Chapter 3

Looking Under The Hood: 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 begin to explore the compilation process and the XC32 compiler installation.

3.1 The Virtual Memory Map

In the last chapter we learned about the PIC32 physical memory map. The physical memory map is relatively easy to understand: the CPU can access any SFR, or any location in data RAM, program flash, or boot flash, by a 32-bit address that it puts on the bus matrix. Since we don't really have 2^{32} bytes, or 4 GB, to access, many choices of the 32 bits don't address anything.

In this chapter we focus on the virtual memory map. This is because almost all software refers only to virtual memory addresses. Virtual addresses (VAs) specified in software are translated to physical addresses (PAs) by the fixed mapping translation (FMT) unit in the CPU, which is simply

$$\text{PA} = \text{VA} \ \& \ \text{0x1FFFFFFF}$$

This bitwise AND operation simply clears the first three bits, the three most significant bits of the most significant hex digit.

If we're just throwing away those three bits, what's the point of them? Well, those first three bits are used by the CPU and the prefetch module we learned about in the previous chapter. If the first three bits of the virtual address of a program instruction are 100 (so the corresponding most significant hex digit of the VA is an 8 or 9), then that instruction can be cached. If the first three bits are 101 (corresponding to an A or B in the leftmost hex digit of the VA), then it cannot. 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.[1]

You don't need to worry about the mysteries of cacheable vs. noncacheable instructions. Suffice to say that your program instructions will be made cacheable, speeding up execution.

The relationship of the physical memory map to the virtual memory map is illustrated in Figure 3.1. One important thing to note from the figure is that the SFRs are not included in the KSEG0 cacheable virtual memory segment. This is because the SFRs correspond to physical devices, e.g., peripherals, and their values cannot be cached. For example, if port B is configured as a digital input port, then the SFR PORTB contains the current input values. When your program asks for these values, it needs the current values; it cannot pull them from the cache.

For the rest of this chapter we will deal only with virtual addresses like 0x9D000000 and 0xBD000000, and you should realize that these refer to the same physical address. Since virtual addresses start at 0x80000000,

[1] Another cacheable segment called USEG, for "user segment," is available in the lower half of virtual memory. This memory segment is set aside for "user programs" that are running under an operating system installed in a kernel segment. For safety reasons, programs in the user segment cannot access the SFRs or boot flash. We will never use the user segment.

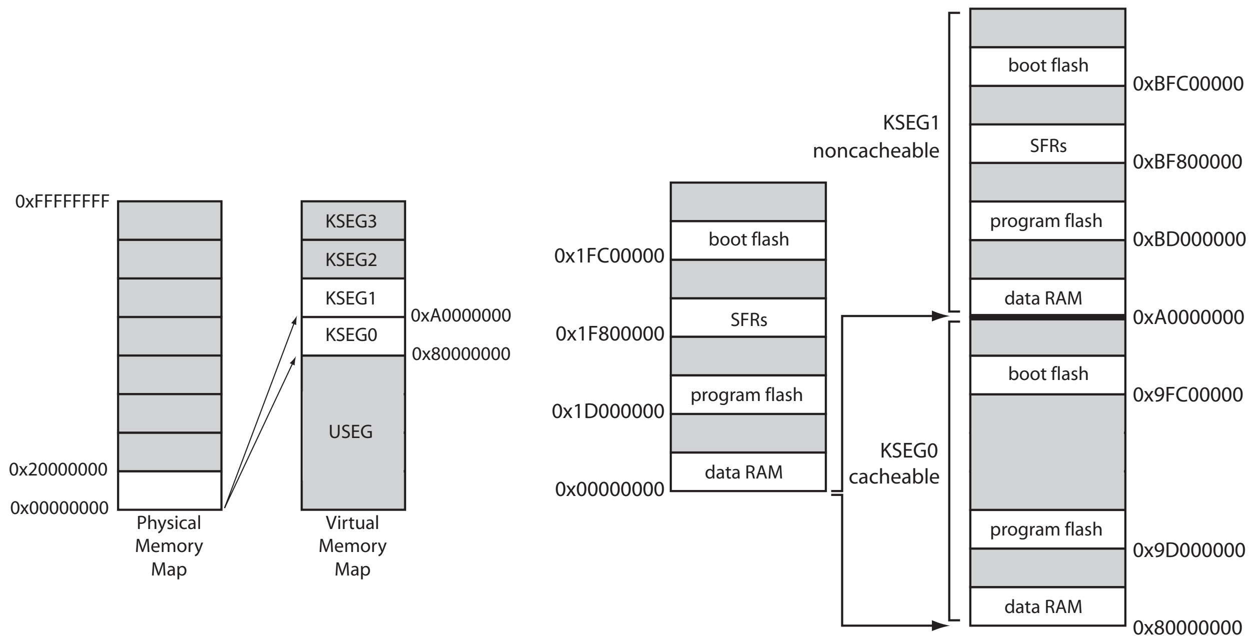

Figure 3.1: (Left) The 4 GB physical and virtual memory maps are divided into 512 MB segments. The mapping of the valid physical memory addresses to the virtual memory regions KSEG0 and KSEG1 is illustrated. The PIC32 does not use the virtual memory segments KSEG2 and KSEG3, which are allowed by the MIPS architecture, and we will not use the user segment USEG, which sits in the bottom half of the virtual memory map. (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.

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: The Bootloaded `simplePIC.c` Program

Let's build the `simplePIC.c` bootloaded executable from Chapter 1.4.2. For convenience, here is the program again:

Code Sample 3.1. `simplePIC.c`. Blinking lights, unless the USER button is pressed.

```
#include <plib.h>

void delay(void);

int main(void) {
  DDPCONbits.JTAGEN = 0; // Disable JTAG, make pins 4 and 5 of Port A available.
  TRISA = 0xFFCF;        // Pins 4 and 5 of Port A are LED1 and LED2.  Clear
                         // bits 4/5 to zero, for output.  Others are inputs.
  LATAbits.LATA4 = 0;    // Turn LED1 on and LED2 off.  These pins sink ...
  LATAbits.LATA5 = 1;    // ... current on NU32, so "high" = "off."

  while(1) {
    delay();
```

```
    LATAINV = 0x0030;    // toggle the two lights
  }
  return 0;
}

void delay(void) {
  int j;
  for (j=0; j<1000000; j++) { // number is 1 million
    while(!PORTDbits.RD13);   // Pin D13 is the USER switch, low if pressed.
  }
}
```

Following the same procedure as in Chapter 1.4.1 (command line) or in Chapter 1.4.2 (MPLAB X IDE), build the executable and load it onto your NU32. Remember to use the linker script `NU32bootloaded.ld`. When you have the program loaded and running, the NU32's two LEDs should alternate on and off, and stop while you press the USER button.

This program refers to SFRs named TRISA, LATAINV, etc. These names align with the SFR names in the Data Sheet and Reference Manual sections on I/O Ports. We will consult the Data Sheet and Reference Manual often when programming the PIC32. We will come back to understanding the use of these SFRs shortly.

3.3 What Happens When You Build?

Now let's begin to understand how you created the `.hex` file in the first place. Figure 3.2 gives a schematic of what happens when you click "Build" in your MPLAB X IDE or type `make` with a makefile.

First the **preprocessor** strips out comments and inserts `#include`d header files. 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 this in Chapter 4.

Then the **compiler** turns the C files into MIPS32 assembly language files, machine commands that are directly understood by the PIC32's MIPS32 CPU. So while some of your C code may be easily portable to non-MIPS32 microprocessors, your assembly code generally will not be. These assembly files are readable by a text editor, and it is possible to program the PIC32 directly in assembly language.

The **assembler** then 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 for us!

Finally, the **linker** takes multiple object files and links them together 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 as to where to place the data and program instructions in virtual memory. The end result is an executable and linkable format (`.elf`) file, a standard format. This file contains a plethora of information that is useful for debugging and *disassembling* the file into the assembly code produced by the compiler (Chapter 5.1.3). In fact, 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 that is suitable for placing directly into the memory of your PIC32.

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

3.4 What Happens When You Reset the PIC32?

You've got your program running. Now you hit the RESET button on the NU32. What happens next?

The first thing your PIC32 does is to jump to the first address in boot flash, 0xBFC00000, and begin

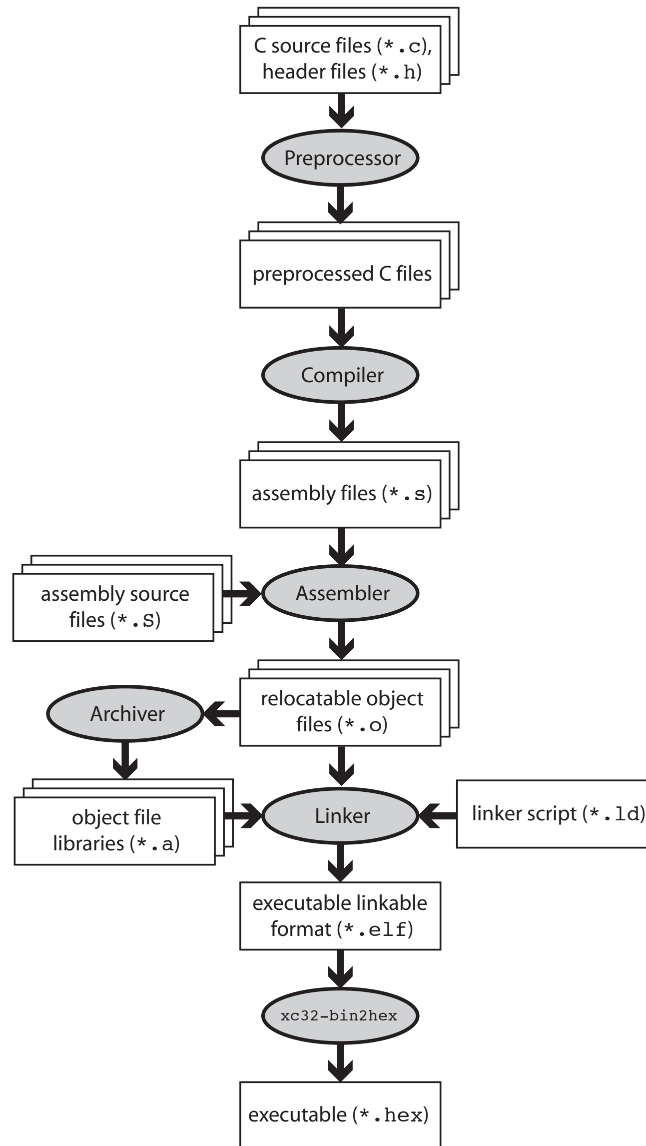

Figure 3.2: The “compilation” process.

executing instructions there.[2] For an NU32 with a bootloader installed, the preloaded bootloader program sits at that location in memory. The bootloader first checks to see if you are pressing the USER button. If so, it knows that you want to reprogram your PIC32, so the bootloader attempts to establish communication with the bootloader communication utility on your computer. When communication is established, the bootloader receives your executable `.hex` file and writes it to your PIC32's program flash. (See question 3.) Let's call the virtual address where your program is installed `_RESET_ADDR`.

Note: The PIC32's reset address 0xBFC00000 is fixed in hardware and cannot be changed. On the other hand, there is nothing too special about the choice of the program flash address where the bootloader writes your program.

Let's say you weren't pressing the USER button when you reset the PIC32. Then the bootloader jumps

[2]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.

Virtual Address (BF88_#)	Register Name	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	
6000	TRISA	31:16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
		15:0	TRISA15	TRISA14	—	—	—	TRISA10	TRISA9	—	TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	C6FF

Figure 3.3: Port A registers, taken from the PIC32 Data Sheet.

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. That is the desired behavior in embedded control. If your program exits, the PIC32 will just sit in a tight loop, doing nothing until it is reset.

3.5 Understanding `simplePIC.c`

OK, let's get back to understanding `simplePIC.c`. The `main` function is very simple. It initializes values of DDPCONbits, TRISA, and LATAbits, then enters an infinite `while` loop. Each time through the loop it calls `delay()` and then assigns a value to LATAINV. The `delay` function simply goes through a `for` loop a million times. Each time through the `for` loop it enters a `while` loop, which checks the value of (!PORTDbits.RD13). If PORTDbits.RD13 is 0 (FALSE), then the expression (!PORTDbits.RD13) evaluates to TRUE, and the program stays stuck here, doing nothing but checking the expression (!PORTDbits.RD13). When this evaluates to FALSE, the `while` loop is exited, and the program continues with the `for` loop. After a million times through the `for` loop, control returns to `main`.

Special Function Registers (SFRs) The only reason this program is even a little interesting is that TRISA, LATA, and PORTD all refer to peripherals that interact with the outside world. Specifically, TRISA and LATA correspond to port A, an input/output port, and PORTD corresponds to port D, another input/output port.[3] We can start our exploration by consulting the table in Section 1 of the Data Sheet which lists the pinout I/O descriptions. We see that port A, with pins named RA0 to RA15, consists of 12 different pins, and port C, with pins named RC1 to RC15, has 8 pins. These are in contrast to port B, which has a full 16 pins, labeled RB0 to RB15.

Now turn to the Data Sheet section on I/O Ports to get some more information. We find that TRISA, short for "tri-state A," is used to control the direction, input or output, of the pins on port A. For each pin, there is a corresponding bit in TRISA. If the bit is a 0, the pin is an output. If the bit is a 1, the pin is an input. ($0 = \text{O}_{\text{utput}}$ and $1 = \text{I}_{\text{nput}}$. Get it?) We can make some pins inputs and some outputs, or we can make them all have the same direction.

If you're curious what 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! But after a bit of searching, you find that TRISA sits at virtual address 0xBF886000, and its default value upon reset is 0x0000C6FF. (We've reproduced part of this table for you in Figure 3.3.) In binary, this would be

$$0\text{x}0000\text{C}6\text{FF} = 0000\ 0000\ 0000\ 0000\ 1100\ 0110\ 1111\ 1111.$$

The leftmost four hex digits (two bytes, or 16 bits) are all 0. This is because those bits don't exist, technically. Microchip calls them "unimplemented." No I/O port has more than 16 pins, so we don't need those bits, which are numbered 16–31. (The 32 bits are numbered 0–31.) Of the remaining bits, since the 0'th bit (least significant bit) is the rightmost bit, we see that bits 0–7, 9–10, and 14–15 have a value 1, while the rest have value 0. The bits with value 1 correspond precisely to the pins we have available: RA0-7, RA9-10, and

[3] DDPCON corresponds to JTAG debugging, which we do not use in this book. The `DDPCONbits.JTAGEN = 0` command simply disables the JTAG debugger so that pins RA4 and RA5 are available for digital I/O. See the Special Features section of the Data Sheet.

RA14-15. (There is no RA8, for example.) This is for safety reasons; when we power on the PIC32, each pin will take its default direction before the program has a chance to change it. If an output pin were connected to an external circuit that is also trying to control the voltage on the pin, the two devices would be fighting 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

```
TRISA = 0xFFCF;
```

clears bits 4 and 5 to 0, implicitly clears bits 16–31 to 0 (which is ignored, since the bits are not implemented), and sets the rest of the bits to 1. It doesn't matter that we try to set some unimplemented bits to 1; those bits are simply ignored. The result is that port A pins 4 and 5, or RA4 and RA5 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 don't get lost counting bits, you could have equally written

```
TRISA = 0b1111111111001111;
```

Another option would have been to use the instructions

```
TRISAbits.TRISA4 = 0; TRISAbits.TRISA5 = 0;
```

This allows us to change individual bits without worrying about specifying the other bits. We see this kind of notation later in the program, with LATAbits.LATA4 and PORTDbits.RD13, for example.

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

```
LATAbits.LATA5 = 1;
```

sets pin RA5 high. Finally, PORTD contains the digital inputs on the port D pins. (Notice we didn't configure port D as input; we relied on the fact that it's the default.) PORTDbits.RD13 is 0 if 0 V is present on pin RD13 and 1 if approximately 3.3 V is present.

Pins RA4, RA5, and RD13 on the NU32 Figure 2.4 shows how pins RA4, RA5, and RD13 are wired on the NU32 board. LED1 (LED2) is on if RA4 (RA5) is 0 and off if it is 1. When the USER button is pressed, RD13 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

```
LATAINV = 0x0030;
```

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

```
LATAINV = 0x30;      // flips (inverts) bits 4 and 5 of LATA; all others unchanged
LATAINV = 0b110000;  // same as above
LATASET = 0x0005;    // sets bits 0 and 2 of LATA to 1; all others unchanged
LATACLR = 0x0002;    // clears bit 1 of LATA to 0;  all others unchanged
```

A less efficient way to toggle bits 4 and 5 of LATA is

```
LATAbits.LATA4 = !LATAbits.LATA4; LATAbits.LATA5 = !LATAbits.LATA5;
```

We'll look at efficiency in Chapter 5.

You can go back to the table in the Data Sheet to see the VA addresses of the CLR, SET, and INV registers. They are always offset from their base register by 4, 8, and 12 bytes, respectively; they are consecutive in the memory map. Since LATA is at 0xBF886020, LATACLR, LATASET, and LATAINV are at 0xBF886024, 0xBF886028, and 0xBF88602C, respectively.

OK, we should now have a pretty good understanding of how simplePIC.c works. But we have been ignoring the fact that we never declared TRISA, etc., before we started using them. We know you can't do that in C; they must be declared somewhere. The only place they could be declared is in the included file plib.h. We've been ignoring that #include <plib.h> statement until now. Time to take a look.[4]

3.5.1 Down the Rabbit Hole

But where do we find plib.h? If our program had the preprocessor command #include "plib.h", the preprocessor would look for plib.h in the same directory as the C file including it. But we had #include <plib.h>, and the <...> notation means that the preprocessor will look in directories specified in your *include path*. This include path was generated for you automatically when you installed the XC32 compiler. For us, the default include path means that the compiler finds plib.h sitting in the directory path

```
microchip/xc32/v1.30/pic32mx/include/plib.h
```

Your path or version number might be slightly different. In this book the directory separator character is /, consistent with Linux, Unix, and Mac OS X. On Windows, the directory separator character is \.

Including plib.h gives us access to many data types, variables, constants, macros, and functions that Microchip has provided for our convenience. While plib.h stands for "peripheral library," including plib.h provides functionality beyond just the peripherals.

Before we open up plib.h, let's look at the directory structure that was created when we installed the XC32 compiler. There's a lot here! We certainly don't need to understand all of it at this point, but let's try to get a sense of what's going on. Let's start at the level microchip/xc32/v1.30 and summarize what's in the nested set of directories, without being exhaustive.

1. bin: This 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 is notable because it contains the .c C files, .h header files, and .S assembly files needed to create the object code for a number of pre-built PIC32 functions, particularly for the peripherals. Later we might want to look at these more closely, as they are some of the best documentation on the peripheral library functions. A few notable subdirectories:
   (a) peripheral: The subdirectories under this directory contain the C code for peripheral library functions.
   (b) include/peripheral: This directory contains header files for peripheral library functions. Though there is a plib.h here, it is not the one the compiler finds when building simplePIC.c.
   (c) dsp: This directory contains some C functions that call vector math, filter, and Fourier transform code in the MIPS Digital Signal Processing library.
   (d) libpic32: This directory contains a number of C and assembly files for basic PIC32 functions, such as code that should be executed at the beginning of any executable (see next).

[4] Microchip often makes changes to the software it distributes, so there may be differences in details, but the broad strokes described here will be the same.

(e) `libpic32/startup/crt0.S`: This "C run-time startup" assembly code gets inserted at the beginning of every program we create. This code takes care of a number of initialization tasks. For example, if your program uses uninitialized global variables, `crt0` writes zeros into their data RAM locations. If your global variables are initialized at the time of definition, e.g., `int k=3`, then `crt0` copies the initialized values from program flash to data RAM.

6. `pic32mx`: This directory has a number of files we are interested in.

   (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:

      i. `crt0.o`: This is the compiled object code of `crt0.S`. The linker combines this code with our program's object code and makes sure that it is executed first.

      ii. `libmchp_peripheral.a`: This library contains the `.o` object code versions of the `.c` core timer functions in the top-level `pic32-libs` library.

      iii. `libmchp_peripheral_32MX795F512L.a`: This library contains the `.o` object code versions of the `.c` peripheral library functions in the top-level `pic32-libs` library. There are versions of this file for every type of PIC32MX.

      iv. `libdsp.a`: This library contains MIPS implementations of finite and infinite impulse response filters, the fast Fourier transform, and various vector math functions.

      v. `ldscripts/elf32pic32mx.x`: For a standalone program, this is the default linker script that gives the linker rules as to where it is allowed to finally place the relocatable object codes in virtual memory. This script includes `procdefs.ld`, below.
      If your program is built to be bootloaded, the linker uses your custom linker script instead, such as `NU32bootloaded.ld`.

      vi. `proc/32MX795F512L/procdefs.ld`: This file is included by the default linker script, above. It declares segments of data RAM and program flash where the linker can place data and instructions, and it is specific to your particular PIC32 model. It also includes `processor.o`, below.

      vii. `proc/32MX795F512L/processor.o`: This object file gives the SFR virtual memory addresses for your particular PIC32. We can't 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
...
bf886000 A TRISA
bf886004 A TRISACLR
bf88600c A TRISAINV
bf886008 A TRISASET
...
```

      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. This tells the linker that it must use these addresses when making final address assignments. This makes sense; the SFR's are implemented in hardware and can't be moved! The listing above indicates that TRISA is located at VA 0xBF886000, agreeing with the Memory Organization section of the Data Sheet.

      viii. `proc/32MX795F512L/configuration.data`: This file describes some constants used in setting the configuration bits in DEVCFG0 to DEVCFG3 (Chapter 2.1.4). The main purpose of these constants is to make your code more readable. For example, a standalone program like `simplePIC_standalone.c` from Chapter 1.5 has the following code:

```
#pragma config DEBUG = OFF          // Background Debugger disabled
#pragma config FPLLMUL = MUL_20     // PLL Multiplier: Multiply by 20
```

These #pragmas are nonstandard C code, and for our particular compiler, they are used to write to the DEVCFGx bits the values defined by constants like MUL_20, defined in configuration.data. Many of these #pragmas are used to set up the timing generation circuit that turn our 8 MHz resonator into an 80 MHz SYSCLK, an 80 MHZ PBCLK, and a 48 MHz USBCLK. You can learn more about the DEVCFGx configuration bits in the Special Features section of the Data Sheet.

For bootloaded code, the configuration bits are set by the bootloader program, so these #pragmas are not necessary.

(b) include: This directory contains a number of .h header files, including the one we've been looking for, plib.h.

i. plib.h: This is the file that was found in our compiler's include path. If we open it up, we find that it includes a bunch of other files! One of them is peripheral/ports.h, so let's open that one up.

ii. peripherals/ports.h: This file provides constants, macros, and function prototypes for library functions that work with the I/O ports. More importantly, for now, is that it includes xc.h. This file is found one directory up in the directory tree. Let's open that next.

iii. xc.h: This file does a few different things, such as defining constants and macros, and including files defining MIPS constants and macros, but the most important for the moment is that it includes proc/p32mx795f512l.h because of the lines

```
#elif defined(__32MX795F512L__)
#include <proc/p32mx795f512l.h>
```

Why does this code include proc/p32mx795f512l.h? When you were setting up your simplePIC project in the first place, you had to specify the particular PIC32 you were using. The MPLAB X IDE passed your answer to the compilation process by "defining" the constant __32MX795F512L__. This allows the preprocessor to include the right files for your PIC32. Let's open proc/p32mx795f512l.h.

iv. proc/p32mx795f512l.h: Whoa! This file is over 40,000 lines long. It also includes one other file in the same directory, ppic32mx.h, which is over 1000 lines long. With these we have reached the bottom of our include chain. Let's pop out of this big directory tree we are sitting in and look at those two files in a little more detail.

3.5.2 The Include Files p32mx795f512l.h and ppic32mx.h

The first 30% of p32mx795f512l.h, about 14,000 lines, consists of code like this:

```
extern volatile unsigned int        TRISA __attribute__((section("sfrs")));
typedef union {
  struct {
    unsigned TRISA0:1;   // TRISA0 is bit 0 (1 bit long), interpreted as an unsigned int
    unsigned TRISA1:1;   // bits are in order, so the next bit, bit 1, is called TRISA1
    unsigned TRISA2:1;   // ...
    unsigned TRISA3:1;
    unsigned TRISA4:1;
    unsigned TRISA5:1;
    unsigned TRISA6:1;
    unsigned TRISA7:1;
    unsigned :1;         // don't give a name to bit 8; it's unimplemented
    unsigned TRISA9:1;   // bit 9 is called TRISA9
    unsigned TRISA10:1;
    unsigned :3;         // skip 3 bits, 11-13
    unsigned TRISA14:1;
    unsigned TRISA15:1;  // later bits are not given names
  };
  struct {
    unsigned w:32;       // w is a field referring to all 32 bits; the 16 above, and 16 more
```

```
  };
} __TRISAbits_t;
extern volatile __TRISAbits_t TRISAbits __asm__ ("TRISA") __attribute__((section("sfrs")));
extern volatile unsigned int        TRISACLR __attribute__((section("sfrs")));
extern volatile unsigned int        TRISASET __attribute__((section("sfrs")));
extern volatile unsigned int        TRISAINV __attribute__((section("sfrs")));
```

The first line, beginning `extern`, indicates that TRISA is an `unsigned int` variable that has been defined elsewhere; no space has to be allocated for it.[5] The `processor.o` file is the one that actually defines the VA of the symbol TRISA, as mentioned earlier. (The `__attribute__` syntax tells the linker that TRISA is in the `sfrs` section of memory.)

The next section of code defines a data type called `__TRISAbits_t`. The purpose of this is to provide a struct that gives easy access to the bits of the SFR. After defining this type, a variable named TRISAbits is declared of this type. Again, since it is an `extern` variable, no memory is allocated, and, in fact, the `__asm__ ("TRISA")` syntax means that TRISAbits is at the same VA as TRISA. The definition of the *bit field* TRISAbits allows us to use TRISAbits.TRISA0 to refer to bit 0 of TRISA. In general, fields do not have to be one bit long; for example, TRISA.w is the `unsigned int` created from all 32 bits, and 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 8 bits long and a second field AMASK that is 4 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 TRISA and TRISAbits, we see declarations of TRISACLR, TRISASET, and TRISAINV. The presence of these declarations in this included header file allows `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 VAs.

With these declarations in `p32mx795f512l.h`, the `simplePIC.c` statements

```
TRISA = 0xFFCF;
LATAINV = 0x0030;
while(!PORTDbits.RD13)
```

finally make sense; these statements write values to, or read values from, SFRs at VAs specified by `processor.o`. You can see that `p32mx795f512l.h` declares a lot of SFRs, but no memory has to be allocated for them; they exist at fixed addresses in the PIC32's hardware.

The next 9% of `p32mx795f512l.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 about 17,500 lines into the file, we see 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
```

[5]The `volatile` keyword, applied to all the SFRs, means that the value of this variable could change without the CPU knowing it. Thus the CPU should reload it every time it is needed, not assume that its value is unchanged just because the CPU has not changed it.

These refer to the Timer 1 SFR T1CON. Consulting the information about T1CON in the Timer1 section of the Data Sheet, we see that bit 1, called TCS, controls whether Timer 1'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 Timer 1 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 bits 5 and 4, which are unchanged and shifted to 1 and 0
```

Another example usage is in `pic32mx/include/peripheral/timer.h`, where we find the constant definition

```
#define T1_PS_1_64    (2 << _T1CON_TCKPS_POSITION)    /* 1:64 */
```

`T1_PS_1_64` is set to the value of 2, or binary 0b10, left-shifted by `_T1CON_TCKPS_POSITION` positions, yielding 0b100000. If this is bitwise OR'ed with other constants, you can specify the properties of Timer 1 using code that is readable without consulting the Data Sheet or Reference Manual. For example, you could use the statement

```
T1CON = T1_ON | T1_PS_1_64 | T1_SOURCE_INT;
```

to turn the timer on, set the prescaler to 1:64, and set the source of the timer to be the internal PBCLK. Of course you have to read the file `timer.h` to know what the available constants are! You might find it easier to consult the Data Sheet or Reference Manual and assign the bit values based on the information there.

The definitions of the `POSITION`, `LENGTH`, and `MASK` constants take up most of the rest of the file. 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, `p32mx795f512l.h` concludes by including `ppic32mx.h`, which defines a number of other constants, again with the intent to help you write more readable code. These constants are common for all PIC32 types, unlike those defined in `p32mx795f512l.h`.

3.5.3 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 three lines near the beginning:

```
INPUT("processor.o")
OPTIONAL("libmchp_peripheral.a")
OPTIONAL("libmchp_peripheral_32MX795F512L.a")
```

The first line tells the linker to load the `processor.o` file specific to your PIC32. This allows the linker to resolve references to SFRs to actual addresses. The next two lines tell the linker to give the code access to the `.o` object codes for the PIC32 peripheral library.

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
 * 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 + 0x940 + 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
  /* PIC32MX795F512L 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. This is why the `boot_mem` sections above are defined to be in program flash. The label `boot_mem` simply 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 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 last thing the bootloader does is 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 Summarizing the Build

Section 3.5 was a long one, so let's summarize by looking at what happens when you build a project. If you look at the output when you type `make out.hex` at the command line, or at your IDE's Output window after pressing "Build," you see the actual command line commands that were executed. They look approximately like this:

```
xc32-gcc -g -x c -c -mprocessor=32MX795F512L -o simplePIC.o  simplePIC.c

xc32-gcc -mprocessor=32MX795F512L -o out.elf  simplePIC.o \
             -Wl,--defsym=__MPLAB_BUILD=1,--script="NU32bootloaded.ld",-Map=out.map

xc32-bin2hex out.elf
```

The first line preprocesses, compiles, and assembles the `simplePIC.c` program, the second links the resulting `.o` file to make a `.elf` file, and the third line converts the `.elf` file to a `.hex` file.[6]

Using the XC32 compiler guide, in the `docs` directory of our XC32 distribution, we see that the `-g` flag in the first line tells the compiler to produce debugging information, which is useful for examining the assembly code compiled from your source code; the `-x c` sequence indicates that the input file is C source code; the `-c` flag tells the compiler to compile and assemble, but not to link, and instead produce a `.o` object file; `-mprocessor=32MX795F512L` indicates the device so that the proper device-specific files are used in compilation; and the `-o` flag indicates the name of the output file. The last argument, `simplePIC.c`, is the name of the file to be compiled.

In the second line, the linker guide tells us that `-mprocessor=32MX795F512L` specifies the device type (e.g., which `processor.o` file to load); `-o` indicates the name of the output file; and `-Wl,--defsym=__MPLAB_BUILD=1,--script="NU32bootloaded.ld,-Map=out.map"` tells the linker to put the symbol `__MPLAB_BUILD` at address 1, to use the linker script `NU32bootloaded.ld`, and to create a "map" file with information on where instructions and global variables are placed in memory. We will learn more about map files in Chapter 5. The option `-Wl` is "-W ell" not "-W one."

Note that in the first line, `xc32-gcc` compiles and assembles, based on the specified options, while in the second line, `xc32-gcc` links because the input file is object code.

To summarize, here is what you need to know about the creation of your final `.hex` executable from your `simplePIC.c` source file:

- **IDE setting of the processor type.** If you build your project using the IDE, you must choose the processor type. This tells the IDE which `-mprocessor` to specify to the `xc32-gcc` command which preprocesses, compiles, and assembles, and to the `xc32-gcc` command which links to create the executable.
- **Including the Microchip `plib.h` file.** By including `plib.h` at the beginning of your program, we get access to variables for all the SFRs, as well as a number of other constants, macros, and prototypes for functions that Microchip provides. The `simplePIC.c` file, which contains references to these, will now compile and assemble successfully, and these references will be resolved at the linking stage.
- **Linking.** The object code `simplePIC.o` is linked with (a) the `crt0.o` C run-time startup library, which performs functions such as initializing global variables; (b) the `processor.o` object code with the

[6]You can see even more information about the build by specifying the `--verbose` option for the compiler and linker in the IDE. Type `--verbose` under Additional Options at both **Run** > **Set Project Configuration** > **Customize** > **xc32-gcc** and **Run** > **Set Project Configuration** > **Customize** > **xc32-ld**. At the command line, put `--verbose` at the end of the compile command and `,--verbose` immediately at the end of the linker `-Wl` options, with no space in front of the comma.

SFR VAs; and (c) code from object code libraries such as `libpic32.a`, `libmchp_peripheral.a`, and `libmchp_peripheral_32MX795F512L.a`. The linker script `NU32bootloaded.ld`, which is specific to the bootloader and the PIC32MX795F512L, provides information to the linker on the allowable absolute virtual addresses for the program instructions and data. The result of the linker is a fully self-contained executable in `.elf` format, which is then converted to `.hex` format by `xc32-bin2hex`. The address of the first instruction in the executable is the same address the bootloader jumps to.

- **Installing the program.** The last step is to use the NU32 bootloader and the host computer's bootloader communication 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.

3.7 Building `simplePIC_standalone.c`

In the case of the standalone version `simplePIC_standalone.c` in Code Sample 1.2 in Chapter 1.5, the build process is very similar to that of the bootloaded version in Section 3.6, with the following exceptions:

- **Source code differences.** The source code has the following additions compared to the bootloaded version:

  1. **Configuration bits.** A number of lines beginning `#pragma config` define the configuration bits of the Device Configuration Registers (Chapter 2.1.4). These bits define fundamental operating behavior of the PIC32 that should not be changed during execution. Examples include bits that control the conversion of the external oscillator frequency into the SYSCLK and PBCLK. These XC32-specific preprocessor commands will cause the final `.hex` file to contain values to be written to the Device Configuration Registers. The constants used in these `#pragma`s, like `MUL_20`, are defined in the `configuration.data` file. You can learn more about the configuration bits in the Special Features section of the Data Sheet.

     When using a bootloader, the configuration bits are set by the bootloader, so the `#pragma`s are not needed.

  2. **Configuring the cache and flash wait cycles.** The other additions to the source code are the preprocessor command

     ```
     #define SYS_FREQ 80000000          // 80 million Hz
     ```

     and the code statement

     ```
       SYSTEMConfig(SYS_FREQ, SYS_CFG_ALL);
     ```

     The preprocessor command simply defines the constant `SYS_FREQ` for use in `SYSTEMConfig()`. Defining `SYS_FREQ` does not actually affect SYSCLK; SYSCLK is determined by the configuration bits and the frequency of the external oscillator. We must make sure `SYS_FREQ` is consistent with these.

     The command `SYSTEMConfig()` is defined in `pic32mx/include/peripheral/system.h`. Its purpose is to maximize performance by turning on the prefetch cache module and setting the smallest possible number of flash wait cycles. The number of flash wait cycles is the number of cycles that the CPU must stall while waiting for an instruction to load from flash. Based on the SYSCLK frequency (80 MHz) and the maximum frequency of flash access (30 MHz), the number of flash wait cycles is set to 2. This wait time is set in the CHECON SFR, described in the Prefetch Cache Module chapter of the Reference Manual.

     When using a bootloader, the bootloader configures the cache and flash wait cycles.

- **The linker script.** The standalone version of the program uses the default linker script `elf32pic32mx.x`, not `NU32bootloaded.ld`. Opening `elf32pic32mx.x`, we see the command `INCLUDE procdefs.ld`. If there happens to be a `procdefs.ld` file in the same folder as `simplePIC_standalone.c`, that file will be included in the linker script. Otherwise, the linker will find the file in the directory `pic32mx/lib/proc/MX795F512L/`. This file contains default definitions of the size of RAM and flash memory. Since there is no bootloader, there is no concern of the new executable overwriting a bootloader. Your standalone executable will be installed beginning at the hardware-defined reset address, 0xBFC00000.

  Since the linker script `NU32bootloaded.ld` is no longer needed, we do not have the `--script` option to `-Wl` in the linker command:

```
xc32-gcc -mprocessor=32MX795F512L -o out.elf  simplePIC.o \
             -Wl,--defsym=__MPLAB_BUILD=1,-Map=out.map
```

3.8 Useful Command Line Utilities

The `bin` directory of the XC32 installation contains a number of useful command line utilities. These can be used directly at the command line, and some of them are invoked by the MPLAB X IDE. We have already seen the first two of these utilities, as described in Section 3.6:

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 to 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. 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
```

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

```
xc32-objdump -D file.o
xc32-objdump -d -S file.elf                 // change -d to -D to see debugging info
xc32-objdump -d -S file.elf > file.disasm   // send output to the file file.disasm
```

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-`. You can search online for more information on these utilities. To learn the options available for a command called `xc32-cmdname`, you can type `xc32-cmdname ?` at the command line.

3.9 Chapter Summary

OK, that's a lot to digest. Don't worry, you can view much of this chapter as reference material; you don't 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 `plib.h` initiates a chain of included files that gives our program access to variables, data types, constants, macros, and prototypes of functions that significantly simplify programming. C source files can be found in `pic32-libs/peripheral`, header files in `pic32mx/include`, and compiled libraries in `pic32mx/lib`.
- The included file `pic32mx/include/proc/p32mx795f512l.h` contains variable declarations, like TRISA, that allow us to read from and write to the SFRs. We have several options for manipulating these SFRs. For TRISA, for example, we can directly assign the bits with `TRISA=0x30`, 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 TRISA using TRISAbits.TRISA3. 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/crt0.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. Standalone executables have their first instruction (of the `crt0` startup code) at this address. For a PIC32 with a bootloader, the `crt0` 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.
- If the PIC32 has a bootloader, the bootloader sets the Device Configuration Registers, turns on the prefetch cache module, and minimizes the number of CPU wait cycles for instructions to load from flash. If a program is standalone, it must have code to perform these functions.
- 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. A standalone program uses the default `elf32pic32mx.x` linker script, which makes use of a default `procdefs.ld` for the particular processor.
- Command line utilities like `xc32-ar`, `xc32-nm`, `xc32-objdump`, and `xc32-readelf` allow us to learn more about our compiled code, outside of the MPLAB X IDE.

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. Explain the differences between a standalone PIC32 program and a program that is meant to be loaded with a bootloader. Which commands or functions must appear in a standalone program that are not needed in a program loaded by a bootloader? What differences are there in the linker scripts used by a standalone program and a program loaded by a bootloader?

3. Look at the linker script used with bootloaded programs for the NU32. Where does the bootloader install your program in virtual memory? (Hint: look at the `_RESET_ADDR`.)

4. 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 A through G. For the PIC32MX795F512L 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.

5. 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.

6. 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.

7. Give the VAs and reset values of the following SFRs. (a) I2C2CON. (b) TRISC.

8. The `processor.o` file linked with your `simplePIC` project is much larger than your final `.hex` file. Explain how that is possible.

9. The building of a typical PIC32 program makes use of a number of files in the XC32 compiler distribution. Let's 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, list four things that the startup code does.

   (b) Copy the library file `pic32mx/lib/libmchp_peripheral_32MX795F512L.a` to a directory where you can experiment. Using the archiver command `xc32-ar`, find the object codes that belong to the library by the command

   ```
   xc32-ar -t libmchp_peripheral_32MX795F512L.a
   ```

   You should see that one of the included object codes is `pcache.o`. Now extract this object file using

   ```
   xc32-ar -x libmchp_peripheral_32MX795F512L.a pcache.o
   ```

   Finally we can disassemble the object code to see the corresponding assembly code using

   ```
   xc32-objdump -D pcache.o
   ```

   The two functions of interest are `CheKseg0CacheOn` and `CheKseg0CacheOff`, which are used to turn on and off the cache. Find the C source file corresponding to the object code under `pic32-libs/peripheral`. For the function `CheKseg0CacheOff`, give the one line of C code in the C source file that corresponds to the two lines of assembly code beginning with `and` (a bitwise AND) and `ori` (a bitwise OR).

   (c) Using the command `xc32-nm -n processor.o`, give the names and addresses of the five SFRs with the highest addresses.

   (d) Give five files included by `pic32mx/include/plib.h`.

   (e) In `pic32mx/include/peripheral/ports.h`, explain how the macro `mPORTAReadBits()` works and give an example call to this macro that reads bits 0 and 3.

(f) Open the file `p32mx795f512l.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?

10. Give three C commands, using TRISASET, TRISACLR, and TRISAINV, that set bits 2 and 3 of TRISA to 1, clear bits 1 and 5, and flip bits 0 and 4.