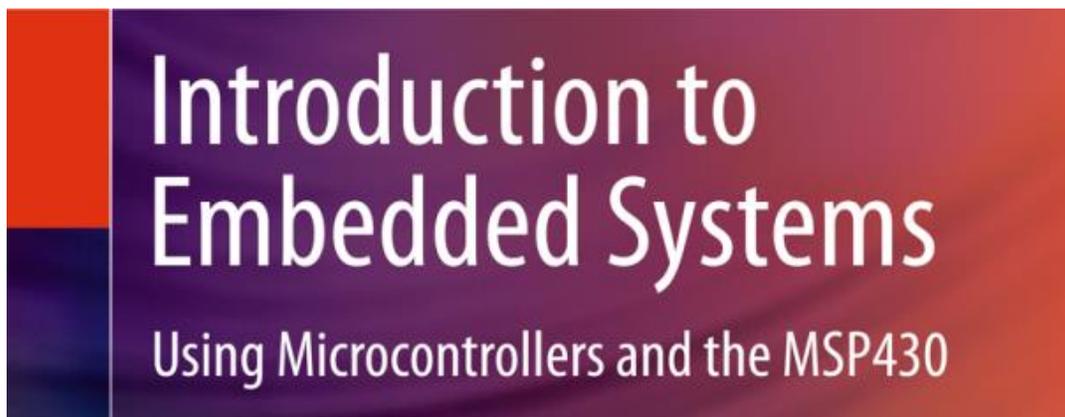


EXPERIMENT'S MANUAL

Introductory Experiments

To accompany the textbook:



By

Manuel Jiménez, Rogelio Palomera, and Isidoro Couvertier

Electrical and Computer Engineering Department
University of Puerto Rico at Mayaguez

2014

© 2014 by M. Jiménez, R. Palomera, I. Couvertier
Electrical and Computer Engineering Department
University of Puerto Rico at Mayaguez

ACKNOWLEDGMENT

The authors would like to thank Cesar Aceros for his valuable help developing the LaTeX template used for preparing this manual.

DISCLAIMER

Although the authors have made every effort to verify their correctness of this Experiment's Manual, the materials contained herein are provided "as is". Any express or implied warranties, including, but not limited to, the implied warranties of fitness for any particular purpose are disclaimed. Under no circumstance or event shall the authors or the copyright owners be liable for any direct, indirect, incidental, exemplary, or consequential damages arising from the use of these materials.

Contents

| | |
|---|----|
| Experiment 1 | |
| Introduction to MSP430 Tools | 1 |
| Experiment 2 | |
| MSP430 Instruction Set Part 1 | 19 |
| Experiment 3 | |
| MSP430 Instruction Set Part 2 | 27 |
| Experiment 4 | |
| MSP430 Poll-based I/O | 32 |
| Experiment 5 | |
| MSP430 Interrupt-based I/O | 36 |
| Experiment 6 | |
| MSP430 Timer A | 42 |
| Experiment 7 | |
| MSP430 Interrupt-based I/O using the C Language | 47 |
| Experiment 8 | |
| The MSP430X..... | 54 |

Experiment 1

Introduction to MSP430 Tools

Objectives

- Become familiar with the MSP-EXP430G2 LaunchPad board and its basic components.
- Understand the basic anatomy of an assembly language program.
- Become familiar with the process of assembling, uploading, debugging, and executing an assembly language program using IAR Embedded Workbench (IAR)
- Learn how to examine and modify MSP430 memory and register contents.

Duration

2 Hours

Materials

- IAR Embedded Workbench (IAR) Application.
- MSP-EXP430G2 LaunchPad development board.

1.1 Introduction

The MSP430 IAR Embedded Workbench is IAR's Integrated Development Environment (IDE) for the MSP430. It reduces the development time and optimizes the performance for MSP430 applications. For all the upcoming experiments IAR will be used as the compiler, assembler, linker, and code debugger for the MSP430 LaunchPad.

The MSP-EXP430G2 LaunchPad is a complete MSP430 development platform. It includes all the hardware and software components to evaluate the MSP430 and to develop a complete project in a convenient board. The module is divided into two sections: a USB communications interface with an on-board chip programmer/debugger interface and an MSP430 target board. Figure 1.1 below shows a LaunchPad board, with a 14-pin MSP430G2211 included, where each section can be identified. The dotted line beneath the “Emulation” label separates both sections.

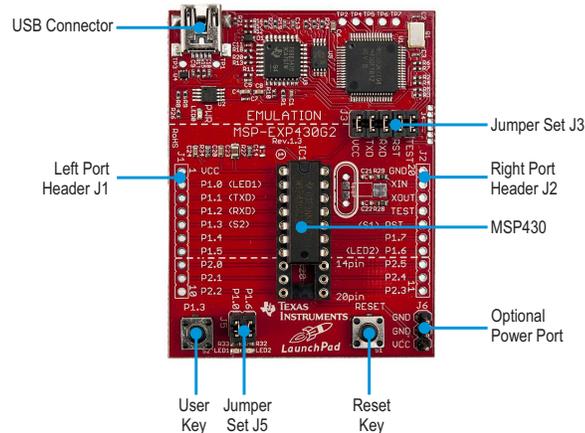


Figure 1.1: EXP430G2 Launchpad.

The integrated USB-based emulator section offers all the hardware and software necessary to develop applications for all MSP430G2xx series devices. The USB interface also provides power to operate the MSP430 when connected to a USB terminal at a computer. In addition, it generates the signals to program and debug code in the MSP430 memory.

The male header connector J6 provides access for external power if desired. It is important to understand that you must never connect together any of the VCC and GND exposed parts or terminals in the J6 connector. You should also make sure that the USB cable male connectors are correctly inserted: the mini connector inside the MSP-EXP430G2 USB female connector and the regular connector inside the appropriate USB connector in your computer, otherwise you may get some strange errors when trying to download and debug as part of the rest of this experiment.

The MSP430 section has an integrated DIP target socket that supports up to 20 pins. The Launchpad supports all MSP430G2xx and MSP430F20xx devices in PDIP14 or PDIP20 packages. As a bonus for enthusiasts, the LaunchPad experimenter board is capable of programming the eZ430-RF2500T target boards, the eZ430-Chronos watch module or the eZ430-F2012T/F2013T target boards.

The MSP-EXP430G2 Launchpad packages comes with two fully functional MSP430 microcontrollers with over and a target board providing 20 user accessible pins for extended experiments, as well as several LED indicators and push-buttons. Early packages came with MSP430G2211 and MSP430G2231 units ; the newest 1.5 version comes with a MSP430G2553 and a MSP430G2452.

Let us take a quick tour of the MSP430G2 LaunchPad. In the upper left corner of the board we find a USB female connector. To the right of the “MSP-EXP430G2” label is jumper set J3 with five (5) removable jumpers in place. This set constitutes the Spy-Bi-Wire and MSP430 application UART. Jumper set J3 has five (5) jumpers in place: TEST, RST, RXD, TXD, and VCC, while jumper set J5 has two (2): P1.0 and P1.6.

In the center of the lower region we have socket IC1 where the target MSP430 microcontroller resides. Look at the position and orientation of the MSP430 chip on the socket. Please be aware that you could have a different MSP430 model from the one shown in the above in the picture. This one is a 14-pin dual-in-line (dip) MSP430 chip, but you could have a 20-pin dip chip. Regardless of which one you have, note that the chip has a notch resembling a U on one end and a recessed full circle on the other end. The chip must be inserted with the U pointing towards the label MSP-EXP430G2 in such a way that the pin to the left is inserted in the first available position on the left side of the socket while the circle must be pointing towards the label “Texas Instruments”.

You should have received the LaunchPad with the MSP430 chip correctly inserted in the socket. If for any reason the chip is not inserted as explained above, you should carefully remove it and reinsert it in the socket after making sure that it is properly oriented and aligned. If the MSP430 is not appropriately inserted, then the board will malfunction and the MCU chip might be damaged.

In the picture, the chip is an MSP430G2231, but, as noted before, you may be using a different one. You need to know exactly which chip version you have for the procedure that follows, specifically for step 8.

Note also in the picture in Figure 1 the location of the removable jumpers J3 and J5. These jumpers must be correctly inserted across the connector, rather than along it, for proper operation of the LaunchPad.

To the extreme left and right of the socket we find two rows of ten empty holes, or PBC female connectors, called J1 and J2, respectively, for the Port Headers. Newer versions have male headers already soldered there. These spaces gives access to the MSP430 pins. The lines are numbered from 1 to 10 at J1, and 11 to 20 at the right J2. The right top hole is labeled GND (meaning electrical ground) with pin

number 20 and the lowest right hole is labeled P2.3 with a pin number 11.

On the left side of the board, the top hole is labeled VCC with a pin number 1 and the lowest hole is labeled P2.2 with a pin number 10. On the lower right corner we find an optional power port J6 and to its left a push button used as a reset key. If we continue moving to the left, we find jumper set J5 with two jumpers in place. Finally, on the lower left corner we have another user key implemented as a push button labeled P1.3.

Jumpers at J5 connect pins P1.0 and P1.6 to LEDs LED1 and LED2, respectively (See Fig. 1.2). You should remove the jumper when using these pins for other connections. There are no other removable jumpers in the LaunchPad.

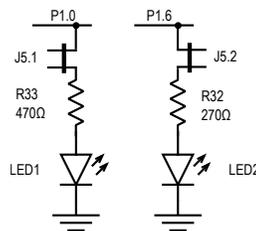


Figure 1.2: LED connections to pins in EXP430G2 Launchpad.

Finally, there are two pushbuttons S1 and S2 in the low region. Switch S1 is connected to the RST/NMI/SBWDIO, terminal of the socket 16 and RST of the Spy-Bi-wire. Switch S2 is connected to terminal P1.3, to be used as input. The connection is done via pull-up resistors and capacitors, as shown in Figure 1.3. Newer launchpad versions do not include the pull-up resistor R34 or capacitor C24 at P1.3. This means that when using this terminal as input, the internal pull-up resistor must be used, as explain in the respective laboratory.

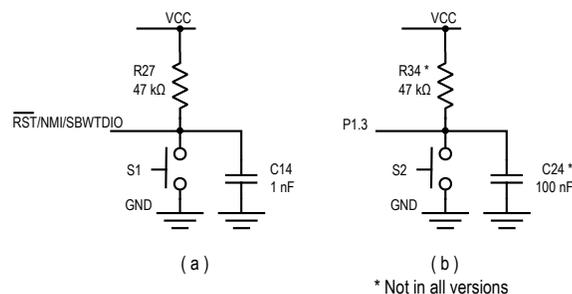


Figure 1.3: Connection of pushbuttons in EXP430G2 Launchpad: (a) Reset button S1; (b) Input button S2 at P1.3 (newer versions do not include R34 or C24)

We end our tour mentioning terminals XIN and XOUT (19 and 18, respectively). These are signals of the LFX1T1 oscillator and can support low-frequency os-

cillators like a watch crystals of 32768 Hz or a standard crystal with a range defined in the model associated data sheet. The kit includes an MS3V-T1R 32.768 kHz, not connected. Information for this crystal can be found at <http://www.microcrystal.com/CMSPages/Get0886-41a2-87fc-6b04e14f0226>.

The XIN and XOUT terminals can also be used as P2.7 and P2.6 pins for port 2. Models with 14 pins supported by the Launchpad only have these two pins from port 2.

1.2 Procedure

IAR Tutorial (This tutorial assumes the IAR IDE is already installed in your computer. If IAR is not installed, follow the installation instructions given at the end of this experiment. At the time of printing, the current IAR version is 5.20.1.50215).

1. On the workstation go to 'Start > All Programs > IAR Systems > IAR Embedded Workbench Kickstart'. Additional letters and numbers will follow after the word Workbench, depending on the installed version. The initial IAR Embedded Workbench screen appears as shown in Figure 1.4.

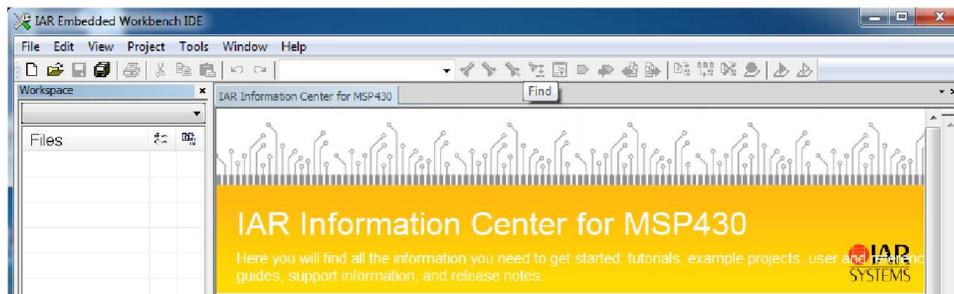


Figure 1.4: IAR initial window.

2. To create a new project: select 'Project > Create New Project ...'. Make sure MSP430 is chosen in the Tool Chain dropdown menu in the 'Create New Project' window. Choose 'asm' in the 'Project templates:' by clicking the '+' sign in 'asm' and then 'asm' in the tree that opens up. Click 'OK'. Figure 1.5 illustrates the screen you should get from the IDE if you follow the steps explained above. To the left of the screenshot we have an empty project tree view pane. What is new here is the 'Create New Project' window with the 'Project templates' pane in it.

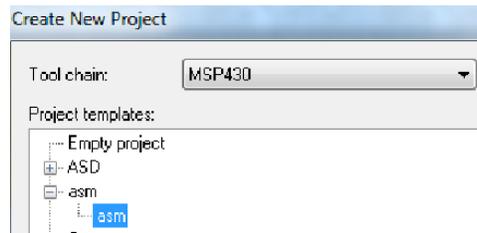


Figure 1.5: Creating new project window.

3. The 'Save as' window opens. Choose a destination directory and type the name "EXP1" for your project in the 'File name:' box. If you prefer to create a new folder, you will need to click the 'New Folder' icon, give a folder a name, we will name our new folder "Experiment1" (otherwise it will be called New Folder), now click 'Open' to open your folder and then enter a "EXP1" in the 'File name:' textbox and click 'Save'. Since this step could be somewhat

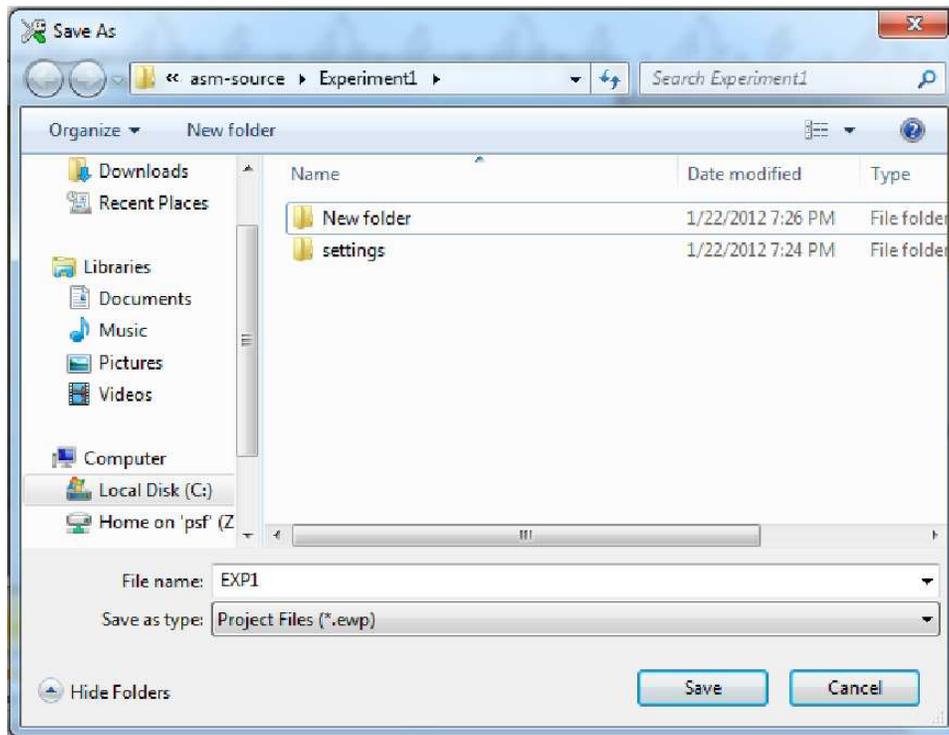


Figure 1.6: Creating new projectw with new folder window.

confusing to some of our readers we chose to show it for you. The picture shown illustrates what you should see in the screen if you chose to create a new folder as already explained above. Note that the default name for the folder is

New Folder as shown in Figure 1.6 and explained in the previous steps.

- Right click the 'asm.s43' tab and choose 'Close' in order to close it. The only reason we are asking you to close this tab is to eliminate the possibility of confusion when you are looking at the different opened files that are available for you to see. You could choose to keep the tab open if you want, but the rest of the lab assumes it has been closed. We will be assigning a '.asm' extension to our assembly language file as opposed to the default '.s43' in the IAR Embedded Workbench. Although we closed the '.s43' tab, the file is still shown in the project tree view located to the left of the screen. We will deal with this in the next step. Figure 1.7 illustrates the available actions when right clicking the 'asm.s43' tab.

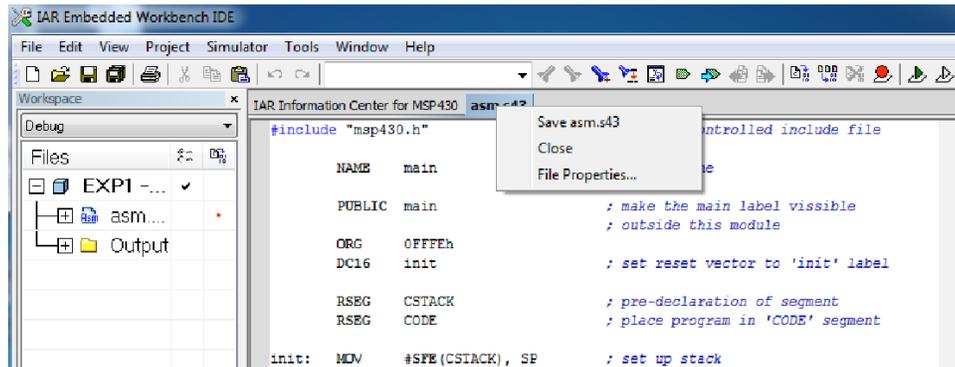


Figure 1.7: Closing asm.s43 tab.

- In the project tree view right click the 'asm.s43' file and remove it by choosing 'Remove'. You will need to confirm removing the 'asm.s43' file, confirm it and the file will be removed from the project tree view. This is shown in Figure 1.8.
- Click 'Tools > Options ... > Editor' and make sure you click the radio button for 'Insert Tab' under the 'Tab Key Function:' alternatives. You may want to check the 'Show line numbers' option. Then click 'OK'. If you do not check the 'Insert Tab' radio button, then eight (8) space characters will be inserted in your source file whenever you press the Tab key as opposed to just one (1) tab character. Your source file could then grow rather fast in size. The 'Show line numbers' option, although not strictly needed, could turn out to be rather useful. These alternatives are illustrated in Figure 1.9 taken as a screenshot in IAR Embedded Workbench.

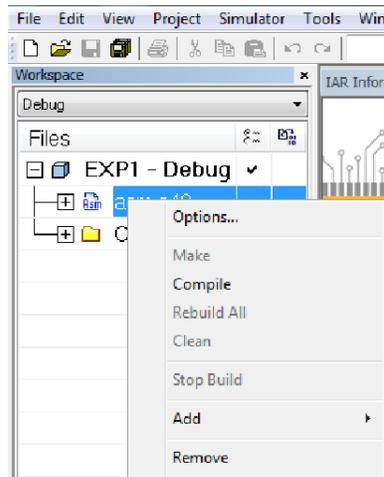


Figure 1.8: Removing asm.s43 from the project tree.

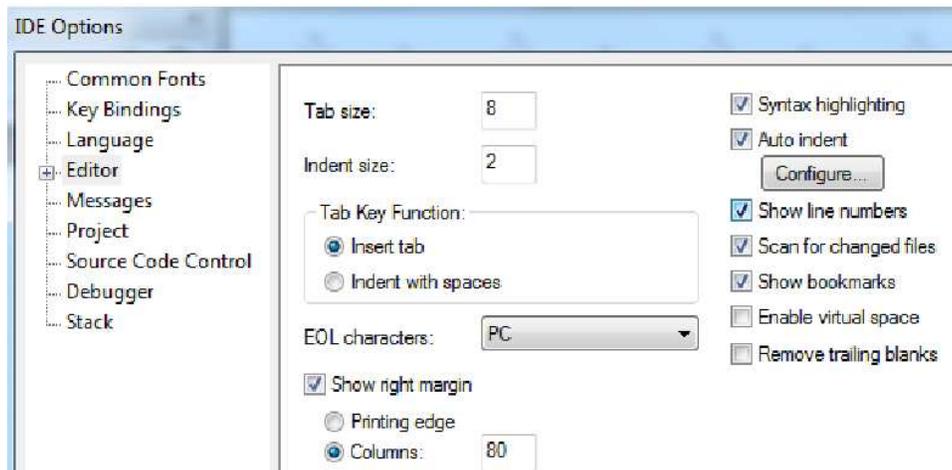


Figure 1.9: IAR integrated development environment (IDE) options.

7. Make sure that you have your project name selected in the project tree view on the left side of your screen. Click Project > Options Now, under Category: click Debugger > Setup and choose FET Debugger in the Driver dropdown menu. Depending on the options you chose when you installed your IDE you may or may not have the Driver dropdown menu. It is important that you make sure you have selected Texas Instrument USB-IF in the dropdown menu. The rest of the options are chosen for you. DO NOT click OK yet. See the illustration in Figure 1.10.
8. Now, while still in the Options for node EXP1 window, click General Options >

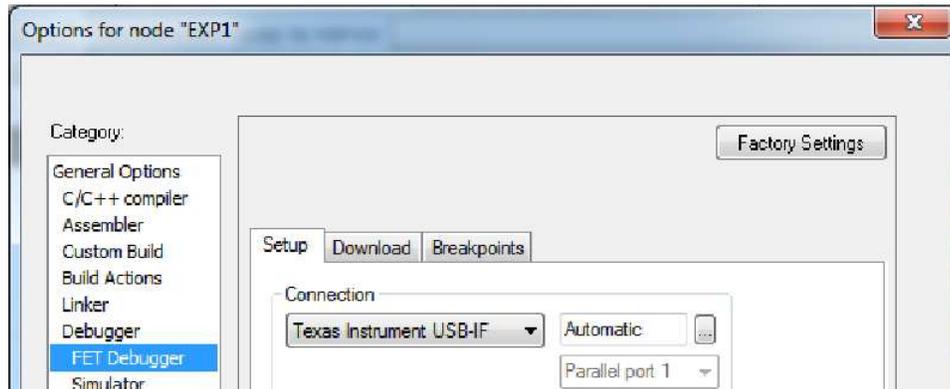


Figure 1.10: Options for EXP1.

Target > MSP430Gxxx Family > MSP430G2231. If you are using a different MSP430 device, choose the device you are using instead. Now click OK. This is shown in Figure 1.11.

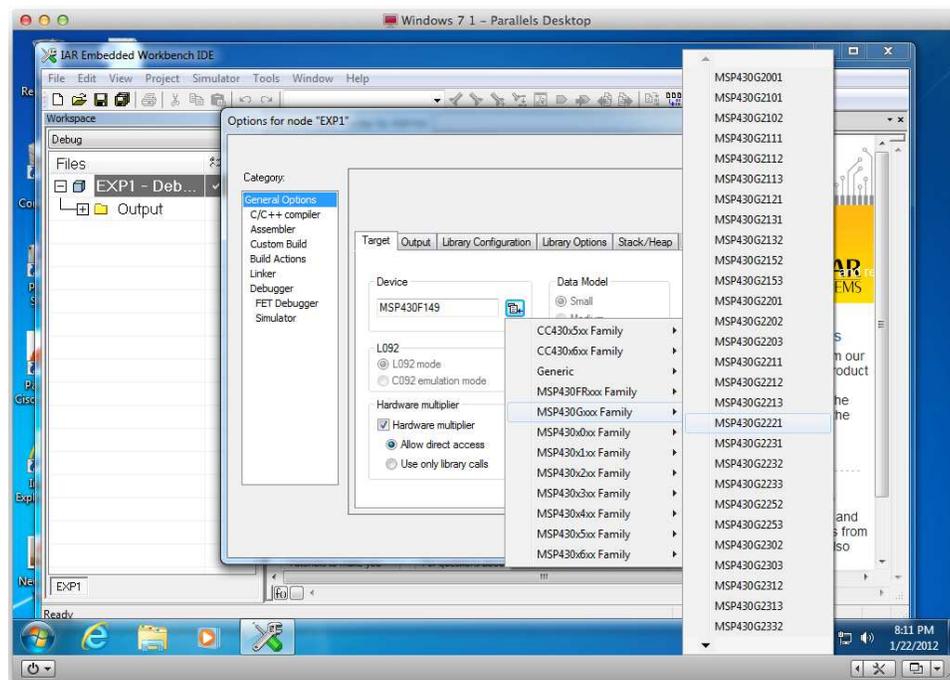


Figure 1.11: Choosing the MSP430 target.

9. Now click File > New > File.
10. Type the code shown below as is into the source window and then save the

work done by clicking File > Save As.

Listing 1.1: Your First Assembly Language Program

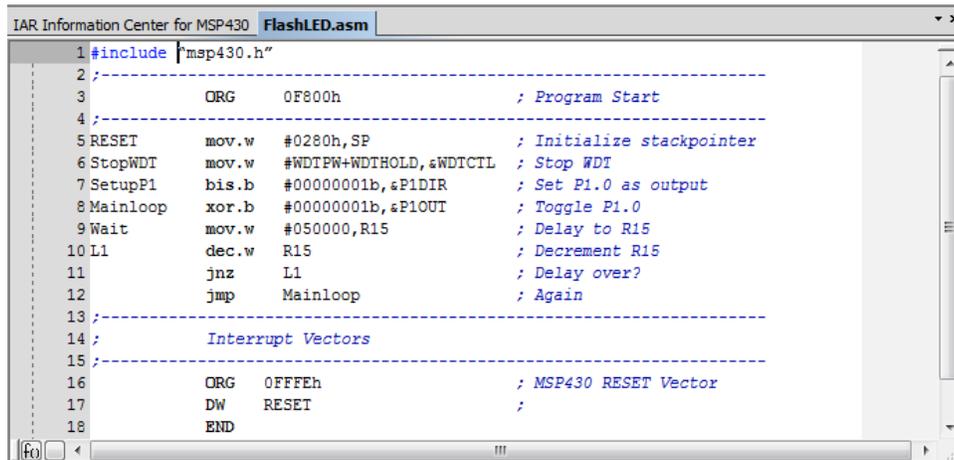
```

1      #include      msp430.h
2  ;-----
3      ORG          0F800h ; Program Start
4  ;-----
5  RESET  mov      #0280h,SP      ; Initialize stackpointer
6  StopWDT mov     #MDTPW+WDTHOLD,&WDTCTL ; Stop WDT
7  SetupP1 bis.b   #1,&P1DIR      ; P1.0 as output
8  Main   xor.b   #1,&P1OUT      ; Toggle P1.0
9  Wait   mov     #50000,R15     ; Delay to R15
10 L1     dec     R15           ; Decrement R15
11       jnz     L1           ; Delay over?
12       jmp     Main        ; Again
13 ;-----
14 ;           Interrupt Vectors
15 ;-----
16       ORG     0FFFEh        ; MSP430 RESET Vector
17       DW     RESET          ;
18       END

```

Do not type in the line numbers in the leftmost column of the listing. These are only for your reference. Name your file FlashLED.asm. Your code should look like the one in the IAR FlashLED.asm window shown in Figure 1.12.

11. Click Project > Add Files and choose the source file saved in the previous step. Save your workspace by clicking File > Save Workspace and then click Open as shown in the Figure 1.13.
12. Connect the LaunchPad to the computer via the USB cable provided with it. If this is the first time the LaunchPad is connected, you will probably notice that Windows will install the appropriate driver. This driver is supplied as part of IAR. The rest of this experiment relies on the assumption that the driver is properly installed. Note that the LaunchPad comes with a program already in flash memory and as soon as you plug it in it will start off and begin to turn on and off the red and green LEDs in turn.
13. You can now click Project > Download and Debug.



```

IAR Information Center for MSP430 FlashLED.asm
1 #include "msp430.h"
2
3 -----
4          ORG    0F800h          ; Program Start
5 -----
6 RESET    mov.w  #0280h, SP      ; Initialize stackpointer
7 StopWDT  mov.w  #WDTPW+WDTHOLD,&WDTCIL ; Stop WDT
8 SetupP1  bis.b  #00000001b,&P1DIR ; Set P1.0 as output
9 Mainloop xor.b  #00000001b,&P1OUT ; Toggle P1.0
10 Wait    mov.w  #050000,R15     ; Delay to R15
11 L1      dec.w  R15             ; Decrement R15
12        jnz    L1              ; Delay over?
13        jmp    Mainloop        ; Again
14 -----
15          ; Interrupt Vectors
16 -----
17          ORG    0FFFFh         ; MSP430 RESET Vector
18          DW    RESET          ;
19          END

```

Figure 1.12: Code for FlashLED source program.

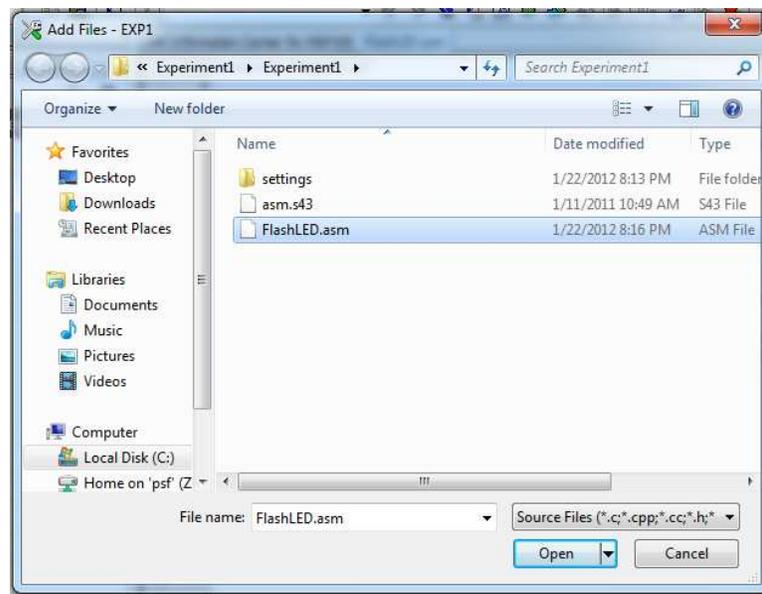


Figure 1.13: File Save Workspace window.

14. Assuming you entered the code with no errors, you might still get the Stack Warning window with a warning message. If it shows up, just ignore it and click OK. Now go to Tools > Options > Stack and uncheck the Stack pointer(s) not valid until program reaches: option. Click OK. Refer to Figure 1.14.
15. If you did enter some errors, they must be syntax errors as opposed to logic errors or programming errors. Note that when the assembler finds a semicolon (;) on any line, it ignores the rest of the line. This means that it is not important

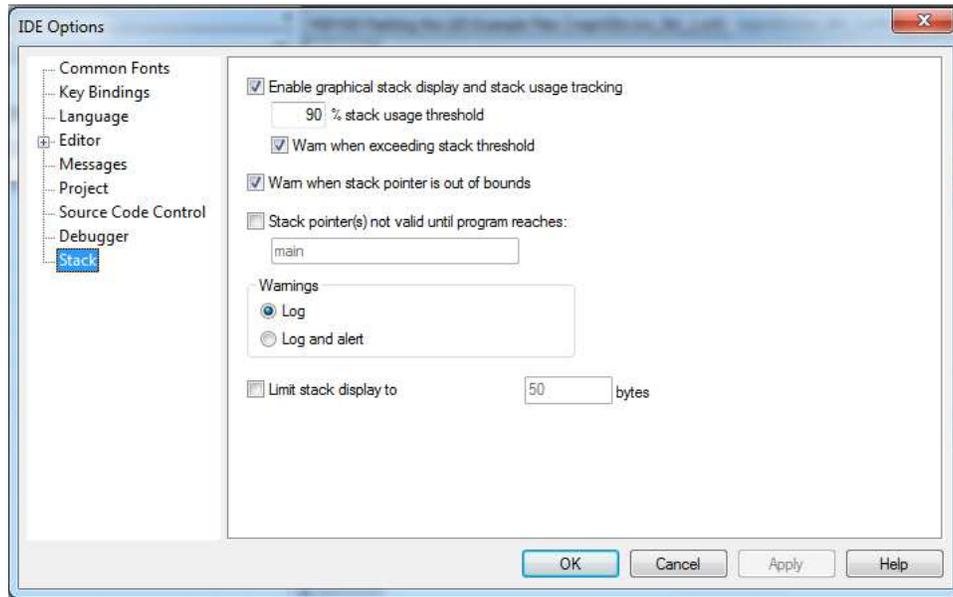


Figure 1.14: Tools options stack window.

if you have made a typographical error after a semicolon. Review each line to make sure that the code entered is exactly as shown above. The code has been properly tested to make sure that the steps shown here will produce the same results for you.

16. You should now be in the Debugger environment. You should see the Debug toolbar, the second toolbar beginning at the toolbar after the menu bar, and the Disassembly window on the right as shown in Figure 1.15.

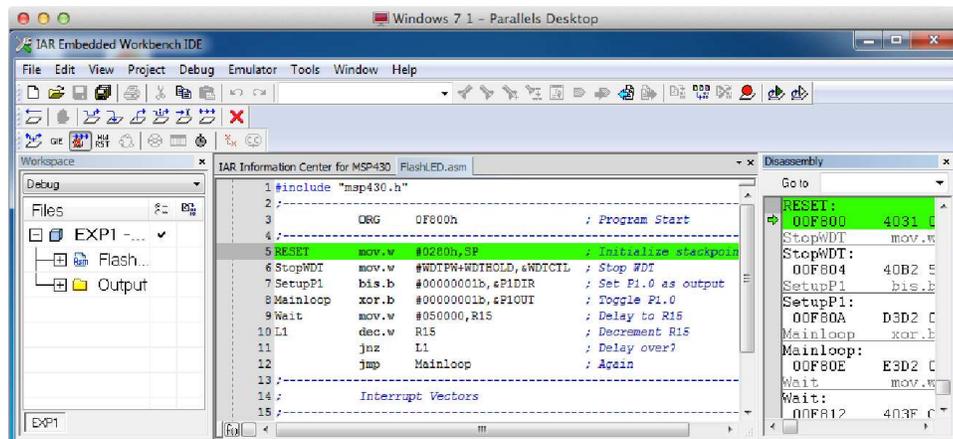


Figure 1.15: IAR debug window.

17. Mouse over the Debug toolbar shown in Figure 1.16. The first icon should

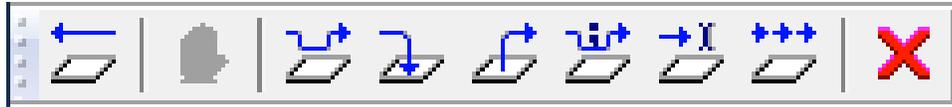


Figure 1.16: IAR debug toolbar.

give you a Reset message, the second a Break message, the third a Step Over message, then a Step Into with the fourth, Step Out, Next Statement, Run to Cursor, the second to last is Go, and the last is Stop Debugging.

18. Click View > Register and note that the Register window is added. The Register window should look something like the one shown in Figure 1.17. Now click

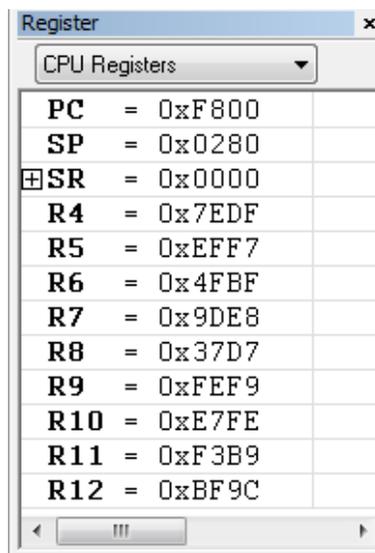


Figure 1.17: IAR register window.

View > Memory and the Memory window will be added. The memory window will look something like the one shown in the Figure 1.18. Notice that some of the other windows will most probably be reduced in size to accommodate the Memory window.

19. Click the Go icon or the F5 key.
20. The red LED should be toggling on and off in the MSP-EXP430G2 LaunchPad.

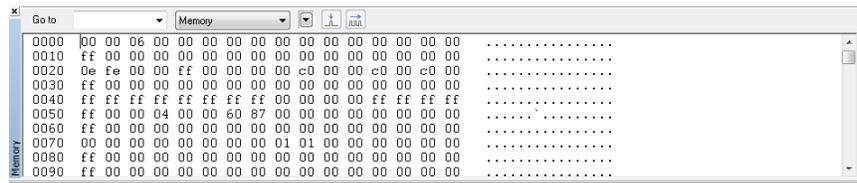


Figure 1.18: IAR memory window.

1.3 Exercises

1. Click the Stop Debugging icon in case your program is in the Debugger perspective. Replace the #50000 in line 9 with #100000 (depending on your IDE you may get an error here, in that case use a smaller number) in your program. You can now click Project > Download and Debug. Your changes will be saved automatically. Click F5 to run your program. What change did you observe on the red LED toggling frequency?
2. Repeat the above exercise replacing the #50000 or #100000 with #2500. Click the Break icon. Now click Step Into several times and watch the Register window. You should pay particular attention to the PC and R15 registers which should be changing each time you click Step Into. Go ahead and change the value for register R5 by clicking the value for R5. Now enter decimal number 256 and hit the Enter key. You should see that the new value is 0x0100, which is the hex representation of the value just entered. Now change it to 0xabcd. You should see that it changed the letter digits to (uppercase) 0xABCD.
3. Go to the Disassembly window and type 0xF800, the address shown in the first ORG statement in your program in line 3, in the Go to textbox. You should see something like what is shown in Figure 1.19:

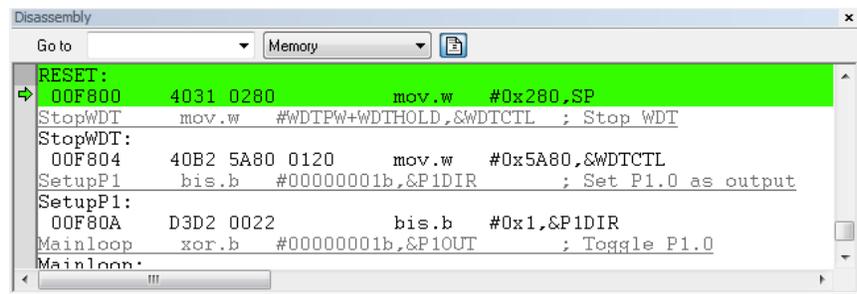


Figure 1.19: Disassembly window.

For the sake of clarity, the entire program in the Disassembly window is shown below.

Listing 1.2: Code in Disassembly Window.

```

1 RESET:
2 00F800 4031 0280      mov      #0x0280 ,SP
3 StopWDT:
4 00F804 40B2 5A80 0120  mov      #0x5a80 ,&WDTCTL
5 SetupP1:
6 00F80A D3D2 0022      bis .b   #1,&P1DIR
7 Main:
8 00F80E E3D2 0021      xor .b   #1,&P1OUT
9 Wait:
10 00F812 403F 5000     mov      #0x5000 ,R15
11 L1:
12 00F816 831F          dec      R15
13 00F818 23FE      jne     L1
14 00F81A 3FF9          jmp     Main

```

You will probably recognize the instructions in lower case letters since you entered them when you were writing the code. There are, however, several things that were added to your program. For example, the first instruction is displayed:

```
00F800 4031 0280      mov      #0x0280 ,SP
```

The above line shows the strings 00F800 4031 0280 at the beginning of the line. The string 00F800 indicates the address in memory of the first byte of the program. Strings 4031 and 0280 are the memory contents starting at location 00F800. The convention used in this representation is the little endian notation. These last two character strings are the machine language representation of the assembly language instruction `mov #0x0280,SP`. The first of these two strings, 4031, indicates that this is a MOV instruction working on two bytes or 16 bits at a time, while the second string, #0x0280, specifies the source operand for the MOV instruction. If you only leave the program memory addresses and their content you should see this

```

F800      4031 0280 40B2 5A80 0120 D3D2 0022 E3D2
F810      0021 403F 5000 831F 23FE 3FF9 FFFF FFFF

```

- Now click View > Memory and choose FLASH in the dropdown Memory menu. You should see a display similar to the above. At this time you should not be concerned with the actual meaning of the assembly language instruction or its

machine language version. Our intention at this time was to show you that with IAR you can see the program memory and its actual content. This will come in handy in the future.

5. For this part of the exercise we will assume that the red LED is toggling on and off. Locate connector J5. Now grab jumper P1.0 with your fingers and pull it out of the connector. The red LED should be off because no power is delivered to it although the program is still running. Replace the jumper and the red LED should once again begin to toggle on and off. If the green LED were toggling on and off, you could do the same by removing and reinserting jumper P1.6.
6. Another exercise is to turn each LED in turn. One way to accomplish this is to go to Listing 1.1 and change line 7 and insert an instruction right after it as follows

```
7 SetupP1 bis.b    #01000001b,&P1DIR        ; P1.0/P1.6 outputs
8          bic.b    #01000001b,&P1OUT        ; both LEDs off
```

so that the above two lines are numbered as shown above.

Now, insert the following instructions

```
13          xor.b    #01000001b,&P1OUT        ; P1.0 off , P1.6 on
14          mov     #50000,R15
15 L2       dec     R15
16          jnz     L2
17          xor.b    #01000001b,&P1OUT        ; P1.0 on, P1.6 off
```

just before the instruction `jmp Main` so that they are numbered in your code as shown above.

After performing the changes your code should now look like this:

Listing 1.3: Your code after the changes.

```
1          #include    "msp430.h"
2 ;-----
3          ORG         0F800h    ; Program Start
4 ;-----
5 RESET    mov     #0280h,SP      ; Initialize stackpointer
6 StopWDT  mov     #MDTPW+WDTHOLD,&WDTCTL ; Stop WDT
7 SetupP1  bis.b    #01000001b,&P1DIR        ; P1.0/P1.6 outputs
8          bic.b    #01000001b,&P1OUT        ; both LEDs off
9 Main     xor.b    #00000001b,&P1OUT        ; Toggle P1.0
```

```

10 Wait    mov     #50000,R15      ; Delay to R15
11 L1      dec     R15             ; Decrement R15
12         jnz     L1             ; Delay over?
13         xor.b  #01000001b,&P1OUT ; P1.0 off , P1.6 on
14         mov     #50000,R15
15 L2      dec     R15
16         jnz     L2
17         xor.b  #01000001b,&P1OUT ; P1.0 on, P1.6 off
18         jmp     Main          ; Again
19 ;-----
20 ;                Interrupt Vectors
21 ;-----
22         ORG     0FFFFeh ; MSP430 RESET Vector
23         DW     RESET
24         END

```

1.4 Installing IAR Workbench

1. Obtain a copy of the installation files for the IAR Embedded Workbench. You can obtain a copy of IAR at <http://www.ti.com>. If the files are in a compressed format (usually they are), then you should extract the files to a destination directory of your choice.
2. Double click the CCS setup application. The install wizard displays progress information.
3. Follow the wizard prompts during the installation. Click Next if an action is required from you.
4. After reading the license agreement, accept it if you agree with it and still want to install IAR. Click 'Next'.
5. Choose the Complete in the Setup Type window and then click Next.
6. If this is the first time IAR will be installed on your computer, choose the default path settings for the installation location, unless you want to install it in another location. Click Next. **If you have already installed another version on your computer, make sure you choose a different path for this installation.**

7. You will be asked to choose a Program Folder. Click Next if you agree with the default.
8. Click Install. The wizard displays setup status information, including the Visual C++ installation. Click Finish when asked to do so.
9. The IAR Embedded Workbench IDE should open along with a browser showing the Release Notes.

The installation instructions shown above are general in nature and assume you are using the Windows XP or Windows 7 operating system (OS). The installation procedure has also been tested on a MacBook Pro with Mac OS X Snow Leopard with a Windows XP and a Windows 7 virtual machine. Depending on your OS and the IAR version, the setup wizard may behave differently. You should make sure that your computer and OS meet the requirements for installing IAR. You should also be aware that, on some operating systems, you may have to make the installation from the Administrator account.

Experiment 2

MSP430 Instruction Set Part 1

Objectives

- Become familiar with the MSP430 instruction set data transfer instructions, arithmetic instructions, logic instructions, and the program control instructions.
- Learn more ways to control program execution using the IAR Integrated Development Environment.

Duration

2 Hours

Materials

- IAR Embedded Workbench (IAR) Application.
- MSP-EXP430G2 LaunchPad development board.

2.1 Introduction

The MSP430 instruction set has 27 core instructions and 24 emulated instructions. Emulated instructions are used to make it easier for people who are already familiar with them instructions to use them while programming the MSP430. However, when the assembler finds an emulated instruction it substitutes it with the corresponding core instruction. So, if you are new to assembly language programming, you do not need to worry about learning the emulated instructions and you can concentrate on learning only the core instructions. On the other hand, even if you are new to assembly language programming, you may find that an emulated instruction is more

intuitive than its corresponding core instruction counterpart. If this is the case, you may want to also learn the emulated instructions.

The MSP430 instructions are divided into several categories: data transfer instructions, arithmetic instructions, logic instructions, and program control instructions. This lab is not intended as a comprehensive and in depth study of the MSP430 instruction set. However, at the end of this lab you should feel more comfortable with both assembly language and the MSP430 assembly language instructions.

An MSP430 source statement may be an assembler directive, an assembly instruction, a macro directive, or a comment. The format is shown below, where brackets indicate an optional field. Fields are separated by one or more blank spaces.

```
[Label[:]] [mnemonic] [operand] [; comments]
```

The Label must begin on the first column and can be followed by a colon. The mnemonic is an identifier used for directives, macros or CPU instructions. If there are two operands, then they will be separated by a comma. Any part of a line after and including a semicolon (;) is considered a comment. Comments are used to document the program and are ignored by the assembler, accordingly, comments have no effect on the execution of a program.

Let us revisit the FlashLed.asm program that we used in experiment 1 and shown below for ease of reference. It includes examples of most of the MSP430 assembly language instruction categories. It also includes examples of directives and comments. The FlashLed.asm program has two assembler directives at the beginning and three at the end of the program. In between the assembler directives we have several assembly language instructions. The `#include`, `ORG`, `DW` and `END` are all assembler directives. The first directive `#include` tells the assembler that it will use C style declarations that are to be found in the specified header file, i.e. `msp430.h`.

The `ORG` directive tells the assembler the address of the first byte it should use, in this case `0F800h`, for the beginning of the code section. The next `ORG` directive instructs the compiler to start at the address included, i.e. `0FFFEh`, the reset interrupt vector address. The `DW` directive indicates to the assembler to reserve a word, i.e. 16 bits, location. The directive is indicating the assembler which value to place at the location of the reset interrupt vector. For this program, the assembler is instructed to place the address associated with the `RESET` label at the location reserved for the reset vector, i.e. the address of the first byte of the first word of the machine language instruction corresponding to the assembly language instruction `mov #0280h,SP`. Finally, the `END` directive tells the assembler that this is the end of the program and that it should ignore anything written after it. We will have more to say about the interrupt vectors in another experiment. For now, suffice it to say

that what the second ORG and the DW directives are accomplishing is making sure that the program starts with the first instruction each time the Reset button in the LaunchPad is pressed.

Listing 2.1: FlashLED.asm Program

```

1      #include          msp430 . h
2  ;-----
3      ORG      0F800h   ; Program Start
4  ;-----
5  RESET  mov      #0280h,SP      ; Initialize stackpointer
6  StopWDT mov     #MDTPW+WDTHOLD,&WDTCTL ; Stop WDT
7  SetupP1 bis .b  #1,&P1DIR      ; P1.0 as output
8  Main   xor .b   #1,&P1OUT      ; Toggle P1.0
9  Wait   mov      #50000,R15     ; Delay to R15
10 L1     dec      R15           ; Decrement R15
11       jnz      L1           ; Delay over?
12       jmp      Main         ; Again
13 ;-----
14 ;           Interrupt Vectors
15 ;-----
16       ORG      0FFFEh        ; MSP430 RESET Vector
17       DW      RESET          ;
18       END

```

In Listing 2.1 we have three data transfer instructions, i.e. the three mov instructions, two logic instructions, the bis.b and the xor.b instructions. We also have one arithmetic instruction, namely the dec instruction, and two program control instructions represented by the jnz and jmp instructions. Of course, the MSP430 instruction set includes more data transfer instructions, more logic instructions, more arithmetic instructions, and also more program control instructions. We will just focus on the instructions found in the program above for now.

The mov or mov.w instruction, the .w is used to make explicit the fact that the operation will be on a word (16-bits), transfers the contents of the source operand onto the destination operand. Since this is a two-operand instruction, the left operand is the source operand and the right operand is the destination operand. Thus, the mov #0280h,SP instruction places a copy of the hexadecimal value 0x280 into the contents of register SP, where mov is the mnemonic.

The next mov instruction deals with the watchdog timer, a peripheral device

inside every MSP430 microcontroller. The watchdog timer has many uses, one of which is to reset the CPU after a specified amount of time elapses. The idea is that it will reset the CPU in case, for example, it gets stuck in some loop. Since we do not want the watchdog timer to reset the CPU while we are working with it, we need to turn it off. This is accomplished by writing a specific value to the watchdog timer control register, WDTCTL, using a `mov` instruction. In order to write to the 16-bit watchdog timer control register it is always necessary to write 0x5A in the upper byte. The immediate value WDTPW accomplishes this. Also, since the watchdog timer must be stopped, we need to write a 1 in bit 7, which is represented by the immediate value WDTWDT. You need to refer to the WDTCTL bit distribution in order to determine this. The values for WDTPW and WDTWDT are defined in the `msp430.h` file.

Next we have the `bis .b #1,&P1DIR` instruction. This is an example of a logic instruction. The `bis .b` instruction is an emulated instruction. It has a `.b` suffix meaning it is a byte operation. It so happens that P1DIR, a special function register or SFR, is a byte location. You could also byte-operate on a word location. P1DIR determines whether a bit will be an input or an output bit. If we set a bit in P1DIR to 1, that bit will be an output, otherwise it will be an input bit. With this instruction we are performing a bitwise inclusive-or (OR) operation. Thus, we are setting bit P1.0 as an output bit ($1 \text{ OR } 0 = 1$, $1 \text{ OR } 1 = 1$). The rest of the bits in port 1 are unaffected ($0 \text{ OR } 0 = 0$, $0 \text{ OR } 1 = 0$).

The next instruction is `xor .b #1,&P1OUT`. This is again a logic instruction which is also operating on a byte. This is the exclusive-or (XOR) instruction in the MSP430 instruction set. Since $0 \text{ XOR } 0 = 0$, $0 \text{ XOR } 1 = 1$, $1 \text{ XOR } 0 = 1$, and $1 \text{ XOR } 1 = 0$ we see that the effect of the `xor .b` instruction is to leave unchanged a bit XORed with a 0 and toggle any bit XORed with a 1. Thus, P1.0 is toggled each time this instruction is executed. If the red LED is connected to P1.0 and it was turned off, then it will be lit. If it was on, then it will be turned off.

The `mov #50000,R15` instruction loads decimal number 50000 into R15. Register R15 is used to hold the number of times left for the loop to execute. This value is decremented by one using the `dec R15` instruction that follows. This is an example of a single operand arithmetic instruction. Both arithmetic and logic instructions affect the flags located in the status register (SR). The next instruction is `jnz L1`. This is an example of a program control instruction. Specifically, it is a conditional jump instruction. It is used to test if the `dec` instruction made the contents of R15 not zero, i.e. if Z, the zero flag in the SR register, equals zero. Each time the condition is satisfied, program control will be transferred back to the instruction found at label L1, i.e. the `dec R15` instruction. This will go on 50000 times until finally

the zero flag will be set to 1 because R15 will be zero. At this point the condition will not be satisfied and program execution will continue at the next instruction in sequence, `jmp Main`. This last instruction, i.e. the `jmp Main` instruction, is another example of a program control instruction. In this case, however, the jump instruction is an unconditional jump. This means that program control will be transferred to the instruction at label `Main`, i.e. the `xor` instruction without considering any condition. The program will never stop executing unless power is removed.

2.2 Procedure

1. We will assume that a project with the above source program is already in place. If this is not the case, you will need to perform experiment 1.
2. Connect the LaunchPad to the computer via the USB cable provided with it.
3. You can now click 'Project > Download and Debug' or `Ctrl+D`. The Debug perspective is displayed and the code downloads. You should see that the Debug toolbar was added along with a Disassembly window, among other things.
4. Now, before you execute your program, go to the `FlashLed.asm` window in the Debug perspective. Lets assume we want program execution to stop before executing

```
RESET    mov    #0280h, SP
```

Make sure that you are in the `FlashLED.asm` window and place your cursor to the left of the number where the above line is displayed. If you have the same program shown here, then that line number should be 5. Thus, place your cursor to the left of the 5 and double click. You should see a small symbol appear to the left of the 5 and the entire corresponding part to be highlighted red in both the `FlashLED.asm` and the Disassembly windows. This means you have placed a breakpoint at that line that will allow you to stop program execution just before executing line 5. Do the same, i.e. place another breakpoint, at line 6.

5. Open the Register window by clicking 'View > Register'. What we are looking for is for the contents of `SP`. The instruction will transfer a copy of the hexadecimal constant 280, i.e. `0x0280`, to the contents of register `SP`, the stack pointer. Hit the `F5` key for the program to begin. The program should run and execute only the part of the code before the breakpoint, in this case, nothing.

Change the value in SP to 0. Now hit F5 again and you should be able to see that SP now has the value 0x0280.

6. Now go ahead and place a breakpoint at line 11 to watch the contents of R15. This time you should have trouble placing the breakpoint. You will need to toggle at least one of the previous two breakpoints in order to set a new one. Double click at each of the two previous breakpoint locations to toggle them. Now set a breakpoint at line 11. Assume we want to start the program from the beginning. While in the Debug perspective, click 'Debug > Reset'. Now hit F5 to execute. It will run until it finds the breakpoint at line 11. If you watch the value of R15 in the Registers window you should be able to see it decrementing its value by one, remember though, that the number system is hexadecimal. Instruction L1 dec R15 is an example of an arithmetic instruction in the MSP430 instruction set. As opposed to the data transfer instruction, like the mov instruction, arithmetic instructions can alter the status of the flags in the status register SR. You could also expand the SR in the Register window and watch the value of Z, it should say $Z = 0$.
7. If you do not want to set a breakpoint, there is still a way to run your program and stop before executing a particular instruction. Before we do that, let us eliminate the breakpoint we have so far. Make sure the cursor is at the line where you want to toggle the breakpoint and click the 'Toggle Breakpoint' icon, i.e. the third to last on the top toolbar, i.e. the Main toolbar.
8. Another way to execute all instructions prior to a specific instruction is to use 'Debug > Run to Cursor'. In order to do this, place the cursor over the instruction in the Debug perspective where you want the program to stop. Now you can either right click the line and choose 'Run to Cursor' or click 'Debug > Run to Cursor'. When you do this, the program will run and stop prior to executing the instruction. You can now examine the registers and memory locations.

2.3 Exercises

1. What we would like you to do now is to set breakpoints to examine the following instructions

```
SetupP1  bis .b    #00000001b,&P1DIR
Mainloop  xor .b    #00000001b,&P1OUT
```

Thus, what you need to do is to examine two logic instructions and then two program control instructions in another exercise. The logic instructions will affect the contents of P1DIR and P1OUT. Both P1DIR and P1OUT control how port P1 in the MSP430 behaves. P1DIR is used to set each of the 8 bits in port P1 as either an input port, if the corresponding bit is set to 0, or as an output port if it is set to 1. In this program we are setting pin 0 in P1, i.e. P1.0, as an output port by ORing (inclusive- or) P1.0 with 1. Now, bits 1 through 7 in P1, are being ORed with 0 and thus their actual configuration is not affected. This is as opposed to setting them to 0. If we wanted to set each and every bit in P1DIR, then we could use a mov instruction. In this case, however, we only wanted to affect P1.0 and leave unaffected the rest of the bits, thus the use of the bis instruction as opposed to using the mov instruction. Generally speaking, logic instructions can affect the flags in register SR. However, the bis instruction does not affect any of the flags. In order to see the effect on P1DIR, simply choose Port 1/2 in the Register window dropdown menu. Expand it and you should be able to see each bit in P1DIR.

In order to actually send a bit out P1 we use P1OUT. If you send a bit out an input pin it has no effect on the bit and nothing is sent through it. You can only send out bits through output pins. We could have chosen to send bits out P1.0 using one of several other instructions, but most all of those instructions might change the value of any other output pin. What we would like to do is to send a 0 to turn the red LED off or a 1 to turn it on. However, in this experiment, we chose not to initialize the output pin and just work with whichever value it had after powering on the LaunchPad. The xor instruction has the ability to toggle some bits and do nothing to the rest. Thus, in this experiment we use the xor instruction to toggle P1.0 leaving alone the rest of the bits. As most logic instructions will do, the xor instruction does affect the flags in the SR register. In this case, however, we are not interested in the effect the xor instruction has on the flags, but you should be aware of the fact that it does have the ability to affect the flags.

2. Set breakpoints at instructions

```
jnz      L1
jmp      Main
```

These are two examples of the program control instructions in the MSP430 instruction set. The jnz mnemonic indicates this is a conditional jump instruction. The mnemonic means, "jump if not zero". In other words, jump to the instruction whose first byte is at the location in memory indicated by label L1 if

the zero flag in the SR register has the value 0, i.e. $Z = 0$, otherwise continue with the next instruction, which in this case is the unconditional jump instruction `jmp Main`. Remember that the conditional jump instruction comes after the `dec` instruction. This means that it is the effect of decrementing register R15 by one which is being whose condition is being tested by the conditional jump instruction. We already saw how R15 was being decremented. After placing a breakpoint at the conditional jump instruction you will be able to see how program control is changed when you run the program. Watch both the contents of R15 in the Registers window and also see how program execution goes from the conditional jump instruction back to the decrement instruction and so on.

At some point where program execution is stopped because of the breakpoint at the conditional jump instruction, go to the Registers windows and change the value of R15 to say 2. In this way we will be able to see program control to pass from the conditional jump instruction to the unconditional jump instruction when the contents of R15 reaches 0. At this point the microprocessor is about to execute `jmp Main`. If you run the program again, you will see program control being passed to the instruction at label `Main`, i.e. the `xor` instruction. You should have seen how the content of PC was updated to reflect this fact.

Experiment 3

MSP430 Instruction Set Part 2

Objectives

- Become familiar with the MSP430 instruction set data transfer instructions, arithmetic instructions, logic instructions, and the program control instructions.
- Learn more ways to control program execution using the IAR Integrated Development Environment.

Duration

2 Hours

Materials

- IAR Embedded Workbench (IAR) Application.
- MSP-EXP430G2 LaunchPad development board.

3.1 Introduction

As mentioned in experiment 2, the MSP430 instruction set has 27 core instructions divided into several categories: data transfer instructions, arithmetic instructions, logic instructions, and program control instructions. In this lab we will be presenting two more data transfer instructions, push and pop, along with two of the MSP430 instructions that affect the flow of control in a program, call and ret.

We know by now that an MSP430 source statement may be an assembler directive, an assembly instruction, a macro directive, or a comment. The format

is shown below, where brackets indicate an optional field. The different fields have already been explained in a previous experiment.

```
[Label[:]] [mnemonic] [operand] [; comments]
```

The MSP430G2231 memory map begins at address 0x0 and ends at address 0x0FFFF. From address 0x0 to address 0x0F it has space allocated for the special function registers (SFR). These locations must be accessed using byte instructions. From 0x010 through 0x0FF the space is allocated to the 8-bit peripherals modules, which must also be accessed using byte instructions. In the range 0x0100 to 0x01FF are allocated the 16-bit peripherals modules, which should be accessed with word instructions. The 128B of R/W RAM begin at address 0x0200 and end at 0x027F. Information memory is located in Flash in the range 01000h to 010FFh for a total of 256B. The range 0x0F800 to 0x0FFFF, 2KB also in Flash, includes program memory and the interrupt vector table located in the space 0x0FFC0 to 0x0FFFF.

The stack is simply memory where data is stored. Since this data has to be accessed and modified throughout the execution of the program, then it is clear that the stack must be located in R/W RAM. On the other hand, since the program instructions do not change throughout the execution of the program and must be in place when the unit is powered up, the instructions are located in Flash.

There are two data transfer instructions whose semantics is bound to be performed on the stack: push and pop. Whenever a push is executed, a copy of the content of the source operand is placed on the top of the stack (TOS). The way this is performed is by first decrementing by 2 the content of register SP and then copying the value onto the TOS, i.e. the 16-bit location in memory whose first byte address is pointed to by SP. This means that SP is always pointing to the TOS and that the TOS is dynamic. The pop instruction works on the other direction. That is, when a pop is executed, a copy of the 16-bit TOS is first copied onto the destination operand and then the content of SP is incremented by 2. Push and pop are both single operand instructions. The push instruction single operand is always a source operand because the destination operand is always the new TOS, i.e. the destination operand is implicit. On the other hand, the single operand for the pop instruction is always a destination operand because the source operand is always the current TOS.

The call and ret instructions also affect the stack, but they are program control instructions. When a call instruction is executed, a copy of the content of the program counter (PC), i.e. the address of the next instruction in sequence, is pushed automatically by the microprocessor onto the stack and then the content of the destination operand is copied onto the PC, effectively transferring control to the called subroutine. As part of the automatic push, the content of SP is decremented

by 2. It is the address of the first byte of the first instruction in the subroutine that gets copied onto the PC. On the other hand, when a `ret` instruction is executed, a copy of the TOS is automatically popped and copied into the PC register. As you might have already guessed, as part of this popping of the TOS the content of SP is properly incremented by 2.

In this exercise, we will be using a slight variation of our flashing LED program. Our source program is shown in Listing 3.1. Note that what we have done is to move the code for creating the delay out of the main program starting in line number 14 and ending in line number 17, and then invoke it using a `call` instruction in line number 9.

Listing 3.1: Subroutine example program

```

1      #include      msp430 . h
2  ;-----
3      ORG      0F800h  ; Program Start
4  ;-----
5  RESET  mov      #0280h,SP      ; Initialize stackpointer
6  StopWDT  mov      #MDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
7  SetupP1  bis .b   #1,&P1DIR      ; Set P1.0 as output
8  Main    xor .b   #1,&P1OUT      ; Toggle P1.0
9          call     #Wait      ; Delay
10         jmp      Main
11 ;-----
12 ;          Subroutine Wait
13 ;-----
14 Wait    mov      #50000,R15      ; Delay to R15
15 L1      dec      R15      ; Decrement R15
16         jnz     L1      ; Delay over?
17         ret      ; Go back to the caller
18 ;-----
19 ;          Interrupt Vectors
20 ;-----
21         ORG     0FFFEh  ; MSP430 RESET Vector
22         DW     RESET
23         END

```

In Section 3.3 we will be using the code in Listing 3.2 to pass the delay count to the subroutine by means of the stack using the code shown. Note that the delay count is pushed onto the stack in line number 9 before the subroutine call and then

it is retrieved in line number 15 using a pop instruction from within the subroutine body.

Listing 3.2: Subroutine example passing delay using the stack.

```

1      #include      msp430 . h
2  ;-----
3      ORG      0F800h  ; Program Start
4  ;-----
5  RESET  mov.w    #0280h,SP      ; Initialize stackpointer
6  StopWDT mov.w   #MDTPW+WDTHOLD,&WDTCTL ; Stop WDT
7  SetupP1 bis.b   #00000001b,&P1DIR ; Set P1.0 as output
8  Main   xor.b   #00000001b,&P1OUT ; Toggle P1.0
9        push    #50000
10       call   #Wait    ; Delay
11       jmp    Main
12 ;-----
13 ;           Subroutine Wait
14 ;-----
15 Wait   mov     R15,2(SP)      ; Delay to R15
16 L1     dec.w   R15           ; Decrement R15
17       jnz    L1            ; Delay over?
18       ret     ; Go back to the caller
19 ;-----
20 ;           Interrupt Vectors
21 ;-----
22       ORG    0FFFEh  ; MSP430 RESET Vector
23       DW    RESET
24       END

```

Note to the reader. The previous programs are used exclusively for explaining how to call a subroutine and how to push onto and retrieve values from the stack. If you take the time to analyze what we were doing, from an efficiency point of view, you will probably conclude that it is more efficient not to use these methods for this particular program and keep our original way of flashing the red LED. If you arrive at this conclusion, you will be right.

3.2 Procedure

1. Create a project with the first version of the project shown in Listing 3.1. If you are unsure about how to do this, you should go through Experiment 1 first.
2. Place a breakpoint at the `call` instruction, i.e. instruction in line number 9 and another one at the first instruction in the body of the subroutine, i.e. the instruction with label `Wait` in line number 14 in Listing 3.1.
3. Click `Ctrl+D` or 'Project > Download and Debug'. It will probably ask you to save your work and to create a workspace.
4. Click `F5` to run your program. It should stop just before executing the `call` instruction. Click 'View > Register' and pay particular attention to the PC register contents. It should be pointing to the address of the next instruction in sequence, i.e. the `jmp Main` instruction in line number 11.
5. Click 'Step Into' to see program control being transferred to the first instruction in the subroutine. The program will stop before executing this line. Check out the contents of the PC register. It should now be pointing to the next instruction in sequence, i.e. the second instruction in the subroutine.

3.3 Exercises

In this experiment we want you to try on your own the second version of the program, i.e. Listing 3.2. We want you to observe the behavior of both the `push` and the `pop` instructions. Thus, you should pay particular attention to the contents of registers `SP` and `R15`. Accordingly, you should place your breakpoints at the `call` and `pop` instructions. On the other hand, you could do the same without using breakpoints by clicking 'Step Into'.

Experiment 4

MSP430 Poll-based I/O

Objectives

- Become familiar with the MSP430 port 1 poll-based I/O.
- Understand how to determine that an I/O event has occurred by polling its status.

Duration

1-2 Hours

Materials

- IAR Embedded Workbench (IAR) Application.
- MSP-EXP430G2 LaunchPad development board.
- One (1) 100nF capacitor (optional).

4.1 Introduction

In previous experiments we have shown you how to control the time the LEDs in the MSP430 Launchpad are turned either on or off using software methods. We first showed you how to accomplish this task with a monolithic assembly language program in experiments 1 and 2. Then, in experiment 3, we showed you how to use a subroutine to accomplish the same task passing a parameter to the subroutine using the stack. The parameter represented the number of cycles the LED will be on or off.

In this experiment we will be using the status of pin P1.3 in the Launchpad in order to toggle the red LED on and off. This pin is connected to a pushbutton labeled S2 in the Launchpad and thus its status can be readily determined. If the button is pushed, then a zero (0) will be read, otherwise a one (1) will be read. This process of reading the status of a switch is usually known as polling. Thus, in this experiment we will be polling pushbutton S2 in the MSP430G Launchpad.

Now, among the latest revisions in the Launchpad, in revision 1.5 pull-up resistor R34 and capacitor C24 are not populated. R34 is a 47k Ω resistor whereas C24 is a 100nF capacitor. Although both components are important for this experiment, our main concern is with the absence of R34. If this resistor is not present in your board, then P1.3 will not be pulled-up to a high voltage and when the status of this pin is checked it may yield a false reading. In order to make sure that when we read a one (1) in this pin it is because we have a one (1), we must make sure that we have a pull-up resistor connected to P1.3. Fortunately, the MSP430 included with the MSP430G Launchpad have internal resistors that can be configured to be connected to the pins in port P1. This is accomplished by setting to one (1) the corresponding bit in P1REN and P1OUT. If you need to enable the internal pull-up resistor for pin P1.3, then you need to select bit 3 in P1REN and then make it a pull-up by sending a one (1) to the corresponding bit in P1OUT. In other words, insert `bis .b #00001000b,&P1REN` and `bis .b #00001000b,&P1OUT` before the instruction in line 12, i.e. before executing instruction `bit .b #00001000b,&P1IN`.

Pushbutton S2 is a mechanical switch and thus it bounces both when it is pushed and also when it is released. Accordingly, when we push S2, although we pushed it only once, the contacts will bounce. Thus, since the microprocessor is fast enough, we could detect several pushes when even though we pushed it just once. We could also detect a one (1) during bouncing when the button was pushed or a zero (0) when the button is released. There are several strategies to go around this problem using either hardware or a software solution. Since an in-depth solution of this problem is beyond the scope of this experiment we will only suggest the following: If your Launchpad lacks capacitor C24 you should connect an external capacitor to P1.3 similar to C24, the other end of the capacitor should be connected to ground. **Note that if your Launchpad has R34 and C24 already installed you do not need to do any of these corrective steps because the circuit including these two components already takes care of pulling P1.3 up and debouncing S2.**

You will use the following assembly code for your program.

Listing 4.1: Poll-based I/O example program

```
1 #include msp430.h
```

```

2 ;-----
3         ORG      0F800h  ; Program Start
4 ;-----
5 RESET   mov.w    #0280h,SP      ; initialize stackpointer
6 StopWDT mov.w    #WDTPW+WDTHOLD,&WDTCTL ; stop WDT
7         bic.b    #00001000b,&P1DIR      ; P1.3 as input port
8         bis.b    #1,&P1DIR      ; P1.0 as output port
9         bic.b    #1,&P1OUT      ; red LED off
10        ;        bis.b    #00001000b,&P1REN; select internal resistor
11        ;        bis.b    #00001000b,&P1OUT; make it pull-up
12 POLL   bit.b    #00001000b,&P1IN      ; poll P1.3
13        jz      POLL      ; again if zero
14 ;-----
15        ;          P1.3 Service Code
16 ;-----
17 PBSCode xor.b    #00000001b,&P1OUT      ; toggle red LED
18        jmp     POLL
19 ;-----
20        ;          Interrupt Vectors
21 ;-----
22        ORG      0FFFEh  ; MSP430 RESET Vector
23        DW      RESET   ; address of label RESET
24        END

```

This is pretty much the same program we have been using all along with some modifications. You should notice that we have removed the delay part of the program which accounted for how long the LED would be on or off. In this experiment the delay part of the program is provided by your pushing the pushbutton. We will explain only some of the instructions since the rest of them have been explained in previous experiments.

In Listing 4.1 instruction `bit.b #00001000b,&P1IN` at line number 12 is testing pin 3 on port P1 (P1.3) in order to determine whether the pushbutton is pushed. If it is not pushed, then the program will continue to poll the status of the pushbutton. Otherwise, execution will continue at the instruction labeled `PBSCode`, i.e. the red LED is toggled, and execution is again given back to the polling instruction using an unconditional jump instruction at line 18.

4.2 Procedure

1. Create a project and use the above program for your “.asm“ file. You could name your project PBFlashLED. Your “.asm“ file could be named PBInterrupt.
2. Click Ctrl+D or ‘Project > Download and Debug’. It will probably ask you to save your work and to create a workspace.
3. Click F5 to run your program. If everything is ok, and it should, nothing is happening to the red LED. Now push the push button. The red LED should toggle each time you push it.

4.3 Exercises

1. Place breakpoints at the instructions containing `bit .b` in line number 12 and `jz` in line 13 so that you can watch the contents of these bits before and after the instructions are executed.
2. Remove the breakpoints you placed in the previous step, and place a breakpoint in the instructions with `xor .b` in line number 17 and `jmp` in line 18. This should allow you to stop the execution of the program and how the status of the red LED is toggled just prior to jumping back to line 12.

Experiment 5

MSP430 Interrupt-based I/O

Objectives

- Become familiar with the MSP430 port 1 interrupt-based I/O and interrupt capabilities in general.
- Understand the MSP430 interrupt sequence and how to manage the interrupt vector table.

Duration

2 Hours

Materials

- IAR Embedded Workbench (IAR) Application.
- MSP-EXP430G2 LaunchPad development board.

5.1 Introduction

In previous experiments we have shown you how to control the time the LEDs in the MSP430 Launchpad are turned either on or off using software methods. We first showed you how to accomplish this task with a monolithic assembly language program in experiments 1 and 2. Then, in experiment 3, we showed you how to use a subroutine to accomplish the same task passing a parameter to the subroutine using the stack. The parameter represented the amount of cycles the LED will be on or off. In experiment 4 we showed you how to do it by polling the status of the pushbutton labeled S2 in the LaunchPad.

In this experiment we will be examining the interrupt capabilities of the MSP430G2231 general purpose input/output ports. In particular, we will be using pin P1.3 in the Launchpad to generate an interrupt to control the toggling of the red LED. This pin is connected to a push button labeled S2 in the Launchpad and thus can be used to generate an interrupt to the microprocessor inside the MSP430.

Remember that when a call to a subroutine is executed control is transferred to the first instruction in the body of the subroutine (the callee) after some house-keeping is performed to secure the contents of the program counter register (PC) so that control can be transferred back to the caller as part of the execution of a ret instruction. Without saving the address of the instruction immediately following the call the microprocessor would have no way to return back to the place it was at the time of the call. As mentioned before, the content of the PC is saved by pushing it onto the stack. As part of the semantics of executing the ret instruction, the microprocessor will get the address it needs to return to by copying the current value at the top of the stack (TOS) into register PC. From that point on execution will continue from the instruction immediately following the call.

Generally speaking, there are two classifications of interrupts: maskable interrupts and non-maskable interrupts. Maskable interrupts are governed by the state of a general interrupt enable bit. If this bit is enabled, then maskable interrupts will be recognized and serviced accordingly. Otherwise, the interrupt will be ignored. In the MSP430 microcontrollers this bit is called the global interrupt enable (GIE) bit and it is located in bit 8 of the status register (SR). You can globally enable maskable interrupts for the MSP430 by executing the assembly language instruction eint. You can globally disable maskable interrupts by executing dint. There could also be another interrupt enable bit which is local to the peripheral you are using as in the case of this experiment. We will not elaborate on non-maskable interrupts other than to say that this type of interrupts cannot be ignored when they occur. Non-maskable interrupts are beyond the scope of this experiment.

When a maskable interrupt which is enabled both locally and globally occurs the microprocessor begins the interrupt sequence. The interrupt sequence is, as its name implies, a sequence of steps that will make the microprocessor save its current state and transfer control to the first instruction in the interrupt service routine (ISR). When the microprocessor executes the reti instruction (not the ret) within the ISR, control is transferred back to the interrupted program. Please, note that in order to properly return from a hardware invoked ISR we need to use reti and not the ret instruction used for software invoked subroutines. They may look similar, but they are very different.

The first thing that the microprocessor does when an interrupt is recognized

and must be serviced is that, in case an instruction is being executed, it finishes execution of the current instruction. Then the content of the PC, which is pointing to the next instruction in sequence in the interrupted program, is pushed onto the stack. A copy of the appropriate interrupt vector is placed into the PC. Then the content of register SR is pushed onto the stack and SR is cleared afterwards. Each time an item is pushed onto the stack register SP is first updated by adding two (2) to its contents and then the item is copied into the new TOS. Note that one thing that happens when SR is cleared is that the GIE bit is cleared. This means that the microprocessor cannot be interrupted by another (or the same) maskable interrupt while servicing an interrupt. Control is now transferred to the instruction whose address is loaded into PC, i.e. the address of the first byte of the first instruction in the ISR, which is obtained from the interrupt vector, and execution of the ISR begins. The process described above takes 6 clock cycles. When the microprocessor executes a `reti` instruction, the SR register is loaded with a copy of the TOS, the SP register is updated by subtracting two (2) from its current contents, now a copy of this new TOS is copied into the PC, two (2) is subtracted again from the SP register to update it, and control is transferred to the instruction whose address is the contents of register PC. This process takes 5 additional clock cycles. The total overhead for an interrupt is thus 11 clock cycles.

We mentioned previously that a copy of the interrupt vector is copied into the PC as part of the interrupt sequence. Since the MSP430 can be interrupted by several sources, the designers decided that a way to discriminate among the different sources needed to be put in place. Thus, a precedence number was assigned to the different sources and the way to determine the precedence is by using an interrupt vector table. This table is usually placed in flash and extends from location `0xFFE0` to `0xFFFF` in the MSP430G2231. Thus, if more than one interrupt source is active when an interrupt is recognized, the interrupt with the highest interrupt vector address within the interrupt vector table will be serviced. In this experiment we will generate an interrupt from port P1 whose interrupt vector is at location `0xFFE4`. Note that an interrupt vector always starts in an even numbered address and that it takes up 16 bits. Thus, there are 16 vectors in the interrupt vector table in the MSP430G2231.

You will use the following assembly code for your program.

Listing 5.1: Interrupt based I/O.

```

1      #include          msp430 . h
2  ;-----
3      ORG          0F800h  ; Program Start
4  ;-----
```

```

5 RESET   mov.w   #0280h,SP           ; initialize SP
6 StopWDT mov.w   #MDTPW+WDTHOLD,&WDTCTL ; stop WDT
7         bic.b   #00001000b,&P1SEL    ; default
8         bic.b   #00001000b,&P1DIR    ; P1.3 as input
9         bis.b   #1,&P1DIR            ; P1.0 as output
10        bic.b   #1,&P1OUT            ; red LED off
11 ;     bis.b   #00001000b,&P1REN; select internal resistor
12 ;     bis.b   #00001000b,&P1OUT; make it pull-up
13        bis.b   #00001000b,&P1IE    ; enable P1.3 int.
14        eint   ; global interrupt enable
15 HERE   jmp     HERE               ; wait
16 ;-----
17 ;           P1.3 Interrupt Service Routine
18 ;-----
19 PBISR   bic.b   #00001000b,&P1IFG    ; clear int. flag
20        xor.b   #1,&P1OUT            ; toggle red LED
21        reti   ; return from ISR
22 ;-----
23 ;           Interrupt Vectors
24 ;-----
25        ORG    0FFFEh ; MSP430 RESET Vector
26        DW     RESET ; address of label RESET
27        ORG    0FFE4h ; interrupt vector 2
28        DW     PBISR ; address of label PBISR
29        END

```

This is pretty much the same program we have been using all along. What we did this time to move the toggling face inside an ISR called PBISR which begins at the instruction in line number 19. You should also notice that we have removed the delay part of the program which accounted for how long the LED would be on or off. In this experiment the delay part of the program is provided by your pushing the pushbutton. We will explain only some of the instructions since the rest of them have been explained in a previous experiment.

Instruction `bic.b #00001000b,&P1SEL` in line 7 is clearing bit 3 in register P1SEL. This instruction is not really necessary since the default for register P1SEL is that it is cleared. If we have a bit cleared in P1SEL we are telling the peripheral, in this case port P1, that that pin is allowed to interrupt. Instruction `bis.b #00001000b,&P1IE` is locally enabling the interruption in pin 3 on port P1. This is accomplished by setting P1IE.3 to a 1. After the previous instruction we find `eint` in line 14. As already explained, `eint` globally enables maskable interrupts by

setting bit GIE in register SR to a 1. If you do not enable a maskable interrupt both locally and globally, that interrupt will not be serviced. After `eint` the program just waits for an interrupt to occur.

If the pushbutton is pushed, control will be transferred from the main program to the PBISR interrupt service routine. The first thing that is done inside the ISR is to clear the flag that was set when the interrupt was received on P1. This is accomplished with the instruction `bic.b #00001000b,&P1IFG` in line 19. What this does is to clear the port 1 interrupt flag. Since only P1.3 is allowed to interrupt, we do not have to check which of the 8 pins included in P1 generated the interrupt, nor do we need to discriminate among them. After this we toggled the status of the red LED and return from the interrupt with instruction `reti`.

The one thing we have left to explain is how the microprocessor got to the ISR. Note that we used a label PBISR in the first instruction of the ISR. There is nothing new about using a label, but we then used this label as part of a directive in the very same way we have done in previous experiments and in this with label RESET. Using the `ORG 0xFFE4` directive in line 27 we instruct the assembler to do what the `DW PBISR` directive is telling it to do, i.e. to reserve a 16 bit memory location beginning at the address 0xFFE4, the interrupt vector location for port P1, and initialize it with the address of the first byte of the instruction labeled PBISR, i.e. the first instruction in the ISR. Now, when the pushbutton connected to P1.3 is pushed, the microprocessor will execute the ISR thus toggling the red LED.

5.2 Procedure

1. Create a project and use the above program for your “.asm“ file. You could name your project PBFlashLED. Your “.asm“ file could be named PBInterrupt.
2. Click Ctrl+D or ‘Project > Download and Debug’. It will probably ask you to save your work and to create a workspace.
3. Click F5 to run your program. If everything is ok, and it should, nothing is happening to the red LED. Now push the pushbutton. The red LED should toggle each time you push it.

5.3 Exercises

1. Place breakpoints at the instructions containing `bis.b` in line number 13 and `eint` in line 14 so that you can watch the contents of these bits before and after the instructions are executed.
2. Remove the breakpoints you placed in the previous step, and place a breakpoint in the first and last instruction inside the ISR, i.e. the instructions with `bic.b` in line number 19 and `reti` in line 21. This should allow you to stop the execution and watch how the content of `P1IFG` is cleared and how the status of the red LED is toggled just prior to returning to the interrupted program.

Experiment 6

MSP430 Timer_A

Objectives

- Become familiar with the MSP430 Timer_A
- Understand how to configure the MSP430 Timer_A to generate the timing needed to control turning on and off the LED.

Duration

2 Hours

Materials

- IAR Embedded Workbench (IAR) Application.
- MSP-EXP430G2 LaunchPad development board.

6.1 Introduction

Thus far we have used several features of the MSP430G2231 to control the toggling on and off of one or both of the onboard LEDs. We have used both software and hardware features to accomplish this toggling of LEDs but none of our timing sources so far could be said to be accurate. One of the main features of the MSP430 family of microcontrollers is its ability to generate the timing signals you need without having to add any hardware. The MSP430G2231 has two different timers: the watchdog timer (WDT) and Timer_A. In this experiment we will be using Timer_A to generate the timing needed to control the LEDs included as part of the MSP-EXP430G2 LaunchPad development board.

Timer_A is a 16-bit timer that includes an asynchronous 16-bit timer/counter TAR with four operating modes (Stop, Up, Continuous, and Up/Down), it has a selectable and configurable clock source able to select from four different clock sources: two internal (ACLK and SMCLK) and two external (TACLK and INCLK), three (3) configurable capture/compare registers (CCR0-2), configurable outputs with pulse width modulation (PWM) capability, asynchronous input and output latching, and interrupt vector register for fast decoding of all Timer_A interrupts.

There are two interrupt vectors associated with Timer_A in the MSP430G2231. Their locations in the interrupt vector table are 0FFF2h and 0FFF0h with interrupt priorities 25 and 24, respectively. Both interrupts are maskable. Location 0FFF2h is used for Timer_A capture compare register 0 (TACCR0) and location 0FFF0h is used for Timer_A capture compare register 1 (TACCR1). In this experiment we will be using TACCR0, which has TACCR0 CCIFG as interrupt flag.

Now, in the up mode the timer will count clock pulses starting at zero (0) up to the value stored in TACCR0 or CCR0 at which point it will set its interrupt flag to a one (1), i.e. TACCR0 CCIFG equals one (1) indicating that an interrupt is pending. In the next clock pulse the timer will begin counting again from zero (0) and at that point it will set flag TAIFG to one (1). As long as Timer_A is operating in the up mode it will continue to behave as explained above. Before we go on we should point out the fact that when the CPU enters an interrupt service routine (ISR) for Timer_A, its flag is reset, i.e. TACCR0 CCIFG is automatically reset to zero (0). This makes it unnecessary for our code to have to reset the flag to zero (0) and accordingly we will not do so.

As mentioned above, Timer_A can be sourced from four (4) different clock sources. In order to choose the clock source you send two bits to a 4-by-1 multiplexer using Timer_A clock select signals (TASSELx), i.e. bits 8 and 9 in the Timer_A control register (TACTL). If you send the combination 00 you will be choosing the TACLK clock, if 01 is sent instead ACLK will be used. Sending 10 will choose SMCLK and 11 chooses clock INCLK. The corresponding clock pulses are divided by either 1, 2, 4, or 8, depending on bits 6 and 7 in the TACTL register. These two bits form a field called IDx or input divider. A 00 in IDx will divide the clock pulses by 1, a 01 will divide them by 2, a 10 will divide them by 4, and a 11 will divide them by 8. Thus, the count value in timer A register TAR is really the clock source chosen with TASSELx divided by the value specified by IDx. Finally, Timer_A mode of operation is chosen by specifying the value in bits 4 and 5 in register TACTL. These two bits are known as field MCx. An MCx value of 00 chooses the "Stop mode" in which Timer_A is halted; 01 chooses the "Up mode" in which the timer counts from zero (0) up to the value stored in TACCR0; 10 chooses "Continuous mode" and counts

from zero (0) up to 0FFFFh; and 11 chooses “Up/down mode” in which the timer counts from zero (0) to TACCR0 then down to 0000h.

The code that you will be using is shown below:

Listing 6.1: Timer_A programming.

```

1      #include msp430.h
2  ;-----
3      ORG      0F800h ; Program Start
4  ;-----
5  RESET  mov.w  #0280h,SP      ; initialize SP
6  StopWDT mov.w #WDTPW+WDTHOLD,&WDTCTL ; stop WDT
7          bis.b  #BIT0,&P1DIR   ; P1.0 as output port
8          bic.b  #BIT0,&P1OUT   ; initialize red LED off
9  SetupC0 mov.w #CCIE,&CCTL0   ; enable CCR0 interrupt
10         mov.w  #50000,&CCR0
11  SetupTA mov.w #TASSEL_2+MC_1,&TACTL ; use SMCLK, up mode
12         bis.w  #GIE+LPM0,SR   ; enable interrupts and
13         ; enter low power mode 0
14         nop      ; required for debugger
15 ;-----
16 ;          TIMER_A TACCR0 Interrupt Service Routine (ISR)
17 ;-----
18 TACCR0_ISR  xor.b  #BIT0,&P1OUT ; toggle red LED
19             ; CCIFG automatically reset
20             ; when TACCR0 ISR is serviced
21         reti          ; return from ISR
22 ;-----
23 ;          Interrupt Vectors
24 ;-----
25         ORG      0FFFEh ; MSP430 RESET Vector
26         DW      RESET ; address of label RESET
27         ORG      0FFF2h ; interrupt vector (TACCR0)
28         DW      TACCR0_ISR ; address of label TACCR0_ISR
29         END

```

The instruction SetupC0 `mov.w #CCIE,&CCTL0` sets bit four (4) in the Timer_A capture compare control register TACCTL0 or CCTL0 to a 1 enabling interrupts originating in CCR0. The next instruction `mov.w #50000,&CCR0` loads the decimal value 50,000 to capture compare register zero (TACCR0 or CCR0). Thus, Timer_A will count from zero (0) to 50,000 at which point it will set TACCR0

CCIFG to one (1) and interrupt the CPU effectively waking it up. The next two (2) instructions

```
SetupTA  mov.w    #TASSEL_2+MC_1,&TACTL
          bis.w    #GIE+LPM0,SR
```

configure TACTL so that Timer_A will be sourced from SMCLK and the operating mode will be “Up mode“ and load the status register SR in such a way that interrupts are globally enabled and low power mode 0 (LPM0) is selected, i.e. bits 3 and 4 in SR are set to one (1). The MSP430 can be placed into one of several low power modes. Each of the low power modes disables the CPU. As the digit in the low power mode scheme increases, additional peripheral devices are disabled to conserve energy. LPM0 means that the CPU will be off or disabled. LPM4 means that the CPU and all clocks are disabled. Except for LMP0, all low power modes disable the SMCLK clock. Since we need to use SMCLK we cannot disable it.

As shown in the comment the instruction nop is used to allow the debugger to work properly. The instruction following an enable interrupt is always executed. The “no operation“ instruction, as its name implies, does nothing except to consume one (1) clock cycle. Also when using the debugger there is the possibility that control is given back to the debugger when single stepping over an instruction that manipulates flash before flash manipulation is complete, thus displaying wrong information. The nop instruction is used to allow the debugger to synchronize.

6.2 Procedure

1. Create a project and use the above program for your .asm file. You could name your project TAFlashLED. Your “.asm“ file could be named TACCR0.
2. Click Ctrl+D or ‘Project > Download and Debug’. It will probably ask you to save your work and to create a workspace.
3. Click F5 to run your program. The red LED should toggle every 50,000 SMCLK clock periods.

6.3 Exercises

1. Place breakpoints to allow you to see how the bits of Timer_A capture compare control register CCTLO and Timer_A control register TACTL are affected.

2. Find the amount of time the 50000 number used in the code above corresponds to.
3. Modify the code shown above to count the SMCLK clock pulses divided by 8.
4. Modify the code shown above to count the ACLK clock pulses.

Experiment 7

MSP430 Interrupt-based I/O using the C Language

Objectives

- Become familiar with programming the MSP430 using the C language.
- Review the MSP430 interrupt sequence and how to manage the interrupt vector table.

Duration

2 Hours

Materials

- IAR Embedded Workbench (IAR) Application.
- MSP-EXP430G2 LaunchPad development board.

7.1 Introduction

In previous experiments we have shown you how to control the time the LEDs in the MSP430 Launchpad are turned either on or off using software methods. From experiments 1 thru 4 we used purely software methods. Then in experiment 5 and 6 we added the powerful concept of hardware interrupts. The underlying software vehicle for performing all of these experiments has been the MSP430 assembly language. In this experiment we will be using the C high level language to program the MSP430.

EXPERIMENT 7. MSP430 INTERRUPT-BASED I/O USING THE C LANGUAGE48

In particular, we will be performing a task similar to what we did in experiment 5 but this time we will do everything using C.

In this experiment we will be examining the interrupt capabilities of the MSP430G2231 general purpose input/output ports. In particular, we will be using pin P1.3 in the Launchpad to generate an interrupt to control the toggling of the red LED. This pin is connected to a push button labeled S2 in the Launchpad and thus can be used to generate an interrupt to the microprocessor inside the MSP430.

Remember that there are two classifications of interrupts: maskable interrupts and non-maskable interrupts. Maskable interrupts are governed by the state of a general interrupt enable bit. If this bit is enabled, then maskable interrupts will be recognized and serviced accordingly. Otherwise, the interrupt will be ignored. In the MSP430 microcontrollers this bit is called the global interrupt enable (GIE) bit and it is located in bit 8 of the status register (SR). You can globally enable maskable interrupts for the MSP430 by executing the assembly language instruction `eint`. You can globally disable maskable interrupts by executing `dint`. There could also be another interrupt enable bit which is local to the peripheral you are using as in the case of this experiment. Non-maskable interrupts are beyond the scope of this experiment.

We mentioned previously that a copy of the interrupt vector is copied into the PC as part of the interrupt sequence. Since the MSP430 can be interrupted by several sources, the designers decided that a way to discriminate among the different sources needed to be put in place. Thus, a precedence number was assigned to the different sources and the way to determine the precedence is by using an interrupt vector table. This table is usually placed in flash and extends from location `0xFFE0` to `0xFFFF` in the MSP430G2231. Thus, if more than one interrupt source is active when an interrupt is recognized, the interrupt with the highest interrupt vector address within the interrupt vector table will be serviced. In this experiment we will generate an interrupt from port P1 whose interrupt vector is at location `0xFFE4`. Note that an interrupt vector always starts in an even numbered address and that it takes up 16 bits. Thus, there are 16 vectors in the interrupt vector table in the MSP430G2231.

You will use the following C code shown in Listing 7.1 for your program. Although it may not be immediately obvious, this program in C accomplishes the same thing as the assembly language version presented in experiment 5.

Listing 7.1: Interrupt based I/O using C.

```
1      #include      'msp430.h'  
2      #include      'intrinsics.h'
```

EXPERIMENT 7. MSP430 INTERRUPT-BASED I/O USING THE C LANGUAGE 49

```
3 void    main(void)
4 {
5     WDTCTL = WDTPW + WDTHOLD; // stop WDT
6     P1DIR = P1DIR | 0x01; // P1.0 as output
7     P1OUT = P1OUT & ~0x01; // red LED off
8     P1IE = P1IE | 0x08; // enable P1.3 interrupt
9     // P1REN = P1REN | 0x08; // select P1.3 internal resistor
10    // P1OUT = P1OUT | 0x08; // as pull-up
11    __bis_SR_register(GIE); // global interrupt enable
12    here: ; // wait
13    goto here;
14 }
15 ;-----
16 ;           P1.3 Interrupt Service Routine
17 ;-----
18 #pragma vector = PORT1_VECTOR
19 __interrupt void port1_pin3_interrupt_service_routine(void)
20 {
21     P1IFG = P1IFG & ~0x08; // clear int. flag
22     P1OUT = P1OUT ^ 0x01; // red LED off
23 }
```

We will now explain the instructions in Listing 7.1. Line number 2 is not an instruction in C language but a directive. We need this directive in order to be able to use the intrinsic in line number 11. Intrinsic are an efficient way to accomplish inside a C program what some specific assembly language instruction does without having to include assembly language instructions. Line number 3 is the declaration for the main function. Every program in C must have one function and that function's name is main. We are using void for both the value returned by main and for its actual parameter because in this experiment we are not returning any parameter from main and we are not passing any parameter to main.

The instruction in line number 5 is an assignment instruction in C language and is used to stop the MSP430 watchdog timer. Next, in line number 6 we use the bitwise OR operator in the C language, i.e. |, to make sure that pin P1.0 is set as an output pin. In line number 7 we use the bitwise AND operator in C, i.e. &, and the bitwise complement operator ~. The rules of the C language state that the bitwise complement will be taken first and then the bitwise AND will take place. Thus, 0x01 will be converted into 0xFE or 11111110b and then the contents of P1OUT will be bitwise anded with 0xFE effectively clearing bit 0 in port P1.

EXPERIMENT 7. MSP430 INTERRUPT-BASED I/O USING THE C LANGUAGE 50

For your reference, these are the operators available as part of the C language. They are shown in order of precedence, shown as Pre in the table. Note that the bitwise complement operator `~` has a precedence of 2 whereas the bitwise or operator `|` has a precedence of 10 and the assignment operator `=` has a precedence of 14. We remind you that, in C, the lower the precedence number the higher the precedence of the operator.

| Pre | Op | Descrip | Pre | Op | Descrip | Pre | Op | Descrip |
|-----|--------|----------|-----|----|----------|-----|-----|---------|
| 1 | () | parenth | 3 | / | div | 11 | && | and |
| 1 | [] | subscr | 3 | % | mod | 12 | | or |
| 1 | . | dir memb | 4 | + | add | 13 | ?: | cond |
| 1 | -> | ind memb | 4 | - | substr | 14 | = | assign |
| 2 | ++ | incr | 5 | << | shift l | 14 | += | assign |
| 2 | -- | decr | 5 | >> | shift r | 14 | -= | assign |
| 2 | * | deref | 6 | < | lt | 14 | *= | assign |
| 2 | & | ref | 6 | <= | le | 14 | %= | assign |
| 2 | ! | neg | 6 | > | gt | 14 | = | assign |
| 2 | ~ | bw comp | 6 | >= | ge | 14 | >>= | assign |
| 2 | + | unary + | 7 | == | eq | 14 | <<= | assign |
| 2 | - | unary - | 7 | != | ineq | 14 | &= | assign |
| 2 | sizeof | size | 8 | & | bw and | 14 | ^= | assign |
| 2 | (cast) | cast | 9 | ^ | bw ex-or | 15 | , | comma |
| 3 | * | mult | 10 | | bw or | | | |

In line number 8 we again use the bitwise OR operator in C only this time we are it to set pin 3 in P1IE to 1. By doing this we are locally enabling pin P1.3 to interrupt the microprocessor. This by itself does nothing unless the global interrupt enable (GIE) bit in register SR is set to a 1. Now in line 11 we globally enable maskable interrupt sources to interrupt the microprocessor. This is accomplished using `__bis_SR_register(GIE)`. This is one of the intrinsics found in file `intrinsics.h` and what it does is, as its name implies, to enable interrupts at the global level by setting to one (1) the GIE bit in register SR. The empty label here in line number 12 is used as a destination for the `goto` instruction. A better way to do this is to send the microprocessor to sleep using one of the low power modes as it was shown in experiment 6, but since we wanted to show you how to implement experiment 5 we had to use the `goto` instruction in line 13. Indeed, the `goto` instruction in C is equivalent to the unconditional `jmp` instruction in assembly language. This is the end of the main program in our C code.

EXPERIMENT 7. MSP430 INTERRUPT-BASED I/O USING THE C LANGUAGE 51

If the pushbutton is pushed, control will be transferred from the main program to the port 1 interrupt service routine (ISR). The interrupt vector corresponding to the port 1 interrupt is loaded with the address of the function which follows it. This is accomplished using the `pragma` directive in line 18 and the function declaration beginning in line 19. The first thing that is done inside the ISR is to clear the flag that was set when the interrupt was received on P1. This is accomplished with the instruction in line 21. What this does is to clear the port 1 interrupt flag. Since only P1.3 is allowed to interrupt, we do not have to check which of the 8 pins included in P1 generated the interrupt, nor do we need to discriminate among them. After this we toggle the status of the red LED with instruction 22 and then return from the interrupt.

The total number of actual C instructions in our program is 8, we do not count directives as instructions. On the other hand, the total number of actual assembly language instructions in our program in experiment 5 was 12, again we do not count the directives. Do not be deceived by this and jump to wrong conclusion. Regardless of the programming language that is used, a microprocessor can only understand its own machine language. As we have already seen in our previous experiments, there is a 1-to-1 correspondence between the number of instructions in assembly language and the number of instructions in machine language. This means that if our assembly language program has 12 instructions, then its machine language version will also have 12 instructions. We will see that this 1-to-1 correspondence does not hold for every C language instruction and, in general, a program written in a high level language will produce one or more machine language instruction for every high level language instruction.

7.2 Procedure

1. In IAR click 'Project > Create New Project'. Make sure 'MSP430' is selected in the 'Tool chain:' pull down menu.
2. Now Select 'Empty project' in the 'Project templates:' window. Click 'OK'.
3. Make sure you choose 'workspace' in the 'Documents Library' window. Enter PBFlashLED in the 'Filename:' text box in the 'Save As' window. Click 'Open' and then click 'Save'.
4. Select PBFlashLED in the project tree.
5. Click 'File > New File' and enter the code shown in Listing 7.1.

EXPERIMENT 7. MSP430 INTERRUPT-BASED I/O USING THE C LANGUAGE⁵²

6. Click 'File > Save As' and enter 'PBInterrupt'. After the filename is accepted you should see the tab name changing to 'PBInterrupt.c'.
7. Click 'Project > Add Files' and choose 'PBInterrupt.c'. After the file is accepted you should see the project tree changing to reflect this fact.
8. Click 'Project > Options'. While in the 'General Options' category go to the 'Target' tab and choose 'MSPGxxx Family' in the 'Device' pulldown menu. When the new options appear choose 'MSP430G22311'. If you are using a different device, e.g. MSP430G2221, MSP430G2452, MSP430G2553, etc., then make sure that you perform the corresponding steps to choose the device you have. Otherwise you will get errors when trying to run your program.
9. In the 'Debugger' category go to the 'Setup' tab to the right and choose 'FET Debugger' under 'Driver'. Make sure that you distinguish between the 'FET Debugger' in the 'Setup' tab and the 'FET Debugger' as a subcategory of the 'Debugger' category. The 'Debugger' category has two subcategories: 'FET Debugger' and 'Simulator'. At the same time the 'Setup' tab has two alternatives under the 'Driver' pulldown menu: 'Simulator' and 'FET Debugger'.
10. Now go to the 'FET Debugger' subcategory under the 'Debugger' category and make sure that 'Texas Instrument USB-IF' is selected. Click 'OK'.
11. Make sure that your MSP430 is connected before proceeding.
12. Click Ctrl+D or 'Project > Download and Debug'. It will probably ask you to save your work and to create a workspace, if this is the case, give a name to the workspace such as 'PushButton' in the 'File name' text box.
13. Click F5 to run your program. If everything is ok, and it should, nothing is happening to the red LED. Now push the pushbutton. The red LED should toggle each time you push it.

7.3 Exercises

1. Examine the 'Memory' window to the right of your screen. Make sure you go to the 0F800 address. You should see a lot of information here. For example, you will see the instructions in machine language, the instructions in C and the corresponding instructions in assembly language. Some instructions in C language correspond to just one instruction in assembly and machine language. However, some instructions in C correspond to more than one instruction in

EXPERIMENT 7. MSP430 INTERRUPT-BASED I/O USING THE C LANGUAGE 53

assembly language. You may also notice that the C compiler took care for you of providing the entry point address and loading the SP register.

2. Set breakpoints at your C language instructions to stop the execution and watch how the content of P1IFG is cleared and how the status of the red LED is toggled just prior to returning to the interrupted program.
3. Show the sequence of instructions in assembly language to
 - Enable Timer_A Capture Compare Register 1 interrupt.
 - Load Capture Compare Register 1 with 32768.
 - Select the SMCLK clock.
 - Configure the timer for continuous mode of operation.
4. An Engineering student needs to generate a square waveform to drive a device. The student decides to use the MSP430's Timer_A with a duty cycle of 60 percent at 32.768 khz. Show the sequence of instructions in assembly language that the student might have used to configure Timer_A to accomplish the requirements.
5. Setup the MSP430 LaunchPad to use Timer_A to toggle the red LED each time the timer counts to 50,000. You should be able to do this using:
 - A monolithic assembly language program.
 - An assembly language program that calls a subroutine to configure Timer_A to toggle the LED every 50,000 cycles.
 - An assembly language program that sets up an interrupt from Timer_A every 50,000 cycles.
 - Repeat the previous parts using the C language.
6. It was stated in this chapter that the watchdog timer could be configured as an interval counter. Configure the MSP430 Watchdog Timer to toggle the red LED in the MSP430 LaunchPad every three (3) seconds. To conserve energy the MSP430 needs to enter a low power mode after your configuration is complete.
7. Write a small program to test the WDT+ failsafe feature of the MSP430. Have the MSP430 perform some function like toggling one of the LaunchPad LEDs and then attempt to disable all clocks.

Experiment 8

The MSP430X

8.1 Introduction

The MSP430X (CPUx) is a 20-bit version of the Texas Instruments MSP430 microcontroller. It was introduced with the second generation of MSP430 chips, i.e. with the MSP430x4xx family of microcontrollers, and it has been present with all subsequent versions, except for the MSP430x1xx.

The CPUx 20-bit address bus allows it to reach the 1-MB address space without paging. Remember that the traditional MSP430 is a 16-bit processor both at its address and data buses and can reach a 64-KB address space. The CPUx is backward compatible with the MSP430 CPU and it can address bytes, 16-bit words, and 20-bit words. It maintains its orthogonal RISC architecture allowing any CPU register to be used as an operand. Several instructions have been extended for 20 bit operation. However, using a prefix any instruction can be extended to 20 bit. It (CPUx) has fewer interrupt overhead cycles and fewer instruction cycles in some cases than the MSP430 CPU.

8.2 The CPUx Architecture

Figure 8.1 shows the MSP430X block diagram. As it can be seen in the figure, except for the SR register, all the MSP430X registers are 20-bit long. Note also that the data bus remains at 16-bit while the address bus is now 20-bit wide.

8.3 Differences between the CPUx and the MSP430

As already mentioned before, the main difference between the MSP430X and the regular MSP430 CPU is its 20 bit address bus. In addition to this and except for the SR register, all the CPUx registers were extended to 20 bits. There are several instructions that were extended to take advantage of this increased address space,

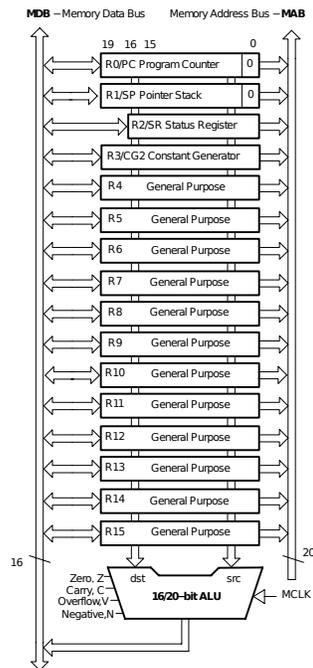


Figure 8.1: MSP430X (CPUX) Block Diagram.

namely: MOVA, CALLA, ADDA, SUBA, CMPA. ADDA, SUBA, and CMPA are restricted to the immediate and register addressing modes only, i.e. you cannot use the rest of the addressing modes with these three (3) instructions.

There are several instructions for performing multi-bit shifts (1, 2, 3, or 4 bits): RRCM, RRAM, RLAM, RRUM. You can also push or pop several registers with the instructions PUSHM and POPM. You can push or pop anywhere from 1 to 16 registers with these instructions. Interrupt latency is 5 cycles for the CPUx vs 6 cycles for the regular MSP430. Returning from an interrupt is 3 cycles for the CPUx vs 5 cycles for the MSP430.

By using an additional word of op-code called an extension word, all addresses, indexes, and immediate values are extended to 20 bit. For example, ADDX adds the source word to the destination word, ADDX.B adds the source byte to the destination byte, and ADDX.A adds the source address-word to the destination address-word. On the other hand, BISX sets bit set in the source word in the destination word, BISX.B sets bit set in the source byte in the destination byte, and BISX.A sets bit set in the source address-word in the destination address-word. You can do similar operations with ADDX, ADDCX, ANDX, BICX, BISX, BITX, CMPX, DADDX, MOVX, POPM, PUSHM, PUSHX, RLAM, RRAM, RRAX, RRCM, RRCX, RRUM, RRUX, SUBX, SUBCX, SWPBX, SXTX, and XORX.