CHAPTER 5

Time and Space

How long does your program take to execute? How much RAM does it require? How much flash does it occupy? Fast processing speeds and plentiful memory have reduced the importance of these questions for programmers of personal computers. After all, if a process takes a few microseconds longer or uses a few extra megabytes of RAM, the user may not notice the difference. On embedded systems, however, efficiency is important: processors are relatively slow, control commands must execute in a timely manner, and RAM and flash are precious resources. Additionally, specific timing requirements may be imposed on your system due to physics: for example, you might need to provide commands to a motor at regular, timely intervals to control it properly. If your code is too slow, it may fail to accomplish its purpose.

Writing efficient code is a balancing act. Code can be time-efficient (runs fast), RAM-efficient (uses less RAM), flash-efficient (has a smaller executable size), but perhaps most importantly, programmer-time-efficient (minimizes the time needed to write and debug the code, or for a future programmer to understand and modify it). Often these interests compete with each other. Some XC32 compiler options reflect this trade-off, allowing you to explicitly make space-time tradeoffs.[1] For example, the compiler could "unroll" loops. If a loop is known to be executed 20 times, for example, instead of using a small piece of code, incrementing a counter, and checking to see if the count has reached 20, the compiler could simply write the same block of code 20 times. This may save some execution time (no counter increments, no conditional tests, no branches) at the expense of using more flash to store the program.

This chapter explains some of the tools available for understanding the time and space consumed by your program. These tools will not only help you squeeze the most out of your PIC32; by writing more efficient code you can often choose a less expensive microcontroller. More importantly, though, you will better understand how your software works.

[1] Some options are not available in the free version of the compiler.

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00005-6

5.1 Compiler Optimization

The XC32 compiler provides five levels of optimization. Their availability depends on whether you have a license for the free version of the compiler, the Standard version, or the Pro version:

Version	Label	Description
All	`00`	no optimization
All	`01`	level 1: attempts to reduce both code size and execution time
Standard, Pro	`02`	level 2: further reduces code size and execution time beyond `01`
Pro	`03`	level 3: maximum optimization for speed
Pro	`0s`	maximum optimization for code size

The greater the optimization, the longer it takes the compiler to produce the assembly code. You can learn more about compiler optimization in the XC32 C/C++ Compiler User's Guide.

When you issue a `make` command with the `Makefile` from the quickstart code, you see that the compiler is invoked with optimization level `01`, using commands like

```
xc32-gcc -g -O1 -x c ...
```

`-g -O1 -x c` are compiler flags set in the variable `CFLAGS` in the `Makefile`. The `-O1` means that optimization level 1 is being requested.

In this chapter, we examine the assembly code that the compiler produces from your C code. The mapping between your C code and the assembly code is relatively direct when no optimization is used, but is less clear when optimization is invoked. (We will see an example of this in Section 5.2.3.) To create clearer assembly code, we will find it useful to be able to `make` files with no optimization. You can override the compilation flags by specifying the `CFLAGS` `Makefile` variable at the command line:

```
> make CFLAGS="-g -x c"
```

or

```
> make write CFLAGS="-g -x c"
```

Alternatively, you could edit the `Makefile` line to remove the `-O1` where `CFLAGS` is defined and just use `make` and `make write` as usual.

In these examples, since no optimization level is specified, the default (no optimization) is applied. Unless otherwise specified, all examples in this chapter assume that no optimization is applied.

5.2 Time and the Disassembly File

5.2.1 Timing Using a Stopwatch (or an Oscilloscope)

A direct way to time something is to toggle a digital output and look at that digital output using an oscilloscope or stopwatch. For example:

```
   ...                        // digital output RF0 has been high for some time
LATFCLR = 0x1;                // clear RF0 to 0 (turn on NU32 LED1)
   ...                        // some code you want to time
LATFSET = 0x1;                // set RF0 to 1 (turn off LED1)
```

The time that RF0 is low (or LED1 is on) approximates the execution time of the code.

If the duration is too short to measure with your scope or stopwatch, you could modify the code to something like

```
   ...                         // digital output RF0 has been high for some time
LATFCLR = 0x1;                 // clear RF0 to 0 (turn on NU32 LED1)
for (i=0; i<1000000; i++) {    // modify 1,000,000 as appropriate for you
   ...                         // some code you want to time
}
LATFSET = 0x1;                 // set RF0 to 1 (turn off LED1)
```

Then you can divide the total time by 1,000,000.[2] Remember, however, that the `for` loop introduces additional overhead: for example, instructions to increment the counter and check the inequality. We will examine the overhead in Section 5.2.3. If the code you want to time uses only a few assembly instructions, then the time you actually measure will be dominated by the implementation of the `for` loop.

5.2.2 Timing Using the Core Timer

A more accurate time can be obtained using a timer onboard the PIC32. The NU32's PIC32 has six timers: a 32-bit *core* timer, associated with the MIPS32 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 (see Chapter 8). The core timer increments once for every two ticks of SYSCLK. For a SYSCLK of 80 MHz, the timer increments every 25 ns. Because the timer is 32 bits, it rolls over every $2^{32} \times 25$ ns $= 107$ s.

[2] If you use optimization in compiling your program, however, the compiler might recognize that you are not doing anything with the results of the loop, and not generate assembly code for the loop at all! You can place a `_nop()` macro in the loop to force it to remain. The `_nop()` inserts a `nop` assembly instruction that does nothing but the compiler cannot remove it.

The include file `<cp0defs.h>` contains two macros for accessing the core timer: `_CP0_GET_COUNT()` and `_CP0_SET_COUNT(val)`. When you include `NU32.h`, it includes `<xc.h>` which, in turn, includes `<cp0defs.h>`, so you typically already have access to the required macros. The macro `_CP0_GET_COUNT()` returns the current core timer count while `_CP0_SET_COUNT(val)` sets the count to `val`.

```
unsigned int elapsedticks, elapsedns;

_CP0_SET_COUNT(0);                // set the core timer counter to 0
   ...                            // some code you want to time
elapsedticks = _CP0_GET_COUNT();  // read the core timer
elapsedns = elapsedticks * 25;    // duration in ns, for 80 MHz SYSCLK
```

If the core timer is being used to time different things, do not reset the counter to zero. Instead, read the value `initial` at the start of the timing, then the value `final` at the end, and subtract.

```
unsigned int initial, final, elapsed;
initial = _CP0_GET_COUNT();       // read the initial time
   ...                            // some code you want to time
final = _CP0_GET_COUNT();         // the end duration
elapsed = final - initial;        // total elapsed time, in ticks
```

The above code works even if `final` is less than `initial` due to a single timer rollover.[3] If the timer rolls over twice the answer will be incorrect, but your code will have taken longer than 107 s.

5.2.3 Disassembling Your Code

By looking at the assembly code the compiler produces, you can determine approximately how long your code takes to execute. Fewer instructions mean faster code.

In Chapter 3.5, we claimed that the code

```
LATFINV = 0x3;
```

is more efficient than

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

[3] Unsigned arithmetic actually computes $(a \odot b) \bmod 2^N$, where N is the number of bits in the type and $\odot$ represents an operator such as $+$ or $-$. Thus, unsigned subtraction computes the distance between the two numbers, modulo 2^N.

Let us examine that claim by looking at the assembly code of the following program. This program delays by executing a `for` loop 50 million times, then toggles RF1 (LED2 on the NU32).

Code Sample 5.1 `timing.c`**. RF1 Toggles (LED2 on the NU32 Flashes).**

```
#include "NU32.h"          // constants, functions for startup and UART
#define DELAYTIME 50000000 // 50 million

void delay(void);
void toggleLight(void);

int main(void) {
  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init

  while(1) {
    delay();
    toggleLight();
  }
}

void delay(void) {
  int i;
  for (i = 0; i < DELAYTIME; i++) {
      ; //do nothing
  }
}

void toggleLight(void) {
  LATFINV = 0x2; // invert LED2 (which is on port F1)
  // LATFbits.LATF1 = !LATFbits.LATF1;
}
```

Put `timing.c` in a project directory together with the `Makefile`, `NU32.c`, `NU32.h`, and `NU32bootloaded.ld`, and nothing else. Then `make` with no optimization, as described above. The `Makefile` automatically disassembles the `out.elf` file to the file `out.dis`, but if you wanted to do it manually, you could type

```
> xc32-objdump -S out.elf > out.dis
```

Open `out.dis` in a text editor. You will see a listing showing the assembly code corresponding to `out.hex`. The file interleaves your C code and the assembly code it generated.[4] Each assembly line has the actual virtual address where the assembly instruction is placed in memory, the 32-bit machine instruction, and the equivalent human-readable (if you know assembly!) assembly code. Let us look at the segment of the listing corresponding to the command `LATFINV = 0x2`. You should see something like

[4] Sometimes the output from the `xc32-objdump` duplicates part of the C code, due to the way in which the C gets translated into assembly.

```
  LATFINV = 0x2; // invert the LED2 (which is on port F1)
9d00221c: 3c02bf88  lui v0,0xbf88
9d002220: 24030002  li v1,2
9d002224: ac43616c  sw v1,24940(v0)
  // LATFbits.LATF1 = !LATFbits.LATF1;
```

We see that the `LATFINV = 0x2` instruction has expanded to three assembly statements. Without going into detail, the `li` instruction stores the base-10 value 2 (or hex 0x2) in the CPU register `v1`, which is then written by the `sw` command to the memory address corresponding to LATFINV (`v0`, or 0xBF88, is bits 16-31 of the address, and the base-10 value 24940, or hex 0x616C, is bits 0-15).[5]

If instead we comment out the `LATFINV = 0x2`; command and replace it with the bit manipulation version, we get the following disassembly:

```
  // LATFINV = 0x2; // invert the LED2 (which is on port F1)
  LATFbits.LATF1 = !LATFbits.LATF1;
9d00221c: 3c02bf88  lui v0,0xbf88
9d002220: 8c426160  lw v0,24928(v0)
9d002224: 30420002  andi v0,v0,0x2
9d002228: 2c420001  sltiu v0,v0,1
9d00222c: 304400ff  andi a0,v0,0xff
9d002230: 3c03bf88  lui v1,0xbf88
9d002234: 90626160  lbu v0,24928(v1)
9d002238: 7c820844  ins v0,a0,0x1, 0x1
9d00223c: a0626160  sb v0,24928(v1)
```

The bit manipulation version requires nine assembly statements. Basically the value of LATF is being copied to a CPU register, manipulated, then stored back in LATF. In contrast, with the LATFINV syntax, there is no copying the values of LATFINV back and forth.

Although one method of manipulating the SFR bit appears three times slower than the other, we do not yet know how many CPU cycles each consumes. Assembly instructions are generally performed in a single clock cycle, but there is still the question of whether the CPU is getting one instruction per cycle (due to the slow program flash.) We will investigate further by manipulating the prefetch cache module in Section 5.2.4. For now, though, we time the 50 million iteration delay loop. Here is the disassembly for `delay()`, with comments added to the right:

```
void delay(void) {
9d0021c0: 27bdfff0  addiu sp,sp,-16    // manipulate the stack pointer on ...
9d0021c4: afbe000c  sw s8,12(sp)       // ... entering the function (see text)
9d0021c8: 03a0f021  move s8,sp
  int i;
  for (i = 0; i < DELAYTIME; i++) {
9d0021cc: afc00000  sw zero,0(s8)      // initialization of i in RAM to 0
```

[5] You can refer to the MIPS32 documentation if interested.

```
9d0021d0: 0b400879  j 9d0021e4          // jump to 9d0021e4 (skip adding 1 to i),
9d0021d4: 00000000  nop                 // but "no operation" executed first
9d0021d8: 8fc20000  lw v0,0(s8)         // start of loop; load RAM i into
                                           register v0
9d0021dc: 24420001  addiu v0,v0,1       // add 1 to v0 ...
9d0021e0: afc20000  sw v0,0(s8)         // ... and store it to i in RAM
9d0021e4: 8fc30000  lw v1,0(s8)         // load i into register v1
9d0021e8: 3c0202fa  lui v0,0x2fa        // load the upper 16 bits and ...
9d0021ec: 3442f080  ori v0,v0,0xf080    // ... lower 16 bits of 50,000,000
                                           into v0
9d0021f0: 0062102a  slt v0,v1,v0        // store "true" (1) in v0 if v1 < v0
9d0021f4: 1440fff8  bnez v0,9d0021d8    // if v0 not equal to 0, branch to top of
                                           loop,
9d0021f8: 00000000  nop                 // but branch "delay slot" is executed
                                           first
      ; //do nothing
  }
}
9d0021fc: 03c0e821  move sp,s8          // manipulate the stack pointer on exiting
9d002200: 8fbe000c  lw s8,12(sp)
9d002204: 27bd0010  addiu sp,sp,16
9d002208: 03e00008  jr ra               // jump to return address ra stored by jal,
9d00220c: 00000000  nop                 // but jump delay slot is executed first
```

There are nine instructions in the delay loop itself, starting with `lw v0,0(s8)` and ending with the next `nop`. When the LED turns on, these instructions are carried out 50 million times, and then the LED turns off. (There are additional instructions to set up the loop and increment the counter, but the duration of these is negligible compared to the 50 million executions of the loop.) So if one instruction is executed per cycle, we would predict the light to stay on for approximately 50 million $\times$ 9 instructions $\times$ 12.5 ns/instruction $=$ 5.625 s. When we time by a stopwatch, we get about 6.25 s, which implies ten CPU (SYSCLK) cycles per loop. So our cache module has the CPU executing one assembly instruction almost every cycle.

In the code above there are two "jumps" (`j` for "jump" to the specified address and `jr` for "jump register" to jump to the address in the return address register `ra`, which was set by the calling function) and one "branch" (`bnez` for "branch if not equal to zero"). For MIPS32, the command after a jump or branch is executed before the jump actually occurs. This next command is said to be in the "delay slot" for the jump or branch. In all three delay slots in this code is a `nop` command, which stands for "no operation."

You might notice a few ways you could have written the assembly code for the delay function to use fewer assembly commands. Certainly one of the advantages of coding directly in assembly is that you have direct control of the processor instructions. The disadvantage, of course, is that MIPS32 assembly is a much lower-level language than C, requiring significantly more knowledge of MIPS32 from the programmer. Until you have already invested a great deal of time learning the assembly language, programming in assembly fails the "programmer-time-efficient" criterion! (Not to mention that `delay()` was designed to waste time, so no need to minimize assembly lines!)

You may have also noticed, in the disassembly of `delay()`, the manipulation of the *stack pointer* (`sp`) upon entering and exiting the function. The *stack* is an area of RAM that holds temporary local variables and parameters. When a function is called, its parameters and local variables are "pushed" onto the stack. When the function exits, the local variables are "popped" off of the stack by moving the stack pointer back to its original position before the function was called. A *stack overflow* occurs if there is not enough RAM available for the stack to hold all the local variables defined in currently-called functions. We will see the stack again in Section 5.3.

The overhead due to passing parameters and manipulating the stack pointer upon entering and exiting a function should not discourage you from writing modular code. Function call overhead should only concern you when you need to squeeze a final few nanoseconds out of your program execution time.

Finally, if you revert back to the `LATFINV` method for toggling the LED and compile `timing.c` with optimization level 1 (the optimization flag `-O1`), you see that `delay()` is optimized to

```
void delay(void) {
9d00211c: 3c0202fa  lui v0,0x2fa      // load the upper 16 bits and ...
9d002120: 3442f080  ori v0,v0,0xf080  // ... lower 16 bits of 50,000,000 into v0
9d002124: 2442ffff  addiu v0,v0,-1    // subtract 1 from v0
  int i;
  for (i = 0; i < DELAYTIME; i++) {
9d002128: 1440ffff  bnez v0,9d002128  // if v0 !=0, branch back to the same line,
9d00212c: 2442ffff  addiu v0,v0,-1    // but before branch, subtract 1 from v0
     ; //do nothing
  }
}
9d002130: 03e00008  jr ra             // jump to return address ra stored by jal
9d002134: 00000000  nop               // no operation in jump delay slot
```

No local variables are stored in RAM, and there is no stack pointer manipulation upon entering and exiting the function. The counter variable is simply stored in a CPU register. The loop itself has only two lines instead of nine, and it has been designed to count down from 49,999,999 to zero instead of counting up. The branch delay slot is actually used to implement the counter update instead of having a wasted `nop` cycle.

More importantly, however, `delay()` is never called by the assembly code for `main` in our `-O1` optimized code! The compiler has recognized that `delay()` does not do anything.[6] As a result, the LED toggles so quickly that you cannot see it by eye. The LED just looks dim.[7]

[6] A better optimization would not have produced code for `delay` at all, reducing flash usage.

[7] To prevent `delay()` from being optimized away, we could have added a "no operation" `_nop();` command inside the delay loop. Or we could have accessed a `volatile` variable inside the loop. Or we could have polled the core timer to implement a desired delay.

5.2.4 The Prefetch Cache Module

In the previous section, we saw that our `timing.c` program executed approximately one assembly instruction every clock cycle. We achieved this performance because the bootloader (and `NU32_Startup()`) enabled the prefetch cache module and selected the minimum number of CPU wait cycles for instructions loading from flash.[8]

We can disable the prefetch cache module to observe its effect on the program `timing.c`. The prefetch cache module performs two primary tasks: (1) it keeps recent instructions in the cache, ready if the CPU requests the instruction at that address again (allowing the cache to completely store small loops); and, (2) for linear code, it retrieves instructions ahead of the current execution location, so they are ready when needed (prefetch). We can disable each of these functions separately, or we can disable both.

Let us start by disabling both. Modify `timing.c` in Code Sample 5.1 by adding

```
// Turn off function (1), storing recent instructions in cache
__builtin_mtc0(_CP0_CONFIG, _CP0_CONFIG_SELECT, 0xa4210582);
CHECONCLR = 0x30;  // Turn off function (2), prefetch
```

right after `NU32_Startup()` in `main`. Everything else stays the same. The first line modifies a CPU register, preventing the prefetch cache module from storing recent instructions in cache. As for the second line, consulting the section on the prefetch cache module in the Reference Manual, we see that bits 4 and 5 of the SFR CHECON determine whether instructions are prefetched, and that clearing both bits disables predictive prefetch.

Recompiling `timing.c` with no compiler optimizations and rerunning, we find that the LED stays on for approximately 17 s, compared to approximately 6.25 s before. This corresponds to 27 SYSCLK cycles per delay loop, which we saw earlier has nine assembly commands. These numbers make sense—since the prefetch cache is completely disabled, it takes three CPU cycles (one request cycle plus two wait cycles) for each instruction to get from flash to the CPU.

If we comment out the second line, so that (1) the cache of recent instructions is off but (2) the prefetch is enabled, and rerun, we find that the LED stays on for about 8.1 s, or 13 SYSCLK cycles per loop, a small penalty compared to our original performance of 10 cycles. The prefetch is able to run ahead to grab future instructions, but it cannot run past the `for` loop conditional statement, since it does not know the outcome of the test.

[8] The number of "wait cycles" is the number of extra cycles the CPU must wait for instructions to finish loading from flash if they are not cached. Since the PIC32's flash operates at a maximum of 30 MHz and the CPU operates at 80 MHz, the number of wait cycles is configured as two in the bootloader and `NU32_Startup()`, to allow three total cycles for a flash instruction to load. Fewer wait cycles would result in errors and more wait cycles would slow performance unnecessarily.

Finally, if we comment out the first line but leave the second line uncommented, so that (1) the cache of recent instructions is on but (2) the prefetch is disabled, we recover our original performance of approximately 6.25 s or 10 SYSCLK cycles per loop. The reason is that the entire loop is stored in the cache, so prefetch is not necessary.

5.2.5 Math

For real-time systems, it is often critical to perform mathematical operations as quickly as possible. Mathematical expressions should be coded to minimize execution time. We will delve into the speed of various math operations in the Exercises, but here are a few rules of thumb for efficient math:

- There is no floating point unit on the PIC32MX, so all floating point math is carried out in software. Integer math is much faster than floating point math. If speed is an issue, perform all math as integer math, scaling the variables as necessary to maintain precision, and only convert to floating point when needed.
- Floating point division is slower than multiplication. If you will be dividing by a fixed value many times, consider taking the reciprocal of the value once and then using multiplication thereafter.
- Functions such as trigonometric functions, logarithms, square roots, etc. in the math library are generally slower to evaluate than arithmetic functions. Their use should be minimized when speed is an issue.
- Partial results should be stored in variables for future use to avoid performing the same computation multiple times.

5.3 Space and the Map File

The previous section focused on execution time. We now examine how much program memory (flash) and data memory (RAM) our programs use.

The linker allocates virtual addresses in program flash for all program instructions, and virtual addresses in data RAM for all global variables. The rest of RAM is allocated to the *heap* and the *stack*.

The heap is memory set aside to hold dynamically allocated memory, as allocated by `malloc` and `calloc`. These functions allow you to, for example, create an array whose length is determined at runtime, rather than specifying a (possibly space-wasteful) fixed-sized array in advance.

The stack holds temporary local variables used by functions. When a function is called, space on the stack is allocated for its local variables. When the function exits, the local variables are discarded and the space is made available again by moving the stack pointer. The stack grows "down" from the end of RAM—as local variables are "pushed" onto the stack, the stack pointer address decreases, and when local variables are "popped" off the stack after exiting a

function, the stack pointer address increases. (See the assembly listing for `delay()` in `timing.c` in Section 5.2.3 for an example of moving the stack pointer when a function is called and when it exits.)

If your program attempts to put too many local variables on the stack (stack overflow), the error will not appear until run time. The linker does not catch this error because it does not explicitly set aside space for temporary local variables; it assumes they will be handled by the stack.

To further examine how memory is allocated, we can ask the linker to create a "map" file when it creates the `.elf` file. The map file indicates where instructions are placed in program memory and where global variables are placed in data memory. Your `Makefile` automatically creates an `out.map` file for you by including the `-Map` option to the linker command:

```
> xc32-gcc [details omitted] -Wl,--script="NU32bootloaded.ld",-Map="out.map"
```

The map file can be opened with a text editor.

Let us examine the `out.map` file for `timing.c` as shown in Code Sample 5.1, and compiled with no optimizations. Here's an edited portion of this rather large file:

```
Microchip PIC32 Memory-Usage Report

kseg0 Program-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.text                    0x9d001e00          0x2b4         692  App's exec code
.text.general_exception  0x9d0020b4           0xdc         220
.text                    0x9d002190           0xac         172  App's exec code
.text.main_entry         0x9d00223c           0x54          84
.text._bootstrap_except  0x9d002290           0x48          72
.text._general_exceptio  0x9d0022d8           0x48          72
.vector_default          0x9d002320           0x48          72
.text                    0x9d002368           0x5c          92  App's exec code
.dinit                   0x9d0023c4           0x10          16
.text._on_reset          0x9d0023d4           0x8           8
.text._on_bootstrap      0x9d0023dc           0x8           8
      Total kseg0_program_mem used  :         0x5e4         1508  0.3% of 0x7e200

kseg0 Boot-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
         Total kseg0_boot_mem used  :            0            0  <1% of 0x970

Exception-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.app_excpt               0x9d000180           0x10          16  General-Exception
.vector_0                0x9d000200            0x8           8  Interrupt Vector 0
.vector_1                0x9d000220            0x8           8  Interrupt Vector 1
```

```
  [[[ ... omitting long list of vectors ...]]]

.vector_51              0x9d000860            0x8           8  Interrupt Vector 51
          Total exception_mem used  :       0x1b0         432  10.5% of 0x1000

kseg1 Boot-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.reset                   0xbd001970          0x1f4         500  Reset handler
.bev_excpt               0xbd001cf0           0x10          16  BEV-Exception
         Total kseg1_boot_mem used  :       0x204         516  44.2% of 0x490
        ------------------------------------------------------------------------
         Total Program Memory used  :       0x998        2456  0.5% of 0x80000
        ------------------------------------------------------------------------
```

The kseg0 program memory usage report tells us that 1508 (or `0x5e4`) bytes are used for the main part of our program. The first entry is denoted `.text`, and holds program instructions. It is the largest single section, using 692 bytes, described as `App's exec code`, and installed starting at VA 0x9d001e00. Searching for this address in the map file, we see that this is the code for `NU32.o`, the object code associated with the NU32 library.

Subsequent sections of kseg0 program memory (some also denoted as `.text`), are packed tightly and in order of decreasing section size. The next section is `.text.general_exception`, which corresponds to a routine that is called when the CPU encounters certain types of "exceptions" (run-time errors). This code was linked from `pic32mx/lib/libpic32.a`. The next `.text` section, also labeled `App's exec code`, is the object code `timing.o` and is 172 (or `0xac`) bytes long. Searching for `timing.o` we find the following text:

```
.text           0x9d002190       0xac
 .text          0x9d002190       0xac timing.o
                0x9d002190                main
                0x9d0021c0                delay
                0x9d002210                toggleLight
```

Our functions `main`, `delay`, and `toggleLight` of `timing.o` are stored consecutively in memory. The addresses agree with our disassembly file from Section 5.2.3.

Continuing, the kseg0 boot memory report indicates that no code is placed in this memory region. The exception memory report indicates that placeholders for instructions corresponding to interrupts occupy 432 bytes. Finally, the kseg1 boot memory report indicates that the C runtime startup code installed reset functions that occupy 516 bytes. The address of the `.reset` section is the address that the bootloader (already installed in the 12 KB boot flash) jumps to.

In all, 2456 bytes of the 512 KB of program memory are used.

Continuing further in the map file, we see

```
kseg1 Data-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
         Total kseg1_data_mem used  :           0           0  <1% of 0x20000
        ------------------------------------------------------------------------
            Total Data Memory used  :           0           0  <1% of 0x20000
        ------------------------------------------------------------------------


Dynamic Data-Memory Reservation
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
heap                    0xa0000008              0           0  Reserved for heap
stack                   0xa0000020        0x1ffd8      131032  Reserved for stack
```

There are no global variables, so no kseg1 data memory is used. The heap size is zero, so essentially all data memory is reserved for the stack.

Now let us modify our program by adding some useless global variables, just to see what happens to the map file. Let us add the following lines just before `main`:

```
char my_cat_string[] = "2 cats!";
int my_int = 1;
char my_message_string[] = "Here's a long message stored in a character array.";
char my_small_string[6], my_big_string[97];
```

Rebuilding and examining the new map file, we see the following for the data memory report:

```
kseg1 Data-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.sdata                  0xa0000000             0xc          12  Small init data
.sbss                   0xa000000c             0x6           6  Small uninit data
.bss                    0xa0000014            0x64         100  Uninitialized data
.data                   0xa0000078            0x34          52  Initialized data
         Total kseg1_data_mem used  :          0xaa         170  0.1% of 0x20000
        ------------------------------------------------------------------------
            Total Data Memory used  :          0xaa         170  0.1% of 0x20000
        ------------------------------------------------------------------------
```

Our global variables now occupy 170 bytes of data RAM. The global variables have been placed in four different data memory sections, depending on whether the variable is small or large (according to a command line option or `xc32-gcc` default) and whether it is initialized:

section name	**data type**	**variables stored there**
`.sdata`	small initialized data	`my_cat_string`, `my_int`
`.sbss`	small uninitialized data	`my_small_string`
`.bss`	larger uninitialized data	`my_big_string`
`.data`	larger initialized data	`my_message_string`

Searching for the .sdata section further in the map file, we see

```
.sdata          0xa0000000        0xc timing.o
                0xa0000000                my_cat_string
                0xa0000008                my_int
                0xa000000c                _sdata_end = .
```

Even though the string my_cat_string uses only seven bytes, the variable my_int starts eight bytes after the start of my_cat_string. This gap occurs because variables are aligned on four-byte boundaries, meaning that their addresses are evenly divisible by four. Similarly, the strings my_message_string, my_small_string, and my_big_string occupy memory to the next four-byte boundary. Due to data alignment, a five-byte string uses the same amount of memory as an eight-byte string.

Apart from the addition of these sections to the data memory usage report, we see that the global variables reduce the data memory available for the stack, and the .dinit (global data initialization, from the C runtime startup code) section of the kseg0 program memory report has grown to 112 bytes, meaning that our total kseg0 program memory used is now 1604 bytes instead of 1508.

Now let us change the definition of my_cat_string to put the qualifier const in front of it, becoming

```
const char my_cat_string[] = "2 cats!";
```

This makes the array a constant; my_cat_string cannot be changed later in the program. Global variables declared with the const qualifier are placed in flash rather than RAM (this behavior is XC32 specific). Building again and examining the map file, we see the following changes in kseg0 program memory usage

```
kseg0 Program-Memory Usage
section                   address  length [bytes]      (dec)  Description
-------                   ----------  --------------------------  -----------
  [[[ .dinit has shrunk by 16 bytes; no initialization code for my_cat_string ]]]
.dinit                  0x9d00223c           0x60          96
  [[[ a new eight-byte section, .rodata, has been added to hold my_cat_string ]]]
.rodata                 0x9d002424            0x8           8  Read-only const
```

and the following change in kseg1 data memory usage

```
kseg1 Data-Memory Usage
section                   address  length [bytes]      (dec)  Description
-------                   ----------  --------------------------  -----------
  [[[ .sdata has shrunk since RAM no longer holds my_cat_string ]]]
.sdata                  0xa0000000            0x4           4  Small init data
```

The new section in kseg0 program memory, `.rodata` (read-only data), contains eight bytes to hold `my_cat_string`. This constant character array is stored in flash memory at the address 0x9d002424. Correspondingly, the `.sdata` section in RAM (kseg1 data memory) has dropped by eight bytes, since the eight-byte `my_cat_string` is no longer a variable that has to be stored in RAM. Finally, the `.dinit` section in flash program memory has shrunk by 16 bytes, since we no longer need assembly code to initialize `my_cat_string`.

One last change. Let us move the definition

```
const char my_cat_string[] = "2 cats!";
```

inside the `main` function, so that `my_cat_string` is now local to `main`. Building the program again, we find only one change: one `.text` section in the kseg0 program memory report has grown by 24 bytes.

```
kseg0 Program-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  -------------------------  -----------
  [[[ this .text section has grown by 24 bytes ]]]
.text                    0x9d002190           0xc4         196  App's exec code
```

Examining the disassembly file, we see that six lines of assembly code were added in `main` that copy the string to RAM. Since this is a local variable, the local copy uses the stack, and therefore there is no memory allocated for it in the kseg1 data memory report.

Finally, we might wish to reserve some RAM for a heap for dynamic memory allocation using `malloc` or `calloc`. By default, the heap size is set to zero. To set a nonzero heap size, we can pass a linker option to `xc32-gcc`:

```
xc32-gcc [details omitted] -Wl,--script="NU32bootloaded.ld",-Map="out.map",--defsym=_
min_heap_size=4096
```

This defines a heap of 4 KB. After building, the map file shows

```
Dynamic Data-Memory Reservation
section                    address  length [bytes]      (dec)  Description
-------                  ----------  -------------------------  -----------
heap                     0xa00000a8         0x1000        4096  Reserved for heap
stack                    0xa00010c0        0x1ef30      126768  Reserved for stack
```

The heap is allocated at low RAM addresses, close after the global variables, starting in this case at address 0xa00000a8. The stack occupies most of the rest of RAM.

For most embedded applications, there is no need to use a heap.

5.4 Chapter Summary

- The CPU's core timer increments once every two ticks of the SYSCLK, or every 25 ns for an 80 MHz SYSCLK. The commands `_CP0_SET_COUNT(0)` and `unsigned int dt = _CP0_GET_COUNT()` can be used to measure the execution time of the code in between to within a few SYSCLK cycles.
- To generate a disassembly listing at the command line, use `xc32-objdump -S filename.elf > filename.dis`.
- With the prefetch cache module fully enabled, your PIC32 should be able to execute an assembly instruction nearly every cycle. The prefetch allows instructions to be fetched in advance for linear code, but the prefetch cannot run past conditional statements. For small loops, the entire loop can be stored in the cache.
- The linker assigns specific program flash VAs to all program instructions and data RAM VAs to all global variables. The rest of RAM is allocated to the heap, for dynamic memory allocation, and to the stack, for function parameters and temporary local variables. The heap is zero bytes by default.
- A map file provides a detailed summary of memory usage. To generate a map file at the command line, use the `-Map` option to the linker, e.g.,

```
xc32-gcc [details omitted] -Wl,-Map="out.map"
```

- Global variables can be initialized (assigned a value when they are defined) or uninitialized. Initialized global variables are stored in RAM memory sections `.data` and `.sdata` and uninitialized globals are stored in RAM memory sections `.bss` and `.sbss`. Sections beginning with `.s` mean that the variables are "small." When the program is executed, initialized global variables are assigned their values by C runtime startup code, and uninitialized global variables are set to zero.
- Global variables are packed tightly at the beginning of data RAM, 0xA0000000. The heap comes immediately after. The stack begins at the high end of RAM and grows "down" toward lower RAM addresses. Stack overflow occurs if the stack pointer attempts to move into an area reserved for the heap or global variables.

5.5 Exercises

Unless otherwise specified, compile with no optimizations for all problems.

1. Describe two examples of how you can write code differently to either make it execute faster or use less program memory.
2. Compile and run `timing.c`, Code Sample 5.2, with no optimizations (`make CFLAGS="-g -x c"`). With a stopwatch, verify the time taken by the delay loop. Do your results agree with Section 5.2.3?

3. To write time-efficient code, it is important to understand that some mathematical operations are faster than others. We will look at the disassembly of code that performs simple arithmetic operations on different data types. Create a program with the following local variables in `main`:

```
char c1=5, c2=6, c3;
int i1=5, i2=6, i3;
long long int j1=5, j2=6, j3;
float f1=1.01, f2=2.02, f3;
long double d1=1.01, d2=2.02, d3;
```

   Now write code that performs add, subtract, multiply, and divide for each of the five data types, i.e., for `char`s:

```
c3 = c1+c2;
c3 = c1-c2;
c3 = c1*c2;
c3 = c1/c2;
```

   Build the program with no optimization and look at the disassembly. For each of the statements, you will notice that some of the assembly code involves simply loading the variables from RAM into CPU registers and storing the result (also in a register) back to RAM. Also, while some of the statements are completed by a few assembly commands in sequence, others result in a jump to a software subroutine to complete the calculation. (These subroutines are provided with our C installation and included in the linking process.) Answer the following questions.

   a. Which combinations of data types and arithmetic functions result in a jump to a subroutine? From your disassembly file, copy the C statement and the assembly commands it expands to (including the jump) for one example.
   b. For those statements that do not result in a jump to a subroutine, which combination(s) of data types and arithmetic functions result in the fewest assembly commands? From your disassembly, copy the C statement and its assembly commands for one of these examples. Is the smallest data type, `char`, involved in it? If not, what is the purpose of extra assembly command(s) for the `char` data type vs. the `int` data type? (Hint: the assembly command `ANDI` takes the bitwise AND of the second argument with the third argument, a constant, and stores the result in the first argument. Or you may wish to look up a MIPS32 assembly instruction reference.)
   c. Fill in the following table. Each cell should have two numbers: the number of assembly commands for the specified operation and data type, and the ratio of this number (greater than or equal to 1.0) to the smallest number of assembly commands in the table. For example, if addition of two `int`s takes four assembly commands, and this is the fewest in the table, then the entry in that cell would be 1.0 (4). This has been filled in below, but you should change it if you get a different result. If a statement results in a jump to a subroutine, write J in that cell.

	char	int	long long	float	long double
+ − * /		1.0 (4)			

d. From the disassembly, find out the name of any math subroutine that has been added to your assembly code. Now create a map file of the program. Where are the math subroutines installed in virtual memory? Approximately how much program memory is used by each of the subroutines? You can use evidence from the disassembly file and/or the map file. (Hint: You can search backward from the end of your map file for the name of any math subroutines.)

4. Let us look at the assembly code for bit manipulation. Create a program with the following local variables:

```
unsigned int u1=33, u2=17, u3;
```

and look at the assembly commands for the following statements:

```
u3 = u1 & u2;     // bitwise AND
u3 = u1 | u2;     // bitwise OR
u3 = u2 << 4;     // shift left 4 spaces, or multiply by 2^4 = 16
u3 = u1 >> 3;     // shift right 3 spaces, or divide by 2^3 = 8
```

How many commands does each use? For unsigned integers, bit-shifting left and right make for computationally efficient multiplies and divides, respectively, by powers of 2.

5. Use the core timer to calculate a table similar to that in Exercise 3, except with entries corresponding to the actual execution time in terms of SYSCLK cycles. So if a calculation takes 15 cycles, and the fastest calculation is 10 cycles, the entry would be 1.5 (15). This table should contain all 20 entries, even for those that jump to subroutines. (Note: subroutines often have conditional statements, meaning that the calculation could terminate faster for some operands than for others. You can report the results for the variable values given in Exercise 3.)
To minimize uncertainty due to the setup and reading time of the core timer, and the fact that the timer only increments once every two SYSCLK cycles, each math statement could be repeated ten or more times (no loops) between setting the timer to zero and reading the timer. The average number of cycles, rounded down, should be the number of cycles for each statement. Use the NU32 communication routines, or any other communication routines, to report the answers back to your computer.
6. Certain math library functions can take quite a bit longer to execute than simple arithmetic functions. Examples include trigonometric functions, logarithms, square roots, etc. Make a program with the following local variables:

```
float f1=2.07, f2;        // four bytes for each float
long double d1=2.07, d2;  // eight bytes for each long double
```

Also be sure to put `#include <math.h>` at the top of your program to make the math function prototypes available.

a. Using methods similar to those in Exercise 5, measure how long it takes to perform each of `f2 = cosf(f1)`, `f2 = sqrtf(f1)`, `d2 = cos(d1)`, and `d2 = sqrt(d1)`.
b. Copy and paste the disassembly from a `f2 = cosf(f1)` statement and a `d2 = cos(d1)` statement into your solution set and compare them. Based on the comparison of the assembly codes, comment on the advantages and disadvantages of using the eight-byte `long double` floating point representation compared to the four-byte `float` representation when you compute a cosine with the PIC32 compiler.
c. Make a map file for this program, and search for the references to the math library `libm.a` in the map file. There are several `libm.a` files in your C installation, but which one was used by the linker when you built your program? Give the directory.

7. Explain what stack overflow is, and give a short code snippet (not a full program) that would result in stack overflow on your PIC32.
8. In the map file of the original `timing.c` program, there are several `App's exec code`, one corresponding to `timing.o`. Explain briefly what each of the others are for. Provide evidence for your answer from the map file.
9. Create a map file for `simplePIC.c` from Chapter 3. (a) How many bytes does `simplePIC.o` use? (b) Where are the functions `main` and `delay` placed in virtual memory? Are instructions at these locations cacheable? (c) Search the map file for the `.reset` section. Where is it in virtual memory? Is it consistent with your `NU32bootloaded.ld` linker file? (d) Now augment the program by defining `short int`, `long int`, `long long int`, `float`, `double`, and `long double` global variables. Provide evidence from the map file indicating how much memory each data type uses.
10. Assume your program defines a global `int` array `int glob[5000]`. Now what is the maximum size of an array of `int`s that you can define as a local variable?
11. Provide global variable definitions (not an entire program) so that the map file has data sections `.sdata` of 16 bytes, `.sbss` of 24 bytes, `.data` of 0 bytes, and `.bss` of 200 bytes.
12. If you define a global variable and you want to set its initial value, is it "better" to initialize it when the variable is defined or to initialize it in a function? Explain any pros and cons.
13. The program `readVA.c` (Code Sample 5.2) prints out the contents of the 4-byte word at any virtual address, provided the address is word-aligned (i.e., evenly divisible by four). This allows you to inspect anything in the virtual memory map (Chapter 3), including the representations of variables in data RAM, program instructions in flash or boot flash, SFRs, and configuration bits. You can use code like this in conjunction with the map and disassembly files to better understand the code produced by a build. Here's a sample of the program's output to the terminal:

```
Enter the start VA (e.g., bd001970) and # of 32-bit words to print.
Listing 4 32-bit words starting from address bd001970.
 ADDRESS  CONTENTS
bd001970  0f40065e
bd001974  00000000
bd001978  401a6000
bd00197c  7f5a04c0
```

a. Build and run the program. Check its operation by consulting your disassembly file and confirming that 32-bit program instructions (in flash) listed there match the output of the program. Confirm that you get the same results whether you reference the same physical memory address using a cacheable or noncacheable virtual address. What happens if you specify a virtual address that is not divisible by four?
b. Examine the map file. At what virtual address in RAM is the variable `val` stored? Confirm its value using the program.
c. The values of the configuration bits (the four words DEVCFG0 to DEVCFG3, see Chapter 2.1.4) were set by the preinstalled bootloader. Use the program to print the values of these device configuration registers.
d. Consulting the map or disassembly file, what is the address of the last instruction in the program? Use the program to provide a listing of the instructions from a few addresses before to a few addresses after the last instruction. Addresses that do not have an instruction were erased when the program was loaded by the bootloader, but no instructions were written there. Knowing that, and by looking at your program's output, what value does an erased flash byte have?
e. Modify the program so the `unsigned int`s are defined as local to `main`, so that they are on the stack. Since `val` is no longer given a specific address by the linker, you do not find it in the map file. Use your program to find the address in RAM where `val` is stored.

Code Sample 5.2 `readVA.c`. Code to Inspect the Virtual Memory Map.

```
#include "NU32.h"
#define MSGLEN 100

// val is an initialized global; you can find it in the memory map with this program
char msg[MSGLEN];
unsigned int *addr;
unsigned int k = 0, nwords = 0, val = 0xf01dab1e;

int main(void) {

  NU32_Startup();
  while (1) {
    sprintf(msg, "Enter the start VA (e.g., bd001970) and # of 32-bit words to print: ");
    NU32_WriteUART3(msg);

    NU32_ReadUART3(msg,MSGLEN);
    sscanf(msg,"%x %d",&addr, &nwords);
```

```
    sprintf(msg,"\r\nListing %d 32-bit words starting from VA %08x.\r\n",nwords,addr);
    NU32_WriteUART3(msg);

    sprintf(msg," ADDRESS  CONTENTS\r\n");
    NU32_WriteUART3(msg);

    for (k = 0; k < nwords; k++) {
      sprintf(msg,"%08x  %08x\r\n", addr,*addr); // *addr is the 32 bits starting at addr
      NU32_WriteUART3(msg);
      ++addr;                  // addr is an unsigned int ptr so it increments by 4 bytes
    }
  }
  return 0;
}

// handle cpu exceptions, such as trying to read from a bad memory location
void _general_exception_handler(unsigned cause, unsigned status)
{
  unsigned int exccode = (cause & 0x3C) >> 2; // the exccode is reason for the exception
  // note: see PIC32 Family Reference Manual Section 03 CPU M4K Core for details
  // Look for the Cause register and the Status Register
  NU32_WriteUART3("Reset the PIC32 due to general exception.\r\n");
  sprintf(msg,"cause 0x%08x (EXCCODE = 0x%02x), status 0x%08x\r\n",cause,exccode,status);
  NU32_WriteUART3(msg);
  while(1) {
    ;
  }
}
```

Further Reading

MIPS32 M4K processor core software user's manual (2.03 ed.). (2008). MIPS Technologies.
PIC32 family reference manual. Section 04: Prefectch cache module. (2011). Microchip Technology Inc.