1

HELLO WORLD

In this chapter, you'll create and execute your first program, "Hello World." This is about the simplest program you can make and the first program in almost all C books. But you'll go beyond just creating it: you'll learn what's actually going on behind the scenes during its creation.

The tools you'll use are designed to make things quick and easy, which is good for regular programming but can be bad for embedded programming. The compiler, GCC, is actually a wrapper that runs a whole bunch of other tools. We'll look at what each tool does to get your program from code to execution. In the process, you'll discover that the GCC optimizer has a surprise for us. Although our program is very simple, the optimizer will decide to rewrite part of it to make it more efficient—*and it won't tell us about the rewrite!* In fact, we would never know about it if we didn't look under the hood to see what's going on. (I won't tell you what it will do to us; you'll have to read the rest of the chapter to find that out.)

Installing GCC

In order to run the program in this chapter, you'll need to download and install the GNU C compiler (GCC) on your system, along with related tools. The instructions for doing so vary based on your operating system.

On Windows, install Minimalist GNU for Windows (MinGW), which can be found at *http://www.mingw.org*. See *https://nostarch.com/bare-metal-c* for detailed instructions.

On macOS, the GCC compiler is part of the developer packages that can be accessed with the following command:

```
$ xcode-select --install
```

Select the Command Line Tools option for installation.

Linux installation instructions depend on which distribution you are using. For Debian systems such as Ubuntu and Linux Mint, use the following command:

```
$ sudo apt-get install build-essential
$ sudo apt-get install manpages-dev
```

For Red Hat–based systems (such as Fedora or CentOS), use the following command:

```
$ dnf groupinstall "Development Tools"
```

For any other Linux-based system, use the package manager that came with the system or search online to find the command needed for installation.

After installing the software, open a terminal window and issue the command **gcc**. If you get a "no input files" error, you've installed successfully.

```
$ gcc
gcc: fatal error: no input files
compilation terminated.
```

Downloading System Workbench for STM32

System Workbench for STM32 is an IDE we'll use to write C programs for our embedded device. We won't use it until Chapter 2, but the download will take some time, so I recommend you start it now. By the time you finish reading this chapter, the download should be complete.

Go to *http://openstm32.org/HomePage*, locate the link for System Workbench for STM32, and click it. Register (it's free), or log in if you have an account, and then follow the links to the installation instructions. Install the IDE from the installer and not from Eclipse. When the download starts, return here and continue reading.

Tools and installation procedures may change over time. If you encounter any issues, visit *https://nostarch.com/bare-metal-c* to check for updated instructions.

Our First Program

Our first program is called *hello.c*. Begin by creating a directory to hold this program and jump into it. Navigate to the root directory of your workspace, open a command line window, and enter these commands:

```
$ mkdir hello
$ cd hello
```

Using a text editor such as Notepad, Vim, or Gedit, create a file called *hello.c* and enter the following code:

```
#include <stdio.h>
int main()
{
    printf("Hello world!\n");
    return (0);
}
```

We'll walk through this program in detail in the following sections. First, though, we have to run it.

Compiling the Program

The file you just created is known as a *source file*, and it contains code in human-readable format. (Yes, really; this is supposed to be human readable.) It's the source of all the other files we are going to produce. The content of the file is called *source code*. The computer does not understand source code; it only understands *machine code*, a set of instructions in a numeric format. So, we need to transform our source code into machine code, a process called *compiling*.

To do this, we execute the following compiler command for macOS or Linux:

```
$ gcc -o hello hello.c
```

On Windows, we execute the following command:

```
$ gcc -o hello.exe hello.c
```

If you get no output, just a command prompt, the command was successful. Otherwise, you'll get error messages.

This command tells the program *GCC* to *compile* and *link* the program, putting the output in a file called *hello* on macOS and Linux or *hello.exe* for

Microsoft Windows. We can now run our program using the following command on macOS or Linux:

```
$ ./hello
Hello world!
```

On Windows, run the following:

```
$ hello
Hello world!
```

Making Mistakes

Let's introduce a mistake and see what happens. Change the second line so that it looks like this:

```
intxxx main()
```

Now let's try to compile the program.

```
$ gcc -o hello hello.c
hello.c:2:1: error: unknown type name 'intxxx'
 intxxx main()
 ^
```

The output tells us that there is a problem in line 2 of the program and that the error was discovered at character position 1. In this case, where the compiler was expecting a type, it got something different—namely, the garbage we deliberately put in. Fix the program by changing the line back.

Next let's take something out—specifically, the semicolon on the fourth line:

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

This gives us a different error message:

```
$ gcc -o hello hello.c
hello.c: In function 'main':
hello.c:5:5: error: expected ';' before 'return'
 return (0);
 ^
```

You'll notice that the compiler pointed to line 5 when issuing the error message. That's because although we made a mistake on line 4, the compiler didn't detect it until it looked at line 5.

Sometimes errors on a previous line will not be detected for one or more lines, so don't look just at the line specified by the error; look above as well.

Understanding the Program

Now let's go through our program line by line to see what it is doing. Take a look at the first line:

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

In order to build our program, we are using components that come with the compiler—namely, the standard input/output (I/O) package. The functions in this package are defined in the */usr/include/stdio.h* file. (Windows may use a slightly different directory.) Specifically, we use the standard I/O function `printf` later in the program.

Next, we define the starting point for our program:

```
int main()
```

The name `main` is special and indicates the main body of the program. All programs start at `main`. This is followed by a set of statements enclosed in curly brackets:

```
{
...
}
```

The curly brackets denote the body of `main`. In other words, they're used to group the statements that follow. We indent the statements inside the curly brackets by four spaces for readability, but you are free to use other indentation sizes. In fact, the C compiler doesn't care how much whitespace we use. We could have used no indentation at all, but no indentation makes the program impossible to read, so most C programmers indent their code.

Inside the curly brackets is our first executable statement:

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

This tells the program to use the standard I/O function `printf` to output a string to the standard output location (our terminal). The `\n` is a special character in this string. The backslash (\) is called the escape character. It tells C that the following character should be treated as code. In this case, the *n* tells C to output a "newline," which means the next character will be printed on a new line. Some of the more common escape characters are shown in Table 1-1.

Table 1-1: Common Escape Characters

Escape character	Result
\n	Newline (also known as *line feed*)
\t	Tab
\"	"
\\	\
\r	Carriage return

Finally, the program ends with this statement:

```
return (0);
```

This causes the program to stop and exit, returning an exit code of 0 to the operating system, which indicates that the program terminated normally. A nonzero exit code indicates an error.

Adding Comments

So far we've confined ourselves exclusively to writing code. In other words, everything we've seen is designed to be read by the computer and processed. Programs can also contain *comments*, which aren't seen by the compiler; instead, they're designed to be read by the person viewing the program. Comments commonly begin with /* and end with */. For example, the following is a comment:

```
/* Hello World - A nothing program */
```

It tells you what the programmer who wrote this thought of the program. Let's put some comments at the beginning of our program:

```
/*
 * Hello World -- not the most complicated program in
 *      the universe but useful as a starting point.
 *
 * Usage:
 *      1. Run the program.
 *      2. See the world.
 */
```

Another style of comment starts with // and goes to the end of the line. As you see more programs, you can determine for yourself which is better to use.

Always add comments to your code when you write a program, because that's when you know what you are doing. Five minutes later, you might forget. Five days later, you *will* forget. For example, I once had to do a complex bitmap transformation in order to translate a raster image into a firing command for an inkjet nozzle. The transformation involved taking a horizontal raster image, turning the row data into column data for the nozzles, and then, since the nozzles were offset, shifting the data left to match the nozzle location. I wrote out a page of comments describing every factor that affected the firing order. Then I added half a page of ASCII art diagramming what I had just described. Only after doing this and making sure I understood the problem did I write the code. And because I had to organize my thoughts in order to document them, the program worked after the first try.

When creating the answers to the programming problems presented in this book, get in the habit of writing comments. The really good programmers are fanatical comment writers.

Improving the Program and Build Process

When it comes to our little "Hello World" program, manually compiling it isn't a problem. But for a program with thousands of modules in it, keeping track of what needs to be compiled and what doesn't can be quite difficult. We need to automate the process to be efficient and avoid human error.

In this section, we tweak our program to improve it and automate the build process. Ideally, you should be able to build a program using a single command and no parameters, which would indicate you have a consistent and precise build process.

The make Program

One problem with our build process is that we have to enter the compilation command each time we build the program. This would be tedious for a program with several thousand files in it, each of which would need to be compiled. To automate the build process, we'll use the `make` program. It takes as its input a file called a makefile, which tells `make` how to build a program.

Create a file called *Makefile* containing the following on macOS or Linux:

```
CFLAGS=-ggdb -Wall -Wextra

all: hello

hello: hello.c
       gcc $(CFLAGS) -o hello hello.c
```

On Windows, the makefile should contain the following:

```
CFLAGS=-ggdb -Wall -Wextra
 +
all: hello.exe

hello.exe: hello.c
       gcc $(CFLAGS) -o hello.exe hello.c
```

It's important that the indented lines begin with a tab character. Eight spaces won't work. (Horrible file design, but we're stuck with it.) The first line defines a macro. As a result of this definition, whenever we specify `$(CFLAGS)` in the makefile, the `make` program will replace this with `-ggdb -Wall -Wextra`. Next, we define the target `all`, which is the default target by convention. When `make` is run with no parameters, it tries to build the first one it sees. The definition of this target, `all: hello`, tells the `make` program, "When you try to build `all`, you need to build `hello`." The final two lines of the makefile are the specification for `hello` (or *hello.exe* on Windows). These tell `make` that `hello` is made from *hello.c* by executing the command `gcc $(CFLAGS) -o hello hello.c`. This command contains the macro we defined, `$(CFLAGS)`, which expands to `-ggdb -Wall -Wextra`. You'll notice that we added a couple of extra flags to our compilation. We'll discuss those in the next section.

Now let's make the program using the `make` command:

```
$ make
gcc -ggdb -Wall -Wextra -o hello hello.c
```

As you can see, the program ran the commands to build the executable. The `make` program is smart. It knows that `hello` is made from *hello.c*, so it will check the modification dates of these two files. If `hello` is newer, then it does not need to be recompiled, so if you attempt to build the program twice, you'll get the following message:

```
make: Nothing to be done for 'all'.
```

This is not always the correct behavior. If we change the flags in our makefile, we've changed the compilation process and should rebuild our program. However, `make` doesn't know about this change and won't rebuild the program unless we edit *hello.c* and save the file or delete the output file.

Compiler Flags

The GCC compiler takes a number of options. In fact, the list of options for this compiler exceeds eight pages. Fortunately, we don't have to worry about them all. Let's take a look at the ones we used for our program.

`-ggdb` Compiles the program so we can debug it. Mostly, this adds debugging information to the output file that allows the debugger to understand what is going on.

`-Wall` Turns on a set of warnings that will flag correct but questionable code. (This book will teach you not to write questionable code.)

`-Wextra` Turns on extra warnings in an effort to make our code more precise.

`-o hello` Puts the output of our program in the file *hello*. (This option is `-o hello.exe` for Windows users.)

How the Compiler Works Behind the Scenes

In order to best make use of the compiler, you need to understand what goes on behind the scenes when you run it. That's because, when you're writing software for embedded devices, you'll often need to circumvent some of the operations the compiler performs automatically, which consist of a number of steps:

1. The *source code* is run through a *preprocessor*, which handles all the lines that begin with `#`, called directives. In our original source file, this is the `#include` statement. Later, you will learn about additional directives.
2. The compiler proper takes the preprocessed source code and turns it into *assembly language* code. C code is supposedly machine-independent and can be compiled and run on multiple platforms. Assembly

language is machine-dependent and can be run on only one type of platform. (Of course, it is still possible to write C code that will work on only one machine. C tries to hide the underlying machine from you, but it does not prevent you from directly accessing it.)

3. The assembly language file is passed through an *assembler*, which turns it into an *object file*. The object file contains just our code. However, the program needs additional code to work. In our case, the object file for *hello.c* needs a copy of the `printf` function.
4. The *linker* takes the object code in the object file and combines (links) it with useful code already present on your computer. In this case, it's `printf` and all the code needed to support it.

Figure 1-1 illustrates the process. All these steps are hidden from you by the `gcc` command.

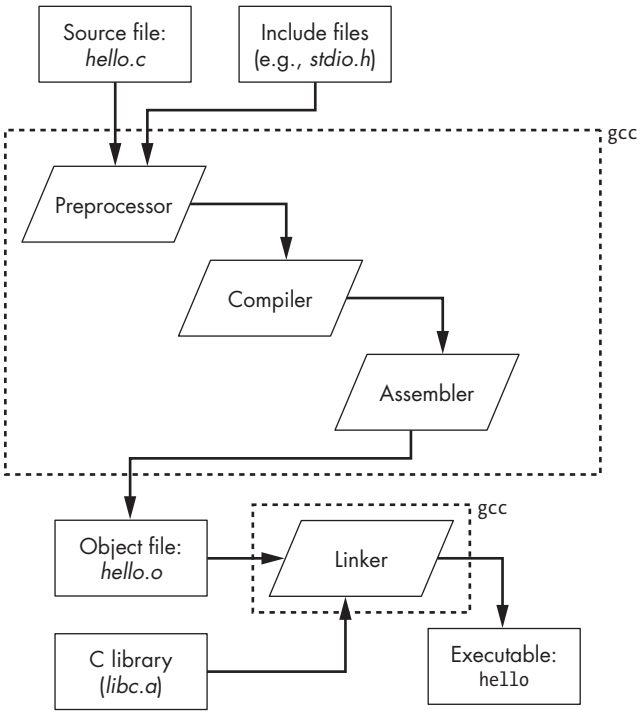

Figure 1-1: The steps needed to produce a program

You'll notice that the `gcc` command is acting as both compiler and linker. In fact, `gcc` is designed as a sort of executive program. It looks at the arguments and decides which other programs it needs to run in order to do its job. This might include the preprocessor (`cpp`), the C compiler (`cc1`), the assembler (`as`), the linker (`ld`), or other programs as needed. Let's walk through these components in more detail.

The Preprocessor

The first program run is the preprocessor, which is a *macro processor* (a type of automatic text editor) that handles all the lines that begin with `#`. In our program, it processes the `#include` line. We can get the output of the preprocessor with this command:

```
$ gcc -E hello.c >hello.i
```

The output of this command is stored in the *hello.i* file. If we look at this file, we see that it's more than 850 lines long. That's because the `#include <stdio.h>` line causes the entire *stdio.h* file to be copied into our program, and because the *stdio.h* file has its own `#include` directives, the files included by *stdio.h* get copied in as well.

We needed *stdio.h* for the `printf` function, and if we look through *hello.i*, we find the definition of this function, which is now included in our program:

```
extern int printf (const char *__restrict __format, ...);

extern int sprintf (char *__restrict __s,
      const char *__restrict __format, ...) __attribute__ ((__nothrow__));
```

The preprocessor also removes all the comments and annotates the text with information indicating what file is being processed.

The Compiler

Next, the compiler turns the C language code into assembly language. We can see what's generated with this command:

```
$ gcc -S hello.c
```

This should produce a file that starts with the following lines:

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello world!"
```

Notice that the compiler translated the C string `"Hello World!\n"` to the assembly language `.string` command. If you have sharp eyes, you'll also notice that the `\n` is missing. We'll discover why a little later.

The Assembler

The assembly language file goes into the assembler, where it is translated into machine code. The `gcc` command has an option (`-Wa`) that lets us pass flags to the assembler. Since it's impossible to understand the machine code unless

you're a machine, we will use the following command to ask for an assembly language listing that prints the machine code in human-readable format, with the corresponding assembly language statements that generated that code:

```
$ gcc -Wall -Wextra -g -Wextra -Wa,-a=hello.lst -c hello.c
```

The `-Wa` option tells GCC that what follows is to be passed to the assembler. The `-a=hello.lst` option tells the assembler to produce a listing called *hello.lst*. Let's take a look at that file. It begins as follows:

```
4                            .section        .rodata
5                    .LC0:
6 0000 48656C6C              .string "Hello world!"
6      6F20776F
6      726C6421
6      00
```

Assembly language differs on each machine. In this file, you're looking at x86 assembly language. It might seem like a confused mess, even in comparison to other assembly languages. You probably won't understand it completely, and that's okay; this chapter should only give you a sense of what assembly language looks like. In later chapters, when we get to the ARM processor, you'll see a much saner and easier-to-understand assembly.

The first column is a line number from the assembly language file. The second column, if present, indicates the address of the data being stored. All computer memory slots have a numerical address. In this case, the string `"Hello World!"` is being stored at address 0000 relative to the section that is currently being used (in this case, a section titled `.rodata`). When we discuss the linker, we'll see how this relative address is translated into an absolute one.

The next column contains the numerical values to be stored in memory in hexadecimal format. Then comes the text of the assembly language code itself. In the file, we can see that the `.string` directive tells the assembler to generate the codes for a text string.

Later in the file, we find the code for `main`:

```
15 0000 55                   pushq   %rbp
16                           .cfi_def_cfa_offset 16
17                           .cfi_offset 6, -16
18 0001 4889E5               movq    %rsp, %rbp
19                           .cfi_def_cfa_register 6
12:hello.c       ****     printf("Hello world!\n");
20                           .loc 1 12 0
21 0004 BF000000             movl    $.LC0, %edi
21      00
22 0009 E8000000             call    puts
22      00
```

On line 15, we can see the assembly language instruction `55`, which will be stored at location 0 in this section. This instruction corresponds to `pushq %rbp`, which does some bookkeeping at the start of the procedure. Also notice that some machine instructions are 1-byte long and others as long as

5 bytes. The instruction at line 21 is an example of a 5-byte instruction. You can see that this instruction is doing something with `.LC0`. If we look at the top of our listing, we see that `.LC0` is our string.

As a C programmer, you're not expected to fully understand what the assembly language does. Complete understanding would require absorbing several thousand pages of reference material. But we can, sort of, understand the instruction at line 22, which calls the function `puts`. This is where things get interesting. Remember that our C program didn't call `puts`—it called `printf`.

It seems that our code has been optimized behind the scenes. In embedded programming, "optimized" can be a dirty word, so it's important to understand what happened here. Essentially, the C compiler looked at the line `printf("Hello World!\n");` and decided it was identical to the following:

```
puts("Hello World!");
```

The truth is that these functions aren't actually identical: `puts` is a simple, efficient function, whereas `printf` is a large, complex one. But the programmer isn't using any of the advanced `printf` features, so the optimizer decided to rewrite the code to make it better. As a result, your `printf` call became `puts` and the end of line (`\n`) was removed from the string, as the `puts` call adds one automatically. When you get especially close to the hardware, little things like this can make a big difference, so it's important to know how to view and sort of understand assembly code.

The output of the assembler is an object file containing the code we wrote and nothing more. In particular, it does not contain the `puts` function, which we need. The `puts` function resides, along with hundreds of other functions, in the C standard library (*libc*).

The Linker

Our object file and some of the components of *libc* need to be combined to make our program. The linker's job is to take the files needed to make up the program, combine them, and assign real memory addresses to each component. As we did with the assembler, we can tell the `gcc` command to pass flags to the linker using this command:

```
$ gcc -Wall -Wextra -static -Wl,-Map=hello.map -o hello hello.o
```

The `-Wl` tells GCC to pass the option that follows (`-Map=hello.map`) to the linker. The map tells us where the linker put things in memory. (More on this later.) We've also added the directive `-static`, which changes the executable from dynamic to statically linked so that the memory map will look more like what we will see with our embedded systems. That way, we can avoid having to discuss the complexities of dynamic linking.

Object files such as *hello.o* are relocatable. That is, they can go anywhere in memory. It is the job of the linker to decide exactly where in memory they go. It is also the linker's job to go through the libraries used by the program, extract any needed object files, and include them in the final

program. The linker map tells us where things went and what library components were included in our program. For example, a typical linker entry might look like this:

```
.text           0x000000000040fa90      0x1c8 /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64
                                        -linux-gnu/libc.a(ioputs.o)
                0x000000000040fa90              puts
                0x000000000040fa90              _IO_puts
*fill*          0x000000000040fc58        0x8
```

Remember that we didn't write `puts`, even though it appears in this linker entry. As mentioned, it came from the standard C library file (*libc.a*). We can see here that the code for this function is located at `0x000000000040fa90`. This information can be useful if, say, our program crashed somewhere between `0x40Fa90` and `0x40fc58`. In that case, we would know that `puts` caused the crash.

We also know that `puts` takes up `0x1C8` bytes (`40fc58-40fa90`). This is 456 decimal bytes, or a little under .5K. The amount of memory will concern us when we start programming our microprocessor, which has limited memory.

You should now have a good idea of every element of a C program and what these various pieces do. Most of the time, you can let the compiler take care of these details without worrying about what's going on under the hood. But when you're programming small chips with limited resources, you do need to worry about what's going on inside.

Adding to Your Makefile

Explore the various aspects of the GCC compiler, assembler, and linker on your own by amending your makefile to generate all the files described in the previous section:

```
CFLAGS=-Wall -Wextra -ggdb

all: hello hello.i hello.s

hello.o: hello.c
        gcc $(CFLAGS) -Wa,-a=hello.lst -c hello.c

hello: hello.o
        gcc $(CFLAGS) -static -Wl,-Map=hello.map -o hello hello.o

hello.i: hello.c
        gcc -E hello.c >hello.i

hello.s: hello.c
        gcc -S hello.c

# Type "make verbose" to see the whole command line
verbose:
        gcc -v $(CFLAGS) -Wextra -c hello.c
```

```
clean:
        rm -f hello hello.i hello.s hello.o
```

As described earlier, the first non-blank line defines a macro that tells `make` to replace `$(CFLAGS)` with `-Wall -Wextra -ggdb` everywhere in the rest of the file. Next, we define a *target* (an item that needs to be built) named `all`. Since this is the first target in the file, it is also the default one, which means you can build it simply by entering the following:

```
$ make
```

This target is what we call a *phony target*, as it doesn't result in a file named *all*. Instead, every time you execute the `make all` command, `make` will check whether it needs to re-create its dependencies. You can see these dependencies listed in the makefile after the keyword `all` and the colon. In order to make the target `all`, we need to make the targets `hello`, `hello.i`, and `hello.s`. The following lines clarify how to make those targets. For example, to make the target `hello.i`, we must use the target `hello.c`. If `hello.i` is newer than `hello.c`, then `make` will do nothing. If `hello.c` has undergone recent changes and `hello.i` is not up to date, `make` will produce `hello.i` using the following command:

```
gcc -E hello.c >hello.i
```

Thus, if you edit `hello.c` and then execute the command `make hello.i`, you'll see `make` do its job:

```
$ (Change hello.c)
$ make hello.i
gcc =E hello.c > hello.i
```

Another target in our makefile, `clean`, removes all the generated files. To get rid of the generated files, execute the following command:

```
$ make clean
```

GNU `make` is a very sophisticated program with a manual that is more than 300 pages long. The good news is you need to deal with only a very small subset of its commands in order to be productive.

Summary

Making a "Hello World" program is one of the simplest things a C programmer can do. However, understanding everything that happens behind the scenes to create and run that C program is a bit more difficult. Luckily, you don't have to be an expert. But while you don't need to master every bit of the assembly language generated by the program, any embedded programmer should understand enough to be able to spot potential problems or

unusual behavior, such as `puts` showing up in a program that calls `printf`. Paying attention to these details will allow us to get the most out of our small machines.

Questions

1. Where does the documentation for GNU `make` reside?
2. Is C code portable between different types of machines?
3. Is assembly language code portable between different types of machines?
4. Why does a single statement in assembly language code generate just one machine instruction when one statement in C can generate many?