r0,@(0,r0)

08004014 <start_addr>:
 8004014:   08 00           .word 0x0800
 8004016:   40 1c           shad    r1,r0

08004018 <___errno>:
 8004018:   00 00           .word 0x0000
 800401a:   00 00           .word 0x0000

0800401c <_main>:
 800401c:   2f e6           mov.l   r14,@-r15
 800401e:   6e f3           mov r15,r14
 8004020:   e1 00           mov #0,r1
 8004022:   60 13           mov r1,r0
 8004024:   6f e3           mov r14,r15
 8004026:   6e f6           mov.l   @r15+,r14
 8004028:   00 0b           rts
 800402a:   00 09           nop

0800402c <_etext>:
 800402c:   00 00           .word 0x0000
 800402e:   00 00           .word 0x0000
```

The output of `sh-elf-objdump` is a total of 41 lines. We see from the disassembled binary that the first two instructions are an **AND** and an **LDC**, and that these will be loaded into memory at address **0x08004000** as we explicitly requested in the linker command line arguments. These instructions are the first two instructions of the assembly language file `init.S`.

There is no requirement in the C language for a program to have a function called **main**. It is however necessary for an operating system (or, by extension, a command shell) to know where (e.g., at what address or offset) in a binary to start executing and the signature [28] of the function in an executable it will invoke. Therefore, by convention, an executable to be launched by the operating system has a function called **main** whose signature is typically either

[28] Recall from Section 4.2.1 that the signature of a function is the types and number of its arguments and return value.

int main(void); or **int main(int argc, char * argv[]);**.

Including a function called **main** in a C program also has other side effects which are important when compiling programs to run directly over a processor without an operating system. At the linking stage, if the compiler's linker detects a function called **main** in an object file, it links in a special object file (often named crt0.o[29]). The object file crt0.o (or its equivalent) is generated from a special startup code sequence, usually in an assembly language file (e.g., crt0.S). The assembly language instructions in a given system's crt0.S perform important tasks such as setting up the processor's stack and frame pointer registers before jumping to the first instruction in **main**.

For the GCC compiler, you can control whether or not crt0.o and other functionality will be linked in, by either building the compiler with an appropriate *"specs"* configuration file, or by controlling the compiler's behavior by specifying a specs file using the flag **-specs=***specs-file-name*. In the example above, the compiler, sh-elf-gcc, is preconfigured to use a configuration which does not link in crt0.o. In practice, you are likely to run into C compilers that are not configured this way and which treat **main** differently. On the coursework server, we have also installed a version of GCC[30] that for compiling programs for Sunflower that treats the presence of **main()** the same way a C compiler on your Desktop does by linking in **crt0.o**.

We can repeat the example above using the C compiler that behaves more closely to C compilers on your desktop by running make[31] in the directory benchmarks/source/SimpleMainExample/:

```
    sh-coff-gcc -E init.S > init.i; sh-coff-as init.i -o init.o
    sh-coff-gcc -DM32 -Wall  -gstabs3 -O0  -c simple.c
    sh-coff-ld -Ttext 0x08004000 -TsuperH.ld -L/usr/local/src/git/sunflower-
     simulator-coff-build-github-clone-a1798/tools/tools-lib/superH -L/usr/local/
     src/git/sunflower-simulator-coff-build-github-clone-a1798/sys/libOS/mOS -Map
     simple.map init.o simple.o  -o simple -lc -lgcc -lm -lmOS-superH
    /usr/local/src/git/sunflower-simulator-coff-build-github-clone-a1798/tools/bin/
     sh-coff-objcopy -O srec simple simple.sr
```

We can again *disassemble* the binary in the file simple[32] to see what instructions and data it contains and at what memory addresses those instructions and data will be loaded when the binary is loaded into a processor's memory:

```
% sh-coff-objdump -d simple

simple:     file format coff-sh

Disassembly of section .text:

0000000008004000 <_text>:
 8004000:   c9 00           and #0,r0
 8004002:   40 0e           ldc r0,sr

    ... 152 lines deleted ...

```

[29] **crt0** stands for *C Run-Time o* and is low-level initialization code that sets up the stack and calls **main**.

[30] It is installed as sh-coff-gcc and its associated assembler, linker, and object dump tools are sh-coff-as, sh-coff-ld, and sh-coff-objdump, respectively.

[31] If running this in your clone of your fork of the Sunflower repository on the coursework server, and you have not gone through the process of configuring and compiling your complete Sunflower repository, you can to use the command make TREEROOT=usr/local/src/git/sunflower- to use the pre-installed *alternative Sunflower coff-format compiler tools* on the coursework server.

[32] Remember, this is compiled from exactly the same source files as the example above.

```
 80040ee:   00 09          nop
```

The output of `sh-coff-objdump` this time is a total of 160 lines. Here, GCC detects a function named main. Because it has not been told to behave differently by its default *specs*, it links in a function named **`__main`** from object file `__main.o` in the library `libgcc.a`. This in turn expects to be linked against the symbol **`atexit`** which the linker finds in the object file `atexit.o` from the library `libmOS-superH.a`.

As described above, he objective of GCC linking in these additional pieces of code is GCC's expectation that, given the special nature of the function **`main`** (the main entry point of a C program), it needs to appropriately setup the processor's stack pointer before execution and to appropriately cleanup after the function **`main`** exits.

4.15.2 Program 2: A simple C program with no `main`

To see the difference the name of the function in the body of the C program has, for our example on the Hitachi SH architecture, we compile a program that *does not* contain the function **`main`** to an object file. We will then disassemble that object file and the assembly code we get is very different from the assembly code we get if the program contains a function called **`main`** for the second version of the compiler (`sh-coff-gcc`). The directory `benchmarks/source/SimpleNoMainExample/` in the Sunflower GitHub repository contains the source code and additional files for a simple example *without* a function named **`main`**.

```
int
noMain(void)
{
    return 0;
}
```

Compiling this just as we did for the previous example and disassembling this binary, the output of `sh-elf-objdump` is now just of 38 lines, both for `sh-coff-gcc` as well as for `sh-elf-gcc`:

```
% sh-elf-objdump -d simple

simple:     file format coff-sh

Disassembly of section .text:

0000000008004000 <_text>:
 8004000:   c9 00          and #0,r0
 8004002:   40 0e          ldc r0,sr
 8004004:   df 02          mov.l   8004010 <stack_addr>,r15    ! 0xc001000
 8004006:   d0 03          mov.l   8004014 <start_addr>,r0 ! 0x8004020 <_noMain>
 8004008:   40 0b          jsr @r0
 800400a:   00 09          nop
 800400c:   e4 01          mov #1,r4
```

```
 800400e:   c3 22           trapa   #34

0000000008004010 <stack_addr>:
 8004010:   0c 00           .word 0x0c00
 8004012:   10 00           mov.l   r0,@(0,r0)

0000000008004014 <start_addr>:
 8004014:   08 00           .word 0x0800
 8004016:   40 20           shal    r0

0000000008004018 <___errno>:
 8004018:   00 00           .word 0x0000
 800401a:   00 00           .word 0x0000
 800401c:   00 09           nop
 800401e:   00 09           nop

0000000008004020 <_noMain>:
 8004020:   2f e6           mov.l   r14,@-r15
 8004022:   6e f3           mov r15,r14
 8004024:   e1 00           mov #0,r1
 8004026:   60 13           mov r1,r0
 8004028:   6f e3           mov r14,r15
 800402a:   6e f6           mov.l   @r15+,r14
 800402c:   00 0b           rts
 800402e:   00 09           nop
```

If you try compiling this same program, this time not using the cross-compiler, but instead using the host compiler, the compilation will fail:

```
SimpleNoMainExample$ gcc simple.c
/usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crt1.o: In function `
    _start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

This is because the version of GCC on the host machine will by default always link in the C runtime initialization library[33]. As a result, there will be a reference to the symbol **main** which will be missing when the linker (ld) attempts to link together the object files to generate the final binary.

[33] Unless overridden with a *specs* file.

4.15.3 *Program 3: Program with only* **main** *and* **#include <stdio.h>**

The directory benchmarks/source/SimpleMainStdioExample/ contains the source, makefile, and other associated files for the program below:

```
#include <stdio.h>

int
main(void)
{
    return 0;
}
```

Compiling and disassembling the program above, we obtain the same output as for the example in Section 4.15.1. The preprocessor directive **#include**

<stdio.h> only causes the definition of function prototypes and constants. It does not cause the inclusion of any source code.

4.15.4 *Program 4: A "Hello, World" program with a function named* **main**

The directory benchmarks/source/HelloWorld/ contains the source code and associated files for the example below:

```
#include <stdio.h>

int
main(void)
{
    printf("Hello, World\n");

    return 0;
}
```

In the example above, we have added a call to the standard C library function **printf** to the example of Section 4.15.3. If you compile, link, and then disassemble the resulting binary, you will observe that there are now 5039 lines in the output (4124 lines when using the COFF version of the tools). To determine exactly what object files are linked into a binary, Section 4.16 describes how we can use *map files*. Map files are *output* by the linker when it is invoked with the flag -Map followed by a file into which to place the generated map. Unlike linker script files which are *inputs* to the compiler and linker and give the linker additional instructions for what to do during linking, map files are outputs by the linker and describe what the linker did during linking.

4.16 *Program Variables, Object Addresses, Memory Locations, Machine Registers, and Debugging Information*

Variables defined global to the scope of a C program are either stored in the program's **.data** section if they are initialized with values in the source program, or in the **.bss** section of the binary if they are global uninitialized variables. We can determine the memory locations of all objects in a compiled program by looking at its map file. For example, for the C program in the Sunflower Github repository in the directory benchmarks/source/SimpleNoMainMemoryObjectsExample/,

```
int gInitializedInt = 42;
int gUninitializedInt;

int
noMain(void)
{
    int i, localInt;

    for (i = 0; i < 10; i++) {
        localInt += gInitializedInt + gUninitializedInt;
    }

```

```
    return 0;
}
```

We can compile, link, and instruct the linker to generate a map file as we did in the examples above. The resulting map file output by the linker for this example is:

```
Allocating common symbols
Common symbol       size              file

gUninitializedInt   0x4               simple.o

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

Address of section .text set to 0x8004000

.text           0x0000000008004000       0x80
                0x0000000008004000                _text = .
 *(.text)
 .text          0x0000000008004000       0x1c init.o
                0x0000000008004018                __errno
 .text          0x000000000800401c       0x5c simple.o
                0x000000000800401c                noMain
                0x0000000008004078                _etext = .

.data           0x0000000008004078       0x10
                0x0000000008004078                _data = .
 *(.data)
 .data          0x0000000008004078        0x0 init.o
 .data          0x0000000008004078        0x4 simple.o
                0x0000000008004078                gInitializedInt
                0x000000000800407c                _edata = .

.bss            0x000000000800407c        0x4
                0x000000000800407c                _bss = .
 *(.bss)
 .bss           0x000000000800407c        0x0 init.o
 .bss           0x000000000800407c        0x0 simple.o
                0x000000000800407c                _ebss = .
                0x000000000800407c                _end = .
 COMMON         0x000000000800407c        0x4 simple.o
                0x000000000800407c                gUninitializedInt
LOAD init.o
LOAD simple.o
OUTPUT(simple coff-sh)

.stab           0x0000000000000000      0x1e0
 .stab          0x0000000000000000      0x1e0 simple.o

.stabstr        0x0000000000000000      0x360
 .stabstr       0x0000000000000000      0x355 simple.o

.comment        0x0000000000000000       0x20
```

```
.comment       0x0000000000000000       0x12 simple.o
```

For each of the sections defined in the linker script file used to create a given object file or final linked binary (left-most column in the example above), the map file output by the linker describes the object files, the global symbols within those object files, and the memory addresses to which they are mapped. For example, we see from the example above that the **`.text`**, **`.data`**, and **`.bss`** sections of the binary for our example begin at memory addresses **`0x0800 4000`**, **`0x0800 4078`**, and **`0x0800 407c`** respectively. The global initialized variable **`gInitializedInt`** is in the **`.data`** segment like it should be (and that it is at address **`0x0800 4078`**)[34] and the uninitialized global variable **`gUninitializedInt`** is in the **`.bss`** segment like it should be (and that it is at address **`0x0800 407c`**)[35].

[34] Recall that the **`.data`** segment is also known as the *initialized data segment*.

[35] Recall that the **`.bss`** segment is also known as the *uninitialized data segment*.

The map file provides information about all the symbols in a binary and is useful to determine what objects the linker linked into a given executable as well as to determine what symbols are visible outside of the scope of an object file during the linking process.

The map file however does not provide us any information about *automatic variables*: Variables allocated by the compiler to registers or allocated space on the stack. To figure out whether variables in a given program are allocated to registers, allocated on the stack, and to what registers or at what offset on the stack they are allocated, we can use information generated by the compiler to aid debugging.

There are several debugging formats in use today. These include the DWARF format,[36] the STABS[37] format, and several others. Most programmers never manually extract the debugging information from their binaries but instead indirectly benefit from the debugging information generated by compilers, by using a debugger such as `gdb` or `lldb`.

[36] Eager et al. 2007.

[37] Menapace, Kingdon, and MacKenzie 1992.

The format in which debugging information is stored in binaries is not generally well documented. One debugging format which is reasonably well documented is the STABS format. The example in `benchmarks/source/SimpleNoMainMemoryObjectsExample/` of the Sunflower GitHub repository is setup to dump debugging information from the final linked binary into a human-readable text file. It does so using the `sh-elf-objdump` command which we used previously to convert the linked binary into an S-record file. To extract the STABS debugging information from the binary for our example, we invoke `sh-elf-objdump` with the flag **`-G`**:

```
% sh-elf-objdump -G simple

simple:     file format coff-sh

Contents of .stab section:

Symnum n_type n_othr n_desc n_value  n_strx String
```

```
8
9  -1   HdrSym 0    39   00000355 1
10 0    SO     0    2    0800401c 10   simple.c
11 1    OPT    0    0    00000000 19   gcc2_compiled.
12 2    LSYM   0    0    00000000 34   int:t(0,1)=r(0,1);-2147483648;2147483647;
13 3    LSYM   0    0    00000000 76   char:t(0,2)=r(0,2);0;127;
14 4    LSYM   0    0    00000000 102  long int:t(0,3)=r(0,3);-2147483648;2147483647;
15 5    LSYM   0    0    00000000 149  unsigned int:t(0,4)=r(0,4);0;4294967295;
16 6    LSYM   0    0    00000000 190  long unsigned int:t(0,5)=r(0,5);0;4294967295;
17 7    LSYM   0    0    00000000 236  __int128:t(0,6)=r(0,6);0;-1;
18 8    LSYM   0    0    00000000 265  __int128 unsigned:t(0,7)=r(0,7);0;-1;
19 9    LSYM   0    0    00000000 303  long long int:t(0,8)=r(0,8);-9223372036854775808;9223372036854775807;
20 10   LSYM   0    0    00000000 373  long long unsigned int:t(0,9)=r(0,9);0;-1;
21 11   LSYM   0    0    00000000 416  short int:t(0,10)=r(0,10);-32768;32767;
22 12   LSYM   0    0    00000000 456  short unsigned int:t(0,11)=r(0,11);0;65535;
23 13   LSYM   0    0    00000000 500  signed char:t(0,12)=r(0,12);-128;127;
24 14   LSYM   0    0    00000000 538  unsigned char:t(0,13)=r(0,13);0;255;
25 15   LSYM   0    0    00000000 575  float:t(0,14)=r(0,1);4;0;
26 16   LSYM   0    0    00000000 601  double:t(0,15)=r(0,1);8;0;
27 17   LSYM   0    0    00000000 628  long double:t(0,16)=r(0,1);8;0;
28 18   LSYM   0    0    00000000 660  _Float32:t(0,17)=r(0,1);4;0;
29 19   LSYM   0    0    00000000 689  _Float64:t(0,18)=r(0,1);8;0;
30 20   LSYM   0    0    00000000 718  _Float32x:t(0,19)=r(0,1);8;0;
31 21   LSYM   0    0    00000000 748  void:t(0,20)=(0,20)
32 22   GSYM   0    0    00000000 768  gUninitializedInt:G(0,1)
33 23   GSYM   0    0    00000000 793  gInitializedInt:G(0,1)
34 24   FUN    0    0    0800401c 816  noMain:F(0,1)
35 25   SLINE  0    6    00000000 0
36 26   SLINE  0    9    00000006 0
37 27   SLINE  0    9    0000000e 0
38 28   SLINE  0    11   00000012 0
39 29   SLINE  0    11   0000001c 0
40 30   SLINE  0    9    0000002a 0
41 31   SLINE  0    9    00000038 0
42 32   SLINE  0    14   00000044 0
43 33   SLINE  0    15   00000046 0
44 34   LSYM   0    0    00000004 830  i:(0,1)
45 35   LSYM   0    0    00000000 838  localInt:(0,1)
46 36   LBRAC  0    0    00000000 0
47 37   RBRAC  0    0    0000005c 0
48 38   SO     0    0    08004078 0
```

The second column of the output indicates the type of symbol described by a given row (line) in the output. For example, **`FUN`** indicates a function definition, in this case, indicating the function **`noMain`**. The fifth column contains information specific to the symbol type (in the case of **`noMain`**, its location in the address space of the program when the program is running); for the local variable **`localInt`**, this column indicates the offset from the frame pointer on the stack, at which storage for the variable is allocated.

The Sunflower emulator can decode the output of `sh-elf-objdump -G`. By supplying Sunflower with both a binary in S-record format as well as the STABS debugging output report, Sunflower can generate a report of the history of all accesses to any program variable, whether that variable is on the

stack, in a register, or in the **.data** or **.bss** segments:

```
% sf
Initialized random number generator with seed -548296926...

Sunflower 1.1 (build 10-12-2018-12:38:22-pip@physcomplab-cpu0-Linux)
Authored, 1999-2018, by Phillip Stanley-Marbell <phillip.stanleymarbell@gmail.com>
Public key fingerprint 62A1 E95D 304D 9876 D5B1  1FB2 BF7E B65F BD89 20AB
This software is provided with ABSOLUTELY NO WARRANTY. Read LICENSE.txt

New node created with node ID 0

[ID=0 of 1][PC=0x8000000][3.3E+00V, 6.0E+01MHz] sizemem 96000000
Set memory size to 93750 Kilobytes
[ID=0 of 1][PC=0x8000000][3.3E+00V, 6.0E+01MHz] srecl simple.sr
Loading S-RECORD to memory at address 0x8004000
[M] Done.
[ID=0 of 1][PC=0x8004000][3.3E+00V, 6.0E+01MHz] registerstabs simple.stabs
Global variable "gUninitializedInt" in global scope, typename "int",    ptr=0
Global variable "gInitializedInt" in global scope, typename "int",  ptr=0
Stack var "i" in fn scope, typename "int", fn start=0x800401c, frame offset=4,  ptr=0
Stack var "localInt" in fn scope, typename "int", fn start=0x800401c, frame offset=0,   ptr=0
[ID=0 of 1][PC=0x8004000][3.3E+00V, 6.0E+01MHz] run
args = [], argc = 0
R4 = [0x00000000], R5 = [0x0db7d700]
Running...

[ID=0 of 1][PC=0x8004000][3.3E+00V, 6.0E+01MHz] on
[ID=0 of 1][PC=0x8004000][3.3E+00V, 6.0E+01MHz]

NODE 0 exiting...
User Time elapsed = 0.000000 seconds.
Simulated CPU Time elapsed = 5.466667E-06 seconds.
Simulated Clock Cycles = 328
Estimated CPU-only Energy = 4.423023E-06



[ID=0 of 1][PC=0x8004016][3.3E+00V, 6.0E+01MHz] valuestats

Name:               int  localInt
PCstart:            0x800401c
Frame offset:       0x0
Size:               0x4
Read accesses:      11
Write accesses:     10

Value History: 0 42 42 84 84 126 126 168 168 210 210 252 252 294 294 336 336 378 378 420 0

Name:               int  i
PCstart:            0x800401c
Frame offset:       0x4
Size:               0x4
Read accesses:      21
Write accesses:     11

Value History: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9 10 10
[ID=0 of 1][PC=0x8004016][3.3E+00V, 6.0E+01MHz]
```

4.17 Memory-Mapped I/O

In most architectures, several of the processor status facilities are implemented as *memory-mapped registers*. These are essentially addresses in the address space which, when read, yield the value of a hardware system register. For example, for the Hitachi SH architecture modeled in Sunflower, the **`EXCP_INTEVT`** memory-mapped register is a hardware register which is accessed by reading from memory address **`0xFFFFFFD4`**. Typically, in most architectures, some memory-mapped registers are byte addressed, others are word (16-bit) addressed, and yet others are long-word (32-bit) addressed.

4.18 Compiling C programs: A recap using the ARM architecture

Many people who have written C programs and even some of those who write and modify small or large programs as part of their daily activities do not really know what goes on behind the scenes when those programs are compiled and run. What does **`#include <stdio.h>`** in a C program do? Is **`#include`** part of the C language syntax? What about **`main`**? Is **`main`** a reserved keyword in C? Can you create an executable without a function called **`main`**? What is the difference between an executable and an object file?

Even if you don't write programs yourself (maybe you consider yourself a *hardware person*; see the quote by Alan Kay at the start of this chapter), you might occasionally have to compile a program and run it. What does a compiler do to convert a C program into an executable? What is in an executable? What are the parts of an executable? Does everything in an executable get loaded into the processor to run? What are some of the different executable and object file formats? What tools can you use to convert between different object file formats? What is the difference between an executable file and an object file? What is a `Makefile`? What is the difference between `make`, `gmake`, and `cmake`?[38] What is a `configure` script?

When writing software that runs directly on a processor without the intermediation of an operating system (often referred to as "running over bare metal"), you invariably need to know the answers to all the above questions. In addition, you will need to be intimately familiar with the answers to many more related questions, such as: What is the memory map of a binary? How can you control the memory map of a binary? What command line tools can you use to inspect the memory map of a binary?

We can illustrate the answers to the above questions with an example. For the remainder of the chapter, we will switch from using the Hitachi SH in our examples, to using the ARM Cortex-Mo. As you will see, all of the ideas presented so far using the Hitachi SH carry over directly to the ARM with

[38] And there's also `mk`

only a change of the name prefix of the tools we invoke from `sh-elf-` to `arm-none-eabi-`. Consider the following C program in the file `simple.c`:

```
int
main(void)
{
    return 0;
}
```

We compile it from the command line using `gcc` for the ARM architecture (i.e., `arm-none-eabi-gcc`):

```
% arm-none-eabi-gcc -c simple.c
```

The result is the file `simple.o`. The file `simple.o` is an object file. As we saw previously in Section 4.2, an object file contains the compiled code (machine instructions) for a given source file, but does not contain any code for libraries that might be referenced from the source file. We can use the utility `arm-none-eabi-objdump` to view the contents of the object file `simple.o`:

```
% arm-none-eabi-objdump -d simple.o

main.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
   4:   e28db000    add fp, sp, #0
   8:   e3a03000    mov r3, #0
   c:   e1a00003    mov r0, r3
  10:   e24bd000    sub sp, fp, #0
  14:   e49db004    pop {fp}        ; (ldr fp, [sp], #4)
  18:   e12fff1e    bx  lr
```

The values on the left hand side of the output (before each colon) are addresses, and the rest of each line is a disassembly of the binary machine code contained in the object file at the given address. Thus, for example, at address **`0x18`** (the last address) in the object file, there is a **`bx lr`** (branch exchange) machine instruction.

What would the object file look like if we included one of the typical standard header files such as **`stdio.h`**? For the following modified `simple.c`

```
#include <stdio.h>

int
main(void)
{
    return 0;
}
```

if we again compile using `arm-none-eabi-gcc`, and dump the object file contents using `arm-none-eabi-objdump`,

```
% arm-none-eabi-gcc -c simple.c
% arm-none-eabi-objdump -d simple.o

main.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
   4:   e28db000    add fp, sp, #0
   8:   e3a03000    mov r3, #0
   c:   e1a00003    mov r0, r3
  10:   e24bd000    sub sp, fp, #0
  14:   e49db004    pop {fp}        ; (ldr fp, [sp], #4)
  18:   e12fff1e    bx  lr
```

the contents of the object file `simple.o` are unchanged.

Header files (such as **`stdio.h`**) are intended to be used to include data structure definitions, function prototypes, and the like. By convention, they should never contain executable code. Standard system header files such as **`stdio.h`** do not contain any executable program statements. So, the statement **`#include <stdio.h>`** does not add any code to `simple.c`. It simply adds new function prototypes, constant definitions, and the like which might be needed if we were to have program statements that need those definitions, such as a call to the function **`printf()`**.

The object file `simple.o` is not an executable, i.e., we cannot run it. To obtain an executable, we would have had to either compile the original `simple.c` file without the `-c` flag to `arm-none-eabi-gcc`, or we would have to *link* the object file `simple.o` to obtain an executable. In the following example, we also explicitly specify a linker script file[39], `arm.ld`, via the `-T` flag, to instruct the linker how to lay out the symbols and sections in memory:

[39] More on linker script files in Section 4.11 below

```
% arm-none-eabi-ld -Tarm.ld -o simple simple.o
% arm-none-eabi-objdump  -d simple

examples/simple:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
   4:   e28db000    add fp, sp, #0
   8:   e3a03000    mov r3, #0
   c:   e1a00003    mov r0, r3
  10:   e24bd000    sub sp, fp, #0
  14:   e49db004    pop {fp}        ; (ldr fp, [sp], #4)
  18:   e12fff1e    bx  lr
```

What happens if we now include a call to a function in the standard I/O library:

```c
#include <stdio.h>

int
main(void)
{
    printf("Hello\n");

    return 0;
}
```

Compiling this source file to obtain a new object file:

```
% arm-none-eabi-gcc -c hello.c -o hello.o
% arm-none-eabi-objdump -d hello.o

examples/hello.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e92d4800    push    {fp, lr}
   4:   e28db004    add fp, sp, #4
   8:   e59f0014    ldr r0, [pc, #20]   ; 24 <main+0x24>
   c:   ebfffffe    bl  0 <puts>
  10:   e3a03000    mov r3, #0
  14:   e1a00003    mov r0, r3
  18:   e24bd004    sub sp, fp, #4
  1c:   e8bd4800    pop {fp, lr}
  20:   e12fff1e    bx  lr
  24:   00000000    .word   0x00000000
```

We can ask the compiler to link the object file to generate an executable. In the following example, because the library which we need is the standard C library, we do not need to explicitly specify the path to the library (for which we would have needed to specify the linker flag `-L`) or the name of the library to be found at that path (for which we would have needed to specify the linker flag `-l`).

```
% arm-none-eabi-gcc -Tmkl25z4.ld -o hello hello.o
% arm-none-eabi-objdump -d hello

examples/hello:     file format elf32-littlearm


Disassembly of section .text:

00000410 <__do_global_dtors_aux>:
     410:   e92d4010    push    {r4, lr}
     414:   e59f4028    ldr r4, [pc, #40]   ; 444 <__do_global_dtors_aux+0x34>

     /*     36 lines omitted    */

000004a4 <_mainCRTStartup>:
     4a4:   e59f30f0    ldr r3, [pc, #240]  ; 59c <_mainCRTStartup+0xf8>
     4a8:   e3530000    cmp r3, #0
     4ac:   059f30e4    ldreq   r3, [pc, #228]  ; 598 <_mainCRTStartup+0xf4>
```

```
19
20      /*     64 lines omitted    */
21
22  000005b8 <main>:
23      5b8:   e92d4800    push    {fp, lr}
24      5bc:   e28db004    add fp, sp, #4
25      5c0:   e59f0014    ldr r0, [pc, #20]   ; 5dc <main+0x24>
26      5c4:   eb000096    bl  824 <puts>
27      5c8:   e3a03000    mov r3, #0
28      5cc:   e1a00003    mov r0, r3
29      5d0:   e24bd004    sub sp, fp, #4
30      5d4:   e8bd4800    pop {fp, lr}
31      5d8:   e12fff1e    bx  lr
32      5dc:   000034fc    .word   0x000034fc
33
34  000005e0 <exit>:
35      5e0:   e92d4008    push    {r3, lr}
36
37      /*     3162 lines omitted      */
```

The output above is trimmed by 3262 lines.

4.19 Linking to Assembly Sources

You can use the **`-c`** flag to compile assembly language files into object files. You can then link those object files together with other object files (compiled from either C or assembler) to obtain your final binary.

The C compiler, by convention, prefixes the names of all functions in your C source file with an underscore (_) when generating the object file. You do not have functions *per se* in assembly, but you can name sections of assembly language code with *labels*. You can then expose these labels within the object file compiled from your assembly language source by using the assembly language directive **`.global`**. Unlike the C compiler, the assembler does not rename labels (or references to labels) to add an underscore. To reference a function in your C code from your assembly language code, you therefore need to prefix the C function's name with _ when referring to it from assembly. Similarly, if you prefix your labels with _ in your assembly language code (e.g., **`_sleep`**) and include the appropriate assembler directive to make the label visible in the object file, you will be able to call the corresponding block of assembler from your C program without the underscore (e.g., **`sleep`**).

4.20 Makefiles

The program `make` provides a way to automate sequences of commands at the command line, in particular when those sequences of commands have dependencies between them. This concept of automating the execution of commands that have dependencies between them is central to understanding what `make` is and how it works. The facility for automating sequences of de-

pendent programs means that `make` program is ideally suited for automating the process of compiling source files into object files using a compiler like `gcc`, and grouping those object files into archives or libraries using `ar`. The `make` utility allows you to manage dependencies such as ensuring that only those object files whose source files have been changed need to be rebuilt, and so on. Every makefile has two main kinds of entries or constituent lines: *definitions* and *rules*.

Definition statements introduce synonym variables for constants or expressions that are needed repeatedly inside the rest of the makefile. Typically, these definitions are all uppercase. So, for example, to define the set of warning flags that should be used in all C compiler invocations, a makefile might contain the following definition statement:

```
WFLAGS      = -Wall -Werror
```

Once a variable (e.g., **`WFLAGS`**) has been defined, it can be referred to using the variable name in parenthesis preceded by a **`$`** (e.g., **`$(WFLAGS)`**).[40]

[40] There are a number of different variations on how to refer to variables in makefiles depending on which version of the implementation of `make` you are using. The two common version of `make` are the BSD `make` and the GNU `make` (also called `gmake`). The program `cmake` on the other hand is a different thing altogether.

Rules in makefiles consist of three parts: (1) a *target* name, followed by a colon, followed by (2) a list of *dependent targets*. On the lines which follow, *which must each be offset with a tab (not with spaces)*, are commands to be executed to generate the target of the rule. Thus, for example, the following simple makefile will build the executable `simple` by first building the object files `simple.o` and then invoking the linker to link `simple.o` to yield the executable.

```
TARGET  = simple

OBJECTS = simple.o

all: $(OBJECTS)
        gcc $(OBJECTS) -o $(TARGET)

simple.o: simple.c
        gcc -c simple.c

clean:
        rm -f $(OBJECTS) $(TARGET)
```

The first target (the first string followed by a colon) is the default target that will be built by `make`. This target is conventionally named **`all`**. In the example above, the default target, **`all`**, depends on the variable **`$(OBJECTS)`**. In this example, **`$(OBJECTS)`** refers to a single object file, but it would typically be a list of object files. To build the default target, `make` will detect that it first needs to have a file `simple.o`. It uses the rule for `simple.o` (which in turn depends on there being a file `simple.c`), and builds the object file using the command `gcc -c simple.c`. With `simple.o` generated and the prerequisites for the rule **`all`** satisfied, `make` then builds the executable `simple` using the command `gcc $(OBJECTS) -o $(TARGET)` where **`$(OBJECTS)`** will be substi-

tuted with **simple.o** and **$(TARGET)** will be substituted with **simple**. Finally, the target **clean** provides a quick way to remove the intermediate files and outputs of the compilation process by typing `make clean` at the command prompt/shell in the directory containing the makefile.

The program `make` and its input makefiles support both a rich input language as well as a rich set of command line parameters and there are many ways to make the above makefile more succinct.

4.21 Summary

In embedded systems, there is often no operating system below your programs or *firmware* (software running directly over the processor). The programs you write typically have unfettered access to control the underlying hardware. Platforms such as Arduino shield you from having to deal with all the intricacies of hardware, but they also shield you from developing a complete understanding of what goes on under the hood.

This chapter introduced the inner workings of C programs for embedded systems compiled with the GCC compiler. We used the Sunflower embedded system emulator[41,42] to allow us to interactively explore the execution of programs. The concepts and examples introduced in this Chapter apply largely unchanged to other target processor architectures such as the ARM Cortex-Mo+ used in the coursework, and also apply largely unchanged to other embedded compiler toolchains.

[41] Stanley-Marbell and Hsiao 2001.
[42] Stanley-Marbell and Marculescu 2007.

4.22 Exercises

Go over the questions in Section 4.1.5 and make sure you can explain the answers to each of the questions to one of your peers. Many of the questions in that list were covered in this chapter, so you should be able to answer them once you have understood the material in this chapter. In addition, you will find the books listed in Section 4.23 and Section 4.24 useful.

4.23 Further Reading

The book *The C Programming Language (2nd Edition)* by the creators of the language is a concise introduction to most of what you'll ever need to know about C programming. If you don't already know how to program in C, it is a must-read. The book *C: A Reference Manual (5th Edition)*, is a reference book that you will want to own a copy of, if you do any C programming. A good introductory online course on C programming is available on EdX.

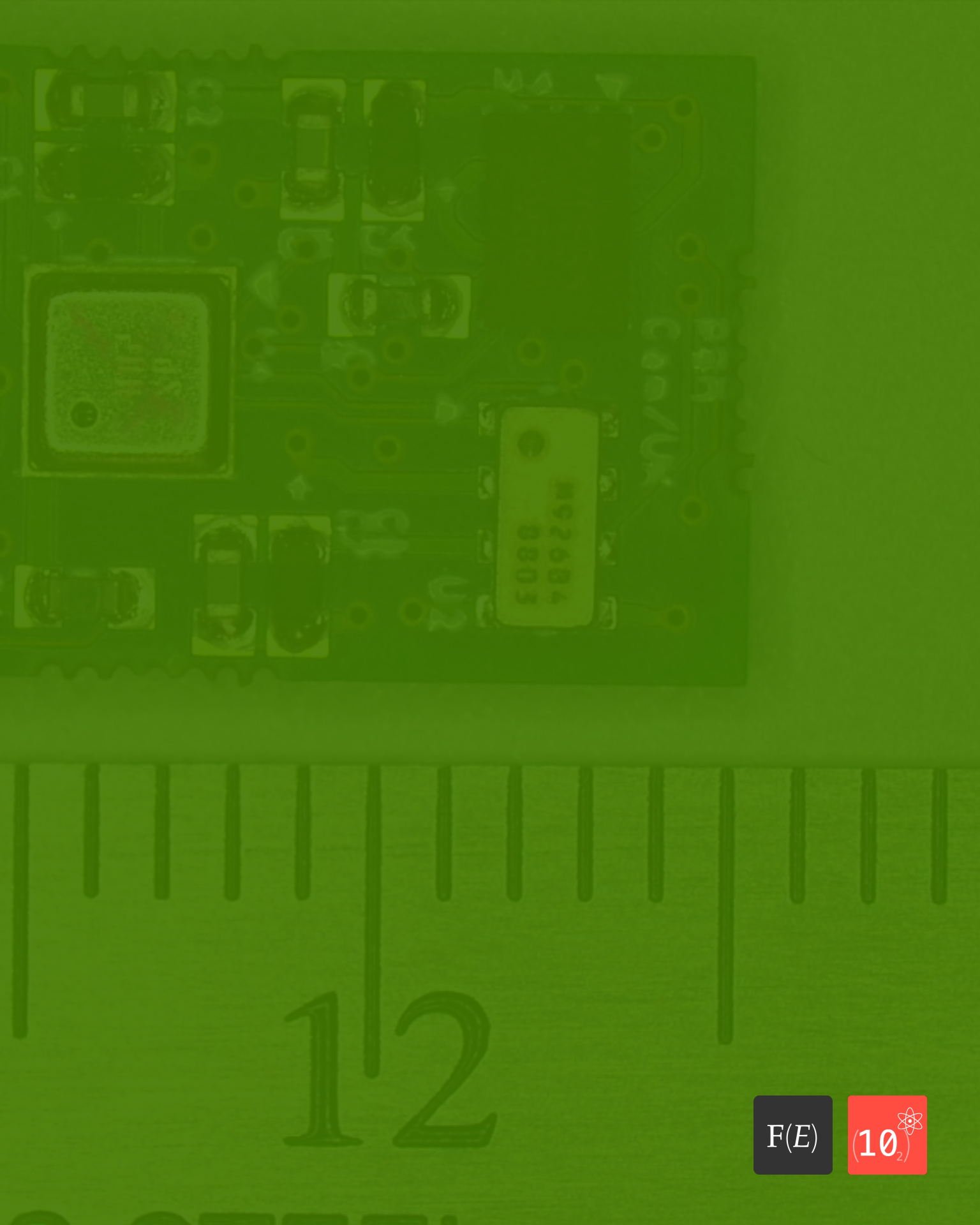
4.24 Relevant Books Available in the Engineering Library

1. *Linkers and Loaders*, ISBN: 978-1558604964.
2. *Programming Embedded Systems: With C and GNU Development Tools*, 2nd Edition, ISBN: 978-0596009830.
3. *C: A Reference Manual*, 5th Edition, ISBN: 978-0130895929.
4. *The Practice of Programming*, ISBN: 978-0201615869.
5. *Expert C Programming*, ISBN: 978-0131774292.

4.25 The Muddiest Point

Think about the following two questions and submit your responses through this link.

1. What was least clear to you in this chapter? (You can simply list the section numbers or write a few words.)
2. What was most clear to you in this chapter? (You can simply list the section numbers or write a few words.)


12