In Program 3.1 the assumption was the software module had access to all of Port F. In other words, this software owned all pins of Port F. The TM4C123 Port F has only 5 pins, and we used them all. In most cases, a software module needs access to only some of the port pins. If two or more software modules access the same port, a conflict will occur if one module changes modes or output values set by another module. It is good software design to write friendly software, which only affects the individual pins as needed. Friendly software does not change the other bits in a shared register. The Texas Instruments mechanism allows collective access to 0 to 8 bits in a data port. We define eight address offset constants in Table 3.6.

If we wish to access bit	Constant
7	0x0200
6	0x0100
5	0x0080
4	0x0040
3	0x0020
2	0x0010
1	0x0008
0	0x0004

Table 3.6. Address offsets used to specify individual data port bits.

There 256 possible bit combinations we might be interested in accessing, from all of them to none of them. Each possible bit combination has a separate address for accessing that combination. For each bit we are interested in, we add up the corresponding constants from Table 3.6 and then add that sum to the base address for the port. The base addresses for the data ports for each microcontroller can be found in its data sheet; open the data sheet for your microcontroller, go to the **GPIO** chapter, **Register Descriptions** section, and search for **GPIODATA**. Figure 3.14 shows a snapshot of the TM4C123 data sheet, illustrating the base address for Port A is 0x4000.4000. For example, assume we are interested in Port A bits 1, 2, and 3 on the TM4C123. We look up the constants for bits 1, 2, 3 in Table 3.6, which are 0x0008, 0x0010, and 0x0020. The sum of 0x4000.4000 + 0x0008 + 0x0010 + 0x0020 is the address 0x4000.4038. If we read from 0x4000.4038 only bits 1, 2, and 3 will be returned. If we write to this address only bits 1, 2, and 3 will be modified.

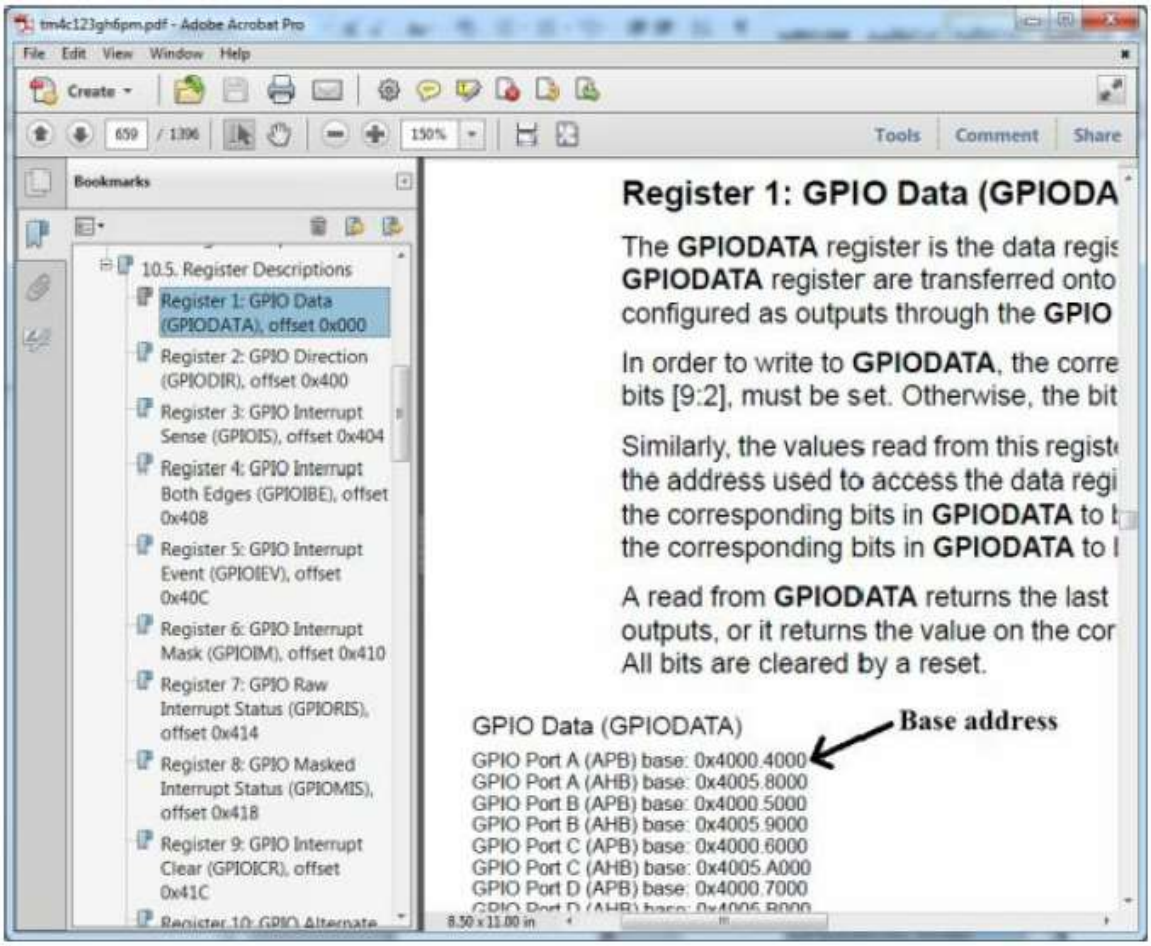

Figure 3.14. Snapshot of the TM4C123 data sheet, showing how to look up GPIO base addresses.

4.2.2. Switch Inputs and LED Outputs

There are four ways to interface a switch to the microcontroller as shown in Figure 4.15.

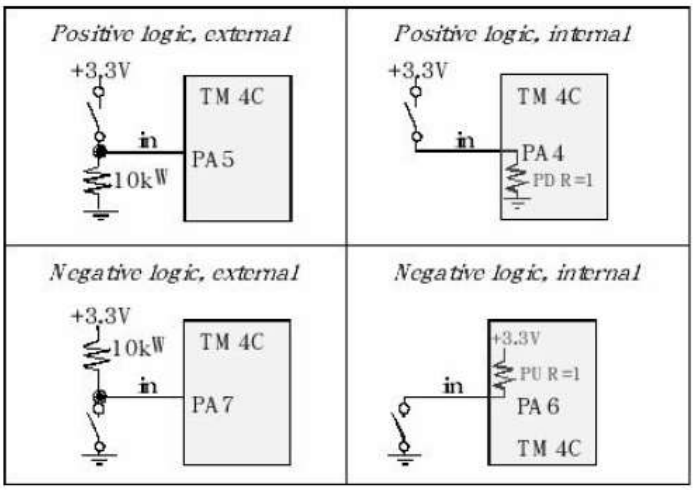

Figure 3.15. Interface of a switch to a microcomputer input.

We can use either positive or negative logic, and we can use an external resistor or select an internal resistor. Notice the positive logic circuit with external resistor is essentially the same as the positive logic circuit with internal resistance; the difference lies with whether the pull-down resistor is connected externally as a 10 kΩ resistor or internally by setting the corresponding PDR bit during software initialization.

In all cases we will initialize the pin as an input. The initialization function will enable the clock, clear the direction register bit to specify input, and enable the pin. In Program 3.2, we will interface PA5 to a switch using an external resistor and positive logic. Notice the software is friendly because it just affects PA5 without affecting the other bits in Port A. The input function reads Port A and returns a true (0x20) if the switch is pressed and returns a false (0) if the switch is not pressed. The first function uses the bit-specific address to get just PA5, while the second reads the entire port and selects bit 5 using a logical AND.

```
PA5 EQU 0x40004080 Switch_Init
LDR R1,=SYSCTL_RCGCGPIO_R
LDR R0, [R1]
ORR R0, R0, #0x01
STR R0, [R1] ; Port A clock
NOP  ; time for clock to finish NOP
LDR R1, =GPIO_PORTA_DIR_R
LDR R0, [R1]
BIC R0, #0x20 ; PA5 input
STR R0, [R1]
LDR R1, =GPIO_PORTA_DEN_R
LDR R0, [R1]
ORR R0, #0x20 ; 7) digital    STR R0,
[R1]  ;    on PA5
BX LR
Switch_Input
LDR R1, =PA5 ; 0x40004080
 LDR R0, [R1] ; read just PA5
BX LR      ; 0x20 or 0x00
Switch_Input2
LDR R1, = GPIO_PORTA_DATA_R
LDR R0, [R1]  ; read port
AND R0, #0x20 ; just bit 5
BX LR      ; 0x20 or 0x00
```

```
#define PA5   (*((volatile uint32_t *)0x40004080))
 void Switch_Init(void){
  SYSCTL_RCGCGPIO_R |= 0x01;
 // 1) activate clock for Port A
 while((SYSCTL_PRGPIO_R&0x01) == 0)
 {};       // ready?
 // 2) no need to unlock GPIO Port A
 GPIO_PORTA_DIR_R &= ~0x20;
 // 5) direction PA5 input
  GPIO_PORTA_DEN_R |= 0x20;
// 7) enable PA5 digital port
}
```

```
 // return 0x20(pressed)
 // or 0(not pressed)
 uint32_t Switch_Input(void){
  return PA5;
  }
// return 0x20(pressed)
// or 0(not pressed)
uint32_t Switch_Input2(void){
return (GPIO_PORTA_DATA_R&0x20); }
```

Program 4.2. Software interface for a switch on PA5 (Switch_xxx.zip).

Example 4.3: Design an embedded system that flashes LEDs in a 0101, 0110, 1010, 1001 binary repeating pattern.

Solution: This system will need four LEDs, and the computer must be able to activate/deactivate them. In this lecture, we will constrain all our designs to include a TM4C microcontroller. Because we have +3.3 V microcontroller systems, we will specify the system to run on +3.3 V power. We have in stock HLMP-4740 green LEDs that operate at 1.9 V and 2 mA, so we will use them.

The data flow graph in Figure 3.19 shows information as it flows from the controller software to the four LEDs. The data flow graph will be important during the subsequent design phases because the hardware blocks can be considered as a preliminary hardware block diagram of the system. The call graph, also shown in Figure 3.19, illustrates this master/slave configuration where the controller software will manipulate the four LEDs. The hardware design of this system could have used four copies of the LED interface presented earlier in Figure 3.9. The TM4C microcontroller can source or sink up to 8 mA. We can save money by using low-current LEDs, which can be connected directly to the microcontroller without a driver.

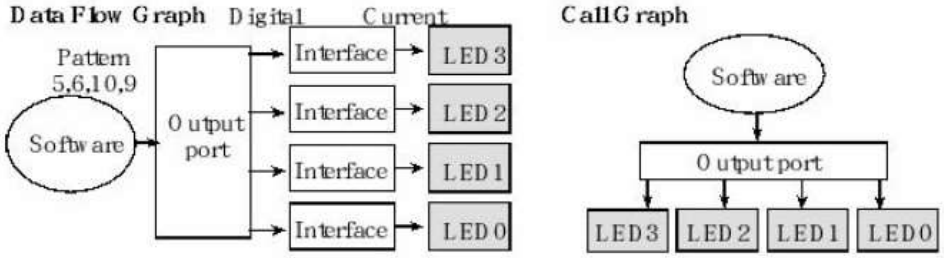

Figure 3.19. Data flow graph and call graph of the LED output system.

Figure 3.20 shows four simple negative logic LED interfaces. A low output will turn on the LED, and a high output will turn it off. Notice the similarity of the data flow graph in Figure 3.19 with the hardware circuit in Figure 3.20.

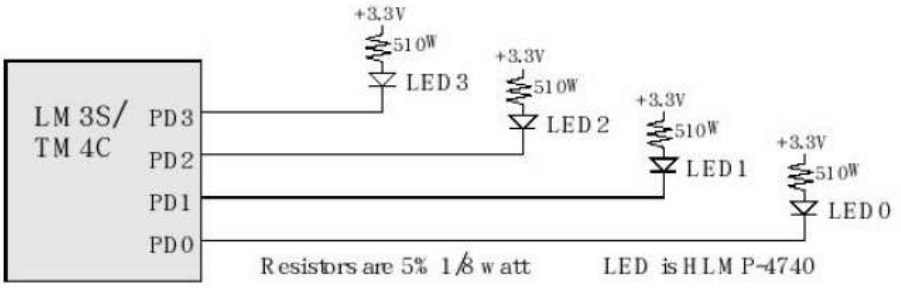

Figure 3.20. Hardware circuit for the LED output system.

The only data required in this problem is the 5–6–10–9 sequence. To output the negative logic pattern 1010 to the LEDs, we will output a 5 to the bottom 4 bits of Port D on the TM4C microcontroller. The **LEDS** definition implements friendly access to pins PD3 – PD0.

```
LEDS EQU 0x4000703C
GPIO_Init
LDR R1, =SYSCTL_RCGCGPIO_R
LDR R0, [R1]  ; 1) D clock
ORR R0, R0, #0x00000008
STR R0, [R1]
NOP      ; time to finish
NOP  ; 2) no need to unlock
LDR R1, =GPIO_PORTD_DIR_R
LDR R0, [R1]     ; 5) direction
ORR R0, R0,#0x0F ; PD3-0 output
STR R0, [R1]
LDR R1, =GPIO_PORTD_DEN_R
LDR R0, [R1]
```

```
#define LEDS  (*((volatile uint32_t *)0x4000703C))
// C implementation
void GPIO_Init(void){
// 1) Port D clock
 SYSCTL_RCGCGPIO_R |= 0x08;
while((SYSCTL_PRGPIO_R&0x08) == 0)
{};// ready?
// 2) no need to unlock PD3-0
// 5) PD3-0 outputs
  GPIO_PORTD_DIR_R |= 0x0F;
// 7) digital I/O on PD3-0
  GPIO_PORTD_DEN_R |= 0x0F;
}
void main(void){
  GPIO_Init();
  while(1){
```

```
 ORR R0, R0, #0x0F ;
7)PD3-0 digital
 STR R0, [R1]
 BX  LR
Start BL GPIO_Init
LDR R0, =LEDS    ; R0 =
0x4000703C
MOV R1, #10      ; R1 = 10
MOV R2, #9       ; R2 = 9
MOV R3, #5       ; R3 = 5
MOV R4, #6      ; R4 = 6 loop STR
R1, [R0]     ; LEDS =   10
STR R2, [R0]     ; LEDS = 9
 STR R3, [R0]     ; LEDS = 5
 STR R4, [R0]     ; LEDS = 6
 B   loop
```

```
      LEDS = 10; // 1010
      LEDS = 9;  // 1001
      LEDS = 5;  // 0101
      LEDS = 6;  // 0110
      }
      }
```

Program 3.5. C software for the LED output system.

2.3. Phase-Lock-Loop

Normally, the execution speed of a microcontroller is determined by an external crystal. The Stellar is EKK-LM3S1968 evaluation board has an 8 MHz crystal. The Texas Instruments Tiva EKLM4F120XL, EK-TM4C123GXL, and EK-TM4C1294-XL boards have a 16 MHz crystal. Most microcontrollers have a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the bus clock will require less power to operate and generate less heat. Speeding up the bus clock obviously allows for more calculations per second.

The default bus speed of the LM3S1968 and TM4C microcontrollers is that of the internal oscillator, also meaning that the PLL is not initially active. For example, the default bus speed for the LM3S1968 kit is 12 MHz ±30%. The default bus speed for the TM4C internal oscillator is 16 MHz ±1%. The internal oscillator is significantly less precise than the crystal, but it requires less power and does not need an external crystal. This means for most applications we will activate the main oscillator and the PLL so we can have a stable bus clock.

There are two ways to activate the PLL. We could call a library function, or we could access the clock registers directly. In general, using library functions creates a better design because the solution will be more stable (less bugs) and will be more portable (easier to switch microcontrollers).

First, we can include the Stellaris/Tiva library and call the **SysCtlClockSet** function to change the speed. This function is defined in the **sysctl.c** file. Assume we wish to run an LM3S with an 8 MHz crystal at 50 MHz. The desired bus speed is set by the **SYSCTL_SYSDIV_4** parameter, which in this case will be 200 MHz divided by 4. The library function activates the PLL because of the **SYSCTL_USE_PLL** parameter. The main oscillator is the one with the external crystal attached. The last parameter specifies the frequency of the attached crystal.

```
SysCtlClockSet ( SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
SYSCTL_OSC_MAIN | SYSCTL_XTAL_8MHZ);
```

Assume we wish to run an LM3S microcontroller with a 6 MHz crystal at 20 MHz. The divide by 10 reduces the 200 MHz base frequency to 20 MHz.

```
SysCtlClockSet( SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL |
SYSCTL_OSC_MAIN | SYSCTL_XTAL_6MHZ);
```

Assume we wish to run an TM4C with a 16 MHz crystal at 80 MHz. The divide by 2.5 creates a bus frequency of 80 MHz, implemented as 400 MHz divided by 5.

SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL |
SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

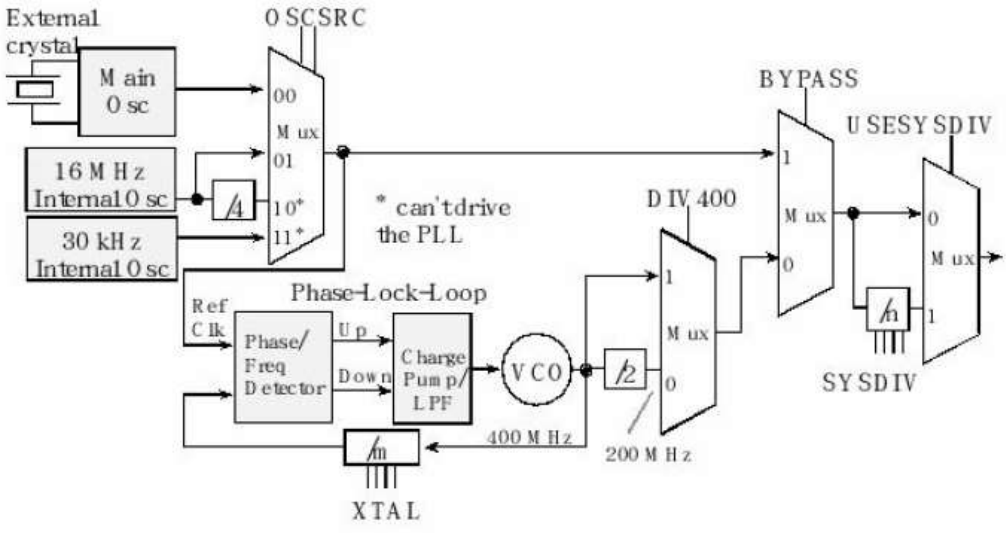

Figure 3.23. Block diagram of the main clock tree on the TM4C123 including the PLL.

To make our code more portable, it is a good idea to use library functions whenever possible. However, we will present an explicit example illustrating how the PLL works. An external crystal is attached to the TM4C microcontroller, as shown in Figure 3.23. The PLLs on the other Stellari s/ Tiva microcontrollers operate in the same basic manner. Table 3.9 shows the clock registers used to define what speed the processor operates. The output of the main oscillator (Main Osc) is a clock at the same frequency as the crystal. By setting the OSCSRC bits to 0, the multiplexer control will select the main oscillator as the clock source.

Program 3.6 shows a program to activate a microcontroller with a 16 MHz main oscillator to run at 80 MHz. 0) Use RCC2 because it provides for more options. 1) The first step is set BYPASS2 (bit 11). At this point the PLL is bypassed and there is no system clock divider. 2) The second step is to specify the crystal frequency in the four XTAL bits using the code in Table 3.9. The OSCSRC2 bits are cleared to select the main oscillator as the oscillator clock source. 3) The third step is to clear PWRDN2 (bit 13) to activate the PLL. 4) The fourth step is to configure and enable the clock divider using the 7-bit SYSDIV2 field. If the 7-bit SYSDIV2 is **n**, then the clock will be divided by **n**+1. To get the desired 80 MHz from the 400 MHz PLL, we need to divide by 5. So, we place a 4 into the SYSDIV2 field. 5) The fifth step is to wait for the PLL to stabilize by waiting

for PLLRIS (bit 6) in the **SYSCTL_RIS_R** to become high. 6) The last step is to connect the PLL by clearing the BYPASS2 bit. To modify this program to operate on other microcontrollers, you will need to change the crystal frequency and the system clock divider. Program 3.6 is on the book web site as PLL_xxx.zip.

XTAL	Crystal Freq (MHz)	XTAL	Crystal Freq (MHz)
0x0	Reserved	0x10	10.0 MHz
0x1	Reserved	0x11	12.0 MHz
0x2	Reserved	0x12	12.288 MHz
0x3	Reserved	0x13	13.56 MHz
0x4	3.579545 MHz	0x14	14.31818 MHz
0x5	3.6864 MHz	0x15	16.0 MHz
0x6	4 MHz	0x16	16.384 MHz
0x7	4.096 MHz	0x17	18.0 MHz
0x8	4.9152 MHz	0x18	20.0 MHz
0x9	5 MHz	0x19	24.0 MHz
0xA	5.12 MHz	0x1A	25.0 MHz
0xB	6 MHz (reset value)	0x1B	Reserved
0xC	6.144 MHz	0x1C	Reserved
0xD	7.3728 MHz	0x1D	Reserved
0xE	8 MHz	0x1E	Reserved
0xF	8.192 MHz	0x1F	Reserved

Table 3.9a. XTAL field used in the SYSCTL_RCC_R register of the TM4C123.

Address	26-23	22	13	11	10-6	5-4	Name
$400FE060	SYSDIV	USESYSDIV	PWRDN	BYPASS	XTAL	OSCSRC	SYSCTL_RCC_R
$400FE050					PLLRIS		SYSCTL_RIS_R
	31	30	28-22	13	11	6-4	
$400FE070	USERCC2	DIV400	SYSDIV2	PWRDN2	BYPASS2	OSCSRC2	SYSCTL_RCC2_R

Table 3.9b. Main clock registers for the TM4C123.

```
#define SYSDIV2 4
void PLL_Init(void){
// 0) Use RCC2
SYSCTL_RCC2_R |= 0x80000000; // USERCC2
// 1) bypass PLL while initializing
SYSCTL_RCC2_R |= 0x00000800; // BYPASS2, PLL bypass
// 2) select the crystal value and oscillator source
SYSCTL_RCC_R = (SYSCTL_RCC_R & ~0x000007C0) // clear bits 10-6
+ 0x00000540; // 10101, configure for 16 MHz crystal
SYSCTL_RCC2_R &= ~0x00000070; // configure for main oscillator source
// 3) activate PLL by clearing PWRDN
SYSCTL_RCC2_R &= ~0x00002000;
// 4) set the desired system divider
SYSCTL_RCC2_R |= 0x40000000; // use 400 MHz PLL
```

```
SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~0x1FC00000)+(SYSDIV2<<22); // 80 MHz
// 5) wait for the PLL to lock by polling PLLLRIS
while((SYSCTL_RIS_R&0x00000040)==0){}; // wait for PLLRIS bit
// 6) enable use of PLL by clearing BYPASS
SYSCTL_RCC2_R &= ~0x00000800;
      }
```

Program 3.6a. Activate the TM4C123 with a 16 MHz crystal to run at 80 MHz (PLL_xxx.zip).

4.4. SysTick Timer

SysTick is a simple counter that we can use to create time delays and generate periodic interrupts. It exists on all Cortex -M microcontrollers, so using SysTick means the system will be easy to port to other microcontrollers. Table 3.10 shows some of the register definitions for SysTick. The basis of SysTick is a 24-bit down counter that runs at the bus clock frequency. There are four steps to initialize the SysTick timer. First, we clear the **ENABLE** bit to turn off SysTick during initialization. Second, we set the **RELOAD** register. Third, we write to the **NVIC_ST_CURRENT_R** value to clear the counter. Lastly, we write the desired mode to the control register, **NVIC_ST_CTRL_R** . We set the **CLK_SRC** bit specifying the core clock will be used. We must set **CLK_SRC**=1, because **CLK_SRC**=0 external clock mode is not implemented on the LM3S/TM4C family. We will set **INTEN** to enable interrupts, but in this first example we clear **INTEN** so interrupts will not be requested. We need to set the **ENABLE** bit so the counter will run. When the **CURRENT** value counts down from 1 to 0, the **COUNT** flag is set. On the next clock, the **CURRENT** is loaded with the **RELOAD** value. In this way, the SysTick counter (**CURRENT**) is continuously decrementing. If the **RELOAD** value is n, then the SysTick counter operates at modulo n+1 (…n, n-1, n-2 … 1, 0, n, n- 1 …). In other words, it rolls over every n+1 counts. The **COUNT** flag could be configured to trigger an interrupt. However, in this first example interrupts will not be generated.

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

Table 3.10. SysTick registers.

If we activate the PLL to run the microcontroller at 80 MHz, then the SysTick counter decrements every 12.5 ns. In general, if the period of the core bus clock is t, then the **COUNT** flag will be set

every (n+1) t. Reading the **NVIC_ST_CTRL_R** control register will return the **COUNT** flag in bit 16 and then clear the flag. Also, writing any value to the **NVIC_ST_CURRENT_R** register will reset the counter to zero and clear the **COUNT** flag.

Program 3.7 uses the SysTick timer to implement a time delay. For example, the user calls **SysTick_Wait10ms (123);** and the function returns 1.23 seconds later. The **RELOAD** register is set to the number of bus cycles one wishes to wait. If the PLL function of Program 3.6 has been executed, then the units of this delay will be 12.5 ns. Writing to **CURRENT** will clear the counter and will clear the count flag (bit 16) of the **CTRL** register. After SysTick has been decremented **delay** times, the count flag will be set and the while loop will terminate. Since SysTick is only 24 bits, the maximum time one can wait with **SysTick_Wait** is 224*12.5ns, which is about 200 ms. To provide for longer delays, the function **SysTick_Wait10ms** calls the function **SysTick_Wait** repeatedly. Notice that 800,000*12.5ns is 10ms.

```
void SysTick_Init(void){
NVIC_ST_CTRL_R = 0; // 1) disable SysTick during setup
NVIC_ST_RELOAD_R = 0x00FFFFFF; // 2) maximum reload value
NVIC_ST_CURRENT_R = 0; // 3) any write to current clears it
NVIC_ST_CTRL_R = 0x00000005; // 4) enable SysTick with core clock
}
// The delay parameter is in units of the 80 MHz core clock. (12.5 ns)
void SysTick_Wait(uint32_t delay){
NVIC_ST_RELOAD_R = delay-1; // number of counts to wait
NVIC_ST_CURRENT_R = 0; // any value written to CURRENT clears
while((NVIC_ST_CTRL_R&0x00010000)==0){ // wait for count flag
}
} // 10000us equals 10ms
void SysTick_Wait10ms(uint32_t delay){
uint32_t i;
for(i=0; i<delay; i++){
SysTick_Wait(800000); // wait 10ms
}
}
```

Program 3.7a. Timer functions that implement a time delay (SysTick_xxx.zip).