CHAPTER 3

Software

In this chapter we explore how a simple C program interacts with the hardware described in the previous chapter. We begin by introducing the virtual memory map and its relationship to the physical memory map. We then use the `simplePIC.c` program from Chapter 1 to explore the compilation process and the XC32 compiler installation.

3.1 The Virtual Memory Map

In the previous chapter we learned about the PIC32's physical memory map, which allows the CPU to access any SFR or any location in data RAM, program flash, or boot flash, using a 32-bit address. The PIC32 does not actually have 2^{32} bytes, or 4 GB, worth of SFRs and memory; therefore, many physical addresses are invalid.

Rather than use physical addresses (PAs), software refers to memory and SFRs using virtual addresses (VAs). The fixed mapping translation (FMT) unit in the CPU converts VAs into PAs using the following formula:

$$PA = VA \ \& \ 0x1FFFFFFF$$

This bitwise AND operation clears the three most significant bits of the address; thus multiple VAs map to the same PA.

If the mapping from the VA to the PA just discards the first three bits, why bother having them? Well, the CPU and the prefetch cache module we learned about in the previous chapter use them. If the first three bits of the virtual address are 0b100 (corresponding to an 8 or 9 as the most significant hex digit of the VA), then the contents of that memory address can be cached. If the first three bits are 0b101 (corresponding to an A or B as the most significant hex digit of the VA), then it cannot be cached. Thus the segment of virtual memory 0x80000000 to 0x9FFFFFFF is cacheable, while the segment 0xA0000000 to 0xBFFFFFFF is noncacheable. The cacheable segment is called KSEG0 (for "kernel segment") and the noncacheable segment is called KSEG1.

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00003-2

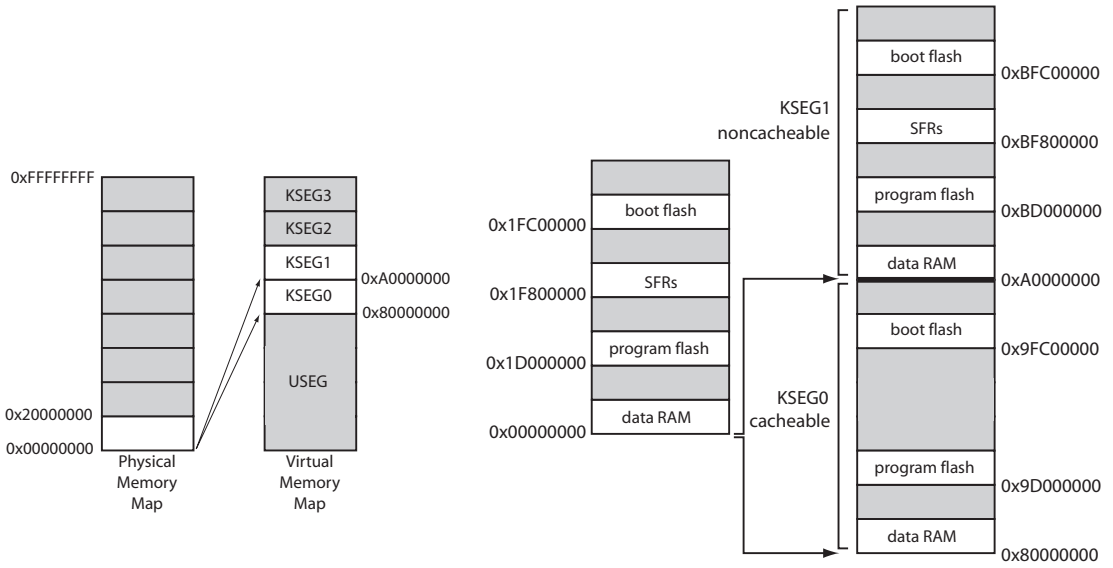

Figure 3.1

(Left) The 4 GB physical and virtual memory maps are divided into 512 MB segments. The mapping of the valid physical memory addresses to the virtual memory regions KSEG0 and KSEG1 is illustrated. We use only KSEG0 and KSEG1, not KSEG2, KSEG3, or the user segment USEG. (Right) Physical addresses mapped to virtual addresses in cacheable memory (KSEG0) and noncacheable memory (KSEG1). Note that SFRs are not cacheable. The last four words of boot flash, 0xBFC02FF0 to 0xBFC02FFF in KSEG1, correspond to the device configuration words DEVCFG0 to DEVCFG3. Memory regions are not drawn to scale.

Figure 3.1 illustrates the relationship between the physical and virtual memory maps. Note that the SFRs are excluded from the KSEG0 cacheable virtual memory segment. SFRs correspond to physical devices (e.g., peripherals); therefore their values cannot be cached. Otherwise, the CPU could read outdated SFR values because the state of the SFR could change between when it was cached and when it was needed by the CPU. For instance, if port B were configured as a digital input port, the SFR PORTB would contain the current input values of some pins. The voltage on these pins could change at any time; therefore, the only way to retrieve a reliable value is to read directly from the SFR rather than from the cache.

Also note that program flash and data RAM can be accessed using either cacheable or noncacheable VAs. Typically, you can ignore this detail because the PIC32 will be configured to access program flash via the cache (since flash memory is slow), and data RAM without the cache (since RAM is fast).

Going forward, we will use virtual addresses like 0x9D000000 and 0xBD000000, and you should realize that these refer to the same physical address. Since virtual addresses start at 0x80000000, and all physical addresses are below 0x20000000, there is no possibility of confusing whether we are talking about a VA or a PA.

3.2 An Example: `simplePIC.c`

Let us build the `simplePIC.c` executable from Chapter 1. For convenience, here is the program again:

Code Sample 3.1 `simplePIC.c`**. Blinking Lights, Unless the USER Button Is Pressed.**

```
#include <xc.h>          // Load the proper header for the processor

void delay(void);

int main(void) {
  TRISF = 0xFFFC;        // Pins 0 and 1 of Port F are LED1 and LED2.  Clear
                         // bits 0 and 1 to zero, for output.  Others are inputs.
  LATFbits.LATF0 = 0;    // Turn LED1 on and LED2 off.  These pins sink current
  LATFbits.LATF1 = 1;    // on the NU32, so "high" (1) = "off" and "low" (0) = "on"

  while(1) {
    delay();
    LATFINV = 0x0003;    // toggle LED1 and LED2; same as LATFINV = 0x3;
  }
  return 0;
}

void delay(void) {
  int j;
  for (j = 0; j < 1000000; j++) { // number is 1 million
    while(!PORTDbits.RD7) {
        ;   // Pin D7 is the USER switch, low (FALSE) if pressed.
    }
  }
}
```

Navigate to the `<PIC32>` directory. Following the same procedure as in Chapter 1.3, build `simplePIC.hex` and load it onto your NU32. We have reprinted the instructions here (you may need to specify the full path to these commands):

```
> xc32-gcc -mprocessor=32MX795F512H
   -o simplePIC.elf -Wl,--script=skeleton/NU32bootloaded.ld simplePIC.c
> xc32-bin2hex simplePIC.elf
> nu32utility <COM> simplePIC.hex
```

When you have the program running, the NU32's two LEDs should alternate on and off and stop while you press the USER button.

Look at the source code: the program refers to SFRs named TRISF, LATFINV, etc. These names align with the SFR names in the Data Sheet and Reference Manual sections on input/output (I/O) ports. We will often consult the Data Sheet and Reference Manual when programming the PIC32. We will explain the use of these SFRs shortly.

3.3 What Happens When You Build?

First, let us begin to understand what happens when you create `simplePIC.hex` from `simplePIC.c`. Refer to Figure 3.2.

First the **preprocessor** removes comments and inserts `#included` header files. It also handles other preprocessor instructions such as `#define`. You can have multiple `.c` C source files and `.h` header files, but only one C file is allowed to have a `main` function. The other files may contain helper functions. We will learn more about projects with multiple C source files in Chapter 4.

Then the **compiler** turns the C files into MIPS32 assembly language files, machine instructions specific to the PIC32's MIPS32 CPU. Basic C code will not vary between processor architectures, but assembly language may be completely different. These assembly files are readable by a text editor, and it is possible to program the PIC32 directly in assembly language.

The **assembler** turns the assembly files into machine-level *relocatable object code*. This code cannot be inspected with a text editor. The code is called relocatable because the final memory addresses of the program instructions and data used in the code are not yet specified. The **archiver** is a utility that allows you to package several related `.o` object files into a single `.a` library file. We will not be making our own archived libraries, but we will certainly be using `.a` libraries that have already been made by Microchip!

Finally, the **linker** takes one or more object files and combines them into a single executable file, with all program instructions assigned to specific memory locations. The linker uses a linker script that has information about the amount of RAM and flash on your particular PIC32, as well as directions about where in virtual memory to place the data and instructions. The result is an executable and linkable format (`.elf`) file, a standard executable file format. This file contains useful debugging information as well as information that allows tools such as `xc32-objdump` to *disassemble* the file, which converts it back into assembly code (Section 3.8). This extra information adds up; building `simplePIC.c` results in a `.elf` file that is hundreds of kilobytes! A final step creates a stripped-down `.hex` file of less than 10 KB. This `.hex` file is a representation of your executable suitable for sending to the bootloader program on your PIC32 (more on this in the next section) that writes the program into flash on your PIC32.

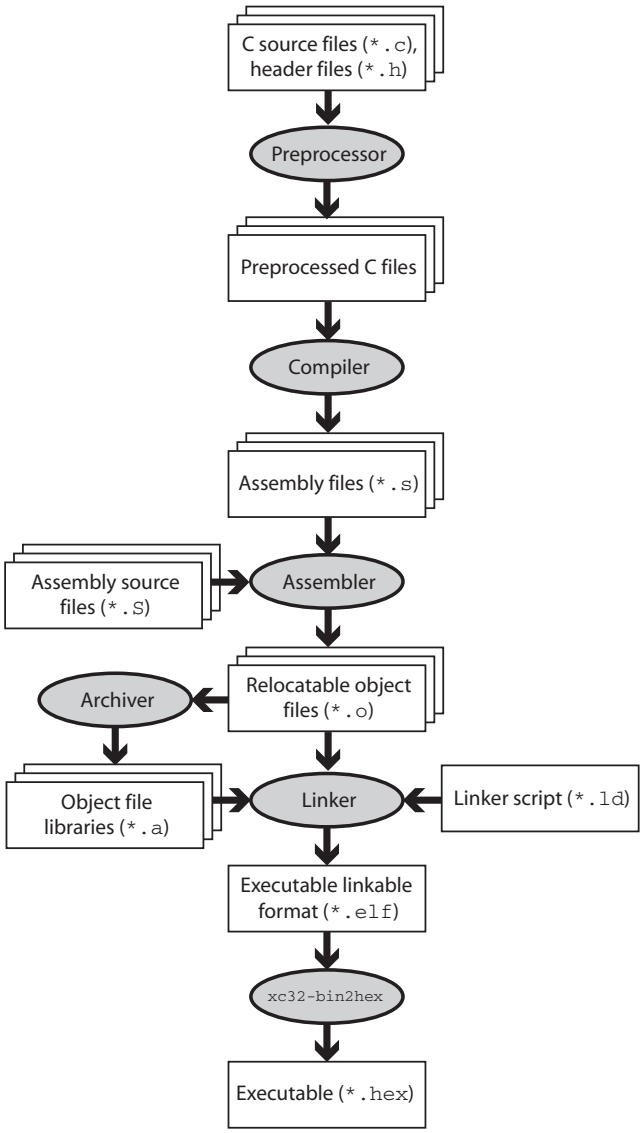

Figure 3.2
The "compilation" process.

Although the entire process consists of several steps, it is often referred to as "compiling" for short. "Building" or "making" is more accurate.

3.4 What Happens When You Reset the PIC32?

Your program is running. You hit the RESET button on the NU32. What happens next?

First the CPU jumps to the beginning of boot flash, address 0xBFC00000, and starts executing instructions.[1] For the NU32, the boot flash contains the *bootloader*, a program used to load other programs onto the PIC32. The bootloader first checks to see if you are pressing the USER button. If so, it knows that you want to reprogram the PIC32, so it attempts to communicate with the bootloader utility (`nu32utility`) on your computer. With communication established, the bootloader receives the executable `.hex` file and writes it to the PIC32's program flash (see Exercise 2). We refer to the virtual address where your program is installed as `_RESET_ADDR`.

> **Note:** The PIC32's reset address 0xBFC00000 is hardwired and cannot be changed. The address where the bootloader writes your program, however, can be changed in software.

Now assume that you were not pressing the USER button when you reset the PIC32. In that case the bootloader jumps to the address `_RESET_ADDR` and begins executing the program you previously installed there. Notice that our program, `simplePIC.c`, is an infinite loop, so it never stops executing, the desired behavior in embedded control. If a program exits, the PIC32 will sit in a tight loop, doing nothing until it is reset. (Interrupts, described in Chapter 6, will continue to execute.)

3.5 Understanding `simplePIC.c`

Let us return to understanding `simplePIC.c`. The `main` function initializes values of TRISF and LATFbits, then enters an infinite `while` loop. Each time through the loop it calls `delay()` and then assigns a value to LATFINV. The `delay` function executes a `for` loop that iterates one million times. During each iteration it enters a `while` loop, which checks the value of (!PORTDbits.RD7). If PORTDbits.RD7 is 0 (FALSE), then the expression (!PORTDbits.RD7) evaluates to TRUE, and the program remains here, doing nothing except checking the expression (!PORTDbits.RD7). When this expression evaluates to FALSE, the `while` loop exits, and the program continues with the `for` loop. After the `for` loop finishes, control returns to `main`.

Special function registers (SFRs)

The main difference between `simplePIC.c` and programs that you may have written for your computer is how it interacts with the outside world. Rather than via keyboard or mouse,

[1] If you are just powering on your PIC32, it will wait a short period while electrical transients die out, clocks synchronize, etc., before jumping to 0xBFC00000.

`simplePIC.c` accesses SFRs like TRISF, LATF, and PORTD, all of which correspond to peripherals. Specifically, TRISF, LATF, and PORTD refer to the digital I/O ports F and D. Digital I/O ports allow the PIC32 to read or set the digital voltage on a pin. To discover what these SFRs control we start by consulting the table in Section 1 of the Data Sheet, which lists the pinout descriptions. For example, we see that port D, with pins named RD0 to RD11, has 12 pins, and port F, with pins RF0, RF1, RF3, RF4, and RF5, has five pins. Port B has 16 pins, labeled RB0 to RB15.

We now turn to the Data Sheet section on I/O Ports for more information. We find that TRISF, short for "tri-state F," controls the direction, input or output, of the pins on port F. Each port F pin has a corresponding bit in TRISF. If this bit is 0, the pin is an output. If the bit is a 1, the pin is an input. ($0 = \text{O}_{\text{utput}}$ and $1 = \text{I}_{\text{nput}}$.) We can make some pins inputs and some outputs, or we can make them all have the same direction.

If you are curious about which direction the pins are by default, you can consult the Memory Organization section of the Data Sheet. Tables there list the VAs of many of the SFRs, as well as the values they default to upon reset. There are a lot of SFRs! After some searching, you will find that TRISF sits at virtual address 0xBF886140, and its default value upon reset is 0x0000003B. (We have reproduced part of this table for you in Figure 3.3.) In binary, this would be

$$0\text{x}0000003\text{B} = 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0011 \quad 1011.$$

The four most significant hex digits (two bytes, or 16 bits) are all 0. This is because those bits, technically, do not exist. Microchip calls them "unimplemented." No I/O port has more than 16 pins, so we do not need those bits, which are numbered 16-31. (The 32 bits are numbered 0-31.) Of the remaining bits, since the 0th bit (least significant bit) is the rightmost bit, we see that bits 0, 1, 3, 4, and 5 are 1, while the rest are 0. The bits set to 1 correspond precisely to the pins we have available, meaning that they are inputs. (The other pins are unimplemented.). I/O pins are configured as inputs on reset for safety reasons; when we power on the PIC32, each pin will take its default direction before the program can change it. If an output pin were

Virtual address (BF88_#)	Register name(1)	Bit range	Bits																All resets
			31/15	30/14	29/13	28/12	27/11	26/10	25/9	24/8	23/7	22/6	21/5	20/4	19/3	18/2	17/1	16/0	
6140	TRISF	31:16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
		15:0	—	—	—	—	—	—	—	—	—	—	TRISF5	TRISF4	TRISF3	—	TRISF1	TRISF0	003B

Figure 3.3
The TRISF SFR, taken from the PIC32 Data Sheet.

connected to an external circuit that is also trying to control the voltage on the pin, the two devices would fight each other, with damage to one or the other a possibility. No such problems arise if the pin is configured as an input by default.

So now we understand that the instruction

```
TRISF = 0xFFFC;
```

clears bits 0 and 1, implicitly clears bits 16-31 (which is ignored, since the bits are not implemented), and sets the rest of the bits to 1. It does not matter that we try to set some unimplemented bits to 1; those bits are ignored. The result is that port F pins 0 and 1, or RF0 and RF1 for short, are now outputs.

Our PIC32 C compiler allows the use of binary (base 2) representations of unsigned integers using 0b at the beginning of the number, so if you do not get lost counting bits, you could have equally written

```
TRISF = 0b1111111111111100;
```

or simply

```
TRISF = 0b111100;
```

since no bits are implemented after RF5.

Another option would have been to use the instructions

```
TRISFbits.TRISF0 = 0; TRISFbits.TRISF1 = 0;
```

This allows us to specify individual bits without affecting the other bits. We see this kind of notation later in the program, with LATFbits.LATF0 and LATFbits.LATF1, for example.

The two other basic SFRs in this program are LATF and PORTD. Again consulting the I/O Ports section of the Data Sheet, we see that LATF, short for "latch F," is used to write values to the output pins. Thus

```
LATFbits.LATF1 = 1;
```

sets pin RF1 high. Finally, PORTD contains the digital inputs on the port D pins. (Notice we did not configure port D as input; we relied on the fact that it's the default.) PORTDbits.RD7 is 0 if 0 V is present on pin RD7 and 1 if approximately 3.3 V is present. Note that we use the latch when writing pins and the port when reading pins, for reasons explained in Chapter 7.

Pins RF0, RF1, and RD7 on the NU32

Figure 2.3 shows how pins RF0, RF1 and RD7 are wired on the NU32 board. LED1 (LED2) is on if RF0 (RF1) is 0 and off if it is 1. When the USER button is pressed, RD7 registers a 0, and otherwise it registers a 1.

The result of these electronics and the `simplePIC.c` program is that the LEDs flash alternately, but remain unchanging while you press the USER button.

CLR, SET, and INV SFRs

So far we have ignored the instruction

```
LATFINV = 0x0003;
```

Again consulting the Memory Organization section of the Data Sheet, we see that associated with the SFR LATF are three more SFRs, called LATFCLR, LATFSET, and LATFINV. (Indeed, many SFRs have corresponding CLR, SET, and INV SFRs.) These SFRs are used to easily change some of the bits of LATF without affecting the others. A write to these registers causes a change to LATF's bits, but only in the bits corresponding to bits on the right-hand side that have a value of 1. For example,

```
LATFINV = 0x3;      // flips (inverts) bits 0 and 1 of LATF; all others unchanged
LATFINV = 0b11;     // same as above
LATFSET = 0x9;      // sets bits 0 and 3 of LATF to 1; all others unchanged
LATFCLR = 0x2;      // clears bit 1 of LATF to 0;  all others unchanged
```

A nominally less efficient way to toggle bits 0 and 1 of LATF is

```
LATA5LATFbits.LATF0 = !LATFbits.LATF0; LATFbits.LATF1 = !LATFbits.LATF1;
```

The compiler, however, sometimes optimizes these instructions into the equivalent more efficient operation. We shall look at efficiency in Chapter 5. In most cases, the difference between the methods is negligible so you should access the fields using the bit structures (e.g., LATFbits.LATF0) for code clarity.

You can return to the table in the Data Sheet to see the VAs of the CLR, SET, and INV registers. They are always offset from their base register by 4, 8, and 12 bytes, respectively. Since LATF is at 0xBF886160, LATFCLR, LATFSET, and LATFINV are at 0xBF886164, 0xBF886168, and 0xBF88616C, respectively.

You should now understand how `simplePIC.c` works. But we have ignored the fact that we never declared TRISF, LATFINV, etc., before we started using them. We know you cannot do

that in C; these variables must be declared somewhere. The only place they could be declared is in the included file `xc.h`. We have ignored that `#include <xc.h>` statement until now. Time to take a look.[2]

3.5.1 Down the Rabbit Hole

Where do we find `xc.h`? The line `#include <xc.h>` means that the preprocessor will look for `xc.h` in directories specified in the *include path*.

For us, the default include path means that the compiler finds `xc.h` sitting at

```
<xc32dir>/<xc32ver>/pic32mx/include/xc.h
```

You should substitute your install directory in place of `<xc32dir>/<xc32ver>`.

Including `xc.h` gives us access to many data types, variables, and constants that Microchip has provided for our convenience. In particular, it provides variable declarations for SFRs like TRISF, allowing us to access the SFRs from C.

Before we open `xc.h`, let us examine the directory structure of the XC32 compiler installation. There's a lot here! We certainly do not need to understand it all now, but we should get a sense of what's going on. We start at the level of your XC32 install directory and summarize the important nested directories, without being exhaustive.

1. `bin`: Contains the actual executable programs that do the compiling, assembling, linking, etc. For example, `xc32-gcc` is the C compiler.
2. `docs`: Some manuals, including the XC32 C Compiler User's Guide, and other documentation.
3. `examples`: Some sample code.
4. `lib`: Contains some `.h` header files and `.a` library archives containing general C object code.
5. `pic32-libs`: This directory contains the source code (`.c` C files, `.h` header files, and `.S` assembly files) needed to create numerous Microchip-provided libraries. These files are provided for reference and are not included directly in any of your code.
6. `pic32mx`: This directory has several files we are interested in because many of them end up in your project.

[2] Microchip often changes the software it distributes, so there may be differences in details, but the essence of what we describe here will be the same.

a. `lib`: This directory consists mostly of PIC32 object code and libraries that are linked with our compiled and assembled source code. For some of these libraries, source code exists in `pic32-libs`; for others we have only the object code libraries. Some important files in this directory include:
   - `proc/32MX795F512H/crt0_mips32r2.o`: The linker combines this object code with your program's object code when it creates the `.elf` file. The linker ensures that this "C Runtime Startup" code is executed first, since it performs various initializations your code needs to run, such as initializing the values of global variables. Different PIC32 models have different versions of this file under the appropriate `proc/<processor>` directory. You can find readable assembly source code at `pic32-libs/libpic32/startup/crt0.S`.
   - `libc.a`: Implementations of functions that are part of the C standard library.
   - `libdsp.a`: This library contains MIPS implementations of finite and infinite impulse response filters, the fast Fourier transform, and various vector math functions.
   - `proc/32MX795F512H/processor.o`: This object file provides the virtual memory addresses for the PIC32's SFRs; each specific model has its own `processor.o` file. We cannot look at it directly with a text editor, but there are utilities that allow us to examine it. For example, from the command line you could use the `xc32-nm` program in the top-level `bin` directory to see all the SFR VAs:

     ```
     > xc32-nm processor.o
     bf809040 A AD1CHS
     ...
     bf886140 A TRISF
     bf886144 A TRISFCLR
     bf88614c A TRISFINV
     bf886148 A TRISFSET
     ...
     ```

     All of the SFRs are printed out, in alphabetical order, with their corresponding VA. The spacing between SFRs is four, since there are four bytes (32 bits) in an SFR. The "A" means that these are absolute addresses. The linker must use these addresses when making final address assignments because the SFRs are implemented in hardware and cannot be moved! The listing above indicates that TRISF is located at VA 0xBF886140, agreeing with the Memory Organization section of the Data Sheet.
   - `proc/32MX795F512H/configuration.data`: This file describes some constants used in setting the configuration bits in DEVCFG0 to DEVCFG3 (Chapter 2.1.4). These bits are set by the bootloader (Section 3.6), so you do not need to worry about them in your programs. It is possible to use a programmer device to load programs onto the PIC32 without having a bootloader pre-installed on the PIC32 (that's how the

bootloader got there in the first place!), in which case you would need to worry about these bits. See Section 3.6 for more information about programs that do not use a bootloader.

b. `include`: This directory contains several `.h` header files.
   - `cp0defs.h`: This file defines constants and macros that allow us to access functions of coprocessor 0 (CP0) on the MIPS32 M4K CPU. In particular, it allows us to read and set the *core timer* clock that ticks once every two SYSCLK cycles using macros like `_CP0_GET_COUNT()` (see Chapters 5 and 6 for more details). More information on CP0 can be found in the "CPU for Devices with the M4K Core" section of the Reference Manual.
   - `sys/attribs.h`: In the directory `sys`, the file `attribs.h` defines the macro syntax `__ISR` that we will use for interrupt service routines starting in Chapter 6.
   - `sys/kmem.h`: Contains macros for converting between physical and virtual addresses.
   - `xc.h`: This is the file we include in `simplePIC.c`. The main purpose of `xc.h` is to include the appropriate processor-specific header file, in our case `include/proc/p32mx795f512h.h`. It does this by checking if `__32MX795F512H__` is defined:

     ```
     #elif defined(__32MX795F512H__)
     #include <proc/p32mx795f512h.h>
     ```

     If you look at the command for compiling `simplePIC.c`, you may notice the option `-mprocessor=32MX795F512H`. This option defines the constant `__32MX795F512H__` to the compiler, allowing `xc.h` to function properly. This file also defines some macros for easily inserting some specific assembly instructions directly from C.
   - `proc/p32mx795f512h.h`: Open this file in your text editor. Whoa! This file is over 40,000 lines long! It must be important. Time to look at it in more detail.

3.5.2 The Header File `p32mx795f512h.h`

The first 31% of `p32mx795f512h.h`, about 14,000 lines, consists of code like this, with line numbers added to the left for reference:

```
1  extern volatile unsigned int        TRISF __attribute__((section("sfrs")));
2  typedef union {
3    struct {
4      unsigned TRISF0:1;   // TRISF0 is bit 0 (1 bit long), interpreted as
                               unsigned int
5      unsigned TRISF1:1;   // bits are in order, so the next bit, bit 1, is
                               TRISF1
6      unsigned TRISF2:1;   // TRISF2 doesn't actually exist (unimplemented)
7      unsigned TRISF3:1;
8      unsigned TRISF4:1;
```

```
 9      unsigned TRISF5:1;  // later bits are not given names, since they're
                               unimplemented
10    };
11    struct {
12      unsigned w:32;      // w refers to all 32 bits
13    };
14  } __TRISFbits_t;
15  extern volatile __TRISFbits_t TRISFbits __asm__ ("TRISF") __attribute__
    ((section("sfrs")));
16  extern volatile unsigned int        TRISFCLR __attribute__((section("sfrs")));
17  extern volatile unsigned int        TRISFSET __attribute__((section("sfrs")));
18  extern volatile unsigned int        TRISFINV __attribute__((section("sfrs")));
```

The first line, beginning `extern`, declares the variable TRISF as an `unsigned int`. The keyword `extern` means that no RAM has to be allocated for it; memory to hold the variable has been allocated for it elsewhere. In a typical C program, memory for the variable has been allocated by another C file using syntax without the `extern`, like `volatile unsigned int TRISF;`. In this case, however, no RAM has to be allocated for TRISF because it refers to an SFR, not a word in RAM. The `processor.o` file actually defines the VA of the symbol TRISF, as mentioned earlier.

The `volatile` keyword, applied to all the SFRs, means that the value of this variable could change without the CPU knowing it. Thus the compiler should generate assembly code to reload TRISF into the CPU registers every time it is used, rather than assuming that its value is unchanged just because no C code has modified it.

Finally, the `__attribute__` syntax tells the linker that TRISF is in the `sfrs` section of memory.

The next section of code, lines 2-14, defines a new data type called `__TRISFbits_t`. Next, in line 15, a variable named TRISFbits is declared of type `__TRISFbits_t`. Again, since it is an `extern` variable, no memory is allocated, and the `__asm__ ("TRISF")` syntax means that TRISFbits is at the same VA as TRISF.

It is worth understanding the new data type `__TRISFbits_t`. It is a `union` of two `structs`. The `union` means that the two `structs` share the same memory, a 32-bit word in this case. Each `struct` is called a *bit field*, which gives names to specific groups of bits within the 32-bit word. Thus declaring a variable TRISFbits of type `__TRISFbits_t`, and forcing it to be located at the same VA as `TRISF` allows us to use syntax like TRISFbits.TRISF0 to refer to bit 0 of TRISF.

A named set of bits in a bit field need not be one bit long; for example, TRISFbits.w refers to the entire `unsigned int` TRISF, created from all 32 bits. The type `__RTCALRMbits_t` defined earlier in the file by

```
typedef union {
  struct {
    unsigned ARPT:8;
```

```
    unsigned AMASK:4;
 ...
} __RTCALRMbits_t;
```

has a first field ARPT that is eight bits long and a second field AMASK that is four bits long. Since RTCALRM is a variable of type `__RTCALRMbits_t`, a C statement of the form `RTCALRMbits.AMASK = 0xB` would put the values 1, 0, 1, 1 in bits 11, 10, 9, 8, respectively, of RTCALRM.

After the declaration of TRISF and TRISFbits, lines 16-18 contain declarations of TRISFCLR, TRISFSET, and TRISFINV. These declarations allow `simplePIC.c`, which uses these variables, to compile successfully. When the object code of `simplePIC.c` is linked with the `processor.o` object code, references to these variables are resolved to the proper SFR VAs.

With these declarations in `p32mx795f512h.h`, the `simplePIC.c` statements

```
TRISF = 0xFFFC;
LATFINV = 0x0003;
while(!PORTDbits.RD7)
```

finally make sense; these statements write values to, or read values from, SFRs at VAs specified by `processor.o`. You can see that `p32mx795f512h.h` declares many SFRs, but no RAM has to be allocated for them; they exist at fixed addresses in the PIC32's hardware.

The next 9% of `p32mx795f512h.h` is the `extern` variable declaration of the same SFRs, without the bit field types, for assembly language. The VAs of each of the SFRs is given, making this a handy reference.

Starting at just over 17,000 lines into the file, we see more than 20,000 lines with constant definitions like the following:

```
#define _T1CON_TCS_POSITION                     0x00000001
#define _T1CON_TCS_MASK                         0x00000002
#define _T1CON_TCS_LENGTH                       0x00000001

#define _T1CON_TCKPS_POSITION                   0x00000004
#define _T1CON_TCKPS_MASK                       0x00000030
#define _T1CON_TCKPS_LENGTH                     0x00000002
```

These refer to the Timer1 SFR T1CON. Consulting the information about T1CON in the Timer1 section of the Data Sheet, we see that bit 1, called TCS, controls whether Timer1's clock input comes from the T1CK input pin or from PBCLK. Bits 4 and 5, called TCKPS for "timer clock prescaler," control how many times the input clock has to "tick" before Timer1 is incremented (e.g., TCKPS = 0b10 means there is one clock increment per 64 input ticks). The constants defined above are for convenience in accessing these bits. The `POSITION` constants indicate the least significant bit location in TCS or TCKPS in T1CON—one for TCS and four for TCKPS. The `LENGTH` constants indicate that TCS consists of one bit and TCKPS consists of

two bits. Finally, the MASK constants can be used to determine the values of the bits we care about. For example:

```
unsigned int tckpsval = (T1CON & _T1CON_TCKPS_MASK) >> _T1CON_TCKPS_POSITION;
// AND MASKing clears all bits except 5 and 4, which are unchanged and shifted to
// positions 1 and 0, so tckpsval now contains the value T1CONbits.TCKPS
```

The definitions of the POSITION, LENGTH, and MASK constants take up most of the rest of the file. Of course, there is also a T1CONbits defined that allows you to access these bits directly (e.g., T1CONbits.TCKPS). We recommend that you use this method, as it is typically clearer and less error prone than performing direct bit manipulations.

At the end, some more constants are defined, like below:

```
#define _ADC10
#define _ADC10_BASE_ADDRESS    0xBF809000
#define _ADC_IRQ               33
#define _ADC_VECTOR            27
```

The first is merely a flag indicating to other .h and .c files that the 10-bit ADC is present on this PIC32. The second indicates the first address of 22 consecutive SFRs related to the ADC (see the Memory Organization section of the Data Sheet). The third and fourth relate to interrupts. The PIC32MX's CPU is capable of being interrupted by up to 96 different events, such as a change of voltage on an input line or a timer rollover event. Upon receiving these interrupts, it can call up to 64 different interrupt service routines, each identified by a "vector" corresponding to its address. These two lines say that the ADC's "interrupt request" line is 33 (out of 0 to 95), and its corresponding interrupt service routine is at vector 27 (out of 0 to 63). Interrupts are covered in Chapter 6.

Finally, p32mx795f512h.h concludes by including ppic32mx.h, which contains legacy code that is no longer needed but remains for backward compatibility with old programs.

3.5.3 Other Microchip Software: Harmony

Installed in your Harmony directory (<harmony>) is an extensive and complex set of libraries and sample code written by Microchip. Because of the complexity and abstraction it introduces, we avoid using Harmony functions until Chapter 20, when our programs are complex enough that low-level access to the peripherals through SFRs becomes more difficult.[3]

[3] Even though most sample code in the book does not use Harmony, we had you install it and record its installation directory in the Makefile; this way, your system is prepared for Harmony when we are ready to use it.

3.5.4 The NU32bootloaded.ld Linker Script

To create the executable .hex file, we needed the C source file simplePIC.c and the linker script NU32bootloaded.ld. Examining NU32bootloaded.ld with a text editor, we see the following line near the beginning:

```
INPUT("processor.o")
```

This line tells the linker to load the processor.o file specific to your PIC32. This allows the linker to resolve references to SFRs (declared as extern variables in p32mx795f512h.h) to actual addresses.

The rest of the NU32bootloaded.ld linker script has information such as the amount of program flash and data memory available, as well as the virtual addresses where program elements and global data should be placed. Below is a portion of NU32bootloaded.ld:

```
_RESET_ADDR             = (0xBD000000 + 0x1000 + 0x970);

/*************************************************************************
 * NOTE:  What is called boot_mem and program_mem below do not directly
 * correspond to boot flash and program flash.  For instance, here
 * kseg0_boot_mem and kseg1_boot_mem both live in program flash memory.
 * (We leave the boot flash solely to the bootloader.)
 * The boot_mem names below tell the linker where the startup codes should
 * go (here, in program flash).  The first 0x1000 + 0x970 + 0x490 = 0x1E00 bytes
 * of program flash memory is allocated to the interrupt vector table and
 * startup codes.  The remaining 0x7E200 is allocated to the user's program.
 *************************************************************************/
MEMORY
{
  /* interrupt vector table */
  exception_mem             : ORIGIN = 0x9D000000, LENGTH = 0x1000
  /* Start-up code sections; some cacheable, some not */
  kseg0_boot_mem            : ORIGIN = (0x9D000000 + 0x1000), LENGTH = 0x970
  kseg1_boot_mem            : ORIGIN = (0xBD000000 + 0x1000 + 0x970), LENGTH =
                              0x490
  /* User's program is in program flash, kseg0_program_mem, all cacheable       */
  /* 512 KB flash = 0x80000, or 0x1000 + 0x970 + 0x490 + 0x7E200                */
  kseg0_program_mem    (rx) : ORIGIN = (0x9D000000 + 0x1000 + 0x970 + 0x490),
                              LENGTH = 0x7E200
  debug_exec_mem            : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
  /* Device Configuration Registers (configuration bits) */
  config3                   : ORIGIN = 0xBFC02FF0, LENGTH = 0x4
  config2                   : ORIGIN = 0xBFC02FF4, LENGTH = 0x4
  config1                   : ORIGIN = 0xBFC02FF8, LENGTH = 0x4
  config0                   : ORIGIN = 0xBFC02FFC, LENGTH = 0x4
  configsfrs                : ORIGIN = 0xBFC02FF0, LENGTH = 0x10
  /* all SFRS */
  sfrs                      : ORIGIN = 0xBF800000, LENGTH = 0x100000
  /* PIC32MX795F512H has 128 KB RAM, or 0x20000 */
  kseg1_data_mem       (w!x) : ORIGIN = 0xA0000000, LENGTH = 0x20000
}
```

Converting virtual to physical addresses, we see that the cacheable interrupt vector table (we will learn more about this in Chapter 6) in `exception_mem` is placed in a memory region of length 0x1000 bytes beginning at PA 0x1D000000 and running to 0x1D000FFF; cacheable startup code in `kseg0_boot_mem` is placed at PAs 0x1D001000 to 0x1D00196F; noncacheable startup code in `kseg1_boot_mem` is placed at PAs 0x1D001970 to 0x1D001DFF; and cacheable program code in `kseg0_program_mem` is allocated the rest of program flash, PAs 0x1D001E00 to 0x1D07FFFF. This program code includes the code we write plus other code that is linked.

The linker script for the NU32 bootloader placed the bootloader completely in the 12 KB boot flash with little room to spare. Therefore, the linker script for our bootloaded programs should place the programs solely in program flash. Therefore, the `boot_mem` sections above are defined to be in program flash. The label `boot_mem` tells the linker where the startup code should be placed, just as the label `kseg0_program_mem` tells the linker where the program code should be placed. (For the bootloader program, `kseg0_program_mem` was in boot flash.)

If the LENGTH of any given memory region is not large enough to hold all the program instructions or data for that region, the linker will fail.

Upon reset, the PIC32 always jumps to 0xBFC00000, where the first instruction of the startup code for the bootloader resides. The bootloader's final action is to jump to VA 0xBD001970. Since the first instruction in the startup code for our bootloaded program is installed at the first address in `kseg1_boot_mem`, `NU32bootloaded.ld` *must* define the ORIGIN of `kseg1_boot_mem` at this address. This address is also known as `_RESET_ADDR` in `NU32bootloaded.ld`.

3.6 Bootloaded Programs vs. Standalone Programs

Your executable is installed on the PIC32 by another executable: the bootloader. The bootloader has been pre-installed in the boot flash portion of flash memory using an external programming tool such as the PICkit 3. The bootloader, which always runs first when the PIC32 is reset, has already defined some of the behavior of the PIC32, so you did not need to specify it in `simplePIC.c`. Particularly, the bootloader performs tasks such as such as enabling the prefetch cache module, enabling multi-vector interrupts (see Chapter 6), and freeing some pins to be used as general I/O. The bootloader code also defines the PIC32's *configuration bits*. These bits, which control the PIC32's low-level behavior, are located in the last four words of boot flash and are written by the programming tool. When the bootloader was installed, it also set the configuration bits using XC32-specific commands that begin with `#pragma config`. The configuration bits that were set when the bootloader was installed are

```
#pragma config DEBUG    = OFF       // Background Debugger disabled
#pragma config FPLLMUL = MUL_20     // PLL Multiplier: Multiply by 20
#pragma config FPLLIDIV = DIV_2     // PLL Input Divider:  Divide by 2
#pragma config FPLLODIV = DIV_1     // PLL Output Divider: Divide by 1
```

```
#pragma config FWDTEN = OFF          // WD timer: OFF
#pragma config WDTPS = PS4096        // WD period: 4.096 sec
#pragma config POSCMOD = HS          // Primary Oscillator Mode: High Speed xtal
#pragma config FNOSC = PRIPLL        // Oscillator Selection: Primary oscillator
                                        w/ PLL
#pragma config FPBDIV = DIV_1        // Peripheral Bus Clock: Divide by 1
#pragma config UPLLEN = ON           // USB clock uses PLL
#pragma config UPLLIDIV = DIV_2      // Divide 8 MHz input by 2, mult by 12 for
                                        48 MHz
#pragma config FUSBIDIO = ON         // USBID controlled by USB peripheral when it
                                        is on
#pragma config FVBUSONIO = ON        // VBUSON controlled by USB peripheral when it
                                        is on
#pragma config FSOSCEN = OFF         // Disable second osc to get pins back
#pragma config BWP = ON              // Boot flash write protect: ON
#pragma config ICESEL = ICS_PGx2     // ICE pins configured on PGx2
#pragma config FCANIO = OFF          // Use alternate CAN pins
#pragma config FMIIEN = OFF          // Use RMII (not MII) for ethernet
#pragma config FSRSSEL = PRIORITY_6 // Shadow Register Set for interrupt priority 6
```

The directives above

- disable some debugging features;
- turn the PIC32's watchdog timer off and set its period (see Chapter 17);
- configure the PIC32's clock generation circuit to take the external 8 MHz resonator signal, divide its frequency by 2, input the divided frequency into a phase-locked loop (PLL) that multiplies the frequency by 20, and divide the PLL's output frequency by 1, creating a SYSCLK of $8/2 \times 20/1$ MHz $= 80$ MHz;
- set the PBCLK frequency to be SYSCLK divided by 1 (80 MHz);
- use a PLL to generate the 48 MHz USBCLK by first dividing the 8 MHz signal frequency by 2 before multiplying by a fixed factor of 12;
- allow two pins to be controlled by the USB peripheral when USB is enabled;
- disable the secondary oscillator (this could provide an alternative clock source for power-saving modes or as a backup);
- prevent the boot flash from being written when a program is running;
- connect the CAN modules to the alternate pins instead of the default pins;
- configure the ethernet module to use the reduced media-independent interface (RMII); and
- set the shadow register set to be used for interrupts of priority level 6 (see Chapter 6).

Remember, these bits are set by the programming tool, so they are only stored when the bootloader is written to the PIC32; using these commands in a bootloaded program has no effect. The file `pic32-libs/proc/32MX795F512H/configuration.data` contains definitions for the values you can use in the `#pragma config` directives. The Data Sheet and Configuration section of the Reference Manual have more details about the configuration bits.

If you decide not to use a bootloader, and instead use a programming tool like the PICkit 3 (Figure 2.4) to install a standalone program, you must set the configuration bits, enable the

prefetch cache module, perform other configuration tasks, and use the default linker script. If you use the NU32 library you need not write this code: `NU32.c` contains the necessary configuration bit settings (which have no effect for a bootloaded program) and `NU32_Startup` performs the necessary setup tasks (which are redundant but harmless for a bootloaded program).[4] To use the default linker script (a copy of which is located at `pic32-libs/proc/32MX795F512H/p32MX795F512H.ld`) when you build a program using `make` (Chapter 1.5), change the line `LINKSCRIPT="NU32bootloaded.ld"` to `LINKSCRIPT=` in the `Makefile`.

After building a standalone hex file, you must load it onto the PIC32 using a programming tool. The easiest method for loading a hex file is to use the MPLAB X IDE. In the IDE, create a new "precompiled" project, selecting your processor model, programming tool, and hex file. Next, hit "run," and the hex file will be written to the PIC32. Remember, the PIC32 must be powered for it to be programmed.

3.7 Build Summary

Recall that what we colloquially refer to as "compiling" actually consists of multiple steps. You initiated these steps by invoking the compiler, `xc32-gcc`, at the command line:

```
> xc32-gcc -mprocessor=32MX795F512H
   -o simplePIC.elf -Wl,--script=skeleton/NU32bootloaded.ld simplePIC.c
```

This step creates the `.elf` file, which then needs to be converted into a `.hex` file that the bootloader understands:

```
> xc32-bin2hex simplePIC.elf
```

The compiler requires multiple *command line options* to work. It accepts arguments, as detailed in the XC32 Users Manual, and some important ones are displayed by typing `xc32-gcc --help`. The arguments we used were

- `-mprocessor=32MX795F512H`: Tells the compiler what PIC32MX model to target. This also causes the compiler to define `__32MX795F512H__` so that the processor model can be detected in header files such as `xc.h`.
- `-o simplePIC.elf`: Specifies that the final output will be named `simplePIC.elf`.

[4] Technically, performing the configuration tasks a second time in NU32 startup wastes an iota of program memory and computation time, but it allows you to use the same code in both bootloaded and standalone modes without modification.

- `-Wl:` Tells the compiler that what follows are a comma-separated list of options for the linker.
- `--script=skeleton/NU32bootloaded.ld:` A linker option that specifies the linker script to use.
- `simplePIC.c:` The C files that you want compiled and linked are listed. In this case the whole program is in just one file.

Another option that may be useful when exploring what the compiler does is `-save-temps`. This option will save all of the intermediate files generated during the build process, allowing you to examine them.

Here is what happens when you build and load `simplePIC.c`.

- **Preprocessing.** The preprocessor (`xc32-cpp`), among other duties, handles include files. By including `xc.h` at the beginning of your program, we get access to variables for all the SFRs. The output of the preprocessor is a `.i` file, which by default is not saved.
- **Compiling.** After the preprocessor, the compiler (`xc32-gcc`) turns your C code into assembly language specific to the PIC32. For convenience, (`xc32-gcc`) automatically invokes the other commands required in the build process. The result of the compilation step is an assembly language `.S` file, containing a human-readable version of instructions specific to a MIPS32 processor. This output is also not saved by default.
- **Assembling.** The assembler (`xc32-as`) converts the human-readable assembly code into object files (`.o`) that contain machine code. These files cannot be executed directly, however, because addresses have not been resolved. This step yields `simplePIC.o`
- **Linking.** The object code `simplePIC.o` is linked with the `crt0_mips32r2.o` C run-time startup library, which performs functions such as initializing global variables, and the `processor.o` object code, which contains the SFR VAs. The linker script `NU32bootloaded.ld` provides information to the linker on the allowable absolute virtual addresses for the program instructions and data, as required by the bootloader and the specific PIC32 model. Linking yields a self-contained executable in `.elf` format.
- **Hex file.** The `xc32-bin2hex` utility converts `.elf` files into `.hex` files. The `.hex` is a different format for the executable from the `.elf` file that the bootloader understands and can load into the PIC32's program memory.
- **Installing the program.** The last step is to use the NU32 bootloader and the host computer's bootloader utility to install the executable. By resetting the PIC32 while holding the USER button, the bootloader enters a mode where it tries to communicate with the bootload communication utility on the host computer. When it receives the executable from the host, it writes the program instructions to the virtual memory addresses specified by the linker. Now every time the PIC32 is reset without holding the USER button, the bootloader exits and jumps to the newly installed program.

3.8 Useful Command Line Utilities

The `bin` directory of the XC32 installation contains several useful command line utilities. These utilities can be used directly at the command line and many are invoked by the `Makefile`. We have already seen the first two of these utilities, as described in Section 3.7:

xc32-gcc The XC32 version of the `gcc` compiler is used to compile, assemble, and link, creating the executable `.elf` file.
xc32-bin2hex Converts a `.elf` file into a `.hex` file suitable for placing directly into PIC32 flash memory.
xc32-ar The archiver can be used to create an archive, list the contents of an archive, or extract object files from an archive. An archive is a collection of `.o` files that can be linked into a program. Example uses include:

```
xc32-ar -t lib.a         // list the object files in lib.a
                            (in current directory)
xc32-ar -x lib.a code.o  // extract code.o from lib.a to the current directory
```

xc32-as The assembler.
xc32-ld This is the actual linker called by `xc32-gcc`.
xc32-nm Prints the symbols (e.g., global variables) in an object file. Examples:

```
xc32-nm processor.o      // list the symbols in alphabetical order
xc32-nm -n processor.o   // list the symbols in numerical order of their VAs
```

xc32-objdump Displays the assembly code corresponding to an object or `.elf` file. This process is called *disassembly*. Example:

```
xc32-objdump -S file.elf > file.dis   // send output to the file file.dis
```

xc32-readelf Displays a lot of information about the `.elf` file. Example:

```
xc32-readelf -a filename.elf  // output is dominated by SFR definitions
```

These utilities correspond to standard "GNU binary utilities" of the same name without the preceding `xc32-`. To learn the options available for a command called `xc32-cmdname`, you can type `xc32-cmdname --help` or read about them in the XC32 compiler reference manual.

3.9 Chapter Summary

OK, that's a lot to digest. Do not worry, you can view much of this chapter as reference material; you do not have to memorize it to program the PIC32!

- Software refers almost exclusively to the virtual memory map. Virtual addresses map directly to physical addresses by PA = VA & 0x1FFFFFFF.
- Building an executable `.hex` file from a source file consists of the following steps: preprocessing, compiling, assembling, linking, and converting the `.elf` file to a `.hex` file.

- Including the file `xc.h` gives our program access to variables, data types, and constants that significantly simplify programming by allowing us to access SFRs easily from C code without needing to specify addresses directly.
- The included file `pic32mx/include/proc/p32mx795f512h.h` contains variable declarations, like TRISF, that allow us to read from and write to the SFRs. We have several options for manipulating these SFRs. For TRISF, for example, we can directly assign the bits with `TRISF=0x3`, or we can use bitwise operations like `&` and `|`. Many SFRs have associated CLR, SET, and INV registers which can be used to efficiently clear, set, or invert certain bits. Finally, particular bits or groups of bits can be accessed using bit fields. For example, we access bit 3 of TRISF using TRISFbits.TRISF3. The names of the SFRs and bit fields follow the names in the Data Sheet (particularly the Memory Organization section) and Reference Manual.
- All programs are linked with `pic32mx/lib/proc/32MX795F512H/crt0_mips32r2.o` to produce the final `.hex` file. This C run-time startup code executes first, doing things like initializing global variables in RAM, before jumping to the `main` function. Other linked object code includes `processor.o`, with the VAs of the SFRs.
- Upon reset, the PIC32 jumps to the boot flash address 0xBFC00000. For a PIC32 with a bootloader, the `crt0_mips32r2` of the bootloader is installed at this address. When the bootloader completes, it jumps to an address where the bootloader has previously installed a bootloaded executable.
- When the bootloader was installed with a device programmer, the programmer set the Device Configuration Registers. In addition to loading or running executables, the bootloader enables the prefetch cache module and minimizes the number of CPU wait cycles for instructions to load from flash.
- A bootloaded program is linked with a custom linker script, like `NU32bootloaded.ld`, to make sure the flash addresses for the instructions do not conflict with the bootloader's, and to make sure that the program is placed at the address where the bootloader jumps.

3.10 Exercises

1. Convert the following virtual addresses to physical addresses, and indicate whether the address is cacheable or not, and whether it resides in RAM, flash, SFRs, or boot flash. (a) 0x80000020. (b) 0xA0000020. (c) 0xBF800001. (d) 0x9FC00111. (e) 0x9D001000.
2. Look at the linker script used with programs for the NU32. Where does the bootloader install your program in virtual memory? (Hint: look at the `_RESET_ADDR`.)
3. Refer to the Memory Organization section of the Data Sheet and Figure 2.1.
   a. Referring to the Data Sheet, indicate which bits, 0-31, can be used as input/outputs for each of Ports B through G. For the PIC32MX795F512H in Figure 2.1, indicate which pin corresponds to bit 0 of port E (this is referred to as RE0).

   b. The SFR INTCON refers to "interrupt control." Which bits, 0-31, of this SFR are unimplemented? Of the bits that are implemented, give the numbers of the bits and their names.
4. Modify `simplePIC.c` so that both lights are on or off at the same time, instead of opposite each other. Turn in only the code that changed.
5. Modify `simplePIC.c` so that the function `delay` takes an `int cycles` as an argument. The `for` loop in `delay` executes `cycles` times, not a fixed value of 1,000,000. Then modify `main` so that the first time it calls `delay`, it passes a value equal to `MAXCYCLES`. The next time it calls `delay` with a value decreased by `DELTACYCLES`, and so on, until the value is less than zero, at which time it resets the value to `MAXCYCLES`. Use `#define` to define the constants `MAXCYCLES` as 1,000,000 and `DELTACYCLES` as 100,000. Turn in your code.
6. Give the VAs and reset values of the following SFRs. (a) I2C3CON. (b) TRISC.
7. The `processor.o` file linked with your `simplePIC` project is much larger than your final `.hex` file. Explain how that is possible.
8. The building of a typical PIC32 program makes use of a number of files in the XC32 compiler distribution. Let us look at a few of them.
   a. Look at the assembly startup code `pic32-libs/libpic32/startup/crt0.S`. Although we are not studying assembly code, the comments help you understand what the startup code does. Based on the comments, you can see that this code clears the RAM addresses where uninitialized global variables are stored, for example. Find and list the line(s) of code that call the user's `main` function when the C runtime startup completes.
   b. Using the command `xc32-nm -n processor.o`, give the names and addresses of the five SFRs with the highest addresses.
   c. Open the file `p32mx795f512h.h` and go to the declaration of the SFR SPI2STAT and its associated bit field data type _ _SPI2STATbits_t. How many bit fields are defined? What are their names and sizes? Do these coincide with the Data Sheet?
9. Give three C commands, using TRISDSET, TRISDCLR, and TRISDINV, that set bits 2 and 3 of TRISD to 1, clear bits 1 and 5, and flip bits 0 and 4.

Further Reading

MPLAB XC32 C/C++ compiler user's guide. (2012). Microchip Technology Inc.
MPLAB XC32 linker and utilities user guide. (2013). Microchip Technology Inc.
PIC32 family reference manual. Section 32: Configuration. (2013). Microchip Technology Inc.