Chapter 3

Looking Under The Hood: Software

In the last chapter, we learned about the PIC32 physical and virtual memory maps. Of these, the physical memory map is easier to understand: the CPU can access any SFR, or any location in data RAM, program flash, and 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 fact, we found that all RAM is located at addresses of the form 0x000*****, all flash is located at addresses 0x1D0*****, all SFRs are located at addresses 0x1F8*****, and all boot flash is located at addresses 0x1FC*****. The five least significant hex digits ***** in the address can refer to 16^5 different locations, or 1 MB, more than we need for any of the memory regions.

In this chapter we will be focusing on the virtual memory map. This is because all software refers only to the virtual memory map. Virtual memory addresses are translated to physical memory addresses by the fixed mapping translation (FMT) unit in the CPU, which, as mentioned in the last chapter, is simply

Physical Address = Virtual Address & 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. (Remember, the prefetch module's main job is to grab instructions from flash before the CPU needs them, so the CPU doesn't have to wait around because of slow flash access.) 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 and the noncacheable segment is called KSEG1.[1]

We will set aside the mysteries of which instructions should be cacheable or noncacheable for now. Suffice to say that you won't have to worry about it; all of your program instructions will be in KSEG0 program flash, and all of your data will be in KSEG1 data RAM. After all, you want to be able to use the cache to speed up your program execution (hence your program instructions will be cacheable), and there is no need to cache data in RAM, since RAM can be accessed in one cycle, unlike flash.

For the rest of this chapter we will deal only with virtual addresses like 0x9D000000 and 0xBD000000, and you should know that these refer to the same physical address. Since virtual addresses start at 0x80000000, and all physical addresses are below that, there is no possibility of confusion about whether we are talking about a VA or a PA.

[1]The virtual memory segment 0x00000000 to 0x7FFFFFFF is a cacheable segment called USEG, indicating "user segment," as opposed to KSEG, indicating "kernel segment." We will never use this virtual memory segment. Instructions in this virtual segment cannot access the SFRs or boot flash.

3.1 A Simple Program

Create a project for your NU32 board called `simple` with the following source code. Make sure `procdefs.ld` is in the same directory.

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

```
#include <plib.h>

void delay(void);

void main(void) {
  // Port A dig I/O pins 4 and 5 are connected to LEDs 1 and 1.  Make them outputs.
  TRISA = 0xFFCF;  // Pins 4 and 5 are cleared to 0, for output; other pins of Port A are inputs.

  // Turn LED1 on and LED2 off.  These pins sink current on NU32, so "high" = "off."
  LATAbits.LATA4 = 0; LATAbits.LATA5 = 1;

  while(1) {
    delay();
    LATAINV = 0x0030;  // toggle the two lights
  }
}

void delay(void) {
  int j = 0;
  for (j=0;j<1000000;j++) {
    while(!PORTCbits.RC13);   // Pin C13 is the USER switch, low if pressed.
  }
}
```

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. You will be consulting these two sources often when you program the PIC32. We will come back to understanding the use of these SFRs shortly.

3.2 What Happens When You Reset the PIC32?

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

The first thing your PIC32 does is jump to the first address in boot flash, 0xBFC00000, and begin executing instructions there.[2] For the NU32, we have pre-loaded a "bootloader" program starting 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 a utility program on your computer. When communication is established, the bootloader receives your executable .hex file and writes it to your PIC32's program flash. The bootloader program is written to install your new program starting at 0xBD006100.

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 0xBD006100 where the bootloader writes our program.

The main purpose of using a bootloader on the PIC32 is to allow you to program the PIC32 directly from your computer's USB port. Otherwise, the more traditional way to program the PIC32 is to use another

[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.

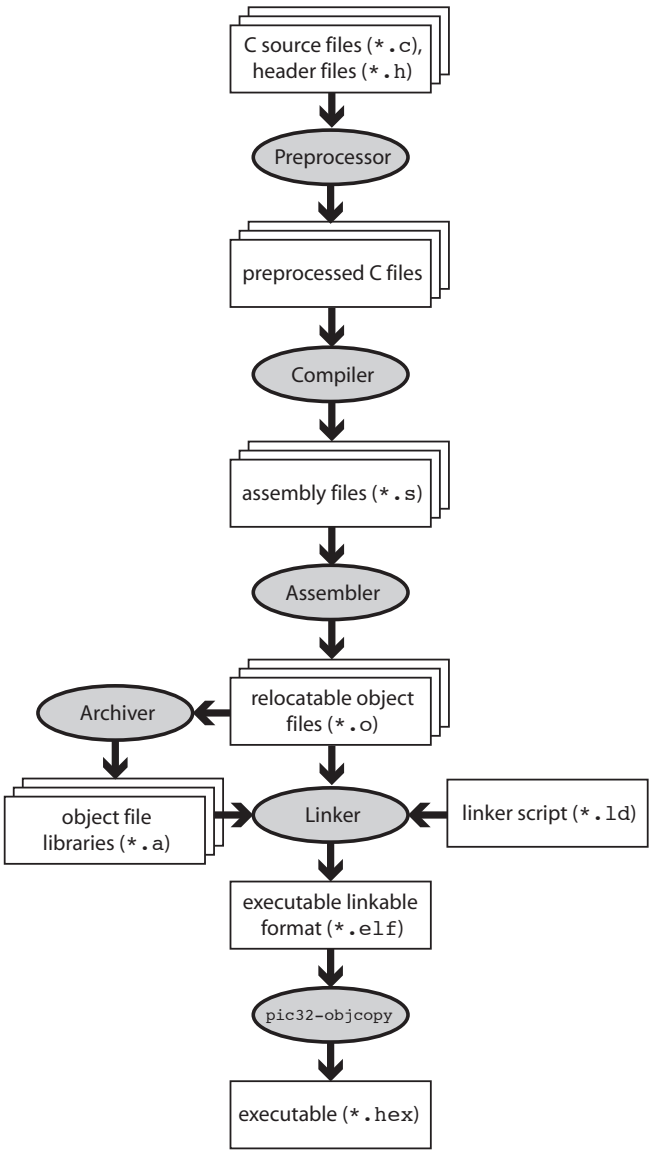

Figure 3.1: The "compilation" process.

device between your USB port and the PIC32 called a "programmer." An example is the Microchip PICkit 3 programmer, which is what we used to put the bootloader program on your PIC32 in the first place.

Anyway, let's say you weren't pressing the USER button. Then the bootloader jumps to 0xBD006100, where your application is sitting, and begins executing. Notice that our program is an infinite loop, so it never stops executing. That is the desired behavior in embedded control. If your program ever does exit, the PIC32 will just sit in a tight loop, doing nothing.

3.3 What Happens When You Compile?

Now let's begin to understand how you created the .hex file in the first place. Figure 3.1 gives a schematic of what happens when you press "Build" in your MPLAB X IDE.

First the **preprocessor** strips out comments and inserts include files. You can have multiple C source files, but only one is allowed to have a `main` function. The other files may contain helper functions, for

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 3.2: Port A registers, taken from Section 4 of the PIC32 Data Sheet.

example.

Then the **compiler** turns these C files into MIPS32 assembly language files, machine commands that are directly understood by the PIC32's CPU. So while some of your C code may be easily "portable" to another microcomputer, your assembly code 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 is 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 libraries, but we will certainly be using `.a` libraries that have already been made for us!

Finally, the **linker** takes multiple object files and produces a single executable file, with all data and program instructions assigned to specific memory locations. The result is an executable and linkable format (`.elf`) file, a standard format. One more step turns the `.elf` file into a `.hex` file 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" is perhaps more accurate, and that is the term the MPLAB X IDE uses.

3.4 Understanding `simple.c`

OK, let's get back to understanding `simple.c`. The `main` function is very simple. It initializes values of 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 we enter a `while` loop, which checks the value of (!PORTCbits.RC13). If PORTCbits.RC13 is 0 (FALSE), then the expression (!PORTCbits.RC13) evaluates to TRUE, and the program stays stuck here, doing nothing but checking the expression (!PORTCbits.RC13). When this evaluates to FALSE, the `while` loop is exited, and we continue 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 PORTC all refer to peripherals that interact with the outside world. Specifically, TRISA and LATA correspond to port A, an input/output port, and PORTC corresponds to port C, another input/output port. We can start our exploration by consulting the table in Section 1 of the Data Sheet (DS) 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 Section 12 of the DS 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. ($0 = \mathrm{O_{utput}}$ and $1 = \mathrm{I_{nput}}$. 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 Section 4 of the DS. Tables there list the VA of many of the SFRs, as well as the value it defaults to upon reset. There are a lot of SFRs!

But after a bit of searching, you find that TRISA sits at 0xBF886000, and its default value upon reset is 0x0000C6FF. (We've reproduced part of this table for you in Figure 3.2.) In binary, this would be

$$0x0000C6FF = 0000\ 0000\ 0000\ 0000\ 1100\ 0110\ 1111\ 1111.$$

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. So all of our pins are configured as inputs, by default. 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.

As a standard, every SFR has 32 bits. Not every SFR uses all of the bits, however, as we see for TRISA.

So now we understand that the instruction

```
TRISA = 0xFFCF;
```

clears bit 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 PORTCbits.RC13, for example.

The two other basic SFRs in this program are LATA and PORTC. Again consulting Section 12 of the DS, 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, PORTC contains the digital inputs on the port C pins. (Notice we didn't configure port C as input; we relied on the fact that it's the default.) PORTCbits.RC13 is 0 if 0 V is present on pin RC13 and 1 if approximately 3.3 V is present.

Pins RA4, RA5, and RC13 on the NU32 Figure 3.3 shows how pins RA4, RA5, and RC13 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, RC13 registers a 0, and otherwise it registers a 1.

The result of these electronics and the `simple.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 Section 12, we see that associated with the SFR LATA are three more SFRs, called LATACLR, LATASET, and LATAINV. (Indeed, all of our port SFRs have corresponding CLR, SET, and INV SFRs.) These are used to easily change some of the bits of LATA without worrying about 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,

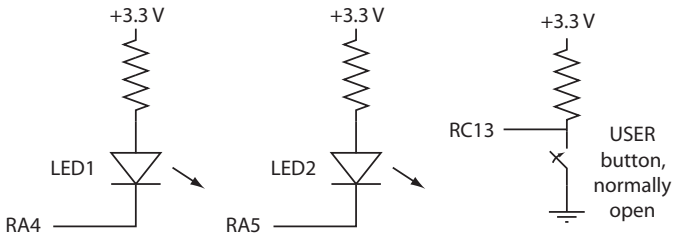

Figure 3.3: The NU32 connection of pins RA4, RA5, and RC13 to LED1, LED2, and a button, respectively.

```
LATAINV = 0x0030;    // flips the values of bits 4 and 5 of LATA; all others are unchanged
LATAINV = 0x30;      // same as above
LATAINV = 0b110000;  // same as above
LATASET = 0x0005;    // sets bits 0 and 2 of LATA to 1; all others are unchanged
LATACLR = 0x0002;    // clears bit 1 of LATA to 0;  all others are 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 later.

You can go back to the table in Section 4 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 `simple.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 `plib.h`. We've been ignoring that `#include <plib.h>` statement until now. Time to take a look.

3.4.1 Down the Rabbit Hole

The `plib` in `plib.h` stands for *peripheral library*, a library of C functions, macros, constants, data types, and variable definitions that Microchip has created for our convenience. But where do we find it? If your program had the preprocessor command `#include "plib.h"`, the preprocessor would start by looking 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. For me, the default include path means that the compiler finds `plib.h` sitting in the directory path

```
microchip/mplabc32/v2.02/pic32mx/include/plib.h
```

Before we open up `plib.h`, let's look at the directory structure that was created when we installed the C32 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 what's going on. Let's start at the level `microchip/mplabc32/v2.02` 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, `pic32-gcc` is the C compiler.
2. `doc`: Some manuals 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 contain the `.c` C files and the `.h` header files needed to create the object code for the PIC32 peripheral library functions. Later on we might want to look at these more closely, as they are some of the best documentation on how the peripheral library functions work. 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 `simple.c`.

   (c) `libc/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 global variables, `crt0` initializes them by writing zeros into their data RAM locations. If your global variables are initialized at the time of declaration, e.g., `int k=3`, then `crt0` copies the initialized values from program flash to data RAM.

6. `pic32-mx`: This directory has a number of files we are interested in.

   (a) `bin`: This directory contains copies of some of the executable programs. We can ignore it.

   (b) `lib`: Important files in this directory include:

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

      ii. `ldscripts/elf32pic32mx.x`: This is a linker script that gives the linker rules on where it is allowed to finally place the relocatable object codes in memory. It uses `procdefs.ld`, which is sitting in your `simple.c` directory. For example, your `procdefs.ld` includes the `processor.o` file with its definitions of the SFR VAs, sets the reset address at 0xBD006100 (which agrees with the bootloader), and allocates segments of data RAM and program flash where the linker is allowed to place the data and instructions.

      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 PIC32. There are also subdirectories with versions of this library with the same functionality, but optimized for speed and code size.

      iv. `proc/32MX795F512L/processor.o`: This object file defines the SFR virtual memory addresses for our PIC32. We can't look at it directly with a text editor, but there are utilities that allow us to examine it. For example, if you are comfortable executing from the command line, you could use the `pic32-nm` program in the top level `bin` directory to see all the SFR VAs:

```
> pic32-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 Section 1 of the Data Sheet.

      v. `proc/32MX795F512L/configuration.data`: This file describes some constants used in setting the configuration bits in DEVCFG0 to DEVCFG3 (Chapter 2.1.4). Our bootloader program chooses the values of these bits. The bootloader program contains the following lines, for example:

```
#pragma config UPLLEN   = ON        // USB PLL Enabled
#pragma config UPLLIDIV = DIV_2     // USB PLL Input Divider
#pragma config FPLLMUL = MUL_20, FPLLIDIV = DIV_2, FPLLODIV = DIV_1, FWDTEN = OFF
#pragma config POSCMOD = HS, FNOSC = PRIPLL, FPBDIV = DIV_1
#pragma config FSOSCEN = OFF
```

These `#pragma`s 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`. Most of these `#pragma`s are used to set up our 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 Section 28 Special Features of the Data Sheet.

(c) `include`: Here we finally find what we were looking for:

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 `p32xxxx.h`. This file is found one directory up in the directory tree. Let's open that next.

iii. `p32xxxx.h`: This file does a few different things, 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? When you were setting up your `simple` project in the first place, you had to specify the processor being used. The MPLAB X IDE passed your answer to the compiler by "defining" the constant `__32MX795F512L__`. This allows the compiler to find the right information about your particular 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 this 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.

3.4.2 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 an unsigned int constructed from bit 0 of this data type
    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 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 declared elsewhere; the compiler does not have to allocate space for it. The `processor.o` file is the one that actually defines the VA of 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 defined 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. The 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`

```
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.

After the definition of TRISA and TRISAbits, we see declarations of TRISACLR, TRISASET, and TRISAINV. They all inherit the VA's specified by `processor.o`.

With these definitions in `p32mx795f512l.h`, the `simple.c` statements

```
TRISA = 0xFFCF;
LATAINV = 0x0030;
while(!PORTCbits.RC13) { }
```

finally make sense. You can see that `p32mx795f512l.h` defines a lot of variables, but no memory has to be allocated for them.

The next 9% of `p32mx795f512l.h` is the `extern` variable declaration of the same SFRs, without the bit field types, for assembly language. These are not useful for us, but 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
```

These refer to the Timer 1 SFR T1CON. Consulting the information about T1CON in Section 14 of the Reference Manual, 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—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
                        // MASKing clears all bits, except bits 4 and 5, which are unchanged
```

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 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, while leaving all other bits of the SFR at their default values. 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 Reference Manual.

The definitions of the `POSITTION`, `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 Section 4 of the Data Sheet). The third and fourth relate to interrupts. The PIC32's MIPS 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).

Finally, `p32mx795f512l.h` concludes by including `ppic32mx.h`, which defines a number of other constants, again with the intent to make your code more readable.

3.5 Summary

OK, that's a lot to digest. Don't worry, you can view the previous few pages as reference material; you don't have to memorize them to program the PIC32!

The main point is that the power of your microcontroller comes from its peripherals, and you interact with your peripherals by reading from and writing to their SFRs. The SFRs are at fixed locations in the virtual memory map, and the linker knows these locations because of the `processor.o` file. And you have access to these locations through conveniently-named variables like TRISA defined in `p32mx795f512l.h`. These variable names correspond to the SFR names in the Data Sheet and Reference Manual, which will always be your definitive sources for information about using the peripherals.

To summarize some of what you've learned:

- Your PIC32 is preloaded with a bootloader. The bootloader has set the configuration bits in DEVCFG0 to DEVCFG3 to turn the 8 MHz resonator input into an 80 MHz SYSCLK and PBCLK and a 48 MHz USBCLK, among other things. The bootloader begins at the PIC32's hardware reset VA 0xBFC00000, so it is the first thing to execute after every reset. If you are not requesting to bootload a new program, the bootloader jumps to the program that you have already installed at 0xBD006100. Otherwise, the bootloader receives the new `.hex` file over its serial port and writes the new program to flash at 0xBD006100.

- All programs are linked with `pic32mx/lib/crt0.o` in producing 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.

- The included file `pic32mx/include/proc/p32mx795f512l.h` contains variable definitions that allow us to read from and write to the SFRs. The VAs of these SFR variable names are set by `pic32mx/lib/proc/32MX795F512L/processor.o`.
- We haven't made use of any of the peripheral library functions yet, but when we do, all of the functions are available for use by linking with the function object codes in the library `pic32mx/lib/libmchp_peripheral_32MX795F512L.a`. These functions are compiled from the source code in `pic32-libs/peripheral`. Header files with function prototypes, constants, and macros for the peripheral library are available in `pic32mx/include`.
- The linker script `pic32mx/lib/ldscripts/elfpic32mx.x` combines with the `procdefs.ld` file sitting in your project's `main` directory to give rules to the linker as to the VA memory segments where it is allowed to place your program instructions and data RAM.

Chapter 5

Space and Time

5.1 Timing

The PIC32 has 6 timers: a 32-bit *core* timer, associated with the MIPS CPU, and five 16-bit *peripheral* timers. We can use the core timer for pure timing operations, leaving the much-more-flexible peripheral timers available for other tasks. The core timer increments for every two ticks of SYSCLK, meaning that the 32-bit counter is incrementing 40 million times per second. Here are some commands you can use:

```
unsigned int elapsed;

WriteCoreTimer(0);             // set the counter value to 0
   ...                         // some code you want to time
elapsed = ReadCoreTimer();     // read the core timer
```

You can use the core timer to check how long it takes certain operations to execute, particularly interrupt service routines.

5.2 Space

The linker allocates VA memory to hold all your program instructions and global variables. The rest of the memory is allocated to the *heap* and the *stack*. The heap is memory set aside to hold dynamic memory allocation, as used by `malloc` and `calloc`, for example. These functions allow you to declare a variable size of memory for an array, for example, while the program is running. By default, MPLAB X assumes you will not use dynamic memory allocation and sets the heap size to zero, so a call to `malloc` would generate an error.

The stack holds local variables used by functions. When a function is called, space in the stack is allocated for its variables. When the function exits, the local variables are thrown away, and the space is made available again.[1]

Here are things you can try to better understand the size of the hex code you are generating:

- Right-click your project name, select Properties, choose the pic32-gcc category, and add `--verbose` to the "Additional options" textbox. Now when you build, you'll see some information on where the compiler is looking for include files, etc.
- Right-click your project name, select Properties, choose the pic32-ld>Diagnostics, and type `-verbose` in the "Additional options" textbox to see information on the linking process.
- Right-click your project name, select Properties, choose the pic32-ld>Diagnostics, and type `program.map` in the "Generate map file" text box. Then open up the `program.map` file in MPLAB X, or any text editor, to see information on the memory used by your program.

[1] Assuming the variable has not been declared with the `static` keyword.

- Choose Window>Output>Disassembly Listing File to see the assembly code version of your program. If you used library functions, you will see a lot of lines of code that you didn't write!
- If you like to dynamically allocate memory using `calloc` or `malloc`, you have to allocate memory to the *heap*, which is allocated zero memory by default. Right-click your project name, select Properties, choose pic32-ld, and set the heap size in bytes.

5.3 Navigating

In your program, if you want to see where certain constants, functions, or variable names are defined in the Microchip library, you can right-click on the symbol in your program, select Navigate, and go to the declaration of the symbol. This may be easier than rooting around in the Microchip directories manually.

5.4 Other Utilities

In the top level `bin` directory of the mplabc32 installation, there are a number of utilities you can run from the command line to inspect various files that are not readable by a text editor. Examples are

- `pic32-ar`: Make library archives and examine them. For example, if you want to see what files are in the math library `libm.a`, you can do `pic32-ar -t libm.a` to see the listing of `.o` object files, then extract one of them using `pic32-ar -x libm.a sqrt32.o`, for example. Now you have that `.o` file, which you can examine using `pic32-objdump`.
- `pic32-objdump`: Try `pic32-objdump -D sqrt32.o` to see the assembly language listing. Or do it on `processor.o` to see all the VAs for the SFRs.
- `pic32-nm`: Another way to see symbols from an object file is `pic32-nm processor.o`.
- `pic32-readelf`: Use `pic32-readelf -a yourprogram.X.production.elf` to see variable and function names and the virtual addresses they were assigned.

Chapter 6

Interrupts

Say your PIC32 is attending to some mundane task when an important event occurs. For example, the user has pressed a button. We want the PIC32 to respond immediately. To do so, we have this event generate an interrupt, or *interrupt request* (IRQ), which interrupts the program and sends the CPU to execute some other code, called the *interrupt service routine* (or ISR). Once this code has completed, the CPU returns to its original task.

Interrupts are a key concept in real-time control, and they can arise from many different events. This page provides a summary of PIC32 interrupt handling.

6.1 Overview

Interrupts can be generated by the processor core, peripherals, and external inputs. Example events include

- a digital input changing its value,
- information arriving on a communication port,
- the completion of some task a PIC32 peripheral was executing in parallel with the CPU, and
- the elapsing of a specified amount of time.

As an example, to guarantee performance in real-time control applications, we must read the sensors and calculate new control signals at a known fixed rate. For a robot arm, a common control loop frequency is 1 kHz. So we would configure one of the PIC32's counter/timers to use the peripheral bus clock as input, and choose prescaler and period register values so that the counter rolls over every 1 ms. This roll-over "event" generates the interrupt that calls our feedback control ISR, which reads sensors and produces output. In this case, we would have to make sure that our control ISR is efficient code that always executes in less than 1 ms. (For example, you could use the core timer to measure the time between entering and exiting the ISR.)

Say the PIC32 is currently controlling the robot arm to hold steady in a particular position. It then receives new information from the user, who asks the arm to move to a new location. This new information also generates an interrupt, and the corresponding ISR reads in the information and stores it in global variables representing the desired state. These desired states are used in the feedback control ISR.

So what happens if we are in the middle of executing the control ISR and a communication interrupt is generated? Or if we are in the middle of the communication ISR and a control interrupt is generated? We must make a choice about which has higher priority. If a high priority interrupt occurs while a low priority ISR is executing, the CPU will jump to the high priority ISR, complete it, and then return to finish the low priority ISR. If a low priority interrupt occurs while a high priority ISR is executing, the low priority ISR will wait patiently until the high priority ISR is finished executing. When it is finished, the CPU jumps to the low priority ISR.

In our example, communication could be slow, taking several milliseconds to complete, and we might not have a guarantee as to the duration. To ensure the stability of the robot arm, we would probably choose the control interrupt to have higher priority than the communication interrupt.

Every time an interrupt is generated, the CPU must save the contents of the internal CPU registers, called the "context," to data RAM. It then uses its registers in the execution of the ISR. After the ISR completes, it copies the context from RAM back to its registers, so that it can start where it left off before the interrupt. The copying of register data back and forth is called "context save and restore." If interrupts are piling up (one ISR interrupts another which has interrupted another, etc., before any of them finish), then the PIC32 could potentially run out of RAM, causing a "stack overflow" and the program to crash. These errors can be very difficult to debug, so it is a good idea to ensure that your ISRs execute quickly if you are using several different interrupt sources.

The address of the ISR in virtual memory is derived from the *interrupt vector*, and the PIC32 can support up to 64 unique interrupt vectors (and therefore 64 ISRs) arising from up to 96 different interrupt request sources (IRQs). If all interrupts jump to the same ISR, the PIC32 is in "single vector mode." This is the default on reset. If each interrupt has its own ISR, the PIC32 is in "multi-vector mode." This is the way we will typically use it.

6.2 Details

Before doing anything else, the SFR INTCON (Interrupt Control) should be set to multi-vector mode, since we want the flexibility to call different ISRs depending on the interrupt condition. This is done by setting bit 12 of INTCON to 1, as we will see below. We refer to this bit as INTCON$\langle 12 \rangle$.

The CPU jumps to an ISR when three conditions are satisfied: (1) the interrupt has been enabled by setting a bit in the SFR IECx (Interrupt Enable Control) to 1; (2) an interrupt has been requested by setting the same bit in the SFR IFSx (Interrupt Flag Status) to 1; and (3) the priority of the interrupt, as represented in the SFR IPCy (Interrupt Priority Register), is greater than the current priority of the CPU. If the first two conditions are satisfied, but not the third, the interrupt remains pending until the CPU's priority drops lower.

The "x" in the IECx and IFSx SFRs above can be 0, 1, or 2, allowing up to (3 registers) $\times$ (32 bits) = 96 interrupt sources. The "y" in IPCy takes values $0 \ldots 15$, and each of the IPCy registers can contain the priority level for four different interrupt vectors, i.e., up to (16 registers) $\times$ (four vectors per register) = 64 vectors. Each of the 64 priority levels is represented by five bits, three indicating the priority (taking values 0 to 7) and two indicating the subpriority (taking values 0 to 3). Thus each IPCy has 20 relevant bits: five for each of the four vectors.

As an example, an input change notification (CN) pin can generate an interrupt when its voltage changes. The change notification's interrupt has x=1 and y=6, so information about this interrupt is stored in IFS1, IEC1, and IPC6. Specifically, IFS1$\langle 0 \rangle$ is its interrupt flag status bit, IEC1$\langle 0 \rangle$ is its interrupt enable bit, IPC6$\langle 20{:}18 \rangle$ are the three priority bits for its interrupt vector, and IPC0$\langle 17{:}16 \rangle$ are its two subpriority bits.

The list of interrupt sources and their corresponding bit locations are given in the table below, reproduced from Section 7 of the Data Sheet.

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION

Interrupt Source[1]	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
Highest Natural Order Priority						
CT – Core Timer Interrupt	0	0	IFS0<0>	IEC0<0>	IPC0<4:2>	IPC0<1:0>
CS0 – Core Software Interrupt 0	1	1	IFS0<1>	IEC0<1>	IPC0<12:10>	IPC0<9:8>
CS1 – Core Software Interrupt 1	2	2	IFS0<2>	IEC0<2>	IPC0<20:18>	IPC0<17:16>
INT0 – External Interrupt 0	3	3	IFS0<3>	IEC0<3>	IPC0<28:26>	IPC0<25:24>
T1 – Timer1	4	4	IFS0<4>	IEC0<4>	IPC1<4:2>	IPC1<1:0>
IC1 – Input Capture 1	5	5	IFS0<5>	IEC0<5>	IPC1<12:10>	IPC1<9:8>
OC1 – Output Compare 1	6	6	IFS0<6>	IEC0<6>	IPC1<20:18>	IPC1<17:16>
INT1 – External Interrupt 1	7	7	IFS0<7>	IEC0<7>	IPC1<28:26>	IPC1<25:24>
T2 – Timer2	8	8	IFS0<8>	IEC0<8>	IPC2<4:2>	IPC2<1:0>
IC2 – Input Capture 2	9	9	IFS0<9>	IEC0<9>	IPC2<12:10>	IPC2<9:8>
OC2 – Output Compare 2	10	10	IFS0<10>	IEC0<10>	IPC2<20:18>	IPC2<17:16>
INT2 – External Interrupt 2	11	11	IFS0<11>	IEC0<11>	IPC2<28:26>	IPC2<25:24>
T3 – Timer3	12	12	IFS0<12>	IEC0<12>	IPC3<4:2>	IPC3<1:0>
IC3 – Input Capture 3	13	13	IFS0<13>	IEC0<13>	IPC3<12:10>	IPC3<9:8>
OC3 – Output Compare 3	14	14	IFS0<14>	IEC0<14>	IPC3<20:18>	IPC3<17:16>
INT3 – External Interrupt 3	15	15	IFS0<15>	IEC0<15>	IPC3<28:26>	IPC3<25:24>
T4 – Timer4	16	16	IFS0<16>	IEC0<16>	IPC4<4:2>	IPC4<1:0>
IC4 – Input Capture 4	17	17	IFS0<17>	IEC0<17>	IPC4<12:10>	IPC4<9:8>
OC4 – Output Compare 4	18	18	IFS0<18>	IEC0<18>	IPC4<20:18>	IPC4<17:16>
INT4 – External Interrupt 4	19	19	IFS0<19>	IEC0<19>	IPC4<28:26>	IPC4<25:24>
T5 – Timer5	20	20	IFS0<20>	IEC0<20>	IPC5<4:2>	IPC5<1:0>
IC5 – Input Capture 5	21	21	IFS0<21>	IEC0<21>	IPC5<12:10>	IPC5<9:8>
OC5 – Output Compare 5	22	22	IFS0<22>	IEC0<22>	IPC5<20:18>	IPC5<17:16>
SPI1E – SPI1 Fault	23	23	IFS0<23>	IEC0<23>	IPC5<28:26>	IPC5<25:24>
SPI1RX – SPI1 Receive Done	24	23	IFS0<24>	IEC0<24>	IPC5<28:26>	IPC5<25:24>
SPI1TX – SPI1 Transfer Done	25	23	IFS0<25>	IEC0<25>	IPC5<28:26>	IPC5<25:24>
U1E – UART1 Error	26	24	IFS0<26>	IEC0<26>	IPC6<4:2>	IPC6<1:0>
SPI3E – SPI3 Fault						
I2C3B – I2C3 Bus Collision Event						
U1RX – UART1 Receiver	27	24	IFS0<27>	IEC0<27>	IPC6<4:2>	IPC6<1:0>
SPI3RX – SPI3 Receive Done						
I2C3S – I2C3 Slave Event						
U1TX – UART1 Transmitter	28	24	IFS0<28>	IEC0<28>	IPC6<4:2>	IPC6<1:0>
SPI3TX – SPI3 Transfer Done						
I2C3M – I2C3 Master Event						
I2C1B – I2C1 Bus Collision Event	29	25	IFS0<29>	IEC0<29>	IPC6<12:10>	IPC6<9:8>
I2C1S – I2C1 Slave Event	30	25	IFS0<30>	IEC0<30>	IPC6<12:10>	IPC6<9:8>
I2C1M – I2C1 Master Event	31	25	IFS0<31>	IEC0<31>	IPC6<12:10>	IPC6<9:8>
CN – Input Change Interrupt	32	26	IFS1<0>	IEC1<0>	IPC6<20:18>	IPC6<17:16>
AD1 – ADC1 Convert Done	33	27	IFS1<1>	IEC1<1>	IPC6<28:26>	IPC6<25:24>

Note 1: Not all interrupt sources are available on all devices. See Table 1, Table 2 and Table 3 for the list of available peripherals.

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION (CONTINUED)

Interrupt Source(1)	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
PMP – Parallel Master Port	34	28	IFS1<2>	IEC1<2>	IPC7<4:2>	IPC7<1:0>
CMP1 – Comparator Interrupt	35	29	IFS1<3>	IEC1<3>	IPC7<12:10>	IPC7<9:8>
CMP2 – Comparator Interrupt	36	30	IFS1<4>	IEC1<4>	IPC7<20:18>	IPC7<17:16>
U3E – UART2A Error SPI2E – SPI2 Fault I2C4B – I2C4 Bus Collision Event	37	31	IFS1<5>	IEC1<5>	IPC7<28:26>	IPC7<25:24>
U3RX – UART2A Receiver SPI2RX – SPI2 Receive Done I2C4S – I2C4 Slave Event	38	31	IFS1<6>	IEC1<6>	IPC7<28:26>	IPC7<25:24>
U3TX – UART2A Transmitter SPI2TX – SPI2 Transfer Done IC4M – I2C4 Master Event	39	31	IFS1<7>	IEC1<7>	IPC7<28:26>	IPC7<25:24>
U2E – UART3A Error SPI4E – SPI4 Fault I2C5B – I2C5 Bus Collision Event	40	32	IFS1<8>	IEC1<8>	IPC8<4:2>	IPC8<1:0>
U2RX – UART3A Receiver SPI4RX – SPI4 Receive Done I2C5S – I2C5 Slave Event	41	32	IFS1<9>	IEC1<9>	IPC8<4:2>	IPC8<1:0>
U2TX – UART3A Transmitter SPI4TX – SPI4 Transfer Done IC5M – I2C5 Master Event	42	32	IFS1<10>	IEC1<10>	IPC8<4:2>	IPC8<1:0>
I2C2B – I2C2 Bus Collision Event	43	33	IFS1<11>	IEC1<11>	IPC8<12:10>	IPC8<9:8>
I2C2S – I2C2 Slave Event	44	33	IFS1<12>	IEC1<12>	IPC8<12:10>	IPC8<9:8>
I2C2M – I2C2 Master Event	45	33	IFS1<13>	IEC1<13>	IPC8<12:10>	IPC8<9:8>
FSCM – Fail-Safe Clock Monitor	46	34	IFS1<14>	IEC1<14>	IPC8<20:18>	IPC8<17:16>
RTCC – Real-Time Clock and Calendar	47	35	IFS1<15>	IEC1<15>	IPC8<28:26>	IPC8<25:24>
DMA0 – DMA Channel 0	48	36	IFS1<16>	IEC1<16>	IPC9<4:2>	IPC9<1:0>
DMA1 – DMA Channel 1	49	37	IFS1<17>	IEC1<17>	IPC9<12:10>	IPC9<9:8>
DMA2 – DMA Channel 2	50	38	IFS1<18>	IEC1<18>	IPC9<20:18>	IPC9<17:16>
DMA3 – DMA Channel 3	51	39	IFS1<19>	IEC1<19>	IPC9<28:26>	IPC9<25:24>
DMA4 – DMA Channel 4	52	40	IFS1<20>	IEC1<20>	IPC10<4:2>	IPC10<1:0>
DMA5 – DMA Channel 5	53	41	IFS1<21>	IEC1<21>	IPC10<12:10>	IPC10<9:8>
DMA6 – DMA Channel 6	54	42	IFS1<22>	IEC1<22>	IPC10<20:18>	IPC10<17:16>
DMA7 – DMA Channel 7	55	43	IFS1<23>	IEC1<23>	IPC10<28:26>	IPC10<25:24>
FCE – Flash Control Event	56	44	IFS1<24>	IEC1<24>	IPC11<4:2>	IPC11<1:0>
USB – USB Interrupt	57	45	IFS1<25>	IEC1<25>	IPC11<12:10>	IPC11<9:8>
CAN1 – Control Area Network 1	58	46	IFS1<26>	IEC1<26>	IPC11<20:18>	IPC11<17:16>
CAN2 – Control Area Network 2	59	47	IFS1<27>	IEC1<27>	IPC11<28:26>	IPC11<25:24>
ETH – Ethernet Interrupt	60	48	IFS1<28>	IEC1<28>	IPC12<4:2>	IPC12<1:0>
IC1E – Input Capture 1 Error	61	5	IFS1<29>	IEC1<29>	IPC1<12:10>	IPC1<9:8>
IC2E – Input Capture 2 Error	62	9	IFS1<30>	IEC1<30>	IPC2<12:10>	IPC2<9:8>
IC3E – Input Capture 3 Error	63	13	IFS1<31>	IEC1<31>	IPC3<12:10>	IPC3<9:8>
IC4E – Input Capture 4 Error	64	17	IFS2<0>	IEC2<0>	IPC4<12:10>	IPC4<9:8>

Note 1: Not all interrupt sources are available on all devices. See Table 1, Table 2 and Table 3 for the list of available peripherals.

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION (CONTINUED)

Interrupt Source(1)	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
IC4E – Input Capture 5 Error	65	21	IFS2<1>	IEC2<1>	IPC5<12:10>	IPC5<9:8>
PMPE – Parallel Master Port Error	66	28	IFS2<2>	IEC2<2>	IPC7<4:2>	IPC7<1:0>
U4E – UART4 Error	67	49	IFS2<3>	IEC2<3>	IPC12<12:10>	IPC12<9:8>
U4RX – UART4 Receiver	68	49	IFS2<4>	IEC2<4>	IPC12<12:10>	IPC12<9:8>
U4TX – UART4 Transmitter	69	49	IFS2<5>	IEC2<5>	IPC12<12:10>	IPC12<9:8>
U6E – UART6 Error	70	50	IFS2<6>	IEC2<6>	IPC12<20:18>	IPC12<17:16>
U6RX – UART6 Receiver	71	50	IFS2<7>	IEC2<7>	IPC12<20:18>	IPC12<17:16>
U6TX – UART6 Transmitter	72	50	IFS2<8>	IEC2<8>	IPC12<20:18>	IPC12<17:16>
U5E – UART5 Error	73	51	IFS2<9>	IEC2<9>	IPC12<28:26>	IPC12<25:24>
U5RX – UART5 Receiver	74	51	IFS2<10>	IEC2<10>	IPC12<28:26>	IPC12<25:24>
U5TX – UART5 Transmitter	75	51	IFS2<11>	IEC2<11>	IPC12<28:26>	IPC12<25:24>
(Reserved)	—	—	—	—	—	—
Lowest Natural Order Priority						

Note 1: Not all interrupt sources are available on all devices. See Table 1, Table 2 and Table 3 for the list of available peripherals.

Since there are more IRQs than interrupt vectors, some IRQs share the same vector. For example, IRQs 26, 27, and 28, each corresponding to UART1 events, all share the same vector.

When the CPU has multiple interrupts pending, it first processes the one whose vector has the highest priority level. (Note priorities are associated with the vectors, not the IRQs.) If there is more than one at the same highest priority level, the CPU first processes the one with the higher subpriority. If interrupts have the same priority and subpriority, then their priority is resolved using the "natural order priority" table given above. (In this case, however, lower IRQ numbers have higher priority!) Finally, if the CPU is currently processing an IRQ at a particular priority level, and it receives an IRQ request at the same priority, it will complete its current ISR before servicing the other IRQ, regardless of its subpriority.

If the priority of a vector is zero, then the interrupt is disabled. We have seven enabled priority levels.

The last thing any ISR should do is clear the interrupt flag (clear the appropriate bit of IFSx to zero), indicating that the interrupt has been serviced and the CPU is free to return to the the program state when the ISR was called.

When setting up an interrupt, you will set a bit in IECx to 1 indicating the interrupt is enabled (all bits are set to zero upon reset) and assign values to the associated IPCy priority bits. (These priority bits default to zero upon reset, which will keep the interrupt disabled.) As mentioned above, you will also clear the IFSx bit at the end of the ISR. You will generally never write code setting the IFSx bit to 1. Instead, when you set up the device that generates the interrupt (e.g., a UART or counter/timer), you will indicate that it should set the interrupt flag upon the appropriate event.

The Shadow Register Set The PIC32's CPU provides an internal *shadow register set*, which is a full extra set of registers. You can take advantage of this extra register set to avoid the time needed for context save and restore. When processing an ISR using the SRS, the CPU simply switches to this extra set of internal registers. When it finishes the ISR, it switches back to its original register set, without needing to save and restore them. We will see an example of this in Section 6.3. ISRs using the SRS obviously should not be interrupted by other ISRs wishing to use the SRS! Typically there is no need to use the SRS unless you care about timing down to the microsecond or so.

In single vector mode, you can choose the value of INTCON⟨16⟩ to either use or not use the SRS. I don't know why you wouldn't always use it.

Special Function Registers

Apart from the SFRs INTCON, IECx, IFSx, and IPCy, described above, two other SFRs are relevant to interrupts: INTSTAT (Interrupt Status) and TPTMR (Temporal Proximity Timer). The SFRs are summarized below.

INTCON (Interrupt Control) Determines whether the interrupt controller operates in single vector or multi-vector mode. Also determines whether the five external interrupt pins INT0 ... INT4 generate an interrupt on a rising edge or a falling edge.

INTSTAT (Interrupt Status) Read-only: information on the address and priority level of the latest IRQ given to the CPU when in single vector mode. We will not need it.

IPTMR (Interrupt Proximity Timer Register) A timer can be used to implement a delay to queue up interrupt requests before presenting them to the CPU. For example, upon receiving an interrupt request, the timer starts counting clock cycles, queuing up any subsequent interrupt requests, until IPTMR cycles have passed. By default, this timer is turned off by INTCON, and you will typically leave it that way.

IFSx (Interrupt Flag Status) Three 32-bit SFRs for up to 96 interrupt sources (x = 0, 1, or 2). A 1 indicates an interrupt has been requested, a 0 indicates no interrupt is requested.

IECx (Interrupt Enable Control) Three 32-bit SFRs for up to 96 interrupt sources (x = 0, 1, or 2). A 1 enables the interrupt, a 0 disables it.

IPCy (Interrupt Priority Control) Sixteen registers, each with 5 bit priority values for 4 different interrupt vectors (64 vectors total).

In this chapter only, we reproduce some register information from the PIC32 Reference Manual. You should always consult the appropriate sections from the Reference Manual and the Data Sheet for more information.

Register 8-1: INTCON: Interrupt Control Register

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
	—	—	—	—	—	—	—	—
23:16	U-0	U-0	U-0	U-0	U-0	U-0	U-0	R/W-0
	—	—	—	—	—	—	—	SS0
15:8	U-0	U-0	U-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
	—	—	—	MVEC	—	TPC<2:0>		
7:0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	INT4EP	INT3EP	INT2EP	INT1EP	INT0EP

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit	P = Programmable bit
-n = Bit Value at POR:('0', '1', x = Unknown)			

- INTCON⟨16⟩, known as SS0: 1 = use the shadow register set when in single vector mode, 0 = do not use
- INTCON⟨12⟩, known as MVEC: 1 = interrupt controller in multi-vector mode, 0 = single vector mode
- INTCON⟨10:8⟩, known as TPC⟨2:0⟩: control bits for the IPTMR (we leave it at the default of 000 = IPTMR off)
- INTCON⟨x⟩, for x = 0 to 4, known as INTxEP: 1 = external interrupt pin x triggers on a rising edge, 0 = triggers on a falling edge

Register 8-4: IFSx: Interrupt Flag Status Register(1)

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IFS31	IFS30	IFS29	IFS28	IFS27	IFS26	IFS25	IFS24
23:16	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IFS23	IFS22	IFS21	IFS20	IFS19	IFS18	IFS17	IFS16
15:8	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IFS15	IFS14	IFS13	IFS12	IFS11	IFS10	IFS09	IFS08
7:0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IFS07	IFS06	IFS05	IFS04	IFS03	IFS02	IFS01	IFS00

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit	P = Programmable bit
-n = Bit Value at POR:('0', '1', x = Unknown)			

1 = interrupt has been requested, 0 = no interrupt has been requested. See Section 7 of the Data Sheet, or the table reproduced earlier, for the the register number x in IFSx, and the bit number, for a particular IRQ source. For example, the change notification interrupt request bit is IFS1⟨0⟩.

Register 8-5: IECx: Interrupt Enable Control Register(1)

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IEC31	IEC30	IEC29	IEC28	IEC27	IEC26	IEC25	IEC24
23:16	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IEC23	IEC22	IEC21	IEC20	IEC19	IEC18	IEC17	IEC16
15:8	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IEC15	IEC14	IEC13	IEC12	IEC11	IEC10	IEC09	IEC08
7:0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IEC07	IEC06	IEC05	IEC04	IEC03	IEC02	IEC01	IEC00

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit	P = Programmable bit
-n = Bit Value at POR:('0', '1', x = Unknown)			

1 = interrupt has been enabled so that requests are allowed, 0 = interrupt is disabled. See Section 7 of the Data Sheet, or the table reproduced earlier, for the the register number x in IECx, and the bit number, for a particular IRQ source. For example, the change notification interrupt enable bit is IEC1⟨0⟩.

Register 8-6: IPCx: Interrupt Priority Control Register(1)

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	IP03<2:0>			IS03<1:0>	
23:16	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	IP02<2:0>			IS02<1:0>	
15:8	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	IP01<2:0>			IS01<1:0>	
7:0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	IP00<2:0>			IS00<1:0>	

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit	P = Programmable bit
-n = Bit Value at POR:('0', '1', x = Unknown)			

Each IPCy, y = 0 to 15, contains five priority and subpriority bits for each of four different interrupt vectors. For example, consulting the table, we see that IPC6⟨20:18⟩ are the three priority bits for the change notification interrupt vector, and IPC6⟨17:16⟩ are its two subpriority bits.

6.3 Sample Code

You need to do three things to use an interrupt:

1. Write an ISR, with an assigned priority level, that ends by clearing the appropriate interrupt flag IFSx⟨bit⟩.
2. Configure the device that you want to generate interrupts, so that it will generate an interrupt on the appropriate event. The priority that you set in this configuration must match the priority in the ISR.
3. Configure the CPU to receive interrupts in multi-vector mode and enable the CPU to process interrupts.

The last of these involves commands to the CPU, so we will use the peripheral library commands; there is no version based on manipulating SFRs.

Sometimes it is necessary to disable interrupts. You can achieve this by

```
INTDisableInterrupts();
```

You might do this when you are reconfiguring interrupt behavior, for example, so you don't get a spurious interrupt while you are making the change.

6.3.1 Core Timer Interrupt

Let's toggle a digital output once a second based on an interrupt from the CPU's core timer. To do this, we place a value in the CPU's CP0_COMPARE register, and whenever the core timer counter value is equal to CP0_COMPARE, an interrupt is generated. Since the core timer runs at half the frequency of the system clock, we set CP0_COMPARE to 40,000,000 to toggle the digital output once per second.

To make the effect visible, we will toggle pin RA5, which corresponds to LED2 on the NU32 board. Let's go ahead and use priority level 7 to use the shadow register set.

Code Sample 6.1. A core timer interrupt using the shadow register set.

```
// toggle a digital out at a fixed interval using a core timer interrupt

#include <plib.h>

#define CORE_TICKS 40000000                          // 40 M ticks per second

void __ISR(_CORE_TIMER_VECTOR, IPL7SRS) CoreTimerISR(void) {
  LATAINV = 0x20;                                    // invert pin RA5 only
  WriteCoreTimer(0);                                 // set core timer counter to 0
  _CP0_SET_COMPARE(CORE_TICKS);                      // must set CP0_COMPARE again after interrupt
  INTClearFlag(_CORE_TIMER_IRQ);                     // clear the interrupt flag
}

void main(void) {
  TRISACLR = 0x30;                                   // pins RA4 and RA5 are outputs

  mConfigIntCoreTimer(CT_INT_ON | CT_INT_PRIOR_7);   // enable CT interrupts with IPL7
  _CP0_SET_COMPARE(CORE_TICKS);                      // CP0_COMPARE register set to 40 M

  INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR); // set CPU for multi-vector interrupts
  INTEnableInterrupts();                             // set CPU to be able to receive interrupts

  WriteCoreTimer(0);                                 // set core timer counter to 0

  while(1);                                          // infinite loop
}
```

Following our three major steps in using an interrupt, we have

Step 1. The ISR.

```
void __ISR(_CORE_TIMER_VECTOR, IPL7SRS) CoreTimerISR(void) {
  // ...
  INTClearFlag(_CORE_TIMER_IRQ);                     // clear the interrupt flag
}
```

We are allowed to call our ISR whatever we want, and in this example we call it `CoreTimerISR`. The `__ISR` syntax is Microchip-specific (not a C standard) and tells the compiler and linker that this function should be treated as an interrupt handler. The two arguments to this syntax are the interrupt vector for the core timer `_CORE_TIMER_VECTOR` (defined as 0 in `p32mx795f512l.h`, which agrees with the table) and the interrupt priority level. The interrupt priority level is specified using the syntax `IPLnSRS` or `IPLnSOFT`, where `n` is 1 to 7, `SRS` indicates that the shadow register set should be used, and `SOFT` indicates that software context save and restore should be used. Use `IPL7SRS` if you'd like to use the shadow register set, as in this example[1], and `IPLnSOFT` otherwise, where you would typically choose `n` to be 1 to 6. You don't specify subpriority in defining the ISR; that's for the setting up of the device that generates the interrupt, next.

Note that the last task in the ISR is to clear the interrupt flag, `_CORE_TIMER_IRQ`, again defined in `p32mx795f512l.h`.

Step 2. Configuring the core timer to interrupt.

```
  mConfigIntCoreTimer(CT_INT_ON | CT_INT_PRIOR_7);   // enable CT interrupts with IPL7
  _CP0_SET_COMPARE(CORE_TICKS);                      // CP0_COMPARE register set to 40 M
```

[1]The interrupt priority level that uses the shadow register set is actually defined by DEVCFG3, but this defaults to 7 for us.

The "m" in `mConfigIntCoreTimer` indicates that it is a macro, not a function, and it is defined in `pic32mx/include/peripheral/timer.h`. It clears the interrupt flag IFS0⟨0⟩, writes the priority and subpriority to IPC0⟨4:2⟩ and IPC0⟨1:0⟩, and enables the core timer interrupt at IEC0⟨0⟩. In this example, we did not specify a subpriority (the default is OK for us). The second line sets the core timer's CP0_COMPARE value so that an interrupt is generated when the core timer counter reaches `CORE_TICKS`.

Step 3. Configure the CPU to receive multi-vector interrupts and tell the CPU to accept interrupt requests.

```
INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR); // set CPU for multiple ISRs
INTEnableInterrupts();                             // set CPU to be able to receive interrupts
```

The first line tells the CPU to enter multi-vector interrupt mode and sets INTCON⟨12⟩. (The other argument option is `INT_SYSTEM_CONFIG_SINGLE_VECTOR`.) The second line sets the appropriate bit in a CPU register so that the CPU begins to accept interrupts.

6.3.2 External Interrupt

What does the following program do?

Code Sample 6.2.

```
#include <plib.h>

void __ISR(_EXTERNAL_0_VECTOR, IPL2SOFT) CoreTimerISR(void) {
  LATA = 0x00;
  WriteCoreTimer(0);
  while(ReadCoreTimer()<10000000);
  LATA = 0x30;
  IFS0CLR = 1<<3;
}

void main(void) {
  TRISACLR = 0x30;
  INTCONSET = 1;
  IEC0SET = 1<<3;
  IPC0 = 9<<24;
  IFS0bits.INT0IF = 0;
  INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);
  INTEnableInterrupts();
  while(1);
}
```