GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Fall 2018

Lab #3: Clocks, Timers and Interrupts

Contents

1 Background Material

1.1 Introductory Comments

The objective of this lab is to learn how to use an interrupt to execute a time-critical function at a programmable rate, a design feature utilized in digital control systems. Tasks 1 and 2 of this lab build on Lab 2 in the sense that reading from GPIO inputs and writing to GPIO outputs will now be done using interrupts at a programmable rate, rather than by continuous polling and continuous updating. Task 2 of this lab relates to implementation of controllers in the sense that counting requires updating a discrete-time state variable (the count value) in memory once per cycle, just as a digital controller must update one or more discrete-time state variables each cycle.

Interrupts are hardware-driven or software-driven events that cause the CPU to suspend its current program sequence and execute a subroutine called an interrupt service routine (ISR). Our microcontroller supports both hardware interrupts and software interrupts, and we may use either. Hardware interrupts are triggered by a microcontroller pin or by a peripheral module, whereas software interrupts are triggered by executing code (e.g. by writing to a register). If two interrupts are triggered simultaneously, they would be serviced in sequence according to priority. At the CPU,

an interrupt is either maskable, meaning that it is enabled or disabled through software, or it is non-maskable, meaning that it cannot be blocked. We use only maskable interrupts.

As stated above, when an interrupt is processed, a particular block of code lines gets executed. On the surface, this sounds similar to what happens when a function gets called; however, interrupts and functions are very different concepts, and it is important to understand the distinction. First of all, when you call a function, that action is completely deterministic; in other words, at compile time, the compiler can see precisely where in your code you have called a function, so no special systems within the microcontroller are needed to manage the execution of your code. On the other hand, a hardware interrupt—the type of interrupt used to achieve time-periodic code execution for a differential equation based control algorithm—occurs at points in code not known to the compiler at compile time. Since the compiler cannot know what line of code will be executing when a hardware interrupt happens to occur, interrupt processing requires special-purpose dedicated hardware inside the microcontroller. The internal state of the CPU must be saved when an interrupt occurs (a process called "context save"), as it will be needed again (for "context restore") after the interrupt has been serviced. There are numerous hardware interrupt sources, so internal circuits must identify which interrupt source has been triggered, and then the CPU must locate the address of the particular ISR associated with the triggering interrupt source. This interrupt processing hardware is designed to minimize the latency from the time of the triggering event to the time at which the corresponding ISR begins to execute.

1. Function
   (a) A function is invoked by calling it via software; its execution is deterministic.
   (b) Functions can have arguments, and they can return a value; a typical function prototype might have the form `float32 myFun (float32 x, float32 y);`
   (c) Local variables may be passed as arguments to a function, since they are guaranteed to be in scope at the point where the function is called.
2. Interrupt Service Routine (ISR)
   (a) An ISR is (typically) invoked by hardware events; its execution is non-deterministic.
   (b) An ISR cannot have arguments, and it cannot return a value; an ISR prototype has the special form `interrupt void myISR (void);`
   (c) Global variables must be used for exchanging data with an ISR, since the local variables of interest may not be in scope when the ISR is invoked.

1.2 Relevant Microcontroller Documentation

The overall objective of these lab projects is to teach you how to do embedded design with microcontrollers in a general sense, not just how to approach one specific application using one specific microcontroller. Therefore, the guidance provided herein focuses more on general thought processes and programming recommendations; step-by-step instructions of an extremely specific nature have been intentionally omitted. Use fundamental documentation as your primary source of information as you work through details of implementation. Being able to read and understand such documentation is an important skill to develop, as similar documentation would need to be consulted in order to use other microcontrollers or other application hardware. By making the effort to extract required details from fundamental documentation yourself, you will have developed transferable skills that will serve you well in your engineering career. For this lab, review:

- F28379D Launch Pad Schematic
- F28379D Datasheet
  - Figures 4-1 through 4-4 and Table 4-1, for pin descriptions.
- F28379D Technical Reference Manual
  - §2.7, for details of the clocking system.
  - §2.8, for details of the timer system.
  - §2.4, for details of the interrupt system.

1.3 Target Hardware Schematic Diagrams and Data Sheets

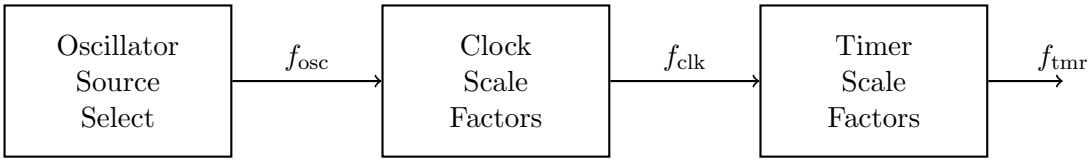
Consult the Datasheet document, Figures 4-1 through 4-4 and Table 4-1, to determine which microcontroller pins are used to connect an external crystal to the internal oscillator circuit; these pins are labeled X1 and X2. Determine the frequency of the crystal on our Launch Pad by locating those same pins on the corresponding schematic diagram; this defines one possible f_{osc} value.

Header pins on target hardware may be used for two purposes; to connect adjacent pins with a jumper (e.g. JP1, JP2 and JP3 on the Launch Pad), or to access signals for interfacing and/or measurement. Using the schematic diagram of the Launch Pad, locate header pin J8-10; we will probe that header pin to measure the frequency and amplitude of a GPIO output signal with an oscilloscope during periodic operation.

2 Timer Interrupts: Step-by-Step Guidelines

2.1 Initialize the Clock and Timer Registers

In order to achieve time-periodic behavior, it is necessary to generate interrupt requests at a desired constant frequency. There are three steps to generating the desired constant frequency: the oscillator source must be selected from four possible options; the clock system scale factors must be set to establish a desired clock frequency from a given oscillator frequency; and finally the timer system scale factors must be set to establish a desired timer reset frequency from a given clock frequency. This chain of dependencies is illustrated in the diagram below.



2.1.1 Select the Oscillator Source

A source of oscillation is needed to generate the system clock signal, and §2.7.1 of Technical Reference Manual describes four sources from which to choose. Two options are internal oscillators, referred to as INTOSC1 and INTOSC2; these options have the advantage of requiring no external components, but they are not extremely accurate. The remaining options are external oscillators, either a crystal placed between the X1-X2 pins, or a signal fed to a GPIO pin; the external crystal option provides the most accurate clock frequency. In later labs, we will use the CAN (Controller Area Network) peripheral module, so we will choose a non-default oscillator source at this time, external crystal option XTAL, due to the following stipulation in §2.7.6:

If CAN or USB is required, an external clock source with a precise frequency must be used as a reference clock. Otherwise, it may be possible to use only INTOSC2 and avoid the need for more external components.

2.1.2 Set the System Clock Frequency

The generation and distribution of various clock signals within the microcontroller are described in §5.9.3 of DATASHEET and in §2.7.2–2.7.3 of TECHNICAL REFERENCE MANUAL. We are primarily concerned with the clock signals referred to as PLLSYSCLK, SYSCLK or CPUCLK throughout these documents, and this section summarizes the information needed to set their frequency.

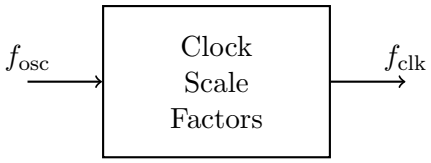


Figure 2-5 of TECHNICAL REFERENCE MANUAL illustrates how the system clock is produced. An oscillator circuit produces the oscillator clock signal OSCCLK, which has frequency f_{osc} (our notation) determined by source register `CLKSRCCTL1`. A phase-locked loop (PLL) multiplier circuit and associated divider circuit produces the system clock signal PLLSYSCLK, which has frequency f_{clk} (our notation) determined by multiplier register `SYSPLLMULT` and divider register `SYSCLKDIVSEL`. The programmable settings of the PLL multiplier circuit and associated divider circuit result in either $f_{\text{clk}} > f_{\text{osc}}$, $f_{\text{clk}} = f_{\text{osc}}$ or $f_{\text{clk}} < f_{\text{osc}}$, as desired. Figure 2-5 shows how the system clock signal is distributed within the microcontroller; note that clock signals SYSCLK and CPUCLK have the same frequency as PLLSYSCLK, but are gated when the CPU enters certain modes of operation (idle, standby or halt).

TI Clock Name	Signal Description	Frequency (Hz)
OSCCLK	Oscillator Clock	f_{osc}
PLLSYSCLK, SYSCLK, CPUCLK	System Clock	f_{clk}

The flexibility to program the system clock frequency is motivated by a trade-off; lower clock frequencies reduce power consumption (a goal of battery-powered portable electronics), but higher clock frequencies permit more computation per sampling interval (a necessity for demanding control applications); see Figures 5-1 and 5-2 of DATASHEET. The relationship between oscillator clock frequency and system clock frequency is summarized by

$$f_{\text{clk}} = f_{\text{osc}} \left(\frac{\text{multiplier}}{\text{divider}} \right)$$

as specified in §2.7.6.1 of TECHNICAL REFERENCE MANUAL. In this expression, the multiplier value is assigned using the `SYSPLLMULT` register described in Figure 2-145 and Table 2-155, whereas the divider value is assigned using the `SYSCLKDIVSEL` register described in Figure 2-150 and Table 2-160. You must consult these register diagrams and field descriptions to learn how the registers should be populated to achieve desired multiplier and divider values. Many combinations of multiplier and divider produce the same system clock frequency; however, to operate within specification, Table 5-12 of DATASHEET specifies the following bounds that must be respected:

$$2 \text{ MHz} \leq f_{\text{clk}} \leq 200 \text{ MHz}, \quad 120 \text{ MHz} \leq f_{\text{osc}} \times \text{multiplier} \leq 400 \text{ MHz}$$

A specific procedure is required to initialize the system clock, as described in §2.7.6.2 of Technical Reference Manual. The essential steps of the procedure are outlined in the code snippet below. Clock frequency boosting requires the PLL to experience a transient response, so it is good practice to postpone execution of frequency-critical code until the PLL has reached steady state; the `LOCKS` bit of the `SYSPLLSTS` register indicates when the PLL has precisely locked on to the requested frequency.[1] Since the PLL will take an indeterminate length of time to reach steady state, this code snippet would be executed prior to `while(1)` between the watchdog disable and watchdog enable code lines.

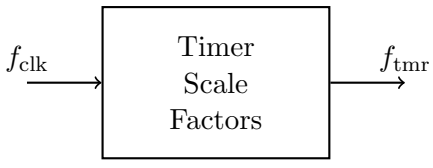
```
ClkCfgRegs.CLKSRCCTL1.bit.OSCCLKSRCSEL = ?;
ClkCfgRegs.SYSPLLCTL1.bit.PLLCLKEN = ?;
ClkCfgRegs.SYSCLKDIVSEL.all = ?;
ClkCfgRegs.SYSPLLMULT.all = ?;
while(ClkCfgRegs.SYSPLLSTS.bit.LOCKS != ?);
ClkCfgRegs.SYSCLKDIVSEL.all = ?;
ClkCfgRegs.SYSPLLCTL1.bit.PLLCLKEN = ?;
ClkCfgRegs.SYSCLKDIVSEL.all = ?;
```

The default value of `SYSPLLMULT` is `0x0000` which corresponds to a multiplier of 1, whereas the default value of `SYSCLKDIVSEL` is `0x0002` which corresponds to a divider of 4. Based on this information, one might expect that the default clock frequency would be $f_{\text{clk}} = f_{\text{osc}}/4$, but this is not the case as indicated in §3.9.8 of Technical Reference Manual which describes clock initializations established by the boot ROM code. Table 3-41 clarifies the following:

1. power-on resets lead to a clock frequency of $f_{\text{clk}} = f_{\text{osc}}$;
2. debugger resets lead to re-use of the clock frequency in use prior to the reset.[2]

2.1.3 Set the Timer Reset Frequency

Digital controllers do essentially the same thing once each sampling period; (1) they monitor the present system response from sensors, (2) they decide what corrective actions are needed by making a calculation which typically requires approximating the solution of a state-space system of differential equations, and (3) they exert influence on the future system response through actuators. The data converters that interface the continuous-time and discrete-time parts of the overall system perform their conversions at specific, though not necessarily equal, frequencies. The notation we will use to describe the operating frequency of the GPIO data converters in this lab is f_{tmr}, a design parameter derived from f_{clk}. How is f_{tmr} programmed?



[1] The provided code snippet should work fine, but TI has issued an advisory concerning the unlikely possibility that (i) the PLL might not start properly after the microcontroller powers up and (ii) the `LOCK` bit might falsely report the PLL reaching steady state. If you encounter this unlikely situation while using the provided code snippet, the PLL would not produce a system clock and the CPU would stop executing instructions. Page 17 of the Errata document discusses this issue and suggests a workaround.

[2] To guarantee that programs execute with intended clock frequencies when using CCS in debug mode, power cycle your device when modifying a project's clock settings or when switching projects.

To establish a desired value for f_{tmr}, we will rely on a peripheral known as a CPU timer as well as the hardware interrupt system. Figure 5-15 in DATASHEET indicates that our microcontroller incorporates three CPU timers for each of its two CPUs. Only one of the timers for each CPU—CPU1 Timer 0 and CPU2 Timer 0—utilizes the peripheral interrupt expansion (PIE) system; for this reason, we will typically be using either CPU1 Timer 0 or CPU2 Timer 0. Like watchdog timers, CPU timers are available for both CPUs; common HAL variable names are used for both instances, since programs written for CPU1 and CPU2 reside in separate CCS projects.

Operation of each CPU timer is described in §2.8 of TECHNICAL REFERENCE MANUAL. According to Figure 2-10, the timer peripheral utilizes two counters; a 16-bit prescale counter that is periodically reloaded with a value stored in `TDDRH:TDDR`, and a 32-bit main counter that is periodically reloaded with a value stored in `PRDH:PRD`. The prescale counter decrements at the system clock frequency, so its period will be one system clock cycle more than `TDDRH:TDDR`. Each time the prescale counter reaches zero, the main counter will decrement once. Consequently, the period of the main counter will be one prescale clock cycle more than `PRDH:PRD`. Combining the influence of both counters, we find the frequency relationship to be

$$f_{\text{tmr}} = \frac{f_{\text{clk}}}{(\texttt{(TDDRH:TDDR)+1})(\texttt{(PRDH:PRD)+1})}$$

As an illustrative example, suppose that `TDDRH:TDDR` is set equal to 3 and that `PRDH:PRD` is set equal to 4. In this case, the prescale counter (above) and the main counter (below) would exhibit the following periodic behavior, where each column corresponds to one system clock cycle:

$\cdots$	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	$\cdots$
$\cdots$	4	4	4	4	3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0	$\cdots$

The total number of system clock cycles in one complete counting cycle would be 20, and this result can be viewed as a special case of the formula above; $f_{\text{tmr}} = f_{\text{clk}}/(3+1)(4+1) = f_{\text{clk}}/20$.

If your application requires counting with 32 (or fewer) bits, then the most systematic solution is to use 16-bit prescale value `TDDRH:TDDR = 0x0000`—its default value—since in this case the 32-bit main value `PRDH:PRD` will meet your requirements on its own. `TDDRH:TDDR` is only needed for timers with very large periods. Focusing on this case, and introducing the HAL-defined variable name for the 32-bit main value `PRDH:PRD`, the above formula reduces to

$$f_{\text{tmr}} = \frac{f_{\text{clk}}}{\texttt{CpuTimer?Regs.PRD.all + 1}}$$

The timer period register `PRD` is described in Figure 2-26 and Table 2-20.

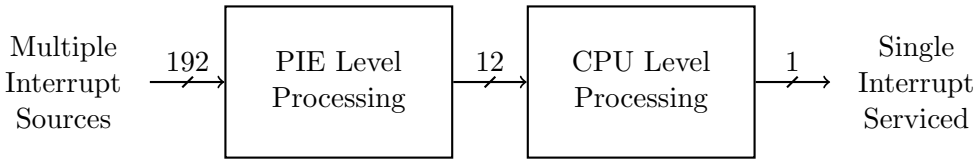
To prepare a CPU timer for action, you will need to interact with its control register; stop the timer using field `TSS`, load the timer using field `TRB`, and enable timer interrupts using field `TIE`. Once the entire interrupt system is set up (see the next section), then you will also need to start the timer using field `TSS`. The HAL-defined variable names available for these fields of the timer control register are as follows:

```
CpuTimer?Regs.TCR.bit.TSS

CpuTimer?Regs.TCR.bit.TRB

CpuTimer?Regs.TCR.bit.TIE
```

The timer control register `TCR` is described in Figure 2-27 and Table 2-21.

2.2 Initialize the Interrupt System Registers

The interrupt system is described in §2.4 of Technical Reference Manual. Since the peripheral modules provide many more interrupt sources than the CPU has interrupt lines, our microcontroller incorporates a peripheral interrupt expansion (PIE) system. At the PIE level, 16 peripheral interrupt sources are multiplexed according to priority[3] onto each of 12 CPU interrupt lines; consequently, a total of 192 possible interrupt sources is supported, each having its own interrupt vector (ISR address). At the CPU level, 12 interrupt lines are multiplexed according to priority[4] so as to determine a single interrupt to service. This interrupt processing system is illustrated at the highest level by the diagram below.



2.2.1 Load the PIE Vector Table

The interrupt vector table can be mapped to various locations in memory, but typically any application that requires peripheral interrupt sources will use the PIE vector table for this purpose. The PIE vector table may be enabled after device reset by properly assigning the `ENPIE` field of the `PIECTRL` register. The diagram and field description for this register are found in Figure 2-30 and Table 2-26 of Technical Reference Manual, and the HAL-defined variable name available for initializing this register field is as follows:

```
PieCtrlRegs.PIECTRL.bit.ENPIE
```

The contents of the PIE vector table (undefined after device reset) must be initialized by application code. This initialization step is quite different from our normal approach to initializing peripheral register fields, and so it deserves a separate explanation. The HAL supports the PIE vector table through the header file `F2837xD_pievect.h`, in which the code line

```
typedef interrupt void (*PINT)(void);
```

defines a pointer-to-interrupt type. The HAL-defined variable name for initializing the PIE vector table for use with CPU Timer 0 is as follows:

```
PieVectTable.TIMER0_INT
```

All such HAL-defined variables incorporate peripheral interrupt source names in accordance with Table 2-2 of Technical Reference Manual; look there or at header file `F2837xD_pievect.h` if you need to work with interrupt sources not listed above. To load the PIE vector table, use expressions such as

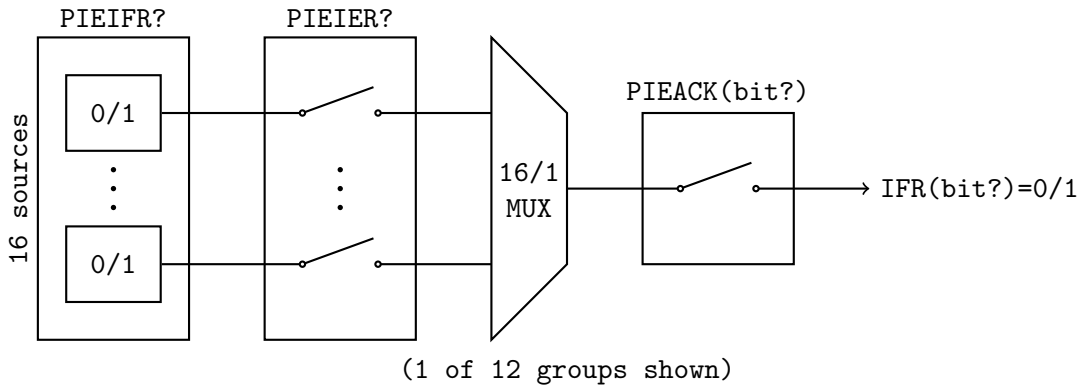
```
PieVectTable.TIMER0_INT = &timerISR;
```

in order to point to an ISR that you write called (for example) `timerISR`.

[3] PIE level interrupt priorities are recorded in column 7 of Table 2-4, Technical Reference Manual.
[4] CPU level interrupt priorities are recorded in column 6 of Table 2-4, Technical Reference Manual.

2.2.2 Enable Interrupts at the PIE Level

The structure of the interrupt processing system at the PIE level—which reduces 192 possible interrupt sources to 12 possible interrupt requests—is illustrated in the diagram below. Each of the 12 groups consists of feed paths from 16 sources. An interrupt request originates within a peripheral module and is indicated by 1 of the 16 source bits being set in 1 of the 12 `PIEIFR?` group registers. The interrupt request reaches the 16-to-1 MUX of its group if its feed path is enabled by setting the appropriate source bit in the appropriate `PIEIER?` group register. Each of the MUXs allows passage of at most one interrupt request; if more than one interrupt request is seen simultaneously by a single MUX, then that MUX allows passage of the interrupt request having highest priority. Interrupt requests complete their journey through the PIE level if their final feed paths are enabled, and this would require that the appropriate group bit in the `PIEACK` register has been cleared; however, each time an interrupt request passes through this final feed path, hardware automatically sets the corresponding acknowledge bit, so the application code must clear that corresponding acknowledge bit at the end of each visit to each ISR in order to allow subsequent processing of future interrupt requests.[5]



Considering the PIE level interrupt diagram shown above, application code is essentially responsible for assigning appropriate states to the "switch" elements. The HAL-defined variable names for enabling interrupts at the PIE level prior to `while(1)` is

```
PieCtrlRegs.PIEIER?.bit.INTx?
```

Proper assignment of the fields within these registers requires review of the register diagrams and register field descriptions in §2.15.3 of Technical Reference Manual. The HAL-defined variable name for re-enabling interrupts at the PIE level at the end of an ISR is

```
PieCtrlRegs.PIEACK.all
```

The `PIEACK` register is special in the sense that one must "write 1 to clear" any bit within this register, and this has implications when using the hardware abstraction layer which ultimately influences targeted registers using "read-modify-write" assembly instructions; see §6 of Hardware Abstraction Layer for more details. Consequently, proper acknowledgment of an interrupt should only be done by writing to all bits within the entire `PIEACK` register simultaneously, and the header file `F2837xD_device.h` includes the macro definitions

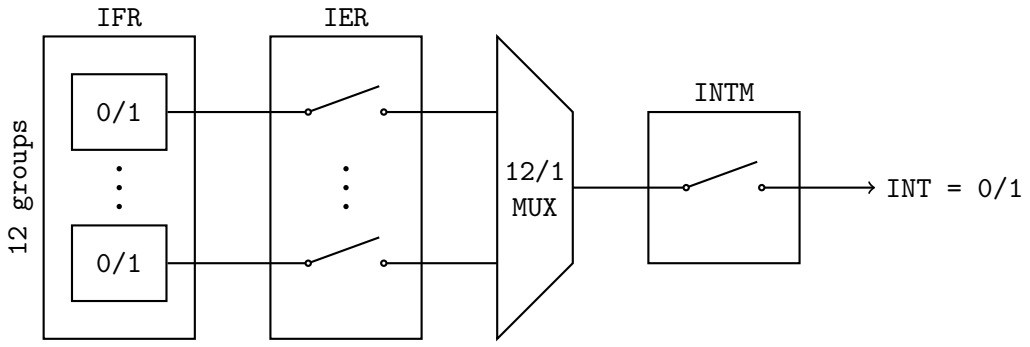
[5]This feature guarantees that a new interrupt request will not be honored until any presently active interrupt has been fully processed. In other words, this practice avoids nested execution of interrupt service routines.

```
#define M_INT1  0x0001
#define M_INT2  0x0002
#define M_INT3  0x0004
#define M_INT4  0x0008
#define M_INT5  0x0010
#define M_INT6  0x0020
#define M_INT7  0x0040
#define M_INT8  0x0080
#define M_INT9  0x0100
#define M_INT10 0x0200
#define M_INT11 0x0400
#define M_INT12 0x0800
```

to systematize the process of acknowledging an interrupt on a per-group basis. Assign a value from this list to correctly acknowledge the appropriate interrupt group, e.g. assign the value `M_INT1` for Group 1, to conform with Figure 2-31 and Table 2-27 of Technical Reference Manual.

2.2.3 Enable Interrupts at the CPU Level

The structure of the interrupt processing system at the CPU level—which reduces 12 possible interrupt requests to a single interrupt request—is illustrated below; see §3 of CPU Instruction Set for a detailed description. Interrupt requests that reach the CPU level are indicated by flag bits being set in the `IFR` register. The interrupt requests reach the 12-to-1 MUX if the corresponding feed path is enabled by setting the appropriate enable bits in the `IER` register. The MUX allows passage of at most one interrupt request; if more than one interrupt request is seen simultaneously by the MUX, then it allows passage of the interrupt request having highest priority. An interrupt request completes its journey through the CPU level if the final feed path is enabled, i.e. if the `INTM` bit has been cleared. Application code is essentially responsible for assigning appropriate states to the "switch" elements in the diagram.



The CPU interrupts `INT1` through `INT12` are individually enabled by assigning an appropriate value to `IER`, a 16-bit CPU control register; the diagram and field description for this register are found in Figure 3-2 of CPU Instruction Set and adjacent text. This is a CPU control register, not a peripheral register, so a special process must be used to assign its value. According to §6.5.2 of Optimizing C Compiler, the `cregister` keyword allows access to CPU control registers directly from C. The header file `F2837xD_device.h` includes the declaration

```
extern cregister volatile unsigned int IER;
```

so assigning a 16-bit value to the IER register is achieved simply by incorporating an assignment statement of the form

```
IER = ?;
```

in your main.c file. The proper choice of ? must be determined by consulting the figure and text referred to above, as it will depend on the identity of the interrupt source(s).

The CPU interrupts INT1 through INT12 are globally enabled by clearing the global enable bit INTM of the 16-bit CPU status register ST1; the diagram and field description for this register are found in Table 2-12 of CPU Instruction Set and adjacent text. This is a CPU status register, not a peripheral register, so assembly language instructions must be used to manipulate it; we have already been manipulating one bit of this CPU status register through use of macro definitions for the assembly language instructions EALLOW and EDIS. Additional macro definitions found within the header file F2837xD_device.h are

```
#define  EINT   asm(" clrc INTM")
#define  DINT   asm(" setc INTM")
```

so globally enabling the CPU interrupts is achieved simply by incorporating the statement

```
EINT;
```

in your main.c file; if the need arises to globally disable CPU interrupts, use statement

```
DINT;
```

in your main.c file.

2.3 Utilize the Timer Interrupts

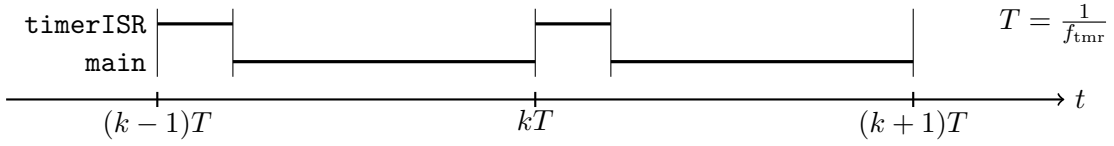
Utilization of timer interrupts requires proper initialization of the clock and timer registers as well as the interrupt system registers and the interrupt vector table. You will also be adding an ISR to your program code, which will now have the general structure shown below. Note that, unlike in Lab 2, the while(1) loop is no longer responsible for polling the GPIO inputs and updating the GPIO outputs; these tasks are now the responsibility of the ISR, and this is desirable since we know by design that we will visit the ISR at a constant frequency that we can choose. This mode of time-synchronized operation is a key feature of our future digital control system designs.

```
#include "F2837xD_device.h"

interrupt void timerISR(void);

// Declare global variables for subroutine timerISR here.

void main(void)
{
    // Put here the code that runs only once for initialization.
    // This is where you should prepare peripherals and interrupts.
    // Do not enable interrupts until you are ready to handle them.

    while(1)
```



```
    {
        // Put here code that runs whenever interrupts are not being serviced.
        // This is an appropriate place for you to service the watchdog timer.
    }
}

interrupt void timerISR(void)
{
    // Put here the code that runs each time the interrupt is serviced.
    // To get data in and out of subroutine timerISR, use global variables.
    // Acknowledge the interrupt before returning from subroutine timerISR.
}
```

The `interrupt` keyword tells the compiler that function `timerISR` is an interrupt function that requires a context save/restore upon entry/exit. The temporally periodic flow resulting from use of this code structure is visualized in the diagram below. At the beginning of each cycle, the `while(1)` continuous loop within function `main` is interrupted by function `timerISR`; once that function has completed its execution, the continuous loop resumes from where it had been interrupted.



3 Lab Assignment

3.1 Pre-Lab Preparation

Each individual student must work through the pre-lab activity and prepare a pre-lab deliverable to be submitted *by the beginning of the lab session*. The pre-lab deliverable consists of a brief typed statement, no longer than two pages, in response to the following pre-lab activity specification:

1. Read through this entire document, and describe the overall purpose of this week's project.
2. Describe in specific terms how the relevant registers will be used to complete the tasks assigned in §3.2. If you will write to a HAL variable, state the numerical value to be written; if you will read from a HAL variable, state how the numerical value read will be interpreted. You will need to consult circuit diagrams in order to complete this part of your pre-lab preparation.
3. Assume that the system clock frequency is $f_{\text{clk}} = 100$ MHz and the desired timer frequency is $f_{\text{tmr}} = 500$ Hz. What value should `CpuTimer0Regs.PRD.all` be assigned?

Please note that it is not essential to write application code prior to the lab session; the point of the pre-lab preparation is for you to arrive at the lab session with firm ideas regarding register usage and other relevant issues in relation to the tasks assigned in §3.2.

3.2 Specification of the Assigned Tasks

3.2.1 Generation of Constant Frequency Waveform

Develop code that generates a 1 kHz squarewave voltage on header pin J8-10, and measure the voltage waveform on an oscilloscope[6] to verify its frequency. Use oscillator source XTAL and $f_{\text{clk}} = 20$ MHz. Use linker command file `user_RAM.cmd` to run your code from RAM.

Instructor Verification (separate page)

3.2.2 Display of Constant Frequency Counting

Develop code that updates a variable every 0.5 s in the repeating sequence $0, 1, 2, 3, 0, 1, 2, 3, \ldots$ and displays the binary result on the red/blue LEDs (`0=off`,`1=on`). Use oscillator source XTAL and $f_{\text{clk}} = 100$ MHz. Use linker command file `user_RAM.cmd` to run your code from RAM.

Instructor Verification (separate page)

[6] To minimize the risk of introducing unintended short circuits that will damage our target hardware, always use jumper wires so that the oscilloscope probe makes its connections some distance away from the Launch Pad.

GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Fall 2018

Lab #3: Clocks, Timers and Interrupts

Instructor Verification Page

Lab Section	Begin Date	End Date
L01, L02	September 11	September 18
L03, L04	September 13	September 20

To be eligible for full credit, do the following:

1. Submissions required by each student (one per student)
   (a) Upload your pre-lab deliverable to canvas before lab session begins on begin date.
   (b) Upload your `main.c` file for §3.2.2 to canvas before lab session ends on end date.
2. Submissions required by each group (one per group)
   (a) Submit a hard-copy of this verification page before lab session ends on end date.

Name #1: _____

Name #2: _____

Checkpoint: Verify completion of the task assigned in §3.2.1.

Verified: _____ **Date/Time:** _____

Checkpoint: Verify completion of the task assigned in §3.2.2.

Verified: _____ **Date/Time:** _____