GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Summer 2020

Lab #2: General Purpose Inputs and Outputs

Contents

1 Background Material

1.1 Introductory Comments

The versatility of microcontrollers is a consequence of their ability to combine internal numerical computation with external component interaction to achieve some desired function from a single programmable chip. General-purpose inputs and outputs (GPIO) represent the simplest of the various digital interfacing options. The objective of this lab is to introduce GPIO, including how to initialize and then utilize the GPIO module. You will develop code that leverages the hardware abstraction layer (HAL) in order to interact with the GPIO configuration and data registers in the most straightforward manner, and this skill will be applied to the design of systems that read from external switches[1] and write to external light-emitting diodes.

[1]An external switch will be simulated this semester, since you are accessing the target hardware remotely.

For stand-alone operating capability, data may be stored in on-chip volatile memory but code is preferably stored in on-chip non-volatile memory to avoid the need for an external non-volatile memory and the associated interfacing. Hence, in this lab you will learn about flash memory wait states, the desire to use non-default wait state values to efficiently run a program (i.e. at the highest possible execution speed) directly from flash memory, and the need to copy some instructions from flash memory to RAM memory after the boot process to configure the flash memory.

1.2 Relevant Microcontroller Documentation

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

- F28379D Launch Pad Schematic
  - Note how "wire names" are used to label the printed circuit board's copper traces that establish electrical connections between components mounted on the printed circuit board. The entire circuit is displayed using ten pages of interconnected schematic diagrams, and a single "wire name" may appear on multiple pages to reveal all the components that are connected at a single node. Our microcontroller chip package has a 337-pin ball grid array, so this one component is distributed across three of the ten pages (pp. 7–9).
- F28379D Datasheet
  - Table 3-1, for a list of features available on the microcontroller.
  - Figures 4-1 through 4-4, for pin assignments on the chip package.
  - Table 4-1, for a list of all internal pin multiplexing options.
- F28379D Technical Reference Manual[2]
  - §7.1, for an overview of the GPIO module (especially Figure 7-1).
  - §7.2, for a discussion of GPIO module initialization.
  - §7.3, for a discussion of GPIO module utilization.
  - §7.9.2, for register field descriptions associated with GPIO initialization.
    - `GP?GMUX?`, `GP?MUX?`, `GP?DIR` and `GP?PUD`
  - §7.9.3, for register field descriptions associated with GPIO *input* utilization.
    - `GP?DAT`
  - §7.9.3, for register field descriptions associated with GPIO *output* utilization.
    - `GP?SET`, `GP?CLEAR` and `GP?TOGGLE`

[2]Throughout this and future lab documents, similar register and field names are distinguished from each other by the symbol `?` which represents a missing letter or number.

1.3 Target Hardware Schematic Diagrams and Data Sheets

When interfacing a microcontroller-equipped circuit board to some external circuitry, it is necessary to examine both device-level documentation and board-level documentation prior to writing your program code. In our case, the device-level documentation consists of the F28379D Datasheet, Technical Reference Manual and Errata documents, whereas the board-level documentation consists of the F28379D Launch Pad Schematic and the data sheets of any critical components located on that board (aside from the microcontroller device).

- *Device-level documentation.* We are using the F28379D microcontroller component with the 337-pin ball grid array package option (ZWT), and pin assignments for this package option are shown in Figures 4-1 through 4-4 of the Datasheet document. The ball grid array is organized by rows (labeled by letters) and columns (labeled by numbers). Many pins on the package are labeled as GPIO pins, but these may be used for other purposes as well. For example, Figure 4-4 identifies pin C8 (row C, column 8) by the label GPIO0 whereas Table 4-1 indicates that this pin may serve either as GPIO0 (GPIO module, pin 0) or EPWM1A (PWM module 1, pin A) or SDAA (I2C module A, serial data pin), depending on how the multiplexer registers are configured. This means that device pin C8 may be used for any one of three distinct purposes, with code lines located prior to the `while(1)` loop determining which of three internal circuits will be electrically connected to that device pin. Other pins on the device offer even more flexibility to the designer; pin N17 is labeled GPIO58, but the multiplexer presents options for 8 different internal connections.

- *Board-level documentation.* Unlike the simple circuit diagrams used to teach circuit analysis in text books, the F28379D Launch Pad Schematic is ten pages long and features electrical connections that span multiple pages. For example, consider page 6 of the schematic diagram, which shows a collection of 80 header pins that allow the Launch Pad to be physically connected to external components or boards called Booster Packs. On page 6, find the wire labeled as GPIO0/PWMOUT1A which connects to pin 10 of header J4, and observe that this same wire appears on page 9 where it enters the F28379D device package at pin C8 which is internally labeled with GPIO0/EPWM1A/SDAA. Therefore, it is now clear that three distinct circuits inside the microcontroller device may be internally connected to device pin C8 and interfaced to circuitry external to the Launch Pad board via pin 10 of header J4. In the above example, the wire label appeared on just two pages of the schematic diagram, but sometimes a wire label appears more frequently; e.g. the wire labeled +3V3 appears on eight pages of the schematic diagram.

- *Switch Interfacing.* The concept of pull-up and pull-down resistors, for GPIO pins multiplexed as *inputs*, is now reviewed. The objective of such resistors is to guarantee that the logic levels presented to internal logic gates are always well defined, i.e. logic 0 (voltages close to 0 V) or logic 1 (voltages close to 3.3 V). A resistor located between a pin and the supply terminal is referred to as a pull-up resistor, whereas a resistor located between a pin and the ground terminal is referred to as a pull-down resistor.

  - *Pull-Up.* In Figure 1, a mechanical switch has been introduced so as to provide a means for establishing a voltage of 0 V at the pin when the switch is closed. In the schematic on the left, the external circuit allows the pin to float at an undetermined voltage if the switch is opened; on the other hand, in the schematic on the right, the external circuit includes a pull-up resistor to impose a voltage of approximately 3.3 V at the pin if the switch is opened. Microcontrollers may provide the option of enabling or disabling internally located pull-up (red) or pull-down (green) resistors, in order to reduce the number of required external components. For the cases under consideration, internal resistance is needed only in the left schematic, and it should be a pull-up resistance.

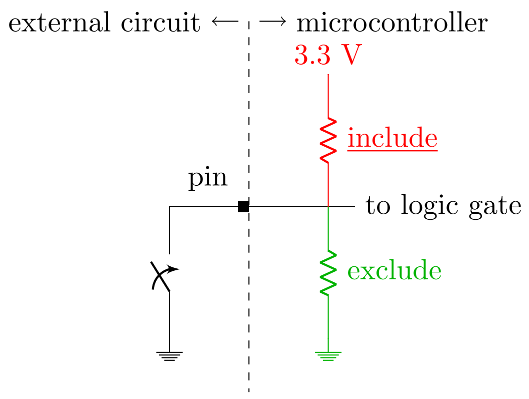
external circuit ← | → microcontroller
3.3 V
include
pin
to logic gate
exclude
external circuit ← | → microcontroller
3.3 V
3.3 V
exclude
pin
to logic gate
exclude

Figure 1: Guidelines for inclusion or exclusion of internal pull-up resistance.

– *Pull-Down*. Now consider Figure 2, in which a mechanical switch has been utilized for the purpose of imposing a voltage of 3.3 V at the pin when the switch is closed. On the left the pin voltage imposed by the external circuit is undefined when the switch is opened, whereas on the right the pin voltage imposed by the external circuit is approximately 0 V when the switch is opened. If it is desired to minimize the number of external components, internal resistance may be used instead of external resistance, in which case the internal resistance should be a pull-down resistance.

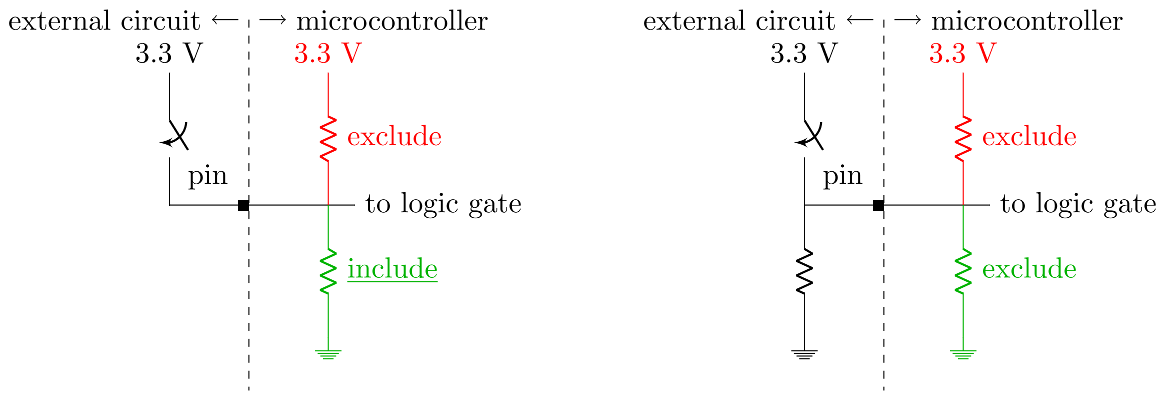

Figure 2: Guidelines for inclusion or exclusion of internal pull-down resistance.

– *Our Microcontroller*. For GPIO pins, our microcontroller offers internal pull-up resistance but not internal pull-down resistance. Hence, if any pull-down resistance is needed for switch interfacing, then such resistance must be added externally (on the printed circuit board) rather than enabled internally (inside the microcontroller device). Our microcontroller's internal pull-up resistance option is actually implemented using transistors rather than resistors, so the effective value of pull-up resistance has not been specified. For GPIO *input* pins, internal pull-up resistance is enabled or disabled by the user as appropriate for switch interfacing. For GPIO *output* pins, internal pull-up resistance would be redundant and hence should be disabled to reduce power consumption.

2 GPIO Module: Step-by-Step Guidelines

2.1 Initialize the GPIO Registers

2.1.1 Set the Multiplexers

To provide maximum design flexibility, many pins on the microcontroller are internally multiplexed. For example, a single pin may serve as a GPIO pin or in support of some other peripheral circuit. The internal circuitry that establishes the desired internal connections to the microcontroller pins is referred to as multiplexer circuitry, and the `GP?GMUX?` and `GP?MUX?` registers are used to set the multiplexers to achieve the desired configuration. The two-bit fields within each of the `GP?GMUX?` and `GP?MUX?` registers determine if the microcontroller pin is internally connected to GPIO circuits or to specific peripheral circuits. The HAL-defined variable names available for initializing the `GP?GMUX?` and `GP?MUX?` registers are as follows:

```
GpioCtrlRegs.GP?GMUX?.bit.GPIO?

GpioCtrlRegs.GP?MUX?.bit.GPIO?
```

Proper assignment of the fields within these registers requires review of the information in §7.7 of Technical Reference Manual.

2.1.2 Assign the Signal Directions

All microcontroller pins set to provide GPIO functionality may be assigned as inputs or outputs, and the `GP?DIR` registers are used for this purpose. The assignment of direction is made by specifying the appropriate one-bit fields within each of the `GP?DIR` registers. The HAL-defined variable names available for initializing the `GP?DIR` registers are as follows:

```
GpioCtrlRegs.GP?DIR.bit.GPIO?
```

Proper assignment of the fields within these registers requires review of the relevant register diagrams and register field descriptions in §7.9 of Technical Reference Manual.

2.1.3 Configure the Pull-Up Resistors

All microcontroller pins configured as GPIO pins may be internally connected to effective pull-up resistors if desired, and the `GP?PUD` registers are used for this purpose. If enabled, the pull-up effect is implemented by a transistor circuit. Enabling or disabling pull-up resistors is achieved by specifying the appropriate one-bit fields within each of the `GP?PUD` registers. The HAL-defined variable names available for initializing the `GP?PUD` registers are as follows:

```
GpioCtrlRegs.GP?PUD.bit.GPIO?
```

Proper assignment of the fields within these registers requires review of the relevant register diagrams and register field descriptions in §7.9 of Technical Reference Manual.

2.2 Utilize the GPIO Pins

2.2.1 Read from a GPIO Input Pin

The `GP?DAT` registers indicate the current state of the GPIO pins (after qualification), as opposed to the state of the output latch associated with these pins, and this is true irrespective of how these pins have been configured (GPIO inputs, GPIO outputs, or some other peripheral functions). Each input-output port has one such register and each bit within these registers corresponds to one GPIO pin. Typically, one reads from the `GP?DAT` registers but does not write to them.

Figure 3 visualizes the signal flow associated with `GP?DAT` registers. The HAL-defined variable names available for accessing `GP?DAT` registers are as follows:

```
GpioDataRegs.GP?DAT.bit.GPIO?
```

The relevant register diagrams and register field descriptions may be found in §7.9 of Technical Reference Manual.

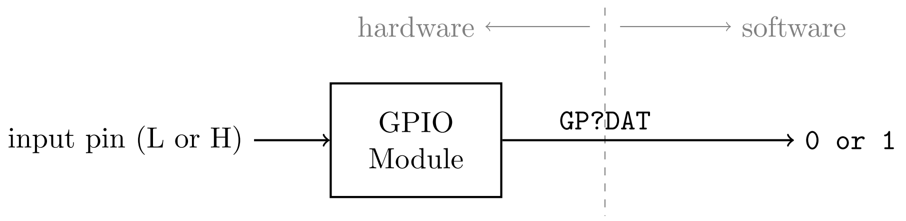

Figure 3: Signal flow associated with `GP?DAT` registers (for reading from pins).

2.2.2 Write to a GPIO Output Pin

The `GP?SET` registers are used to drive specified GPIO pins high without disturbing other pins. Each input-output port has one such register and each bit within these registers corresponds to one GPIO pin. If a GPIO pin is configured as an output, then writing a `1` to the corresponding bit in the `GP?SET` register will set the output latch and the pin will be driven high. If the pin is not configured as a GPIO output, then the value will be latched but the pin will not be driven. Only if the pin is later configured as a GPIO output will the latched value be driven onto the pin. Writing a `0` to any bit in the `GP?SET` registers has no effect.

Figure 4 visualizes the signal flow associated with `GP?SET` registers. The HAL-defined variable names available for manipulating `GP?SET` registers are as follows:

```
GpioDataRegs.GP?SET.bit.GPIO?
```

The relevant register diagrams and register field descriptions may be found in §7.9 of Technical Reference Manual.

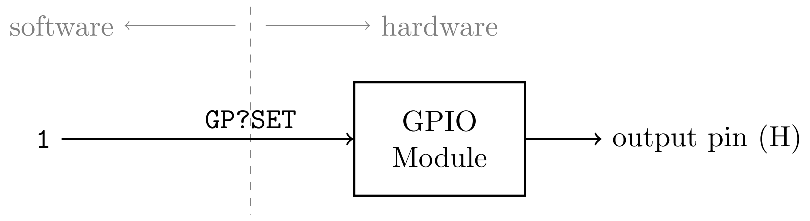

Figure 4: Signal flow associated with `GP?SET` registers (for writing to pins).

The `GP?CLEAR` registers are used to drive specified GPIO pins low without disturbing other pins. Each input-output port has one such register and each bit within these registers corresponds to one GPIO pin. If a GPIO pin is configured as an output, then writing a `1` to the corresponding bit in the `GP?CLEAR` register will clear the output latch and the pin will be driven low. If the pin is not configured as a GPIO output, then the value will be latched but the pin will not be driven. Only if the pin is later configured as a GPIO output will the latched value be driven onto the pin. Writing a `0` to any bit in the `GP?CLEAR` registers has no effect.

Figure 5 visualizes the signal flow associated with `GP?CLEAR` registers. The HAL-defined variable names available for manipulating `GP?CLEAR` registers are as follows:

```
GpioDataRegs.GP?CLEAR.bit.GPIO?
```

The relevant register diagrams and register field descriptions may be found in §7.9 of Technical Reference Manual.

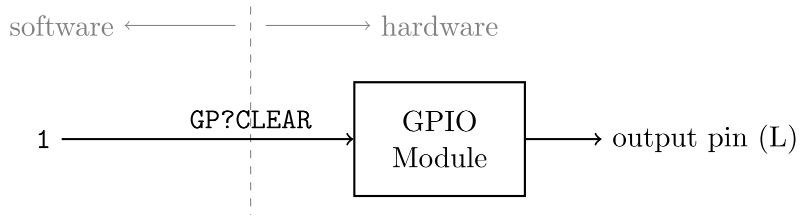

Figure 5: Signal flow associated with `GP?CLEAR` registers (for writing to pins).

The `GP?TOGGLE` registers are used to drive specified GPIO pins to an opposite level without disturbing other pins. Each input-output port has one such register and each bit within these registers corresponds to one GPIO pin. If a GPIO pin is configured as an output, then writing a `1` to the corresponding bit in the `GP?TOGGLE` register flips the output latch and drives the pin as follows: if the output pin is low, then it will be driven high; if the output pin is high, then it will be driven low. If the pin is not configured as a GPIO output, then the value will be latched but the pin will not be driven. Only if the pin is later configured as a GPIO output will the latched value be driven onto the pin. Writing a `0` to any bit in the `GP?TOGGLE` registers has no effect.

Figure 6 visualizes the signal flow associated with `GP?TOGGLE` registers. The HAL-defined variable names available for manipulating `GP?TOGGLE` registers are as follows:

```
GpioDataRegs.GP?TOGGLE.bit.GPIO?
```

The relevant register diagrams and register field descriptions may be found in §7.9 of Technical Reference Manual.

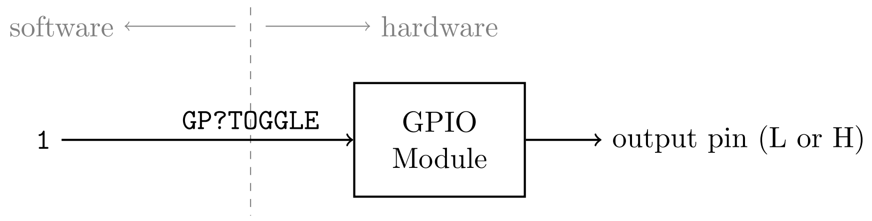

Figure 6: Signal flow associated with `GP?TOGGLE` registers (for writing to pins).

3 Flash-Based Implementation

3.1 Types of Microcontroller Memory

The normal work flow during embedded code development includes working in CCS edit mode to create or modify the `main.c` file and then switching to CCS debug mode to actually run the resulting executable program on the microcontroller. A linker command file—either a default option added by CCS during project creation or a custom option developed by the user and added manually—determines which portions of the microcontroller memory will be used for storing the code (program instructions) and for storing the data (program variables).

Throughout the initial stages of embedded code development, it is recommended to use on-chip RAM memory to store both the code and the data; with this approach, each debug session will be launched more quickly, and you will avoid wearing out the on-chip flash memory by repeated erasing and reprogramming. One default linker command file that may be added to our CCS projects is `2837x_RAM_lnk_cpu1.cmd` and, as its name suggests, the corresponding type of debug session is one in which on-chip RAM memory is the only type of memory being used. However, since RAM is volatile memory which does not retain its contents after power is removed, this type of debug session does not actually prepare the microcontroller for deployment in an embedded stand-alone product. Fortunately, the F28379D microcontroller has internal flash memory, a type of non-volatile memory, in which to store application code to enable true stand-alone operation.

3.2 Flash Programming Procedure

Assuming that you have developed a RAM-based project that you are satisfied with, you can convert it to a flash-based project for stand-alone operation by doing the following:[3]

1. Delete the RAM-based linker command file, e.g. the TI provided `2837x_RAM_lnk_cpu1.cmd` or some customized `user_RAM.cmd` as appropriate.
2. Add a flash-based linker command file, e.g. the TI provided `2837x_FLASH_lnk_cpu1.cmd` or some customized `user_FLASH.cmd` as appropriate.
3. Add the TI provided file `F2837xD_CodeStartBranch.asm` to enable the boot loader to automatically begin running your flash-based program immediately after power up.
4. Build the flash-based project and enter the debug mode in order to automatically initiate the process of programming the flash memory.
5. Once programming is complete, terminate the debug session without running code.
6. Disconnect target hardware from development system to begin stand-alone operation.

The purpose of initiating a debug session for the flash-based project is simply to program the flash memory inside the microcontroller, and this activity will occur automatically when the debug session is initiated. You will see messages indicating that the flash memory is being erased, programmed and verified. Once the flash memory has been properly prepared, no more messages will be displayed and the debug session may be terminated. At this point, you can simply disconnect your target hardware from the development system and then use it as a stand-alone embedded component.

The procedure listed above is sufficient for applications that require programming the flash memory just once, but there are other applications that require programming the flash memory at

[3]Right click on a project file (e.g. a `cmd` file) and select `Exclude from Build` to toggle project build options.

run time after the system has been put into use (e.g. to save some critical parameters each time the system is turned off). In that type of application, which is beyond the scope of this lab, CCS is not available to assist with flash memory programming, so the code developer will need to make explicit use of a TI-provided flash API library.

3.3 Flash Wait States

The use of on-chip flash memory is certainly a convenient way to achieve stand-alone operating capability. Its use eliminates any need for external non-volatile memory or a host processor from which to boot-load the application code. On the other hand, flash memory is not as fast as RAM memory; after a device reset, flash memory accesses will require 15 wait states by default compared to RAM memory accesses which require 0 wait states. These wait states may be thought of as "wasted" clock cycles that slow down program execution. In some applications, these wait states and resulting slower operation may not be a problem. For those applications that cannot afford to waste any clock cycles, there exist two alternatives.

1. One approach would be to configure the flash control registers to reduce the number of wait states to a (typically nonzero) minimum, and then to enable the flash cache/prefetch mode.
2. Another approach is to copy time-critical portions of application code, or even all application code, from flash to RAM prior to execution at run time (if there is enough on-chip RAM).

In this lab, we consider the first of these approaches which is described below.

3.4 Flash Configuration

Table 5-19 of DATASHEET specifies the minimum number of flash wait states in relation to the system clock frequency; lower clock frequencies permit 0 wait states, whereas higher clock frequencies require up to 3 wait states. Figure 2-287 of TECHNICAL REFERENCE MANUAL indicates that there will be 15 flash wait states by default. In this lab, we will configure the flash to operate with 3 wait states to cover the worst case situation; even with this conservative choice, we will be reducing the number of wait states from the default value of 15 to 3, resulting in code that executes from flash five-times faster than would have been the case otherwise.

As pointed out above, flash memory requires wait states whereas RAM memory does not; moreover, after reset the default number of flash wait states is 15, whereas for a 200 MHz system clock frequency the minimum number of flash wait states is just 3. Although there are flash configuration registers that allow a user to reduce the number of flash wait states and enable the flash cache/prefetch mode, it is not possible to modify these registers from code that itself executes from flash memory. According to §2.12.3 of TECHNICAL REFERENCE MANUAL:

> User application software must initialize wait-states using the `FRDCNTL` register, and configure cache/prefetch features using the `RD_INTF_CTRL` register, to achieve optimum system performance. Software that configures flash settings like wait-states, cache/prefetch features, and so on, must be executed only from RAM memory, *not* from flash memory.

Therefore, the solution is to first copy the appropriate configuration code from flash to RAM and then execute this code from RAM, all prior to initiating execution of the primary application code from the re-configured flash. The details are discussed in §4.4 of TI Application Report SPRA958L; the required functionality is achieved by following the step-by-step instructions listed below.

1. Include the following code in the preamble of your `main.c` file (i.e. prior to function `main`).

```
#pragma CODE_SECTION(InitFlash, "RamFuncs")
void InitFlash(void)
{
    Flash0CtrlRegs.FPAC1.bit.PMPPWR = 0x1;
    Flash0CtrlRegs.FBFALLBACK.bit.BNKPWR0 = 0x3;
    Flash0CtrlRegs.FRD_INTF_CTRL.bit.DATA_CACHE_EN = 0;
    Flash0CtrlRegs.FRD_INTF_CTRL.bit.PREFETCH_EN = 0;
    Flash0CtrlRegs.FRDCNTL.bit.RWAIT = 0x3;
    Flash0CtrlRegs.FRD_INTF_CTRL.bit.DATA_CACHE_EN = 1;
    Flash0CtrlRegs.FRD_INTF_CTRL.bit.PREFETCH_EN = 1;
    Flash0EccRegs.ECC_ENABLE.bit.ENABLE = 0xA;
    asm(" RPT #6 || NOP");
}

extern Uint16 RamFuncs_loadstart;
extern Uint16 RamFuncs_loadsize;
extern Uint16 RamFuncs_runstart;
```

2. Include the following code inside function `main` but prior to `while(1)`, e.g. just after the watchdog disable step.

   ```
   memcpy(&RamFuncs_runstart, &RamFuncs_loadstart, (Uint32) &RamFuncs_loadsize);
   InitFlash();
   ```

3. Follow the procedure outlined in §3.2, using the files

   ```
   user_FLASH.cmd
   F2837xD_CodeStartBranch.asm
   ```

   that have been uploaded to canvas.

4 Lab Assignment

4.1 Preparation

Before attempting this lab project, do the following:

1. Read through this entire document.
2. Study what is connected to microcontroller pins using target hardware circuit diagrams.
3. Determine how the appropriate registers will be used to complete the tasks assigned in §4.2. Focus on those HAL variables identified in §2, and consult appropriate circuit diagrams.

4.2 Tasks

For both tasks below, place the following code lines inside function `main` but prior to `while(1)`, e.g. just after the watchdog disable step, in order to dictate 10 MHz clocking.[4]

[4]The motivation here is that some CCS actions at the launch of debug sessions will lead to unpredictable non-default clock frequencies, unless user code explicitly assigns specific values to these registers.

```
ClkCfgRegs.SYSPLLMULT.all = 0;
ClkCfgRegs.SYSCLKDIVSEL.all = 0;
```

As in Lab 1 (and all future labs), disable the watchdog before configuring your device, enable the watchdog after configuring your device, and service the watchdog in the `while(1)` loop.

4.2.1 Monitor the Status of a Pushbutton

To simulate the presence of an external ground-referenced pushbutton connected to the header pin corresponding to `GPIO0` (as shown on the left side of Figure 1), place the following code lines inside function `main` but prior to `while(1)`, e.g. just after the watchdog disable step:

```
GpioCtrlRegs.GPAGMUX1.bit.GPIO0 = 0;
GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;
GpioCtrlRegs.GPAODR.bit.GPIO0 = 1; // explain what this does
GpioCtrlRegs.GPAPUD.bit.GPIO0 = 1; // consider both 1 and 0
```

Develop code that assigns the status of this simulated external pushbutton. Implement this feature by declaring a variable `output` of type `Uint16`, such that `output = 0` corresponds to the pressed state and `output = 1` corresponds to the unpressed state; for this purpose, assign values to

`GpioDataRegs.GPACLEAR.bit.GPIO0` or `GpioDataRegs.GPASET.bit.GPIO0`

in the `while(1)` loop as appropriate. Display `output` in the Expressions Window, and modify its value in that window as desired to simulate active use of the pushbutton during a debug session. Monitor the influence of the simulated pushbutton on the microcontroller pin by displaying

`GpioDataRegs.GPADAT.bit.GPIO0`

in the Expressions Window. Demonstrate how the pull-up setting determines if the microcontroller does, or does not, successfully detect simulated pushbutton presses. Use linker command file `user_RAM.cmd` in order to run your code from RAM.

4.2.2 Toggle an LED to Indicate System Status

Develop code that toggles the red LED once every 10^5 executions of the `while(1)` loop, to visually indicate that your program is running as intended; the blue LED should remain off at all times. Since you cannot see the LEDs, use the Expressions Window to reveal their behavior. Use linker command file `user_FLASH.cmd` in order to run your code from flash; run your code twice—first with the default number of flash wait states (15) by excluding function call `InitFlash()`, and second with the minimum acceptable number of flash wait states (3) by including function call `InitFlash()`—in order to observe the differences in execution speed.

4.3 Deliverables

To be eligible for full credit, do the following:

1. Upload narrated videos showing completion of §4.2.1 and §4.2.2.

2. Upload your `main.c` file for §4.2.2.