EK307 – Microcontrollers Lab

Background

The purpose of this lab is to provide you with a basic understanding of microcontrollers. The microcontroller is a small computational device built on a single integrated-circuit “chip.” The microcontrollers has become so commonplace that we are often unaware of its presence. Microcontrollers are found everywhere: in tablets, computers, automobiles, airplanes, cell phones, toys, medical implants, kitchen appliances, wind turbines, solar panels, security systems, lighting systems, robots, and a host of other devices and systems. A microcontroller is used whenever localized digital control is required. For example, the keyboard on your laptop contains its own dedicated microcontroller whose sole function is to interpret keystrokes and send them to the laptop’s main processor. An optical or touchpad mouse also contains a microcontroller that monitors movement and button clicks, converting them into a signal that is then sent to the main computer. Complex products, such as cars or airplanes, incorporate many microcontrollers, wherein each one performs one specific function. In an automobile, for example, separate microcontrollers are used for engine monitoring, fuel injection, accelerator-pedal control, tire-pressure sensing, cruise control, ABS breaking, emissions control, GPS integration, entertainment system coordination – and that’s just for a simple family sedan. A modern aircraft contains hundreds of microcontrollers that each perform different functions.

There are literally hundreds of different kinds of microcontroller chips made by dozens of manufacturers. In EK307, will use the popular MSP430 microcontroller made by Texas Instruments to examine various digital operations. In the lab, and MSP430 chip is mounted on a test and development board called the LaunchPad, where the latter is very similar to the popular Arduino controller. The LaunchPad is designed so that you can program the microcontroller, and then take the chip out and install it on a breadboard or printed-circuit board of your choosing. (You may purchase your own LaunchPad if you wish.)
The MSP430 chip “thinks” in binary machine code (instructions are all sequences of 1 and 0 bits,) but it is programmed in the computer language C using a program called Code ComposerStudio. This software has been installed on the computers in PHO105, but you may also download your own personal copy of Code Composer (PC or MAC) from http://processors.wiki.ti.com/index.php/Download_CCS

**Simplified Description of the MSP430-G-2553**

The various chips in the MSP430 family of microcontrollers are distinguished by a letter (e.g., “G series”) followed by four numbers. In EK307, we will focus on the MSP430-G2553, a basic level member of the MPS430 family, in conjunction with the LaunchPad test and development board (shown below to the right). As is the case for all microcontrollers, the “2553” operates in a binary world wherein a “high” voltage (between about 3 to 3.3 V) represents a Logic 1, and a low voltage (about 0 V) represents a Logic 0.

The ‘2553’ has twenty pins. Of these, two are reserved for power (VCC = 3.3 V) and ground (GND = 0 V). One pin is used as a RESET pin (connected by default to VCC via a 1-kΩ resistor), and one is a programming TEST pin (which we will not use in this class).

Of the remaining sixteen pins, eight comprise an 8-bit input/output (I/O) port labeled P1, and another eight comprise an 8-bit I/O port labeled P2. The bits of Port 1 are consecutively numbered P1.0 through P1.7, while those of Port 2 numbered P2.0 through P2.7. A pin-out diagram of the 2553 DIP chip is shown below to the left. Note the “half-moon” notch which locates the pin-1/pin-20 end of the chip:

---

1. In digital systems, the symbol Ø is sometimes used to represent Logic 0, so that it will not be confused with the letter O.

2. You do not need to worry about connecting the programming-pin resistor, VCC, or ground when using the USB-connected LaunchPad programmer. These connections are made automatically on the LaunchPad. If you remove a programmed MSP430 chip from the Launchpad to install in your own breadboard circuit (for example, as part of your final project), will you need to make these connections on your own.
The 2553 performs tasks in binary machine language (e.g., “assembly code”), but you can program it with C-language commands using the Code Composer Studio program which converts C code into MSP430 machine code via a process called compiling. Code Composer also allows you to download compiled machine-code programs to the microcontroller via the USB connection between the LaunchPad and a PC.

Note: You can take the actual microcontroller chip out of its LaunchPad socket and plug it directly into your breadboard. The MSP430 must be powered from a 3.3-V source with its RESET pin tied to VCC. [Important: Do not power the MSP430 with a VCC higher than about 3.6 V or it will be damaged.]

The various pins of the microcontroller are brought to pins on the sides of the Launchpad development and programming board, as shown in the diagram to the right. For ease in testing, debugging, and eventual integration onto your breadboard, you can use the “rainbow” jumpers that come with your EK307 Lab Kit to make connections between the LaunchPad and your breadboard. Note that you can separate the jumper strip into individual wires if you wish, or you can leave them as a flat
Programming the MSP430

The MSP430 chip recognizes numerous commands that tell it what to do with digital data collected from (or sent to) its input/output (I/O) pins. The 2553 can also perform various programmable logic functions such as AND, OR, if statements, etc. Using C code, each of the ports is written to or read from using decimal (base-10) numbers to map to individual port pins. Suppose, for example, that Port 1 is set up so that all of its pins are input bits, where the logic levels (3.3 V and 0 V) are sent from a circuit on your breadboard. Further suppose that the program inside the 2553 executes a command to read Port 1 and set the value of an integer variable \( n \) to the data received from the port. The value of \( n \) will be determined by the eight individual bits read from Port 1. The data bits, starting from bit P1.0 (the LSB) and proceeding to bit P1.7 (the MSB), where each bit carries a weight of increasing powers of 2. Here LSB is the “least significant bit” (smallest weight) and MSB is the “most significant bit” (largest weight):

\[
\begin{array}{cccccccc}
\text{PORT BIT:} & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\text{MSB} & & & & & & & & \\
\text{Carries decimal weight:} & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\end{array}
\]

The value of \( n \) is determined by adding together the weights of all those bits that are set to Logic 1. Thus, for example, if the MSP430 reads Port 1, and its bits have been set to 1000 1001

\[
\begin{array}{cccccccc}
\text{weight:} & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\text{MSB} & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
\text{Sum of decimal weights:} & 128 + 0 + 0 + 0 + 8 + 0 + 0 + 1 = 137 \\
\end{array}
\]

By the circuit to which the microcontroller is connected (see figure above), then the value of \( n \) read in from Port 1 will be 128 + 8 + 1 = 137, because a Logic 1 is found in the 128, 8, and 1 positions of the port.
Ports 1 and 2 each have eight pins, hence the minimum and maximum values of \( n \) that can be read from (or sent to) these ports are, respectively, \( 0 = 0000\,0000 \) and \( 255 = 1111\,1111 \).

**Setup Commands**

To program the MSP430, you must write and compile C-code using the *Code Composer Studio* program. Various statements tell the MSP430 what to do and how to behave. Many common commands have been pre-written for you and are explained in more detail the program *ek307starter_code.c* available on the EK307 lab website: sites.bu.edu/engcourses/ek307 or Blackboard: learn.bu.edu. Here several examples are provided below, but the starter code has many more.

**DIR Command**

The DIR command tells the MSP430 whether a given port pin is to be used for **input** or **output**. Specifically, if a bit in the DIR command is a logic 1, then the corresponding port pin will be set to be an **output**. Conversely, if that same bit pin in the DIR command is a logic 0, then the corresponding port pin is set to be an **input**. The input or output status of a port can be set on a pin-by-pin basis and can be changed any time in your program.

Here are some examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1DIR = 0</td>
<td>Set all bits of Port 1 so that they receive (input) data</td>
</tr>
<tr>
<td>P1DIR = 255</td>
<td>Set all bits of Port 1 so that they send (output) data</td>
</tr>
<tr>
<td>P2DIR = 0</td>
<td>Set the two bits of Port 2 so that they receive (input) data</td>
</tr>
</tbody>
</table>

As explained above, the DIR command functions on a bit-by-bit (i.e., pin-by-pin) basis. Thus some of the pins of a given port can be set to inputs, and others as outputs. For example, the decimal number 64 corresponds to the binary number 0100 0000. Thus, the command

\[
P1DIR = 64 \quad \text{(equivalent to } P1DIR = 0100\,0000)\]

sets Port 1 so that P1. 6 is an output, and all other Port 1 pins are inputs.
Likewise, the decimal number 65 corresponds to the binary number 0100 0001. Thus, the command

\[
P1DIR = 65 \quad \text{(equivalent to P1DIR = 0100 0001)}
\]

sets Port 1 so that P1.6 and P1.0 are outputs, and all other Port 1 pins are inputs. Similarly,

\[
P2DIR = 128 \quad \text{(equivalent to P2DIR = 1000 0000)}
\]

sets P2.7 to be an output, with all other pins as inputs as an input, and

\[
P2DIR = 64
\]

The either the command:

\[
P2DIR = 128 + 64 \quad \text{(equivalent to P2DIR = 1100 0000)}
\]

or the command:

\[
P2DIR = 192 \quad \text{(because 128 + 64 = 192, these statements are the same)}
\]

Sets P2.7 and P2.6 as output pins, and the rest of the Port 2 pins as inputs.

**PIN Command**

The PIN command tells the MSP430 to read data from one of its ports. The data pins are set to either 1 or 0 (Vcc or GND) by the circuit or connections external to the chip (or LaunchPad). In the following example, suppose that \( n \) is a variable declared in the C program:

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = \text{P1IN} )</td>
<td>Sets the integer variable ( n ) to the bit values applied to Port 1 or Port 2 by an external circuit connected to Port 1 (e.g., by a circuit on your breadboard). Only the pins designated as inputs via the PDIR command are read.</td>
</tr>
</tbody>
</table>

**Select Command**

The SELECT command is unique to the MSP430. It tells the chip to use pins P2.6 and P2.7 as input output (I/O) pins, rather than as crystal-clock inputs\(^3\). The following command must always be in your program if you want to use Port 2:

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
</table>

---

\(^3\) We do not run the MSP430 with a clock crystal. The latter makes its timing ultra-precise in certain applications.
P2SEL=0 
Tells the MSP430 that P2.6 and P2.7 are to be used as I/O pins.

**Procedures in C**
While commands tell the MSP430 how to set up its hardware and ports, procedures tell the MSP430 how to process data. The file *ek307starter_code.c* provides many procedures (and commands) that have been prewritten and preceded by “comment” bars //. To use any line of the starter code, simply remove the // to make the statement active. You can also modify any lines, or write your own. Just observe these basic rules:

1. A procedure, which can consist of multiple lines, is enclosed within squirrelly brackets { .... }
2. Every line of code must end with a semicolon ;
3. Any integers that you use in the program must be declared at the start of the program, e.g.
   ```c
   int n;  // declares n as an integer whose value you will set later in the program
   int n=0;  // declares n as an integer with an initial value of 0 that can be changed later by the program.
   ```

**Conditional Procedures**
Two very useful constructs are the logical *if* and *while* statements which have the following syntax:

```c
if (condition) { Do the commands inside the brackets that follow one time if “condition” is true;}
while (condition) { Do the commands inside the brackets that follow repeatedly as long as “condition” is true;}
```

**Notes:**
- In C code, the binary digit 1 is interpreted as a “true” by *if* and *while* statements. The digit 0 equals “false”.
- For a *while* procedure, the loop will go on forever if the condition contained in the brackets { } never changes to “false”.

**Example** - The following code sequences the integer n from 0 to 5, and then returns it n = 0:
```c
int n=0; // sets n to an integer with an initial value of 0.
while(1) // Because 1 equates to “true”, the bracketed procedure that
    // follows will go on indefinitely.
{
    // Here is the bracket that begins the while procedure.
    if (n<6) {n = n+1;} // Increments n by 1 if n is less than 6.
    if (n==6) {n=0;} // Resets n to 0 if it reaches the value 6.
    // Note that an “equals” sign that is part of an if statement is a double “==”
}
    // Here is the bracket that ends the while procedure.
```

### Programming and Running the MSP430 Using Code Composer

To setup, compile, and run your code, follow this procedure exactly:

- Plug the LaunchPad into the PC using the USB cable provided.

**First time only:**

1. Log on to your kerberos account on the PC at your lab station.
3. Make a new folder on the X: drive called “MSP Folder” (or whatever else you choose to call it).
4. Download `ek307starter_code .c` from the course website and save it in X:MSP Folder

**Open Code Composer**

5. From the START menu in Windows, find and open the TI folder, where you should click on the icon for Code Composer. You may choose to create a shortcut of this icon on your desktop.

**From the Code Composer pull-down menu:**

6. Select *New CSS Project*
7. Give your project a name such as *MyLab* (or whatever you want.)
8. Follow the prompts and steps, but be sure to choose
   a. Project Type: MSP430
   b. Output Type: Executable
   c. Project Settings: MSP430G2553
d. Everything else: Choose the default settings

9. IMPORTANT: Set the Default Folder in Code Composer to X:MSP Folder (or whatever you may have called it) from Step 3.

10. Drag-n-drop the *ek307starter_code .c* program from your desktop into the MyLab icon on the left-hand column of Code Composer. This action will cause Code Composer to generate all the needed auxiliary files.
Designing and Testing Your Microcontroller Program:

1. Modify, rewrite, or otherwise suitably alter *ek307starter_code.c*, then save it.

2. Click on “Build Active Project” (first time) or “Rebuild Active Project” (thereafter) from the pull-down menu. This action will compile your C-code.

3. Click on the “Debug” icon from the pull-down menu. This action will load your compiled machine code into the 2553 chip mounted in the socket on the LaunchPad.

4. Click on “Run” to execute and test your code on the 2553 via LaunchPad. Click “Stop” to halt execution.
Activities: Learning to Use the MSP430 Microcontroller

Your goal is to experiment with the MSP430 so that you can master its operation. For Level 1, do an exercise that is labeled as a Level 1 challenge. To get credit for Level 2, do an exercise that is labeled as a Level 2 challenge.

The MSP430 circuit boards are available for loan at the service window in the lab.

Exercise #1 – LED Bar Graph: (LEVEL 1)

Write a program that does the following:

- Set up Port 1 as an output port and Port 2 as an input port.
- Read a 3-digit binary input \( n \) from Port 2 (e.g., pins P2.0 – P2.2). Using rainbow jumpers, set the Port 2 pins to either high (3.3 V) or low (0 V) by connecting them to \( V_{CC} \) or GND. Because \( n \) will be a 3-bit binary number, the decimal value of \( n \) can thus be a decimal number between 0 and 7 (i.e., \( n \) will have eight possible values, depending on the inputs read by Port2.
- Drive the eight pins of Port 1 (P1.0 through P1.7) so that, when a given bit is in the \( 1 \) state, it will light an LED that has been connected between the pin and ground. Your eight LEDs should light up according to the chart shown to the below.

<table>
<thead>
<tr>
<th>Port 2 Input</th>
<th>Port 1 LEDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>●●●●●●●●●●</td>
</tr>
<tr>
<td>1</td>
<td>●●●●●●●●●●</td>
</tr>
<tr>
<td>2</td>
<td>●●●●●●●●●●</td>
</tr>
<tr>
<td>3</td>
<td>●●●●●●●●●●</td>
</tr>
<tr>
<td>4</td>
<td>●●●●●●●●●●</td>
</tr>
<tr>
<td>5</td>
<td>●●●●●●●●●●</td>
</tr>
<tr>
<td>6</td>
<td>●●●●●●●●●●</td>
</tr>
<tr>
<td>7</td>
<td>●●●●●●●●●●</td>
</tr>
<tr>
<td>8</td>
<td>●●●●●●●●●●</td>
</tr>
</tbody>
</table>
Exercise #2 – Best of Seven (LEVEL 1)
Configure Port 1 as in input port, and Port 2 as an output port. Connect LEDs (in series with 1-kΩ current limiting resistors) to any two pins of Port 2. Call these LED1 and LED2. Write a program that reads the first seven bits of Port 1 (P1.0 through P1.6), then sets the LEDs connected to Port 2 to indicate the majority status of Port 1. Specifically:

- Majority of Port 1 bits high – LED1 on, LED2 off.
- Majority of Port 1 bits low – LED1 off, LED2 on.

Exercise #3 – Motor Speed Controller (LEVEL 2)
Connect a motor and a PN2222 transistor (available from the instrument window) using the circuit shown below. The 2N2222 transistor operates as a binary switch that either connects the motor between \( V_{DC} \) and ground (logic 1 output from the MSP430) or disconnects the motor (logic 0 output from the MSP430).

![Circuit diagram of motor speed controller](image)

The speed of the motor will depend on the average dc value of the voltage applied to it. In this case, we can alter the average DC using the technique of *pulse-width modulation*.

If the digital signal sent to the 2N2222 transistor has a regular periodicity \( T_{tot} \) but a variable on time \( T_{on} \), as illustrated below, the “duty cycle”, defined by \( D = T_{on}/T_{tot} \), will be the fraction of time that the signal is high. In such a case, the average value of voltage will be \( DV_{DC} \).

\[ DV_{DC} = \frac{T_{on}}{T_{tot}} \times V_{DC} \]

\[ \text{Duty Cycle} = \left( \frac{T_{on}}{T_{tot}} \right) \times 100\% \]

** The duty cycle is sometimes expressed as a percentage of on time: Duty Cycle = (\( T_{on} / T_{tot} \)) \times 100\%
Write a program that will set the motor to different speeds (at least four) depending on the high/low status of the Port 1 inputs. For example, your settings might yield the following result:

<table>
<thead>
<tr>
<th>Port 1 Bits</th>
<th>Duty Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>25%</td>
</tr>
<tr>
<td>0010</td>
<td>50%</td>
</tr>
<tr>
<td>0100</td>
<td>75%</td>
</tr>
<tr>
<td>1000</td>
<td>100%</td>
</tr>
</tbody>
</table>

A reasonable value for $T$ might be 128 ms but, for the motor provided, it could be anything under about a half second.

Exercise #4 – Morse Code SOS Generator (LEVEL 2)
Write a program that will blink an LED and sound a loudspeaker according to the universally-known distress signal SOS: • • • – – – • • •