GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Fall 2018

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 TMS320F28379D (F28379D) device, but your individual efforts on Work-at-Home projects 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 canvas, and we will need to refer to them at various times throughout the semester. For the F28379D 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
F28379D Device ($20.20 in quantity)		
Datasheet	SPRS880J	220
Technical Reference Manual	SPRUHM8G	2668
Errata	SPRZ412I	36
F28027 Device ($3.14 in quantity)		
Datasheet	SPRS523L	139
System Control and Interrupts	SPRUFN3D	139
PWM Module	SPRUGE9E	141
ADC Module	SPRUGE5F	54
SCI Module	SPRUGH1C	37
SPI Module	SPRUG71B	39
I2C Module	SPRUFZ9D	40
Errata	SPRZ292O	20
C2000 Family		
Optimizing C Compiler	SPRU514P	203
Assembly Language Tools	SPRU513P	337
CPU Instruction Set	SPRU430F	551
Extended Instruction Sets	SPRUHS1A	404

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 peripheral modules, such as the general-purpose input-output module or GPIO (beginning in Lab 2), the analog-to-digital converter module or ADC and the digital-to-analog converter module or DAC (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. All peripheral modules 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 F28379D.

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 General-Purpose Input/Output section of that document. Specifically, §7.9 is devoted to the topic of register bit definitions. A search for `GPAMUX1` will bring you to Figure 7-7 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 five columns. For `GPAMUX1`, the field description is provided in Table 7-20. 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 indicates the type of the relevant field (read, write, or both). The fourth column indicates the default value of the relevant field. The fifth column tells you the effect of writing to (or reading from) the relevant field; interpretations for possible field values are listed in this column or appear above the register diagram.

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 boards to be used this semester have been separately listed below by TI part number, and documents describing these components have been posted on canvas. The two Launch Pads we will use, listed below, represent the two extremes of the C2000 family of microcontrollers; the F28027 Launch Pad features a lower-cost single-core microcontroller with native capability for integer math, whereas the F28379D Launch Pad features a higher-cost dual-core microcontroller with native capability for floating-point math.

Target Hardware Description	TI Part Number
F28379D Launch Pad	LAUNCHXL-F28379D
F28027 Launch Pad	LAUNCHXL-F28027
Driver Booster Pack	BOOSTXL-DRV8305EVM
Sensor Booster Pack	BOOSTXL-SENSORS

2.2.1 Launch Pads

TI offers evaluation modules for various microcontrollers in a standardized format, which they refer to as Launch Pads. These evaluation modules include a microcontroller along with essential support circuitry such as a debug probe (JTAG emulator), switches to modify boot mode, LEDs to indicate status and header pins to allow interfacing to external components. The Launch Pad standard systematizes which types of signals may appear on specific header pins, in order to enhance compatibility with expansion boards which they refer to as Booster Packs.

2.2.2 Booster Packs

We will use two Booster Packs this semester, as indicated above. One of the Booster Packs is a power stage that enables DC or AC motor control with high current capability; it incorporates numerous protection features and an SPI interface to assign programmable parameters and to detect warnings and faults. The other Booster Pack provides numerous sensors useful for navigation such as an accelerometer, a gyroscope and a magnetometer, and also sensors for measuring temperature, pressure and humidity; an I2C interface is used for writing to and reading from these sensors.

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 `F2837xD_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 may be necessary if certain 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 "F2837xD_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 `F2837xD_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 `F2837xD_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 `F2837xD_device.h` header file excerpt:

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

The generic data types, such as `long`, `unsigned 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 `F2837xD_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 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 `F2837xD_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 prior to `while(1)`.
   (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 36 files constituting a hardware abstraction layer for the F28379D microcontroller 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 36 files, please see Appendix A and/or the TI document number SPRAA85E (which is posted on canvas).

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 briefly summarized from the user's perspective; as mentioned above, more detail is provided in Appendix A for those who want a more complete description of how the framework actually does its job.

The HAL establishes a direct link between peripheral registers (physical locations in memory) and the numerical values held in variables that TI has declared in header files. Depending on the register in question, accesses to the register will involve reading from or writing to C structure variables named using (typically) the following conventions.

```
CategoryName.RegisterName.all
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 §2.9 and §2.15.4 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.

```
WdRegs.WDCR.all
WdRegs.WDKEY.all
```

Note that `WdRegs` represents the `CategoryName` since both registers are affiliated with the Watchdog Register category, whereas `WDCR` and `WDKEY` represent the `RegisterName` since these are the register names as specified in §2.9 and §2.15.4 of TECHNICAL REFERENCE MANUAL.

The `all` option treats the entire register as a single entity. The `bit` option is very useful, as it provides the possibility of reading from or writing to the specific field identified by the label `FieldName`. However, for the `WDCR` register, the `bit` option is only useful for reads whereas the `all` option must be used for writes; this fact is made clear by considering the description of this register's `WDCHK` field. For the `WDKEY` register, the `all` option is appropriate for reads or writes since there is just one field in this register.

To help organize your code development efforts, each lab document will list the names of all C structure variables that you will need to interact with for that specific lab; the only such variables required for this particular lab are listed above. If you were working from scratch, then you would need to extract those variable names yourself by consulting the variable declarations made in the header files associated with the specific peripheral modules you need to work with.

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 `TMS320F28379D` to identify our microcontroller.
4. At `Connection` select `TI XDS100v2 USB Debug Probe` to identify our JTAG connection.
5. Click `Finish`.
6. Right click on your project in `Project Explorer`.

7. Select Properties, Build, C2000 Compiler, Predefined Symbols.

8. At Pre-define NAME select Add... and type in CPU1.

9. Click Apply and Close.

At this point, the new project will have been added to your workspace and configured such that the build process will be restricted to CPU1. 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 2837x_FLASH_lnk_cpu1.cmd appears as your linker command file and that TMS320F28379D.ccxml appears as your target connection file; both of these files are default options, but we will learn about how to create customized versions of them. 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 "F2837xD_device.h" to your main.c file.
   (provides peripheral structure definitions, macro definitions, type definitions)

2. Add C:\F28379D\include to your project path.[1]
   (locates the device header file and 33 peripheral-specific header files)

3. Copy C:\F28379D\F2837xD_GlobalVariableDefs.c into your project.[2]
   (provides global declaration of peripheral structures and linker interface)

4. Copy C:\F28379D\F2837xD_Headers_nonBIOS_cpu1.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.

3.3.2 Debugging a Program

The next step is to connect the debug probe (also referred to as the JTAG emulator) 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); the first time you do this, you will need to select CPU1 as the processor core to load your code onto.

[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 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.

This process 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 is 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 §2.9 and §2.15.4 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.[4] 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 in practical applications is to "feed the dog" so as to periodically reset this counter to zero so that it never reaches its maximum value when your program is running as intended. Under normal circumstances, 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.[5] The above-mentioned document describes how the `WDCR` register may be used to disable and enable the watchdog, and how the `WDKEY` register may be used to periodically service the watchdog; the HAL provides access to these registers via `WdRegs.WDCR.all` and `WdRegs.WDKEY.all`.

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` (the array variable length), and assigning the appropriate `Start Address` (the array variable name). From a graph, right-clicking will provide the option of exporting the

[4] In this lab, no effort is made to influence what happens when such a reset occurs.

[5] For the F28379D device, the default GEL file used by CCS when launching a debug session disables the watchdog module, but it is critical to learn how to properly use the watchdog module when developing stand-alone applications. For this reason, in this and future labs you will be running your programs with the watchdog module enabled, instead of leaving the watchdog disabled (its default state when debug sessions begin due to the default GEL file).

graph data into a CSV file that could be read, for example, using the Matlab function `xlsread`; 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 (and not on the C drive).

4.3 Linker Command Files

Each CCS project will have two linker command files (`cmd` files) associated with it; the first `cmd` file is part of the hardware abstraction layer that maps structure variables onto peripheral registers (discussed in Appendix A), whereas the second `cmd` file serves to reserve space for program memory and data memory (discussed here). The file `user.cmd`, to be used in Task 2, is listed below. By consulting Table 6-1 in the DATASHEET document (i.e. the memory map of the F28379D device), it is easy to understand the code shown under the `MEMORY` directive. The code shown under the `SECTIONS` directive is where the user defines how the available memory is to be allocated to executable code (`text`), initialized global variables (`cinit`) and uninitialized global variables (`ebss`). More details on `cmd` file design are provided in the ASSEMBLY LANGUAGE TOOLS document.

```
MEMORY
{
PAGE 0:    /* Program Memory */
   RAMLS4               : origin = 0x00A000, length = 0x000800
   RAMLS5               : origin = 0x00A800, length = 0x000800
   RAMGS0123            : origin = 0x00C000, length = 0x004000

PAGE 1:    /* Data Memory */
   RAMM0                : origin = 0x000000, length = 0x000400
   RAMM1                : origin = 0x000400, length = 0x000400
   RAMLS0               : origin = 0x008000, length = 0x000800
   RAMLS1               : origin = 0x008800, length = 0x000800
   RAMLS2               : origin = 0x009000, length = 0x000800
   RAMLS3               : origin = 0x009800, length = 0x000800
   RAMD0                : origin = 0x00B000, length = 0x000800
   RAMD1                : origin = 0x00B800, length = 0x000800
   RAMGS4               : origin = 0x010000, length = 0x001000
   RAMGS5               : origin = 0x011000, length = 0x001000
   RAMGS6               : origin = 0x012000, length = 0x001000
   RAMGS7               : origin = 0x013000, length = 0x001000
   RAMGS89ABCDEF        : origin = 0x014000, length = 0x008000
}

SECTIONS
{
   .text                : > RAMGS0123,             PAGE = 0
   .cinit               : > RAMGS0123,             PAGE = 0
   .ebss                : > RAMGS89ABCDEF,         PAGE = 1
   .stack               : > RAMM1,                 PAGE = 1
   .reset               : > RAMGS0123,             PAGE = 0
}
```

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 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 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 (`WdRegs.WDCR.all` and `WdRegs.WDKEY.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 §5.2.

5.2 Specification of the Assigned Tasks

5.2.1 Watchdog Timer Fundamentals

Create a first CCS project called Task 1. Develop code that includes a never-ending `while(1)` loop. Prior to the `while(1)` loop but inside `main`, write to `WdRegs.WDCR.all` in two consecutive lines, first to disable the watchdog timer and second to enable the watchdog timer; in future labs, these two lines will be separated by code that configures various microcontroller features. Inside the `while(1)` loop, write to `WdRegs.WDKEY.all` in two consecutive lines to repeatedly service the watchdog timer; in this and future labs, these two lines "feed the dog" frequently enough so that bug-free code will not result in a watchdog-initiated reset. Prior to `main` declare two global variables of type `Uint32`, initialized at 0, called `slow` and `fast`. Inside the `while(1)` loop add code that increments `fast` every loop execution and `slow` every 10^5 loop executions; `slow` will be used as a means to visually monitor program execution from the expressions window. Once your code builds without errors, do the following to (hopefully) observe successful operation:

1. click `Debug` to initiate a debug session;
2. add `slow` and `WdRegs.WDCR.bit.WDDIS` to the expressions window;
3. explain why `WdRegs.WDCR.bit.WDDIS = 1` prior to program execution;

4. click `Continuous Refresh` to allow the expressions window to update periodically;
5. click `View Menu` to define your preferred `Continuous Refresh Interval...`;
6. click `Resume` to monitor program execution and explain the results;
7. click `Terminate` to end the debug session.

To learn what a watchdog-initiated reset looks like, do the following:

1. comment out the writes to `WdRegs.WDKEY.all` and rebuild your code;
2. click `Debug` to initiate a debug session (expressions window will already be set up);
3. click `Resume` to monitor program execution and explain the results;
4. click `Terminate` to end the debug session.

Instructor Verification (separate page)

5.2.2 Data Logging and Data Export

Create a second CCS project called Task 2. Copy the source code in `main.c` from Task 1 to initiate your programming for Task 2. Prior to `main` declare global variables `x` and `y`, as arrays of type `float32` and length `5000`. Inside the `while(1)` loop, add code that conditionally computes and stores coordinate pairs (x, y) describing the function

$$y = \cos(2\pi x), \quad 0 \leq x < 1,$$

where an `if` statement keeps you from attempting to populate the array variables `x` and `y` beyond their bounds. Attempt to build your program using the following linker command files, in the sequence listed (only one of these should be seen in `Project Explorer` for each build attempt):

1. `2837x_FLASH_lnk_cpu1.cmd` (default `cmd` file assigned by CCS, to load into FLASH)
2. `2837x_RAM_lnk_cpu1.cmd` (default `cmd` file assigned by CCS, to load into RAM)
3. `user.cmd` (custom `cmd` file provided in canvas, to load into RAM)

In the first two cases, the build will be unsuccessful because of how the `cmd` file has been written; an error message will appear stating `the program will not fit into available memory`. In the third case, there should be no build problem since the customized `cmd` file allocates a block consisting of 32,768 (`0x8000`) 16-bit words of RAM for uninitialized global variables (open the file in CCS to see for yourself); such a large block of RAM can actually store 16,384 (`0x4000`) 32-bit words of data, which easily accommodates the 10,000 `float32` data values required here. After a successful build, run your program in a debug session for a brief moment and then pause its execution. Plot separately `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 36 files listed below.

header file:

`F2837xD_device.h`

source file:

`F2837xD_GlobalVariableDefs.c`

linker command files:

`F2837xD_Headers_nonBIOS_cpu1.cmd`

`F2837xD_Headers_nonBIOS_cpu2.cmd`

peripheral-specific header files (incorporated in `F2837xD_device.h` by `#include`):

```
F2837xD_adc.h
F2837xD_analogsubsys.h
F2837xD_can.h
F2837xD_cla.h
F2837xD_cmpss.h
F2837xD_cputimer.h
F2837xD_dac.h
F2837xD_dcsm.h
F2837xD_dma.h
F2837xD_ecap.h
F2837xD_emif.h
F2837xD_epwm.h
F2837xD_epwm_xbar.h
F2837xD_eqep.h
F2837xD_flash.h
F2837xD_gpio.h
F2837xD_i2c.h
F2837xD_input_xbar.h
F2837xD_ipc.h
F2837xD_mcbsp.h
F2837xD_memconfig.h
F2837xD_nmiintrupt.h
F2837xD_output_xbar.h
F2837xD_piectrl.h
F2837xD_pievect.h
F2837xD_sci.h
F2837xD_sdfm.h
F2837xD_spi.h
F2837xD_sysctrl.h
F2837xD_upp.h
```

```
F2837xD_xbar.h
F2837xD_xint.h
```

These 36 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 `F2837xD_xint.h` which represents one very small part of this extensive programming framework. To understand the concept, it is necessary to review the description of the external interrupt control registers in TECHNICAL REFERENCE MANUAL. Figures 2-68 through 2-72 depict the registers `XINT1CR` through `XINT5CR`. 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 *bit fields*, one of which is shown below, where the colon notation indicates both the name and the width of each individual component.

```
struct XINT1CR_BITS {
    Uint16 ENABLE:1;        // 0 XINT1 Enable
    Uint16 rsvd1:1;         // 1 Reserved
    Uint16 POLARITY:2;      // 3:2 XINT1 Polarity
    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, *unions* such as the one shown below are defined.

```
union XINT1CR_REG {
    Uint16  all;
    struct  XINT1CR_BITS  bit;
};
```

Figures 2-73 through 2-75 depict external interrupt counter registers `XINT1CTR` through `XINT3CTR`. Each of these registers is 16-bits wide and has no individual bits singled out.

To represent this entire set of external interrupt registers, the following *structure* is defined.

```
struct XINT_REGS {
    union       XINT1CR_REG     XINT1CR;        // XINT1 configuration register
    union       XINT2CR_REG     XINT2CR;        // XINT2 configuration register
    union       XINT3CR_REG     XINT3CR;        // XINT3 configuration register
    union       XINT4CR_REG     XINT4CR;        // XINT4 configuration register
    union       XINT5CR_REG     XINT5CR;        // XINT5 configuration register
    Uint16                      rsvd1[3];       // Reserved
    Uint16                      XINT1CTR;       // XINT1 counter register
    Uint16                      XINT2CTR;       // XINT2 counter register
    Uint16                      XINT3CTR;       // XINT3 counter register
};
```

The five instances of Figures 2-68 through 2-72 and the three instances of Figures 2-73 through 2-75 appear clearly in this structure. The ordering and locations of these instances within the structure are established by the offset values specified just above Figures 2-68 through 2-75. The final piece of code in this header file is the line

```
extern volatile struct XINT_REGS XintRegs;
```

which declares a variable `XintRegs` of type `struct XINT_REGS`.

In addition to the header file just examined, the hardware abstraction layer programming framework utilizes two or three additional files. The source file `F2837xD_GlobalVariableDefs.c` includes the following lines of code.

```
// Excerpt from F2837xD_GlobalVariableDefs.c
// Global Peripheral Variable Definitions and Data Section Pragmas

#pragma DATA_SECTION(XintRegs,"XintRegsFile");
volatile struct XINT_REGS XintRegs;
```

The first line exchanges information with the linker command file considered next, whereas the second line provides a global declaration of the variable `XintRegs`. The linker command files `F2837xD_Headers_nonBIOS_cpu1.cmd` and `F2837xD_Headers_nonBIOS_cpu2.cmd` include the following instruction that overlays the structure just described onto the memory locations known to hold the external interrupt registers.

```
// Excerpt from F2837xD_Headers_nonBIOS_cpuX.cmd (X = 1 or 2)
// Peripheral Registers Linker Command File

MEMORY
{
 PAGE 0:    /* Program Memory */

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

SECTIONS
{
...
   XintRegsFile          : > XINT          PAGE = 1
...
}
```

Table 2-16 provides the start and end addresses in memory that correspond to any of the 16-bit registers that are accessible through the structure variable `XintRegs`; this memory block starts at `0x7070` and ends at `0x707F`. By writing to or reading from the structure variable `XintRegs`, the programmer's software interacts in a convenient way with all the hardware registers associated with the external interrupt peripheral module.

The complete programming framework we will be utilizing includes these basic features for every peripheral present on our F28379D 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 2018

Lab #1: Introduction to Microcontrollers

Instructor Verification Page

Lab Section	Begin Date	End Date
L01, L02	August 28	September 4
L03, L04	August 30	September 6

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 §5.2.2 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.
   (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:** _____