GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Fall 2017

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:

- Datasheet
  - Figure 4-2, for pin assignments on the 100-pin microcontroller component.
- Technical Reference Manual
  - §1.3.2, for details of the clocking system.
  - §1.3.5, for details of the timer system.
  - §1.6, for details of the interrupt system.

In addition to these documents, you should also review the schematic diagrams of the microcontroller daughtercard and the peripheral explorer motherboard.

1.3 Target Hardware Schematic Diagrams and Data Sheets

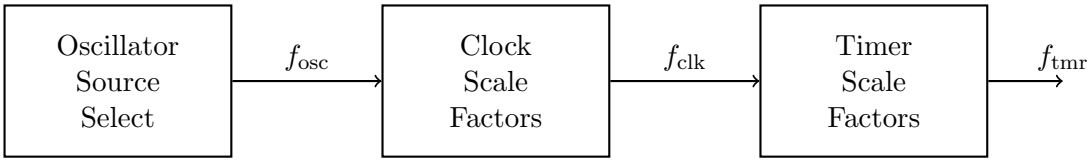
Consult the Datasheet document, Figure 4-2, to determine which F28069 microcontroller pins are used to connect an external crystal to the internal oscillator circuit; these pins are labeled X1 and X2. Determine the frequency of the crystal on our microcontroller daughtercard by locating those same pins 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, or to access signals for measurement. Using the schematic diagram of the peripheral explorer motherboard, locate a header pin that is connected to LD1. We will probe that header pin to measure GPIO output signal voltage 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.

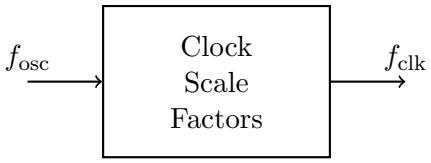


2.1.1 Select the Oscillator Source

A source of oscillation is needed to generate the system clock signal, and §1.3.2.1 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. 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. We will use the default option, internal oscillator INTOSC1; its nominal frequency is displayed in Figure 1-19 of Technical Reference Manual, but more detailed frequency specifications are provided in Table 5-7 of Datasheet.

2.1.2 Set the System Clock Frequency

The generation and distribution of various clock signals within the microcontroller are described in §6.6 of DATASHEET and in §1.3 of TECHNICAL REFERENCE MANUAL. Our primary interest is the system clock signal referred to as SYSCLKOUT throughout these documents, and this section summarizes the information needed to set the frequency of the system clock signal. Table 5-5 of DATASHEET specifies the valid frequency range for the system clock signal.



Figures 1-19 and 1-23 of TECHNICAL REFERENCE MANUAL illustrate how the system clock signal is produced. An oscillator circuit produces the oscillator clock signal OSCCLK, which has frequency f_{osc} (our notation) determined by one of four possible sources. A phase-locked loop (PLL) circuit produces the CPU clock signal CLKIN, which has frequency f_{clk} (our notation) determined by values assigned to the PLL control register `PLLCR` and the PLL status register `PLLSTS`. The programmable settings of the phase-locked loop circuit result in either $f_{\text{clk}} > f_{\text{osc}}$ (frequency boost is only possible if the phase-locked loop circuit is active), $f_{\text{clk}} = f_{\text{osc}}$ or $f_{\text{clk}} < f_{\text{osc}}$, as desired. Figure 1-13 of TECHNICAL REFERENCE MANUAL shows how the system clock signal is ultimately distributed to various peripheral circuits.

TI Clock Name	Signal Description	Frequency (Hz)
OSCCLK	Oscillator Clock	f_{osc}
CLKIN	CPU Clock	f_{clk}
SYSCLKOUT	System Clock	f_{clk}

The relationship between f_{osc} and f_{clk} is specified in Table 6-13 of DATASHEET and Table 1-24 of TECHNICAL REFERENCE MANUAL. The first column of these tables considers various settings of a 5-bit value known as `DIV` (a field of the `PLLCR` register), whereas the remaining columns consider various settings of a 2-bit value known as `DIVSEL` (a field of the `PLLSTS` register). According to Figures 1-25 and 1-26 of TECHNICAL REFERENCE MANUAL, the default value of `DIV` is `00000` and the default value of `DIVSEL` is `00`. The table entries seem to indicate that the default system clock frequency would be $f_{\text{clk}} = f_{\text{osc}}/4$, but Table 6-13 of DATASHEET includes a footnote indicating that the boot ROM code establishes a default system clock frequency of $f_{\text{clk}} = f_{\text{osc}}/1$.

By manipulating the values in `DIV` and `DIVSEL`, it is possible to establish non-default system clock frequencies if desired. This type of design flexibility 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 5-2 and 5-3 of DATASHEET. There are two cases to consider.

Case 1: If the PLL is turned off to achieve low-noise low-power operation, then the system clock frequency will be programmed according to

$$f_{\text{clk}} = \begin{cases} f_{\text{osc}}/4 & \text{, if (DIV = 0) and (DIVSEL = 0 or 1)} \\ f_{\text{osc}}/2 & \text{, if (DIV = 0) and (DIVSEL = 2)} \\ f_{\text{osc}}/1 & \text{, if (DIV = 0) and (DIVSEL = 3)} \end{cases}$$

and consequently it follows that $f_{\text{clk}} \leq f_{\text{osc}}$. In this case, `DIVSEL` is the primary determinant of the system clock frequency.

Case 2: If the PLL is being utilized to establish a system clock frequency that exceeds the oscillator frequency, then the system clock frequency will be programmed according to

$$f_{\text{clk}} = \begin{cases} \texttt{DIV}\, f_{\text{osc}}/4 & \text{, if } (1 \leq \texttt{DIV} \leq 18) \text{ and } (\texttt{DIVSEL} = 0 \text{ or } 1) \\ \texttt{DIV}\, f_{\text{osc}}/2 & \text{, if } (1 \leq \texttt{DIV} \leq 18) \text{ and } (\texttt{DIVSEL} = 2) \\ \texttt{DIV}\, f_{\text{osc}}/1 & \text{, if } (1 \leq \texttt{DIV} \leq 18) \text{ and } (\texttt{DIVSEL} = 3) \end{cases}$$

and consequently it is possible to achieve $f_{\text{clk}} > f_{\text{osc}}$ if desired (but you need to make sure that f_{clk} will not exceed the specification listed in Table 5-5 of DATASHEET). In this case, `DIVSEL` and `DIV` in combination determine the system clock frequency.

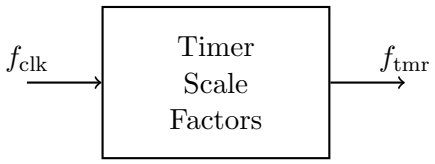
A specific procedure is required to assign `DIVSEL` and `DIV`. Since the PLL takes many clock cycles to reach steady-state behavior, the watchdog timer should be disabled before attempting to assign the system clock frequency; after initialization is complete, the watchdog timer may be enabled once again. The procedure for assigning `DIVSEL` and `DIV` is described in §1.3.2.5 of TECHNICAL REFERENCE MANUAL, specifically the flow chart in Figure 1-24. The relevant register field descriptions are provided in §1.3.2.6 of the same document. The HAL-defined variable names available for accessing the register fields referred to in the flow chart of Figure 1-24 are as follows:

```
SysCtrlRegs.PLLSTS.bit.DIVSEL

SysCtrlRegs.PLLSTS.bit.MCLKOFF

SysCtrlRegs.PLLSTS.bit.MCLKSTS

SysCtrlRegs.PLLSTS.bit.PLLLOCKS

SysCtrlRegs.PLLCR.bit.DIV
```

When in doubt regarding the active values of `DIVSEL` and `DIV`, copy the full HAL-defined variable names for these values into the expressions window to check them. This helps to resolve what the boot ROM code has done, and allows you to verify the modifications your code has made.

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?



To establish a desired value for f_{tmr}, we rely on a peripheral known as a CPU timer as well as the hardware interrupt system. Figure 6-15 in DATASHEET indicates that our microcontroller incorporates three CPU timers. Only one of these timers—CPU Timer 0—utilizes the peripheral interrupt expansion (PIE) system; we will typically be using this timer.

The operation of CPU Timer 0 is described in §1.3.5 of TECHNICAL REFERENCE MANUAL. According to Figure 1-42, the timer peripheral utilizes two counters; a 16-bit prescale counter that is periodically reloaded with a 16-bit value stored in `TDDRH:TDDR`, and a 32-bit main counter that is periodically reloaded with a 32-bit value stored in `PRDH:PRD`.[1] 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; $f_{\text{tmr}} = f_{\text{clk}}/(3+1)(4+1) = f_{\text{clk}}/20$.

In the formula above, the notations `TDDRH:TDDR` and `PRDH:PRD` are as defined in Figure 1-42. However, Table 1-46 indicates that these notations are actually abbreviations of register names; the full register names do not omit the timer name. Since we will be using CPU Timer 0, the notation `PRDH:PRD` associated with the 32-bit main counter must be interpreted as the combination of the two 16-bit registers `TIMER0PRD` and `TIMER0PRDH` (see Tables 1-49 and 1-50). The HAL-defined variable name available for initializing the entire 32-bit main counter is as follows:[2]

```
CpuTimer0Regs.PRD.all
```

The notation `TDDRH:TDDR` associated with the 16-bit prescale counter must be interpreted as the combination of 8-bit fields within the two 16-bit registers `TIMER0TPR` and `TIMER0TPRH` (see Tables 1-52 and 1-53). The HAL-defined variable names available for initializing the 8-bit fields associated with the 16-bit prescale counter are as follows:[3]

```
CpuTimer0Regs.TPR.bit.TDDR
```

```
CpuTimer0Regs.TPRH.bit.TDDRH
```

The field descriptions for timer control register `TIMER0TCR` are found in Table 1-51. To prepare CPU Timer 0 for action, you will need to 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 initializing these fields of the timer control register are as follows:[4]

[1] If your application requires counting with 32 (or fewer) bits, then the most systematic solution is to assign zero to the 16-bit prescale counter, since in this case the 32-bit main counter will meet your requirements on its own.

[2] In this case, the HAL variable naming convention deviates slightly from the normal convention.

[3] In this case, the HAL variable naming convention deviates slightly from the normal convention.

[4] In this case, the HAL variable naming convention deviates slightly from the normal convention.

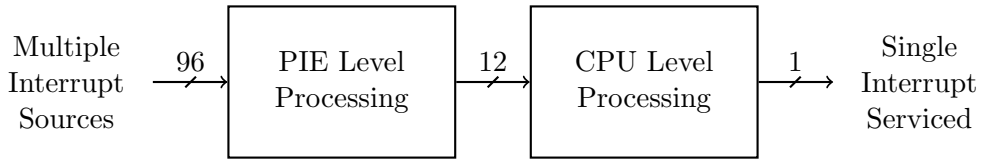
`CpuTimer0Regs.TCR.bit.TSS`

`CpuTimer0Regs.TCR.bit.TRB`

`CpuTimer0Regs.TCR.bit.TIE`

2.2 Initialize the Interrupt System Registers

The interrupt system is described in §1.6 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, 8 peripheral interrupt sources are multiplexed according to priority[5] onto each of 12 CPU interrupt lines; consequently, a total of 96 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[6] 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 1-91 and Table 1-121 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 `F2806x_PieVect.h`, in which the code line

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

defines a pointer-to-interrupt type. The HAL-defined variable names available for initializing the PIE vector table are as follows (only peripheral interrupt sources we will use are included in this list, and each of these is from PIE Group 1 which feeds into CPU interrupt line `INT1`):

`PieVectTable.TINT0`

`PieVectTable.ADCINT1` (first used in Lab 4 for ADC interrupts)

[5] PIE level interrupt priorities are recorded in column 7 of Table 1-119, Technical Reference Manual.
[6] CPU level interrupt priorities are recorded in column 6 of Table 1-119, Technical Reference Manual.

These HAL-defined variables incorporate peripheral interrupt source names in accordance with Table 1-118 of Technical Reference Manual; look there or at header file `F2806x_PieVect.h` if you need to work with interrupt sources not listed above. To load the PIE vector table, use expressions such as

```
PieVectTable.TINT0 = &YourISR
```

in order to point to `YourISR`.

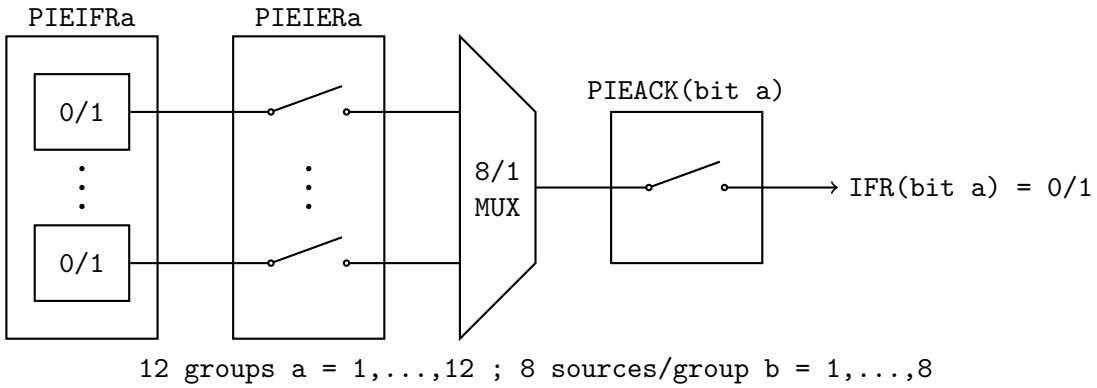
2.2.2 Enable Interrupts at the PIE Level

The structure of the interrupt processing system at the PIE level—which reduces 96 possible interrupt sources to 12 possible interrupt requests—is illustrated in the diagram below. Each of the 12 groups (`a = 1,...,12`) consists of feed paths from 8 sources (`b = 1,...,8`). An interrupt request originates within a peripheral module and is indicated by flag bit `b` being set in the `PIEIFRa` register. The interrupt request reaches the 8-to-1 MUX of group `a` if its feed path is enabled by setting bit `b` in the `PIEIERa` 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 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.[7] 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 `F2806x_Device.h` includes the macro definitions

```
#define PIEACK_GROUP1   0x0001
#define PIEACK_GROUP2   0x0002
#define PIEACK_GROUP3   0x0004
#define PIEACK_GROUP4   0x0008
#define PIEACK_GROUP5   0x0010
#define PIEACK_GROUP6   0x0020
#define PIEACK_GROUP7   0x0040
#define PIEACK_GROUP8   0x0080
#define PIEACK_GROUP9   0x0100
#define PIEACK_GROUP10  0x0200
#define PIEACK_GROUP11  0x0400
#define PIEACK_GROUP12  0x0800
```

to systematize the process of acknowledging an interrupt on a per-group basis; assign the appropriate value from this list using the `.all` hardware abstraction layer variable identified below.

[7] 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.



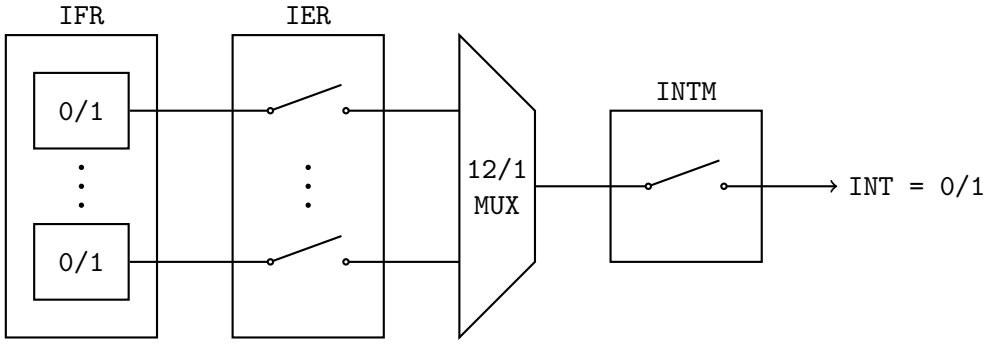
Considering the PIE level interrupt diagram shown above, application code is essentially responsible for assigning appropriate states to the "switch" elements. The HAL-defined variable names available for enabling interrupts at the PIE level using the PIE registers are as follows (the first variable is used at the initialization stage, whereas the second variable is used at the utilization stage within an ISR, as explained above):

```
PieCtrlRegs.PIEIERa.bit.INTxb

PieCtrlRegs.PIEACK.all
```

Proper assignment of the fields within these registers requires review of the register diagrams and register field descriptions in §1.6.5 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. 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 1-96 and Table 1-126 of TECHNICAL REFERENCE MANUAL. 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 `F2806x_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 = YourValue;
```

in your `main.c` file. The proper choice of `YourValue` must be determined by consulting the figure and table referred to above.

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 AND INSTRUCTION SET. 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 `F2806x_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

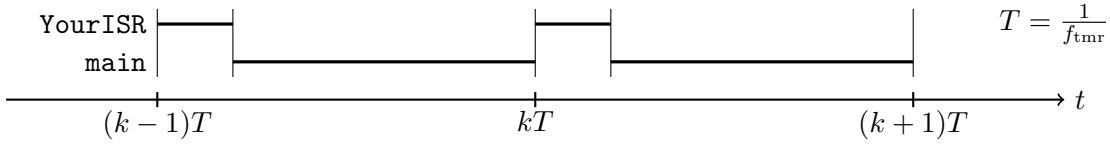
Utilization of timer interrupts requires proper initialization of the clock and timer registers as well as the interrupt system registers and the interrupt vector table. You will also be adding an ISR to your program code, which will now have the general structure shown below. Note that, unlike in Lab 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 "F2806x_Device.h"

interrupt void YourISR(void);

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

The `interrupt` keyword tells the compiler that function `YourISR` 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 `YourISR`; 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 one page, 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. Using Figure 1-24 of TECHNICAL REFERENCE MANUAL, describe how to set the system clock frequency f_{clk} to a desired value; be specific about HAL variable names, the values to be assigned to HAL variables, and the sequence of steps that will be required.
3. Assume that the system clock frequency is $f_{\text{clk}} = 90$ MHz and the desired timer reset frequency is $f_{\text{tmr}} = 500$ Hz. What numerical value should be assigned to `CpuTimer0Regs.PRD.all`?

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 Display of Constant Frequency Heartbeat

Develop code that toggles an LED to indicate that the system is alive; set the system clock frequency to 10 MHz, and toggle LD1 at a frequency of 2 Hz (LD1 should turn on once per second). Measure the GPIO output signal feeding LD1 on an oscilloscope by probing the appropriate header pin, in order to check operating frequency.

Instructor Verification (separate page)

3.2.2 Display of Constant Frequency Counting

Develop code that performs 4-bit continuous counting from `0x0` to `0xF`, and displays the binary count value on the LED array (where LD4/LD1 is the most/least significant bit). The count value displayed on the LED array should change once every 0.5 seconds. The system clock frequency should be set to a value of 90 MHz. Measure the GPIO output signal feeding LD1 on an oscilloscope by probing the appropriate header pin, in order to check operating frequency.

Instructor Verification (separate page)

GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Fall 2017

Lab #3: Clocks, Timers and Interrupts

Instructor Verification Page

Lab Section	Begin Date	End Date
L01, L02	September 12	September 19
L03, L04	September 14	September 21

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 tsquare before lab session begins on begin date.
   (b) Upload your `main.c` file for §3.2.2 to tsquare 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:** _____