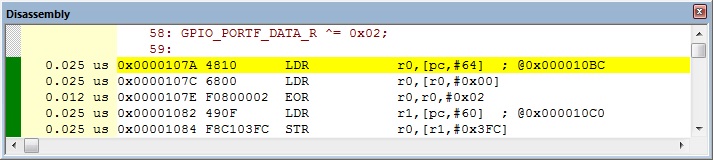
**Debugging Theory**

Every programmer is faced with the need to debug and verify the correctness of his or her software. A debugging instrument is hardware or software used for the purpose of debugging. In this class, we will study hardware-level probes like the logic analyzer, oscilloscope, and Joint Test Action Group (JTAG standardized as the IEEE 1149.1) interface; software-level tools like simulators, monitors, and profilers; and manual tools like inspection and print statements. Non-intrusiveness is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in system performance by the debugging instrument itself. For example, a print statement added to your source code is very intrusive because it significantly affects the real-time interaction of the hardware and software. It is important to quantify the intrusiveness of an instrument. Let *t* be the average time it takes to run the software code comprising the debugging instrument. This time*t* is how much less time the system has, in order to perform its regular duties. Let *Δt* be the average time between executions of the instrument. A quantitative measure of intrusiveness is

*t*/*Δt*

which is the fraction of the time consumed by the process of debugging itself. A debugging instrument is classified as **minimally intrusive** if it has a small but negligible effect on the system being debugged. In other words, if *t*/*Δt* is so small that the debugging activities have a finite but inconsequential effect on the system behavior, we classify it as minimally intrusive. In a real microcomputer system, breakpoints and single-stepping are intrusive, because the real hardware continues to change while the software has stopped. When a program interacts with real-time events, the performance can be significantly altered when using intrusive debugging tools.

For example, the heartbeat code **GPIO\_PORTF\_DATA\_R ^= 0x02;** requires only 9 bus cycles to execute. If the heartbeat runs every 1ms, and the bus clock is 80 MHz, then *t*/*Δt* is equal to 9/80000. Normally, if this ratio is less than 1/1000 we classify it minimally intrusive. The following measurement was obtained in the simulator with **Debug->ExecutionProfile->ShowTime** activated.



In this class the goal of debugging is to maintain and improve software, and the role of a debugger is to support this endeavor. The debugging process is defined as testing, stabilizing, localizing, and correcting errors. Although testing, stabilizing, and localizing errors are important and essential to debugging, they are auxiliary processes: the primary goal of debugging is to remedy faults or to correct errors in a program. **Stabilization** is process of fixing the inputs so that the system can be run over and over again yielding repeatable outputs.

Although, a wide variety of program monitoring and debugging tools are available today, in practice it is found that an overwhelming majority of users either still prefer or rely mainly on “rough and ready” manual methods for locating and correcting program errors. These methods include desk-checking, dumps, and print statements, with print statements being one of the most popular manual methods. Manual methods are useful because they are readily available, and they are relatively simple to use. But, the usefulness of manual methods is limited: they tend to be highly intrusive, and they do not provide adequate control over repeatability, event selection, or event isolation. A real-time system, where software execution timing is critical, usually cannot be debugged with simple print statements, because the print statement itself will require too much time to execute.

**Black-box testing** is simply observing the inputs and outputs without looking inside. Black-box testing has an important place in debugging a module for its functionality. On the other hand, **white-box testing** allows you to control and observe the internal workings of a system. A common mistake made by new engineers is to just perform black box testing. Effective debugging uses both. One must always start with black-box testing by subjecting a hardware or software module to appropriate test-cases. Once we document the failed test-cases, we can use them to aid us in effectively performing the task of white-box testing.

**Common Error:**The most common debugging mistake new programmers make is to simply observe the overall inputs and outputs system without looking inside the device. Then they go to their professor and say, “My program gives incorrect output. Do you know why?”

**Observation:**There are two important components of debugging: having control over events and being able to see what is happening. Remember: **control** and **observability**!

The emergence of concurrent languages and the increasing use of embedded real-time systems place further demands on debuggers. The complexities introduced by the interaction of multiple events or time dependent processes are much more difficult to debug than errors associated with sequential programs. The behavior of non-real-time sequential programs is reproducible: for a given set of inputs their outputs remain the same. In the case of concurrent or real-time programs this does not hold true. Control over repeatability, event selection, and event isolation is even more important for concurrent or real-time environments.

A **print** statement is a common example of a debugging instrument. Using the editor, one adds print statements to the code that either verify proper operation or illustrate the programming errors.

Although using print statements for debugging is widely used in software development, we will avoid using print output when debugging embedded systems because

1. Outputting to a display device takes a 1ms or more, so print statements are usually intrusive.
2. Most embedded systems do not have a display device onto which you can print.
3. If an embedded system has a display it is typically dedicated for the operation of the system and not available for displaying debugging information.

For these reasons we will use a **dump**, which records strategic information into memory.

If we test a system, then remove the instruments, the system may actually stop working, because of the importance of timing in embedded systems. If we leave debugging instruments in the final product, we can use the instruments to test systems on the production line, or test systems returned for repair. On the other hand, sometimes we wish to provide a mechanism to reliably and efficiently remove all instruments when the debugging is done. Consider the following mechanisms as you develop your own unique debugging style.

• Place all instruments in a unique column, so you can easily distinguish instruments from regular programs.

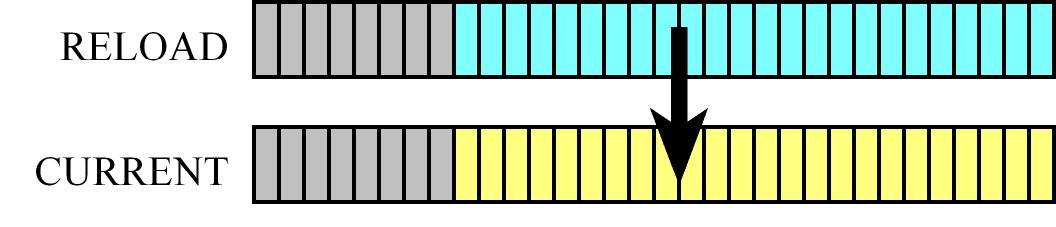
• Define all debugging instruments as functions that all have a specific pattern in their names. In this way, the find/replace mechanism of the editor can be used to find all the calls to the instruments.

• Define the instruments so that they test a run time global flag. When this flag is turned off, the instruments perform no function. Notice that this method leaves a permanent copy of the debugging code in the final system, causing it to suffer a runtime overhead, but the debugging code can be activated dynamically without recompiling. Many commercial software applications utilize this method because it simplifies “on-site” customer support.

• Use conditional compilation (or conditional assembly) to turn on and off the instruments when the software is compiled. When the assembler or compiler supports this feature, it can provide both performance and effectiveness.

**Systick Timer**

Time is an important factor for embedded systems. The SysTick timer is simple mechanism we can use to measure and control time. It is a 24-bit timer such that CURRENT counts down every bus cycle. After CURRENT counts to 0, it is automatically reloaded with the RELOAD value and continues to count.



Before we describe the process of instrumentation, we will discuss a feature that exists on all Cortex-M microcontrollers, a timer, called **SysTick**. Therefore, the use of SysTick in designing your system will assure you that your system will easily port to other Cortex-M microcontrollers. SysTick is a simple counter that we can use to create time delays and generate periodic interrupts. Table 9.1 shows the register definitions for SysTick. The basis of SysTick is a 24-bit down counter, called **CURRENT**, which counts down at the bus clock frequency.

There are four steps involved in the initialization of the SysTick timer. First, we clear the **ENABLE** bit to turn off SysTick during initialization. Second, we set the **RELOAD** register. Third, we write any value 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**. The mode involves the **CLK\_SRC** **INTEN** and **ENABLE** bits. We will set **CLK\_SRC**=1, so the counter runs off the system clock. In Chapter 12, 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. Once the initialization is complete, the timer starts to count down, i.e., **CURRENT** is decremented once every bus cycle. In this class most of the labs run at 80 MHz, so the counter decrements every 12.5ns.

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 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. In this chapter, we set **RELOAD** to 0x00FFFFFF, so the **CURRENT** value is a simple indicator of what count is now. Noting what the count was at some point and then what it is now, allows us to calculate the time that has elapsed

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 0 | 24-bit RELOAD value | | | | |  |
| $E000E018 | 0 | 0 | 24-bit CURRENT value of SysTick counter | | | | |  |

*Table 1. SysTick registers*

Without activating the phase-lock-loop (PLL), our TM4C123 LaunchPad will run at 16 MHz, meaning the SysTick counter decrements every 62.5 ns. For most of the labs in this class, we will activate the PLL to run the microcontroller at 80 MHz, so the SysTick counter decrements every 12.5 ns. In general, if the period of the clock is *t*, and if the **RELOAD** value is *n*, then the **COUNT** flag will be set every (*n*+1)*t*. Note that, reading the **NVIC\_ST\_CTRL\_R** control register will return the **COUNT** flag in bit 16. The act of reading this register when the**COUNT** flag is set will automatically clear it (post read). Also, writing any value to the **NVIC\_ST\_CURRENT\_R** register will reset the counter to zero and clear the **COUNT** flag. Program 1 initializes the SysTick. To determine the time, one simply reads the **NVIC\_ST\_CURRENT\_R** register.

With the **RELOAD** set to 0x00FFFFFF, SysTick behaves like a clock. Every*t* time it counts down by one, and when it reaches zero, the counter is reloaded with 0x00FFFFFF and it continues to count.

If we were to clear **CLK\_SRC** bit to 0, SysTick would run off the precision internal oscillator (16 MHz) divided by 4. Because the internal oscillator has only a 1% accuracy, in this class, we will always set **CLR\_SRC** to 1 so we generate timing off the extremely accurate crystal. The NX5032GA 16-MHz crystal on the LaunchPad has frequency tolerance ±50\*10-6.

#define NVIC\_ST\_CTRL\_R (\*((volatile unsigned long \*)0xE000E010))

#define NVIC\_ST\_RELOAD\_R (\*((volatile unsigned long \*)0xE000E014))

#define NVIC\_ST\_CURRENT\_R (\*((volatile unsigned long \*)0xE000E018))

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

**}**

1. *C language*

NVIC\_ST\_CTRL\_R EQU 0xE000E010

NVIC\_ST\_RELOAD\_R EQU 0xE000E014

NVIC\_ST\_CURRENT\_R EQU 0xE000E018

SysTick\_Init

LDR R1**,** **=**NVIC\_ST\_CTRL\_R

**MOV** R0**,** #0

**STR** R0**,** **[**R1**]** ; 1) disable SysTick during setup

LDR R1**,** **=**NVIC\_ST\_RELOAD\_R

**MOV** R0**,** #0x00FFFFFF

**STR** R0**,** **[**R1**]** ; 2) maximum reload value

LDR R1**,** **=** NVIC\_ST\_CURRENT\_R

**MOV** R0**,** #0

**STR** R0**,** **[**R1**]** ; 3) any write to current clears it

LDR R1**,** **=**NVIC\_ST\_CTRL\_R

**MOV** R0**,** #0x00000005

**STR** R0**,** **[**R1**]** ; 4) enable SysTick with core clock

**BX** LR

1. *Assembly language*

*Program 2. Initialization of SysTick*

Program 2 shows how to measure the elapsed time between calls to a function. Assume the system calls the function **Action()**over and over. The variable **Now** is the time (in 12.5ns units) when the function has been called. The variable **Last** is the time (also in 12.5ns units) when the function was called previously. To measure elapsed time, we perform a time subtraction. Since the SysTick counts down we subtract **Last-Now**. Since the time is only 24 bits and the software variables are 32 bits we “**and”** with 0x00FFFFFF to create a 24-bit difference.

unsigned long Now**;** // 24-bit time at this call (12.5ns)

unsigned long Last**;** // 24-bit time at previous call (12.5ns)

unsigned long Elapsed**;** // 24-bit time between calls (12.5ns)

void Action**(**void**){** // function under test

Now **=** NVIC\_ST\_CURRENT\_R**;** // what time is it now?

Elapsed **=** **(**Last**-**Now**)&**0x00FFFFFF**;** // 24-bit difference

Last **=** Now**;** // set up for next...

**}**

*Program 2. Use of SysTick to measure elapsed time.*

The first measurement will be wrong because there is no previous execution from which to measure. The system will be accurate as long as the elapsed time is less than 0.209 second. More precisely, as long as the elapsed time is less than 224\*12.5ns. This is similar to the problem of using an analog clock to measure elapsed time. For example you notice the clock says 10:00 when you go to sleep, and you notice it says 7:00 when you wake up. As long as you are sure you slept less than 12 hours, you are confident you slept for 9 hours.

Our TM4C123 microcontroller has some 32-bit and some 64-bit timers, but we will use SysTick because it is much simpler to configure. We just have to be aware that we are limited to 24 bits.