The EZ430 Microcontroller: A New Tool For EE251

Miguel Morales
morales.amiguel@gmail.colostate.edu

Bill Eads
eadsinco@mesanetworks.net

Abstract

As microcontroller technologies evolve, so should the tools used to educate the engineers in electrical and computer engineering universities nationwide. The EZ430 is a full development kit, implemented in a USB stick format that presents many benefits to an introduction to microcontroller classroom. It employs a very recent technology; the software comes free with the purchase of the tool and is easy to use; it is portable, modular, and inexpensive. Since the EZ430 is so new, little classroom material exists for a classroom implementation. Hence, this project presents a written set of labs to move towards a set of materials ready for classroom use. The project will be completed by the end of the fall semester of 2006 by a continuing group of senior design students, resulting eight full labs, tested and approved by a set of students throughout a semester’s time.
Table of Contents

Acknowledgements 3
Introduction 4
Summary of Previous Work 6
Lab Design and Implementation 14
Conclusion and Future Work 16
Lab Examples 18
   Lab 3: Software Setup and Introductory Assembly Programs 18
   Lab 7: Parallel I/O and Keyboard Scanning 27
   Lab 8: Maskable Interrupts 36
   Lab 9: Input Capture with the F2013 44
   Lab 10: Analog to Digital Conversion 49
Introductory Material 66
   Intro to the MSP430F2013 Microcontroller 66
   Assembly Programming 74
   Overview of the F2013’s I/O Ports 90
   Interrupts 93
   Timer_A2 MSP430 100
   Fundamental Concepts of a Sigma-Delta A/D Converter 113
   The F2013’s A/D Converter 125

List of Figures and Tables

Figure 1 – The EZ430-F2013 Development Tool 6
Figure 2 – The Budget for the Project 7
Figure 3 – An IAR Screenshot 10
Figure 4 – The PCB Diagram for the EZ430 12
Acknowledgements

A special thanks goes out to the two members of my senior design team:

- Bill Eads                        Senior Design Advisor
- Adrian Valenzuela               Texas Instruments Advisor

Bill Eads has consistently shown his dedication to the undergraduate program by volunteering his time and efforts far outside of his responsibilities. On two previous occasions, he has agreed to sponsor my Honors projects as extensions of the course material and he is held in high regard among students. This project is in great part due to his dedication and guidance.

Adrian Valenzuela has also been an integral part in this project’s completion. Without his support, it would have been much more difficult to attain the tools and resources necessary to complete the project. Whether the issue was architecture questions, implementation troubles, or contact information, Adrian has made himself readily available the entire semester, providing quick replies and continual encouragement. The resulting material is very much a product of his help and contributions. I look forward to working with him at Texas Instruments.
Introduction

The majority of all electrical systems today employ some sort of microcontroller technology. A microcontroller’s low-cost, flexible, and self-sufficient design allows it to command almost any modern task that employs some form of embedded systems. From cars to refrigerators to handheld devices, microcontrollers play a dominant role in the development of many different products for many different companies. CSU does well by making its Introduction to Microcontrollers course (EE251) mandatory for all ECE undergraduate students. And now thanks to the professor for the course, Bill Eads, CSU does even better for its efforts to keep the tools it teaches in its microcontroller course both recent and relevant.

This project presents a set of five labs, out of the necessary eight that would need to be written, that would allow the Introduction to Microcontroller course to move towards a more recent technology, with better software, that is more portable, more modular, and would save the students money for use in the classroom. The tool is called the Texas Instruments EZ430. The EZ430 is a USB stick implementation of a full development kit – power supply, I/O access, additional debugging hardware and extra peripherals – for TI’s Ultra-Low Power microcontroller, the MSP430. The suggested transition would be from the current tool for the class, the Motorola 6800HCS12 microcontroller.

But before the decision is made to move towards a new tool for which there exists little classroom material, away from a well established and proven classroom technology like the 6800HCS12, the benefits must be closely analyzed. The resulting material for the project must also be equally as, if not more robust than the material for the currently
employed technology. This paper presents those benefits, the lab materials that have been re-written to fit the EZ430 and an explanation of the project’s development from its beginning at the end of last semester to its end, scheduled for next semester and to be continued by a group of students as their senior design project. The labs include all necessary introductory material, the labs themselves, and their solutions (with the exception of Lab 10).
Summary of Previous Work

A. The Project Launch

This project began at the end of last semester when I applied, interviewed, and accepted a job with Texas Instruments in Dallas, TX as an Applications Engineer for their Ultra-Low Power MSP430 Microcontroller. Looking to find a way to mix school with some experience focused around my future job, I arranged to meet with my microcontroller professor Bill Eads. He was looking to update the tools for the EE251 course and had heard about the EZ430. His only concern was that there existed no material for teaching the course. Hence, I suggested translating his set of labs to the EZ430. With a set of labs, he could teach the students from a more general microcontroller text book and use the EZ430 as his lab tool.

![Figure 1 – The EZ430-F2013 Development Tool](image)

After the decision to move forward was made, I switched from my previous senior design group over to the microcontroller project and began writing the labs. Immediately, the implications of no existing MSP430 texts became clear, as I had no reference materials from which the students or I could learn. I contacted my associates at Texas Instruments and quickly received all the material I would need to complete the project. This included:
- Two EZ430 packages
- A full copy of the Advanced Technical Conference CD held by Texas Instruments for the MSP430 including training labs, presentations, and professional material
- Three extra MSP-EZ430D boards
- A more advanced MSP430 development kit
- A multimeter for current measurements
- Wire kit for measurement connections

With such strong support from Texas Instruments, the remaining budget was only:

<table>
<thead>
<tr>
<th>Necessary Hardware</th>
<th>Quantity</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>EZ430 Device</td>
<td>2</td>
<td>$20.00</td>
</tr>
<tr>
<td>14 Pin Header</td>
<td>2</td>
<td>$0.30</td>
</tr>
<tr>
<td>14 Pin Connector</td>
<td>2</td>
<td>$0.25</td>
</tr>
<tr>
<td>14 Wire 2-Sided DIP Connector</td>
<td>1</td>
<td>$7.00</td>
</tr>
<tr>
<td>Roll Of Heat Shrink Tape</td>
<td>1</td>
<td>$10.00</td>
</tr>
<tr>
<td><strong>Hardware Subtotal (Including Provided Materials)</strong></td>
<td></td>
<td><strong>$58.10</strong></td>
</tr>
<tr>
<td><strong>Hardware Subtotal (Not Including Provided Materials)</strong></td>
<td></td>
<td><strong>$17.00</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
<th>Quantity</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Software Subtotal</strong></td>
<td></td>
<td><strong>$0.00</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Presentation Materials For Engineering Days</th>
<th>Quantity</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poster Board</td>
<td>1</td>
<td>$8.00</td>
</tr>
<tr>
<td>Sticky Letters</td>
<td>1</td>
<td>$10.00</td>
</tr>
<tr>
<td>Carboard Paper</td>
<td>1</td>
<td>$4.00</td>
</tr>
<tr>
<td>Glue Stick</td>
<td>1</td>
<td>$3.00</td>
</tr>
<tr>
<td><strong>Presentation Materials Subtotal</strong></td>
<td></td>
<td><strong>$25.00</strong></td>
</tr>
</tbody>
</table>

**Total (Not Including Provided Materials)** $42.00

Figure 2 – The Budget for the Project
Hence, it was reasonable for me to cover the costs out-of-pocket and I did not plan to apply for any kind of grant money or search for further donations.

B. Benefits to the Students

The benefits that the EZ430 will bring to the students of CSU are extensive.

1) An update to new technology

The central processing unit for Motorola 6800 architectures is designed as a Complex Instruction Set Computer (CISC) architecture CPU. This architecture was released in the early 70’s when memory sizes were extremely small. This meant that code sizes had to be kept as small as possible so that programs could actually be stored and executed in the controller. Therefore, the set of possible instructions was extensive and hardware was designed so that each instruction would execute as many low-level operations as possible. This made code readable and concise, fitting the limitations of its time.

The MSP430 (the controller for the EZ430), employs a Reduced Instruction Set Computer architecture (RISC) CPU. The RISC architecture was realized by the early 80’s after memory sizes had grown for a good decade – following Moore’s law, 10 years $= 2^{10} = 1024 \times$ the size and capability – and computational speed far surpassed memory access speeds. RISC architectures stress a smaller set of low-level, single cycle instructions (instructions that typically execute within one clock cycle) and pipelining alongside an extended register set for reduced memory access. This combination results in extremely fast, flexible, and generally low-power implementations that were not attainable with a CISC processor.
The MSP430 is also an orthogonal design, which is extremely nice. It has 16 registers, all of which can be accessed by the complete instruction set, 27 basic instructions, and 7 addressing modes. An orthogonal design means that any instruction can access any register in any addressing mode. As far as code implementations are concerned, this is a great improvement over CISC designs that have very specific, high level hardware, allowing certain instructions to be used with only certain addressing modes or with specific, designated registers. CISC processors tend to leave a programmer with an instruction manual that must be continually referenced to make sure he/she is using the correct registers/modes, taking time and resulting in many headaches.

It would be unfair to call CISC processors obsolete, or even a dying technology. In fact, some of the most popular processors including the Pentium and Motorola 68000 architectures use a CISC processor. Compiling code to a set of CISC instructions is much quicker than compiling to a RISC instruction set. Digital Signal Processing controller implementations also lend themselves a bit more towards CISC architectures due to their specific and commonly complex processing needs.

As far as the students of the ECE department are concerned, however, switching to a RISC architecture implementation will result in a more general, flexible, and updated education.
2) An improvement of the software tools

The software for the EZ430, IAR Embedded Workbench, comes free with the purchase of the tool. Though it is a “kickstart” version, meaning there exists a 4 kB limit of code, the standard microcontroller with which it comes is limited to 2 kB of memory. IAR carries both a C compiler and an assembler. The code size limitations would be an issue if the microcontroller class was taught for development in the C programming language, but the fact that the EZ430 is used for an introductory course in assembly language makes the limitations non-restrictive.

The IAR Embedded Workbench really sets itself apart from the software for the Motorola HCS12 through its easy-to-use, advanced Graphical User Interface. Specifically, it is the debugging tools that present a great improvement over the terminal debugging interface for the HCS12.

![Figure 3 – An IAR Screenshot](image)
All registers and memory can be easily accessed and are presented in an itemized fashion. If the code causes any bits to change within memory, the bits that have changed are highlighted for you, and all the traditional stepping and breakpoints are still available.

Overall, the facts that the IAR Embedded Workbench is so easy to use and that it comes free with the purchase of the EZ430 incentivize a transfer to its use. If a student wanted to continue on and use an MSP430 device for, say, a senior design project, he/she would also already know the software that could be used for development. The IAR Embedded Workbench definitely presents a justification for the switch to the EZ430.

3) Portability

Though it may seem obvious, the portability of the MSP430 should be mentioned in the benefits its use would bring to the students. Its size being that of a memory stick, the EZ430 is very easily transported. Even with the extra hardware, like the small bus that connects the header to any external circuits or a male to female USB connector, the EZ430 takes less space than the Motorola development kit (with a power converter and the RS-232 cable).

Most importantly, the fact that the EZ430 has a USB interface makes it one of the most flexible tools on the market. It can plug into any modern computer through its USB connection. No RS-232 connections, parallel port connections, or other
methods of interfacing are necessary, so the students can truly work on their labs on any computer that can install the IAR Embedded Workbench.

4) Modular design

The following PCB diagram shows the arrangement of the hardware on the EZ430. Notice how the actual MSP430 attaches to the debugging and USB interfacing hardware through a 4-pin, JTAG port.

![PCB Diagram for the EZ430](image)

Figure 4 – The PCB Diagram for the EZ430

This allows for two benefits. Most importantly, the students could use another microcontroller that is not the MSP430F2013. Both the F2013 and F2012 controller boards are available in packs of three that retail at $10. Their slight differences in peripheral architecture would allow a student to explore different implementations of like concepts in application. And since three of the boards would cost only $10, a student that managed to overheat his chip could have a replacement within days, and for nearly free. The Axiom boards that EE251 currently uses cost about $50 and take two weeks to fix. Once again, the EZ430 outperforms its competition.
5) The price is better

In an educational setting, as in most settings, cost is of big concern to a university and its relatively poor student body. The entire EZ430 package, including the tool and the software, retails at $20. Plus the extra hardware, the entire Hardware package could total about $25 - $30. Currently, the school spends $80 on the Motorola development kits and the students pay $60 per kit. The reduction in cost could seriously decrease that number, greatly benefiting the students.

The EZ430 holds many undeniable improvements for the students. It is therefore recommendable to create this set of labs so that EE251 can switch over as soon as possible.
Lab Design and Implementation

A total of five labs were written as a result of this semester’s work. Some of the material is still being polished, but the project is about 95% finished. The introductory material will also be elaborated over the summer so that the material be presented to the students in non-outline format. The following is a short explanation of each lab and the introductory materials that were written. The labs and the introductory material come attached at the end of the document.

Lab 3 – An Introduction to the EZ430

This lab introduces the students to the EZ430’s hardware architecture and software use.

Assignment:
The students must run and debug a provided code sample.

Introductory Material:
- Intro to the F2013’s Architecture – presents the basics of what the students need to know about how the MSP430F2013 controller operates
- Intro to Assembly – presents the basic of assembly coding with the instruction set for the MSP430

Lab 7 – Parallel I/O and Keyboard Scanning

This lab introduces the students to Digital I/O with the MSP430 and some basic polling methods.

Assignment:
The students must design the circuit to interface a keyboard. Using polling, their program must show the hexadecimal representation of the key that is pressed in a variable on-screen.

Introductory Material:
- Overview of the F2013’s I/O Ports – presents the basics of interfacing the digital I/O ports for the F2013 microcontroller, including registers and example code
Lab 8 – Maskable Interrupts

This lab introduces students to the concepts of interrupts and interrupt handling, as well as the basic features of the Timer module on the F2013.

Assignment:
The students must make a working stopwatch with START, STOP, and RESET buttons. All inputs from the buttons and from a DIP switch that changes the speed of the timer should be handled with interrupts.

Introductory Material:
- **Interrupts** – presents the basics of interrupts, interrupt handling, and the difference between maskable and non-maskable interrupts
- **Timer A2 on the MSP430** – presents the timer module and its registers, including some example code

Lab 9 – Input Capture with the MSP430F2013

This lab introduces students to the more advanced features of the Timer_A2 module for the F2013 microcontroller, like input capture.

Assignment:
The students must use the input capture feature of the F2013’s I/O ports to gather information like duty cycle, period, and frequency of an input waveform.

Introductory Material:
- Same as for Lab 8

Lab 10 – A/D Conversion

This lab introduces students to analog to digital conversion using the Sigma-Delta converter on the F2013.

Assignment:
The students must use the A/D converter to display an input signal ranging from -600mV to 600mV on their program screen.

Introductory Material:
- **Fundamental Concepts of the Sigma Delta A/D Conversion Method** – presents the “how” to the sigma-delta method of analog to digital conversion
- **The F2013’s A/D Converter** – presents how to operate the Sigma-Delta converter on the F2013 specifically, including registers and some example code
Conclusions and Future Work

A. Future Work

The termination of this project will be the end of this next semester, after the next EE251 course is taught. The task of finishing the project will be left to a group of three senior design students. Their task will be fourfold:

1) Finish writing the set of labs
2) Write PowerPoint presentations for the introductory material to the labs
3) Perform each lab, checking for clarity and concept as well as making sure that the students have everything they need to accomplish the task
4) TA a set of students that have volunteered to do the TI labs instead of the Motorola labs through the class

I will also continue to work on the labs. The introductory material needs to be written out so that is not presented in outline format, and the solution to lab 10 still needs to be made. The end result will be a robust, tested, and proven set of labs for an introductory course in microcontrollers. By the fall of 2008, Professor Eads can use the PowerPoint slides and the lab material as a great starting point for his new classroom tool, the EZ430.

B. Conclusion

Bill Eads, along with the Colorado State ECE department, has made a statement by supporting this senior design project. He and the college have shown their dedication to the continual improvement of ECE’s undergraduate education and their concern for the students’ needs, both in the case of EE251 students and in mine. They have been flexible with the project, allowing a student to essentially create his own experience and supportive, providing equipment, suggestions and time. In the process, we have made
three great steps forward. The learning tools for EE251 have been improved as explained by the benefits portion of this text. I have gained knowledge and experience that empowers me with a great momentum going into my new job at TI. And we have forged yet another strong relationship with a leading technological company, Texas Instruments, which can be mined for years to come. For a semester’s work the project has truly been a success.

The attached labs include all information necessary for the understanding of the EZ430, its architecture, and its implementations. Please read through them and gain some insight as to the work that has gone into them and the quality of the material. It is possible the introductory outlines will be developed into a book within two to three years, further advancing the MSP430 and its influence in the education of undergraduates nationwide.

C. Bibliography


3

Software Setup & Introductory Assembly Programs

3.1 – Objectives

This introductory lab will walk you through the entire process of typing up a program, assembling it, downloading it to the EZ430 (the development kit for the TI-MSP430F2013) and executing the program. When you complete this lab, you should be able to:

- Understand the IAR Embedded Workbench IDE (the software used to write assembly/C language for Texas Instrument’s MSP430.
- Assemble (build) a program using the IAR Embedded Workbench.
- Download a program onto your EZ430.
- Run a program, examine and change memory registers.
- Use trace and breakpoints to debug the program.

Because you will perform all of these procedures in every lab you do after this, a complete understanding of the material in this lab is necessary.

3.2 – Reading Material

I. Introduction to the MSP430-F2013 controller
II. Introduction to assembly programming
III. Assembly Operators
IV. Instruction Reference

3.3 – Board Startup

3.3.1. EZ430 USB Stick

Please refer to the EZ430 User’s Guide for all introductory information.

3.3.2 Hardware Install
3.3.3 Software Install

Please refer to the EZ430 User’s Guide for hardware installation.

3.4 – Introduction to the IAR Embedded Workbench IDE

Resource used for this section of lab:
- Embedded Workbench User’s Guide

Invoke the IAR Embedded Workbench in one of the following two ways:

1. Left click your mouse on the START menu on the bottom left of your screen. Then follow the path, Programs → IAR Systems → IAR Embedded Workbench Kickstart for MSP430 V3 → IAR Embedded Workbench.

2. Open Windows explorer and locate IAR Embedded Workbench.exe in the folder C:\Program Files\IAR Systems\Embedded Workbench 4.0\common\bin\ iarIdePm.exe

To create the shortcut on your desktop, right click the mouse on iarIdePm.exe and then left click on create shortcut. Then click on this shortcut to invoke the IAR Embedded Workbench application.

This part of the lab is considered the tutorial that will be repeated for your TA:

**Step 1 – Creating a Project**

The first thing you will see is the *Embedded Workbench Startup* screen. This screen will be a good shortcut to open existing workspaces and create new projects in the future, but for now, click the Cancel button. Note that a *workspace* can hold one or more *projects*, each which can hold groups of *source files* (the files that contain your assembly code).

Click on *File -> New -> Workspace*. This opens a new workspace in which to place your projects. You must always open a workspace before you can create a new project. To create a new project, click *Project -> Create New Project...*. The *Create New Project* box will appear. IAR lets you choose whether you want to create an assembly (*asm*), *C*, or *C++* project. Expand the *asm* tab and click *OK* once you have selected the *asm* option.
IAR will ask you to specify where you would like to save your .ewp project file. It is
recommended you create a folder for your workspaces such as
~/MSP430/Workspaces/ in which you can store all your future labs. Create a folder
called tutorial_01, name your project project_01, and click Save. At this point, you
should also download the add_primes.s43 file provided on the class webpage into the
same folder.

Look at the left of your screen. You will see the *workspace window* where your
projects and source files are organized.

On the right is the *project window*. The project window shows the assembly file
template discussed in *Introduction to assembly programming* from the pre-lab
material. You will actually use another file for this tutorial but, before anything else,
click on *File -> Save Workspace* and save your workspace in the same folder as the
project.
Now add the file we are going to examine in the tutorial. In your workspace window, right click on the project name and select Add -> Add Files... At the bottom of the page, change the Files of type tab to the Assembler Files option. If you downloaded the file from the class webpage, go ahead and select the add_primes.s43 file and click Open. You should see the file added to your workspace window. Right click on the asm.s43 template and remove it from the project then take a look at the add_primes.s43 code. This assembly program has stored the first six prime numbers in an array in memory and adds them up into register six. See if you understand what is going on by referencing the pre-lab material.

**Step 2 – Setting the Project Options**

Before we can compile our code into an executable object file, we need to set the project options. Select the add_primes.s43 file in your workspace window and click Project -> Options or simply right click on the project and select Options... This will bring you to the Options for node “<project_name>” screen.

![Options for node “Tutorial_01”](image)

**Figure 3.3 – The project options screen**

- In the General Options menu:
  - Set the Device to the MSP430F2013
- In the Debugger menu:
  - Set the Driver to FET Debugger. This makes sure that once compiled into an object file, the program will be loaded onto your physical microcontroller, not the IAR simulator.
  - If you would ever like to work on your code but you do not have your EZ430, you can set the Driver option to Simulator and the IAR Embedded Workbench will simulate your microcontroller.
- Under the actual *FET Debugger* menu:
  - Change the *Connection* from *LPT -> LPT1* to *TI USB FET*. This tells IAR to look for your USB stick and not a port on your computer.
  - Change the *Target VCC* option at the bottom of the screen to 3.0
- Click OK to finalize your settings. You are now ready to build and compile your project.

**Step 3 – Running the Program**

Before we compile the project, make sure to save your project to preserve its settings. Select the add_primes.s43 file in the workspace window and click *Project -> Compile*. Alternatively, you can right click on the source file and select *Compile*. When the workbench is finished compiling your source code, you will see a new window at the bottom of the screen called your *Build Messages Window*. It should say that there are 0 errors and 0 warnings.

![Figure 3.4 – The build message window](image)

To see what happens when your code contains an error, erase one of the semicolons preceding a comment in the add_primes.s43 code and re-compile. The build message window displays:

- Any errors/warnings in your code
- The assembler’s best attempts of an explanation
- The name of the files in which the errors/warnings occurred
- The line number where the errors/warnings occurred

Double click the line that says “Error[0]: Invalid syntax.” In the file, your cursor will be placed close to the location of the error. Fix the error, and then re-compile.

Now make sure that your EZ430 is plugged in and click *Project -> Debug*. This will run your code on the EZ430 and put you in debugging mode.
Step 4 – Debugging A Project

A new window has been added to the screen in debugging mode. It is called the Disassembly Window. After the assembler compiles your code into machine code, a disassembler program does the opposite and creates the contents of the disassembly window. You can apply the debugging commands to either the coded assembly file or the disassembly window. Generally, this tool is more useful when the developer has written his/her application using the C programming language and wants to see how the machine interpreted his/her code in assembly.

The workbench allows users to move and dock windows as they wish. To see this, click View -> Memory and View -> Registers. Click on the window labels to drag and dock them in the same tab group as the Debug Log and Build windows. These windows are updated as the program executes, giving the user extremely useful insight to the inner workings of his/her program.

NOTE: If the contents of the window selected in each tab group ever change, the changed contents will be highlighted red to show the change.

The Memory Window

The memory window allows the user to navigate the entire memory map of the MSP430. Since we used assembler directives to load the array of prime numbers starting at memory location 0200h, type 0x0200 into the Go to box. You will see the array stored in consecutive words in memory.

![Figure 3.5 – The memory window](image)

Instead of Memory, IAR also gives you the option to looking at:

- SFR: The special function and peripheral registers
- RAM: Memory
- INFO: Information FLASH memory (ROM)
- FLASH: Main FLASH memory (ROM)
NOTE 01: You can use the information and main memories just the same. The only difference is how the memory is physically segmented and the address.

NOTE 02: If you look at the FLASH memory, notice that this is where the main program is stored. The numbers you see are the instructions of the program that have been stored for program execution.

*The Register Window*

The register window shows the user any of the important registers whether they be peripheral control registers, special function registers, or just the basic CPU registers. For the most part, you will only be interested in the CPU Registers, but know that the other registers are available for viewing. Also note that the Status Register (SR) containing the status bits can be expanded to see each individually labeled bit.

While stepping through the program, periodically check the Memory and Register windows to see how they change according to your code.

*Inspecting Source Statements*

Run your mouse over these buttons in the top left corner of your debugging session and read their function in the bottom left corner of the IAR workbench frame.

![Debugging Buttons](image)

Figure 3.6 – The debugging buttons

- **Program Reset**: Resets the execution of your program. Takes you back to the instruction address located in the reset interrupt vector.

- **Step Over**: Steps over the current step point.

- **Step Into**: When you debug a program with subroutines in assembly, you can choose to “Step Over” the subroutines, meaning that the subroutine is executed but you don’t step through it in the debugger, or you can choose to “Step Into” the subroutine. Stepping into a subroutine will allow the user to see the individual op-codes executed.

- **Step Out Of**: If you have stepped into a subroutine, steps out of the subroutine and to the next instruction after the subroutine call.

- **Step to the Next**: Steps to the next statement, regardless of the circumstances.
Statement

Execute to the Current Cursor Position

Executes the program to the line at which the cursor is currently located.

Run

Runs the program straight through or until a breakpoint.

Stop Debugging

Stops the debugging session returns the user to programming mode.

At the top right-hand side of the page, note this button as well:

Make and Reset

Press when code has been modified so that it needs to be re-compiled and the debugger needs to be reset

Breakpoints

Sometimes it is beneficial, if you want to know how your program is executing at a certain point in its execution, to use breakpoints. To place a breakpoint in the program, double click on the margin next to the line at which you wish to set a breakpoint (or right click on the line and select toggle breakpoint (conditional)). For example, the initial operators in every assembly code project that initialize the controller after a Power Up Clear can often be assumed to work properly. If you would like to run the program starting immediately after this section of code, add a breakpoint at the line labeled main. Since we have two clear operators in the beginning of our code that we know will work, place a breakpoint at the first mov.w instruction line of the main program. Press the Program Reset button to restart program execution and then press the Run to execute until the breakpoint. From here on, play with the other buttons to gain some intuition as to their function.

NOTE: Debugging can also be done in the disassembly window. Try it out and see if you can follow what all the numbers mean in regards to the CPU registers and memory.

For more information on debugging, reference the EW UserGuide. An in-depth understanding of the debugging module is a truly powerful tool for future programming.

After completing this tutorial, you should feel comfortable working with the IAR Embedded Workbench. If you do not, you may want to go back and do it again. Also, make sure you have read and understand the material from lab section 3.2. It will be critical to the execution of future labs.
3.5 – Procedure

1. Set up the EZ430, IAR Embedded Workbench, and all necessary hardware.
2. Complete the tutorial.

Call the lab TA and repeat the tutorial in their presence, without reading the directions. When you can do this without looking at the instructions, the TA will check you off. Review any part of this lab you are not sure about.

3.6 – Questions

1. Describe the register set for the MSP430F2013. What are the Special Function Registers (SFRs) and their functions? Where are they located?

2. How are single stepping and breakpoints used to debug a program? Why might you use breakpoints rather than single stepping?

3. In the Disassembler window, what do the numbers on the far left mean? What does this have to do with the contents in R0 during program execution?

4. Using the debugger, the Disassembler window and the assembly instructions shown, explain how the loop in the add_primes.s43 file is executed. Include the addressing modes used and why they might have been used for each instruction.

3.7. – Lab Report

The lab report is due at the beginning of the next lab. For the lab write up, include the following:

1. A copy of your working .c, .s43, and .lst files for the program.
2. A brief discussion of the objectives of the lab and the procedures performed in the lab.
3. Answer to any questions in the discussion, procedure or question sections of the lab.
7

Parallel I/O and Keyboard Scanning

7.1 Objectives

Microprocessors have the capability to monitor the outside world using input ports. They can also control it using output ports. The MSP430F2013 has one accessible I/O port with 8 pins, to be discussed in this lab.

This lab will introduce you to the I/O capabilities of the MSP430 using a 16-button keyboard, provided in the lab. By performing this lab, you will learn:

- How to perform simple parallel input and output transfers using P0
- How to use polling to do data transfers
- How to interface the keypad with the MSP430
- How to scan the keypad and detect that a key has been pressed
- How to de-bounce the keypad using software.

7.2 Related Material

- Overview of the MSP430F2013 I/O Ports

7.3 Parallel I/O using the MSP430F2013

The MSP430 has two I/O ports but for your labs, you will only have access to P1. The microcontroller uses its address, data, and control buses to control I/O hardware as if it were memory. Controlling I/O this way is called memory mapped I/O. Each port has an address. To output data through that port, configure the port to output mode through the P1DIR register. A toggle of the P1OUT bits will then signify a toggle on the respective output pin.

The pin connections for the P1 I/O port are labeled “Fully Accessible MSP430 Pins”. For us to access these 14 pins we’ve had to solder on a header. The header is then accessed with a connector wired to a DIP-compatible end for breadboard use.
Figure 7.1 – MSP430F2013 board showing the I/O Pins and Respective Mapping

From the diagrams, you should be able to tell that P1.0 maps to P2 on the PCB schematic, P1.1 maps to P3, P1.2 maps to P4 and so on. These port pins are all individually programmable so that some of the bits can be used for parallel input and some for parallel output. This is done by storing a value in an 8-bit register called the data direction register or P1DIR (at memory location 022h). When the bit, P1DIRx, contains the value 0, the respective pin is set to input mode. If the bit is set to 1, then the pin is set to output mode. The port select register, P1SEL (address 026h), must also be set to 0x00 to set the pins to I/O mode. The P1IN (address 020h) register maintains the status of all pins that have been configured as input pins. If a pin is instead configured for output, the P1OUT (address 021h) register holds the values being output.

The following code shows the equate directives you would use if you were going to use port pins P1.0 and P1.1 for output and P1.2 – P1.7 for input.
The parallel I/O capabilities of the MSP430 allow it to control the outside world by connecting it to external hardware. Common external devices include:

- Push button switches
- DIP switches
- Light Emitting Diodes (LEDs)
- Keypads
- Seven segment displays

In this lab, we consider all hardware except seven segment displays. Seven segment displays will be considered in the last lab when we design a digital voltmeter.

Figure 7.2 shows one way to connect DIP switches to the EZ430. These DIP switches provide 4 individual switches with 4 resistors. When a switch is in the open position, logic ‘0’ is provided to the corresponding input port pin by a pull-down resistor. When a switch is in the closed position, the corresponding input port pin is driven high and provides logic ‘1’ to the input port. The resistor limits the current flow when the switch is closed as to not damage the device. In the actual switches, the switch has a tendency to bounce. To debounce a switch using software, you need to introduce a software delay of 40 ms after the first switch contact is read. During this short delay, switch bouncing is effectively locked out.
Figure 7.2 Connection of DIP switches to input ports.

Figure 7.3 shows how to connect LEDs to the EZ430 output ports. When logic high is presented on the output port pin, the inverter converts the logic high to logic low. The difference in potential then causes current to flow from the 3.3 V source through the LED, through the current limiting resistor, R, and into the output terminal of the inverter. The LED then illuminates. However, when logic low is present on the output port pin, there is not a sufficient potential difference between the logic high of the inverter and the supply voltage to satisfy the forward voltage drop requirement of the LED; hence, the LED will not illuminate.

Figure 7.3 Connection of LEDs to output ports

Most keyboards use a number of individual switches arranged in a matrix of columns and rows. This reduces the number of I/O lines needed to interface with the keyboard. A keyboard of 64 keys could be arranged in an 8-by-8 matrix, so that only 16 I/O lines are required instead of 64. Software is used to scan the keyboard to detect a depressed switch and determine which key is pressed. In this lab, we will interface and write the software to scan a 4-by-4 keypad. Figure 7.23 in the text shows the direct connections for the 16-key keypad we will be using.

The keypad circuit is shown in Figure 7.4. The rows are connected to the outputs and the columns are connected to the inputs of the microprocessor. You will use port pins P1.0 – P1.3 for the rows and port pins P1.4 – P1.7 for the columns. The scanning program
output 1’s to the rows and tests all four-column inputs in a continuous loop. From the circuit, you can see that these inputs will all be low until a switch is closed.

You will write a subroutine for scanning this keypad circuit. The flowchart for this subroutine is shown in Figure 7.6. The subroutine will output ‘1’s to all four-row bits, and then read the column bits in a continuous loop until one of them goes high.

When one input goes high, the subroutine will scan the keypad by outputting logic 1 to the first row and ones to the other rows. Each column will then be checked, from left to right, for a low signal. If a high low signal is not detected, a 1 is output to the next row, 0’s are output to the rest of the rows, and the columns are checked again. This continues until the pressed key is found.

Anytime a switch is pressed, mechanical bouncing occurs, causing the switch to open and close rapidly several times. The scanning subroutine is so fast it can scan the whole keypad before the switch gets done bouncing, missing the closed switch. A delay in the subroutine gives the switch time to stop bouncing. You can use the delay subroutine from Lab 5 to provide the delay time of 40 ms. Have your scan program call the ‘DELAY’ subroutine after a key is pressed.

Figure 7.4: Connection of DIP switches to simulate Keypad circuit.
Figure 7.5 shows specifications for a Grayhill keypad, which you will use in the lab. It shows the relationship between the 16 keys, shown as 1-16 in the diagram, with the 8 terminals on the back, labeled E-H and J-M. Careful observation of this diagram, the key labels on our keypads (different from that shown below), and the matrix of terminal-key connections provides sufficient information to create the relationship between Grayhill’s terminal names and the port labels in Figure 7.4 above:

Terminal E = Port P1.
Terminal F = Port P1.
Terminal G = Port P1.
Terminal H = Port P1.
Terminal J = Port P1.
Terminal K = Port P1.
Terminal L = Port P1.
Terminal M = Port P1.

Terminal E = Port P1.
Terminal F = Port P1.
Terminal G = Port P1.
Terminal H = Port P1.
Terminal J = Port P1.
Terminal K = Port P1.
Terminal L = Port P1.
Terminal M = Port P1.

Figure 7.5: Grayhill Series 86 Keypad Specifications
Figure 7.6: Flow chart for keypad SCAN subroutine.
7.5 Procedure:

Before you come to the lab, write the individual programs and flow charts for the following procedures and show them to the TA.

1. Configure the P1.0 – P1.3 lines as all inputs and connect them to 4 DIP switches as shown in Figure 7.2. Configure the remaining 4 P1.4 – P1.6 lines as all outputs and connect them to 4 LEDs as shown in Figure 7.3. Have the lab TA check your circuit before you turn the power ON. It would be a good idea to build this circuit before you come to the lab. Write a program to continuously read the switches connected to lines P1.0 – P1.3 and output the status to the LED’s connected to lines P1.4 – P1.6.

2. Consider the situation where a F2013 is used to control a certain machine. First a key needs to be turned on, then two switches have to be depressed at the same time (within 0.5 seconds of each other), and, lastly, a footswitch must be depressed. These steps must be performed in this order. Write a program to simulate this system. Connect the key switch to P1.0, connect the two switches to P1.1 and P1.2, and the footswitch to P1.3. The program should continuously monitor the inputs until the sequence occurs. When the sequence is done correctly, all LED’s on port P1.4 – P1.6 should light up. Show the TA when this works.

3. Connect the Grayhill keypad to your development board as shown in Figure 7.4. Write a program to scan this keypad (as described in the previous section). Include a subroutine to wait until the key is released before another key is read, as you do not want to read the key that is depressed more than once. The flowchart for this subroutine is included in Figure 7.7. Call the subroutines SCAN and NEXTKEY with the main program. Download the program on the board and display the key pressed on the screen.

IMPORTANT!!!! – This circuit will be used for lab 8, so do not take it apart.

7.6 Questions:

1. Is the polling method to input data very efficient? Why or why not?
2. In the keypad circuit, what will happen if your de-bounce delay is too short? What will happen if it is too long?
3. What key is read if you depress switches 8 and B at the same time? What if you depress 5 and 9 at the same time?

For the lab write up, include:

1. Flowcharts and programs that you wrote before the lab.
2. A copy of your working .asm files.
3. A brief discussion of the objectives of the lab and the procedures performed in the lab.
4. Answers to any questions in the discussion, procedure, or question sections of the lab.

![Main program diagram](image1)

**Main program**

- Start
- Call SCAN
- Display key depressed
- Call NEXTKEY

![NEXTKEY subroutine diagram](image2)

**NEXTKEY subroutine**

- NEXTKEY
- Read column bits
- Is key still pressed?
  - Yes
  - No
    - Return

Figure 7.7
8

Maskable Interrupts

8.1 Objectives

In Lab 7, we used the polling technique for parallel I/O. Using interrupts can improve performance of the computer and make the software easier to organize. In this lab, you will learn:

- What is an interrupt?
- The difference between maskable and non-maskable interrupts.
- The response of the MSP430 to a maskable interrupt.
- How to write interrupt service routines (ISR) for the MSP430F2013.
- How to generate delays using the MSP430 Timer_A2 Module.

8.2 Related Material to Read

- I/O Ports (from lab 7)
- Interrupts, Resets and Operation Modes
- Timer_A2 on the MSP430F2013

8.3 Maskable and Non-Maskable Interrupts

Lab 3 briefly described the register model for the MSP430F2013. This register model contains the Status Register (SR). One of the bits in the SR, the General Interrupt Enable (GIE) bit, is associated with the maskable interrupts. This bit enables the maskable interrupts for the MSP430. When this bit is set to zero, no maskable interrupts can be enabled, including the Timer_A interrupts we will use for this lab. Please refer to Interrupts, Resets and Operation Modes from section 8.2 as provided on the class website for a good introduction to interrupts.

8.4 Interrupt Service Routines

An interrupt service routine, or ISR, is similar to a subroutine. The main program runs in an infinite loop, often times doing other useful work. When an I/O device creates an
event it sets a flag that causes the main program to turn the control over to its respective ISR. The ISR services the I/O device by clearing the flag, doing any data transfer and any other work required. An ISR is similar to a subroutine but it responds to a hardware signal whereas a subroutine is called because of user-written software instructions.

ISRs in the MSP430 are programmed by inserting the address of the user-programmed service routine into the correct location of the interrupt vector table. For example:

```assembly
;---------------------------------------------------------------------
RESEI ORG 0FFFEh        ; Reset Interrupt Vector
DC16 main
;---------------------------------------------------------------------
ORG 0F800h           ; Main Program Segment
mov.w #0280h, SP     ; Initialize stackpointer
StopWDT mov.w #WDTIFW+WDTIOLD, sWDTICIL ; Stop WDT
SetupP1 bis.b #io_mode, &P1SEL ; Sets all port pins to I/O mode
;---------------------------------------------------------------------
```

Figure 8.1 – A Common PUC-reset ISR

By writing the address labeled ‘main’ to the reset interrupt vector, the main program itself becomes the interrupt service routine for a PUC reset.

8.5 Programming the Timer module on the MSP430

In the previous labs, we used software controlled delays. By counting the number of cycles it took to execute the delay loop and multiplying by a decrementing counter, we could design rough delays for the purposes of our applications. If accurate timing is required, it is more efficient programming to use interrupts and the provided Timer module for the MSP430, called Timer_A2.

Timer_A2 is based off of its free running counter, the TAR or Timer_A Register. Depending on its mode of operation, the TAR increments or decrements every rising edge of the clock signal and holds the running count in memory location 0170h.
Figure 8.2 – The TAR register at location 0x0170.

To activate the TAR, the *Timer A Control Register* (TACTL) must be properly initialized.

![TACTL, Timer A Control Register](image)

Figure 8.4 – The Timer A Control Register at location 0x0160.

The Timer A Source Select (TASSELx) bits

The clock signal that the TAR register uses to count can be sourced from four different clocks according to the TASSELx bits.

<table>
<thead>
<tr>
<th>TASSELx</th>
<th>Clock Source Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>TACLK</td>
</tr>
<tr>
<td>01</td>
<td>ACLK</td>
</tr>
<tr>
<td>10</td>
<td>SMCLK</td>
</tr>
<tr>
<td>11</td>
<td>INCLK</td>
</tr>
</tbody>
</table>

Figure 8.3 – The Four Possible Clock Sources for TAR

We will use the *Sub-Main Clock* (SMCLK). The SMCLK, like the MCLK is sourced on a PUC Reset to run at 1.1 MHz.

The Input Divider (IDx) Bits

All clocks can be sourced directly or divided by 2, 4 and 8 using the IDx bits.

<table>
<thead>
<tr>
<th>IDx</th>
<th>Input Divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>
Assuming the TAR counter counts to 0xFFFF, with the SMCLK selected as the source and the IDx bits set to divide the clock by 1, the timer would take \((1 / 1.1\text{MHz}) \times 0xFFFF\) or .059577 seconds. If that same clock is instead divided by eight, then the TAR register would take \(8/1.1\text{ MHz} \times 0xFFFF = .476618\) seconds to roll over. Note how .48 seconds is about equal to .5 seconds. This will be useful for the completion of this lab.

The Mode Control (MCx) Bits

The four modes for the TAR are selected by setting the MCx bits:

<table>
<thead>
<tr>
<th>MCx</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Stop</td>
<td>Timer is halted.</td>
</tr>
<tr>
<td>01</td>
<td>Up</td>
<td>Timer repeatedly counts from zero to the value of TACCR0.</td>
</tr>
<tr>
<td>10</td>
<td>Continuous</td>
<td>The timer repeatedly counts from zero to 0xFFFFh.</td>
</tr>
<tr>
<td>11</td>
<td>Up/Down</td>
<td>The timer repeatedly counts from zero up to the value of TACCR0 and back down to zero.</td>
</tr>
</tbody>
</table>

NOTE: It is strongly recommended that you stop the timer before modifying its operation or accessing the TAR register to avoid errant conditions.

The TACLRL Bit

At any time during program execution, the user may set the Timer A Clear (TACLRL) bit to reset the TAR register. The TACLRL bit also resets the IDx bits.

The TAIE and TAIFG Bits

Timer A interrupts are considered maskable interrupts. Therefore, the GIE bit in the SR must be set for a flag to create an interrupt event. Additionally, the user must set the Timer A Interrupt Enable (TAIE) bit. Setting the TAIE bit allows the TAIFG flag to be set according to the mode of operation.

In continuous mode, the TAIFG bit is set when the TAR register counts from 0xFFFF to zero. Every time the TAIFG bit is set, the controller will execute its interrupt vector at location 0FF0h. When the timer module sets the TAIFG bit, it also generates a number in the Timer A Interrupt Vector (TAIV) register. The TAIV register is a clever method of handling multiple interrupts in one memory location for the timer module.

The TAIV Register
Figure 8.5 – The Timer A Interrupt Vector Register and Contents Table

The highest pending, enabled interrupt for the timer generates a number inside the TAIV register reflecting the flag that was set. This number can then be added to the program counter to automatically enter the appropriate software routine. This means that your interrupt service routine must check the TAIV register each time it is called to see whether it was the TAIFG or TACCR1 CCIFG flag that caused the interrupt, as well as provide default behavior for other, reserved values. Figure 8.6 shows a common timer ISR.

NOTE: If you notice, TACCR0 CCIFG is not in the table. TACCRx registers are capture/compare registers that will be covered in the next lab. For now, know that the TACCR0 register has its own interrupt vector so its interrupt flag will not generate any number in the TAIV. Also notice that we don’t speak of the TACCR2 CCIFG flag in the TAIV register. This is because the MSP430F2013 has only two capture/compare registers, TACCR0 and TACCR1. Other models of the MSP430 with more capability have three capture/compare registers.
This ISR adds the contents of the TAIV register to the program counter. Since \textit{reti} is a two byte instruction and the numbers in the TAIV register increment by two depending on the flag that has been set, this ISR services both our TACCR1 CCIFG flag and our TAIFG flag while maintaining predictable results for all other instances.

Any access of the TAIV register automatically resets the highest pending interrupt flag. If any interrupt requests remain, another interrupt will immediately be requested and the TAIV register will show the respective flag that was set. For code examples of how to program using the TAIV register or the Timer_A module in general, reference \textit{Timer A2 MSP430}.

8.6 Procedure

1. Using the code and circuit from Lab 07, change the code so it uses port interrupts to run a SCAN interrupt service routine instead of polling and a SCAN subroutine. Show and explain the working code to the TA.

2. Using interrupts, write the code for a timer that executes using the following specifications:

   a. P1.0 – P1.3 – display seconds on 4 LED’s; every 5 seconds, the LED’s will increment in count.
   b. P1.4 and P1.5 – display minutes on 2 LED’s; after 12 second counts (because 12 X 5 = 60 seconds), the minute LED’s should increment by one and the seconds LEDs should be reset to zero. Make P1.4 the MSB for the minute LED’s.
   c. P1.6 – START button; if the timer is stopped, pressing the START button should call an interrupt service routine that starts or resumes the timer.
d. P1.7 – STOP button; pressing the STOP button should call an interrupt service routine that stops the timer.
e. P2.6 and P2.7 (XIN and XOUT) – inputs from a DIP switch that control the timer divisor’s IDx bits; dividing by 2 will make your timer count twice as slow; dividing by 4 will make your timer count four times as slow; and dividing by 8 will make your timer count eight times as slow.

<table>
<thead>
<tr>
<th>IDx bits</th>
<th>Timer division</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

For the lab, initialize the timer to divide the SMCLK by 8 in Continuous Mode. The TA will then ask you to change the clock division during your demo.
f. RST/NMI – RESET button for our timer. If at any time the RESET button is pressed, no matter what state our timer is in, it will reset our program and the user must press START to begin the timer from zero.

Recommended Steps:
- Find out how many counts of the timer it takes to make a 1 second delay. Remember, the period is the number of counts in the TACCR0 register + 1.
- Write the ISR for the TAIV register that services a TAIFG overflow. Every 5 times the ISR is called, it should increment a counter that is output to P1.0-3.
- Anytime that P1.0-3 shows the number 12, increment the minutes counter and output to P1.4 and P1.5.
- Once this is working, look at the START and the STOP buttons. They will both be implemented using Port 1’s maskable interrupt. The ISR will have to check the P1IFG register to see which flag has been set and react accordingly.
- Look at the RESET button. Though you can use the RESET mode for the pin, use the NMI mode so you can learn the difference between the non-maskable and the maskable interrupts. Program the ISR to completely reset the microcontroller and restart the timing program.
- Finally, program the DIP switch to divide the clock every time that it interrupt fires. This is a little tricky.
  - When the DIP switch is in its ‘00’ state, the P1.6 and P1.7 interrupts should be set to the rising edge.
  - When the DIP switch is in its ‘01’ state, the P1.6 pin interrupt should be set to rising edge, but the P1.7 pin interrupt should be set to the falling edge!
- This means we have to read port P2, mask it so we only see what’s in P2.6 and P2.7, and then check the status of P2.6 and P2.7 registers to see how we will:
  1. Divide the clock being sourced by Timer_A2
  2. Set the interrupt edge for pins P1.6 and P1.7

8.7 Questions

1. Use your timer program to state some of the differences between maskable and non-maskable interrupts? How are they handled differently in execution?

2. How would you program the interrupts differently if instead of using the TAIIFG flag, you chose to use the TACCR0 CCIFG flag? Give specific addresses and details. What is different regarding when this flag is set?

3. Write the TAIV ISR to service the TACCR1 CCIFG and TAIFG flags. Explain how the program executes the ISR and how you chose to write it. If the CCIFG is serviced, write the program to place 0xFF in R2. If the TAIFG flag is serviced, write the program to place 0x11 in R3.

8.8 Lab Report

For the lab write-up, include

1. Flowcharts and programs that you wrote before the lab.
2. A copy of your working .asm files.
3. A brief discussion of the objectives of the lab and the procedures performed in the lab.
4. Answers to any questions in the discussion, procedure, or question sections of the lab.
9

Input Capture with the MSP430F2013

9.1 Objectives:

The 68HC12 is equipped with a Timer Module (Timer_A2). The timer module has clock-generation circuitry to generate the internal and external clock signals used by the CPU and the on-chip peripherals, and contains two dual-function input capture or output compare channels. This lab will expand on the timer interrupt concepts from the previous lab and introduce more advanced applications of the timer module. While doing this lab, you will learn:

- How to program the input capture features of the TIM system.
- How to program the output compare features of the TIM system.

9.2 Related material to read:

1. Timer_A3 on the F2013 (from Lab 8)

9.3 The MSP430 Capture/Compare Blocks:

The capture/compare blocks of the MSP430 perform two main functions: Input Capture and Output Signal Generation. The input capture feature measures the characteristics of a periodic input signal such as period, duty cycle, and frequency of the signal. The output compare feature allows the generation of an output signal to user specifications, generally Pulse Width Modulated signals. Using these features, the MSP430 can also perform tasks such as pulse accumulation, where the user counts pulses (external events) on a line connected to the ports.

The MSP430 is equipped with two capture/compare registers, labeled TACCR0 and TACCR1. Each of these channels is software configurable for either input capture or output compare operation. Specific pins are used as input capture and output compare pins. For this lab, we will only use the input capture feature of the MSP430 meaning that only P1.1, P1.2, or P1.6 can be used for the Timer I/O operation.
9.4 Input capture:

The capture/compare registers are initialized using their Timer A Capture/Compare Control Register (TACCTLx) in memory locations 0x0162 (TACCR0) and 0x0164 (TACCTL1). The TACCTLx register has many control bits that are important to the operation of the input capture or output compare functions. To enable capture mode, the CAP bit must be set to 1.

![Figure 9.1 – The TACCTLx register](image)

The Capture or Compare Input Select (CCISx) Bits (13 – 12)

The CCISx bits select the TACCRx input signal.

<table>
<thead>
<tr>
<th>CCISx Bits 13 – 12</th>
<th>Selected TACCRx input signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>CClxA</td>
</tr>
<tr>
<td>01</td>
<td>CClxB</td>
</tr>
<tr>
<td>10</td>
<td>GND</td>
</tr>
<tr>
<td>11</td>
<td>Vcc</td>
</tr>
</tbody>
</table>

![Figure 9.2 – The CCISx bits](image)

The GND and Vcc pins are internal and are used for software-initiated captures. You will not use software-initiated captures in this course. These signals are tied to actual physical pins which you access on your EZ430 according to its data sheet. Your F2013 controller gives you access to three pins: the CCl0A, CCl1A, and CCl1B.

<table>
<thead>
<tr>
<th>CClx bit</th>
<th>I/O Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCl0A</td>
<td>P1.1</td>
</tr>
<tr>
<td>CCl1A</td>
<td>P1.2</td>
</tr>
<tr>
<td>CCl1B</td>
<td>P1.6</td>
</tr>
</tbody>
</table>

![Figure 9.3 – The CCISx bits](image)
The Capture Mode (CMx) Bits

Once you know which pin you are going to use as the I/O pin for your input capture or output compare function, you must also select which edge you want to initiate the capture using the Capture Mode (CMx) bits.

<table>
<thead>
<tr>
<th>CMx bits 15 - 14</th>
<th>Capture Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No capture</td>
</tr>
<tr>
<td>01</td>
<td>Capture on rising edge</td>
</tr>
<tr>
<td>10</td>
<td>Capture on falling edge</td>
</tr>
<tr>
<td>11</td>
<td>Capture on both rising and falling edges</td>
</tr>
</tbody>
</table>

Figure 9.4 – The Capture Mode (CMx) bits

Now, assume that the TACCTL1 register is set to capture mode, the CCIIA signal is selected as the input signal (meaning the input signal is tied to P1.2), and the capture mode is selected to capture a signal on the rising edge. When the capture occurs, two important events happen:

1. The timer value (the value inside the TAR register) is copied into the TACCRx register (in this case, TACCR1).
2. The CCIFG interrupt flag is set.

The CCIFG flag being set will fire an interrupt and call upon the TAIV register’s ISR so the user can process the necessary information. At any time, the input signal level may be read through the Capture Compre Input (CCI) bit in the TACCTLx register.

The Synchronize Capture Source (SCS) Bit

In many cases, the input signal can be asynchronous to the timer clock, allowing for race conditions to occur. Setting the Synchronize Capture Source (SCS) bit will synchronize the input capture with the next timer clock. For this reason setting the SCS bit is usually recommended.

Figure 9.5 – A capture event with the SCS bit set
If a second capture is performed before the value from the first capture event (stored in the TACCRx register) is read, the controller provides an overflow bit to inform the user, shown in the TACCTLx register as the *Capture Overflow* (COV) bit. The TACCR0 and TACCR1 registers are stored in memory locations 0x0172 and 0x0174, respectively. The COV bit must be reset by software.

![Figure 9.5 – The TACCRx register](image)

Note that value from the first capture event is no longer available if a second capture is performed. A flow chart explaining the correct steps to handle a capture is provided below.

![Figure 9.6 – A capture flow chart](image)

### 9.5 Procedure:

1. Write a program that uses the input capture feature of the TIM to find the duty cycle, period and frequency of a square wave from the function generator. Connect the function generator signal to one of the pins of Port T and configure the corresponding
channel as an input capture channel. Display the measured characteristics of the signal on the screen. Show the working program to the TA.

2. Modify the procedure from question 1 so that the period of any periodic square wave can be measured, i.e., measured period can be more than the rollover time of the free-running counter. Show the working program to the TA.

3. Write a program that uses the input capture feature of the Timer_A2 to count the number of rising or falling edges in the square wave from the function generator in 10 seconds. Display the counter value on the screen at the end of the program. Show the working program to the TA.

9.6 Questions:

1. Give at least 2 applications each of the input capture.

2. Is it possible to generate a square wave of period 10 seconds using the output compare feature of the TIM? If yes, explain how and if not, explain why not. You do not need to write a program or flow chart to justify your answer.

9.7 Lab report:

For the lab write-up, include

5. Your flowcharts and programs that you wrote before the lab.

6. A copy of your working .asm files.

7. A brief discussion of the objectives of the lab and the procedures performed in the lab.

8. Answers to any questions in the discussion, procedure, or question sections of the lab.
10

Analog-to-Digital Conversion

10.1 Objectives:
The MPS430F2013 is equipped with an analog-to-digital (ATD) conversion system that samples an analog (continuous) signal at regular intervals and then converts each of these analog samples into its corresponding binary value using a sigma-delta modulation technique. While doing this lab, you will learn,

- How to program the F2013’s ATD converter system.

10.2 Related material to read:

- Fundamental Concepts of the Sigma Delta A/D Conversion Method
- The F2013’s A/D Converter

10.3 The F2013’s Sigma Delta A to D conversion system:
The F2013 SD16_A conversion system consists of an 8-channel, multiplexed input, 16-bit output sigma delta analog-to-digital converter block. Its features include a software selectable internal/external voltage, a built-in temperature sensor, up to a 1.1 MHz modulator input frequency, and a selectable low-power conversion mode. The converter block is software programmable to perform either single or continuous conversions into a 16-bit output register called the SD16MEM0 register. The SD16_A module must be initialized using its two control registers, the SD16 control and channel control.
(SD16CTL & SD16CCTL0) registers. When the SD16_A module is not actively converting, it is automatically shut down to preserve power.

10.3.1 The SD16CTL register:

The SD16CTL register is mainly responsible for selecting the clock source, the division of the clock into the sigma delta modulator, and the enabling of the internal voltage reference.

```
<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD16CTL</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSD16</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD16DIV</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD16SEL</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD16VMIDC</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD16REFON</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD16OVIE</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 10.1 – The SD16_A Control Register (SD16CTL) at 0x0100**

The SD16 Clock Divider (SD16XDIVx & SD16DIVx): Bits 11 – 9 & 7 – 6

The clock signal selected for the sigma delta modulator (fM in the notes) is divided using two sets of bits, referred to as the clock divider bits (SD16XDIVx and SD16DIVx).

Having two different divisors allows for a very broad range of selectable modulator
frequencies. Figures 10.2 and 10.3 show the division of the clock signal according to the bit values.

<table>
<thead>
<tr>
<th>SD16XDIVx Bits</th>
<th>Clock Division Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>÷ 1</td>
</tr>
<tr>
<td>001</td>
<td>÷ 3</td>
</tr>
<tr>
<td>010</td>
<td>÷ 16</td>
</tr>
<tr>
<td>011</td>
<td>÷ 48</td>
</tr>
<tr>
<td>1xx</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Figure 10.2 – The SD16X Clock Divisor (SD16XDIVx) Bits**

<table>
<thead>
<tr>
<th>SD16DIVx Bits</th>
<th>Clock Division Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>÷ 1</td>
</tr>
<tr>
<td>001</td>
<td>÷ 2</td>
</tr>
<tr>
<td>010</td>
<td>÷ 4</td>
</tr>
<tr>
<td>011</td>
<td>÷ 8</td>
</tr>
</tbody>
</table>

**Figure 10.3 – The SD16 Clock Divisor (SD16DIVx) Bits**
The SD16 Clock Source Select (SD16SELx): Bits 5 – 4

The clock source to be divided is selected using the clock source select bits, much like the timer module. Figure 10.4 shows the clock that is selected for each bit configuration.

<table>
<thead>
<tr>
<th>SD16SELx Bits</th>
<th>Clock Source Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>MCLK</td>
</tr>
<tr>
<td>01</td>
<td>SMCLK</td>
</tr>
<tr>
<td>10</td>
<td>ACLK</td>
</tr>
<tr>
<td>11</td>
<td>External TACLK</td>
</tr>
</tbody>
</table>

**Figure 10.4 – The Clock Source Select Bits**

The SD16 Reference Generator ON (SD16REFON): Bit 2

The SD16_A module can use an internally provided reference voltage for modulation or it can be provided a user specified voltage reference through specified ports. The internally provided reference voltage has a value of 1.2 V and is used when the SD16REFON bit in the SD16CTL register is set to 1. For the lab, we will not use an external reference voltage, so we will need the internal reference voltage to be enabled.
### 10.3.2 The SD16CCTL0 register:

The SD16CCTL0 register is responsible for the conversion mode, the data output settings, the oversampling ratio, and all interrupt settings.
The SD16 Oversampling Rate (SD16XOSR & SD16OSRx): Bits 11 & 9 – 8

The extended oversampling ratio (SD16XOSR) bit works with the oversampling ratio (SD16OSRx) bits to set the oversampling rate of the converter. The ratio of $f_M/OSR$ equals $f_s$, the rate at which conversion data is output from the digital filter (should equal the frequency at which the original signal was sampled). The following figure shows how the OSR is selected:

<table>
<thead>
<tr>
<th>SD16XOSR Bit</th>
<th>SD16OSRx Bits</th>
<th>Oversampling Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>256</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>128</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>64</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>512</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1024</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Figure 10.7 – The Oversampling Rate Selection Bits
The SD16 Single Conversion Mode Select (SD16SNGL): Bit 10

The SD16 converter is programmable to perform either single or continuous conversions of a signal into the 16-bit output register (SD16MEM0). The SD16MEM0 register hold the output of the A/D conversion and should only be read in a SD16 interrupt service routine to prevent unpredictable results.

<table>
<thead>
<tr>
<th>SD16SNGL Bit</th>
<th>Conversion Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continuous</td>
</tr>
<tr>
<td>1</td>
<td>Single</td>
</tr>
</tbody>
</table>

**Figure 10.8 – The SD16MEM0 register at 0x0112**

The conversion mode of the SD16 _A_ module is selected with the SD16SNGL bit.
In continuous conversion mode, conversions begin when the SD16 Single Conversion (SD16SC) bit is set and continue until it is reset to zero by software. In single conversion mode, a single conversion occurs when the SD16SC bit is set to one. The SD16SC bit is then automatically reset by the hardware in preparation for the next conversion request.

Figure 10.10 – Single vs. Continuous Conversion Modes

Data Output Settings

Conversions can be written to the SD16MEM0 in three different modes:

1) Bipolar Offset Binary
2) Bipolar Two’s Compliment
3) Unipolar
Each mode can be selected by the SD16DF (Bit 4) and the SD16UNI (Bit 12) bits:

<table>
<thead>
<tr>
<th>SD16UNI</th>
<th>SD16DF</th>
<th>Format</th>
<th>Analog Input</th>
<th>SD16MEM0†</th>
<th>Digital Filter Output (OSR = 256)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Bipolar Offset</td>
<td>+FSR</td>
<td>FFFF</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Binary</td>
<td>ZERO</td>
<td>8000</td>
<td>80000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-FSR</td>
<td>0000</td>
<td>00000000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Bipolar 2’s</td>
<td>+FSR</td>
<td>7FFF</td>
<td>7FFFFFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>complement</td>
<td>ZERO</td>
<td>0000</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-FSR</td>
<td>8000</td>
<td>80000000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Unipolar</td>
<td>+FSR</td>
<td>FFFF</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ZERO</td>
<td>0000</td>
<td>80000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-FSR</td>
<td>0000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

† Independent of SD16CSRx and SD16XCSR settings; SD16LSBACC = 0.

Since the number of bits that are output in a conversion from the digital filter is dependent on the OSR and can vary from 15 to 30 bits, it is important to know which bits
are actually written to the SD16MEM0 register. Figure 10.13 shows the digital filter output and its relation for each OSR, LSBACC, and SD16UNI setting.

The Least Significant Bit Access (SD16LSBACC Bit 6) and Least Significant Toggle bits (SD16LSBTOG Bit 7) give access to the least significant bits of the digital filter output. If the LSBACC bit is set to 1, then the 16 least significant bits of the digital filter output are written to the SD16MEM0 register and can be read using either a word or a byte instruction. When the LSBTOG bit is set to 1 the LSBACC is toggled every time the SD16MEM0 register is read, allowing the complete digital filter output result to be read every two reads of the output register.
Figure 10.13 – The Digital Filter Output to the SD16MEM0 Register

Interrupt Settings
For any interrupt to be enabled on the SD16_A converter to be enabled, the GIE bit in the SR must first be set. There exist two interrupts sources for the SD16_A:

1) The SD16IFG flag (Bit 2) – Set when the new conversion results are available for reading in the SD16MEM0 register.
   a. The SD16IFG is automatically reset anytime the SD16MEM0 is read. It can also be cleared by software.
2) The SD16OVIFG (Bit 5) – Set if a new conversion result is written into the SD16MEM0 register before the previous conversion result was read.
   a. Cleared only by software.

Interrupt are serviced only if both the GIE and the interrupt enable bit (SD16IE: Bit 3) are set. Both interrupt requests for the SD16_A are serviced by a single interrupt vector at 0xFFEA. Hence, the ISR for an A/D conversion works much like the timer module’s TAIV register. The SD16 Interrupt Vector (SD16IV) register is used to determine which enabled interrupt requested the interrupt. The highest priority interrupt request will generate a number in the SD16IV register. This number can then be evaluated or added to the PC to automatically enter the appropriate software routine.

If at the return of an interrupt, another interrupt request is pending, the hardware will automatically adjust the value in the SD16IV register and request the next highest priority interrupt service routine.
10.3.3 The Input Settings
The SD16_A has up to five externally available differential input pairs (A0 – A5) that are multiplexed into the Programmable Gain Amplifier (PGA). The analog input into the device is configured using the Input Control (SD16INCTL0) and Analog Input Enable (SD16AE) registers. Setting the SD16AE bits enable the analog circuitry for the respective differential pair of input pins and disable any digital circuitry that might be linked to that pin. Look to the Terminal Functions resource for the analog differential pin mappings.

Figure 10.14 – The Analog Input Enable (SD16AE) Register at 0x00B7
The Input Control (SD16INCTL0) Register

The SD16INCTL0 Register shown in Figure 10.15 is responsible for setting the selected input channel, the gain of the PGA. You will not have to worry about the SD16INTDLYx bits for this lab. During conversion, any changes to the SD16GAINx or SD16INCHx bits will become effective at the next decimation step of the digital filter.

![Figure 10.15 – The Input Control (SD16INCTL0) Register at 0x00B0](image)

The SD16GAINx Bits (3 – 4) are responsible for setting the PGA gain for the analog input signals.

<table>
<thead>
<tr>
<th>SD16GAINx Bits</th>
<th>PGA Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>2</td>
</tr>
<tr>
<td>010</td>
<td>4</td>
</tr>
<tr>
<td>011</td>
<td>8</td>
</tr>
<tr>
<td>100</td>
<td>16</td>
</tr>
<tr>
<td>101</td>
<td>32</td>
</tr>
<tr>
<td>110</td>
<td>Reserved</td>
</tr>
<tr>
<td>111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

![Figure 10.16 – The PGA Gain Bits](image)
The SD16INCHx Bits (0 – 2) are responsible for selecting the analog input into the PGA to be modulated. Since only five inputs are externally available, they are the only ones you should consider.

<table>
<thead>
<tr>
<th>SD16GAINx Bits</th>
<th>PGA Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A0</td>
</tr>
<tr>
<td>001</td>
<td>A1</td>
</tr>
<tr>
<td>010</td>
<td>A2</td>
</tr>
<tr>
<td>011</td>
<td>A3</td>
</tr>
<tr>
<td>100</td>
<td>A4</td>
</tr>
<tr>
<td>101</td>
<td>A5 – (AVcc – AVss) / 11</td>
</tr>
<tr>
<td>110</td>
<td>Temperature Sensor</td>
</tr>
<tr>
<td>111</td>
<td>Short for PGA Offset Measurement</td>
</tr>
</tbody>
</table>

Figure 10.17 – The Input Channel Select Bit

Input Characteristics and Setup

Any input to the SD16 should pass through an external R-C anti-aliasing filter to prevent aliasing of the signal. The cutoff frequency should be < 10 KHz. A 100nF capacitor should also be connected from V_REF to GND to reduce the amount of noise created by the reference voltage.

**** Question for write-up: Is the user responsible for matching the input sampling frequency determined by the following equations
and the sampling frequency resulting from the fm/OSR division at the digital filter output? Do you just get aliased signal readings if you don’t match them? ****

### 10.4 Procedure:

In this lab, you will write a flowchart and a program that will convert an analog input applied to A/D port A4 to its digital value and display it in a watch window. Refer to the Terminal Functions for the pin mappings and connections.

Program the ATD conversion system on the board to convert the analog signal to a 16-bit number between 0x0000 and 0xFFFF in offset binary mode (so 0x0000 = -Vref/2 and 0xFFFF = Vref/2). Convert this number to values between -600 and 600 mVolts, represented as a 3-digit BCD number. Test your software by connecting the A/D input to GND, to VDD, and then to the analog signal from the DC voltage generator, adjusting the voltage setting and observing the changing output voltage in the program. Show the working program to your TA.

### 10.5 Questions:

3. If VREF+ = 1.2 V and VREF− = -1.2 V, what digital value is returned when the SD16_A system converts the following voltage using a 16-bit conversion in Bipolar Two’s Complement mode?

<table>
<thead>
<tr>
<th>Voltage (mV)</th>
<th>Digital Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.11</td>
<td>1.2 V</td>
</tr>
<tr>
<td>3.12</td>
<td>0.5 V</td>
</tr>
<tr>
<td>3.13</td>
<td>0 V</td>
</tr>
<tr>
<td>3.14</td>
<td>-1.2 V</td>
</tr>
</tbody>
</table>

### 10.6 Lab report:

For the lab write-up, include

9. Your flowcharts and programs that you wrote before the lab.
10. A copy of your working \textit{.asm} files.

11. A brief discussion of the objectives of the lab and the procedures performed in the lab.

12. Answers to any questions in the discussion, procedure, or question sections of the lab.
I. Introduction to the MSP430F2013 Microcontroller

A. Objectives
   1. Explain computer, hardware, and assembly language terms.
   2. Explain differences between separate addressing modes.
   3. Write a series of instructions to perform simple operations.

B. Basic Computer Concepts
   1. Computer is made of HW and SW
      a. the processor
      b. the memory
         • ROM vs RAM
      c. input devices
      d. output devices
   2. What is a microcontroller
      a. Central Processing Unit (CPU)
      b. Memory
      c. Control Circuits
      d. Input/Output Devices
         • Parallel I/O interface
         • Asynchronous/Synchronous interface
      e. Timers
      f. Analog to Digital/Digital to Analog converters
   3. Features of the EZ430
      a. EZ430-F2013 development tool including a USB debugging interface and detachable MSP430F2013 target board
      b. LED indicator
      c. Removable USB stick enclosure
      d. Debugging interface supports development with all MSP430F20xx devices
      e. Integrated IAR Kickstart user interface which includes an assembler, linker, simulator, source-level debugger and limited C-compiler
      f. Full documentation on CD-ROM
   4. Features of the MSP430F2013
      a. 16 bit CPU
      b. 16 MHz system clock
      c. 64KB address space
      d. 16 bit sigma-delta analog to digital converter
      e. Brown out reset circuitry
      f. In-system programming
      g. Universal Asynchronous Receiver Transmitter
      h. I²C bus
• An I²C Bus, or an Inter-Integrated Circuit bus uses two bidirectional open-drain lines, a serial data (SDA), and a serial clock (SCL) line.
  i. An open drain terminal is connected to ground in its logic 0 state but has high impedance in logic 1 state
  ii. This inhibits current flow so the device requires an external pull-up resistor connected to a positive voltage rail
• Advantages
  i. The pull up resistor doesn’t have to be connected to the same voltage as chip supply so you can interface two series of devices with different operating levels.
  ii. More than one output can be attached to a single wire because all devices can be in logic 1 states and it only takes one device in logic 0 state to sink the line to low

i. Orthogonal architecture
• An instruction set is said to be orthogonal if any instruction can use data of any type by any addressing mode

j. 16 registers
k. 7 addressing modes
l. 3 instruction formats
m. Watchdog timer
n. Individual timer module
o. Numbers can be represented in binary, octal, decimal, or hexadecimal format.

<table>
<thead>
<tr>
<th>Format</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1010b, b’1010’</td>
</tr>
<tr>
<td>Octal</td>
<td>1234q, q’1234’</td>
</tr>
<tr>
<td>Decimal</td>
<td>1234, -1, d’1234’</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>0xFFFFh, 0xFFFF, h’0xFFFF’</td>
</tr>
</tbody>
</table>

C. The Computer’s Software
1. Machine language
   a. Series of binary sequences
   b. Writing programs in machine language is extremely time consuming and inefficient
2. Assembly language created
   a. Assembly code program is made of mnemonics or code that represent machine code instructions
b. Examples:
   - The SUB instruction means subtract
   - The BIT means bitwise AND

c. A written assembly program is called the source code
d. Another program called an assembler then translates your source code into machine instructions
e. The resulting code is called object code.

D. Memory Addressing
1. Memory is a sequence of addressable locations.
   a. Can be used to store information such as data, instructions, and status of devices.
2. CPU sends the address of needed information across the memory address bus (MAB) of the chip.
3. Data is transferred through the Memory Data Bus (MDB) to and from the CPU or device that placed the request.

4. Amount of memory is referred to in bytes, or sets of 8 bits.
   a. 4 bits = nibble
   b. 8 bits = byte
   c. 16 bits = word
   d. 32 bits = long word
   e. $2^{10}$ bits = 1024 bits = KB (a kilobyte)
   f. $2^{20}$ bits = 1,048,576 bits = MB (a megabyte)
   g. $2^{30}$ bits = 1,073,741,824 bits = GB (a gigabyte)
   h. $2^{40}$ bits = 1,099,511,627,776 = TB (a terabyte)
5. MSP430 has a 16-bit addressing scheme, meaning a total of $2^{16}$ possible addresses, or 64K (65,536) different memory locations. Hence, it has a 64 KB address space.
6. Each address points a single byte in memory.
7. Single address space for entire microcontroller
   a. Special Function Registers (SFRs)
   b. Peripherals
   c. RAM
   d. FLASH/ROM memory
8. Data can be accessed as bytes or words

9. FLASH/ROM
   a. 2 kB FLASH memory in the F2013
   b. Stored in F4E0h to 0FFDFh (2 KB of data)
   c. Can be used for both code and data
      - Any data stored in the FLASH/ROM will be fixed, so it should be expected constants
   d. Interrupt vector table is mapped into the upper 16 words, with the highest priority vector at highest Flash/ROM word address (0FFF Eh).
10. RAM
    a. 128 Bytes of RAM in the F2013
    b. 0200h – 0280h
    c. Can be used for both code and data, but code is usually bigger than 128 bytes, so it is recommendable to load code into the ROM.
11. Peripheral Modules
    a. 16-bit modules should be accessed with word instructions
       i. If byte instructions are used, only the even addresses are permissible and high byte is always = 0
    b. 8-bit modules should be accessed with instructions only.
       i. If word instructions are used, the high byte is ignored
12. Memory Organization
    a. Bytes are located at both even and odd addresses
b. Words are located only at even addresses  
i. The low byte of a word is always at an even address. The high byte is stored in the next odd address.

13. The F2013’s Registers
14. CPU has the following features (from TI manual). Terms that you might not understand will be covered in later sections:
   a. 16 Registers (4 Special Function, 12 General Registers)  
   b. RISC architecture with 27 base instructions and 7 addressing modes  
   c. Orthogonal architecture – every instruction usable with every addressing mode  
   d. Full register access including program counter, status registers, and stack pointer  
   e. Single-cycle register operations  
   f. Large 16-bit register file reduces fetches to memory  
   g. 16-bit address bus allows direct access and branching throughout entire memory range  
   h. 16-bit data bus allows direct manipulation of word-wide arguments  
   i. Constant generator provides six most used immediate values and reduces code size  
   j. Word and byte addressing and instruction formats

15. Example picture:

16. 16 x 16 bit registers  
   a. 12 general purpose  
   b. 4 special function registers (SFRs)
17. Program Counter (R0)
   a. Points to the next instruction to be executed
   b. Each instruction uses an even # of bytes (two, four, or six) and the PC is incremented accordingly
      - Instruction accesses are performed on word boundaries (even addresses)

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

The PC can be addressed with all instructions and addressing modes. A few examples:

- MOV  #LABEL,PC ; Branch to address LABEL
- MOV  LABEL,PC ; Branch to address contained in LABEL
- MOV  #R14,PC ; Branch indirect to address in R14

18. Stack Pointer (R1)
   a. Used by CPU to return addresses of subroutine calls and interrupts
   b. Uses a pre decrement, post increment scheme
   c. Initialized to RAM by the user and aligned to even addresses

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- MOV  2(SP),R6 ; Item I2 -> R6
- MOV  R7,(SP) ; Overwrite TOS with R7
- PUSH  #012b ; Put 012b onto TOS
- POP   R0 ; R0 = 012b

19. Status Register / Constant Number Generator (R2)
   a. Can be used as a source or destination register
   b. Can be used in register mode only addressed with word instructions
   c. Remaining combinations of addressing modes used to support the constant generator

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>V</td>
<td>SCG1</td>
<td>SCG0</td>
<td>DSGCF</td>
<td>CPUOFF</td>
<td>GIE</td>
<td>N</td>
<td>Z</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   d. Description of Bits
20. Constant Generation Registers (Upper byte of R2 and R3)
   a. Without having to create and store an immediate value, the 6 most commonly used constants can be generated with the constant generator registers R2 and R3
   b. According to which addressing mode (As) is used with the CGRs, a different constant will be generated
   c. Addressing modes specify how the hardware recognizes operators and operands from the assembly instruction.
   d. The MSP430 has 7 addressing modes, but three of those modes are “emulated” using 4 basic modes. These 4 basic modes are represented by two binary digits in the table as (As).
   e. More on addressing modes later in the chapter.
   f. Constant number generator table with examples following:

<table>
<thead>
<tr>
<th>Register</th>
<th>As</th>
<th>Constant</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>00</td>
<td>- - - -</td>
<td>Register mode</td>
</tr>
<tr>
<td>R2</td>
<td>01</td>
<td>(0)</td>
<td>Absolute address mode</td>
</tr>
<tr>
<td>R2</td>
<td>10</td>
<td>00004h</td>
<td>+4, bit processing</td>
</tr>
<tr>
<td>R2</td>
<td>11</td>
<td>00008h</td>
<td>+8, bit processing</td>
</tr>
<tr>
<td>R3</td>
<td>00</td>
<td>00000h</td>
<td>0, word processing</td>
</tr>
<tr>
<td>R3</td>
<td>01</td>
<td>00001h</td>
<td>+1</td>
</tr>
<tr>
<td>R3</td>
<td>10</td>
<td>00002h</td>
<td>+2, bit processing</td>
</tr>
<tr>
<td>R3</td>
<td>11</td>
<td>FFFFFh</td>
<td>-1, word processing</td>
</tr>
</tbody>
</table>
g. Registers R2 and R3 used in constant mode cannot be addressed explicitly; they act as source only registers
h. Because of Constant Generation, “emulated instructions” are possible
i. Examples:

```
CLR dst
```

is emulated by the double-operand instruction with the same length:

```
MOV r3, dst
```

where the #0 is replaced by the assembler, and R3 is used with As=00.

```
INC dst
```

is replaced by:

```
ADD 0(r3), dst
```

21. General Purpose Registers (R4 – R16)
   a. All general purpose registers can be used as data registers, address pointers, or index values
   b. Can be accessed with both byte and word instructions.

E. Addressing Modes
   1. Please reference 16-bit RISC CPU (Section 3.3) for information on addressing modes.
II. Assembly Programming

F. CISC vs. RISC architecture

1. Complex Instruction Set Computer (CISC)
   a. Primary goal is to complete the programming task in as few instructions as possible
      i. Achieved by building special processor HW that executes a series of single instruction operations.
   b. Ex:
      \[ \text{MULT R5, R6, R7} \]
      Is a complex instruction that multiplies the numbers held in two registers (R5 and R6) and stores the result in R7
   c. Advantages
   d. compiler has to do little work translating a higher-level language to assembly
   e. little RAM is required to store instructions

2. Reduced Instruction Set Computer (RISC)
   a. Utilizes a small, highly optimized set of instructions
   b. Register-to-register instructions forces user to pull data into registers and minimizes memory access, which takes a long time.
      i. RISC machines therefore, have many registers in which to store necessary data
   c. Ex: MULT turns into (not necessarily 430 instructions):
      \[ \begin{align*}
      \text{LOAD A, [memory location]} \\
      \text{LOAD B, [memory location]} \\
      \text{PROD A,B} \\
      \text{STORE [memory location], A}
      \end{align*} \]
   d. Advantages
      • Each instruction requires only one clock cycle to execute
      • Reduced instruction set requires less transistors and less HW space, leaving room for general purpose registers
      • Because all instructions require the same amount of time, pipelining is possible

3. Table Comparison

<table>
<thead>
<tr>
<th></th>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emphasis on hardware</td>
<td>Includes multi-clock complex instructions</td>
<td>Emphasis on software Single-clock, reduced instructions only</td>
</tr>
<tr>
<td>Memory-to-memory</td>
<td>Memory-to-memory (LOAD and STORE incorporated into complex instructions)</td>
<td>Register-to-register, register-to-memory, and memory-to-register</td>
</tr>
<tr>
<td>Small code sizes, high cycles per second</td>
<td>Low cycles per second, large code sizes</td>
<td></td>
</tr>
<tr>
<td>Spends transistors for</td>
<td>Spends transistors on</td>
<td></td>
</tr>
</tbody>
</table>
4. MSP430 is a RISC architecture

G. Assembly Language Program Structure

1. An assembly program consists of three main components
   a. Assembler directives – Tell the assembler how to process the instructions. Also provide a way to define program constants.
   b. Assembly instructions
   c. Comments – Good comments for code will make it a lot easier to read and understand

2. Programs begin with a reset interrupt, meaning the act of activating the microcontroller triggers a Power-On Reset (POR) that the programmer must take into consideration at the very beginning of their program.

3. Until a branch is reached, the program will execute instructions in order. A program should always branch back into itself or, be told that it is done. Else, the processor will attempt to execute instructions in memory that have not been assigned and will give unpredictable results.

4. Control of program execution is called control flow.
   a. Operations that affect control flow:
      • branching/jumping
      • function calls
      • interrupts – to be covered later

5. Branching and jumping change the next instruction to be executed (held in the Program Counter) to a specified location in memory that’s not the next consecutive instruction
   • By branching to a point before the current instruction in program execution, you can force the program to repeat itself (loops)
   • Conditional branching checks the status register (SR) bits to decide whether to branch or not. See Instruction Reference for information on which instructions affect which SR bits

V. Assembly Code

A. Assembly is a human readable form of machine language.

1. Machine language is a series of bits that informs the processor what it needs to do with data.

2. For example, the following code is a single operation in machine language.

```
0001110010000110
```

For practical reasons, a programmer would rather use the mnemonic assembly representation for the previous operation.

```
ADD     R6,R2,R6     ; Add $R2 to $R6
```
3. Every instruction has a format:

```
[label] [operator] [operands] ;[comments]
```

4. ADD is a typical operator, sometimes referred to as an op-code in assembly language that tells the processor to add the operands, in this case the contents of register six and register two, and store the result back into register six.

5. Anything after the semicolon ‘;’ is considered a comment and is ignored by the assembler.

6. Labels are used as reference points for later instructions (for things like looping and function definition)

B. Often, assembly instructions are very simply and execute one simple instruction

C. Assembly has a one-to-one mapping to machine language. So each line of assembly corresponds to a machine language instruction that can be executed by the processor.

D. An assembler translates assembly code to machine language and a disassembler does the reverse

E. Common types of instructions

1. Moves
   a. Set register to a constant value
   b. Move data from memory to register or vice versa.
   c. Read and write data from memory devices and peripherals
   d. Data must be fetched into a register before a computation is performed.

2. Computation
   a. Add, subtract, multiply and divides
   b. Performing bitwise operations (ANDs, ORs, NOTs)
   c. Value comparison (> , < , <=, >=, ==)
   d. Shift and Rotate operations

3. Control Flow
   a. Jump to another location in program and execute instructions found in that section of memory
   b. Jump to another location if a certain condition holds
   c. Jump to another location, but save location of next instruction as return point (function call)

F. Reference 16-bit RISC CPU (Section 3.4) for an explanation of the MSP430 instruction set

VI. Assembler Introduction

A. Reference Assembler Operators for an explanation of the operators used in the IAR Embedded Workspace IDE.

B. Assembler Directives

1. The assembly program template
a. If you look for the above instructions in your set of operators, you will not find them. These are called **assembler directives**.

b. Assembler directives can have one, two, or three operands, separated by commas.

2. Module Control Directives
   a. These directives mark the beginning and end of source program modules, and for assigning names and types to them
      - The **NAME** directive
        - Begins a program module, and assigns a name for future reference
      - The **END** directive
        - Indicates the end of the source file. Any lines after the END directive are ignored

3. Symbol Control Directives
   a. These directives control how symbols are shared between modules
      - The **PUBLIC** Directive
        - *Exports* symbols to other modules
        - Use to make one or more symbols available to other modules.
        - Symbols declared as PUBLIC can be used in expressions
        - No limit to number of public variables
        - This way, PUBLIC can be used to declare global variables of 32-bit length across modules.
      - The **EXTERN** Directive
        - *Imports* an untyped, external symbol
The following example defines a subroutine to print an error message, and exports the entry address err so that it can be called from other modules. It defines print as an external routine; the address will be resolved at link time.

```assembly
NAME  error
EXTERN print
PUBLIC err

err CALL print
DB  "*** Error ***"
EVEN
RET
END
```

4. SIDE NOTE – Segment modes (When written up, place this section in its own “window” on the page separate from the rest of the material)
   a. A segment or section is a piece of code that cannot be split into smaller elements (a subroutine, an interrupt service routine, the main function)
      - Each assembly code file has at least one segment. The number of segments in code is limited by the number of memory slots available (in our case, 64k).
   b. A programmer should decide if he/she wants to use relocatable or absolute code in his/her application.
      - Assembler allows mix of relocatable and absolute code in a single source file.
      - Main difference between relocatable and absolute code is how symbol addresses are determined.
   c. Absolute Segments
      - For a segment to be absolute, its starting address must be defined at assembly time. The operand of the ORG directive determines the starting address of an absolute segment.
      - Example:

```
#include "msp430x20x3.h"

ORG  0F800h ; main program beginning
       ; Main program

ORG  0FFFFEH ; MSP430 RESET Vector

END
```

- The absolute section of code that begins at 0F800h would be where one would write his/her main program
- 0FFFFEH, if you look at the interrupt vector mapping in the address space, is where the reset vector routine is stored.
- Both are considered absolute segments
- It is the user’s responsibility to ensure no overlap between the absolutely defined segments of code. If segments do overlap, then the last segment that was loaded

d. *Relocatable Segments*
   - The starting address of a relocatable segment is determined at linking time.
   - User does not need to worry about overlapping segments. Linker takes care of it.

5. **Segment Control Directives**
   a. Control how code and data are located
   b. A *program location counter* is created for each segment of code by the assembler, initialized to zero, and then maintained. It can be affected by the Segment Control Directives.
      - This should not be confused as the *program counter*, which is kept in R1. There is only one program counter that points to the next instruction to be executed when the program is running. There are multiple program location counters, one for each segment that point to the current segment’s execution location.
         - This makes it possible to switch segments and modes anytime without the need to save the current program location counter.
         - The program location counter for the current segment of code is referenced to by the ‘$’ symbol.
         - If used in executable code, the ‘$’ symbol stands for the program counter, because the program location counter and the program counter will be aligned within the current segment.
   c. The ORG Directive
      - Sets the program location counter of a section to the value of the operand. Begins an *absolute* section of code if used outside of a relocatable segment.
      - Typically, since we only have 128 Bytes of RAM and we have 2 KB of FLASH/ROM, we want to set the operand for the ORG directive to somewhere between FFDFh and F7DFh (because FFDFh – F7DFh = 200h, the 2KB of FLASH/ROM, and closer to F7DFh since you want everything to fit)
         - The FFFEh location seen as the operand of the ORG directive in the template is the address of the MSP430’s reset routine. This puts the F2013 in its reset state.
   d. The RSEG Directive
      - Begins a *relocatable* section of code.
      - In the assembly code template from the beginning of the section, this is where we would write our main program code.
- It is possible to use the ORG directive inside a relocatable section of code, but the result of the expression must be of the same type as the current segment. Therefore, you couldn’t use ORG 10 (since the expression is absolute) but you could use ORG $+10$ (moving the program location counter 10 byte locations forward in memory).

- Example

  **Setting the location counter**

  The following example uses org to leave a gap of 256 bytes:

  ```
  NAME   org $+256
  begin  mov #12,R4
          sub R5,R4
          ret
  end
  ```

6. Value assignment directive - The EQU directive
   a. Use EQU to assign a permanent value to a symbol in a module
   b. Often used to create a local symbol that denotes a number or an offset
   c. Symbol is only valid in the module in which it was defined, but can be made available to other modules with a PUBLIC directive.
   d. Use EXTERN to import symbols from other modules.

7. The DEFINE directive
   a. Use DEFINE to define symbols that should be known to all modules in source file.
   b. A symbol which has been given a value with DEFINE can be made available to modules in other files with the PUBLIC directive.
   c. Symbols defined with DEFINE cannot be redefined within the same file.

8. The difference between EQU and DEFINE
   a. DEFINE represents a memory location
   b. The EQU directive does not allocate memory; it equates a value to a symbol for use by the assembler.
   c. Example showing difference between EQU and DEFINE
Using local and global symbols

In the following example the symbol value defined in module add1 is local to that module; a distinct symbol of the same name is defined in module add2. The DEFINE directive is used for declaring locn for use anywhere in the file:

```assembly
NAME     add1
locn    DEFINE    160h
value    EQU        77
MOV      locn, R4
ADD      #value, R4
ENDMOD

NAME     add2
value    EQU        88
MOV      locn, R5
ADD      #value, R5
END
```

The symbol locn defined in module add1 is also available to module add2.

9. C-Style Preprocessor Directives
   a. The #include directive
      i. Use the #include directive to insert the contents of another file (like a .h header file in C) into the source file at a specified point.
      ii. Two ways to use it:
          - #include “filename”
            searches the current source file’s directory for the specified file
          - #include <filename>
            searches the c-compiler’s provided pre-written library directory
            this is where you reference libraries like stdio.h or math.h

10. Data Definition and Allocation Directives

<table>
<thead>
<tr>
<th>Size</th>
<th>Reserve and initialize memory</th>
<th>Reserve uninitialized memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit integers</td>
<td>DC8, DB</td>
<td>DS8, DS</td>
</tr>
<tr>
<td>16-bit integers</td>
<td>DC16, DW</td>
<td>DS16, DS 2</td>
</tr>
<tr>
<td>32-bit integers</td>
<td>DC32, DL</td>
<td>DS32, DS 4</td>
</tr>
<tr>
<td>64-bit integers</td>
<td>DC64</td>
<td>DS64, DS 8</td>
</tr>
</tbody>
</table>

- Defining Strings
  i. To define a string:

      mymsg DC8 'Please enter your name'

  ii. To include a quote in the string, just write it twice in the string:

      mymsg DC8 'Hubert’s dog is happy'
iii. Use DC8 because an ASCII character is 7 bits long. Therefore, a string of ASCII characters can be held in memory as an array of 8 bit integer values

VII. Assembly Examples – small programs that will perform simple tasks to show you how a basic assembly program is written.

A. The MSP430’s reset state

1. The MSP430 has two kinds of resets.
   - Power On Reset (POR)
     i. When the device turns on (when the supply voltage for the device goes above a specified threshold voltage for the first time) or…
     ii. A low signal on the RST’/NMI pin when configured in reset mode or…
     iii. Other conditions you don’t need to worry about.
   - Power Up Clear (PUC)
     i. A POR signal occurs or…
     ii. Watchdog timer expires in active watchdog mode or…
     iii. Other conditions you don’t need to worry about.

2. Initial conditions after system reset (VERY IMPORTANT)
   - RST’/NMI pin is configured to reset mode
   - I/O pins are switched to input mode
   - Status Register (SR) is reset
   - Watchdog timer powers up in active watchdog mode.
   - Peripheral modules are reset to states specified in manual.
   - Program Counter (PC) is loaded with address contained at reset vector location 0FFFEh.

3. Software Initialization
   - Since the device is reset when it is first activated (Power On Reset), our programs must begin with a little more than the assembler template gives us.
   - We have to:
     i. Initialize the Stack Pointer (typically to top of RAM), or in our case, 0x0280
     ii. Initialize the watchdog timer to the requirements of the application.
     iii. Configure the peripheral modules to the requirement of the application.
   - The watchdog timer is a feature of the MSP430 we won’t be using in these labs, so we will simply turn it off.
   - The peripheral module initialization is application dependent.

4. Example
• #include “msp430x20x3.h” - includes the msp430x20x3.h header file that defines many useful constants in the MSP430 architecture, some of which you see in the StopWDT line.
• ORG 0F800h – Sets this segment’s program location counter to the absolute address, 0F800h, the beginning of our FLASH/ROM
• RESET – labels this first move instruction as RESET and initializes the stack pointer (SP) to the beginning of our RAM
• StopWDT – stops the watchdog timer so it will not affect further code
• ORG 0FFFEh / DC16 RESET – Creates the 16 bit constant, RESET, at the address, 0FFFEh. When the power-on reset sets the program counter to the value held in the reset interrupt vector, 0FFFEh, it will point to the instruction held at address RESET (The beginning of your program!)
• If you do not understand this, make sure you ask for help. This software initialization is crucial.

B. Writing programs to do simple arithmetic
   1. Addition
      • Write a program that:
         i. Stores the values 0x15 and 0x2 in memory locations 0xFF00 and 0xFF01, respectively
         ii. Adds the values using R4 and R5 and stores the result in R5.
2. Subtraction
   
   - Write a program that:
     
     i. Stores the values 0x15, 0x2, and 0x7 in memory locations 0x0200, 0x0201, 0x0204 respectively
     
     ii. Adds the contents of 0x0200 and 0x0201 and stores the result in 0x0201.
     
     iii. Subtracts the contents of 0x0204 from the contents of 0x0201 and stores the result back in 0x0201
3. The Carry/Borrow Flag
   - Write a program that:
     i. Adds the immediate numbers 0x8645 and 0x9978.

```
#include "msp430x20x3.h"

;--------------------------------------------------------------------------
; Program Reset
;--------------------------------------------------------------------------
RESET    mov.w #0280h,SP        ; Initialize stackpointer
StopWDI  mov.w #WDTPW+WDT HOLD,&WDTCTL ; Stop WDI
Main     mov.w #0x0200, R4
          mov.w #0x0204, R5
          add.b @R4, 1(R4) ; Add contents & address in R4 and
                              ; the address of (R4 + 1) into (R4 + 1)
          sub.b @R5, 1(R4) ; Subtract contents & address in R5
                              ; and address of (R4 + 1) into (R4 + 1)
;--------------------------------------------------------------------------
;  Interrupt Vectors
;--------------------------------------------------------------------------
ORG      0FFFFh
DC16     RESET
;--------------------------------------------------------------------------
; Data Section
;--------------------------------------------------------------------------
DATA     ORG 0200h       ; store my data in RAM
          DC8 0x15       ; store 0x15 in 0200h
          DC8 0x02       ; store 0x02 in 0201h
          ORG $-2        ; skip two bytes
          DC8 0x07       ; store 0x07 in 0204h
DC16     Main
END
```

- Notice that the result should be 0x11FBD because

```
0x8645  
+0x9978
```
Since the MSP430’s registers are only 16 bits wide, the register into which the sum is stored will only show 1FBD.

If you look at your registers in the debugging mode, which you will learn in the tutorial, you see that the Status Register (SR) has set bits V and C. C is the carry bit signifying the carry from the arithmetic operation. The V bit signifies this overflow from the 16 available bits.

- Now write a program that:
  i. Adds the two 32 bit numbers 0x01238645 and 0x02319978.
  ii. NOTE: To do this addition, we will first add the bottom word of both numbers and then add the high word of the number taking the Carry bit into account.

```c
#include "msp430x20x3.h"

; Program Reset

ORG 0F800h

// Initialize stackpointer
RESEI mov.w #0280h, SP

// Stop WDT
StopWDT mov.w #WDTEN+WDTHOLD+WDTCRL ; Stop WDT

// Load low byte of 0x01238645
Main mov.w #0x8645, R4
add.w #0x9978, R4
mov.w #0x0123, R5
adc.w #0x0321, R5

// Load high byte of 0x01238645
add.w #0x0321, R4

// Load high byte of 0x02319978 to R5 plus the carry bit
```

iii. The result is now stored in registers R5 and R4, as 0x04451FBD.
iv. If we had used add.w in place of the addc.w instruction, the result would have been 0x0441FBD and would have been incorrect.

• Write a program that:
  i. Subtracts 0x23487400 from 0x23493900

```c
#include "msp430x20x3.h"

; Program Reset

ORG 0F800h
RESEI mov.w #0280h,SP ; Initialize stackpointer
StopWDT mov.w #WDFW+WDTHold,WDTCTL ; Stop WDT
Main mov.w #0x3900,R4 ; Load low byte of 0x2349[3900]
subc.w #0x7400,R4 ; Sub low byte of 0x2348[7400] to R4
mov.w #0x2349,R5 ; Load high byte of 0x[2349]3900
subc.w #0x2348,R5 ; Sub high byte of 0x[2348]7400

; Interrupt Vectors

ORG 0FFFFh
DC16 RESEI
END
```

ii. The result should be 0x0000C500, stored in R5 and R4.

iii. Note the subc.w instead of the sub.w. Had we used sub.w, the result would have been 0x0001C500.

iv. Make sure you fully understand two’s complement arithmetic to grasp this concept.
C. Program Loops

1. Many applications require repetitive operations. These are often done using loops. A finite loop is executed a defined number of times. An infinite loop always executes.

2. Steps in a finite loop:
   - Initialize loop counter
   - Compare loop counter with a limit to see if it is within bounds. If not, break out of loop.
   - Increment (or decrement) counter
   - Execute loop contents

3. Status Register
   - The contents of the SR are used to determine loop branches.

   You have already been introduced to the carry flag. Pay attention only to the V, N, Z, and C flags for now.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Overflow bit. This bit is set when the result of an arithmetic operation overflows the signed-variable range. ADD (.B), ADDC (.B) Set when: Positive + Positive = Negative Negative + Negative = Positive, otherwise reset SUB (.B), SUBC (.B), CMP (.B) Set when: Positive - Negative = Negative Negative - Positive = Positive, otherwise reset</td>
</tr>
<tr>
<td>N</td>
<td>Negative bit. This bit is set when the result of a byte or word operation is negative and cleared when the result is not negative. Word operation: N is set to the value of bit 15 of the result Byte operation: N is set to the value of bit 7 of the result</td>
</tr>
<tr>
<td>Z</td>
<td>Zero bit. This bit is set when the result of a byte or word operation is 0 and cleared when the result is not 0.</td>
</tr>
<tr>
<td>C</td>
<td>Carry bit. This bit is set when the result of a byte or word operation produced a carry and cleared when no carry occurred.</td>
</tr>
</tbody>
</table>

4. Branch/Jump Instructions
   - Branch and jump instructions cause program flow to change when certain conditions exist. Look at the Instruction Reference for the function of different branch/jump instructions.

5. Code Examples
   - Write a program that:
     i. Statically creates an array of numbers in RAM.
     ii. Adds the numbers in a register using a loop.
D. Shift and Rotate Instructions
   1. Reference 16-Bit RISC CPU for a good explanation of shift and rotate instructions

E. Boolean Logic Instructions
   1. Bitwise logic is useful for setting/cleaning/toggling bits within data values
   2. Example: The following code sets the I/O port pin P1.0 to output by through a bitwise OR on the P1 direction register and then toggles that output using a bitwise XOR.

   ```
   SetupP1  bis.b  #001h,&P1DIR ; P1.0 output
          xor.b  #001h,&P1OUT ; Toggle P1.0
   ```
   3. For more information, reference the Instruction Reference or the 16-Bit RISC CPU documents.
Overview of the MSP430F2013’s I/O Ports

I. Basic I/O Concepts
   A. I/O devices are also called *peripherals* devices.
      1. Used to exchange data with other computers and hardware
      2. Examples:
         a. Switches
         b. LEDs
         c. Monitors
         d. Printers
   B. External devices, hardware and stimuli can have very different characteristics from our controller
      1. Therefore, we interface the outside world with special hardware (peripherals)
         a. Peripheral major function is to synchronize data transfers
         b. Peripheral interfaces contain (not necessarily all for every peripheral)
            • Control Registers – set up parameters for operation
            • Data Registers – intermediate storage for data transfers
            • Status Registers – report the progress of data transfers
            • Data Direction Registers – control dataflow (input or output)
            • Control Circuitry – high level reactions to data
      2. These peripherals all communicate with our CPU through what’s called the Master Data Bus (MDB)
         a. 16 bits wide
         b. Only one device is allowed to transfer info over the MDB
         c. Data flows in two directions:
            • From CPU → Peripherals
            • From Peripherals → CPU
         d. When a device is transmitting, it’s called the Master on the MDB
         e. When a device is receiving, it’s called a Slave on the MDB
      3. The CPU accesses its peripherals through the Master Address Bus
         a. Also 16 bits wide
         b. This is how the peripherals can be part of the 64KB address map of the MSP430

II. Digital I/O with the MSP430
   A. Intro
      1. MSP430’s have up to 6 digital I/O Ports (P1 – P6)
         a. The F2013 has 2 ports, P1 and P2, only one which is accessible on the EZ430 (P1)
      2. Each port has 8 I/O pins
         a. Each pin is individually configurable for input or output
         b. Each I/O line can be written and read to
      3. P1 and P2 have interrupt capabilities
         a. Each interrupt can be individually enabled and configured to provide interrupt on rising or falling edge
b. All P1 lines source a single interrupt vector
c. All P2 lines source a single, different interrupt vector

4. Features:
   a. Independently programmable individual I/O pins
   b. Any combination of input/output
   c. Individually configured P1 and P2 interrupts
   d. Independent Input/Output Registers
   e. Individually configured pull-up and pull-down resistors

B. I/O operation
   1. Input Register: PxIN
      a. Each bit in each PxIN register reflects the value of the input signal at
         the corresponding I/O pin when pin is configured as an I/O function.
      b. If the PxINx = 0, input is low
      c. If the PxINx = 1, input is high

   2. Output Register: PxOUT
      a. Each bit in each PxOUT register reflects the value to be output when
         the pin is configured to:
         - I/O function
         - Output direction
         - Pull-up/Pull-down resistor is disabled
            - If resistor is enabled, the corresponding bit in PxOUT reg
              selects pull-up or pull-down
            - 0 = Pin is pulled down
            - 1 = Pin is pulled up

      NOTE: When using the same port for both input and output, as in this lab,
      the contents of bits set to output will be reflected in PxIN. Therefore, it is
      important to mask the PxIN register (AND the contents of PxIN) with 0’s
      at any bits that are set to output pins when reading from the register.

   3. Direction Registers
      a. Each bit in each PxDIR register sets direction of I/O.
      b. If the PxDIRx = 0, input
      c. If the PxDIRx = 1, output

   4. Pull-up/Pull-down resister enable register PxREN
      a. If the PxDIRx = 0, disabled
      b. If the PxDIRx = 1, enabled

   5. Function selecting register PxSEL
      a. Port pins are commonly multiplexed with other peripheral module
         functions.
         - Therefore, you must tell the processor that you need the pin to
           operate in I/O mode
      b. If the PxSELx = 0, the pin is selected to operate in its I/O function
      c. If the PxSELx = 1, the pin is selected to operate in its peripheral
         module function
         - Different peripheral modules have different PxDIRx
         requirements and specifications, so check documentation for
         functionality
NOTE: if PxSELx = 1, pin’s interrupt function is disabled
6. P1 and P2 interrupts
   a. Interrupt Flag Registers P1IFG, P2IFG
      • The interrupt flags for the ports are set on the programmer-specified edge.
      • The interrupt flags must be reset by software
      • These interrupt flags can also be set by software, providing SW interrupts!

NOTE: Writing to P1OUT, P1DIR, P2OUT, or P2DIR can result in setting P1IFG or P2IFG
7. Interrupt Edge Select Registers P1IES and P2IES
   a. If the PxIESx = 0, the PxIFGx sets on a rising edge
   b. If the PxIESx = 1, the PxIFGx sets on a falling edge

NOTE:

<table>
<thead>
<tr>
<th>PxIESx</th>
<th>PxINx</th>
<th>PxIFGx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 → 1</td>
<td>0</td>
<td>May be set</td>
</tr>
<tr>
<td>0 → 1</td>
<td>1</td>
<td>Unchanged</td>
</tr>
<tr>
<td>1 → 0</td>
<td>0</td>
<td>Unchanged</td>
</tr>
<tr>
<td>1 → 0</td>
<td>1</td>
<td>May be set</td>
</tr>
</tbody>
</table>

8. Interrupt Enable Registers P1IE, P2IE
   a. If the PxIE = 0, the interrupt function is disabled
   b. If the PxIE = 1, the interrupt function is enabled
9. Configuring unused Port Pins
   a. Unused port pins should be configured
      • I/O fn
      • Output Direction
      • Left unconnected on PC board to reduce power consumption
   b. Alternatively, integrated pull-up and pull-down resistors can be enabled (PxREN) to prevent floating input
10. On the F2013, the address mapping for these registers is as such:

<table>
<thead>
<tr>
<th>Port</th>
<th>Register</th>
<th>Short Form</th>
<th>Address</th>
<th>Register Type</th>
<th>Initial State</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Input</td>
<td>P1IN</td>
<td>020h</td>
<td>Read only</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Output</td>
<td>P1OUT</td>
<td>021h</td>
<td>Read/Write</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Direction</td>
<td>P1DIR</td>
<td>022h</td>
<td>Read/Write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Interrupt Flag</td>
<td>P1IFG</td>
<td>023h</td>
<td>Read/Write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Interrupt Edge Select</td>
<td>P1IES</td>
<td>024h</td>
<td>Read/Write</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Interrupt Enable</td>
<td>P1IE</td>
<td>025h</td>
<td>Read/Write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Port Select</td>
<td>P1SEL</td>
<td>026h</td>
<td>Read/Write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Resistor Enable</td>
<td>P1REN</td>
<td>027h</td>
<td>Read/Write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td>P2</td>
<td>Input</td>
<td>P2IN</td>
<td>028h</td>
<td>Read only</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Output</td>
<td>P2OUT</td>
<td>029h</td>
<td>Read/Write</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Direction</td>
<td>P2DIR</td>
<td>02Ah</td>
<td>Read/Write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Interrupt Flag</td>
<td>P2IFG</td>
<td>02Bh</td>
<td>Read/Write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Interrupt Edge Select</td>
<td>P2IES</td>
<td>02Ch</td>
<td>Read/Write</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Interrupt Enable</td>
<td>P2IE</td>
<td>02Dh</td>
<td>Read/Write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Port Select</td>
<td>P2SEL</td>
<td>02 Eh</td>
<td>Read/Write</td>
<td>003h with PUC</td>
</tr>
<tr>
<td></td>
<td>Resistor Enable</td>
<td>P2REN</td>
<td>02Fh</td>
<td>Read/Write</td>
<td>Reset with PUC</td>
</tr>
</tbody>
</table>
Interrupts, Resets and Operation Modes

I. Fundamentals of Interrupts
   A. What is an interrupt?
      1. Interrupts are among the most useful tools that a microcontroller can provide. Through the use of interrupts, CPU utilization is improved and I/O operations can be made much more efficient.
      2. An interrupt is an event in the microcontroller’s execution that forces the CPU to stop current execution and somehow service the event.
         a. In the last lab, you will notice that you had to place the address of the beginning line of code into the memory location FFEEh. This is the memory location of the Power On Reset (POR) Interrupt Service Routine (ISR), the routine that gets called every time that the event of a power on reset occurs.
      3. Interrupts can be internal or external.
         a. Internal interrupts are interrupts that are generated by internal hardware or by the software that was written for the specific application.
            • Software interrupts are commonly referred to as traps or exceptions and are attributed to abnormal program execution such as division by zero or overflow
         b. External interrupts are interrupts that are generated by external circuitry that sets an interrupt flag for some reason
   B. Why are interrupts used?
      1. Free up the CPU
         a. Polling keeps CPU in redundant cycles where it is constantly executing instructions
         b. Interrupts are handled as they occur, freeing up CPU execution cycles for background processes or other functions in general.
      2. Multitasking
         a. CPU’s perform many routine tasks. For example, assume that every 10 seconds, the CPU must check the status of a certain I/O register. One could use a timer to count down from 10 seconds and then fire an interrupt, whose interrupt service routine would check the I/O register.
         b. Another good example of multitasking using interrupts is operating systems. Operating systems commonly give each application that’s running on the machine a certain amount of CPU processing time in a sort of round-robin scheme. Since these increments of time are very, very small, this allows the user to feel like each application he/she is using on the computer has access to the full CPU. This is implemented through a timer count-down resulting in an interrupt on the zero count!
      3. Error handling
         a. Exceptions can be thrown by software and interrupts can provide good knowledge as to why they occurred.
   C. Maskability of Interrupts
      1. Interrupts can be considered maskable or nonmaskable.
2. Nonmaskable interrupts are interrupts that cannot be ignored by the CPU unless individually and explicitly disabled.
3. Maskable interrupts can be ignored by the CPU using an enable bit.
4. An interrupt being disabled means that if the event occurs, it will be ignored by the CPU.
5. And interrupt being enabled means that the CPU will service the event every time it occurs.
6. There are usually both global and local interrupt enable bits.

D. Interrupt Priority
1. When there are multiple interrupts pending, the CPU must decide which interrupts get serviced first. This decision is made according to the priority of the specified interrupt.
2. On the MSP430, the interrupt priorities are fixed and defined by the arrangement of modules shown:

E. MSP430 Interrupts
1. There are 3 kinds of interrupts:
   a. System Resets
   b. Nonmaskable interrupts (NMI)
   c. Maskable interrupts
2. System Resets
   a. System circuitry sources both a power-on reset (POR) and a power-up clear (PUC) signal.
   b. POR – Power On Reset is triggered by:
      - Powering up the device
      - Low signal on RST'/NMI when configured in reset mode
      - SVS low condition when PORON = 1
   c. PUC – Power Up Clear is triggered by:
      - A PUC is always generated when a POR is generated, but not vice versa
• Watchdog timer expiration when watchdog mode is active
• Watchdog timer security key violation
• Flash memory security key violation
• CPU instruction fetch from peripheral address range 0h – 01FFh
d. Brown Out Reset
• Detects low supply voltages such as when a supply is applied or removed from the Vcc terminal.
• Resets the device by triggering a POR signal
• There are more details to a BOR you won’t necessarily have to worry about
e. MSP430’s initial conditions after a POR (review)
• RST’/NMI pin configured in reset mode
• I/O pins are all switched to input mode
• The Status Register is reset
• The watchdog timer powers up in active watchdog mode
• Peripheral modules and registers are initialized as described in the User’s Guide
• Program Counter (PC) is loaded with the address contained at the reset vector location (0FFFeh)
f. Therefore, the necessary software initialization after a POR (also review):
• Initialize the Stack Pointer (SP), typically to the top of RAM
• Initialize the Watchdog timer for application requirements
• Configure peripheral modules for application requirements

3. Non-Maskable Interrupts
a. Not masked by the General Interrupt Enable (GIE) bit but by individual interrupt bits
b. An NMI Can be generated by 3 sources:
• Edge on the RST’/NMI pin when configured in NMI mode (masked with the NMIE bit)
• Oscillator Fault (masked with the OFIE bit)
• Access violation to Flash memory (masked with the ACVIC bit)
c. The RST’/NMI Pin (DOESN’T WORK through a JTAG port!!!!!!)
• The function of the RST’/NMI pin is selected in the WDCTL register (watchdog control register) at memory location 0x0120. A description of the bits of interest.
The WDTCTL register @ 0x0120

- **Bit 5 : WDTNMI – NMI Select**
  1. WDTNMI = 0: Reset Mode
  2. WDTNMI = 1: NMI Mode

- **Bit 6 : WDTNMIES - NMI Edge Select**
  1. WDTNMIES = 0: NMI on rising edge
  2. WDTNMIES = 1: NMI on falling edge

  **IMPORTANT NOTE:** When NMI mode is selected and the edge select bit is changed, an NMI can be generated, depending on the level at the RST/NMI pin.
  1. Hence, the edge select bit should be changed before selecting the NMI mode.

- **Reset Mode**
  - If the RST’/NMI pin is set to *Reset Mode*, CPU is held at reset as long as the RST’/NMI pin is held low

- **NMI Mode**
  - If the RST’/NMI pin is set to *NMI mode*, NMI interrupts must be specifically enabled by the NMIIIE bit (Non-Messageable Interrupt Enable) in the IE1 Register (Interrupt Enable) @ address 0x00.

  1. Since other bits in the register are used by different peripherals, it is recommended to set the NMIIIE bit using the **bis.b** or **bic.b** operators.
  - A signal edge selected by the WDTNMIES bit generates an NMI interrupt and sets the NMIIIFG bit (Non-Messageable Interrupt Flag) in the IE2 Register (Interrupt Flag Register).

    1. The NMIIIFG bit must be reset by software.
    2. Once again, it is recommended to clear the NMIIIFG using the **bis.b** or **bic.b** operators.

  **IMPORTANT CONSIDERATIONS:**
  1. If the controller is set in NMI mode, the signal to RST’/NMI pin should not be held low because a PUC reset will reset the controller in the Reset mode, therefore keeping the program in reset state until it sees a 1 on the RST’/NMI pin.
    a. The easiest workaround is for the NMI interrupt to be set to fire on the falling edge of a signal.
2. Holding this pin at logic “1” means we need to attach a pull-up resistor (64k) from $V_{CC}$ to the pin so we can limit the current going into the pin.

3. After an NMI interrupt is accepted, the NMIIE (interrupt enable) bits are automatically reset and must be set again by the software.

d. Flash Access Violation
   - Can be enabled through the ACCVIE bit
   - The ACCVIFG bit can then be tested
   - Probably won’t have to worry about this one for the lab

e. Oscillator Fault
   - Warns of a possible error condition with the crystal oscillator
   - OFIE bit enables the oscillator fault NMI
   - OFIFG flag can then be tested
   - Probably won’t have to worry about this one either

f. When an NMI is accepted, all NMI bits are automatically reset
   - Software must set required NMI enable bits for interrupt to be re-enabled

g. NOTE: To prevent nested NMI interrupts, ACCVIE, OFIE and NMIE bits should not be set inside of an NMI ISR.

4. Maskable Interrupts
   a. On the MSP430, maskable interrupts are caused by peripherals with interrupt capabilities and the watchdog timer
   b. Each individual interrupt can be disabled or all can be disabled with the GIE bit in the status register

F. Interrupt Processing
   1. The procedure for handling an interrupt:
      a. Any currently executing instruction is completed
      b. The Program Counter (PC) that points to the next instruction is pushed onto the stack
      c. The Status Register (SR) is pushed onto the stack
      d. If multiple interrupts occurred, the highest priority interrupt is selected
      e. **Single source flags are reset automatically, and multiple-source flags remain set for servicing by software.**
      f. SR is cleared, terminating any low-power mode.
      g. The Global Interrupt Enable (GIE) bit is cleared, so further interrupts are disabled.
      h. The contents of the interrupt vector are loaded into the PC, meaning the program continues at that address.
      i. NOTE: The latency to handle an interrupt is 6 cycles.

2. Return from an Interrupt
   a. SR with the previous settings pops from the stack. All previous settings are now in effect.
   b. The PC pops from the stack and returns execution at the point it was interrupted.
   c. NOTE: The interrupt handling routine terminates with the instruction:
NOTE: There is a 5 cycle latency for a return from an interrupt to pull the necessary information from the stack.

3. Interrupt Nesting
   a. Allowing interrupts to occur within other interrupts is called interrupt nesting.
   b. This is enabled if the Global Interrupt Enable bit is set inside an Interrupt Service Routine.
   c. If enabled, any interrupt will interrupt any other interrupt, regardless of priority.

G. Interrupt Vectors
1. An interrupt vector refers to the starting address of the respective interrupt’s Interrupt Service Routine (ISR).
2. An interrupt service routine is the routine to be executed for the respective interrupt.
3. Interrupt vectors are generally stored in an interrupt vector table.
4. The interrupt vector table for the MSP430F2013:

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Flag</th>
<th>System Interrupt</th>
<th>Word Address</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power-up</td>
<td>PORIFG</td>
<td>Reset</td>
<td>0FFFEh</td>
<td>31, highest</td>
</tr>
<tr>
<td>External reset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Watchdog Timer+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash key violation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC out-of-range (see Note 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NMI</td>
<td>NMIIFG</td>
<td>(non)-maskable, (non)-maskable</td>
<td>0FFFFAh</td>
<td>29</td>
</tr>
<tr>
<td>Oscillator fault</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash memory access violation</td>
<td>OFFIFG</td>
<td></td>
<td>0FFFF8h</td>
<td>28</td>
</tr>
<tr>
<td>(see Note 2)</td>
<td>ACCV/IFG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(see Note 2 and 4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparator_A+ (MSP430x20x1 only)</td>
<td>CAIFG (see Note 3)</td>
<td>maskable</td>
<td>0FFF8h</td>
<td>27</td>
</tr>
<tr>
<td>Watchdog Timer+</td>
<td>WDTIFG</td>
<td>maskable</td>
<td>0FFF4h</td>
<td>26</td>
</tr>
<tr>
<td>Timer_A2</td>
<td>TACCR0 CCIFG (see Note 3)</td>
<td>maskable</td>
<td>0FFF2h</td>
<td>25</td>
</tr>
<tr>
<td>Timer_A2</td>
<td>TACCR1 CCIFG, TAIFG (see Note 2 and 3)</td>
<td>maskable</td>
<td>0FFF0h</td>
<td>24</td>
</tr>
<tr>
<td>ADC10 (MSP430x20x2 only)</td>
<td>ADC10IFG (see Note 3)</td>
<td>maskable</td>
<td>0FFE8h</td>
<td>21</td>
</tr>
<tr>
<td>SD16_A (MSP430x20x3 only)</td>
<td>SD16CCTL0 SD16OVIIFG</td>
<td>maskable</td>
<td>0FFE8h</td>
<td>20</td>
</tr>
<tr>
<td>USI (MSP430x20x2, MSP430x20x3 only)</td>
<td>USIIIFG, USIITIIFG</td>
<td>maskable</td>
<td>0FFE8h</td>
<td>20</td>
</tr>
<tr>
<td>I/O Port P2 (two flags)</td>
<td>P2IFG,6 to P2IFG,7</td>
<td>maskable</td>
<td>0FFE0h</td>
<td>19</td>
</tr>
<tr>
<td>I/O Port P1 (eight flags)</td>
<td>P1IFG,0 to P1IFG,7</td>
<td>maskable</td>
<td>0FFE4h</td>
<td>18</td>
</tr>
<tr>
<td>(see Note 5)</td>
<td></td>
<td></td>
<td>0FFE2h</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0FFE0h</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0FFD Eh ... 0FFC0h</td>
<td>15 ... 0, lowest</td>
</tr>
</tbody>
</table>

5. An interrupt vector is programmed by the user with the 16-bit address of the corresponding ISR being placed inside the respective Interrupt Vector

6. An example:
Description: A hi/low transition on P1.4 will trigger P1_ISR which

toggles P1.0. Normal mode is LPM4 ~ 0.1uA.
Internal pullup enabled on P1.4.
ACLK = n/a, MCLK = SCLK = default DCO

MSP430F20xx

/\       XIN
|   -- RST XOUT --
/\  P1.4-o P1.0--LED
|/

M. Buccini / L. Westlund
Texas Instruments Inc.
September 2005
Built with IAR Embedded Workbench Version: 3.40A

#include "msp430x20x3.h"

:0:000h ; Program Reset

RESET mov.w #0280h,SP ; Initialize stackpointer
StopWDT mov.w #WDPW+WDTHOLD,&WDTCRL ; Stop WDT
SetupPl mov.b #001h,&PIIDIR ; P1.0 output, else input
mov.b #010h,&P1OUT ; P1.4 set, else reset
bis.b #010h,&P1REN ; P1.4 pullup
bis.b #010h,&P1IE ; P1.4 interrupt enabled
bis.b #010h,&P1IES ; P1.4 hi/low edge
bic.b #010h,&P1IFG ; P1.4 IFG Cleared
Mainloop bis.w #LPM4+GIE,SR ; LowPowerMode4, enable interrupts
nop ; Required only for debugger

P1_ISR; Toggle P1.0 Output

xor.b #001h,&P1OUT ; P1.0 = toggle
bic.b #010h,&P1IFG ; P1.4 IFG Cleared
reti ; Return from ISR

interrupt vectors

| ORG | OFF4eh | MSP430 RESET VECTOR |
| ORG | OFF44h | P1.x Vector |
| DW | RESET |
| DW | P1_ISR |
Timer_A2 on the MSP430

I. Introduction
   A. Why a whole separate module for timing? Don’t we have already have a clock?
      1. Applications of timers
         a. Time-delay creation and measurement
         b. Period and pulse width measurement
         c. Frequency measurement
         d. Event counting
         e. Arrival time comparison
         f. Waveform generation
         g. Periodic interrupt generation ← What we’re focusing on in this lab!
      2. By having its own hardware module, Timer_A2 gives the user a lot more
         flexibility regarding how he/she uses the clocks provided on the MSP430.
      3. In fact, a lot of the module’s features make time measurements, comparisons,
         and counts a whole lot easier to program.
         a. Instead of coding from the ground up, we can use the provided features!

B. Timer A2
   1. Supports multiple captures and compares (2 capture/compare registers)
      a. Captures - The Timer_A2 module has a register that latches the contents
         of the 16-bit timer when a predefined event occurs.
         • An event means that a pre-programmed rising or falling edge has
           occurred
         • By comparing multiple captures, you can start to see which
           measurements can be made
           - Pulse widths
           - Periods
           - Duty cycles
           - Event arrival times
           - Timing references
      b. Compares – allows the 16-bit timer value to be compared with a specific
         value.
         • According to the comparison results, the compare function of the
           Timer_A2 module can:
           - Set flags in a register
           - Generate interrupts
           - Trigger an action on certain pins
         • Generally used for:
           - Time delays ← What we’re focusing on in this lab!
           - Action triggering for some future time
           - Generating digital waveforms
   2. Timer_A2 features
      a. Asynchronous 16-bit timer/counter with 4 operating modes
      b. Selectable and configurable clock source
      c. Two capture/compare registers
d. Configurable outputs with Pulse Width Modulation (PWM) capabilities
e. Asynchronous input/output latching
f. Interrupt Vector Register for fast decoding of all Timer_A2 interrupts

II. Operation
A. 16-bit Timer Counter Register (TAR)
1. Increments or decrements according to mode of operation with each rising edge of the clock signal
2. Can generate in interrupt on overflow
3. Counter is cleared by setting the TACLR bit in the TACTL register
   a. Setting the TACLR bit also clears the clock divider and count direction for Up/Down mode
   b. TACLR is always automatically reset.
4. NOTE: Always stop the timer before modifying operation or reading from the TAR register to avoid errant operating conditions
5. Clock source select and divider
   a. Clock can be sourced from the following clock signals using the TASSELx bits:
   
   - Remember that setting the TACLR bit resets the IDx bits and, therefore, any division of the sourced clock

B. Starting the Timer
1. Timer is started when:
   a. MCx bits > 0 and clock source is active
   b. When the timer is in Up/Down mode, the timer may be stopped by writing 0 to the period register, TACCR0.
      - The timer can then be restarted by writing a non-zero value to TACCR0.
      - When restarted, the timer starts incrementing in the up direction from zero.
2. Timer Mode Control

<table>
<thead>
<tr>
<th>MCx</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Stop</td>
<td>Timer is halted.</td>
</tr>
<tr>
<td>01</td>
<td>Up</td>
<td>Timer repeatedly counts from zero to the value of TACCR0.</td>
</tr>
<tr>
<td>10</td>
<td>Continuous</td>
<td>The timer repeatedly counts from zero to 0FFFFh.</td>
</tr>
<tr>
<td>----</td>
<td>------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>11</td>
<td>Up/Down</td>
<td>The timer repeatedly counts from zero up to the value of TACCR0 and back down to zero.</td>
</tr>
</tbody>
</table>

a. **“Up” mode**
- Used if timer period must be different from 0FFFFh counts.
- Repeatedly counts up to the compare register, \( TACCR0 \), which defines the period
  - NOTE: The number of timer counts in a period is \( TACCR0 + 1 \)

**Up Mode**

- The \( TAIFG \) (Timer A Interrupt Flag) is set when the timer goes to zero.

**Changing the Period Register \( TACCR0 \) while in an active mode:**
- If the new period counter value is greater than the current value stored in the \( TACCR0 \) register, the timer will simply count to the new period.
- If the new period counter value is less than the current value stored in the \( TACCR0 \) register, the timer will roll to zero and start anew.

b. **“Continuous” mode**
- The timer repeatedly counts up to 0FFFFh and restarts from zero as shown

- The \( TAIFG \) flag is set when the counter counts from FFFFh to zero.
“Up/Down” mode
- NOT NEEDED FOR LAB, BUT SHOULD BE WRITTEN IN THE FUTURE FOR TEXT ON MSP430

3. Capture/Compare blocks
a. Two capture/compare blocks, TACCRx are present in the timer and can be used to:
   - capture timer data or
   - generate time intervals
b. Capture mode
   - Used to record time events
   - Selected when CAP bit is set to 1 in TACCTL Register.
   - There exist two possible capture inputs per capture/compare block. One of the pins must be selected as the input capture pin.
     - CCIxA and CCIxB
     - The pins are connected to specified external pins or internal signals. The input pin is selected with the CCISx bits.

<table>
<thead>
<tr>
<th>CCISx Bits 13 - 12</th>
<th>Selected TACCRx input signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>CCIxA</td>
</tr>
<tr>
<td>01</td>
<td>CCIxB</td>
</tr>
<tr>
<td>10</td>
<td>GND</td>
</tr>
<tr>
<td>11</td>
<td>Vcc</td>
</tr>
</tbody>
</table>

The GND and Vcc are only used for software-initiated captures. You will not use this feature for your lab.

- The CCIxA and CCIxB bits correspond to the following I/O pins

<table>
<thead>
<tr>
<th>CCIx bit</th>
<th>I/O Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCI0A</td>
<td>P1.1</td>
</tr>
<tr>
<td>CCI1A</td>
<td>P1.2</td>
</tr>
<tr>
<td>CCI1B</td>
<td>P1.6</td>
</tr>
</tbody>
</table>

- CMx bits select capture edge of input signal as rising, falling, or both

<table>
<thead>
<tr>
<th>CMx bits 15 - 14</th>
<th>Capture Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No capture</td>
</tr>
<tr>
<td>01</td>
<td>Capture on rising edge</td>
</tr>
</tbody>
</table>
- When a capture occurs:
  - The value of the timer is copied into the TACCRx register
  - The CCIFG interrupt flag in the TACCTLx register is set
- The input signal level can be read at anytime through the CCI bit (TACCTLx register), but it should be noted that the capture signal can be asynchronous to the timer clock, causing a race condition
  - This can and should be avoided by setting the SCS bit in the TACCTLx register. Setting the SCS bit will synchronize the capture with the next timer clock.
- If a second capture occurs before the first capture is read, the COV bit in the TACCTLx register is set to 1.
  - The COV bit must be reset by software.

**c. The Capture Cycle**

**d. Compare Mode**
- Used to generate Pulse Width Modulated output signals (PWM signals) or interrupts at specific time intervals
- For our labs, we will not use the compare mode, as we will not generate any PWM signals and we will use the overflow interrupt to measure time, not the compare mode interrupts.

**C. Timer A Interrupts**
- Interrupts are enabled by setting the TAIE (Timer A Interrupt Enable) bit in the TACTL, or Timer A Control register.
- Two different interrupt vectors are associated with the 16-bit Timer_A
  - TACCR0 interrupt vector for TACCR0 CCIFG
  - TAIIV interrupt vector for all other CFIG flags and the TAIFG
c. CCIFG flags are set:
   a. In capture mode – when timer value is captured in the associated TACCRx register
   b. In compare mode – when TAR counts to the associated TACCRx value

d. Software may set or clear any CCIFG flags, allowing for user-specified SW interrupts.

e. All CCIFG flags request an interrupt when their corresponding CCIE bit and the GIE bit are set.

f. TACCR0 CCIFG interrupts
   a. Has the highest timer_A interrupt priority
   b. TACCR0 CCIFG flag is reset every time the TACCR0 interrupt request is serviced

g. TAIV interrupt vector generator
   a. TACCR1 CCIFG, TACCR2 CCIFG (EZ430 only has TACCR1) and TAIFG flags are prioritized and combined to source a single interrupt vector.
   b. The interrupt vector TAIV is used to determine which flag requested the interrupt
   c. The highest priority, enabled interrupt generates a number in the TAIV register that can then be added to the Program Counter to automatically enter the appropriate software routine.

---

TAIV, Timer_A Interrupt Vector Register

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TAIVx</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

TAIVx Bits 15-0 Timer_A Interrupt Vector value

<table>
<thead>
<tr>
<th>TAIV Contents</th>
<th>Interrupt Source</th>
<th>Interrupt Flag</th>
<th>Interrupt Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>No interrupt pending</td>
<td>-</td>
<td>Highest</td>
</tr>
<tr>
<td>02h</td>
<td>Capture/compare 1</td>
<td>TACCR1 CCIFG</td>
<td></td>
</tr>
<tr>
<td>04h</td>
<td>Capture/compare 2</td>
<td>TACCR2 CCIFG</td>
<td></td>
</tr>
<tr>
<td>06h</td>
<td>Reserved</td>
<td>-</td>
<td>Lowest</td>
</tr>
<tr>
<td>08h</td>
<td>Reserved</td>
<td>-</td>
<td>Lowest</td>
</tr>
<tr>
<td>0Ah</td>
<td>Timer overflow</td>
<td>TAIFG</td>
<td></td>
</tr>
<tr>
<td>0Ch</td>
<td>Reserved</td>
<td>-</td>
<td>Lowest</td>
</tr>
<tr>
<td>0Eh</td>
<td>Reserved</td>
<td>-</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

1 Not implemented in MSP430F1611x, devices
d. Any access of the TAIIV register (read or write) automatically resets the highest pending interrupt flag

e. If another interrupt flag is set, another interrupt is immediately generated after servicing the original interrupt

f. TAIIV example

; Interrupt handler for TACCR0 CCIFG.
CCIFG_0_HND
;
... ; Start of handler Interrupt latency 6
RETI 5

; Interrupt handler for TAIIV, TACCR1 and TACCR2 CCIFG.

TA_HND ... ; Interrupt latency 6
ADD &TAIV,PC ; Add offset to Jump table 3
RETI ; Vector 0: No interrupt 5
.JMP CCIFG_1_HND ; Vector 2: TACCR1 2
JMP CCIFG_2_HND ; Vector 4: TACCR2 2
RETI ; Vector 6: Reserved 5
RETI ; Vector 8: Reserved 5

TAIPG_HND ; Vector 10: TAIIPG Flag
... ; Task starts here
RETI 5

CCIFG_2_HND ; Vector 4: TACCR2
... ; Task starts here
RETI ; Back to main program 5

CCIFG_1_HND ; Vector 2: TACCR1
... ; Task starts here
RETI ; Back to main program 5

2. Register descriptions
### a. TACTL Register

**TACTL, Timer_A Control Register**

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAESLx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>IDx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TACLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAIE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAIFG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Unused** Bits: Unused 15-10
- **TAESLx** Bits: Timer_A clock source select
  - 9-8: TACLK, ACLK, SMCLK, INCLK
- **IDx** Bits: Input divider. These bits select the divider for the input clock.
  - 7-6: /1
  - 5-4: /2, /4
  - 3-0: /8
- **MCx** Bits: Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.
  - 5-4: Stop mode: the timer is halted
  - 10: Continuous mode: the timer counts up to 0FFFFh
  - 11: Up/down mode: the timer counts up to TACCR0 then down to 0000h

### b. TAR Register

**TAR, Timer_A Register**

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TARx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **TARx** Bits: Timer_A register. The TAR register is the count of Timer_A.
  - 15-0
c. TACCTLx Register

**TACCTLx, Capture, Compare Control Register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Capture mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-14</td>
<td>00 No capture</td>
</tr>
<tr>
<td></td>
<td>01 Capture on rising edge</td>
</tr>
<tr>
<td></td>
<td>10 Capture on falling edge</td>
</tr>
<tr>
<td></td>
<td>11 Capture on both rising and falling edges</td>
</tr>
</tbody>
</table>

**CCISx Bit**
Capture/compare input select. These bits select the TACCRx input signal.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Capture/compare input select</th>
</tr>
</thead>
<tbody>
<tr>
<td>13-12</td>
<td>See the device-specific datasheet for specific signal connections.</td>
</tr>
<tr>
<td></td>
<td>00 CCIA</td>
</tr>
<tr>
<td></td>
<td>01 CCIB</td>
</tr>
<tr>
<td></td>
<td>10 GND</td>
</tr>
<tr>
<td></td>
<td>11 Vcc</td>
</tr>
</tbody>
</table>

**SCS Bit**
Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Synchronize capture source</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0 Asynchronous capture</td>
</tr>
<tr>
<td></td>
<td>1 Synchronous capture</td>
</tr>
</tbody>
</table>

**SCCI Bit**
Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit.

**Unused Bit**
Unused. Read only. Always read as 0.

**CAP Bit**
Capture mode

<table>
<thead>
<tr>
<th>Bit</th>
<th>Capture mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0 Compare mode</td>
</tr>
<tr>
<td></td>
<td>1 Capture mode</td>
</tr>
</tbody>
</table>

**OUTMODx Bits**
Output mode. Modes 2, 3, 6, and 7 are not useful for TACCRD because EQUx = EQUO.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Output mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-5</td>
<td>000 OUT bit value</td>
</tr>
<tr>
<td></td>
<td>001 Set</td>
</tr>
<tr>
<td></td>
<td>010 Toggle/reset</td>
</tr>
<tr>
<td></td>
<td>011 Set/reset</td>
</tr>
<tr>
<td></td>
<td>100 Toggle</td>
</tr>
<tr>
<td></td>
<td>101 Reset</td>
</tr>
<tr>
<td></td>
<td>110 Toggle/set</td>
</tr>
<tr>
<td></td>
<td>111 Reset/set</td>
</tr>
</tbody>
</table>

d. TAIV Register
d. Address Mapping

Table 8-3. Timer_A Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Short Form</th>
<th>Register Type</th>
<th>Address</th>
<th>Initial State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer_A control</td>
<td>TACTL</td>
<td>Read/write</td>
<td>0160h</td>
<td>Reset with POR</td>
</tr>
<tr>
<td>Timer_A counter</td>
<td>TAR</td>
<td>Read/write</td>
<td>0170h</td>
<td>Reset with POR</td>
</tr>
<tr>
<td>Timer_A capture/compare control 0</td>
<td>TACCTLO</td>
<td>Read/write</td>
<td>0162h</td>
<td>Reset with POR</td>
</tr>
<tr>
<td>Timer_A capture/compare 0</td>
<td>TACCR0</td>
<td>Read/write</td>
<td>0172h</td>
<td>Reset with POR</td>
</tr>
<tr>
<td>Timer_A capture/compare control 1</td>
<td>TACCTL1</td>
<td>Read/write</td>
<td>0164h</td>
<td>Reset with POR</td>
</tr>
<tr>
<td>Timer_A capture/compare 1</td>
<td>TACCR1</td>
<td>Read/write</td>
<td>0174h</td>
<td>Reset with POR</td>
</tr>
<tr>
<td>Timer_A capture/compare control 2</td>
<td>TACCTL2†</td>
<td>Read/write</td>
<td>0166h</td>
<td>Reset with POR</td>
</tr>
<tr>
<td>Timer_A capture/compare 2</td>
<td>TACCR2†</td>
<td>Read/write</td>
<td>0176h</td>
<td>Reset with POR</td>
</tr>
<tr>
<td>Timer_A interrupt vector</td>
<td>TAIV</td>
<td>Read only</td>
<td>012Eh</td>
<td>Reset with POR</td>
</tr>
</tbody>
</table>

† Not present on MSP430x20xx Devices

3. Example Code for a Timer Overflow
[Texas Instruments sample assembly code: msp430x20x3_ta_03.s49]

a. Analysis – new lines of code to understand
   a. `#include "msp430x20x3.h"` declares:
      - TASSEL_2 – Sets the TASSELx bits so the timer sources the
        SMCLK at 1.1MHz and not the other clocks.

      ```
      #define TASSEL_0 (0*0x100u) /* Timer A clock source select: 0 - TACLK */
      #define TASSEL_1 (1*0x100u) /* Timer A clock source select: 1 - ACLK */
      #define TASSEL_2 (2*0x100u) /* Timer A clock source select: 2 - SMCLK */
      #define TASSEL_3 (3*0x100u) /* Timer A clock source select: 3 - INCLK */
      ```

   - MC_2 – Sets the MCx bits in the TACTL register so the timer is in
     Continuous Up mode

   NOTE: It is defined as 2*0x100u because that would translate to an
   ‘u’nsigned 0x200, or 001000000000b
- **TAIE** – Sets the timer interrupt enable bit in the TACTL register

```c
#define TAIE (0x0002) /* Timer A counter interrupt enable */
```

- **TACTL** – The address of the Timer A control register
- **TAIV** – Address of the Timer A interrupt vector

```c
#define TAIV (0x012E) /* Timer A Interrupt Vector Word */
#define TACTL (0x0160) /* Timer A Control */
```

- **CPUOFF** – CPUOFF, when set, turns off the CPU
- **GIE** – Sets the interrupt enable bit for all maskable interrupts

```c
#define GIE (0x0008)
#define CPUOFF (0x0010)
```

```assembly
SetupTA mov.w #TASSEL_2+MC_2+TAIE,&TACTL ; SMCLK, contmode interrupt
```

b. Logically OR’s the TASSEL_2 bits, the MC_2 bits, and the TAIE bits

\[
\begin{array}{c}
000100000000 \\
00100000 \\
+0010 \\
001001000010
\end{array}
\]

Moves the result into the address of the TACTL register, therefore initializing the timer.

```assembly
Mainloop bis.w #CPUOFF+GIE,SR ; CPU off, interrupts enabled
; Required only for debugger
```

c. Logically OR’s the CPUOFF bit and the Global Interrupt Enable bit

\[
\begin{array}{c}
0000000000001000 \\
0000000000001000 \\
+0000000000001000 \\
00000000000011000 \rightarrow \text{Status Register}
\end{array}
\]

Then sets the status register bits to the result, turning off the CPU (to reduce power consumption – remember, this is an Ultra-Low Power microcontroller) and enabling maskable interrupts.
d. This is a common ISR for programs that service interrupts from CCR1 and the TAIFG only.
   - If you were using the TACCR0 CCIFG flag, you would write its own ISR.
   - This ISR uses the TAIV generator. Note how the first line adds the contents of the TAIV register to the PC counter. If you look at the TAIV specifications, the different interrupt flags being set result in a multiple of 0x02 being generated in the TAIV register. Since `reti` is a two-byte instruction, adding the contents of the TAIV register to the program counter results in a user-specified Interrupt Vector Table.
   - In this case, the programmer wants to toggle the 0-bit in P1OUT anytime that the TAIFG is set. Can you tell how this happens?
I. Fundamental Concepts of an A/D Converter

A. The world we live in is analog
   1. Examples
   2. Weight
   3. Sound
   4. Touch
   5. Air Flow
   6. Temperature
   7. Everything we interact with exists within an infinite set of values that we must digitize if we want to compute
   8. Hence, an analog to digital converter is used

B. A/D converters take only electric wave inputs
   1. Anything else must be translated to electric waves if it is to be processed
   2. Transducers are devices that convert quantities from other signals to voltage inputs
   3. Sometimes, the signals transducers provide are not within a suitable range for an A/D converter to process.
      a. A voltage scaler or a voltage shifter is then applied to amplify signals or reduce signals within range for our controller
      b. These circuits are called signal-conditioning circuits

C. The accuracy of an A/D converter is dictated by the number of bits it uses to represent voltage values. This number is called the A/D converter’s resolution.
   1. An ADC of resolution can therefore represent $2^N$ separate voltage values.

D. Three major conversion methods:
   1. Flash
   2. Sigma-Delta
   3. Successive Approximation

E. The MSP430x20x2 and MSP430x22x4 devices implement 10-bit, high performance, successive approximation ADCs.

F. The MSP430x20x3 devices (OUR DEVICE!) implement a 16-bit Sigma-Delta ADC.

II. The Sigma-Delta Analog to Digital Conversion Method

A. Digitization Error
   1. To discuss why a sigma-delta ADC works, we must first discuss error.
      a. The voltage range of an analog input is infinite whereas we can only compute discrete voltages.
      b. We have to sample the analog signal at discrete time intervals to be able to read its voltage values.
      c. The loss of accuracy due to sampling introduces an error that makes a 100% reconstruction of the signal nearly impossible. Since we can’t
remove the error, we must analyze it and see what we can do to minimize it.

2. The following graph shows how a digital representation of a sampled analog wave is always going to lose information

![Graph showing digital representation of a sampled analog wave](image)

(Walt Kester, mysterious formula)

a. The dotted line represents the analog input, and the “steps” represent the possible digital output states.
b. Each digitized “step” equates to the weight of the ADC’s least significant bit (LSB).
   - This is application specific
   - For example: A 2 Vp-p split into $2^{16}$ (65536) levels means each “level” from 0x0000 to 0xFFFF is worth about 30.5uV.

3. The error due to sampling, as shown below, can be estimated to be (at most) $\frac{1}{2} \times$ LSB.

![Graph showing quantization of noise as a function of time](image)

(Walt Kester, mysterious formula)

The quantization of noise as a function of time

4. The RMS value of error
   a. The equation for error can be described as:
      
      $$ e(t) = st, \quad -q/2s < t < +q/2s. \quad \text{Eq. 1} $$

      Where:
      - $s =$ slope of the error fxn
b. For all analog inputs and their respective digital outputs, the equivalent error can be represented by the $\text{RMS}_{\text{error}}$.

c. The mean square of the error is:

$$e^2(t) = \frac{1}{q} \int_{-q/2}^{+q/2} (st)^2 \, dt$$

which simplifies to

$$e^2(t) = \frac{q^2}{12}.$$  \hspace{1cm} \text{Eq. 2}

d. The square root results in the RMS value:

$$\text{rms quantization noise} = \sqrt{e^2(t)} = \frac{q}{\sqrt{12}}.$$  \hspace{1cm} \text{Eq. 3}

e. As you can see, this error will be relatively small in comparison to a signal that spans from -1 to +1V

B. The Input Signal

1. An input, full-scale digital sine wave can be characterized by the equation:

$$\text{Input FS Sinewave} = v(t) = \frac{q2^N}{2} \sin(2\pi ft).$$  \hspace{1cm} \text{Eq. 5}

where:

a. $N$ is the ADC resolution

b. $f$ is the frequency of the sine wave

c. From our example:

$$((30.5uV^216)/2) \sin(2\pi f') = 1\sin(2\pi f')$$

2. The RMS voltage of the input wave can be calculated in much the same manner:

$$\text{rms value of FS input} = \frac{q2^N}{2\sqrt{2}}.$$  \hspace{1cm} \text{Eq. 6}

C. The Signal To Noise Ratio (SNR)

1. The ratio of RMS values is called the signal to noise ratio (SNR).

a. The SNR is the quantified determination of how accurate a digital conversion will be, and therefore, the resolution that an ADC can physically realize.
Hence, for an N-bit ADC the SNR will be $6.02N + 1.76 = 7.78\text{dB}$

2. The SNR is analyzed in the *frequency domain*.
   a. The frequency domain is a term used to describe the analysis of mathematical functions or signals with respect to frequency. (Wikipedia)
   b. Speaking non-technically, a *time domain graph* shows how a signal changes over time, whereas a *frequency domain graph* shows how much of the signal lies within each given frequency band over a range of frequencies (Wikipedia)
   c. This is based on a concept you will learn in Linear Systems I and II called the Fourier series that states any waveform can be expressed as a sum of sinusoids (sometimes infinitely many). Each sinusoid’s contribution to the signal is shown in the frequency graph.

3. A frequency analysis of the SNR shows the original signal spike as well as an evenly distributed distortion introduced by the signal sampling.

- **The Frequency Domain**

   - **Fs** is the *sampling frequency*.
     - To prevent *aliasing*, a signal must be sampled at a minimum Fs of twice its original frequency. ($F_s > 2F_0$) (LECTURE?)
     - Due to the properties of the Fourier Series, a signal will have equal, reflected components in the negative frequency domain to $-F_s/2$, meaning that all of our signal’s components will be represented within the bounds ($-F_s/2$, $F_s/2$)
   - What the graph shows us is that the distortion (also referred to as *noise*) of the signal due to error is spread out randomly and evenly over the frequency band.
c. If we can reduce the *Average Noise Floor* in relation to the original signal frequency, we can increase the SNR and add “bits” to our ADC’s resolution.

D. Oversampling

1. Because the Gaussian properties of noise will spread it evenly throughout the sampling frequency band, we can use a technique called *oversampling* to drop the average noise floor and therefore increase the SNR.
2. Oversampling is the technique of sampling a signal by a factor of $k$ times the original sampling frequency $F_s$ $(kF_s)$
   a. The resulting frequency domain graph shows the improvement in SNR due to oversampling.

   ![Oversampling by K Times](image)

b. The RMS of the noise is still the same, so the SNR will stay the same unless...

   ![The Digital Filter](image)

   - We pass the resulting signal into a low pass filter!
   - When we filter out the higher frequency components, we lose a large portion of the noise frequency components, therefore reducing the RMS of the error and increasing the SNR.

c. It turns out that for every factor of 4, we get an improvement of 6.02 dB – which is equivalent to gaining a single bit of resolution because $\text{SNR} = (6.02N + 1.76)$ dB
   - As you can see, this is realizable only up to a certain point
- An oversampling rate of $4^3$, or 64X the original sampling frequency, results in a resolution of 4 bits for our ADC
- We have 16 bits. This means we need an oversampling rate of $4^{15}$ to realize our ADC! This is obviously impossible.

d. We need another method of improving the SNR
e. We turn to a method called noise shaping.

E. Noise shaping

1. To understand noise shaping, we look at what is called a *sigma-delta modulator*. We will begin our analysis of modulators with a 1st order sigma-delta modulator.

![Diagram of a 1st order sigma-delta modulator](maxim-ic.com)

2. It includes a difference amplifier, an integrator, and a comparator with feedback loop that contains a 1-bit DAC. (This DAC is simply a switch that connects the negative input of the difference amplifier to a positive or a negative reference voltage.) The purpose of the feedback DAC is to maintain the average output of the integrator near the comparator's reference level (0V).
   a. The density of "ones" at the modulator output is proportional to the input signal. For an increasing input the comparator generates a greater number of "ones," and vice versa for a decreasing input.
   b. The goal is to average the resulting 1-bit string to a multi-bit, equivalent value using the digital filter previously discussed.
   c. Though a bit scary looking at first, let’s split the components and analyze what they do to understand how the modulator is useful.

3. The Differential Amplifier
   a. A *differential amplifier* takes two signals as its input and outputs the difference between them $V_{\text{diff}} = (V^+ - V^-)$ amplified by a gain. For our purposes, let’s assume a gain of 1.
   b. Since $V^-$ is attached to a reference voltage, the differentiator outputs the difference between an input signal and the maximum values of that input signal.

4. The Integrator
a. The purpose of an integrator is to collect the differential outputs over time. By doing so, it collects the running total of input offset from the reference voltage.

b. According to whether that total offset is positive or negative, a quantizer will output a 1 or a 0 (respectively) to the digitizer.

5. The quantizer
   a. Made up of the comparator/DAC feedback loop
      - The comparator compares the latest output of the integrator to its reference voltage (0V). If the integrator output was < 0 V, outputs a 0. If the integrator output was > 0V, outputs a 1.
   b. According to the comparator’s output, the DAC will output either the positive or negative version of the modulator’s reference voltage into the differential input.

6. Overall concept
   a. The result is an output string of length \( m \) that reflects, in its ratio of ones vs. zeros, the average value of the last \( m \) iterations in relation to the modulator’s reference voltage.

   b. This is possible because the integrator collects the differences between the inputs and the reference voltage. If the input is increasing, the differences will add to a value higher than 0, resulting in more ones seen at the output. If the input is decreasing, it will subtract from the total of differences until zeros start showing up at the output.
      - Notice that to make sure the integration stays bounded over time and doesn’t have the opportunity to increase towards infinity, the input voltage range to the modulator will be dependent on:
        - The voltage reference value and
        - The gain of the differential amplifier
   c. So how does this refer back to noise shaping?
      - The modulator can be seen as \( y[n] = x[n] + E[x[n - 1]] \)
        - Wherein \( y \) is the outbound sample value
        - \( x \) is the inbound sample value
- n is the sample number
- \( E(x[n]) = (x[n] \pm V_{REF}) + E(x[n-1]) \) is the accumulated error from the differentiator

- This formula can also be read: The outbound sample is equal to the inbound sample plus the error from the previous inbound sample. (Wikipedia)

- The transfer function for the modulator turns out to be, for a modulator of order L

\[
Y(z) = X(z) + \left(1 - z^{-1}\right)^L E(z)
\]

- Meaning the noise transfer function for our 1\textsuperscript{st} order modulator is:
  \( Y(z) / E(z) = H(z) = 1 / (1 - z^{-1}) \)
- \( H(z) \rightarrow 1 \) as \( w_0 \rightarrow \infty \ldots \) also known as a high pass filter
  
  d. Thus, most of the quantization noise is pushed into higher frequencies. Oversampling has changed not the total noise power, but its distribution.

\[
\text{Noise Shaped Spectrum}
\]

- Note that now, oversampling has changed not the total noise power (as that remains the same), but its distribution towards higher frequencies.

e. And now, when we apply the digital low pass filter to the modulator output, more of the noise has been shaped towards higher frequencies and gets filtered out!
f. This first-order noise shaping results in 9 dB of improvement every oversampling factor of 2 – a considerable improvement over 6dB’s every factor of four.

g. As it turns out, the addition of another integrator in the modulator sequence will further shape the noise towards higher frequencies. Each addition of a stage will continue to add higher decibels of noise suppression. The only thing that keeps ADC designers from implementing more than two stages is stability issues with the modulator that we will not discuss.

h. The sigma-delta ADC on the MSP430F2013 employs a 2nd order modulator, resulting in 15 dB of improvement for every factor of two increase of the oversampling ratio, $k$.

i. A great applet for understanding how a modulator works can be found [here](#).

F. The Decimation Filter (understanding comb-filters)

1. The output of the sigma-delta modulator is a 1-bit data stream at the sampling rate, which can be in the megahertz range. Decimators average this output stream to a multiple of the original reference voltage
2. The purpose of the digital-and-decimation filter is to extract information from this data stream and reduce the data rate to a more useful value.

3. How it works
   a. Decimation filters are an implementation of a moving average filter.
      • Equation for a moving average filter:
        \[
        y(n) = \frac{1}{D} \left[ x(n) + x(n-1) + x(n-2) + x(n-3) + \ldots + x(n-D+1) \right]
        \]
        where
        - D = the number of samples in the averaging window
        - y(n) is the output average
        - x(n) is the current input
   b. It can be proven through substitution that there exists an equivalent filter called a recursive running-sum filter with the equation:
        \[
        y(n) = \frac{1}{D} \left[ x(n) - x(n-D) \right] + y(n-1)
        \]
        • “The input subtraction portion of the CIC filter is called the “comb” section whose delay is D, while the feedback section is typically called an integrator. The comb stage subtracts a delayed input sample from the current input sample, and the integrator is simply an accumulator.”
        • Can be implemented in hardware as:
        ![Diagram of decimation filter](image)
        where the delay D = R, meaning the frequency is being decimated a factor of the delay
   c. The equation can be analyzed in the z-domain as
        \[
        H(z) = \frac{1}{D} \frac{1 - z^{-D}}{1 - z^{-1}}.
        \]
   d. From these equations, you can tell that since each output of the function needs D previous inputs to be calculated, the averaging filter is going to reduce the frequency of the output by a factor of D
   e. A frequency analysis \((z = e^{j2\pi f})\) of the transfer function results in
f. Applying Euler’s equation:

\[
H_{\text{cic}}(e^{j2\pi f}) = \frac{1 - e^{-j2\pi D}}{1 - e^{-j2\pi f}} = \frac{e^{-j2\pi D/2}(e^{j2\pi D/2} - e^{-j2\pi D/2})}{e^{-j2\pi f/2}(e^{j2\pi f/2} - e^{-j2\pi f/2})}
\]

\[
= e^{-j2\pi (D-1)/2} \frac{2j \sin(\pi f / 2)}{2j \sin(\pi f / 2)}.
\]

g. In the frequency domain, looks like (assuming D = 8):

- The BW is set by the 3dB points of the main lobe
- “The bandwidth shaded spectral bands centered about multiples of \( f_{\text{s,in}}/8 \) will alias directly into our passband after decimation by 8” (straight out of the website) Notice the biggest component of noise turns out to be -16dB.
- Notice that the smaller the filter’s BW, the less aliasing that occurs after decimation

h. To increase attenuation of noise, we can increase the order of the filter by cascading filters
- results in a frequency response of:

\[
|H_{\text{cic,Mth-order}}(e^{j2\pi f})| = \left| \frac{\sin(\pi fD)}{\sin(\pi f)} \right|^M.
\]
- These filters are usually referred to as \( \text{sinc}^M \) filters
i. Now, this looks like looks like:

![Diagram](image)

- The filter’s largest noise aliasing is now only -48dB, instead of -16dB

j. The MSP430F2013 uses a sinc³ filter.

III. Signal Conditioning Circuits

A. The optimal voltage range for the S16_A_ACD

1. As stated earlier, what we will refer to as our maximum full scale range voltage $V_{FSR}$ is dependent on two factors:
   a. The reference voltage chosen or supplied by the user
   b. The gain supplied by our differential output. In the EZ430 the gain is generated by a Programmable Gain Amplifier (PGA)

   $$V_{FSR} = \frac{V_{REF}/2}{GAIN_{PGA}}$$

2. For a 1.2 V reference, the maximum full-scale input range for a gain of 1 is:

   $$\pm V_{FSR} = \frac{1.2V/2}{1} = \pm 0.6V$$

B. The Voltage Scaling Circuit

1. *** develop ***

C. The Voltage Shifting/Scaling Circuit

1. ***** develop ****
The F2013’s A/D Converter

IV. The MSP430 S16_A ADC
A. Diagram

B. ADC Core
1. Analog-to-digital conversion takes in one of the differential input signals, amplifies by $G_{\text{PGA}}$ (like a differential amplifier in the sigma-delta ADC description), and outputs the result to a 1-bit, 2\textsuperscript{nd} order Sigma-Delta Modulator.

2. A single-bit comparator within the modulator quantizes the input signal with the modulator frequency $f_M$, which is sourced from one of 4 clocks and can be divided by two logic stages to be fed into the modulator.
   a. The SMCLK = MCLK = 1.1 MHz at normal operation conditions
   b. The clock divider bits are in the $SD\_16\_A$ Control Register ($SD\_16\_CTL$)
      - Bits 5 – 4: SD16SSELx bits ($SD\_16\_A$ Clock Source Select)

      | SD16SSELx bits | SD16_A Clock Source Select |
      |----------------|----------------------------|
      | 00             | MCLK                      |
      | 01             | SCLK                      |
      | 10             | ACLK                      |
      | 11             | External TCLK             |

      - Bits 7 – 6: SD16DIVx bits ($SD\_16\_A$ Clock Divider)

      | SD16DIVx bits | SD16_A Clock Divider |
      |---------------|----------------------|
      | 00            | 1                    |
      | 01            | 2                    |
      | 10            | 4                    |
      | 11            | 8                    |
• Bits 11 – 9: SD16XDIVx bits (SD16_A Clock Divider)

<table>
<thead>
<tr>
<th>SD16DIVx bits</th>
<th>SD16_A Clock Divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>3</td>
</tr>
<tr>
<td>010</td>
<td>16</td>
</tr>
<tr>
<td>011</td>
<td>48</td>
</tr>
<tr>
<td>1xx</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

3. The resulting 1-bit data stream is then averaged by the digital filter for the conversion result, which is stored in the register SD16MEM0.

4. Can you see how this circuit parallels to the example circuit given earlier?

(Demistifying Sigma -.... maxim-ic.com)

C. The Voltage Reference Generator

1. Diagram

2. The SD16_A has a built in 1.2V reference that is enabled by the SD16REFON bit (Bit 2) in the SD16CTL register.
   a. When using the internal reference, an external 100nF capacitor should be connected from VREF to AVSS (analog ground) to reduce noise in the reference voltage. On the EZ430:
b. \( V_{\text{REF}} \) is mapped to P1.3 (Pin 5 of 14)

c. If you look on the data sheet AV\text{SS} is not accessible on the EZ430, but we can connect the capacitor to \( V_{\text{SS}} \) (ground reference) instead. \( V_{\text{SS}} \) is mapped to Pin 14 of 14.

d. For pin mappings, reference Terminal Functions on the class webpage.

3. The internal voltage reference can also be used off-chip when the SD16VMIDON bit (Bit 3) in the SD16CTL register is set to 1.

a. Notice how this is implemented using a \textit{tri-state buffer}. A tri-state buffer can be understood as a switch (as shown in the diagram).

b. Tri state buffers are called tri-state because they have 3 possible output states.

<table>
<thead>
<tr>
<th>Control</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

c. If the control bit is zero, they output a high impedance (can be interpreted as an open circuit)

d. If the control bit is 1, they act as a normal buffer, passing the input to their output

e. SD16VMIDON is the control bit for the output tri-state buffer in the diagram

f. When using the internal reference voltage off-chip, a 470 nF capacitor connected between \( V_{\text{REF}} \) and AV\text{SS} (\( V_{\text{SS}} \) in our case) is required.

g. The buffered output provides 1 mA of drive

4. An external voltage can be provided at \( V_{\text{REF}} \) if SD16REFON and SD16VMIDON are both reset.

D. Channel Selection

1. Diagram
2. The SD16 multiplexes 8 possible differential inputs into the PGA.
3. Up to 5 input pairs, (A0 – A4) are available externally on the device.
   a. Look at the Terminal Functions document for positive and negative pin connections to A0-A4.
4. A resistive voltage divider is accessible through A5 to measure the supply voltage. This is for measuring what’s left of battery voltage in real-life applications.
5. An internal temperature sensor is available through A6.
6. A7 connects are tied to each other and can be used to calibrate the offset voltage for the differential input stage.
7. Analog inputs are configured using the SD16INCTL0 (SD16 Input Control Register 0) and SD16AE (SD16 Analog Input Enable Register) registers.
   a. SD16INCTL0 Register (look in Section XXXXXX in chapter)
      - Bits 2 – 0: SD16INCHx (SD16 Input Channel) bits
        - Select one of the eight differential pairs A0 – A7 of the multiplexer
      - Bits 5 – 3: SD16GAINx (SD16 Preamplifier Gain) bits
        - Selects of six possible gains for the PGA

<table>
<thead>
<tr>
<th>SD16GAINx bits</th>
<th>Resulting GAIN_{PGA}</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>x1</td>
</tr>
<tr>
<td>001</td>
<td>x2</td>
</tr>
<tr>
<td>010</td>
<td>x4</td>
</tr>
<tr>
<td>011</td>
<td>x8</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>100</td>
<td>x16</td>
</tr>
<tr>
<td>101</td>
<td>x32</td>
</tr>
<tr>
<td>110</td>
<td>Reserved</td>
</tr>
<tr>
<td>111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

b. SD16AEx Register (look in Section XXXXXX in chapter)
   - Setting an SD16AEx bit = 1 enables the respective external input. You must enable any input pin that you want to feed data.
   - Setting an SD16AEx bit = 0 disables the respective external input pin and connects the negative differential pin of the input to AVSS.

8. An external RC low-pass filter is recommended for the SD16_A to prevent aliasing of the input signal.
   - The cutoff frequency should be < 10kHz for a 1 MHz modulator clock and OSR (oversampling rate) = 256.

E. Analog Input Characteristics
1. The SD16_A2 uses a switched-capacitor input stage to sample the input waveforms that looks like:

![MSP430 schematic](image)

\[ V_{S+} \text{ Positive external source voltage} \]
\[ V_{S-} \text{ Negative external source voltage} \]
\[ R_S \text{ External source resistance} \]
\[ C_S \text{ Sampling capacitance} \]
\[ AVCC/2 \text{ Average of positive and negative supplies} \]
\[ \text{Not implemented in MSP430x20x3 devices} \]

2. If the buffers are used, R_S doesn’t affect the sampling frequency. If you look at the note, the F2013 does not implement these buffers.
   a. **QUESTION:** If \( f_s \) is set by \( fm/OSR \), what happens if you don’t select a gain and/or reference voltage and/or \( R_S \) that matches \( fm/OSR \)?
   b. The maximum sampling frequency is calculated using the equation:

\[
   f_s \leq (R_S + 1k\Omega) \times C_S \times \ln \left( \frac{V_{REF}}{GAIN \times 2^{17} \times V_{Ax}} \right)
\]

c. Where the sampling frequency, \( f_s \), and \( V_{Ax} \) are calculated as:
\[ f_s = \frac{1}{2 \times t_s} \quad \text{and} \quad V_{At} = \left| \frac{AV_{CC}}{2} - V_s \right| \]

- AV_{CC} is the Analog V_{CC}

d. The sampling capacitance, C_s, is dependent on the GAIN of the PGA

<table>
<thead>
<tr>
<th>PGA Gain</th>
<th>Sampling Capacitance C_s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.25 pF</td>
</tr>
<tr>
<td>2, 4</td>
<td>2.5 pF</td>
</tr>
<tr>
<td>8</td>
<td>5 pF</td>
</tr>
<tr>
<td>16, 32</td>
<td>10 pF</td>
</tr>
</tbody>
</table>

F. The SD16MEM0 register (Sigma Delta 16 Conversion Memory Register)

1. Collecting the Data
   a. Conversion results are moved to the SD16MEM0 register with each decimation step of the digital filter
   b. The SD16IFG flag is set when new data is written into the SD16MEM0 register

2. According to the SD16SNGL (Sigma Delta Single Conversion Bit) the SD_16 channel can be converted once, or it can be converted continuously

<table>
<thead>
<tr>
<th>SD16SNGL</th>
<th>Mode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Single conversion</td>
<td>The channel is converted once.</td>
</tr>
<tr>
<td>0</td>
<td>Continuous conversion</td>
<td>The channel is converted continuously.</td>
</tr>
</tbody>
</table>

a. The SD16SC bit (SD_16 Start Conversion Bit) — Bit 1 in the SD16CCTL0 register
   - If SD16SNGL = 0 (Continuous Conversion Mode), conversions will begin when the SD16SC bit is set and continue until it is reset to zero by software
   - If SD16SNGL = 1 (Single Conversion Mode), a single conversion occurs and is written to the SD16MEM0 register when the SD16SC bit is set and the SD16IFG is set. The SD16SC bit is the automatically cleared by the hardware.

b. Output Data Format
Data can be output to the SD16MEM0 register in one of three formats:
- Bipolar Offset Binary
- Bipolar Two’s Compliment
- Unipolar

Table 17-3. Data Format

<table>
<thead>
<tr>
<th>SD16UNI</th>
<th>SD16DF</th>
<th>Format</th>
<th>Analog input</th>
<th>SD16MEM0↑</th>
<th>Digital Filter Output (OSR = 256)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Bipolar Offset Binary</td>
<td>ZERO</td>
<td>8000</td>
<td>0000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-FSR</td>
<td>0000</td>
<td>0000000000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Bipolar Two’s compliment</td>
<td>ZERO</td>
<td>0000</td>
<td>0000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-FSR</td>
<td>0000</td>
<td>0000000000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Unipolar</td>
<td>ZERO</td>
<td>8000</td>
<td>0000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-FSR</td>
<td>0000</td>
<td>0000000000</td>
</tr>
</tbody>
</table>

↑ Independent of SD16OSR and SD16OSR settings; SD16OSRA[6:0] = 0.

Figure 17–6 shows the relationship between the full-scale input voltage range from \(-V_{FSR}\) to \(+V_{FSR}\) and the conversion result. The data formats are illustrated.

- The output mode is selected using the SD16UNI and SD16DF in the SD16CTL0 register.
  - Bipolar or Unipolar output mode is specified by the SD16UNI bit
    
    | SD16UNI | Bit 12 | Unipolar mode select |
    |---------|--------|-----------------------|
    | 0       | Bipolar mode |
    | 1       | Unipolar mode |

  - Offset binary or two’s complement mode is selected using the SD16DF bit
    
    | SD16DF | Bit 4 | SD16_A data format |
    |--------|-------|---------------------|
    | 0      | Offset binary |
    | 1      | 2’s complement |

G. The Digital Filter
1. The digital filter used to process the 1-bit stream from the modulator is a SINC³ filter with the transfer function:

\[ H(z) = \left( \frac{1}{OSR} \times \frac{1 - z^{-OSR}}{1 - z^{-1}} \right)^3 \]

2. In the frequency domain, this translates to

\[ H(f) = \left[ \frac{\sin(OSR \frac{f}{f_M})}{\sin(\pi \frac{f}{f_M})} \right]^3 = \left( \frac{1}{OSR} \times \frac{\sin(OSR \pi \frac{f}{f_M})}{\sin(\pi \frac{f}{f_M})} \right)^3 \]

   a. The oversampling rate (OSR) is equal to the ratio of \( f_M / f_s \) and **the OSRx bits**?

3. The graph in the frequency domain

4. The filter outputs new conversion results to the SD16MEM0 register at the sample frequency \( f_s \)

H. The Digital Filter Output

1. The number of bits output by the digital filter is dependent on the oversampling ratio and ranges from 15 to 30 bits. (user’s guide).

2. The actual bits stored in the SD16MEM0 register are dependent on the LSBACC bit (**Least Significant Bit Access**) and the SD16UNI bit (**SD16 Unipolar Mode Select**)

   a. If the LSBACC = 1, the 16 least significant bits are written to the SD16MEM0 register

   b. Setting the LSBTOG bit automatically toggles the LSBACC bit every read, allowing the full filter output to be read every two SD16MEM0 reads.
I. Interrupt Handling

1. There are two interrupt sources available for the SD16_A
   a. The SD16IFG flag (SD16 Interrupt Flag)
      - Set when the SD16MEM0 register is written to with a conversion result.
      - The interrupt is generated if the corresponding SD16IE bit and the GIE bit are set
      - Cleared according to operating mode
b. The SD16OVIFG flag (*SD16 Overflow Interrupt Flag*)
   - Set if a conversion result is written into the SD16MEM0 register
     before the previous conversion has been read.
   - Cleared only by software

2. The SD16IV (*SD16 Interrupt Vector*)
   a. All interrupt sources for the SD16 are prioritized and handled in a
      single interrupt vector.
   b. Like the timer module, a number is generated in the SD16IV register
      that can be evaluated or added to the program counter to enter the
      respective software routine.

V. Example Code

```
………………………………………………………………………………………………………
; MSP430F20x3 Demo - SD16A, Sample A1+ Continuously, Set P1.0 if > 0.3V
; Description: A continuous single-ended sample is made on A1+ using internal
; VREF. Unipolar output format used.
; Inside of SD16 ISR, if A1 > 1/2VRef (0.3V), P1.0 set, else reset.
; ACLK = n/a, MCLK = SMCLK = SD16CLK = default DCO
;
; M. Buccini / L. Westlund
; Texas Instruments Inc.
; October 2005
; Built with IAR Embedded Workbench Version: 3.40A
………………………………………………………………………………………………………

#include "msp430x20x3.h"

; ORG 0F800h ; Program Reset

RESET  mov.w #0280h,SP            ; Initialize stack pointer
StopWDT mov.w #WDTPW+WDTHOLD,&WDTCTL ; Stop WDT
SetupP1  bis.b #001h,&P1DIR    ; P1.0 output
SetupSD16 mov.w #SD16REFON+SD16SSSEL_1,&SD16CTL ; 1.2V ref, SMCLK
    mov.b #SD16INCH_1,&SD16INCTL0 ; A1+/-
    mov.w #SD16UNI+SD16IE,&SD16CCTL0 ; 256OSR, unipolar, inter
    mov.b #SD16AE2,&SD16AE ; P1.1 A1+, A1- = VSS
    bis.w #SD16SC,&SD16CCTL0 ; Start conversion

Mainloop  bis.w #CPUOFF+GIE,SR ; CPU off, enable interrupts
          nop ; Required only for debugger

SD16_ISR;
………………………………………………………………………………………………………
    bic.b #01h,&P1OUT ;
    cmp.w #07FFH,&SD16MEM0 ; SD16MEM0 > 0.3V?, clears IFG
    jlo Done ;
    bis.b #01h,&P1OUT ;
Done  reti ;
………………………………………………………………………………………………………
; Interrupt Vectors
………………………………………………………………………………………………………
```
ORG 0FFFEh ; MSP430 RESET Vector
DW RESET ;
ORG 0FFEAh ; SD16 Vector
DW SD16_ISR ;
END
VI. The SD16_A registers
   A. SD16_A Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Short Form</th>
<th>Register Type</th>
<th>Address</th>
<th>Initial State</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD16_A Control</td>
<td>SD16CTL</td>
<td>Read/write</td>
<td>0100h</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td>SD16_A Interrupt Vector</td>
<td>SD16IV</td>
<td>Read/write</td>
<td>0110h</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td>SD16_A Channel 0 Control</td>
<td>SD16CCTL0</td>
<td>Read/write</td>
<td>0102h</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td>SD16_A Conversion Memory</td>
<td>SD16MEM0</td>
<td>Read/write</td>
<td>0112h</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td>SD16_A Input Control</td>
<td>SD16INCTL0</td>
<td>Read/write</td>
<td>0B0h</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td>SD16_A Analog Enable</td>
<td>SD16AE</td>
<td>Read/write</td>
<td>0B7h</td>
<td>Reset with PUC</td>
</tr>
</tbody>
</table>

B. SD16CTL (SD16_A Control Register)

```
15 14 13 12 11 10 9 8
   Reserved  SD16DIVx   SD16LP
   r0  r0  r0  r0  nw-0  nw-0  nw-0  nw-0

   7 6 5 4 3 2 1 0
SD16DIVx  SD16SSSELx  SD16_VMIDON  SD16_REFON  SD16_OVIE  Reserved
rw-0  rw-0  rw-0  rw-0  rw-0  rw-0  rw-0  rw-0  r0

Reserved Bits Reserved 15-12
SD16DIVx Bits SD16_A clock divider
11-9 000 /1
000 /3
000 /16
000 /48
1xx Reserved
SD16LP Bit 8 Low power mode. This bit selects a reduced speed, reduced power mode
0 Low-power mode is disabled
1 Low-power mode is enabled. The maximum clock frequency for the
SD16_A is reduced.
SD16DIVx Bits SD16_A clock divider
7-6 00 /1
01 /2
10 /4
11 /8
SD16SSSELx Bits SD16_A clock source select
5-4 00 MCLK
01 SMCLK
10 ACLK
11 External TACLK
SD16_VMIDON Bit 3 V_MID buffer on
0 Off
1 On
SD16_REFON Bit 2 Reference generator on
0 Reference off
1 Reference on
SD16_OVIE Bit 1 SD16_A overflow interrupt enable. The GIE bit must also be set to enable the
interrupt.
0 Overflow interrupt disabled
1 Overflow interrupt enabled
Reserved Bit 0 Reserved
```
### C. SD16CCTL0 (SD16_A Control Register 0)

SD16CCTL0, SD16_A Control Register 0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Reserved</td>
</tr>
<tr>
<td>14</td>
<td>SD16BUFx</td>
</tr>
<tr>
<td>13</td>
<td>SD16UNI</td>
</tr>
<tr>
<td>12</td>
<td>SD16XOSR</td>
</tr>
<tr>
<td>11</td>
<td>SD16NGL</td>
</tr>
<tr>
<td>10</td>
<td>SD16OSRx</td>
</tr>
<tr>
<td>9</td>
<td>SD16LSBTG</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>SD16LSBACC</td>
</tr>
<tr>
<td>4</td>
<td>SD16OVIFG</td>
</tr>
<tr>
<td>3</td>
<td>SD16OF</td>
</tr>
<tr>
<td>2</td>
<td>SD16IE</td>
</tr>
<tr>
<td>1</td>
<td>SD16IFG</td>
</tr>
<tr>
<td>0</td>
<td>SD16IO</td>
</tr>
</tbody>
</table>

- **SD16BUFx**: High impedance input buffer mode
  - 00: Buffer disabled
  - 01: Slow speed/current
  - 10: Medium speed/current
  - 11: High speed/current

- **SD16UNI**: Unipolar mode select
  - 0: Bipolar mode
  - 1: Unipolar mode

- **SD16XOSR**: Extended oversampling ratio. This bit, along with the SD16OSRx bits, select the oversampling ratio. See SD16OSRx bit description for settings.

- **SD16NGL**: Single conversion mode select
  - 0: Continuous conversion mode
  - 1: Single conversion mode

- **SD16OSRx**: Oversampling ratio
  - 9-8: When SD16XOSR = 0
    - 00: 256
    - 01: 128
    - 10: 64
    - 11: 32
  - 7: When SD16XOSR = 1
    - 00: 512
    - 01: 1024
    - 10: Reserved
    - 11: Reserved

- **SD16LSBTG**: LSB toggle. This bit, when set, causes SD16LSBACC to toggle each time the SD16MEM0 register is read.
  - 0: SD16LSBACC does not toggle with each SD16MEM0 read
  - 1: SD16LSBACC toggles with each SD16MEM0 read

*Reserved in MSP430x20x devices*
SD16 INCTL0 (SD16_A Input Control Register)

SD16 INCTL0, SD16_A Input Control Register

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD16 INTDLy</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD16 GAINx</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD16 INCHx</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SD16 INTDLy Bits 7-6 Interrupt delay generation after conversion start. These bits select the delay for the first interrupt after conversion start.
00 Fourth sample causes interrupt
01 Third sample causes interrupt
10 Second sample causes interrupt
11 First sample causes interrupt

SD16 GAINx Bits 5-3 SD16_A preamplifier gain
000 x1
001 x2
010 x4
011 x8
100 x16
101 x32
110 Reserved
111 Reserved

SD16 INCHx Bits 2-0 SD16_A channel differential pair input
000 A0
001 A1
010 A2
011 A3
100 A4
101 A5- (AVCC - AVSS) / 11
110 A6-Temperature Sensor
111 A7- Short for PGA offset measurement
E. SD16MEM0 (SD16_A Conversion Memory Register)

SD16MEM0, SD16_A Conversion Memory Register

<table>
<thead>
<tr>
<th>10</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversion Results</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
</tr>
</tbody>
</table>

Conversion Bits

Conversion Results. The SD16MEMx register holds the upper or lower 16-bits of the digital filter output, depending on the SD16LSBACC bit.

F. SD16AE (SD16_A Analog Input Enable Register)

SD16AE, SD16_A Analog Input Enable Register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD16AE7</td>
<td>SD16AE6</td>
<td>SD16AE5</td>
<td>SD16AE4</td>
<td>SD16AE3</td>
<td>SD16AE2</td>
<td>SD16AE1</td>
<td>SD16AE0</td>
</tr>
<tr>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
</tr>
</tbody>
</table>

SD16AEEx Bits

SD16_A analog enable
7-0 0 External input disabled. Negative inputs are internally connected to VSS.
    1 External input enabled.

G. SD16IV (SD16_A Interrupt Vector Register)
## SD16IV, SD16_A Interrupt Vector Register

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Bits Description
- **00h**: No interrupt pending
- **02h**: SD16MEMx overflow
- **04h**: SD16_A Interrupt
- **06h**: Reserved
- **08h**: Reserved
- **0Ah**: Reserved
- **0Ch**: Reserved
- **0 Eh**: Reserved
- **0Fh**: Reserved

### Interrupt Source and Flag

<table>
<thead>
<tr>
<th>SD16IV Contents</th>
<th>Interrupt Source</th>
<th>Interrupt Flag</th>
<th>Interrupt Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>No interrupt pending</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>02h</td>
<td>SD16MEMx overflow</td>
<td>SD16IVIRQ</td>
<td>Highest</td>
</tr>
<tr>
<td>04h</td>
<td>SD16_A Interrupt</td>
<td>SD16IVIRQ</td>
<td>-</td>
</tr>
<tr>
<td>06h</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>08h</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0Ah</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0Ch</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0 Eh</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0Fh</td>
<td>Reserved</td>
<td>-</td>
<td>Lowest</td>
</tr>
<tr>
<td>10h</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>