Chapter 5

Time and Space

Of course it is a good idea to write "efficient" code. But "efficient" can mean a number of different things, such as time-efficient (runs fast), RAM-efficient (makes the most of limited RAM), flash-efficient (makes the most of limited flash), but perhaps most importantly, programmer-time-efficient (minimizes the time needed to write and debug the code, or for a future programmer to understand it). Often these interests are in competition with each other. In fact, the XC32 compiler provides a number of compilation options, some of which are not available in the free version of the compiler, that allow you to explicitly make space-time tradeoffs. As one 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 a little bit of execution time (no counter increments, no conditional tests, no branches) at the expense of using more flash to store the program.

The purpose of this chapter is to make you aware of some tools for understanding the time and space consumed by your program. These will help you squeeze the most out of your PIC32, allowing you to do more with a given PIC32 or to choose a cheaper PIC32.

5.1 Time and the Disassembly File

5.1.1 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 RA4 has been high for some time
LATACLR = 0x10;             // clear RA4 to 0 (turn on NU32 LED1)
    ...                     // some code you want to time
LATASET = 0x10;             // set RA4 to 1 (turn off LED1)
```

The time that RA4 is low (or the NU32's LED1 is on) is approximately the duration of the code you want to measure.

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

```
    ...                     // digital output RA4 has been high for some time
LATACLR = 0x10;             // clear RA4 to 0 (turn on NU32 LED1)
for (i=0; i<1000000; i++) { // but modify 1,000,000 to something appropriate for you
    ...                     // some code you want to time
}
LATASET = 0x10;             // set RA4 to 1 (turn off LED1)
```

Then you can divide the total time by 1,000,000. Keep in mind, however, that there is overhead to implement the `for` loop (incrementing a counter, checking the inequality, etc.). We will see this in Section 5.1.3. 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.

5.1.2 Timing Using the Core Timer

A more accurate time can be obtained using a timer onboard the PIC32. The NU32's 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 (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 $= 107$ seconds.

If your program includes `plib.h`, you can use statements such as the following:

```
unsigned int elapsedticks, elapsedns;

WriteCoreTimer(0);              // set the core timer counter to 0; in pic32-libs/peripheral/timer
   ...                          // some code you want to time
elapsedticks = ReadCoreTimer(); // read the core timer
elapsedns = elapsedticks * 25;  // for 80 MHz SYSCLK
```

Writing to and reading from the core timer takes a few processor cycles, and the timer only counts every 2 ticks of SYSCLK. To minimize the uncertainty introduced by these, you can execute the code several times (just copy and paste it) between the write and read of the core timer. Avoid the overhead of implementing a loop.

We can actually do a bit better. Since we are concerned about timing, let's reduce the overhead for writing to and reading from the core timer to the bare minimum. Looking up the source for the `WriteCoreTimer()` and `ReadCoreTimer()` functions in `pic32-libs/peripheral/timer/source`, we see that each is implemented by a single assembly command instruction to the CPU.

```
unsigned int elapsed, start=0;

asm volatile("mtc0   %0, $9": "+r"(start));     // WriteCoreTimer(0);
   ...                                          // some code you want to time
asm volatile("mfc0   %0, $9" : "=r"(elapsed));  // elapsed = ReadCoreTimer();
```

The `asm` command constructs a line of assembly code to be directly inserted by the compiler.

In the next section we look more systematically at the assembly code created by our C code. See also Problem 3.

5.1.3 Disassembling Your Code

A convenient way to examine the time efficiency of your code is to look at the assembly code produced by the compiler. The fewer instructions, the faster your code will execute.

In Chapter 3.5, we claimed that the code

```
LATAINV = 0x30;
```

is more efficient than

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

Let's examine that claim by looking at the assembly code of the following program. This program simply delays by executing a `for` loop 50 million times, then toggles RA5 (LED2 on the NU32).

Code Sample 5.1. `timing.c`. RA5 toggles (LED2 on the NU32 flashes).

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // plib.h, config bits, constants, funcs 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++) {}
}

void toggleLight(void) {
  LATAINV = 0x20;
  // LATAbits.LATA5 = !LATAbits.LATA5;
}
```

After building it in the IDE, go to **Window** > **Output** > **Disassembly Listing File**.[1] You will see a listing showing your C code and, below each C line, the assembly code it generated. Each assembly line has the actual virtual address where the assembly instruction was placed in memory, the 32-bit machine instruction, and the equivalent human-readable (if you know assembly!) assembly code. Let's look at the segment of the listing corresponding to the command `LATAINV = 0x20`. You should see something like

```
22:                   LATAINV = 0x20;
9D003370  3C02BF88   LUI V0, -16504
9D003374  24030020   ADDIU V1, ZERO, 32
9D003378  AC43602C   SW V1, 24620(V0)
23:                   //LATAbits.LATA5 = !LATAbits.LATA5;
```

We see that the `LATAINV = 0x20` command has expanded to three assembly statements. Without going into detail[2], the `ADDIU` stores the value 0x20 (the sum of ZERO and 32 in decimal) into the 32-bit CPU register V1, and the `SW` stores this word into the memory address corresponding to LATAINV.

If instead we comment out the `LATAINV = 0x20` command and replace it with the bit manipulation version, we get the following disassembly:

```
22:                   //LATAINV = 0x20;
23:                   LATAbits.LATA5 = !LATAbits.LATA5;
9D003370  3C02BF88   LUI V0, -16504
9D003374  8C426020   LW V0, 24608(V0)
9D003378  30420020   ANDI V0, V0, 32
9D00337C  2C420001   SLTIU V0, V0, 1
9D003380  304400FF   ANDI A0, V0, 255
9D003384  3C03BF88   LUI V1, -16504
9D003388  8C626020   LW V0, 24608(V1)
9D00338C  7C822944   INS V0, A0, 5, 1
9D003390  AC626020   SW V0, 24608(V1)
```

[1] At the command line, you can instead use `xc32-objdump -d -S filename.elf > filename.disasm` and inspect the file `filename.disasm`. The results may not be quite as easy to interpret, however.

[2] You can look up the MIPS32 assembly instruction set if you're interested.

The bit manipulation version requires nine assembly statements. Basically the value of LATA is being copied to a CPU register, manipulated, then stored back in LATA. With the LATAINV syntax, this is done in hardware.

Although one method of manipulating the SFR bit appears three times slower than the other, we don't yet know how many processor 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. (Recall the issue of slow program flash.) We will look at this further with the prefetch cache module in Section 5.1.4 below. For now, though, let's time that delay loop that is executed 50 million times. Here is the disassembly for `delay()`, with comments added to the right:

```
16:                 void delay(void) {
9D003314  27BDFFF0  ADDIU SP, SP, -16         // manipulate the stack pointer on ...
9D003318  AFBE000C  SW S8, 12(SP)             // ... entering the function (see text)
9D00331C  03A0F021  ADDU S8, SP, ZERO
17:                   int i;
18:                   for (i=0; i<DELAYTIME; i++) {}
9D003320  AFC00000  SW ZERO, 0(S8)            // initialization of i in RAM to 0
9D003324  0B400CCE  J 0x9D003338              // jump to 9D003338 (skip adding 1 to i)
9D003328  00000000  NOP                       // "no operation," let jump complete
9D00332C  8FC20000  LW V0, 0(S8)              // start of the loop; load RAM i into register V0
9D003330  24420001  ADDIU V0, V0, 1           // add 1 to V0 ...
9D003334  AFC20000  SW V0, 0(S8)              // ... and store it to i in RAM
9D003338  8FC30000  LW V1, 0(S8)              // load i into V1
9D00333C  3C0202FA  LUI V0, 762               // these two lines ...
9D003340  3442F080  ORI V0, V0, -3968         // ... load the constant 50,000,000 into V0
9D003344  0062102A  SLT V0, V1, V0            // store "true" (1) in V0 if V1 < V0
9D003348  1440FFF8  BNE V0, ZERO, 0x9D00332C  // if V0 does not equal 0, branch to top of loop
9D00334C  00000000  NOP                       // branch delay slot is executed before branch
19:                 }
9D003350  03C0E821  ADDU SP, S8, ZERO         // manipulate the stack pointer on exiting
9D003354  8FBE000C  LW S8, 12(SP)
9D003358  27BD0010  ADDIU SP, SP, 16
9D00335C  03E00008  JR RA                     // jump to return address RA stored by JAL
9D003360  00000000  NOP
```

There are nine instructions in the delay loop itself, starting with `LW V0, 0(S8)` and ending with the `NOP`. When the LED comes on, these instructions are carried out 50 million times, and then the LED turns off. (There are a few other instructions to set up the loop, but these are 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 50 million $\times$ 9 instructions $\times$ 12.5 ns/instruction = 5.625 seconds. When we timed by a stopwatch, we got about 6.25 seconds, which implies 10 cycles per loop. So our cache module has the CPU executing one assembly instruction almost every cycle.

You might notice a couple of ways you could have written the assembly code for the delay function more time-efficiently. This is certainly one of the advantages of coding directly in assembly: 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! Also, more efficient assembly code can be generated from your C code by employing certain compiler optimizations (Section 5.1.5). (Not to mention that `delay` was designed to waste time, so no need to be efficient!)

Another thing you may have noticed in the disassembly of `delay()` is the manipulation of the *stack pointer* (`SP`) upon entering and exiting the function. The *stack* is an area of memory that holds function 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 the stack is too small for the local variables defined in currently-called functions. We will see the stack again in Section 5.2.

The overhead due to manipulating the stack pointer on entering and exiting a function should not discourage you from writing modular code. This should only be a concern when your code is fully debugged and you are trying to squeeze a final few nanoseconds out of your program execution time.

5.1.4 The Prefetch Cache Module

In the previous section, we saw that our bootloaded `timing.c` program was executing an assembly instruction nearly every clock cycle. This is because `NU32_Startup()` optimized performance by turning on the prefetch cache module and choosing the minimum number of CPU wait cycles for instructions loading from flash. (See Chapters 3.7 and 4.2.)

Let's try turning off the prefetch cache module to see the effect on our program `timing.c`. The prefetch cache module performs two primary tasks: it keeps recent instructions in the cache, ready if the CPU requests the instruction at that address (allowing the cache to completely store small loops); and for linear code it runs ahead so as to have the instruction ready to go when needed (prefetch). We can disable each of these functions separately, or we can disable both.

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

```
CHECONCLR = 0x30;    // SFR in prefetch cache section of Reference Manual
CheKseg0CacheOff();  // defined in pic32-libs/peripheral/pcache/source/pcache.c
```

right after `NU32_Startup()` in `main`. Everything else stays the same. 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. The second line is a library function that uses MIPS32 assembly instructions to turn off the cache.

Rerunning `timing.c` with these two commands, we find that the LED stays on for approximately 17 seconds, compared to approximately 6.25 seconds before. We are seeing the effect of the flash wait cycles—the CPU has to wait two cycles before receiving requested instructions from flash.

If we comment out the first line, so that the prefetch is enabled but the cache is off, and rerun, we find that the LED stays on for about 7.5 seconds, or 12 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.

Finally, if we comment out the second line but leave the first line, so that the prefetch is disabled but the cache is on, we recover our original performance of approximately 6.25 seconds. The reason is that the entire loop can be stored in the cache, so prefetch is not necessary.

5.1.5 Optimization

Compilers can sometimes recognize ways to increase the speed of execution (or decrease program size) by trimming redundant code, eliminating code that doesn't do anything, and many other ways.[3] For example, the function `delay` in `timing.c` accomplishes nothing but wasting time. In our case, that was the desired effect. If we chose a compiler option to optimize running time, however, we shouldn't be surprised if that code were to be optimized away. Depending on the compiler, something like this could happen even if we didn't choose to optimize for execution time. If you want to implement a delay with predictable behavior, consider using the core timer.

5.1.6 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

[3]See the Command Line chapter of the MPLAB XC32/XC32++ Compiler User's Guide in your `docs` directory if you are interested. Certain optimizations are not available in the free version of the XC32 compiler.

math, scaling the variables as necessary to maintain precision, and only convert back 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.

5.2 Space and the Map File

The previous section focused on the time of execution. Now let's look at 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`. The stack holds 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 thrown away and the space is made available again by simply moving the stack pointer.

If you are building with the MPLAB IDE, the easiest way to keep track of the amount of flash and global variable RAM used by your program is to look at **Window > Dashboard** after a build. If your program attempts to put too many local variables on the stack (stack overflow), however, the error won't show up until run time. The linker does not catch this error because it does not explicitly set aside space for specific local variables; it assumes they will be handled by the stack.

To dig a little deeper into 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. For an executable created from two object code files `file1.o` and `file2.o`, we can create a map file at the command line (Section 3.6) with a linker command of the form

```
xc32-gcc -mprocessor=32MX795F512L -o proj.elf file1.o file2.o -Wl,--script="NU32bootloaded.ld",-Map="proj.map"
```

If you are not getting a map file by default in the MPLAB X IDE, under **Run > Set Project Configuration > Customize > xc32-ld > Diagnostics** you can set the name of the map file. The map file can be opened with a text editor.

Let's create a map file for `timing.c` as shown in Code Sample 5.1. There's a lot in this file, but here's an edited portion of it:

```
Microchip PIC32 Memory-Usage Report

kseg0 Program-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.text                   0x9d001e00          0xacc        2764  App's exec code
.rodata                 0x9d0028cc          0x7d4        2004  Read-only const
.text                   0x9d0030a0          0x168         360  App's exec code
.text.general_exception 0x9d003208           0xdc         220
.text                   0x9d0032e4           0xac         172  App's exec code

  [[[ ... snipping long kseg0_program_mem report ...]]]

.text                   0x9d003580           0x44          68  App's exec code

  [[[ ... snipping long kseg0_program_mem report ...]]]
```

```
.text.UARTSetLineContro 0x9d0038e8           0x28          40
.rodata                 0x9d003910           0x1c          28  Read-only const
.text.INTRestoreInterru 0x9d00392c           0x1c          28
.text.CheKseg0CacheOff  0x9d003948           0x18          24
.text.CheKseg0CacheOn   0x9d003960           0x18          24
.rodata                 0x9d003978           0x18          24  Read-only const
.text                   0x9d003990           0x18          24  App's exec code
.dinit                  0x9d0039a8           0x10          16
.text.INTDisableInterru 0x9d0039b8            0x8           8
.text.INTEnableInterrup 0x9d0039c0            0x8           8
.text._on_reset         0x9d0039c8            0x8           8
.text._on_bootstrap     0x9d0039d0            0x8           8
      Total kseg0_program_mem used  :      0x1bd8        7128  1.4% 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_31              0x9d0005e0            0x8           8  Interrupt Vector 31
          Total exception_mem used  :        0x18          24  0.6% of 0x1000

kseg1 Boot-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  -------------------------  -----------
.reset                  0xbd001970          0x1e0         480  Reset handler
.bev_excpt              0xbd001cf0           0x10          16  BEV-Exception
         Total kseg1_boot_mem used  :       0x1f0         496  42.5% of 0x490
        ------------------------------------------------------------------------
         Total Program Memory used  :      0x1de0        7648  1.5% of 0x80000
        ------------------------------------------------------------------------
```

The kseg0 program memory usage report tells us that 7128 bytes are used for the main part of our program. The first entry is denoted `.text`, which stands for program instructions. It is the largest single section, using 2764 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.

Going down through the subsequent sections of kseg0 program memory, we see that the sections are packed tightly and in order of decreasing section size. The next section is `.rodata`, standing for "read-only data." An example of read-only data is the string on the right-hand side of the following initialized `char` array:

```
char str[] = "my initialized string";
```

Searching the map file for the memory address 0x9D0028CC, we see that the read-only data in question is information about the interrupt table (Chapter 6).

Continuing down, we see other `.text` program instruction sections, including ones named for the specific C functions that were compiled to make the object code in that section. Some of these are functions that are called by `NU32_Startup()` in `NU32.c`; others are functions added automatically to the program by `crt0.o`. The `.text` section that is 172 bytes long corresponds to `timing.o`, and searching the map file for the address 0x9D0032E4 reveals

```
.text           0x9d0032e4       0xac
```

```
 .text          0x9d0032e4       0xac build/default/production/timing.o
                0x9d0032e4                main
                0x9d003314                delay
                0x9d003364                toggleLight
```

our functions `main`, `delay`, and `toggleLight` of `timing.o` are stored consecutively in memory. The addresses agree with our disassembly file from Section 5.1.3.

Continuing, the kseg0 boot memory report indicates that no code is placed in this memory region. The exception memory report indicates that instructions corresponding to interrupts occupy 24 bytes. Finally, the kseg1 boot memory report indicates that `crt0.o` installs reset functions that occupy 480 bytes. The `.reset` section is the first section that the bootloader jumps to.

In all, 7648 bytes of program memory are used.

Continuing further in the map file, we see

```
kseg1 Data-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.bss                     0xa0000000            0x20          32  Uninitialized data
         Total kseg1_data_mem used  :        0x20          32  0.0% of 0x20000
        ------------------------------------------------------------------------
            Total Data Memory used  :        0x20          32  0.0% of 0x20000
        ------------------------------------------------------------------------


Dynamic Data-Memory Reservation
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
heap                     0xa0000028               0           0  Reserved for heap
stack                    0xa0000040         0x1ffb8      131000  Reserved for stack
```

The heap size is zero and all data memory is reserved for the stack. Only 32 bytes of kseg1 data memory are set aside for global variables, beginning at the origin of data memory, 0xA0000000. This section is called `.bss`, which is for uninitialized data. These 32 bytes are the 32 bytes reserved by `NU32.c` in the statement

```
char NU32_RS232OutBuffer[32];
```

Now let's modify our program by adding some useless global variables, just to see what happens to the map file. Let's 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            0x84         132  Uninitialized data
.data                    0xa0000098            0x34          52  Initialized data
         Total kseg1_data_mem used  :        0xca         202  0.2% of 0x20000
        ------------------------------------------------------------------------
            Total Data Memory used  :        0xca         202  0.2% of 0x20000
        ------------------------------------------------------------------------
```

Our global variables now occupy 202 bytes of data memory. 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 or not 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 build/default/production/timing.o
                0xa0000000                my_cat_string
                0xa0000008                my_int
                0xa000000c                _sdata_end = .
```

Even though the string `my_cat_string` uses only 7 bytes, the variable `my_int` starts 8 bytes after the start of `my_cat_string`. This is because variables must be aligned on four-byte boundaries. Similarly, the strings `my_message_string`, `my_small_string`, and `my_big_string` occupy memory to the next four-byte boundary. You are not saving memory by defining a string as 5 bytes instead of 8 bytes.

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) section of the kseg0 program memory report has grown from 16 bytes to 128, meaning that our total program memory used is now 7760 bytes compared to 7648 before.

Now let's make one last change. Let's move the definition

```
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 in the data memory report that the initialized global variable section `.sdata` has shrunk by 8 bytes, as expected.

```
kseg1 Data-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.sdata                   0xa0000000             0x4           4  Small init data
.sbss                    0xa0000004             0x6           6  Small uninit data
.bss                     0xa000000c            0x84         132  Uninitialized data
.data                    0xa0000090            0x34          52  Initialized data
         Total kseg1_data_mem used  :         0xc2         194  0.1% of 0x20000
        ------------------------------------------------------------------------
            Total Data Memory used  :         0xc2         194  0.1% of 0x20000
        ------------------------------------------------------------------------
```

Now looking at the program memory report

```
kseg0 Program-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.text                    0x9d001e00           0xacc        2764  App's exec code
.rodata                  0x9d0028cc           0x7d4        2004  Read-only const
.text                    0x9d0030a0           0x168         360  App's exec code
.text.general_exception 0x9d003208            0xdc         220
.text                    0x9d0032e4           0xc4         196  App's exec code

  [[[ ... snipping long kseg0_program_mem report ...]]]

      Total kseg0_program_mem used  :      0x1c58        7256  1.4% of 0x7e200
```

we see that our `timing.o` code is now 196 bytes as compared to 172 before. This is because the assignment of `my_cat_string` is now taken care of by assembly commands in our code, not by the global variable initialization in `.dinit`. Correspondingly, the global data initialization section `.dinit` shrinks from 128 bytes to 112 bytes.

Finally, we might wish to reserve some RAM for dynamic memory allocation using `malloc` or `calloc`. These functions allow you to declare a variable size array, for example, while the program is running, instead of specifying a (possibly space-wasteful) fixed sized array in advance. By default, MPLAB assumes you will not use dynamic memory allocation and sets the heap size to zero. To set a nonzero heap size, go to **Run > Set Project Configuration > Customize > xc32-ld > General**. If we choose 4 KB, or 4096 bytes, the map file after building shows

```
Dynamic Data-Memory Reservation
section                    address  length [bytes]      (dec)  Description
-------                  ----------  ------------------------  -----------
heap                      0xa00000c8         0x1000       4096  Reserved for heap
stack                     0xa00010e0        0x1ef10     126736  Reserved for stack
```

A heap can also be allocated at the command line using a linker command of the form

```
xc32-gcc -mprocessor=32MX795F512L -o proj.elf file1.o file2.o
         -Wl,--script="NU32bootloaded.ld",-Map="proj.map",--defsym=_min_heap_size=4096
```

The heap is allocated just after the global variables, starting in this case at address 0xA00000C8. The stack grows "down" from the end of RAM—as local variables are added to the stack, the stack pointer address decreases, and when local variables are discarded after exiting a function, the stack pointer address increases.

5.3 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 `WriteCoreTimer(0);` and `unsigned int dt = ReadCoreTimer();` can be used to measure the execution time of the code in between to within a few SYSCLK cycles.
- In MPLAB, use **Window > Output > Disassembly Listing File** to see how your C code is compiled to assembly code. At the command line, use `xc32-objdump -d - S filename.elf > filename.disasm`.
- 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 remainder of RAM is allocated to the heap, for dynamic memory allocation, and to the stack, for function parameters and local variables. The heap is zero bytes by default.
- A summary of program flash and global variable data RAM usage can be found at **Window > Dashboard** in MPLAB after a build.
- A map file provides a more detailed summary of memory usage. If a map file is not created by default, you can create one during the build by choosing a map file name in the MPLAB X IDE under **Run > Set Project Configuration > Customize > xc32-ld > Diagnostics**. At the command line, use the `-Map` option:

  ```
  xc32-gcc -mprocessor=32MX795F512L -o proj.elf file1.o file2.o
    -Wl,--script="NU32bootloaded.ld",-Map="proj.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 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.

5.4 Exercises

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 5.1. With a stopwatch, verify the time taken by the delay loop. Do your results agree with Section 5.1.3?
3. When you look at**Window** > **Output** > **Disassembly Listing File**, it shows you the disassembly of your code, not Microchip object code that you may have linked with. You can try the command-line command

   ```
   xc32-objdump -d - S filename.elf > filename.disasm
   ```

   to create a disassembly listing `filename.disasm` of your entire executable. (The listing will look somewhat different than what you see in MPLAB.) Let's do this for two different versions of some timing code.

   (a) Write a short program that uses `WriteCoreTimer(0)` and `elapsed = ReadCoreTimer()` to time a few C statements. Disassemble your executable and look at it. If you assume that one assembly instruction is executed per clock cycle, how many SYSCLK cycles does it take to complete the `WriteCoreTimer` command? How many cycles does it take to complete the `ReadCoreTimer` command? Approximately how much error will you have in your estimate of the timed code? (It's certainly not a sum of the two.)

   (b) Now replace the `WriteCoreTimer(0)` and `elapsed = ReadCoreTimer()` with their assembly equivalents, as in Section 5.1.2. Disassemble and look at the code, and answer the same questions.
4. 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 and look at the disassembly. For each of the statements, you'll 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.)

5. Let's 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.

6. Use the core timer to calculate a table similar to that in Problem 4, 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 Problem 4.)

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.

7. 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 Problem 6, 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.

8. Explain what stack overflow is, and give a short code snippet (not a full program) that would result in stack overflow on your PIC32.

9. 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.

10. 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.

11. 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 for your particular PIC32?

12. 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.

13. 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.