GEORGIA INSTITUTE OF TECHNOLOGY

SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Fall 2017

Lab #1: Introduction to Microcontrollers

Contents

1 Introduction

A microcontroller is a single-chip component that includes a processor, non-volatile memory to store program code, volatile memory to execute program instructions, and various modules known as "peripherals" that can be used to monitor and affect the world outside the chip. Microcontrollers are capable of running an operating system underneath application programs, but we will not use them this way in order to extract higher levels of performance and conserve on-chip resources.

The objectives of this first lab project are as follows: to introduce the microcontroller component and the target hardware boards that we will be using this semester; to review certain aspects of the C programming language; to describe a hardware abstraction layer that will simplify programming the microcontroller in C; to provide step-by-step instructions for working with the selected integrated development environment; and to do some preliminary microcontroller programming.

The objective of the next several lab projects is to learn how to utilize a new microcontroller subsystem each week: general-purpose inputs and outputs; clocks, timers and interrupts; analog-to-digital conversion for monitoring motor currents; pulse-width modulation for applying motor voltages; and quadrature-encoder pulse decoding for measuring motor shaft position. All remaining labs focus on microcontroller implementation of controllers for various practical applications.

2 Hardware Overview

2.1 The Microcontroller Component

Throughout the semester, we will use the Texas Instruments C2000 family of microcontrollers; most labs will use the TMS320F28069 (F28069) device, but your individual work with the so-called Launch Pad kit will use the closely related TMS320F28027 (F28027) device. Although we will be using specific microcontrollers, our goal is to understand generally how to do embedded programming on any microcontroller. Therefore, our development process will be based on documents that describe the microcontroller on a fundamental level. The documents summarized in the table below are posted on tsquare, and we will need to refer to them at various times throughout the semester. For the F28069 device, peripherals are described in separate sections of one large document; for the F28027 device, each peripheral is described in a separate document.

Document Description	TI Document Code	# of Pages
F28069 Device ($8.45 in quantity)		
Datasheet	SPRS698E	177
Technical Reference Manual	SPRUH18E	1192
F28027 Device ($3.32 in quantity)		
Datasheet	SPRS523J	131
System Control and Interrupts	SPRUFN3D	139
PWM Module	SPRUGE9E	141
HRPWM Module	SPRUGE8E	44
ADC Module	SPRUGE5F	54
CAP Module	SPRUFZ8A	35
SCI Module	SPRUGH1C	37
SPI Module	SPRUG71B	39
I2C Module	SPRUFZ9D	40
C2000 Family		
Optimizing C Compiler	SPRU514H	193
Assembly Language Tools	SPRU513H	336
CPU and Instruction Set	SPRU430F	551
FPU and Instruction Set	SPRUEO2B	150

2.1.1 Peripheral Modules

One of the major distinguishing features of microcontrollers over general-purpose microprocessors is the existence of additional on-chip circuits called *peripheral modules*. Our microcontroller incorporates many peripheral modules, listed in the DATASHEET document. This semester we will utilize several specific peripheral modules known as the general-purpose input-output module or GPIO (beginning in Lab 2), the analog-to-digital converter module or ADC (beginning in Lab 4), the pulse-width modulator module or PWM and the quadrature encoder pulse module or QEP (beginning in Lab 5). These peripheral modules interface the microcontroller via external pins to the physical systems that we wish to monitor and control. The particular peripheral modules that we use are documented in the TECHNICAL REFERENCE MANUAL.

2.1.2 Memory Mapped Registers

Microcontrollers use their internal memory bus as the means to communicate with their peripheral modules. Certain areas within the internal memory space are not available for data storage but are reserved for interacting with the peripheral modules. Memory addresses used in this way are called *memory-mapped peripheral registers*. Since CPUs have the ability to read from and write to memory addresses, the use of memory-mapped peripheral registers is a convenient way for chip designers to introduce new peripheral modules without creating new CPU instructions.

Peripheral registers are used for two different purposes; some of them are used to configure peripheral functionality, whereas others are used to hold peripheral data. Most peripheral registers are subdivided into smaller fields, each containing just one or at most several contiguous bits, but some peripheral registers have no such subdivisions. One of the most critical features of C programming for microcontrollers in embedded applications is the need to incorporate lines of code that impose desired utilization of peripherals, and writing such code requires an understanding of peripheral register documentation, essentially a two-step process. To help describe this process, consider for example `GPAMUX1`, one of the GPIO multiplexer registers on the F28069.

1. *Understanding the peripheral register diagram.* The diagrams depicting the peripheral registers are located in peripheral-specific sections of TECHNICAL REFERENCE MANUAL. The GPIO peripheral registers, in particular, are described in the Systems Control and Interrupts section of that document. Specifically, §1.4.6 is devoted to the topic of register bit definitions. A search for `GPAMUX1` will bring you to Figure 1-58 which concisely represents the entire register in the form of a diagram. By looking at that diagram, you can determine four things: the names of fields present in the register; their location within the register; whether they can be read from, written to, or both; and what their default value is.

2. *Understanding the peripheral register field description.* The field descriptions of peripheral registers are located in tables just below the corresponding diagrams, and they explain what each field does and how to use the fields properly. Field description tables have one or more rows and typically four columns. For `GPAMUX1`, the field description is provided in Table 1-66. The first column tells you which bits constitute a given field. The second column gives you the name of the relevant field. The third column gives you a list of values that you can assign to the relevant field. The fourth column tells you the effect of setting the relevant field to each value listed in the previous column.

The previously mentioned document contains detailed descriptions of each peripheral. Programmers refer to the peripheral register diagram to learn the name, size, initial value and read/write status of specific bit fields within the register; then they refer to the peripheral register field description to learn the effect of writing to a bit field or to interpret the result of reading from a bit field.

2.2 The Target Hardware

Since we do not use the microcontroller in isolation but as just one component of a larger system, it will also be necessary to refer to other documents describing our target hardware. The target hardware used in Labs 1–4 and Labs 5–9 have been separately listed below by TI part number, and documents describing these components have been posted on tsquare.

Document Description	TI Part Number
Microcontroller Daughtercard	TMDSCNCD28069
Peripheral Explorer Motherboard	TMDSPREX28335
Microcontroller Daughtercard	TMDSCNCD28069MISO
Motor Control Motherboard	DRV8312-69M-KIT
Motor Driver Chip	DRV8312

2.2.1 Microcontroller Daughtercards

The F28069 is just one member of a large family of microcontrollers, known as the C2000 family. Various microcontrollers within this family are available on a small daughtercard format that TI refers to as the controlCARD format. Since all such daughtercards share a common 100-pin edge connector, it is very convenient to move between various microcontrollers within this family during project development. In the lab, we will be satisfied to use just one microcontroller on two different microcontroller daughtercards, but product development engineers working in industry may choose to migrate from a higher-end microcontroller during the initial stage to a lower-cost microcontroller during the final stage. Microcontroller daughtercards contain all essential support circuitry, such as a crystal oscillator, a DC-DC converter circuit, switches to modify boot mode, LEDs to indicate status, filtering and limiting circuits on the ADC inputs, and possibly JTAG emulation.

2.2.2 Peripheral Explorer Motherboard

The first four lab projects will utilize the peripheral explorer motherboard. The microcontroller daughtercard described above mates with this motherboard through the 100-pin edge connector. As its name implies, this particular motherboard is designed to allow convenient exploration of the various peripheral modules incorporated on the microcontroller. We will focus on input devices such as pushbutton switches, HEX encoders and variable resistors, and on output devices such as LEDs and test pins for monitoring signals on an oscilloscope.

2.2.3 Motor Control Motherboard

The next five lab projects will utilize the motor control motherboard, and its 100-pin edge connector interfaces this motherboard with the microcontroller daughtercard in the manner described above. This motherboard is powered from an external supply, and includes a motor driver chip with multiple transistorized half-bridge circuits; therefore, it provides sufficient flexibility to operate either DC motors or three-phase AC motors. Our use of this motherboard will focus on position control applications with DC motors and AC motors. This motherboard incorporates serial communication ports for isolated SPI and CAN communications with remotely located systems.

3 Software Overview

3.1 The C Programming Language

The C programming language has been around since 1972, so there are many sources of information describing it. The standard reference for the language is *The C Programming Language* by Kernighan and Ritchie (2nd edition, Prentice Hall, 1988), but a more readable and complete introduction to the language is *Programming in C* by Kochan (3rd edition, Sams Publishing, 2005). A resource that goes well beyond the language itself by embracing a bottom-up approach to teaching computing systems is *Introduction to Computing Systems: From Bits and Gates to C and Beyond* by Patt and Patel (2nd edition, McGraw-Hill, 2004). C is used for programming microcontrollers in this course because it is the most popular language for embedded applications. C compilers have been developed for essentially all microcontrollers used today, and these compilers provide a simple way to generate compact and efficient executable code for their associated microcontrollers.

The code that you will write this semester will utilize just a small subset of C, so your review of the language may be confined to a few essentials: you will need the preprocessor directives `#include` and `#define`; influence over program flow will typically be achieved by way of the statements `for`, `if`, `else` and `while`; variables should be declared using only the standard data types defined in the TI-supplied header file `F2806x_Device.h`, e.g. `Uint32`, `int32` and `float32`; you will use TI-defined *structures* when accessing peripheral registers, but the variables that you define for your own purposes will typically be scalars or *arrays* of scalars; the `math.h` library will be necessary if certain functions, such as trigonometric functions, are required; single line comments begin with `//` whereas multiline comments must be enclosed by `/*` and `*/`. As a result of the hardware abstraction layer we work with, you will not generally need to use bit operators or pointers.

3.1.1 Embedded Program Structure

Typical applications of microcontrollers, including our control system applications, require that the program code runs continuously. This is quite different from personal computer programs, which may execute the function `main()` just once and then exit to return some value to an operating system. Since our embedded microcontroller should not exit from `main()`, the program structure that we will be using is as shown below. The condition on the `while` loop always evaluates to `true`, so code placed inside that loop will continue to execute until the microcontroller is turned off.

```
// ECE 4550 Lab 0
// Purpose: To illustrate program structure
// Author: David Taylor

#include "F2806x_Device.h"
#include "math.h"
#define pi 3.14159

// Put here global variable and function declarations.

void main(void)
{
    // Put here the code that should run only once.
    // Separate individual tasks into troubleshooting zones.
```

```
    while(1)
    {
        // Put here the code that should run repeatedly.
    }
}
```

Note the inclusion of the header file `F2806x_Device.h`. This file plays a key role in setting up a hardware abstraction layer that will be described in the following section. Only this one device-level header file needs to be included in `main.c`, since this one file automatically includes all necessary peripheral-level header files as described in the Appendix.

Data types. The header file `F2806x_Device.h` also defines standard data types. The standard data types we will use exclusively, such as `Uint32`, `int32` and `float32`, are automatically defined by the `F2806x_Device.h` header file excerpt:

```
typedef unsigned int       Uint16;
typedef unsigned long      Uint32;
typedef unsigned long long Uint64;
typedef int                int16;
typedef long               int32;
typedef long long          int64;
typedef float              float32;
typedef long double        float64;
```

The generic data types, such as `unsigned long`, `long` and `float`, are ambiguous in the sense that they are compiler dependent, i.e. what they mean depends on the specific microcontroller under consideration. To avoid writing ambiguous code, we will only declare variables using the data types defined by the labels appearing in the right column of the above type definitions; in doing so, we will always know the number of bits (16, 32 or 64) associated with each variable we declare.

Protected registers. The header file `F2806x_Device.h` also defines several useful macros. For example, certain critical registers are protected from spurious writes by a CPU-level protection mechanism; without such protection malfunctioning code could lead to unintended register modification with potentially negative consequences. A particular bit in CPU Status Register 1 determines access to protected registers. The `EALLOW` instruction is used to set this bit (allowing access), and the `EDIS` instruction is used to clear this bit (denying access); these instructions are described in the CPU and Instruction Set document. Since `EALLOW` and `EDIS` are assembly language instructions, they cannot be used directly in C code. However, as described in the Optimizing C Compiler document, the statement `asm(" instruction")` extends the C language by embedding any valid assembly language instruction directly into the assembly language output of the C compiler. For example, `asm(" EALLOW")` and `asm(" EDIS")` may appear in C code to enable and deny access to protected registers. To simplify the coding process, two convenient shorthand macros have been defined in the `F2806x_Device.h` header file excerpt:

```
#define  EALLOW asm(" EALLOW")
#define  EDIS   asm(" EDIS")
```

3.1.2 Recommendations and Common Mistakes

1. Throughout code development, please adhere to the following recommendations:

   (a) Have a clear objective for each block of code you write; establish isolated troubleshooting zones for the initialization of each peripheral module (e.g. use function calls).

   (b) Be organized in the way you use `EALLOW` and `EDIS`; during development, consider using these commands just once with a global scope.
   (c) Global variables (defined prior to `main` function) are generally preferred over local variables (defined within `main` function or other functions you create) for two reasons:
      i. The CCS debugger always provides access to global variables no matter where the program counter is pointing when program execution is paused.
      ii. Global variables are the only means to get data into and out of interrupt service routines, which we will make heavy use of in future labs.

2. Prior to submitting your work, please adhere to the following recommendations:
   (a) Delete lines of code that were commented out during the debugging process.
   (b) Add succinct comments to describe defines, variables, functions and interrupts.

3. Some common programming mistakes are listed below:
   (a) Terminating a macro definition with a semicolon.
   (b) Omitting critical parentheses in a macro definition.
   (c) Using invalid array bounds.
   (d) Using invalid integer widths.
   (e) Misunderstanding casting rules.
   (f) Confusing `=` with `==`.
   (g) Confusing operator precedence.
   (h) Omitting prototype declarations.
   (i) Misplacing a semicolon.

3.2 The Hardware Abstraction Layer

A collection of 26 files constituting a hardware abstraction layer has been installed on your workstation to facilitate your code development. This section explains how to use this hardware abstraction layer and points out some of its benefits. If you want to learn more about how the hardware abstraction layer has been implemented within the 26 files, please see Appendix A and/or the TI document number SPRAA85D (which is posted on tsquare).

3.2.1 HAL Advantages

This hardware abstraction layer offers a number of advantages, the first of which is that you were not forced to create the numerous required files yourself; TI wrote the files we are using. Another significant advantage is that application code written so as to leverage this resource is—compared to alternative bit masking methods—easier to write, easier to read, and easier to update. Moreover, the compile process typically results in very efficient assembly code when peripheral registers are accessed this way. Two final advantages will be especially appreciated by programmers:

1. The CCS editor has a convenient code completion feature; as you type, the editor will automatically provide a list of valid register names and/or field names that you may click on.
2. The CCS expressions window automatically expands registers in terms of individual fields, so that you may examine the bit fields easily without the need to extract them by hand.

3.2.2 HAL Variable Names

In this section, the overall framework is summarized from the user's perspective. Depending on the register in question, accesses to the register will involve reading from or writing to C structure variables named using one or more of the following conventions.

```
CategoryName.RegisterName
CategoryName.RegisterName.all
CategoryName.RegisterName.half.LSW
CategoryName.RegisterName.half.MSW
CategoryName.RegisterName.bit.FieldName
```

The first case corresponds to a register that does not have specific fields or that requires writing to many or all fields simultaneously; examples of this case are the Watchdog Control Register (`WDCR`) and the Watchdog Reset Key Register (`WDKEY`) described in §1.3.4 of the Technical Reference Manual document and utilized later in this lab. The HAL-defined variable names we will use for accessing these two registers are as follows.

```
SysCtrlRegs.WDCR
SysCtrlRegs.WDKEY
```

Note that `SysCtrlRegs` represents the `CategoryName` since both registers are affiliated with the System Control Register category, whereas `WDCR` and `WDKEY` represent the `RegisterName` since these are the register names as specified in §1.3.4 of Technical Reference Manual.

The `all` option will treat the entire register as a single entity, whereas the `half` option provides two halves. The `bit` option will be most frequently used, as it provides the possibility of reading from or writing to the specific field identified by the label `FieldName`.

To help organize your code development efforts, the table below summarizes the prefixes of each structure variable that will be used this semester, and the listing indicates which peripheral header file the declaration comes from. This table includes only the structure variable prefixes we need to implement controllers; the HAL incorporates many other non-listed structure variable prefixes.

Structure Prefixes	Header File
`PieVectTable`	`F2806x_PieVect.h`
`PieCtrlRegs`	`F2806x_PieCtrl.h`
`SysCtrlRegs`	`F2806x_SysCtrl.h`
`GpioCtrlRegs, GpioDataRegs`	`F2806x_Gpio.h`
`AdcRegs, AdcResult`	`F2806x_Adc.h`
`CpuTimerNRegs, N = 0,1,2`	`F2806x_CpuTimers.h`
`EPwmNRegs, N = 1,2,3,4,5,6,7,8`	`F2806x_EPwm.h`
`EQepNRegs, N = 1,2`	`F2806x_EQep.h`

3.3 The Integrated Development Environment

We will be using the integrated development environment known as TI Code Composer Studio (CCS), which is documented at `http://processors.wiki.ti.com/index.php/Main_Page`.

3.3.1 Creating a Project

The first time you use CCS you will need to establish a *workspace*. You may target a network drive for this purpose (P drive or Z drive); another alternative is to use a USB flash drive for your workspace, but you must not use the local C drive. Since workspaces can sometimes get corrupted and become problematic, we recommend that you establish a new workspace for each lab session.

For each separate task assigned within each lab session, you will need to create a new CCS *project*. Create each CCS project as follows:

1. From the pull-down menu, select `Project` followed by `New CCS Project`.
2. Provide a `Project name` and associate it with the appropriate workspace.
3. At `Target` select `TMS320F28069` to identify our microcontroller.
4. At `Connection` select `TI XDS100v2 USB Debug Probe` to identify our JTAG connection.
5. Click `Finish`.

At this point, the new project will have been added to your workspace. You can navigate between all of the projects in your workspace by selecting `View` followed by `Project Explorer` from the pull-down menu. In `Project Explorer`, click to expand your project, and verify that `28069_RAM_lnk.cmd` appears as your linker command file and that `TMS320F28069.ccxml` appears as your target connection file. An empty `main.c` source file will have automatically opened so that you may begin typing in your C program code; at this point, you are in CCS *edit perspective*. While editing, you can auto-indent your source code as follows: select all code using `CTRL-A`; right-click on the selected code; select `Source`; select `Correct Indentation`.

Once your project has been created, follow these additional steps to activate the hardware abstraction layer so that you will be able to easily access peripheral modules using structures:

1. Add `#include "F2806x_Device.h"` to your `main.c` file.
   (provides peripheral structure definitions, macro definitions, type definitions)
2. Add `C:\F28069\include` to your project path.[1]
   (locates the device header file and 23 peripheral-specific header files)
3. Copy `C:\F28069\F2806x_GlobalVariableDefs.c` into your project.[2]
   (provides global declaration of peripheral structures and linker interface)
4. Copy `C:\F28069\F2806x_Headers_nonBIOS.cmd` into your project.[3]
   (links peripheral structures to correct peripheral register addresses)

The initial stage of code development is completed by the build process; for this step, click the `Build` button on the toolbar (the "hammer" button). Both the `Console` and `Problems` windows provide feedback on the build process that may be helpful when recovery from errors is necessary.

The build process begins with C source code. The compiler produces assembly language source code, the assembler translates assembly language source code into machine language object modules, and the linker combines object modules into a single executable object module. Once the build process has been successfully completed, the results of it will appear in the `Debug` project folder; e.g. the `out` file, the `obj` file, and the `map` file.

[1]To do this from `Project Explorer`, right click on your project, select `Properties`, `Build`, `C2000 Compiler`, `Include Options`, `Add dir to #include search path`, `Add...`, `Browse...`, to reach the appropriate folder.

[2]To do this from `Project Explorer`, right click on your project, select `Add Files...`, to reach the file.

[3]To do this from `Project Explorer`, right click on your project, select `Add Files...`, to reach the file.

3.3.2 Debugging a Program

The next step is to connect the emulator (also referred to as the debug probe) on the target hardware to the workstation via the USB cable. To download your code into the microcontroller and prepare for its execution, click the `Debug` button on the toolbar (the "bug" button); doing so will lead to the appearance of the CCS *debug perspective* and associated debugging tools. The `View` pull-down menu lists all of the debugging windows that are available.

One of the most important features of the CCS debug perspective is the ability to dictate how your code will be executing. You can run your program in the normal free-running mode, step through the C code line by line, or even step through the compiler-generated assembly language code one instruction at a time. These options are activated by clicking `Resume`, `Step Into` and `Assembly Step Into`. If it is desired to re-initialize a debug session without terminating it, first click `CPU Reset` and then click `Restart`. A debug session may be terminated by clicking `Terminate`.

If you step through your code, you will notice the highlighted lines (the active lines) moving around in the `main.c` file and in the `Disassembly` window. The lines of code that the arrows point to are the lines that will be executed on the next step of the program. To monitor program variables, use the `Expressions` window; this feature is extremely useful as it gives you access to all program variables so that you can determine the source of any programming errors that are detected. This window is not automatically populated, so to use it you must enter the variables of interest either by keyboard or by mouse. Variables in this window can also be modified by editing their values; changes will take place on the next clock cycle.

4 Initial Considerations

4.1 Watchdog Timer Module

The microcontroller incorporates a watchdog timer module, described in §1.3.4 of Technical Reference Manual. The purpose of this module is to reset the microcontroller—and thereby transfer the system into a safe state—in the event of code malfunction, and it fulfills this purpose by using a counter; the counter counts up, and if it manages to reach its maximum value (presumably due to malfunctioning code) then the microcontroller will be reset. It is possible to disable this counter and hence to avoid watchdog resets altogether, but then malfunctioning code could lead to unpredictable and possibly negative consequences. Therefore, the more prudent approach is to periodically reset this counter to zero so that it never reaches its maximum value under normal operation. The watchdog module begins operating immediately after power is applied or a reset occurs, so the executing code must either disable or service the module within a fixed number of clock cycles or else there would be an endless repetition of watchdog-initiated resets.

The above-mentioned document describes how the `WDCR` register may be used to disable the watchdog and how the `WDKEY` register may be used to periodically service the watchdog; the HAL provides access to these registers via `SysCtrlRegs.WDCR` and `SysCtrlRegs.WDKEY`.

4.2 Graphing and Exporting Data

Data saved in an array may be graphed in CCS by selecting `Tools`, `Graph`, `Single Time`, selecting the appropriate `Dsp Data Type`, selecting the appropriate `Acquisition Buffer Size` and `Display Data Size`, and assigning the appropriate `Start Address` (the array variable name); from a graph, right-clicking will provide the option of exporting the graph data into a CSV file that could be read, for example, by Matlab; click `Browse` when providing a `File Name` so that your exported data file will be located where you want it to be, e.g. on your P, Z or USB flash drive.

5 Lab Assignment

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.

5.1 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. Describe why some registers are protected, and describe the purpose of the watchdog timer.

3. Describe how the appropriate registers will be used to complete the tasks assigned in §5.2. Be specific, i.e. state what numerical values need to be assigned to the relevant HAL variables identified in this week's lab documentation (`SysCtrlRegs.WDCR` and `SysCtrlRegs.WDKEY`).

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 §5.2.

5.2 Specification of the Assigned Tasks

5.2.1 Watchdog Timer Fundamentals

Develop code that includes a never-ending `while(1)` loop. Declare global variables `i` and `j`, of type `Uint32`, to provide the following features.

1. Initialize the integer `i` to `0`, and increment its value once every 10^5 loop executions. This variable will be used as a means to monitor program execution.

2. Assign the integer `j` to modify how the watchdog timer is being used.

   (a) `j=0`, watchdog timer is disregarded.
   (b) `j=1`, watchdog timer is disabled before the loop using `SysCtrlRegs.WDCR`.
   (c) `j=2`, watchdog timer is serviced within the loop using `SysCtrlRegs.WDKEY`.

In a debug session, display `i` and `j` in the `Expressions` window and select the `Continuous Refresh` option; set the `Continuous Refresh Interval` to 100 ms. It is possible to run all three watchdog timer cases without recompiling code; to re-run a particular case, click `CPU Reset` followed by `Restart` to re-initialize the debug session, then modify `j` in the `Expressions` window and click `Resume`. Once each case is running, click `Suspend` followed by `Resume` to verify proper operation.

Instructor Verification (separate page)

5.2.2 Graph Tool and Data Export

Develop code that computes and stores coordinate pairs (x, y) describing the function

$$y = 3\cos(2\pi x) - \cos(6\pi x), \quad 0 \leq x < 3.$$

Declare global variables `x` and `y`, as arrays of type `float32` and length `300`. All work should occur within a never-ending `while(1)` loop; use an `if` statement to conditionally compute and store data, and use `SysCtrlRegs.WDKEY` to service the watchdog timer indefinitely. After the code executes and the debug session has been paused, plot `x` and `y` versus array index in CCS, export the `x` and `y` data to CSV files, and generate a properly labeled plot of `y` versus `x` in Matlab.

Instructor Verification (separate page)

A HAL Implementation

The hardware abstraction layer that you will use throughout the semester is implemented by many lines of code distributed throughout a combination of the 26 files listed below.

header file:

```
F2806x_Device.h
```

source file:

```
F2806x_GlobalVariableDefs.c
```

linker command file:

```
F2806x_Headers_nonBIOS.cmd
```

peripheral-specific header files (incorporated in `F2806x_Device.h` by `#include`):

```
F2806x_PieCtrl.h
F2806x_PieVect.h
F2806x_SysCtrl.h
F2806x_CpuTimers.h
F2806x_Gpio.h
F2806x_Adc.h
F2806x_Comp.h
F2806x_EPwm.h
F2806x_EQep.h
F2806x_ECap.h
F2806x_HRCap.h
F2806x_XIntrupt.h
F2806x_NmiIntrupt.h
F2806x_ECan.h
F2806x_Mcbsp.h
F2806x_Spi.h
F2806x_Sci.h
F2806x_I2c.h
```

```
F2806x_Usb.h
F2806x_Cla.h
F2806x_Dma.h
F2806x_BootVars.h
F2806x_DevEmu.h
```

These 26 files are already installed on each workstation. Although it is not necessary for you to fully understand what is in these files in order to make good use of them, this appendix briefly summarizes the underlying philosophy of these files for those who are interested.

Consider the header file `F2806x_XIntrupt.h` which represents one very small part of this extensive programming framework. To understand the concept, it is necessary to refer to §1.6.6 of Technical Reference Manual which describes the external interrupt control registers. Figure 1-98 depicts the format of the `XINTnCR` registers where `n` ranges from `1` to `3`. In each such register, bit `0` is called `ENABLE`, bit `1` is reserved, bits `2` through `3` are called `POLARITY` and bits `4` through `15` are reserved. This pattern is used to define the following *bit field*, where the colon notation indicates both the name and the width of each individual component.

```
struct XINTCR_BITS {
    Uint16   ENABLE:1;    // 0      enable/disable
    Uint16   rsvd1:1;     // 1      reserved
    Uint16   POLARITY:2;  // 3:2    pos/neg, both triggered
    Uint16   rsvd2:12;    // 15:4   reserved
};
```

In order to be able to access this entire register as a whole, or to be able to access just individual components, a *union* is defined as follows.

```
union XINTCR_REG {
   Uint16               all;
   struct XINTCR_BITS   bit;
};
```

Note also that Figure 1-99 depicts the format of the `XINTnCTR` registers where `n` ranges from `1` to `3`. Each of these registers is 16-bits wide and has no individual bits singled out. To represent this entire set of registers, the following *structure* is defined.

```
struct XINTRUPT_REGS {
   union XINTCR_REG XINT1CR;
   union XINTCR_REG XINT2CR;
   union XINTCR_REG XINT3CR;
   Uint16           rsvd1[5];
   Uint16           XINT1CTR;
   Uint16           XINT2CTR;
   Uint16           XINT3CTR;
   Uint16           rsvd2[5];
};
```

It is easy to see the three instances of Figure 1-98 and the three instances of Figure 1-99 in this structure. Their ordering is established by the register addresses summarized in Table 6-19 of the Datasheet document. The final piece of code in this header file is the line

```
extern volatile struct XINTRUPT_REGS XIntruptRegs;
```

which declares a variable `XIntruptRegs` of type `struct XINTRUPT_REGS`. Additional information extracted from Table 6-19 is the starting address of each 16-bit register that is accessible through `XIntruptRegs`. Specifically, `XINT1CR` starts at `0x7070`, followed by `XINTnCR` for `n` from `2` to `3`. Then there is a reserved section from `0x7073` to `0x7077`. After that, `XINT1CTR` starts at `0x7078`, followed by `XINTnCTR` for `n` from `2` to `3`. Then there is a reserved section from `0x707B` to `0x707F`.

In addition to the header file just examined, the hardware abstraction layer programming framework utilizes two additional files. The source file `F2806x_GlobalVariableDefs.c` includes the following lines of code.

```
// Excerpt from F2806x_GlobalVariableDefs.c
// Global Peripheral Variable Definitions and Data Section Pragmas

#pragma DATA_SECTION(XIntruptRegs,"XIntruptRegsFile");
volatile struct XINTRUPT_REGS XIntruptRegs;
```

The first line exchanges information with the linker command file considered next, whereas the second line provides a global declaration of the variable `XIntruptRegs`. The linker command file `F2806x_Headers_nonBIOS.cmd` includes the following instruction that overlays the structure just described onto the memory locations known to hold the external interrupt control registers.

```
// Excerpt from F2806x_Headers_nonBIOS.cmd
// Peripheral Registers Linker Command File

MEMORY
{
 PAGE 0:    /* Program Memory */

 PAGE 1:    /* Data Memory */
...
   XINTRUPT    : origin = 0x007070, length = 0x000010
...
}

SECTIONS
{
...
   XIntruptRegsFile  : > XINTRUPT,    PAGE = 1
...
}
```

The complete programming framework we will be utilizing includes these basic features for every peripheral present on our F28069 microcontroller. Although this framework amounts to thousands of lines of code that silently support your projects from the background, this discussion illustrates that the large size of the framework is due merely to repetition of one concept again and again for each peripheral. Moreover, it relies only on *structures*, *unions* and *bit fields*, and each of these data types is a formal part of the C programming language. Therefore, this same type of framework could be developed for other microcontrollers from other vendors; indeed, most vendors provide this type of framework since it simplifies microcontroller programming in C.

GEORGIA INSTITUTE OF TECHNOLOGY

SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Fall 2017

Lab #1: Introduction to Microcontrollers

Instructor Verification Page

Lab Section	Begin Date	End Date
L01, L02	August 29	September 5
L03, L04	August 31	September 7

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 §5.2.2 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.
   (b) Attach to this page a hard-copy of the Matlab plot requested in §5.2.2.

Name #1: ________________________________

Name #2: ________________________________

Checkpoint: Verify completion of the task assigned in §5.2.1.

Verified: ____________________ **Date/Time:** ____________________

Checkpoint: Verify completion of the task assigned in §5.2.2.

Verified: ____________________ **Date/Time:** ____________________