Modular Electronics Learning (ModEL) Project

Texas Instruments MSP430 Microcontrollers

© 2020-2022 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 7 September 2022

This is a copyrighted work, but licensed under the Creative Commons Attribution 4.0 International Public License. A copy of this license is found in the last Appendix of this document. Alternatively, you may visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons: 171 Second Street, Suite 300, San Francisco, California, 94105, USA. The terms and conditions of this license allow for free copying, distribution, and/or modification of all licensed works by the general public.
# Contents

1 Introduction 3

2 Case Tutorial 5

2.1 Assembly example: adding two numbers 6
2.2 C example: adding two numbers 9
2.3 Sketch example: adding two numbers 11
2.4 C example: pointers 13
2.5 Assembly example: subtracting two numbers 16
2.6 Assembly example: bitwise operations 18
2.7 C example: bitwise operations 20
2.8 Assembly example: rotate right instruction 22
2.9 Assembly example: alternating LED blink 24
2.10 C example: alternating LED blink 27
2.11 Sketch example: alternating LED blink 30
2.12 Assembly example: blink all Port 1 lines 32
2.13 Assembly example: pushbutton control of LED 35
2.14 C example: pushbutton control of LED 37
2.15 C example: start-stop control 41
2.16 Assembly example: pushing and popping the stack 45
2.17 Assembly example: driving Port 2 lines with Timer 48
2.18 Assembly example: interrupt triggered by pushbutton 51
2.19 C example: interrupt triggered by pushbutton 55
2.20 Assembly example: LED blink with watchdog 57
2.21 C example: pushbutton-triggered timer 61
2.22 C example: Boolean SOP expression 65
2.23 C example: sine wave signal generator 68
2.24 C example: pulse-width modulation output 71
2.25 C example: crude analog input 75
2.26 C example: analog-controlled LEDs 78
2.27 C example: UART serial text transmission 82
2.28 C example: UART serial text and number transmission 86
2.29 C example: UART transmission of analog data 91
2.30 C example: simple datalogger 100
2.31 Example: interposing MCU to a heavy load 107
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.32</td>
<td>C example: sine-wave inverter</td>
<td>114</td>
</tr>
<tr>
<td>3</td>
<td>Tutorial</td>
<td>119</td>
</tr>
<tr>
<td>3.1</td>
<td>Microcontrollers versus microprocessors</td>
<td>119</td>
</tr>
<tr>
<td>3.2</td>
<td>MSP430G2553 pin functions</td>
<td>123</td>
</tr>
<tr>
<td>3.3</td>
<td>Elementary output and input</td>
<td>125</td>
</tr>
<tr>
<td>3.4</td>
<td>Bit-level output instructions</td>
<td>131</td>
</tr>
<tr>
<td>3.5</td>
<td>Bit-level input instructions</td>
<td>134</td>
</tr>
<tr>
<td>3.6</td>
<td>MSP430G2553 architecture</td>
<td>135</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Registers</td>
<td>136</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Memory map</td>
<td>137</td>
</tr>
<tr>
<td>3.6.3</td>
<td>I/O ports</td>
<td>138</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Interrupts</td>
<td>141</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Watchdog timer</td>
<td>146</td>
</tr>
<tr>
<td>3.7</td>
<td>MSP430G2553 auxiliary functions</td>
<td>148</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Clocks</td>
<td>149</td>
</tr>
<tr>
<td>3.7.2</td>
<td>General timers</td>
<td>150</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Watchdog timer</td>
<td>150</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Analog-Digital conversion</td>
<td>150</td>
</tr>
<tr>
<td>3.7.5</td>
<td>Analog comparators</td>
<td>150</td>
</tr>
<tr>
<td>3.7.6</td>
<td>Serial data interfaces</td>
<td>150</td>
</tr>
<tr>
<td>3.8</td>
<td>CCS assembly-language programming</td>
<td>150</td>
</tr>
<tr>
<td>3.9</td>
<td>CCS C-language programming</td>
<td>153</td>
</tr>
<tr>
<td>3.10</td>
<td>Energia programming</td>
<td>158</td>
</tr>
<tr>
<td>3.11</td>
<td>Debugging tools</td>
<td>161</td>
</tr>
<tr>
<td>4</td>
<td>Derivations and Technical References</td>
<td>177</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction to assembly language programming</td>
<td>178</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Machine code to blink an LED</td>
<td>179</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Assembly code to blink an LED</td>
<td>181</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Slowing down the blinking</td>
<td>183</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Simplifying with symbols</td>
<td>186</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Using the stack</td>
<td>187</td>
</tr>
<tr>
<td>5</td>
<td>Questions</td>
<td>191</td>
</tr>
<tr>
<td>5.1</td>
<td>Conceptual reasoning</td>
<td>195</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Reading outline and reflections</td>
<td>196</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Foundational concepts</td>
<td>197</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Terminal assignments and limits</td>
<td>198</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Interrupt capabilities</td>
<td>199</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Disabling the watchdog timer</td>
<td>200</td>
</tr>
<tr>
<td>5.2</td>
<td>Quantitative reasoning</td>
<td>201</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Miscellaneous physical constants</td>
<td>202</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Introduction to spreadsheets</td>
<td>203</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Integer conversion table</td>
<td>206</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Setting up Port 1 I/O</td>
<td>206</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The MSP430 series of microcontrollers are an excellent platform for learning microcontroller technology, featuring a clean design and a multiple development tools. This module will discuss some of the basic features of this microcontroller product line with an emphasis on programming. In no way should the Tutorial contained in this module be considered a substitute for the volumes of technical literature published by Texas Instruments on their product, but rather more of a “Getting Started” or “Quick Reference” guide for the student new to the MSP430 and to microcontrollers in general.

Important concepts related to microcontrollers in general and to the MSP430 series of microcontrollers in particular include busses, ports, development software, registers, memory maps, binary and hexadecimal numeration, masks, interrupts, oscillators, clocks, timers, analog-digital conversion, comparators, and serial data communication.

Here are some good questions to ask of yourself while studying this subject:

- What purposes do microprocessors serve best, versus microcontrollers?
- How does a program get written to the internal memory of a microcontroller?
- What does the program counter do in a microprocessor or microcontroller?
- What does the stack pointer do in a microprocessor or microcontroller?
- What types of information will you find stored within the status register of a microprocessor or microcontroller?
- How do we tell the microcontroller to use an I/O pin as an input versus as an output?
- What is the sequence of operations when the device receives an interrupt signal?
- Why might we want to adjust the clock speed of a microcontroller?

A very important resource for anyone learning to program the MSP430 series of microcontrollers is the Case Tutorial chapter, showing a range of simple programming examples in multiple languages.
Like learning a spoken or written language, it is very helpful to experience practical examples of that language in action.
Chapter 2

Case Tutorial

The idea behind a Case Tutorial is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as a scientific experimenter. Hopefully you will find these cases illuminating, and a good supplement to text-based tutorials.

These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain why the circuits behave as they do?
2.1 Assembly example: adding two numbers

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. .text) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the RESET and StopWDT labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
2.1. ASSEMBLY EXAMPLE: ADDING TWO NUMBERS

Assembly code listing

```
.cdecls C,LISTER,"msp430.h"
.def  RESET
.text
.retain
.retainrefs
RESET  mov.w #__STACK_END,SP     ; Initialize stackpointer
StopWD   mov.w #WDTPW|WDTHOLD,&WDTCTL ; Stop watchdog timer

;================================= Code Begin =================================
MAIN
    mov #0x05,R4
    add #0x06,R4

;================================== Code End ==================================

.global __STACK_END
.sect .stack
.sect ".reset"
.short  RESET
```

When run, the result of this code executing is that the value 0x0B (eleven) is left in register R4. There is no way to verify that the addition has taken place, aside from using the programming software to view the microcontroller’s register values. The microcontroller has no console display, and we have not connected any devices to its pins to indicate bit states.

Note that in assembly-language programming we use semicolons to denote single-line comments: all text to the right of a semicolon are ignored by the assembler and have no impact whatsoever on the final “machine code” instructions executed by the microcontroller. Comments are present only for clarifying purposes, to any human beings reading the assembly code.
It is also possible to specify an absolute address in memory rather than use a general-purpose registers (e.g. R4), as shown here using address 0x0200:

**Assembly code listing**

```assembly
.cdecls C,LIST,"msp430.h"
.def RESET
.text
.retain
.retainrefs

RESET                  ; Initialize stackpointer
    mov.w #__STACK_END,SP

StopWDT                 ; Stop watchdog timer
    mov.w #WDTPW|WDTHOLD,&WDTCTL

;================================= Code Begin =================================
MAIN
    mov #0x05,&0x0200
    add #0x06,&0x0200

;================================== Code End ==================================

.global __STACK_END
.sect .stack
.sect ".reset"
.short RESET
```

When run, the sum will be the same (0x0B or eleven) but stored in address 0x0200 rather than in register R4. Again, to view this sum we would have to utilize software capable of reading the microcontroller’s memory contents.
2.2 C example: adding two numbers

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430.h>`) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for the microcontroller to run.
C code listing

```c
#include <msp430.h>

int x = 5;
int y = 6;
int z;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    z = x + y;
    return 0;
}
```

All characters to the right of double-slash symbols (//) are comments which are ignored by the compiler and appear only as an aid to understanding for any human reading the code.

The following “hex dump” of microcontroller memory from address\(^1\) 0x0200 to address 0x0205 taken after the program ran shows the quantities involved in this calculation:

0x0200 06 00 05 00 0B 00

As you can see, \(x\) is stored at 0x0202 with a value of five, \(y\) is stored at 0x0200 with a value of six, and the sum \(z\) is stored at 0x0204 with a value of eleven (0x0B).

\(^1\) There is nothing in the code suggesting that variables \(x\), \(y\), and \(z\) are stored in this region of the microcontroller’s memory – you would just have to know that from researching the “memory map” for this particular model of microcontroller.
2.3 Sketch example: adding two numbers

Schematic diagram

The popular Arduino microcontroller development platform has popularized a superset of the C programming language called Sketch, and a development tool called Energia allows Sketch programs to be written for and loaded into the Texas Instruments MSP430 devices. On the next page is a listing of the Sketch code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board, version 10 of Code Composer Studio (CCS), and version 1.8.10E23 of Energia.
Sketch code listing

```c
int x = 5;
int y = 6;
int z;

void setup() {}

void loop()
{
    // put your main code here, to run repeatedly:
    z = x + y;
}
```

Every Sketch program contains two important functions: `setup()` and `loop()`. All code appearing between the curly-brace symbols (`{` and `}`) in the `setup()` function is executed once when the microcontroller powers up or gets reset. All code appearing between the curly-brace symbols (`{` and `}`) in the `loop()` function is executed repeatedly with no need to insert “jump” instructions or `while` conditionals. The whole point of Sketch as a programming language is to shield those normal details from the developer who is assumed to be a rank beginner.

All characters to the right of double-slash symbols (`//`) are comments which are ignored by the compiler and appear only as an aid to understanding for any human reading the code.

Note that this program’s function is entirely empty, as there are no special bits that need to be set or cleared, or any other initial conditions that need to be configured, in order to add two numbers together.

Interestingly, the disassembled code listing for this simple program is enormous compared to the assembly and even the C versions. This is the price you pay for a simplified programming language: the compiler has to generate a lot of code to make the microcontroller do what your easy instructions command, and this often results in “code inflation”.

2.4 C example: pointers

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (**include <msp430.h>**) is a **directive** telling the compiler how to convert the C source code into object code and then executable machine code for the microcontroller to run.
CHAPTER 2. CASE TUTORIAL

C code listing

```c
#include <msp430.h>

void main(void)
{
    int x, y, z;
    int *px, *py, *pz;

    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer

    x = 5;
    y = 3;
    z = x + y;

    px = &x;
    py = &y;
    pz = &z;
    *pz = *px - *py;

    while(1) // Endless loop
    { }
}
```

This program should be compiled and run for “Debug” mode to allow viewing the variables in the “memory browser” view of the Code Composer Studio software. When running in “Debug” mode, single-stepping through all the instructions until the execution reaches the `while()` loop, the memory browser view will look like this:

![Memory Browser View](image)

After executing the `z = x + y` instruction the value contained in variable `z` will of course be eight (the sum of five and three). After executing the `*pz = *px - *py` instruction, however, `z` will be re-assigned a value of two which is the difference between five and three. Notice how the pointer variables `px`, `py`, and `pz` all contain the addresses of their respective integer variables `x`, `y`, and `z`.

An additional note here is that all optimizations were turned off for the compilation of this program. The microcontroller defaults to using registers for pointer locations, and these registers are limited in number, so standard compiler optimizations will do tricks with these to conserve their
use. In order to avoid any such trickery, I disabled all compiler optimizations, and this resulted in the three pointers being located in standard memory spaces rather than in registers.
2.5 Assembly example: subtracting two numbers

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. .text) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the RESET and StopWDT labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
Assembly code listing

```assembly
.cdecls C,LIST,"msp430.h"
def RESET
text
retain
retainrefs
RESET mov.w #__STACK_END,SP
StopWDT mov.w #WDTPW|WDTHOLD,&WDTCTL
;============================ BEGIN CODE ===============================
MAIN
    mov #0x22, R5
    sub #0x23, R5
;============================ END CODE ===============================
.globall __STACK_END
.sect .stack
.sect ".reset"
.short RESET
```

The result of this code executing is that the value 0xFF (negative one, in signed integer format) is left in register R4.
2.6 Assembly example: bitwise operations

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. \texttt{.text}) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the \texttt{RESET} and \texttt{StopWDT} labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
Assembly code listing

.cdecls C, LIST, "msp430.h"
.def RESET
.text
.retain
.retainrefs

RESET mov.w #__STACK_END, SP
StopWDT mov.w #WDTPW|WDTHOLD, &WDTCRL

;================================ BEGIN CODE ===============================
MAIN
    mov #0xA471, R4
    bis #0x00FF, R4
    mov #0xA471, R5
    bic #0x00FF, R5
    mov #0xA471, R6
    xor #0x00FF, R6
    mov #0xA471, R7
    and #0x00FF, R7

;================================ END CODE ===============================

.global __STACK_END
.sect .stack
.sect ".reset"
.short RESET

The result of this code executing is the following register values:

- Register R4 = 0xA4FF
- Register R5 = 0xA400
- Register R6 = 0xA48E
- Register R7 = 0x0071
2.7 C example: bitwise operations

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430.h>`) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for the microcontroller to run.
C code listing

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    unsigned int x,y,z;
    x = 0xA471;
    x |= 0x00FF; // Bitwise OR and assign
    y = 0xA471;
    y ^= 0x00FF; // Bitwise XOR (toggle) and assign
    z = 0xA471;
    z &= 0x00FF; // Bitwise AND and assign
}
```

This program makes extensive use of bitwise operations and assignments. The line `x |= 0x00FF;`, for example, is equivalent to the more verbose expression `x = x | 0x00FF`, setting `x` equal to the bitwise-OR function of its former value and `0x00FF`.

The result of this code executing are the following variable values:

- `x = 0xA4FF`
- `y = 0xA48E`
- `z = 0x0071`

These values are viewable when single-stepping the program in “debug” mode.
2.8 Assembly example: rotate right instruction

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. `.text`) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the `RESET` and `StopWDT` labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
Assembly code listing

```assembly
.decls C,LIST,"msp430.h"
.def RESET
.text
.retain
.retainrefs
RESET      mov.w    #__STACK_END,SP
StopWDT    mov.w    #WDTPW|WDTHOLD,&WDTCTL

;================================ BEGIN CODE ===============================
SETUP      mov      #0xAA77, R7
LOOP       rrc      R7
            jnz     LOOP
;================================ END CODE ===============================

.global __STACK_END
.sect .stack
.sect ".reset"
.short RESET

The result of this code executing is the following values in register R7:

- 0xAA77
- 0x553B
- 0xAA9D
- 0xD54E
- 0x6AA7
- ...

These values are viewable when single-stepping the program in “debug” mode.
2.9 Assembly example: alternating LED blink

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. .text) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the RESET and StopWDT labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
2.9. Assembly Example: Alternating LED Blink

Assembly code listing

```
.cdecls C,LIST,"msp430.h" ; Include device header file
.def RESET ; Export program entry-point to
            ; make it known to linker.
.text ; Assemble into program memory.
.retain ; Override ELF conditional linking
            ; and retain current section.
.retainrefs ; Retain any sections that have
            ; references to current section.

RESET   mov.w #__STACK_END,SP ; Initialize stackpointer
StopWDT  mov.w #WDTPW|WDTHOLD,&WDTCTL ; Stop watchdog timer

;.global __STACK_END ; Stack pointer definition
.sect .stack
.sect ".reset" ; MSP430 RESET Vector
.sect ".reset" ; MSP430 RESET Vector

LOOP
    mov #0x01, P1OUT ; Sets P1.0 high (Port 1 bit 0) and others low
    call #DELAY
    mov #0x40, P1OUT ; Sets P1.6 high (Port 1 bit 6) and others low
    call #DELAY
    jmp LOOP

DELAY
    mov #0xFFFF, R4 ; Loads general register R4 with delay value
DELAYLOOP
    sub #0x01, R4 ; Decrement R4 by one
    jnz DELAYLOOP ; Repeat if result is not zero
    ret
```

Note how the main loop of this program calls on the subroutine `DELAY` twice in order to provide a time period following each change of LED status. It is important that a “pound” symbol precede
the name of the subroutine so that the assembler recognizes that as the literal ("immediate") label for the subroutine, because other addressing modes are possible with the call instruction. The jmp ("jump") instruction needs no such modifier because it only operates on labels.
2.10 C example: alternating LED blink

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (#include <msp430.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for the microcontroller to run.
C code listing

```c
#include <msp430.h>

void delayloop(void);

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    P1DIR |= 0xFF; // configure all Port 1 pins as outputs

    while(1)
    {
        P1OUT = 0x01;
        delayloop();
        P1OUT = 0x40;
        delayloop();
    }
}

void delayloop()
{
    unsigned int delay;

    for(delay=0xFFFF; delay>0; --delay);
}
```

The line `P1DIR |= 0xFF;` is an example of a bitwise operation and assignment. It is equivalent to the more verbose expression `P1DIR = P1DIR | 0xFF`, setting `P1DIR` equal to the bitwise-OR function of its former value and `0xFF`.

An interesting point to note about this program is that the compiler identified the `delayloop()` function as an inefficient use of computational power and recommended the use of the MCU’s built-in timer capabilities. With standard optimization turned on (`-O2`) the compiler actually skipped the `for` loop which left the program with no time delay at all! I had to disable all optimizations in order to have the code compile and run as written. Interestingly, when run in debug mode and single-stepped, the program did exactly what it was supposed to do.
We may achieve the same effect by using a special *intrinsic* function provided by Code Composer Studio named `__delay_cycles()` . This instruction, which accepts only an *unsigned long integer constant* value as its argument and cannot work with a variable, generates a definite sequence of assembly-language instructions when compiled to produce the same effect as the `delayloop()` function we made in the previous example. If all you need is a fixed delay time, the `__delay_cycles()` instruction is a very good solution and makes for much more compact C code:

**C code listing**

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    P1DIR |= 0xFF; // configure all Port 1 pins as outputs

    while(1)
    {
        P1OUT = 0x01;
        __delay_cycles(100000);
        P1OUT = 0x40;
        __delay_cycles(100000);
    }
}
```

The actual amount of time delay, as with our `delayloop()` function, depends on the clock speed of the microcontroller.

The two `P1OUT` assignment statements control the alternating P1.0 and P1.6 states. When setting `P1OUT` equal to 0x01, what we are really doing is setting Port 1’s eight bits to the following binary states: 0b00000001, making pin P1.0 “high” and the rest low. When setting `P1OUT` equal to 0x40, what we are really doing is setting Port 1’s eight bits to the following binary states: 0b01000000, making pin P1.6 “high” and the rest low. By editing these values assigned to `P1OUT`, we may specify any two-state “blinking” sequence desired with Port 1’s eight output pins.
2.11 Sketch example: alternating LED blink

Schematic diagram

The popular Arduino microcontroller development platform has popularized a superset of the C programming language called Sketch, and a development tool called Energia allows Sketch programs to be written for and loaded into the Texas Instruments MSP430 devices. On the next page is a listing of the Sketch code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board, version 10 of Code Composer Studio (CCS), and version 1.8.10E23 of Energia.
2.11. SKETCH EXAMPLE: ALTERNATING LED BLINK

Sketch code listing

```cpp
// The setup() function executes once upon reset or startup
void setup()
{
  // Declare P1.0 (pin 2) and P1.6 (pin 14) as outputs
  pinMode(2, OUTPUT);
  pinMode(14, OUTPUT);
}

// The loop() function repeats endlessly
void loop()
{
  digitalWrite(2, LOW); // clear P1.0 (pin 2 on IC)
  digitalWrite(14, HIGH); // set P1.6 (pin 14 on IC)
  delay(500); // wait for 500 milliseconds

  digitalWrite(2, HIGH); // set P1.0 (pin 2 on IC)
  digitalWrite(14, LOW); // clear P1.6 (pin 14 on IC)
  delay(1200); // wait for 1.2 seconds
}
```

Every Sketch program contains two important functions: `setup()` and `loop()`. All code appearing between the curly-brace symbols (`{` and `}`) in the `setup()` function is executed once when the microcontroller powers up or gets reset. All code appearing between the curly-brace symbols (`{` and `}`) in the `loop()` function is executed repeatedly with no need to insert “jump” instructions or `while` conditionals. The whole point of Sketch as a programming language is to shield those normal details from the developer who is assumed to be a rank beginner.

All characters to the right of double-slash symbols (`//`) are comments which are ignored by the compiler and appear only as an aid to understanding for any human reading the code.

In an effort to simplify microcontroller programming, the Sketch language treats all I/O pins by the actual pin number on the integrated circuit. This way the developer is spared the task of having to associate physical pin numbers on the device to port and bit numbers (e.g. Pin 2 = P1.0, Pin 3 = P1.1, etc.).

Interestingly, the disassembled code listing for this simple program is enormous compared to the assembly and even the C versions. This is the price you pay for a simplified programming language: the compiler has to generate a lot of code to make the microcontroller do what your easy instructions command, and this often results in “code inflation”. 
2.12 Assembly example: blink all Port 1 lines

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. `.text`) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the RESET and StopWDT labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
2.12. ASSEMBLY EXAMPLE: BLINK ALL PORT 1 LINES

Assembly code listing

```assembly
.cdecls C,LIST,"msp430.h"
.def RESET
.text
.retain
.retainrefs
RESET	mov.w	__STACK_END,SP
StopWDT	mov.w	#WDTPW|WDTHOLD,&WDTCTL

;================================= Code Begin =================================
mov #0xFF, P1DIR ; Set all Port 1 pin directions to output (1)
mov #0xFF, P1OUT ; Initialize Port 1 pin states to high (1)
LOOP
xor #0xFF, P1OUT ; toggle all bits in P1OUT using XOR function
mov #0xFFF, R4 ; Load general register R4 with delay value
DELAYLOOP
sub #0x01, R4 ; subtract 1 from register R4 . . .
jnz DELAYLOOP ; once we reach zero exit delay loop
jmp LOOP
;================================== Code End ==================================

.global __STACK_END
.sect .stack
.sect ".reset"
.short RESET
```
The disassembled code shows the memory addresses where each instruction is held, as well as the instruction opcodes and operands (shown as hexadecimal values):

```assembly
19  RESET     mov.w  #__STACK_END,SP
$../main.asm:19:42$(), RESET():
c00:   4031 0400  MOV.W  #0x0400,SP
20  StopWDT   mov.w  #WDTPW|WDTHOLD,&WDTCTL
           StopWDT:
c004:  40B2 5A80 0120  MOV.W  #0x5a80,&Watchdog_Timer_WDTCTL
26  mov #0xFF, P1DIR ; Set all Port 1 pin directions to output (1)
c00a:  40B0 00FF 4014  MOV.W  #0x00ff,Port_1_2_P1DIR
27  mov #0xFF, P1OUT ; Initialize Port 1 pin states to high (1)
c010:  40B0 00FF 400D  MOV.W  #0x00ff,Port_1_2_P1OUT
28  mov #0xFF, R4 ; Load general register R4 with delay value
29  xor #0xFF, P1OUT ; toggle all bits in P1OUT using XOR function
30  sub #0x01, R4 ; subtract 1 from register R4 . . .
33  DELAYLOOP:
c01a:  E0B0 00FF 4003  XOR.W  #0x00ff,Port_1_2_P1OUT
33  sub #0x01, R4 ; subtract 1 from register R4 . . .
c020:  8314  DEC.W  R4
34  jnz DELAYLOOP ; once we reach zero exit delay loop
35  jmp LOOP
35  jmp LOOP
35  jmp LOOP
35  jmp LOOP
48  BIS.W  #(0x0010),SR
           $isr_trap.asm:48:59$(), __TI_ISR_TRAP():
c026:  D032 0010  BIS.W  #0x0010,SR
49  JMP __TI_ISR_TRAP
49  JMP __TI_ISR_TRAP
52  JMP __TI_ISR_TRAP
```

Note how the “move word” instructions don’t all have the same opcode even though they share the same “MOV.W” mnemonic. This is one of the complexities of machine language hidden from you when you write code in assembly language. Even though certain instructions may appear to be identical, their opcodes may differ depending on the addressing mode of each operand (e.g. immediate, symbolic, absolute).
2.13 Assembly example: pushbutton control of LED

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. .text) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the RESET and StopWDT labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
The LED is energized when the pushbutton is unpressed (open), and de-energizes when you press (close) the pushbutton. Basically, the logical state of output P1.0 mirrors the logical state of input P1.3.
2.14 C example: pushbutton control of LED

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (\#include <msp430.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for the microcontroller to run.
C code listing

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    P1DIR = 0b01000001; // P1.0 and P1.6 are outputs (1)
    // All others are inputs (0)

    while(1)
    {
        if (P1IN & 0b00001000) // Turn off P1.0 LED if P1.3 is high
            P1OUT &= 0b11111110;
        else
            P1OUT |= 0b00000001; // Turn on P1.0 LED if P1.3 is low
    }
}
```

The LED is de-energized when the pushbutton is unpressed (open) and P1.3 is high. The LED energizes when you press (close) the pushbutton and make P1.3 low. Basically, the logical state of output P1.0 is the inverse of the logical state of input P1.3.

An important detail here is the condition P1IN & 0b00001000. The “ampersand” symbol (&) is the bitwise-AND operator, and the binary value 0b00001000 is a “mask” for the bitwise operation. Each bit in the “mask” is logically AND-ed with each respective bit of P1IN to produce an eight-bit binary result. By having just one bit of the mask set to one and the rest cleared to zero, the AND operation essentially forces all bits in P1IN to be ignored except for the bit corresponding to pin P1.3. If the P1.3 bit is low, then all bits in the bitwise-AND result are low regardless of any other P1 input states, and the if condition is not met. However, if the P1.3 bit happens to be high, the bitwise-AND result will be non-zero, thus meeting the if condition.

When programming microcontrollers in the C language, these bitwise operators are very useful for selecting specific bits out of a multi-bit word. In this program the purpose is for the if/else conditional statements to pay attention only to the state of input P1.3 and to ignore all other input states.
In this next version of the same program, we use hexadecimal values instead of binary values to specify bits in the P1DIR, P1IN, and P1OUT registers. Otherwise, this version of the program is functionally identical to the first:

**C code listing**

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    P1DIR = 0x41; // P1.0 and P1.6 are outputs (1)
                   // All others are inputs (0)

    while(1)
    {
        if (P1IN & 0x08) // Turn off P1.0 LED if P1.3 is high
            P1OUT &= 0xFE;

        else
            P1OUT |= 0x01; // Turn on P1.0 LED if P1.3 is low
    }
}
```

Once again, the purpose of the if conditional is to act if input pin P1.3 is in a “high” state, while the else instructs what to do when P1.3 is “low”. The bitwise-AND’s “mask” value is now written in hexadecimal form (0x08) rather than in binary (0b00001000), but these two values are mathematically identical and so the program still pays attention solely to input P1.3 and ignores the states of all other input pins for Port 1.
Finally, we will modify the bit-specifying values in this program once more to use pre-defined constants in the `msp430g2553.h` header file\(^2\), where `BIT0` is equivalent to `0b00000001`, `BIT1` is equivalent to `0b00000010`, etc. Otherwise, this version of the program is functionally identical to the first two:

**C code listing**

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    P1DIR = BIT6 + BIT0; // P1.0 and P1.6 are outputs (1)
                        // All others are inputs (0)

    while(1)
    {
        if (P1IN & BIT3) // Turn off P1.0 LED if P1.3 is high
            P1OUT &= ~BIT0;
        else
            P1OUT |= BIT0; // Turn on P1.0 LED if P1.3 is low
    }
}
```

It should be clear just how much more legible this C code is to the human reader. No longer do you have to count bit-places to tell which Port 1 pin is being specified by a binary value, or convert hexadecimal to binary and then do the same. Note also the clever use of the complement (`~`) operator where we write to `P1OUT` to turn off the LED: when we need a mask consisting of all 1’s except for bit 0, we just invert the value of `BIT0` (binary `0b00000001`) so that it becomes `0b11111110`.

Of course, the decision to use binary, hexadecimal, bit-symbol, or even decimal values to specify bit positions is completely arbitrary and at the discretion of you, the programmer.

\(^2\)You will note that this source code listing merely includes `msp430.h` rather than the specific header file for the MSP430G2553 model of microcontroller. This is okay, as the generic `msp430.h` header includes links to all the model-specific header files, which are resolved by the CCS compiler based on the specified “target” of the project. Some of the header-specified constants differ from model to model of MSP430 microcontroller, hence the need for model-specific header files. Having a single generic `msp430.h` header file that links to the others is another programming convenience offered to you by the developers of Code Composer Studio.
2.15  C example: start-stop control

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (#include <msp430.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for the microcontroller to run.
CHAPTER 2. CASE TUTORIAL

C code listing

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    P2DIR = 0x01; // Pin P2.0 is an output (LED)
P2OUT = 0x00; // Ensure LED begins in the "off" state
P1DIR = 0x00; // All P1.x pins are inputs
P1REN = 0x18; // Enable pullup resistors for P1.4 and P1.3
P1OUT = 0x18; // Resistors are pullup, not pulldown

    while(1)
    {
        if ((P1IN & BIT4) == 0) // Stop LED if switch P1.4 pressed
            P2OUT &= ~BIT0;
        else if ((P1IN & BIT3) == 0) // Start LED if switch P1.3 pressed
            P2OUT |= BIT0;
    }
}
```

Note the use of the symbolic constants `BIT0`, `BIT3`, and `BIT4` to represent the hexadecimal values `0x01`, `0x08`, and `0x10`, respectively. The “tilde” symbol (˜) is the bitwise logical operator for inversion, which means while `BIT0` represents `0b00000001`, ˜`BIT0` must represent `0b11111110`.

These symbolic constants, as well as many more, are all defined in the header file `msp430.h` for the purpose of making the source code of your program easier to read. The microcontroller doesn’t care whether you use `BIT0` or `0b00000001` or `0x01`, as the assembler replaces all those symbols with the constant values they represent.

This form of program is useful for controlling the starting and stopping of a load such as an LED or an electric motor. In fact, the only modification we would have to make to the circuit to control a motor is to have the microcontroller drive a transistor and/or relay to switch more voltage/current to the motor than output P2.0 is capable of delivering on its own (see an example of this on the next page).

A safety feature of this program is how the “stop” command is given precedence over the “start” command. This is why an `else if` conditional is used for the “start” instruction: if both “start” and “stop” buttons are pressed, the program will execute the “stop” condition and skip any evaluation of the “start” condition. This is also why the P2OUT register is cleared immediately after enabling pin P2.0 as an output: to ensure the LED (or motor) will not begin in an energized state by chance.
The following schematic shows how this same start-stop program could be used to control an electric motor operating on a higher-voltage power supply, using both a transistor and an electromechanical relay to “interpose” between the microcontroller and motor:

Both the +12 Volt and the +3.3 Volt DC supplies must share a common ground connection, so that current from pin P2.0 driving the NPN transistor’s base terminal can find a path back to the +3.3 Volt source after exiting the transistor’s emitter terminal, but otherwise should be separate power sources. The reason for this separation is to eliminate the potential problem of electrical “noise” generated by the motor interfering with the microcontroller (which requires very “clean” DC power). Note also the presence of “decoupling” capacitors across the microcontroller’s power terminals and motor terminals, both helpful in mitigating electrical noise produced by the motor.

The commutating diode connected “backwards” in parallel with the relay’s coil provides a safe discharge path for any inductive “kickback” that may result when the transistor turns off and the stored energy in the relay’s coil inductance acts to maintain current in the same direction as before when the transistor was on. This diode will be reverse-biased and non-conducting when the transistor is turned on, but when the relay coil’s inductance generates a reverse voltage polarity when the transistor turns off this diode will forward-bias and conduct to provide that inductance a non-destructive pathway for its current. Without this commutating diode in place, we run the risk of destroying the transistor (and also the microcontroller!) when de-energizing an inductive load.

What was formerly a current-limiting resistor for the LED is now a current-limiting resistor for the transistor’s base terminal. It must be of sufficient resistance to keep the driving current less than

---

3Without this mitigation, it’s entirely possible that the electrical noise produced by the running motor will cause the microcontroller to “glitch” (i.e. reset).
than both the transistor’s base terminal current limit and the microcontroller’s output pin current limit (usually on the order of several milliAmperes — but check the datasheets to be sure!), but not so large in resistance that the transistor fails to fully turn on (saturate) for efficient operation.
2.16 Assembly example: pushing and popping the stack

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. .text) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the RESET and StopWDT labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
When run, the result of this code executing is that registers R5 and R6 become populated with the values 0x1234 and 0xABCD, respectively, then by using the two push and pop instructions the contents of those two registers becomes swapped so that R5 now contains 0xABCD and R6 now contains 0x1234. This is due to the fact that the microcontroller’s stack acts as a last-in-first-out (LIFO) register. Pushing R5 and R6’s contents onto the stack and then popping back off the stack results in the data coming off the stack in reverse order from how it was pushed on.
This next version does much the same, only with four registers rather than two:

**Assembly code listing**

```
.cdecls C,LIST,"msp430.h"
.def RESET
.text
.retain
.retainrefs

RESET      mov.w #_STACK_END,SP ; Initialize stackpointer
StopWDT    mov.w #WDTPW|WDTHOLD,&WDTCTL ; Stop watchdog timer

;================================= Code Begin =================================
MAIN
    mov #0x1234,R4
    mov #0x5678,R5
    mov #0x9ABC,R6
    mov #0xDEF0,R7
    push R4
    push R5
    push R6
    push R7
    pop R4
    pop R5
    pop R6
    pop R7

;================================== Code End ==================================

.global __STACK_END
.sect .stack
.sect ".reset"
.short RESET
```

The values written to R4, R5, R6, and R7 get sequentially pushed onto the stack, and then when popped off the stack the result is a reversal of order such that R4 now stores what R7 contained, R5 now stores with R6 contained, R6 now stores what R5 contained, and R7 now stores what R4 contained.

Stack operations are particularly useful in assembly-language programming to temporarily store the contents of special registers such as the microcontroller’s status register (SR) which is used and re-used by many different types of instructions. Between executing one status-altering instruction and another, a push SR instruction may be used to save the first status for later use, retrieved by using pop SR later on in the program.
2.17 Assembly example: driving Port 2 lines with Timer

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. .text) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the RESET and StopWDT labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
Assembly code listing

```
.decls C,LIST,"msp430.h"
.def RESET

.text
.retain
.retainrefs

RESET mov.w #__STACK_END,SP
StopWDT mov.w #WDTPW|WDTHOLD,&WDTCTL

;BEGIN CODE

SETUP
mov #0x0220, TA0CTL ; Select SMCLK as source, Continuous mode
mov #0x00, P2SEL ; Necessary to set *all* Port 2 pins as
                 ; I/O (including pins P2.6 and P2.7)
mov #0xFF, P2DIR ; Port 2 pins all outputs (1)

LOOP
mov.b &0x0170, P2OUT ; Move low byte of TA0R to Port 2 output pins
jmp LOOP

;END CODE

.global __STACK_END
.sect .stack
.sect ".reset"
.short RESET
```

The `mov #0x00, P2SEL` instruction is necessary when using Port 2 but not for Port 1 because while all of Port 1’s pins default to use as I/O, this is not true for all of Port 2’s pins. Pins P2.6 and P2.7 default to crystal oscillator connections on the MSP430G2553 microcontroller (i.e. the `P2SEL` register’s default value is 0xC0 rather than 0x00). Note that this has nothing whatsoever to do with the timer’s functionality, but it does differ from previous Case Tutorial examples where we used Port 1 I/O pins instead of Port 2.

Timers are just counters driven by a steady clock. In the MSP430 series we get to assign clock sources, counting patterns, and other details by setting bits in the Timer control registers. Here, `TA0CTL` needs a couple of its bits set to select SMCLK as the clock pulse source and “continuous” mode which specifies the count sequence to be a simple up-counter with no truncation, counting from 0x0000 to 0xFFFF and then “rolling around” back to 0x0000 again.
Timer A in the MSP430 microcontroller series is a 16-bit counter driven by a selectable clock. Each I/O port is only 8 bits wide. So, we must choose which byte of the 16-bit counter to write to the output port. In the previous example we used the lower byte; here we use the upper byte. The difference in counting speed is 256:1, the high byte counting 256 times slower than the low byte.
2.18 Assembly example: interrupt triggered by pushbutton

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g., .text) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the RESET and StopWDT labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
Assembly code listing

```
.cdecls C,LIST,"msp430.h"
.def RESET
.text
.retain
.retainrefs

RESET       mov.w  #__STACK_END,SP
StopWDT      mov.w  #WDTPW|WDTHOLD,&WDTCTL

;============================= BEGIN CODE ===============================

SETUP
    mov #0xF7, P1DIR ; P1.3 as input, all others outputs
    mov #0x08, P1OUT ; Begin with all outputs low
    mov #0x08, P1IE ; Enable interrupt on P1.3
    bis #0x08, P1IES ; Interrupt on high-to-low transition
    bic #0x08, P1IFG ; Clear any pending P1.3 interrupt flag
    bis #0x08, SR ; Set the General Interrupt Enable (GIE) bit

LOOP ; A useless loop -- does absolutely nothing!
    jmp LOOP

P1_ISR
    xor #0x01, P1OUT ; Toggle P1.0 bit
    bic #0x08, P1IFG ; Clear the P1.3 interrupt flag
    reti ; Return from ISR

;============================= END CODE ===============================
```

The LED toggles (i.e. switches states) whenever the pushbutton is pressed, because pressing the pushbutton generates a falling-edge pulse (high-to-low logical transition) which triggers the interrupt and causes the processor to jump out of its useless loop to the interrupt service routine (ISR) where the XOR logical bitwise function executes.

An alternative to using .sect ".int02" followed by .short P1_ISR is to use the single interrupt directive .intvec ".int02", P1_ISR.
The following is a “disassembly” view of the MCU’s memory for this program, showing the program code as well as the interrupt vectors:

```assembly
$../main.asm:7:29$(), RESET():
c000: 4031 0400 MOV.W #0x0400,SP
StopWDT:
c004: 40B2 5A80 0120 MOV.W #0x5a80,&Watchdog_Timer_WDTCTL
SETUP:
c00a: 40B0 00F7 4014 MOV.W #0x00f7,Port_1_2_P1DIR
   c010: 42B0 400F MOV.W #8,Port_1_2_P1OUT
   c014: 42B0 400F MOV.W #8,Port_1_2_P1IE
   c018: D2B0 400A BIS.W #8,Port_1_2_P1IES
   c01c: C2B0 4005 BIC.W #8,Port_1_2_P1IFG
   c020: D232 EINT
LOOP:
c022: 3FFF JMP (LOOP)
P1_ISR:
c024: E390 3FFB XOR.W #1,Port_1_2_P1OUT
   c028: C2B0 3FF9 BIC.W #8,Port_1_2_P1IFG
   c02c: 1300 RETI
   $isr_trap.asm:48:59$(), _TI_ISR_TRAP():
c02e: D032 0010 BIS.W #0x0010,SR
   c032: 3FFD JMP ($isr_trap.asm:48:59$)
c034: 4303 NOP
   * ffda: FFFF FFFF AND.B @R15+,0xffff(R15)
      ffde: FFFF C02E AND.B @R15+,0xc02e(R15)
      ffe2: FFFF C024 AND.B @R15+,0xc024(R15)
      ffe6: C02E BIC.W @PC,R14
      ffe8: FFFF C02E AND.B @R15+,0xc02e(R15)
      ffec: C02E BIC.W @PC,R14
      ffee: C02E BIC.W @PC,R14
      fff0: C02E BIC.W @PC,R14
      fff2: C02E BIC.W @PC,R14
      fff4: C02E BIC.W @PC,R14
      fff6: C02E BIC.W @PC,R14
      fff8: C02E BIC.W @PC,R14
      ffa: C02E BIC.W @PC,R14
      fffc: C000 BIC.W @PC,R14
fffe: C000 BIC.W @PC,R14
```

Note: the AND.B and BIC.W instructions shown at the end are meaningless. Disassembly is a process by which all memory content is interpreted as instructions. However, from 0xffda to 0xffff there is no program code, only interrupt vectors (i.e. memory addresses to “jump” to in the event of a detected interrupt). The address 0xC024 is for the P1_ISR routine and the address 0xC02E is for the RESET routine. The disassembler mistakenly interpreted all 0xFFFF content as AND.B instructions and all reset vectors as BIC.W instructions.
This is a memory map showing the same in unannotated “hex dump” format:

0xC000 31 40 00 04 B2 40 80 5A 20 01 B0 40 F7 00 14 40
0xC010 B0 42 0F 40 B0 42 0F 40 B0 D2 0A 40 B0 C2 05 40
0xC020 32 D2 FF 3F 90 E3 FB 3F B0 C2 F9 3F 00 13 32 D0
0xC030 10 00 FD 3F 03 43 FF FF FF FF FF FF FF FF FF
* 0xFFE0 2E C0 FF FF 24 C0 2E C0 FF FF 2E C0 2E C0 2E C0 2E C0
0xFFF0 2E C0 2E C0 2E C0 2E C0 2E C0 2E C0 2E C0 2E C0

Note how all specified addresses are stored in little-endian format (e.g. the interrupt vector 0xC02E stored at 0xFFE0 actual shows as being 2E C0). The interrupt vector for the pushbutton service routine (P1_ISR, vector 0xC024) is seen starting at address 0xFFE4, as 24 C0.
On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (#include <msp430.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
C code listing

```c
#include "msp430.h"

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Disable the Watchdog Timer

    /* Configure Port 1 pins for LEDs and pushbutton */
    P1DIR = 0x41; // Set P1.0 and P1.6 as outputs, all other pins inputs
    P1IE = 0x08; // Enable P1.3 interrupt
    P1IES = 0x08; // P1.3 interrupt on falling edge (high-to-low)
    P1IFG = 0x00; // Clear all Port 1 interrupt flags

    __bis_SR_register(GIE); // General Interrupts Enabled

    while(1) {} // Useless while() loop
}

#pragma vector=PORT1_VECTOR
__interrupt void Timer_ISR (void)
{
    P1OUT ^= 0x01; // Toggles output P1.0
    P1IFG = 0x00; // Clears all Port 1 interrupt flag bits
}
```

The LED toggles (i.e. switches states) whenever the pushbutton is pressed, because pressing the pushbutton generates a falling-edge pulse (high-to-low logical transition) which triggers the interrupt and causes the processor to jump out of its useless loop to the interrupt service routine (ISR) where the XOR logical bitwise function executes.
2.20 Assembly example: LED blink with watchdog

Schematic diagram

On the next page is a listing of the entire assembly-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS). Only the code found between the “Code Begin” and “Code End” comment lines are specific to this example program’s purpose, with the rest being assembler directives and other instructions necessary for any simple assembly-language program to run on this microcontroller.

All statements beginning with a dot (e.g. .text) are directives telling the assembler how to convert the assembly source code into executable machine code for the microcontroller to run. Similarly, the RESET and StopWDT labels mark general set-up instructions for the microcontroller and are not specifically related to the program’s main function.
Rather than “hold” the watchdog timer near the start of the program as is customary for so many other example programs, here we include a `mov` instruction within the time delay loop that sets the `WDTCNTCL` bit to clear the watchdog timer’s 16-bit counter to zero with every pass through the loop. This way, if anything were to (somehow) halt the normal execution of our program, the watchdog timer would restart the microcontroller and the program would begin afresh.
If we modify the location of this watchdog-clear instruction to be outside of the time delay loop, we find that it fails:

**Assembly code listing**

```assembly
cdecls C,LIST,"msp430.h"
def RESET
text retain retainrefs
RESET mov.w #__STACK_END,SP ; Initialize stack pointer

;================================= Code Begin =================================
mov #0xFF, P1DIR ; Sets all Port 1 pin directions to output (1)
LOOP
mov #0x01, P1OUT ; Sets P1.0 high (Port 1 bit 0) and others low
call #DELAY
mov #0x40, P1OUT ; Sets P1.6 high (Port 1 bit 6) and others low
call #DELAY
jmp LOOP

DELAY
mov #0xFFFF, R4 ; Loads general register R4 with delay value
mov.w #WDTPW | WDTCNTCL, &WDTCTL ; Clear watchdog timer
DELAYLOOP
sub #0x01, R4 ; Decrement R4 by one
jnz DELAYLOOP ; Repeat if result is not zero
ret
;================================== Code End ==================================

.global __STACK_END ; Stack pointer definition
.sect .stack
.sect ".reset" ; MSP430 RESET Vector
.short RESET
```

The reason this program fails is because `DELAYLOOP` takes too long to go through its 0xFFFF-to-0x0000 counting cycle, and the watchdog timer “times out” before the watchdog-clear instruction gets executed. The result is the P1.0 LED remains energized all the time because that output bit gets set with the first instruction following `LOOP`, then the watchdog timer resets the microcontroller before the `DELAY` routine completes, causing the program to start again from the top.

A good experiment would be to alter the numerical value moved into register R4 until a number
is found that results in a short enough **DELAYLOOP** cycle time to permit the watchdog timer to be cleared at a reasonable interval.
2.21 C example: pushbutton-triggered timer

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (\#include <msp430.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
C code listing

```c
#include "msp430.h"
unsigned int tenths;

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Disable the Watchdog Timer

    // Sets DCO oscillator frequency to 100 kHz
    // Note: DCO bits are "fine" adjust, RSEL are "coarse"
    DCOCTL = 0x40; // DCO frequency range (DCO = 2)
    BCSCTL1 = 0x00; // Basic clock frequency (RSEL = 0)

    // Configure Port 1 pins for LED and pushbutton
    P1DIR = 0x01; // Set P1.0 as output, all other pins inputs

    // Configure Timer A
    TA0CTL = TASSEL_2 + MC_1 + TAIE; // SMCLK clock, up mode, enable interrupt
    TACCR0 = 10000; // Set maximum count value to generate an interrupt every
                    // 10000 cycles of the clock which is every 0.1 seconds

    __bis_SR_register(GIE); // General Interrupts Enabled

    while(1)
    {
        if (tenths >= 50) // Turn off P1.0 LED after 50 tenths of a second
            P1OUT &= 0xFE;
        else
            P1OUT |= 0x01;

        if ((P1IN & 0x08) == 0) // Reset timer when P1.3 pressed
            tenths = 0;
    }
}
```

#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_ISR (void) // Timer A interrupt service routine
{
    ++tenths; // Increments "tenths" variable every 0.1 seconds
    TA0CTL &= ~TAIFG; // Clears the Timer A interrupt flag
}
The digitally-controlled oscillator (DCO) for the microcontroller is set to a frequency of 100 kHz\(^4\), and the timer is configured for “up” counting mode where it counts from zero to whatever value is stored in register TACCR0 and then resets. With TACCR0 = 10000 and a clock frequency of 100000 Hz this means the timer will reset itself ten times per second\(^5\), or once every 0.1 seconds. At every reset the timer generates an interrupt, and inside the interrupt service routine we increment an integer variable named tenths as well as clear the interrupt flag before returning execution to the main code’s while() loop. Thus, the variable tenths increments once every tenth of a second, making it a convenient variable to use for time delays with a resolution of 0.1 seconds.

Code inside the while() loop writes to and reads from this tenths variable, turning the P1.0 LED either on or off based on the value stored in tenths. In this program example, tenths is allowed to increment whenever pushbutton P1.3 is released, but clamped to a value of zero when the pushbutton is pressed. If tenths is less than 50, the LED turns on. Therefore, pressing the pushbutton guarantees the LED will energize. When released, tenths is allowed to increment, and when it reaches a count value of 50 the LED turns off. Thus, we have created a classic off-delay timing function that immediately energizes the output when the switch is actuated but delays turning off until 5 seconds after the switch returns to its resting (“normal”) state:

![Diagram of pushbutton and LED behavior](image)

\(^4\)This is not a precision clock source, and so with the DCOCTL and BCSCTL1 parameters shown in the example code your actual clock frequency may vary slightly from the expected value of 100 kHz. Altering the DCOCTL register’s value allows you to “fine-tune” the oscillator to achieve close to 100 kHz. There are seven settings for the DCO frequency range, set by the most-significant three bits of the DCOCTL register.

\(^5\)If adjusting the DCOCTL and/or BCSCTL1 register values still doesn’t get the clock frequency close enough to 100 kHz for your purposes, you may also skew the value stored in register TACCR0 until the interrupt period is close enough to 10 times per second. For example, let’s say that despite your best efforts to get the DCO clock to be exactly 100 kHz, it ends up being just a little bit faster than 100 kHz. To compensate for this, you could increase the value of TACCR0 beyond 10000 so that additional clock cycles were necessary between interrupts, which will have the effect of slowing down the increment of tenths. Given that 10000 is a fairly large number, this gives you very fine-resolution adjustability for the interrupt timing.
CHAPTER 2. CASE TUTORIAL

Obtaining other forms of timing function only requires edits to code inside the `while()` loop. No changes need to be made to the timer setup or the interrupt service routine, as they still work to generate an incrementing `tenths` variable every tenth of a second. For example, here is a code snippet for a classic on-delay timing function with a delay time of 5 seconds:

```c
while(1)
{
    if (tenths >= 50) // Turn on P1.0 LED after 50 tenths of a second
        P1OUT |= 0x01;

    else
        P1OUT &= 0xFE;

    if ((P1IN & 0x08) == 0x08) // Reset timer when P1.3 released
        tenths = 0;
}
```

For applications where the `tenths` variable would be free to increment for long periods of time, we must contend with a problem posed by our original code, namely what happens when `tenths` reaches its maximum 16-bit value of 65535? Any unsigned integer that reaches its maximum positive value and is incremented one more time will “recycle” back to zero. If our timing function is based on `tenths` being greater than some limit, this will mean that condition will reset when `tenths` recycles back to zero again.

A fix for this problem is to limit how high `tenths` is allowed to count, and to do so within the interrupt service routine where we may block its incrementing before counting too far. We just need one more line of code in the interrupt service routine, taking the form of an `if()` statement:

```c
#pragma vector=TIMERO_A1_VECTOR
__interrupt void Timer_ISR (void) // Timer A interrupt service routine
{
    if (tenths < 0xFFFF)
        ++tenths; // Increments if less than maximum positive value
    TA0CTL &= ~TAIFG; // Clears the Timer A interrupt flag
}
```
2.22 C example: Boolean SOP expression

Boolean expression implemented by MCU

\[ \text{Out} = \overline{C} \overline{B} \overline{A} + \overline{C} \overline{B} A + C B A \]

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (\#include <msp430.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for the microcontroller to run.
The basic objective of this program is to turn on the LED connected to P1.0 only if certain switch-state conditions are met. The Boolean expression tells us what these combinations of switch on/off states are:

- **Turn on LED if C is high (switch open) and B is low (switch closed) and A is high (switch open), or**
- **Turn on LED if C is low (switch closed) and B is low (switch closed) and A is high (switch open), or**
- **Turn on LED if C is high (switch open) and B is high (switch open) and A is high (switch open)**
2.22. C EXAMPLE: BOOLEAN SOP EXPRESSION

These discrete inputs represent three bits within the eight-bit “word” associated with the microcontroller’s Port 1. Showing the bits as C, B, and A with all other bits represented as “x”:

xxxxCBAx

In other words, switch C connects to input pin P1.3, switch B to input P1.2, and switch A to P1.1. All higher-order bits are unused (and therefore may be in random states at any time), and of course P1.0 is for the output to the LED.

The if and else if conditional statements in this program check to see if any of those three combinations are met, to turn the LED on. However, a significant challenge here is that we must tell the microcontroller to ignore all the other bits in Port 1 when checking these three, because these conditional statements are all looking for exact matches. For example, if we simply wrote the first conditional instruction as if (P1IN == 0x0A) to check for Port 1’s state being 0b00001010 (C and A high, B low), the program would “think” we were not meeting the required condition if any of the other bits in the P1IN word happened to be 1 instead of 0. This would make the program subject to random conditions on the other input pins and therefore unreliable.

Our solution to this problem is to use a bitwise-AND operation to “mask off” the other five bits in the P1IN word so that we may use the if statement to test just the three bit-states associated with our pushbutton switches. The concept of a “bitwise” operation is that a binary word having the same number of bits is compared, bit-by-bit, with the digital word we’re interested in probing. In the case of a bitwise-AND operation, each bit of a “mask” word is AND-ed with each respective bit in the word of interest to produce a result that also has the same number of bits. This is the meaning of (P1IN & 0x0E) in each of the three conditional statements: the “mask” word is 0x0E which is 00001110 in binary form. When this “mask” is bitwise-ANDed with P1IN, it forces every bit except for P1.3, P1.2, and P1.1 to a zero state, leaving only our C, B, and A switches able to show their states for inspection by the if statement. Then and only then may we safely compare against the desired bit-combinations to tell whether or not we should turn the LED on.

If any of these three conditions are met, we set P1.0 to a high state using another bitwise operator, this time the bitwise-OR and assignment operator. The purpose of this is to set only P1.0 to a high state if desired without affecting the states of any of the other seven bits in the Port 1 word. Conversely, if none of the three (if/else if) conditions are met, the else conditional is true which lets us clear P1.0 to a low state using a bitwise-AND and assignment operator. The bitwise AND/assign operator makes sure that P1.0 is the only bit cleared in the P1OUT word, leaving all other bits in that word unaltered.
2.23 C example: sine wave signal generator

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430G2553.h>`) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run. The code presented on the next page was derived from code written by Jason Wahl on 29 January 2021.
2.23. C EXAMPLE: SINE WAVE SIGNAL GENERATOR

C code listing

```c
#include "msp430G2553.h"

unsigned char wave[127] = // Sine table ranging 0 to 255
{127,134,140,146,153,159,165,171,177,182,
  188,194,199,204,209,214,218,223,227,230,
  234,237,240,243,246,248,250,251,253,254,
  255,255,255,255,254,253,252,251,249,247,
  245,242,239,236,232,229,225,220,216,211,
  206,201,196,191,185,180,174,168,162,156,
  149,143,137,131,124,118,112,106,99,93,
  87,81,75,70,64,59,54,49,44,39,
  35,30,26,23,19,16,13,10,8,6,
  4,3,2,1,0,0,0,0,1,2,
  4,5,7,9,12,15,18,21,25,28,
  32,37,41,46,51,56,61,67,73,78,
  84,90,96,102,109,115,121};

void main(void)
{
    unsigned int i; // Used in for() loop

    WDTCTL = WDTPW + WDTHOLD; // Stop WDT
    DCOCTL = 0xE0; // Maximum DCO frequency range (DCO = 7)
    BCSCTL1 = 0x0F; // Maximum basic clock frequency (RSEL = 15)
    BCSCTL2 = 0x00; // Sets DCOCLK as the source for MCLK and SMCLK
    P2SEL = 0x00; // Sets P2.6 + P2.7 to standard output mode
    P2DIR = 0xFF; // All P2.x pins are outputs

    while(1)
    {
        for (i = 0; i < 127; ++i)
            P2OUT = wave[i];
    }
}
```
The model MSP430G2553 microcontroller lacks a built-in digital-to-analog converter (DAC), and so here we build our own based on a $R-2R$ “ladder” network. This program counts through a series of numerical values ranging from 0 to 255, chosen to emulate the evolution of a sine wave with a center value of 127 peaking at 255 (positive) and 0 (negative). When output to P2OUT this numerical value drives the eight output pins such that they activate the eight terminals of the resistor “ladder” network. Each numerical value represents a different amplitude along the course of the sine wave, and the operational amplifier produces an analog voltage output in response.

A square-wave signal may be obtained on the most significant bit (pin P2.7) with the same frequency as the sine wave, which for this code and this model of microcontroller is approximately 7.4 kHz.
2.24  C example: pulse-width modulation output

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430G2553.h>`) is a `directive` telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
include "msp430G2553.h"

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Disable the Watchdog Timer
    // Set Digitally Controlled Oscillator to 1 MHz
    BCSCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;

    // Configure P1.3 pin
    P1DIR &= ~BIT3; // Set P1.3 as input (for pushbutton)

    // Configure P1.6 pin (Note: BIT6 is a symbol for the constant 0b01000000
    // When P1DIR=1 and P1SEL=1 and P1SEL2=0 for timer output TA0.1 on pin 6
    P1DIR |= BIT6; // Sets bit 6 while leaving the other bits untouched
    P1SEL |= BIT6; // Sets bit 6 while leaving the other bits untouched
    P1SEL2 &= ~BIT6; // Clears bit 6 while leaving the other bits untouched

    // Configure timer A for pulse-width modulation
    TA0CTL = TASSEL_2 + MC_1 + ID_0;
    // TASSEL_2 = SMCLK as input clock,
    // MC_1 = Mode Control 1
    // (counts "up" to TA0CCR0 then resets),
    // ID_0 = No clock frequency division
    TA0CCR0 = 10000; // Maximum count value determines frequency =
    // SMCLK/ClockDivide/TACCRO (1 MHz/1/10000 = 100 Hz)
    TA0CCR1 = 100; // Initialize the duty cycle at 1.00% = TACC1/TACC0
    TAOCCTL1 = OUTMOD_7; // TA0.1 output high when counter < TA0CCR1

    while(1)
    {
        if ((P1IN & BIT3) == 0x00)
            TA0CCR1 = 2500; // 25.00% duty cycle when P1.3 pushbutton pressed
        else
            TA0CCR1 = 7500; // 75.00% duty cycle when P1.3 pushbutton released
    }
}
This program uses Timer A to generate a precise pulse-width modulation output signal to energize the LED connected to pin P1.6, the signal having a frequency of 100 Hz and a duty cycle set for either 25% or 75% depending on whether or not the pushbutton connected to pin P1.3 is pressed.

Noteworthy features of this program include:

- Extensive use of symbolic constants to represent what would otherwise be arcane bit patterns; e.g. \texttt{BIT6} is a symbol for the binary constant \texttt{0b01000000} (hex \texttt{0x40})
- Use of the timer circuit’s functionality to replace what would otherwise be code running within the \texttt{while()} loop to generate the on/off times of the PWM pulse
- Explicit setting of the clock frequency in order to achieve a precise PWM signal frequency

Let’s examine this code section by section.

The two oscillator-control words \texttt{BCSCTL1} and \texttt{DCOCTL} are initialized to configure the system’s clock oscillator for a frequency of 1 MHz.

Pins P1.3 and P1.6 need to be configured as input and output, respectively. However, pin P1.6 will not be used as a general-purpose output where its value is controlled by register \texttt{P1OUT} as you might normally do. Instead, we will use a special feature of the MSP430G2553 microcontroller which assigns that pin to the TA0.1 output bit of Timer A. Since the timer is its own circuit which runs independently of the MCU’s central processing unit (CPU), this allows the timer to drive pin P1.6 all on its own. The only code needed to control the duty cycle of this pulsing will be whatever we write within the \texttt{while()} loop to set the duty cycle percentage — once set, the timer does all the rest! In order to select this special pin assignment, we need to set bit 6 of \texttt{P1SEL} (high) and clear bit 6 of \texttt{P1SEL2} (low). Such special functions vary from one model of MSP430 microcontroller to another, details of which may be found in the datasheet for your specific microcontroller.

Timer A is a sixteen-bit counter capable of counting from \texttt{0x0000} to \texttt{0xFFFF}, but offers a great deal of functionality beyond that simple count sequence. This functionality is controlled by a set of registers, and so the largest section of code within this program is devoted to the initialization of those registers. This includes selecting the timer’s clock source and frequency division ratio, and the Mode Control (in this case we’re telling the timer to truncate its count to be limited to a high value we place within register \texttt{TA0CCR0}. Furthermore, the Output mode is set to a value of seven, representing a mode suitable for pulse-width modulation, where output bit TA0.1 will be set any time the count value is less than that of register \texttt{TA0CCR1} and cleared any other time. As Timer A’s count value increments from zero to \texttt{TA0CCR0} and repeats, output bit TA0.1 alternates between high and low depending on the value of \texttt{TA0CCR1}. This causes TA0.1 to produce a pulse-width modulated signal, with a duty cycle equal to the ratio between \texttt{TA0CCR1} and \texttt{TA0CCR0}. With the maximum count (\texttt{TA0CCR0}) set at 10000, the duty cycle will be 0.00% at \texttt{TA0CCR1 = 0}, 25.00% at \texttt{TA0CCR1 = 2500}, 50.00% at \texttt{TA0CCR1 = 5000}, 75% at \texttt{TA0CCR1 = 7500}, and 100.00% at \texttt{TA0CCR1 = 10000}.

Lastly, the only code in this program that executes repeatedly is the \texttt{while()} loop, and it simply contains an \texttt{if/else} conditional based on the status of input pin P1.3. Note the use of the bitwise-AND function to check for the value of bit 3 in the \texttt{P1IN} register while ignoring all the other bits in that word.
Pulse-width modulation (PWM) is nothing more than an expression of a pulse signal’s “on” time versus its total on+off time (“period”) as shown in this illustration:

\[
\text{Duty Cycle} = \frac{t_{on}}{t_{period}}
\]

An interesting way to use a microcontroller’s PWM output signal is to pass it through an RC network designed to “smooth” the pulse profile using the capacitor’s natural opposition to changes in voltage over time. This allows the microcontroller to generate a pseudo-analog voltage signal using nothing more than one of its digital (on/off) output terminals pulsing on and off:

The longer the RC network’s time constant value \( (\tau) \), the “smoother” the filtered signal will be, but also the slower it will be to adjust to changes in duty cycle. Ideally the time constant value should be many times longer than the PWM pulse signal’s period. The average voltage value of this filtered signal over time will be equal to the PWM pulse’s peak voltage multiplied by its duty cycle. In this program where their duty cycle is either 25% or 75%, the filtered signal’s average value will be \((0.25)(3.3) = 0.825 \text{ Volts}\) or \((0.75)(3.3) = 2.475 \text{ Volts}\), respectively.
2.25  C example: crude analog input

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS) in Debug mode.

The statement beginning with a pound symbol (#include <msp430G2553.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
This is an extremely simple program for reading a single analog input channel on the MSP430 microcontroller, and in fact it is only usable while using the “Step Into” function of Code Composer Studio’s “Debug” mode because there is no time delay in the code or any other provision to give the microcontroller enough time to adequately process the analog signal. By repeatedly clicking on the “Step Into” button within CCS, you may watch the numerical value stored in ADC10MEM as you vary the voltage applied between pin P1.3 and Ground on the microcontroller.

In this example circuit, the analog voltage is provided to pin P1.3 using a three-terminal component called a potentiometer. This resistive device uses a movable “wiper” contact to divide the 3.3 Volt DC power supply voltage by a proportion determined by the wiper’s position. With the wiper placed fully up pin P1.3 sees 3.3 Volts with respect to ground; placed fully down it sees 0 Volts; when placed mid-position the voltage will be 1.65 Volts.
2.25. **C EXAMPLE: CRUDE ANALOG INPUT**

In the following screenshot, you can see the Debug view of CCS showing the program’s execution (currently halted at the `while` instruction) and the decimal value 179 shown within `ADC10MEM`:

![Debug view of CCS](image)

The “Step Into” control button is the one resembling a yellow arrow pointing right and down, immediately to the right of the red-square “Stop” button on the CCS toolbar. The `ADC10MEM` “count” value of 179 just happens to correspond to the analog voltage signal being read by the microcontroller at the time.

All analog-to-digital converters generate a (digital) numerical value in response to the amount of (analog) voltage sensed. In the case of the MSP430G2553 microcontroller, that digital number value has a range of 0 to 1023 and should be directly proportional to the analog voltage over a range of 0 Volts to 3.3 Volts (the microcontroller’s regulated DC power supply voltage). We may express this mathematically as a simple proportion:

\[
\frac{V_{analog}}{3.3} = \frac{n_{digital}}{1023}
\]

Where,

- \(V_{analog}\) = Analog voltage sensed at the input terminal with reference to ground
- \(n_{digital}\) = Digital “count” value produced by the analog-to-digital conversion process
2.26 C example: analog-controlled LEDs

Schematic diagram

On the next page is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430G2553.h>`) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
#include "msp430g2553.h"

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Disable the Watchdog Timer

    /* Configure Output pins for LEDs */
    P1DIR |= BIT0 + BIT6; // Set P1.0 and P1.6 as outputs

    /* Configure Timer A */
    TA0CTL = TASSEL_2 + MC_2 + TAIE; // SMCLK clock, continuous, enable interrupt

    /* Configure ADC */
    ADC10CTL0 = SREF_0 + ADC10ON + ADC10SHT_3; // VCC ref, enable ADC, 64/sample
    ADC10AE0 |= BIT3; // Enable P1.3 as analog (A3) instead of general I/O
    ADC10CTL1 = INCH_3; // Select ADC Channel (channel A3)
    __bis_SR_register(GIE); // General Interrupts Enabled

    while(1)
    {
        if (ADC10MEM > 341) // ADC count value stored in ADC10MEM register
            P1OUT |= BIT6; // Turn on P1.6 if voltage > 1/3 of VCC

        if (ADC10MEM <= 341)
            P1OUT &= ~BIT6; // Turn off P1.6 if voltage <= 1/3 of VCC

        if (ADC10MEM > 682)
            P1OUT |= BIT0; // Turn on P1.0 if voltage > 2/3 of VCC

        if (ADC10MEM <= 682)
            P1OUT &= ~BIT0; // Turn off P1.0 if voltage < 2/3 of VCC
    }
}

#pragma vector=TMR0_A1_VECTOR
__interrupt void Timer_ISR (void) // Timer A interrupt service routine
{
    ADC10CTL0 |= ENC + ADC10SC; // Enables/starts analog-digital conversion
    TA0CTL &= ~TAIFG; // Clears the Timer A interrupt flag
}
This program uses Timer A to generate a periodic interrupt event to re-enable and re-start the analog-to-digital conversion process. Noteworthy features of this program include:

- Extensive use of symbolic constants to represent what would otherwise be arcane bit patterns; e.g. `BIT6` is a symbol for the binary constant \(0b01000000\) (hex \(0x40\))

- Use of the timer circuit’s functionality to replace what would otherwise be code running within the `while()` loop to repeatedly activate the microcontroller’s on-board 10-bit analog-to-digital converter

- Inclusion of an interrupt service routine (ISR) called by Timer A’s interrupt event

Let’s examine this code section by section.

After disabling the watchdog timer, the first instruction sets the “direction” of pins `P1.0` and `P1.6` to both be outputs. We will use these two outputs to drive LEDs, turning each of them on at different sensed voltage levels.

The on-board 10-bit analog-to-digital converter requires a short period of time to complete each conversion of a sampled analog input voltage, and must be re-started after each conversion is complete. A convenient way to do this periodic re-start is to configure the on-board timer (which is nothing more than a 16-bit counter driven by the clock signal) and have that timer generate an interrupt request every time it “rolls over” from full-count back to zero. This is called the timer’s “continuous” mode, where it counts from `0x0000` to `0xFFFF` repeatedly, setting its interrupt request flag (IFG) every time it returns to a value of zero. The `TA0CTL` initialization line sets up Timer A for this purpose.

Next, we need to initialize registers controlling the analog-to-digital converter itself. The example shown here is perhaps the simplest configuration possible for the MSP430’s 10-bit ADC: we declare the analog reference voltages to be the chip’s power supply \((V_{CC} = 3.3\, \text{Volts})\) and \((V_{SS} = 0\, \text{Volts})\), enable the ADC, and specify 64 counts per sample-and-hold cycle to give the analog signal voltage the maximum time to settle. On the model MSP430G2553IN20, every pin associated with Port 1 is able to serve as an analog input rather than a general I/O pin, and with the `ADC10AE0` initialization we declare the pin normally associated with I/O `P1.3` to be `A3` instead. Lastly, we select this analog-enabled pin to be the analog input signal for the ADC.

The last instruction executed before entering the `while()` loop sets the General Interrupt Enable (GIE) bit in the microcontroller’s Status Register, a necessary action if we are to use any general interrupts in our program.

In its default mode, the ADC10 converter generates an unsigned 10-bit count value ranging from 0 to 1023 (decimal) for a corresponding analog voltage signal ranging from the low reference voltage value to the high reference voltage value. Since we are using the chip’s power supply as those reference limits, we should expect to find a count value of 0 in the `ADC10MEM` register when pin `A3` senses 0 Volts, 1023 in that same register when it senses full \((+3.3\, \text{Volts})\) signal voltage, and a count value proportionate to these limits for any voltage in-between +3.3 and 0 Volts. It is the
The intent of this program is to energize LED P1.6 for any voltage greater than $\frac{1}{3}$ of the supply voltage, and energize LED P1.0 for any voltage exceeding $\frac{2}{3}$ the supply voltage. Therefore, the values 341 (one-third of 1023) and 682 (two-thirds of 1023) serve as the conditional thresholds for activating these LEDs.

The interrupt service routine function contains two instructions: one of them re-enables and re-starts the analog-digital conversion process, necessary after each conversion has completed. The second instruction clears the interrupt flag that was set by the timer when it returned to a value of zero. This clearing of the flag bit is typical for interrupt service routines, and is necessary to prevent that routine from being repeatedly called in an endless loop.

\textsuperscript{6}Interestingly, certain interrupt flags within the MSP430 microcontroller do not require the ISR to explicitly clear the flag bit. An example of this is the Capture/Compare interrupt flag generated by certain counting modes of the timer (CCIFG), which gets automatically cleared every time the interrupt request is granted. For most interrupts, expect that you will have to clear that interrupt flag bit within the code you write for the interrupt service request function.
2.27 C example: UART serial text transmission

Schematic diagram

On the next two pages is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (#include <msp430.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
C code listing

```c
#include <msp430.h>

void transmit_ASCII(char * tx_data);  // Prototype for transmit function

void main(void)
{
    unsigned int button, lastbutton;

    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    BCSCTL1 = CALBC1_1MHZ; // Set DCO to 1 MHz
    DCOCTL = CALDCO_1MHZ;
    P1DIR = 0xF7; // P1.3 input, all other P1 pins output
    P1REN = 0x08; // Enables pullup/pulldown resistor on P1.3
    P1OUT = 0x08; // Selects pullup resistor for P1.3
    P1SEL = BIT1 + BIT2; // P1.1 is UCA0RXD and P1.2 is UCA0TXD
    P1SEL2 = BIT1 + BIT2; // when P1SEL and P1SEL2 bits set
    UCA0CTL1 |= UCSSEL_2; // Have USCI use System Master Clock
    UCA0BR0 = 104; // These two USCI_A0 8-bit clock prescalers
    UCA0BR1 = 0; // comprise a 16-bit value to set bit rate.
    // Value of 104 = 9600 bps with 1MHz clock
    // (as per table 15.4 in MSP430 user guide)
    UCA0MCTL = UCBRS0; // Bit rate may be further tweaked using the
    // "modulation" register
    UCA0CTL1 &= ~UCSWRST; // Start the USCI's state machine

    while(1) // Endless loop
    {
        if ((P1IN & BIT3) == BIT3) // "button" represents state of P1.3
            button = 1; // == 1 when P1.3 is high
        else
            button = 0; // == 0 when P1.3 is low

        if((button == 0) && (lastbutton == 1)) // Detects P1.3 high-to-low
            { // without requiring interrupt
                transmit_ASCII("Hello world!\r\n"); // Transmit text
                transmit_ASCII("Goodbye now.\r\n"); // Transmit more text
            }
        lastbutton = button;
    }
```
void transmit_ASCII(char * char_array) // Function accepts char. array pointer
{
    unsigned int n = 0;
    while(char_array[n]) // Increment until null pointer (0) reached
    {
        while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
        UCA0TXBUF = char_array[n]; // Load each character into transmit buffer
        ++n; // Increment variable for array address
    }
}

Pressing the pushbutton brings P1.3 input to a “low” state. When the present state is low and the last state was high it means a high-to-low transition has occurred and this initiates the transmission of our text message. Without some provision to detect the falling edge of this input signal what would happen is the message would be transmitted over and over again for as long as P1.3 remains low. Other methods exist for detecting a high-to-low transition (e.g. using an interrupt generated by P1.3), but this current/last state method is worth considering which is why I presented it in this example.

In the C language it is impossible to pass an array or other complex data structure as an argument to function, and so instead the transmit_ASCII() function receives a pointer to an array comprised of ASCII characters, that array being named char_array within the function. Within the transmit_ASCII function’s while() loop we index this array to read its contents one character at a time until the end is reached as indicated by a “null” value. Note also the inner while() loop which acts to halt the microcontroller’s execution (by uselessly looping) until the status is no longer busy.
When the output signal (P1.2) is measured on a digital oscilloscope capable of RS-232 serial data decoding, we see the following result:
2.28 C example: UART serial text and number transmission

Schematic diagram

On the next two pages is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430.h>`) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
#include <msp430.h>

void transmit_ASCII(char * tx_data); // Prototype for alpha transmit function
void transmit_Num(int num); // Prototype for number transmit function

int x = 1486; // Flame temperature, degrees C

void main(void)
{
    unsigned int button, lastbutton;

    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    BCSCTL1 = CALBC1_1MHZ; // Set DCO to 1 MHz
    DCOCTL = CALDCO_1MHZ;

    P1DIR = 0xF7; // P1.3 input, all other P1 pins output
    P1REN = 0x08; // Enables pullup/pulldown resistor on P1.3
    P1OUT = 0x08; // Selects pullup resistor for P1.3
    P1SEL = BIT1 + BIT2; // P1.1 is UCA0RXD and P1.2 is UCA0TXD
    P1SEL2 = BIT1 + BIT2; // when P1SEL and P1SEL2 bits set

    UCA0CTL1 |= UCSSEL_2; // Have USCI use System Master Clock
    UCA0BR0 = 104; // These two USCI_A0 8-bit clock prescalers
    UCA0BR1 = 0; // comprise a 16-bit value to set bit rate.
    // Value of 104 = 9600 bps with 1MHz clock
    // (as per table 15.4 in MSP430 user guide)
    UCA0MCTL = UCBRS0; // Bit rate may be further tweaked using the
    // "modulation" register
    UCA0CTL1 &= ~UCSRST; // Start the USCI's state machine

    while(1) // Endless loop
    {
        if ((P1IN & BIT3) == BIT3) // "button" represents state of P1.3
            button = 1; // == 1 when P1.3 is high
        else
            button = 0; // == 0 when P1.3 is low

        if((button == 0) && (lastbutton == 1)) // Detects P1.3 high-to-low
            // without requiring interrupt
transmit_ASCII("Flame temp = "); // Transmit text
transmit_Num(x); // Transmit number
transmit_ASCII(" deg C\r\n"); // Transmit more text
}

lastbutton = button;
}
}

void transmit_ASCII(char * char_array) // Function accepts char. array pointer
{
    unsigned int n = 0;
    while(char_array[n]) // Increment until null pointer (0) reached
    {
        while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
        UCA0TXBUF = char_array[n]; // Load each character into transmit buffer
        ++n; // Increment variable for array address
    }
}

void transmit_Num(int num) // Function accepts 4-digit integer number
{
    unsigned int n = 0;
    char char_array[5];

    char_array[0] = (int)(num/1000) + 0x30;
    char_array[1] = (int)(num/100) - 10*(int)(num/1000) + 0x30;
    char_array[2] = (int)(num/10) - 10*(int)(num/100) + 0x30;
    char_array[3] = (int)(num) - 10*(int)(num/10) + 0x30;
    char_array[4] = 0;

    while(char_array[n]) // Increment until null pointer (0) reached
    {
        while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
        UCA0TXBUF = char_array[n]; // Load each character into transmit buffer
        ++n; // Increment variable for array address
    }
}

The purpose of this program is to output ASCII characters describing the temperature of a flame in degrees Celsius, where the flame temperature is simulated by a constant (in this case, 1486 °C). In any real temperature-sensing system the value of that x variable would be measured and digitized by the microcontroller’s analog-to-digital converter. However, for the sake of simplicity this
2.28. C EXAMPLE: UART SERIAL TEXT AND NUMBER TRANSMISSION

The demonstration program merely shows how an integer number may be broken down and transmitted as individual ASCII characters. You would have to add more code to the main() function to perform any real temperature measurement.

Many elements of this program are identical to that of the simpler text-only transmission example, and so will not be repeated here. For details on how the pushbutton’s state transition is detected, or how the transmit_ASCII() function works, refer to the prior example.

The transmit_Num() function receives a single integer as its argument, and through a series of integer arithmetic this number is broken down into its individual digits. ASCII codes for the digits 0 through 9 are \texttt{0b00110000} through \texttt{0b00111001}, respectively, and so all we need to do to convert a single decimal digit into its corresponding ASCII code is add \texttt{0x30} to it. This is done four times, placing the respective ASCII characters into another char array character array. The last element of that array is set to zero (null), and then it is processed just like it was in the transmit_ASCII() function.

Converting a four-digit integer number into four individual characters involves some creative arithmetic. If you force the computer to perform division and to cast the quotient as an integer number, it has the effect of dropping any digits to the right of the decimal point. For example, \texttt{(int)(12 / 5)} would yield a result of 2 even though \texttt{12 / 5} = 2.4 when performed ordinarily. Therefore, dividing a four-digit decimal number by 1000 will yield its most-significant digit. Similarly, dividing a four-digit decimal number by 100 yields its two most significant digits, which may be reduced to the least-significant of those by subtracting ten times the most-significant digit. This process is repeated four times until all four digits of the integer number have been represented as single ASCII character values. There are more computationally elegant ways of doing this, but showing all four steps as separate lines of code makes the algorithm easier to understand, and my purpose here is strictly educational.
When the output signal (P1.2) is measured on a digital oscilloscope capable of RS-232 serial data decoding, we see the following result:
2.29 C example: UART transmission of analog data

Schematic diagram

On the next two pages is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (#include <msp430.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
#include <msp430.h>

void transmit_ASCII(char * tx_data); // Prototype for alpha transmit function
void transmit_Num(int num); // Prototype for number transmit function

void main(void)
{
    unsigned int button, lastbutton;

    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer

    // Configure DCO oscillator for 1 MHz
    BCSCCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;

    // Configure Port 1 I/O
    P1DIR = 0x77; // P1.3 and P1.7 inputs, all other P1 pins output
    P1REN = 0x08; // Enables pullup/pulldown resistor on P1.3
    P1OUT = 0x08; // Selects pullup resistor for P1.3
    P1SEL = BIT1 + BIT2; // P1.1 is UCA0RXD and P1.2 is UCA0TXD
    P1SEL2 = BIT1 + BIT2; // when P1SEL and P1SEL2 bits set

    // Configure Timer A
    TACCTL1 = TASSEL_2 + MC_1 + TAIE; // SMCLK clock, up mode, enable interrupt
    TACCR0 = 10000; // Set maximum count value to generate an interrupt every
                    // 10000 cycles of the clock which is every 0.01 seconds

    // Configure ADC
    ADC10CTL0 = SREF_0 + ADC10ON + ADC10SHT_3; // VCC ref, enable ADC, 64/sample
    ADC10AE0 |= BIT7; // Enable P1.7 as analog (A7) instead of general I/O
    ADC10CTL1 = INCH_7; // Select ADC Channel (channel A7)

    // Configure UART
    UCA0CTL1 = UCSSEL_2; // Have USCI use System Master Clock
    UCA0BR0 = 104; // These two USCI_A0 8-bit clock prescalers
    UCA0BR1 = 0; // comprise a 16-bit value to set bit rate.
                 // Value of 104 = 9600 bps with 1MHz clock
                 // (as per table 15.4 in MSP430 user guide)
    UCA0MCTL = UCBRS0; // Bit rate may be further tweaked using the
                        // "modulation" register
    UCA0CTL1 &= ~UCSWRST; // Start the USCI's state machine
```c
__bis_SR_register(GIE); // General Interrupts Enabled

while(1)
{
    // NOTE: ADC count value stored in ADC10MEM register
    if (ADC10MEM > 341)
        P1OUT |= BIT6; // Turn on P1.6 if voltage > 1/3 of VCC

    if (ADC10MEM <= 341)
        P1OUT &= ~BIT6; // Turn off P1.6 if voltage <= 1/3 of VCC

    if (ADC10MEM > 682)
        P1OUT |= BIT0; // Turn on P1.0 if voltage > 2/3 of VCC

    if (ADC10MEM <= 682)
        P1OUT &= ~BIT0; // Turn off P1.0 if voltage < 2/3 of VCC

    if ((P1IN & BIT3) == BIT3) // "button" represents state of P1.3
        button = 1; // == 1 when P1.3 is high
    else
        button = 0; // == 0 when P1.3 is low

    if((button == 0) && (lastbutton == 1)) // Detects P1.3 high-to-low
        { // without requiring interrupt
            transmit_ASCII("ADC value = "); // Transmit text
            transmit_Num(ADC10MEM); // Transmit number (ADC value)
            transmit_ASCII(" counts\r\n"); // Transmit more text
        }

    lastbutton = button;
}

void transmit_ASCII(char * char_array) // Function accepts char. array pointer
{
    unsigned int n = 0;

    while(char_array[n]) // Increment until null pointer (0) reached
    {
        while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
        UCA0TXBUF = char_array[n]; // Load each character into transmit buffer
        ++n; // Increment variable for array address
    }
}
```
void transmit_Num(int num) // Function accepts 4-digit integer number
{
    char char_array[5];

    char_array[0] = (int)(num/1000) + 0x30;
    char_array[1] = (int)(num/100) - 10*(int)(num/1000) + 0x30;
    char_array[2] = (int)(num/10) - 10*(int)(num/100) + 0x30;
    char_array[3] = (int)(num) - 10*(int)(num/10) + 0x30;
    char_array[4] = 0;

    transmit_ASCII(char_array);
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_ISR (void) // Timer A interrupt service routine
{
    ADC10CTL0 |= ENC + ADC10SC; // Enables/stars analog-digital conversion
    TA0CTL &= ~TAIFG; // Clears the Timer A interrupt flag
}

Pressing the pushbutton brings P1.3 input to a “low” state and initiates the UART data transmission as in previous examples. The two LEDs connected to P1.0 and P1.6 serve as visual indicators of the analog voltage exceeding pre-set limits, and are for diagnostic purposes only.
Serial output signal (P1.2) decoded on a digital oscilloscope with potentiometer turned to 0 Volts (wiper fully down), resulting in a count value of 0000 from the 10-bit ADC:

Serial output signal (P1.2) decoded on a digital oscilloscope with potentiometer turned to 3.3 Volts (wiper fully up), resulting in a count value of 1023 from the 10-bit ADC ($2^{10} - 1$):
Another version of this program automatically transmits the approximate analog voltage every 5 seconds:

```c
#include <msp430.h>

void transmit_ASCII(char * tx_data); // Prototype for alpha transmit function
void transmit_Num(int num); // Prototype for number transmit function

unsigned int hundredths, seconds, minutes;

void main(void)
{
    unsigned int button, lastbutton, time, lasttime;

    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer

    // Configure DCO oscillator for 1 MHz
    BCSCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;

    // Configure Port 1 I/O
    P1DIR = 0x77; // P1.3 and P1.7 inputs, all other P1 pins output
    P1REN = 0x08; // Enables pullup/pulldown resistor on P1.3
    P1OUT = 0x08; // Selects pullup resistor for P1.3
    P1SEL = BIT1 + BIT2; // P1.1 is UCA0RXD and P1.2 is UCA0TXD
    P1SEL2 = BIT1 + BIT2; // when P1SEL and P1SEL2 bits set

    // Configure Timer A
    TA0CTL = TASSEL_2 + MC_1 + TAIE; // SMCLK clock, up mode, enable interrupt
    TACCR0 = 10000; // Set maximum count value to generate an interrupt every
    // 10000 cycles of the clock which is every 0.01 seconds

    // Configure ADC
    ADC10CTL0 = SREF_0 + ADC10ON + ADC10SHT_3; // VCC ref, enable ADC, 64/sample
    ADC10AE0 |= BIT7; // Enable P1.7 as analog (A7) instead of general I/O
    ADC10CTL1 = INCH_7; // Select ADC Channel (channel A7)

    // Configure UART
    UCA0CTL1 |= UCSSEL_2; // Have USCI use System Master Clock
    UCA0BR0 = 104; // These two USCI_A0 8-bit clock prescalers
    UCA0BR1 = 0; // comprise a 16-bit value to set bit rate.
    // Value of 104 = 9600 bps with 1MHz clock
    // (as per table 15.4 in MSP430 user guide)
    UCA0MCTL = UCBRS0; // Bit rate may be further tweaked using the
```
2.29. **C EXAMPLE: UART TRANSMISSION OF ANALOG DATA**

```c
// "modulation" register
UCA0CTL1 &= ~UCSWRST; // Start the USCI state machine
__bis_SR_register(GIE); // General Interrupts Enabled

while(1)
{
    // NOTE: ADC count value stored in ADC10MEM register
    if (ADC10MEM > 341)
        P1OUT |= BIT6; // Turn on P1.6 if voltage > 1/3 of VCC
    if (ADC10MEM <= 341)
        P1OUT &= ~BIT6; // Turn off P1.6 if voltage <= 1/3 of VCC
    if (ADC10MEM > 682)
        P1OUT |= BIT0; // Turn on P1.0 if voltage > 2/3 of VCC
    if (ADC10MEM <= 682)
        P1OUT &= ~BIT0; // Turn off P1.0 if voltage < 2/3 of VCC
    if (((P1IN & BIT3) == BIT3)) // "button" represents state of P1.3
        button = 1; // == 1 when P1.3 is high
    else
        button = 0; // == 0 when P1.3 is low
    if((button == 0) && (lastbutton == 1)) // Detects P1.3 high-to-low
        { // without requiring interrupt
            transmit_ASCII("Time = "); // Transmit text
            transmit_Num(seconds); // Transmit number (time value)
            transmit_ASCII(" seconds\r\n"); // Transmit more text
        }
    lastbutton = button;
    if (seconds % 5 == 0) // "time" represents when we have reached
        time = 1; // the time interval (every 5 seconds)
    else
        time = 0; // We're not at the time interval
    if((time == 1) && (lasttime == 0)) // Detects time interval without
        { // requiring interrupt
            transmit_ASCII("Voltage = "); // Transmit text
        }
```

transmit_Num(ADC10MEM * 64 / 20); // Transmit approx. voltage
transmit_ASCII(" mV\r\n"); // Transmit more text
}

lasttime = time;
}
}

void transmit_ASCII(char * char_array) // Function accepts char. array pointer
{
  unsigned int n = 0;

  while(char_array[n]) // Increment until null pointer (0) reached
  {
    while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
    UCA0TXBUF = char_array[n]; // Load each character into transmit buffer
    ++n; // Increment variable for array address
  }
}

void transmit_Num(int num) // Function accepts 4-digit integer number
{
  char char_array[5];

  char_array[0] = (int)(num/1000) + 0x30;
  char_array[1] = (int)(num/100) - 10*(int)(num/1000) + 0x30;
  char_array[2] = (int)(num/10) - 10*(int)(num/100) + 0x30;
  char_array[3] = (int)(num) - 10*(int)(num/10) + 0x30;
  char_array[4] = 0;

  transmit_ASCII(char_array);
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_ISR (void) // Timer A interrupt service routine
{
  ADC10CTL0 |= ENC + ADC10SC; // Enables/steps analog-digital conversion
  ++hundredths; // Increments "hundredths" variable every 0.01 seconds

  if (hundredths > 99)
  {
    hundredths = 0;
    ++seconds;
In this version, the interrupt service routine increments hundredths, seconds, and minutes variables to give the main function’s while() loop a way of knowing how much time has elapsed. In addition to printing the value of the seconds variable with every press of pushbutton P1.3, this program automatically transmits an approximation of analog voltage every 5 seconds.

The scaling of an integer count value (range = 0 to 1023) to an analog voltage value is approximate because our mathematical operations are limited to integer arithmetic. We multiply ADC10MEM by 64 in order to create a value as close to the 16-bit maximum of 65535 without exceeding it, then divide by another factor (in this case, 20) to yield a value for millivoltage. The calculation isn’t perfect, as a 3.3 Volt input will yield a transmitted number of 3273 mV rather than 3300 mV, but this error decreases with signal voltage.
2.30 C example: simple datalogger

Schematic diagram

On the next two pages is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (#include <msp430.h>) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
#include <msp430.h>

void transmit_ASCII(char * tx_data); // Prototype for alpha transmit function
void transmit_Num(int num); // Prototype for number transmit function

unsigned int hundredths, seconds, minutes, sample;
unsigned int data[128]; // Array for storing logged data

void main(void)
{
    unsigned int button, lastbutton, time, lasttime;

    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer

    // Configure DCO oscillator for 1 MHz
    BCSCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;

    // Configure Port 1 I/O
    P1DIR = 0x77; // P1.3 and P1.7 inputs, all other P1 pins output
    P1REN = 0x08; // Enables pullup/pulldown resistor on P1.3
    P1OUT = 0x08; // Selects pullup resistor for P1.3
    P1SEL = BIT1 + BIT2; // P1.1 is UCA0RXD and P1.2 is UCA0TXD
    P1SEL2 = BIT1 + BIT2; // when P1SEL and P1SEL2 bits set

    // Configure Timer A
    TA0CTL = TASSEL_2 + MC_1 + TAIE; // SMCLK clock, up mode, enable interrupt
    TACCR0 = 10000; // Set maximum count value to generate an interrupt every
                    // 10000 cycles of the clock which is every 0.01 seconds

    // Configure ADC
    ADC10CTL0 = SREF_0 + ADC10ON + ADC10SHT_3; // VCC ref, enable ADC, 64/sample
    ADC10AE0 |= BIT7; // Enable P1.7 as analog (A7) instead of general I/O
    ADC10CTL1 = INCH_7; // Select ADC Channel (channel A7)

    // Configure UART
    UCA0CTL1 |= UCSSEL_2; // Have USCI use System Master Clock
    UCA0BR0 = 104; // These two USCI_A0 8-bit clock prescalers
    UCA0BR1 = 0; // comprise a 16-bit value to set bit rate.
                    // Value of 104 = 9600 bps with 1MHz clock
                    // (as per table 15.4 in MSP430 user guide)
    UCA0MCTL = UCBR0S0; // Bit rate may be further tweaked using the
// "modulation" register
UCA0CTL1 &= ~UCSWRST; // Start the USCIs state machine

__bis_SR_register(GIE); // General Interrupts Enabled

while(1)
{
    // NOTE: ADC count value stored in ADC10MEM register
    if ((P1IN & BIT3) == BIT3) // "button" represents state of P1.3
        button = 1; // == 1 when P1.3 is high
    else
        button = 0; // == 0 when P1.3 is low

    if((button == 0) && (lastbutton == 1)) // Detects P1.3 high-to-low
    {
        for (sample = 0 ; sample < 128 ; ++sample)
        {
            transmit_Num(sample); // Transmit sample number
            transmit_ASCII(" , "); // Transmit comma delimiter
            transmit_Num(data[sample]); // Transmit logged data
            transmit_ASCII("\r\n"); // Transmit new line
        }
    }

    lastbutton = button;

    if (seconds % 5 == 0) // "time" represents when we have reached
        time = 1; // the time interval (every 5 seconds)
    else
        time = 0; // We’re not at the time interval

    // Recording a sample to the data[] array
    if((time == 1) && (lasttime == 0)) // Detects time interval without
    {
        P1OUT ^= BIT6; // Toggle P1.6 to visually indicate new sample
        data[sample & 0x7F] = ADC10MEM;
        ++sample;
    }

    lasttime = time;
}
2.30. C EXAMPLE: SIMPLE DATALOGGER

void transmit_ASCII(char * char_array) // Function accepts char. array pointer
{
    unsigned int n = 0;

    while(char_array[n]) // Increment until null pointer (0) reached
    {
        while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
        UCA0TXBUF = char_array[n]; // Load each character into transmit buffer
        ++n; // Increment variable for array address
    }
}

void transmit_Num(int num) // Function accepts 4-digit integer number
{
    char char_array[5];

    char_array[0] = (int)(num/1000) + 0x30;
    char_array[1] = (int)(num/100) - 10*(int)(num/1000) + 0x30;
    char_array[2] = (int)(num/10) - 10*(int)(num/100) + 0x30;
    char_array[3] = (int)(num) - 10*(int)(num/10) + 0x30;
    char_array[4] = 0;

    transmit_ASCII(char_array);
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_ISR (void) // Timer A interrupt service routine
{
    ADC10CTL0 |= ENC + ADC10SC; // Enables/stops analog-digital conversion
    ++hundredths; // Increments "hundredths" variable every 0.01 seconds

    if (hundredths > 99)
    {
        hundredths = 0;
        ++seconds;
    }

    if (seconds > 59)
    {
        seconds = 0;
        ++minutes;
TA0CTL &= ~TAIFG; // Clears the Timer A interrupt flag

Pressing the pushbutton brings P1.3 input to a “low” state and initiates the UART data transmission, producing a comma-separated variable (CSV) text listing showing 128 samples stored in the data[] array. In this particular example, a new sample is stored in the array every 5 seconds. Pressing the P1.3 switch both prompts the UART transmission of the entire array as well as resets the sample variable so that new data entries begin at the start of the array. The LED connected to P1.6 toggles every time a new sample is saved to the array, providing visual indication that the program is functioning. Every time that LED turns on, a new sample is taken; every time it turns off, another sample is taken.

The sample variable increments once per sample, and since its value may go well beyond the 0-127 limit imposed by the array’s size, we use a bitwise-AND operation with 0x7F to only use the lower seven bits of its 16-bit value. When sample increments to a value equal to or greater than any multiple of 128, new data overwrites old data stored in the beginning elements of the array.

EIA/TIA-232 requires a bipolar signal (above and below ground potential) of at least ±5 Volts for reliable operation. Rather than use a specialized “driver” IC to interface the microcontroller to a personal computer serial port, we have instead cobbled together a simple driver circuit consisting of an opto-isolator and two 6 Volt batteries.
Data is received by a personal computer running a serial terminal software application such as **Termite** or **Kermit** or **Hyperterminal**. Here is a screenshot showing **Termite** capturing the serial data stream after pressing pushbutton P1.3. I used the computer’s mouse to select the first 25 samples and then pasted the text into a text editor, saving it as a file named `data.csv`:

<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0.275</td>
</tr>
<tr>
<td>0001</td>
<td>0.539</td>
</tr>
<tr>
<td>0002</td>
<td>0.633</td>
</tr>
<tr>
<td>0003</td>
<td>0.758</td>
</tr>
<tr>
<td>0004</td>
<td>0.516</td>
</tr>
<tr>
<td>0005</td>
<td>0.403</td>
</tr>
<tr>
<td>0006</td>
<td>0.253</td>
</tr>
<tr>
<td>0007</td>
<td>0.173</td>
</tr>
<tr>
<td>0008</td>
<td>0.252</td>
</tr>
<tr>
<td>0009</td>
<td>0.331</td>
</tr>
<tr>
<td>0010</td>
<td>0.209</td>
</tr>
<tr>
<td>0011</td>
<td>0.127</td>
</tr>
<tr>
<td>0012</td>
<td>0.140</td>
</tr>
<tr>
<td>0013</td>
<td>0.105</td>
</tr>
<tr>
<td>0014</td>
<td>0.105</td>
</tr>
<tr>
<td>0015</td>
<td>0.105</td>
</tr>
<tr>
<td>0016</td>
<td>0.032</td>
</tr>
<tr>
<td>0017</td>
<td>0.032</td>
</tr>
<tr>
<td>0018</td>
<td>0.133</td>
</tr>
<tr>
<td>0019</td>
<td>0.183</td>
</tr>
<tr>
<td>0020</td>
<td>0.263</td>
</tr>
<tr>
<td>0021</td>
<td>0.333</td>
</tr>
<tr>
<td>0022</td>
<td>0.391</td>
</tr>
<tr>
<td>0023</td>
<td>0.447</td>
</tr>
<tr>
<td>0024</td>
<td>0.524</td>
</tr>
<tr>
<td>0025</td>
<td>0.606</td>
</tr>
<tr>
<td>0026</td>
<td>0.465</td>
</tr>
<tr>
<td>0027</td>
<td>0.000</td>
</tr>
<tr>
<td>0028</td>
<td>0.000</td>
</tr>
<tr>
<td>0029</td>
<td>0.000</td>
</tr>
<tr>
<td>0030</td>
<td>0.000</td>
</tr>
<tr>
<td>0031</td>
<td>0.000</td>
</tr>
<tr>
<td>0032</td>
<td>0.000</td>
</tr>
</tbody>
</table>
Once stored as a text file, this CSV data may be read by any spreadsheet software (e.g., Microsoft Excel) to generate a graph:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>275</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>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>539</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>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>633</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>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>758</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>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>518</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>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>403</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>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>253</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>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>173</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>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>252</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>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>331</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>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>299</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>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>127</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>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>340</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>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>105</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>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>105</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>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>105</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>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>32</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>
</tr>
<tr>
<td>18</td>
<td>17</td>
<td>32</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>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>32</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>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>183</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>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>263</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>
</tr>
<tr>
<td>22</td>
<td>21</td>
<td>333</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>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>391</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>
</tr>
<tr>
<td>24</td>
<td>23</td>
<td>447</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>
</tr>
<tr>
<td>25</td>
<td>24</td>
<td>524</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>
</tr>
</tbody>
</table>
2.31 Example: interposing MCU to a heavy load

The following MCU controls the illumination of two LEDs, one connected to output pin P1.0 and the other connected to output pin P1.6:

![Diagram of the MCU circuit]

Suppose we wanted to control a load with much higher current requirements than a simple LED – something requiring far more current than the MCU can source or sink through any of its output pins. This will require some electrically-controlled switching device to “interpose” between the MCU’s output and the high-current load. A transistor is a good candidate for this task.
Connecting a bipolar junction transistor (BJT) of the NPN variety between the output pin and the load will suffice:

![Diagram of the circuit](attachment:diagram.png)

The current-limiting resistor $R_{limit}$ allows just enough current from output pin P1.6 of the microcontroller to drive the BJT into saturation, where it is as fully “on” as it is able. When saturated, we would expect to see approximately 0.7 Volts dropped between the BJT’s base and emitter terminals, and approximately 0.3 Volts dropped between the BJT’s collector and emitter terminals. This means the load receives approximately 11.7 Volts with P1.6 in its “high” state. When P1.6 goes to a “low” state the BJT will turn off, dropping zero voltage between its base and emitter, and dropping the full 12 Volts between its collector and emitter.
In some ways a MOSFET would be a better solution than a BJT because when turned on it presents very little resistance to load current and therefore makes a very power-efficient switch in its “on” state. It may seem as though something this simple might work, but it’s quite possible it will not:

Many power MOSFETs have $V_{GS(on)}$ minimum requirements greater than 3.3 Volts, which is all this MCU is able to output with its 3.3 Volt power supply. Thus, a circuit like this would be unreliable. While the MOSFET will certainly be off whenever pin P1.6 is in a “low” state (0 Volts to ground, logical “0”), the MOSFET will very likely only turn partially on whenever pin P1.6 goes to a “high” state, resulting in less-than-full voltage across the load and overheating of the MOSFET due to it exhibiting substantial $R_{DS(on)}$ resistance.
The following solution is simple, using a BJT directly driven by the MCU to provide a higher-voltage gate-to-source signal for the MOSFET to operate on. The MCU will have no trouble activating the BJT since bipolar junction transistors only require 0.7 Volts base-to-emitter to turn on, which is far less than the several Volts of gate-source voltage typically required to activate a power MOSFET:

The resistor between pin P1.6 and the BJT’s base terminal simply limits base current to a level acceptable to the MCU. The resistor between the +12 Volt power supply terminal and the BJT’s collector terminal simply acts as a “pull-up” to provide a “high” voltage to the MOSFET’s gate terminal whenever the BJT is off, so its value may be quite large.

A caveat to be aware of with this added BJT stage is that the MCU’s output logic state will be the inverse of the load’s energization state: the load will now be energized when MCU pin P1.6 is “low” (0 Volts to ground, which is a “0” digital state) and de-energized when pin P1.6 is “high” (+3.3 Volts to ground, a “1” digital state). However, it is generally easy to alter the programming of the MCU to re-invert the logic so that the load energizes with an internal “1” digital state and turns off with a “0” digital state if that is what you desire.

The only real limit here is that this pull-up resistor must “charge” up the MOSFET’s gate-to-source capacitance in a reasonably short amount of time, but since $C_{GS}$ is typically very small for most MOSFETs this is generally not a concern. Only if the switching frequency is very high would we concern ourselves about sizing that pull-up resistor small enough to charge up $C_{GS}$ quickly enough. Otherwise, a relatively large pull-up resistor value – say, 10 kΩ or more – will minimize power dissipation when the BJT is “on” which is a good thing.
Another caveat applies to loads that are inductive. When switching power to any inductive load (e.g. solenoid actuator, relay coil, electric motor), you will need to include a commutating diode in parallel with that load to provide a safe pathway for current whenever the MOSFET turns off and the load’s magnetic field collapses:

When the MOSFET is on, the inductive load truly behaves as a load with + on top and − on bottom, reverse-biasing this diode so that no current flows through it. However, at the moment in time when the MOSFET turns off, this inductive load will have energy stored in its internal magnetic field, and as that magnetic field collapses the energy must go somewhere. As that field collapses, the inductor becomes a source and its voltage polarity rapidly swaps such that + will be on bottom and − on top. This forward-biases the diode and allows it to safely pass current, dissipating that magnetically-stored energy in the form of heat within the diode. If the diode were not there, the load’s inductance could generate extremely high voltages as it attempted to force current downward through the “off” MOSFET, likely ruining the transistor and possibly even ruining the microcontroller as well!
A good design practice is to electrically isolate the microcontroller from the power circuitry using a device called an opto-isolator. This is a light-sensitive switching device (e.g. transistor or thyristor) packaged together with a light-emitting diode such that energizing the LED turns on the switching device without any need for electrical contact between the controlling and controlled circuits. Using opto-isolators ensures that no mishap on the high-power side of the circuit can damage the microcontroller, which is especially useful for cases where someone not skilled in component selection may be the person connecting loads to the microcontroller:
Opto-isolators are more expensive than ordinary transistors, and generally have low switching current limits, but are unparalleled in their ability to electrically isolate one circuit from another. I highly recommend their use when interposing high-power circuitry to microcontroller I/O pins. The following photograph shows several opto-isolators (model PVT 322) soldered into a printed circuit board:

Larger opto-isolators with higher voltage and/or current switching ratings are called solid state relays (SSRs), and are similarly useful for interposing sensitive microcontroller output pins to high-power circuits:
2.32 C example: sine-wave inverter

Schematic diagram

On the next two pages is a listing of the entire C-language code, tested and run on a model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430.h>`) is a directive telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.
# Example: Sine-Wave Inverter

```c
#include "msp430.h"

// Array of sine values, eight values per quarter-wave, 16 values per half-wave
// 260 * sin((\pi/2) * (n/8)) for n = 1 through 8
unsigned int sinearray[16] = {0, 51, 99, 144, 184, 216, 240, 255, 260, 255, 240, 216, 184, 144, 99, 51};

unsigned int n;

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Disable the Watchdog Timer

    // Set Digitally Controlled Oscillator to 1 MHz
    BCSCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;

    P1DIR = 0xFF; // Set all P1 pins as outputs

    // Configure P1.6 pin (Note: BIT6 is a symbol for the constant 0b01000000)
    // When P1DIR=1 and P1SEL=1 and P1SEL2=0 for timer output TA0.1 on pin 6
    P1DIR |= BIT6; // Sets bit 6 while leaving the other bits untouched
    P1SEL |= BIT6; // Sets bit 6 while leaving the other bits untouched
    P1SEL2 &= ~BIT6; // Clears bit 6 while leaving the other bits untouched

    // Configure timer A for pulse-width modulation
    TA0CTL = TASSEL_2 + MC_3 + ID_0 + TAIE;
    // TASSEL_2 = SMCLK as input clock,
    // MC_3 = Mode Control 3 (up/down)
    // (counts "up" to TA0CCR0 then counts down to 0)
    // ID_0 = No clock frequency division
    // TAIE = Enable timer interrupt

    TA0CCR0 = 260; // Maximum count value determines frequency =
    // SMCLK/ClockDivide/TA0CCR0 (1 MHz/1/260 = 3846.15 Hz)
    // Gives 60 Hz sine wave in up/down mode given 16 steps per half-cycle
    TA0CCR1 = 0; // Initialize the duty cycle at 0% = TACCR1/TA0CCR0
    TA0CCTL1 = OUTMOD_2; // "Toggle/Reset" = TA0.1 high when TA0CCR1 > count

    __bis_SR_register(GIE); // General Interrupts Enabled
}
```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_ISR (void) // Timer A interrupt service routine
{
   ++n;

   TA0CCR1 = sinearray[n & 0x0F]; // Only uses four LSB’s of n for a
   // 0-15 count

   if (TA0CCR1 == 0) // Toggle P1.0 once every zero-crossing
      P1OUT ^= BIT0;

   TA0CTL &= ~TAIFG; // Clears the Timer A interrupt flag
}

The purpose of this program is to output two signals capable of driving a pair of transistors for converting DC into AC power. The timer functions as a pulse-width-modulator, outputting a PWM signal on pin P1.6 designed to mimic the amplitude of a sine wave in sixteen steps every half-cycle. Pin P1.0 simply toggles once at every “zero” point of the pseudo-sine wave PWM signal, serving as a “steering” signal for the two AND gates external to the microcontroller. These AND gates ensure only one power transistor gets activated at any moment in time. With TA0CCR0 set to a value of 260 (establishing the maximum count value for the timer) and a clock frequency of 1 MHz, the “steering” pulse signal out of P1.0 has a frequency very nearly equal to 60 Hz.

Both signals are shown here on an oscilloscope, P1.0 as the red trace and P1.6 as the blue trace:
The interrupt service routine code bears some explanation. Its job is to step through the sixteen \texttt{sinearray} values one by one using \texttt{n} as the index variable. One could write some conditional code to ensure \texttt{n} only counts from 0 to 15 and then recycles back to 0 again, but such instructions could add variable delay times to the ISR’s execution. The code shown here is computationally simpler and consistent in its execution time, simply applying a bitwise-AND mask to the value of \texttt{n} which is in reality a 16-bit unsigned binary number, so that the \texttt{sinearray} only sees an effective \texttt{n} value from 0 to 15 (0x00 to 0x0F).

Note the safeguards in the interface and power circuitry. Opto-isolators provide complete electrical isolation between the microcontroller and the power transistors. A fuse protects the DC source and power circuitry from overcurrent. Zener diodes clamp voltage across each power transistor in the event of inductive “kickback” from the transformer’s primary winding when no load is connected to the secondary. These features are not trivial for power circuitry, but are essential for safety.
Chapter 3

Tutorial

3.1 Microcontrollers versus microprocessors

A processor is a subsystem of a digital computer whose function it is to fetch and execute instructions stored in memory. A microprocessor is a digital processor built on a single semiconductor die, packaged as an integrated circuit. A microcontroller goes one step further to include volatile and nonvolatile memory as well as I/O drivers and other useful peripherals on the same chip. In other words, while a microprocessor is the central element of a digital computer, a microcontroller is a self-contained digital computer on a single chip. With most microcontrollers it is possible to have them execute code with no other integrated circuits connected to it, making them ideal for “embedding” within larger electronic circuits.

General-purpose digital computers typically utilize microprocessors rather than microcontrollers, due to the impracticality of building large amounts of ROM and RAM into a single integrated circuit. It simply makes more sense to build a general-purpose computer from one or more microprocessors surrounded by high-capacity memory devices. This means microcontrollers are designed for relatively small-scale computing applications – in terms of physical size, of power requirements, and also in terms of code size – especially applications where the device is expected to monitor and/or control real-world quantities.

Modern life in an industrialized nation is replete with microcontrollers “embedded” within various products. Most automobiles have several microcontrollers (e.g. engine control unit, drivetrain control unit, entertainment system controller, security system controller, etc.), most full-featured home appliances have at least one, and we even see microcontrollers used within electronic toothbrushes.
When you examine the pin diagram for a typical microprocessor you will find a majority of pin functions for bus lines, both address and data. A diagram showing pin functions on the venerable Zilog Z80 microprocessor is shown below:

As you can see, most of these pins' functions relate to memory access and control of the busses that connect the microprocessor to external devices. This is an IC built to exchange data with other ICs.

---

1Early microprocessors had just a few pins designated for power rails, but some modern microprocessors reserve most of their pins just for + and − DC power. The reason for this is the high current demand of modern high-speed microprocessors, necessitating multiple pins to safely conduct that current between the IC and the printed circuit board, as well as multiple points on the silicon die to convey that current without generating excessive heat.
3.1. MICROCONTROLLERS VERSUS MICROPROCESSORS

By contrast, a pin diagram for the equally venerable Intel 8051 microcontroller shows a very different emphasis:

Instead of bus lines and control lines, we see input/output ports dominating the pin assignments. In a general-purpose computer the I/O ports generally comprise a fairly small portion of the system because the greater emphasis is placed on access to external memory devices. With a microcontroller having its own built-in memory there is less need for that, and therefore more emphasis may be placed on interfacing connections to external sensors, switches, loads, etc. This is an IC built to directly sense and control things.

---

In all fairness to the 8051, most of these ports could also serve auxiliary duty as address, data, and bus control lines. Port 0 (P0.0 through P0.7) comprise data lines D0 through D7 as well as the first eight address lines (A0 through A7). Port 2 (P2.0 through P2.7) supplies the last eight address lines (A8 through A15). Port 3 (P3.0 through P3.7) provides bus control lines.
Microcontroller memory maps also differ from that of microprocessors. For many microprocessors the memory map is defined by the ways in which memory and I/O devices are wired to the IC, giving the computer designer some degree of choice where data will be located in the microprocessor’s addressable space. Microcontrollers tend to have a memory map fixed by the manufacturer, which makes sense given that the memory is built-in to the microcontroller itself.\footnote{It should be noted that most microcontrollers both legacy and modern offer the option of external (supplemental) memory and are not necessarily limited to the on-board memory.}

Instruction sets are another point of distinguishing microprocessors from microcontrollers. Since the emphasis of a microcontroller is on sensing and controlling external discrete devices, you will typically find microcontrollers offering more bit-level instructions (e.g. instructions allowing the programmer to individually set and clear bits within a digital word, and to test the logic state of individual bits as well) than microprocessors.

Furthermore, microcontrollers often contain subcircuits that are rarely if ever found inside of general-purpose microprocessors. These include, but are not limited to:

- Clock oscillators
- Timer circuits
- Analog-digital converters
- Digital-analog converters
- Serial data communication drivers
- Analog comparators
- Watchdog timers\footnote{A “watchdog timer” is a dedicated timing circuit intended to serve as a self-check on the processor’s operation. Routine execution of the user program keeps resetting the watchdog timer before it has an opportunity to “time out”. If the watchdog timer does reach the end of its count without being reset, the assumption is that the program has halted for some reason. This may be used to trigger an automatic reset of the microcontroller to get it running again. This feature is obviously useful for critical measurement and control applications where the consequences of the microcontroller “crashing” may be severe.}
3.2 MSP430G2553 PIN FUNCTIONS

A microcontroller is a complete computer on a single integrated circuit (IC) “chip” designed to receive data from and send data to other digital circuits. Being very small in size means that microcontrollers typically have far less computing power and memory space than general-purpose computers such as personal computers. They find wide application as controllers embedded within devices such as personal appliances (e.g., electronic toothbrushes, stopwatches), complex machines (e.g., automobiles, aircraft), and industrial devices (e.g., pressure sensors, thermostats).

In order to incorporate a microcontroller into any digital circuit design, we must first understand what each of the pins on the microcontroller IC do. This is the purpose of this Tutorial section. We will focus on the Texas Instruments model MSP430G2553 microcontroller in this Tutorial, as it is well-documented, reliable, and very inexpensive.

The model MSP430G2553 microcontroller is part of the larger MSP430 “family” of microcontrollers manufactured by Texas Instruments, and it will be the model most of this Tutorial references. A functional pin diagram for the 20-pin version of the MSP430G2553 is shown here:

![MSP430G2553 Pin Diagram]

Nearly every one of these pins serves multiple functions. For example, the **RESET** and **TEST** pins also function as the “Spy-Bi-Wire” serial programming interface pins used for programming the device’s built-in Flash memory and for information exchange related to program debugging. Also, every port pin supports additional functions such as timer inputs, analog inputs, serial data inputs and outputs, etc. Most of the Port pins default to functioning as digital I/O upon power-up, but some do not: a case in point are the pins shown here as P2.6 and P2.7 which default to crystal oscillator connections (but which may be configured to function as digital I/O by setting the appropriate bits within the **P2SEL** register).
The Texas Instruments MSP-EXP430G2ET “LaunchPad” development board provides a 20-pin DIP socket for the removable MSP430G2553 microcontroller, LED indicators, wire sockets and wire-wrap pins, and a built-in debugging system called the eZ-FET. A photograph of this board appears here, the upper portion containing the eZ-FET debugger and the lower portion containing the removable MSP430G2553 microcontroller:

A USB cable connects the LaunchPad board to any personal computer, where you may run either Code Composer Studio (CCS) or Energia programming software. CCS is a full-featured development application (generally known as an IDE – Integrated Development Environment) while Energia is a simplified programming interface designed to mimic the popular hobbyist-level Arduino microcontroller development software. Texas Instruments also hosts a “cloud-based” version of CCS with Energia built in, so you may write and test software for the microcontroller without needing to download any software at all to your personal computer.

Interesting, the eZ-FET portion of the LaunchPad development board also has current-monitoring capability to measure the microcontroller’s power consumption. This is a useful feature when developing battery-powered applications, as you are able to optimize the microcontroller’s programming for minimal current draw.

The open-source Arduino microcontroller series is a subject unto itself and will not be discussed in this Tutorial. There are so many good tutorials already written for Arduino that it would be redundant for me to write another.
3.3 Elementary output and input

General-purpose I/O (input/output) terminals on a microcontroller are typically organized as ports, in this case each port having eight terminals. In their most basic usage, these pins are digital in nature which means they will either be at a “low” or “high” logic state as represented by a ground-referenced voltage signal. Inside the microcontroller there will either be a special hardware register or an area of general memory associated with each I/O port, and this is how those pins’ logic states are represented to the program executing on the microcontroller’s processor: as individual bits in a data word located at some specific memory address.

Looking closely at a functional pin diagram for the MSP430G2553 microcontroller, we see double-headed arrows on all the port pins. This means these pins may function as either inputs or outputs:

![MSP430G2553 Pin Diagram]

As output pins, their logic states will be driven by bits written into each port’s register by the running code. As input pins, logic states asserted on those pins by external circuitry will be read as individual bits in another register associated with that port. For example, on the MSP430 microcontroller there is an input and an output register pair for Port 1, and an input and output register pair for Port 2 as shown in the following list:

- Port 1 input register = P1IN
- Port 1 output register = P1OUT
- Port 2 input register = P2IN
- Port 2 output register = P2OUT

In fact, in the MSP430 there are several different registers associated with each port, not just input and output. These other registers determine such things as the direction for each pin (whether it will be an input or an output), whether the pin is able to function as an interrupt, etc. Writing
programs to make use of these I/O pins is a matter of writing to and reading from the appropriate registers. This section will describe the most elementary uses of general-purpose I/O ports on the MSP430.

Suppose we wished to use all eight pins of Port 1 to drive LED indicator lamps. Our schematic diagram would look something like this:

The \( V_{CC} \) pin is the microcontroller’s positive DC power supply terminal, and \( V_{SS} \) is its negative supply terminal; between these terminals we must apply 3.3 Volts DC. The Reset (\( RST \)) line is active-low, which means the microcontroller will be forced into its “reset” state and halt execution of any program when this input line is at a logical “low” state (grounded). Therefore, we must connect the Reset line “high” (to +3.3 Volts) in order to ensure the microcontroller will be able to run its program. Here we use a pull-up resistor to make the Reset input high rather than a direct wire connection, because using a resistor gives us the freedom to ground the Reset pin at any time to force the microcontroller to reset without having to disconnect a wire from +V. Lastly, we see the eight LEDs connected to the Port 1 pins, each with its own current-limiting resistor.
Outputting data to these eight LEDs requires at minimum two instructions running in the microcontroller’s program: first, we must configure all eight pins of Port 1 to be outputs rather than inputs; second, we must write the desired bit states to the eight bits of Port 1’s output register.

The input versus output functionality of Port 1’s pins are assigned by bits written to the P1DIR register, with 1 designating an output and 0 designating an input. Once this has been done, we may cause Port 1’s pins to go to “high” or “low” states by writing a value to the P1OUT register. The following code examples show segments of code (not a complete program!) for setting all eight bits of Port 1 as outputs and then turning on an arbitrary pattern of LEDs:

**Assembly source code:**

```assembly
mov #0b11111111,P1DIR
mov #0b01011101,P1OUT
```

As you can see, in MSP430 assembly language we use the “move” (mov) instruction to write numerical values to each of these registers. Here, we specify those values in binary format to make it very easy for us to relate the values to individual pin functions and logic states on Port 1. We could have just as readily specified those values in any other numeration system (e.g. octal, hex, or decimal) and the port would function just as well.

**C source code:**

```c
P1DIR = 0b11111111;
P1OUT = 0b01011101;
```

Performing the same register-write operation in the C programming language consists of treating P1DIR and P1OUT as integer variables, setting each of them equal to the desired binary values.
Using red coloring to indicate energized LEDs, this is the result of writing those numerical values to the P1DIR and P1OUT registers:

```
P1DIR = 0b11111111
P1OUT = 0b0101101
```
To use all eight bits of Port 1 as inputs rather than outputs, we must first connect those pins to external circuitry capable of asserting “high” and “low” states on those pins. Thus, our schematic needs to be modified. Instead of LEDs and current-limiting resistors, we have toggle switches and pull-down resistors connected to the eight Port 1 pins:

Register P1IN is where all input bit states reside in the microcontroller’s memory. We must still write data to the direction register P1DIR, this time clearing (0) all eight bits rather than setting (1) them as shown in these Assembly-language and C-language program instruction examples:

**Assembly source code:**
```
mov #0b00000000,P1DIR
```

**C source code:**
```
P1DIR = 0b00000000;
```

Instead of writing data to the P1OUT register as we did before, we will let the external switches determine the bit states of the P1IN register, which may be read an acted upon by *conditional* instructions we place in our code.
Using red coloring to indicate closed switches and “high” states, this is the result of configuring all of Port 1’s pins to be inputs (P1DIR = 0x00000000) and closing some of the switches:
3.4 Bit-level output instructions

Writing to an output port is as simple as assigning a value to the proper PxOUT register, but bear in mind that this register holds eight bits (for upwards of eight output pins) and therefore your assignment instruction is dealing with all eight pins of that port at a time. When programming in assembly language or in C/C++, the instructions are remarkably similar. For example, suppose we wished to set pin P1.7 high and clear all other port 1 pins low. This would mean writing 0b10000000 to the P1OUT register:

**Assembly-language example:**

```
mov #0b10000000, P1OUT
```

**C-language example:**

```
P1OUT = 0b10000000;
```

This is fine if your intent is to make pin P1.7 high and all other pins on that port low, but what if you just want to control pin P1.7 while leaving the other bits’ states untouched? For this we may use bitwise operations to selectively alter specific bits within the word. For example, to set the P1.7 bit while leaving all the other P1.x bits in their original states, we may use the bitwise-OR-and-assign operator in C/C++ as shown here:

```
P1OUT |= 0b10000000; // Comment: this sets bit P1.7 (outputs a high)
```

The single “1” bit specified by 0b10000000 forces P1.7 high while the seven “0” bits leave the others unaltered, each respective pair of bits in the P1OUT register and the 0b10000000 operand being OR’d with each other. Similarly, if we wish to clear bit P1.7 (i.e. make it low) while leaving the other bits in P1OUT untouched, we may use the bitwise-AND-and-assign operator in C/C++ along with the binary value 0b01111111:

```
P1OUT &= 0b01111111; // Comment: this clears bit P1.7 (outputs a low)
```

The single “0” bit forces P1.7 low while the seven “1” bits leave the others unaltered after being respectively ANDed.
If we happen to be programming the microcontroller in assembly language, we have available to us two instructions specially designed for this purpose: bit set (\texttt{bis}) and bit clear (\texttt{bic}). The way these instructions work is to specify a value delineating which bits of the word we wish to set or clear, respectively. The “bit set” instruction is logically equivalent to a bitwise-OR-and-assign, and looks like this:

\textbf{Assembly-language example showing the “bit set” instruction:}

\begin{verbatim}
bis #0b10000000, P1OUT ; Comment = This sets bit P1.7 (outputs a high)
\end{verbatim}

Similarly, the “bit clear” instruction requires you specify which bit(s) to clear. Unlike “bit set”, the “bit clear” instruction is \textit{not} logically equivalent to any general-purpose bitwise instruction:

\textbf{Assembly-language example showing the “bit clear” instruction:}

\begin{verbatim}
bic #0b10000000, P1OUT ; Comment = This clears bit P1.7 (outputs a low)
\end{verbatim}

Even though the \texttt{bic} instruction achieves the same result as a bitwise-AND, it does so in a different manner. With the bitwise AND we needed to make the operand byte all 1’s \textit{except for} the bit-positions we wished to clear, because the nature of an AND function is that any 0 input forces the output to be 0 while any 1 input lets the other input state pass through. With the \texttt{bic} instruction our operand bit-states are inverted: we need to set those bit-positions we wish to clear and clear those bit-positions we wish to leave unaltered. In other words, \texttt{bic #0b10000000, P1OUT} in MSP430 assembly language is equivalent to \texttt{P1OUT &= 0b01111111} in C or C++. As strange as this may seem for anyone accustomed to using standard bitwise functions with “mask” operands, the existence of these two complementary functions (\texttt{bis} and \texttt{bic}) makes selective-bit programming fairly easy to do in assembly language because they both accept the same operand (mask) values for the same bits: \texttt{bis #0b10000000} sets the MSB and \texttt{bic #0b10000000} clears the MSB.

Of course, if you are programming in the Sketch language within the Energia programming environment, there are very intuitive and easy-to-use instructions for setting and clearing output pins:

\textbf{Sketch-language example:}

\begin{verbatim}
digitalWrite(15, HIGH); // Comment = This sets bit P1.7 (pin 15 on IC)
\end{verbatim}

The desired state is simply coded as either \texttt{HIGH} or \texttt{LOW}, and the pin identifier is actually the physical pin number on the integrated circuit rather than being a bit number within a port’s byte register. Suffice it to say, Sketch is designed to make microcontroller programming \textit{very easy}. 
3.4. BIT-LEVEL OUTPUT INSTRUCTIONS

A very useful programming aid included with Code Composer Studio are **header files**, each containing **symbolic constants** with easily-remembered names to help simplify the setting and clearing of bits within registers. Some of these symbols are generic while others are highly specific to the microcontroller model. By including the header file `msp430.h` the assembler/compiler knows to recognize any of these symbols written in your code as aliases for certain binary values. For example, the following “standard bits” are defined as specific hexadecimal values within every MSP430 header file:

(from `msp430g2553.h`)

```c
#define BIT0 (0x0001)
#define BIT1 (0x0002)
#define BIT2 (0x0004)
#define BIT3 (0x0008)
#define BIT4 (0x0010)
#define BIT5 (0x0020)
#define BIT6 (0x0040)
#define BIT7 (0x0080)
```

To show how this simplifies bit-writing, consider the following C-language statements which do the exact same thing:

```c
P1OUT |= 0b10000000;  // Comment: this sets bit P1.7 (outputs a high)
P1OUT |= BIT7;        // Comment: this sets bit P1.7 (outputs a high)
```

If your goal is to set the output bit P1.7, `BIT7` certainly makes more intuitive sense than `0b10000000`, which in turn makes your code easier to understand and troubleshoot.

Recall that in C we had to use a bitwise-AND function with a “mask” consisting of a single 0 and the other bits 1 in order to clear one bit in a byte. We can also use a `BIT` symbol for this purpose, by prepending the complementation operator (i.e. if `BIT7 = 0b10000000` then `~BIT7 = 0b01111111`). Here again we see two example lines of C code, each one doing exactly the same thing:

```c
P1OUT &= 0b01111111;  // Comment: this clears bit P1.7 (outputs a low)
P1OUT &= ~BIT7;       // Comment: this clears bit P1.7 (outputs a low)
```

Again, we see how the instruction using the `BIT` symbol is much easier to comprehend.
3.5 Bit-level input instructions

Reading from an input port is as simple as reading the value from the proper PxIN register, but just like output registers the input registers each hold eight bits (for upwards of eight input pins) and therefore the binary value read refers to all eight pins of that port at a time. When programming in assembly language or in C/C++, the instructions are remarkably similar. For example, suppose we wished to read all eight pin states of Port 1 (P1.0 through P1.7) into the R5 general-purpose register (assembly) or into an integer variable named switch (C/C++). The following example programs show what the code would look like in each case:

**Assembly-language example:**

```
mov P1IN, R5
```

**C-language example:**

```
switch = P1IN;
```

Often, though, what we really need is the ability to take action on the status of a single input line (i.e. just one of the bits in the PxIN register). MSP430 assembly language offers a special instruction just for this purpose, called bit-test and abbreviated bit. This instruction affects the Zero bit (among others) in the microcontroller’s Status register, which means we may take action on the state of the tested bit by using the jz (“Jump if Zero-bit is set”) instruction immediately afterward:

```
bit #0x04, P1IN ; Checks logical state of P1.2 bit
jz PIN_LOW ; If P1.2=0 the Zero bit will be set, jump to label PIN_LOW
; otherwise if P1.2=1 we just execute the next instruction
```

In other words, executing the bit instruction results in the Status register’s “Zero” bit reflecting the value of the tested bit. From there we may use any appropriate conditional jump instruction to take action on that value.

---

7Understanding the meaning of the Status register’s “Zero” bit can be difficult because the logic seems backwards. For example, if a bit-test or arithmetic operation concludes in a zero result (e.g. the tested bit is zero, or the arithmetic result is zero), then the Status register’s “Zero” bit becomes set to a value of one, flagging us that the test/arithmetic functions’ result was zero. In other words, a calculated result of zero sets the “Zero” bit (1), but a non-zero calculated result clears the “Zero” bit (0). Confusing, no?
Bit-testing in C or C++ requires the use of bitwise operators to select the desired bit(s) from the multi-bit word, similar to how we used bitwise instructions to selectively set certain bits in an output register. We may use the bitwise-AND instruction to “mask off” all bits except for the one we’re interested in, and then use the resulting value as a Boolean true/false condition within an if instruction like this:

```c
if (P1IN & 0x04) // Checks logical state of P1.2 bit
    do_something(); // If P1.2=1 then we do this...
else
    do_something_else(); // Executed if P1.2=0
```

### 3.6 MSP430G2553 architecture

The MSP430 microcontroller series uses 16-bit addressing and 16-bit data busses, and is somewhat unique for a microcontroller in that there is only one of each. This is called *Von Neumann architecture*, and it is the de facto standard for general-purpose microprocessor systems. The more traditional architecture used in microcontrollers is called *Harvard architecture*, where separate memory arrays (and address/data busses) exist for program instructions and program data. Many traditional microcontrollers use Harvard architecture, including the Intel 8051, Motorola 68HC11 and 68HC12, and Microchip PIC microcontroller family.
3.6.1 Registers

Another feature of the MSP430 series is a large number of general-purpose registers for temporarily storing numerical values. The MSP430 has sixteen 16-bit registers, the first four being dedicated to specific purposes and the last twelve open for general use:

- Register R0 = Program Counter
- Register R1 = Stack Pointer
- Register R2 = Status
- Register R3 = Constant Generator
- Register R4 through R15 = General Purpose

Most of these registers simply function as 16-bit binary values. The Program Counter register, for example, contains the memory address of the current instruction being executed. However, the Status register (R2) is one where individual bits have specific meanings of their own, acting as flags for specific conditions within the microcontroller. The following illustration shows all 16 bits of the Status register from MSB (left) to LSB (right), and the meanings of each:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 8</td>
<td>Overflow bit (V)</td>
</tr>
<tr>
<td>Bit 7</td>
<td>Bit 6</td>
</tr>
<tr>
<td>Bit 15</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

For example, the Overflow bit (bit 8) in the Status register defaults to a reset (0) state unless an arithmetic operation on signed integers results in a value exceeding the allowable range, in which case that bit automatically becomes set (1) by the processor’s arithmetic logic unit (ALU). A line of code reading this bit will be able to tell, for example, whether or not the last addition/subtraction operation yielded a valid result.

Another example in this register is the General Interrupt Enable bit (bit 3) which must be written to a set (1) state in order to permit any of the user-defined interrupts to function at all. Unlike the Overflow bit which is written by the ALU and read by the programmed code, this Interrupt bit is written by the programmed code and then read by the interrupt hardware.
3.6.2 Memory map

The MSP430G2553’s memory map is as follows:

- 8-bit Special Function Registers = 0x00 to 0x0F
- 8-bit peripherals = 0x10 to 0xFF
- 16-bit peripherals = 0x100 to 0x1FF
- General (RAM) = 0x200 to 0x3FF
- Information memory (Flash EEPROM) = 0x1000 to 0x10FF
- Code memory (Flash EEPROM) = 0xC000 to 0xFFFF
- Interrupt vectors (Flash EEPROM) = 0xFFC0 to 0xFFFF

All memory in the MSP430 is byte-oriented, i.e. one byte (eight bits) per address. 16-bit words therefore occupy two addresses, little-endian style, and always begin with an even-numbered address.
3.6.3 I/O ports

All I/O on the MSP430 microcontroller is *memory-mapped*, which means it is accessed by the same addressing and the same address bus as the rest of the system’s memory. Each 8-bit I/O port on the microcontroller has several registers dedicated to the port’s configuration. For the model MSP430G2553IN20 with 20 pins\(^8\), the two I/O ports’ memory bytes are organized as follows:

**Port 1**

- Input register = \texttt{P1IN} located at memory address 0x020
- Output register = \texttt{P1OUT} located at memory address 0x021
- Direction register = \texttt{P1DIR} located at memory address 0x022
- Interrupt Flag register = \texttt{P1IFG} located at memory address 0x023
- Interrupt Edge Detect register = \texttt{P1IES} located at memory address 0x024
- Interrupt Enable register = \texttt{P1IE} located at memory address 0x025
- Port Select register = \texttt{P1SEL} located at memory address 0x026
- Resistor Enable register = \texttt{P1REN} located at memory address 0x027
- Port Select 2 register = \texttt{P1SEL2} located at memory address 0x041

**Port 2**

- Input register = \texttt{P2IN} located at memory address 0x028
- Output register = \texttt{P2OUT} located at memory address 0x029
- Direction register = \texttt{P2DIR} located at memory address 0x02A
- Interrupt Flag register = \texttt{P2IFG} located at memory address 0x02B
- Interrupt Edge Detect register = \texttt{P2IES} located at memory address 0x02C
- Interrupt Enable register = \texttt{P2IE} located at memory address 0x02D
- Port Select register = \texttt{P2SEL} located at memory address 0x02E
- Resistor Enable register = \texttt{P2REN} located at memory address 0x02F
- Port Select 2 register = \texttt{P2SEL2} located at memory address 0x042

\(^8\)The 28-pin version of the MSP430G2553 offers a *Port 3* in addition to the standard Port 1 and Port 2.
The following illustration shows all the registers associated with the microcontroller’s pins. Each “x” refers to the port number; e.g. Port 1 registers would be P1IN, P1OUT, P1DIR, etc. while Port 2 registers would be P2IN, P2OUT, P2DIR, etc. Not every port in the MSP430 family supports interrupt inputs, which is why the upper three registers appear in grey:

For example, if we wished to read the status of pin P1.3, we would have to read bit 3 of the Input register P1IN. If we wished to output a high state on pin P2.7, we would have to “set” bit 7 of the Output register P2OUT. If we wished to output a low state on pin P1.6, we would have to “clear” bit 6 of Output register P1OUT.

Since all I/O pins on this microcontroller are bidirectional, we need a way to configure the direction of each pin individually, and this is done with the Direction register for each port. For example, if we wanted to use the pins P1.0 through P1.2 as outputs and the rest of Port 1’s pins as inputs, we would have to write the binary value 0b00000111 (or hexadecimal value 0x07) to P1DIR because the microcontroller regards any “0” bits in the Direction register to represent input and any “1” bits to represent output.

A convenient feature when using I/O pins as inputs is the provision of an internal pull-up/pull-down resistor. This eliminates the need for connecting external resistors to input pins in order to avoid indeterminate logic states. To enable a pin’s internal resistor, simply set (1) the respective bit in the PxREN register. The resistor will pull up if the respective bit in the PxOUT register is set (1) and will pull down if the PxOUT bit is cleared (0).

Pins have multiple functions in the MSP430 microcontroller series, and these are selectable by the Select and Select 2 registers. When the same bit position in both registers are cleared (0,0) that pin will function as basic digital I/O (Px.y). However, for the other three bit combinations (0,1 ; 1,0
in the Select and Select 2 registers the pin will assume other functions. The specific functions selected are model-dependent, so the manufacturer’s literature is your best guide here. Most pins default as I/O (0,0), but not all! A noteworthy example of pins defaulting to one of these “other” functions are pins P2.6 and P2.7 which default to crystal oscillator connections, necessary if the microcontroller is to use an external crystal to regulate its clock frequency. If you wish to use either of these two pins as regular I/O, you must clear those respective bits in the register. By default P2SEL has a hexadecimal value of 0xC0, but to use all Port 2 pins as general I/O we would need to make P2SEL have a value of 0x00.

Lastly, while all port pins on the microcontroller have the ability to function as inputs, some of them also have the ability to function as a special kind of input called an interrupt. Unlike a regular input pin that simply sets or clears its bit in the PxIN register depending on the logic state provided to it by an external device, an interrupt pin causes the running program to cease its current action and immediately jump to a new section of code called the ISR (Interrupt Service Routine) to perform some time-critical function\(^9\). These are maskable interrupts because you have the ability to enable or disable each one with the bit states written to register PxIE. Since these are time-critical events, what matters is the leading edge of the pulse signal sensed by the pin, and so another register (PxIES) defines whether it will be a rising edge (a low-to-high transition) or a falling edge (a high-to-low transition) that will set the interrupt flag. This microcontroller relies on the ISR code to clear the flag upon completion.

When using the MSP430G2553 microcontroller on the MSP-EXP430G2ET “LaunchPad” development board, the following I/O pins come pre-connected to some external devices. This is useful for proof-of-concept demonstration programs:

- Green LED = P1.0
- Red LED = P1.6
- Multicolor LED:
  - Red = P2.1
  - Green = P2.3
  - Blue = P2.5
- Pushbutton switch = P1.3 (pin goes low when pressed)

\(^9\) An example of an interrupt input on a microcontroller would be a timing signal coming from an external clock device or some sensor measuring an event demanding immediate attention (e.g. the signal in an automobile engine controller indicating when the engine’s crankshaft reaches the top-dead-center position once every revolution, as all other time-critical engine control events such as spark timing and fuel injection timing depend on this signal.
3.6.4 Interrupts

An interrupt is a facility within a microprocessor or microcontroller allowing it to have its normal flow of program execution interrupted by some event. When an interrupt event occurs, a bit in one of the special-function registers called an interrupt flag becomes set, causing the program to complete its current instruction execution cycle and then immediately jump to a pre-designated location in memory (called the interrupt vector). Within the interrupt vector address is stored the memory address for the start of some new code to execute. This new section of code is called the interrupt service routine (or ISR) and its purpose is to do whatever needs to be done in response to the interrupt event. The last two lines of code in the ISR terminate the interrupt by clearing the interrupt flag bit and then commanding a return to the main program.

Interrupts, therefore, are similar to subroutines in that the normal thread of execution is halted so that the processor can jump to some other location in program memory and perform a different function, then jump back to resume what it was doing before. The major difference between an interrupt and a subroutine is that while a subroutine is “called” by an instruction in the main program, an interrupt is forced by some event independent of the main program.

Interrupts are necessary when the processor must respond to highly time-critical events. An example of such an event includes incoming data received by a serial receiver from some other device. If the processor ignores incoming data and continues whatever other task its main program would have it do, data bits may be “lost” by the time the processor finishes its other task(s). Therefore, serial data reception is a common interrupt source for microprocessors and microcontrollers alike: incoming serial data causes an interrupt, which halts the main program’s execution and forces the processor to “service” the incoming data to ensure none of it is lost. Other examples include internally-generated interrupts from microcontroller timers for the purpose of ensuring certain calculations are performed at regular intervals (e.g. integral and derivative calculations for a PID control algorithm).

Interrupts are one of those areas of microprocessor/microcontroller architecture that tends to vary widely between models and manufacturers. For this reason it should be understood that the rest of this subsection should be read as a description specific to the Texas Instruments MSP430 microcontrollers, and to not generalize too much to other devices.

In the MSP430 microcontroller family, the highest addresses in Flash EEPROM memory are reserved as interrupt vectors: 0xFFC0 through 0xFFFF, ordered in terms of priority in case multiple interrupt events occur simultaneously. Address 0xFFFE, for example, is the highest priority interrupt vector and it is reserved for resets of the microcontroller (e.g. power-up, external reset input, illegal instruction fetch, etc.). As we can see in this example, sometimes the triggering event for an interrupt is external (e.g. reset pin) while other times it may be internally generated by the microcontroller (e.g. illegal instruction).

---

10 Other differences exist as well between subroutines and interrupts, including the fact that during an interrupt both the Program Counter and the Status Register are automatically pushed to the stack (in that order), and both are automatically popped off the stack when returning from the interrupt service routine. For subroutines only the Program Counter gets pushed to and popped off the stack – the Status Register remains unaffected in case the subroutine requires the last status values to properly function. With interrupts the MSP430 assumes present Status Register bit conditions are irrelevant to the interrupt event and pushes the old Status Register to the stack before clearing all Status Register bits prior to executing the ISR. This also has the effect of clearing the GIE bit (General Interrupt Enable) in the Status Register which means the ISR itself cannot be interrupted by any other interrupt events regardless of priority.
Some interrupts are *maskable* which means they may be enabled or disabled by programming certain bit states in the Special Function Registers. The P1IE and P2IE Interrupt Enable registers are an example of this, where bit states in those registers enable (1) or disable (0) respective I/O pins as external interrupt signal sources. A *non-maskable interrupt* (or NMI), by contrast, cannot be disabled by the user.

When any interrupt event occurs, a *flag* bit becomes set which indicates to the processor that it must reference the interrupt vector and then jump to the appropriate ISR. This flag bit remains set until the user’s program clears it. This is why all ISRs should have a bit-clear instruction\(^\text{11}\) somewhere in their code.

Port-driven interrupts in the MSP430 series are *edge-triggered* just like the “clock” input on a flip-flop. Additional registers exist for the user to specify whether the triggering transition will be a rising edge (low-to-high) or falling edge (high-to-low): P1IES for Port 1 bits and P2IES for Port 2 bits.

When writing programs to use maskable interrupts, one must enable the desired interrupt and also enable the General Interrupt Enable (GIE) bit before the interrupt will be functional. It is also good practice to initialize all enabled interrupt flags to their zero (0) states, just in case one or more of them happened to become set during power-up.

\(^{11}\text{MSP430 assembly language conveniently provides a “bit clear” \texttt{bic} instruction. For example, clearing the interrupt flag for Port 1 bit 3 could be done with the assembly instruction \texttt{bic #0x08, P1IFG}. However, when programming in C or C++ there is no comparable “bit clear” instruction, and so the usual technique is to use a bitwise-AND-assign operator. For example, clearing the interrupt flag for Port 1 bit 3 could be done with the C/C++ instruction \texttt{P1IFG &= 0xF7;} or \texttt{P1IFG &= ~BIT3;}.}\)
The ISR must also be specified by the user’s program in order for an interrupt to be useful. The Code Composer Studio assembler uses sections to define specific regions of code that will be loaded into the microcontroller’s memory. When programming in either C or C++, interrupt service routines are selected using the #pragma directive, each vector having a specific name. These assembly-language sections and C/C++ name are listed here in descending order of priority for the model MSP430G2553:

<table>
<thead>
<tr>
<th>Interrupt source</th>
<th>Vector address</th>
<th>Assembly section</th>
<th>C/C++ name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0xFFF4</td>
<td>.reset</td>
<td>RESET_VECTOR</td>
</tr>
<tr>
<td>Non-maskable</td>
<td>0xFFF3</td>
<td>.int14</td>
<td>NMI_VECTOR</td>
</tr>
<tr>
<td>TIMER1_A0</td>
<td>0xFFF2</td>
<td>.int13</td>
<td>TIMER1_A0_VECTOR</td>
</tr>
<tr>
<td>TIMER1_A1</td>
<td>0xFFF1</td>
<td>.int12</td>
<td>TIMER1_A1_VECTOR</td>
</tr>
<tr>
<td>Comparator A</td>
<td>0xFFF0</td>
<td>.int11</td>
<td>COMPARATORA_VECTOR</td>
</tr>
<tr>
<td>Watchdog timer</td>
<td>0xFFF9</td>
<td>.int10</td>
<td>WDT_VECTOR</td>
</tr>
<tr>
<td>TIMER0_A0</td>
<td>0xFFF8</td>
<td>.int09</td>
<td>TIMER0_A0_VECTOR</td>
</tr>
<tr>
<td>TIMER0_A1</td>
<td>0xFFF7</td>
<td>.int08</td>
<td>TIMER0_A1_VECTOR</td>
</tr>
<tr>
<td>USCIAB0RX</td>
<td>0xFFF6</td>
<td>.int07</td>
<td>USCIAB0RX_VECTOR</td>
</tr>
<tr>
<td>USCIAB0TX</td>
<td>0xFFF5</td>
<td>.int06</td>
<td>USCIAB0TX_VECTOR</td>
</tr>
<tr>
<td>ADC10</td>
<td>0xFFF4</td>
<td>.int05</td>
<td>ADC10_VECTOR</td>
</tr>
<tr>
<td>Port 2 I/O</td>
<td>0xFFF3</td>
<td>.int03</td>
<td>PORT2_VECTOR</td>
</tr>
<tr>
<td>Port 1 I/O</td>
<td>0xFFF2</td>
<td>.int02</td>
<td>PORT1_VECTOR</td>
</tr>
</tbody>
</table>
If we examine a very simple assembly-language program for the MSP430G2553, we see how the reset interrupt vector is configured:

```
 cdecls C, LIST, "msp430.h"
def RESET
text
retain
retainrefs
RESET  mov.w #__STACK_END, SP  ; Initialize stack pointer
StopWDT  mov.w #WDTPW|WDTHOLD, &WDTCTL  ; Stop watchdog timer

;================================= Code Begin =================================
MAIN
  mov #0x05, R4
  add #0x06, R4
;================================== Code End ==================================

.global __STACK_END
.sect .stack
.sect ".reset"
.short RESET
```

The last two lines of this program configure the RESET interrupt vector. First, the appropriate section is declared (.sect ".reset") for this particular interrupt vector, and then the label for the ISR is specified (.short RESET). When any of the interrupt flags belonging to the reset function are set, the vector at 0xFFFE points to the memory address resolved to the label RESET by the assembler, which in turn executes the mov.w instruction to initialize the stack pointer.
An example of a port-driven interrupt is seen in the following assembly-language program, written to toggle the state of output P1.0 every time input P1.3 transitions from a high logic level to a low logic level:

```
.cdecls C,LIST,"msp430.h"
def RESET
text
retain
retainrefs

RESET    mov.w    #-_STACK_END,SP
StopWDT   mov.w    #WDTPW|WDTHOLD,&WDTCTL

;================================ BEGIN CODE =================================

SETUP
    mov #0xF7, P1DIR ; P1.3 as input, all others outputs
    mov #0x08, P1OUT ; Begin with all outputs low
    mov #0x08, P1IE ; Enable interrupt on P1.3
    bis #0x08, P1IES ; Interrupt on high-to-low transition
    bic #0x08, P1IFG ; Clear any pending P1.3 interrupt flag
    bis #0x08, SR ; Set the General Interrupt Enable (GIE) bit

LOOP ; A useless loop -- does absolutely nothing!
    jmp LOOP

P1_ISR
    xor #0x01, P1OUT ; Toggle P1.0 bit
    bic #0x08, P1IFG ; Clear the P1.3 interrupt flag
    reti ; Return from ISR

;================================ END CODE =================================

.global __STACK_END
.sect .stack
.sect ".reset"
.short RESET
.sect ".int02" ; .int02 = Port 1 interrupt vector section
            ; (shared by all P1.x pins)
.short P1_ISR
```

In addition to the obligatory reset interrupt assignment, we also have the interrupt vector for Port 1 assigned to the address resolved to label P1_ISR by the .sect ".int02" and .short P1_ISR directives. Alternatively, we could have used the single directive .intvec ".int02", P1_ISR to do the same.
3.6.5 Watchdog timer

A *watchdog timer* is any electronic timer designed to monitor another system for indications of halting. In the MSP430 series of microcontrollers, the built-in watchdog timer function is a 16-bit counter circuit inside of the microcontroller driven by the clock signal, designed to issue a general system interrupt at the highest level to reset the program if the watchdog timer itself is not periodically reset to zero before reaching the end of its count by the running program. The notion is, if the programmed code happens to “hang” for any reason (e.g. a programming error, an infinite loop, an inordinate delay, etc.) it is better for the microcontroller to re-start itself than to remain “frozen” and inoperative.

This is a safety feature useful for any microcontroller controlling a critical system (e.g. airbag deployment in an automobile), but an annoyance to students, hobbyists, and developers just trying to get their code to function as they want it. For this reason the MSP430’s watchdog timer may be disabled by setting the appropriate bit in its control register, that 16-bit register named $\text{WDTCTL}$. Some of the more important bits within the $\text{WDTCTL}$ register are defined as follows:

- **Bits 15-8 = Watchdog timer register password**
- **Bit 7 = Watchdog timer hold (stops timer when set to 1)**
- **Bit 3 = Watchdog timer counter clear (resets counter value to zero when this bit is set to 1)**
- **Bit 2 = Watchdog timer clock source select (0 = Sub-Main Clock $\text{SMCLK}$, 1 = Auxiliary Clock $\text{ACLK}$)**
- **Bits 1-0 = Watchdog timer interval (00 = clock/32768 ; 01 = clock/8192 ; 10 = clock/512 ; 11 = clock/64)**

Any time a bit state changes in the $\text{WDTCTL}$ register, it must be accompanied by the writing of a “password” value of 0x5A to the upper eight bits of the register. This validates the other bit-changes as being valid. If one or more bits get changed in the $\text{WDTCTL}$ register without the accompanying password written to the most-significant eight bits, the microcontroller gets reset as though the watchdog timer has timed out. Immediately after a $\text{WDTCTL}$ register write operation, those password bits (15 through 8) default to a value of 0x69.

To anyone accustomed to the use of passwords to authenticate human users on a computer system, this use of a password to “protect” an internal microcontroller register may seem strange. After all, who can even access this register once the microcontroller has been programmed, and also how difficult would it be for a programmer to “hack” into this register when there is only one proper password and it’s given to you right in the instruction manual? The purpose, though, is not to guard against improper human intrusion, but rather to guard against accidental overwrites in memory that could result from any number of causes such as *stack overflow*. If the purpose of the watchdog timer is to provide a fail-safe recovery method for a halted or otherwise errant program, the last thing you would want is for this protective feature to become accidently disabled by illegitimate write operations made by that same errant program.
Many simple MSP430 programs begin with a line of code near the very beginning of the source file disabling the watchdog timer prior to any other instructions being executed. Take these two examples, one written in assembly language and another in C:

**Assembly code listing**

```
mov.w #WDTPW | WDTHOLD, &WDTCTL
```

**C code listing**

```
WDTCTL = WDTPW | WDTHOLD;
```

The symbols `WDTPW` (WatchDog Timer PassWord) and `WDTHOLD` (WatchDog Timer HOLD) are both defined in the header file for the particular model of MSP430 microcontroller being programmed, equivalent to `0x5A00` and `0x0080`, respectively. The bitwise-OR operator (|) combines the bit-states of these two symbols into `0x5A80`. This means the following code examples are exactly equivalent:

**Assembly code listing**

```
mov.w #0x5A80, &WDTCTL
```

**C code listing**

```
WDTCTL = 0x5A80;
```

However, using these symbolic constants instead of obscure hexadecimal (or binary!) number values makes the program easier for human readers to interpret.
To actually use the watchdog timer (and not just disable it), a line of code must be inserted into your program at such a location that it should be periodically executed at a time interval shorter than the time it takes for the watchdog timer’s counter to “time out”. This line of code is very similar to the lines shown previously to disable the watchdog timer, except setting the “watchdog timer counter clear” bit (WDTCNTCL) rather than the “Watchdog timer hold” bit (WDTHOLD). Examples in both assembly and C follow:

**Assembly code listing**

```
mov.w #WDTPW | WDTCNTCL, &WDTCTL
```

**C code listing**

```
WDTCTL = WDTPW | WDTCNTCL;
```

### 3.7 MSP430G2553 auxiliary functions

General-purpose microprocessors require an array of supporting hardware to function, including ROM, RAM, I/O drivers, clock pulse sources, etc. Microcontrollers, on the other hand, include most if not all of these functions within a single integrated circuit in order to present a “system-on-a-chip” for minimal-hardware designs. As a result, it is relatively simple to build and use microcontrollers for simple electronic projects, whereas a microprocessor demands a much greater level of design and PCB layout work just to be functional.

The MSP430G2553 microcontroller is no exception, and as one would expect it sports a range of useful auxiliary functions to make it extremely capable as a stand-alone IC. The following subsections summarize these auxiliary features.
3.7. MSP430G2553 AUXILIARY FUNCTIONS

3.7.1 Clocks

All microprocessors require a clock signal to synchronize the operations necessary to fetch and execute instructions stored in memory. General-purpose microprocessors rely on external oscillator circuits for their clock signals, but microcontrollers typically offer internal clock signal sources to relieve the end-user from having to specify and/or design an oscillator to drive the chip.

The MSP430 series offers multiple oscillator sources, and translates these raw pulse sources into multiple clock signals used internally to drive the CPU and peripheral functions:

Oscillator options:

- **DCOCLK** = Digitally Controlled Oscillator (internal to the microcontroller)
- **VLOCLK** = Very Low Power Oscillator (internal to the microcontroller)
- **LFXT1CLK** = Low-Frequency Crystal Oscillator (external to the microcontroller)
- **XT2CLK** = High-Frequency Crystal Oscillator (external to the microcontroller)

Available clock signals:

- **MCLK** = Master Clock, *drives the CPU* and is sourced by any one of the four oscillators with options for 1/2/4/8 frequency division (all software-selectable)
- **SMCLK** = Sub-Main Clock, sourced by any one of the four oscillators with options for 1/2/4/8 frequency division (all software-selectable)
- **ACLK** = Auxiliary Clock, sourced by either LFXT1CLK or VLOCLK (all software-selectable)

A set of Basic Clock System Control registers (BCSCTL1 through BCSCTL3) establish oscillator sources for each of the three clock signals, as well as frequency ranges for the crystal-driven sources.

The Master Clock frequency is obviously important to the speed of program execution, since it is what drives the processor. High speed is good when the program must respond quickly to external events. However, low speed conserves power\(^\text{12}\), which is often a prime consideration for microcontrollers when used in battery-powered and other limited-power applications.

Various auxiliary functions within the microcontroller also require a reliable clock signal, and given the software-selectability of these signals it not only possible but very commonplace to have peripheral functions running on a different clock frequency than the CPU. Such functions include serial data communication, timers, and analog-digital converters.

\(^\text{12}\)The fundamental reason for this is the CMOS circuitry from which modern microprocessors and microcontrollers are fabricated. Unlike bipolar TTL logic which tends to dissipate the same amount of electrical power in heat regardless of operating frequency, MOSFET logic circuits draw most of their current during transitional states. In static conditions a CMOS logic circuit consumes negligible power, but power demand rises with frequency because each complementary MOSFET stage passes some current from source to drain when both transistors are in their partially-on modes. The more often these transistors must transition between fully-on and fully-off states (in either direction), the more power they dissipate. This is why *overclocking* a microprocessor is usually limited by the chip’s cooling system: its ability to reject the additional heat that comes with higher operating frequency.
3.7.2 General timers

3.7.3 Watchdog timer

3.7.4 Analog-Digital conversion

3.7.5 Analog comparators

3.7.6 Serial data interfaces

3.8 CCS assembly-language programming

Texas Instruments’ Code Composer Studio integrated development environment (IDE) offers full support for assembly-language programming of the MSP430 microcontroller series. The example screenshots shown here were taken with version 10 of Code Composer Studio.

When starting a new assembly project, first select “CCS Project” under the “New” project drop-down option:

It is very important to specify that you wish to start a new project as opposed to simply opening a new file, as a “project” entails not just the source code files but also other files (e.g. headers, Makefiles) needed for compilation.
Next, in the “New CCS Project” window you will need to select the correct “target” (i.e. the part number of your microcontroller), determine the type of programming connection (USB1 is default), enter a project name, and select “Empty Assembly-only Project” from the list of options. Note that assembly language is not the default project type, as most modern microcontroller projects are written in C or C++:

A common source of trouble for people new to microcontroller programming is failing to select the proper target when configuring a new project. Since development software such as Code Composer Studio is designed to output executable code for a wide range of microcontroller models, each having its own limitations and programming idiosyncrasies, the user must specify which microcontroller model the code will be sent to and run on. Failing to get this step correct will result in code that “compiles” but will not install correctly on the microcontroller!
This will present a view of a “template” assembly language source file with all the default declarations needed for most assembly-language programs:

Two different options exist for compilation and loading of your program into the microcontroller under the “hammer” icon: Debug and Release. Selecting “debug” will assemble the source code in preparation for a live debugging session where you may single-step the program’s execution to verify its proper operation while viewing register contents, the memory map, etc. Selecting “release” will assemble the source code and ready the object code for loading into the microcontroller with no debugging facilities activated: i.e. the code will be assembled in preparation to run as a final “release” version:

In order to load the assembled object code into the microcontroller, you must click on either the insect icon (debug) or the folder icon containing curly-braces (release).
3.9 CCS C-language programming

Texas Instruments’ Code Composer Studio integrated development environment (IDE) offers full support for both C and C++ language programming of the MSP430 microcontroller series. The example screenshots shown here were taken with version 10 of Code Composer Studio, showing an example program written in C.

When starting a new assembly project, first select “CCS Project” under the “New” project drop-down option:

It is very important to specify that you wish to start a new project as opposed to simply opening a new file, as a “project” entails not just the source code files but also other files (e.g. headers, Makefiles) needed for compilation.
Next, in the “New CCS Project” window you will need to choose the proper “target” (i.e. microcontroller part number), choose a suitable programming connection (USB1 is the default), enter a project name, and select “Empty Project (with main.c)” from the list of options:

A common source of trouble for people new to microcontroller programming is failing to select the proper target when configuring a new project. Since development software such as Code Composer Studio is designed to output executable code for a wide range of microcontroller models, each having its own limitations and programming idiosyncrasies, the user must specify which microcontroller model the code will be sent to and run on. Failing to get this step correct will result in code that “compiles” but will not install correctly on the microcontroller!

This will present a view of a “template” main.c source file with all the default declarations needed for most C-language programs:

As you can see, there isn’t much to this template source file. A single #include directive instructs the compiler to include the contents of the msp430.h header file for compilation, and the main() function helpfully contains a statement disabling the default watchdog timer within the MSP430 microcontroller.
Two different options exist for compilation and loading of your program into the microcontroller under the “hammer” icon: **Debug** and **Release**. Selecting “debug” will compile and link the source code in preparation for a live debugging session where you may single-step the program's execution to verify its proper operation while viewing register contents, the memory map, etc. Selecting “release” will compile and link the source code and ready the object code for loading into the microcontroller with no debugging facilities activated: i.e. the code will be assembled in preparation to run as a final “release” version:

![Image](image.png)

In order to load the assembled object code into the microcontroller, you must click on either the insect icon (debug) or the folder icon containing curly-braces (release).
In an effort to make programming the MSP430 series easy for beginners, the developers included a very simple “Blink The LED” program as a New CCS Project option rather than a mostly-blank main.c template file. This may be selected in lieu of the “Empty Project (with main.c) option shown earlier:

After selecting this option and opening the new project, we see the source code as shown here:

As with the Empty Project option, this main.c source file contains the obligatory header inclusion directive as well as the watchdog timer disabling statement. New additions to the source code include a statement setting the first pin of port 1 to be an output, a variable declaration for an integer named i, a while() loop set to execute without end, a bitwise Exclusive-OR function toggling the state of the P1.0 bit, and a simple for() loop creating a time delay by counting backwards from 10000 to zero. This code is designed to toggle the P1.0 output bit, wait for a short time period, then repeat.
The comment following the variable declaration line deserves further elaboration. When a high-level language such as C or C++ is “compiled” into machine language, there is always more than one practical way to do that translation. Often times the most direct and literal compilation does not yield the most compact machine code, and so all compilers offer a feature called *optimization* whereby the compiled code may be streamlined for some purpose such as minimum memory space or maximum speed. Microcontrollers are limited in memory capacity for the obvious reason that their primary (and often only) memory is on-board the MCU chip, which means compilers designed to compile executable code for microcontrollers often default to optimization favoring minimum memory usage.

Optimization is unknown to assembly-language programming, because the point of assembly-language programming is to mimic the machine code which will be written to memory. The price we pay for programming in assembly language is having to think like the microcontroller, unable to perform complex operations with single statements. Languages such as C and C++ offer vastly more sophisticated capabilities, but at the expense of relatively lengthy machine code and slower execution times.

Code Composer Studio offers some rather aggressive optimization features, one of them being to *eliminate* code it deems useless or redundant. In the default level of optimization, the compiler will evaluate the time-delay for() loop and determine that it performs no “useful” function, and *omit it entirely from the program!* This, of course, would leave the program with no time delay at all which would cause the LED connected to pin P1.0 to blink on and off as fast as it was able rather than at a slow blink pace. This is why the author of this example program forced optimization off by declaring the integer variable *i* in a strange manner. It is also possible to select optimization options prior to clicking on the “build” (compile) icon, but forcing optimization off within the source code makes it so the program is ready to compile in the simplest way with no further actions needed by the new user.
3.10 Energia programming

Energia is an independent project intended to provide Arduino-like programming functionality to the Texas Instruments MSP430 microcontroller series. It integrates with Code Composer Studio, enabling the developer to write code in the Arduino sketch language rather than C, C++, or assembly.

Since Energia is not an official part of TI’s Code Composer Studio software, it must be installed separately on the host computer and then invoked by selecting the “Other” category for New projects. The following example screenshots were taken with Code Composer Studio version 10:

---

13 When you download Energia, it is a simple .zip archive which must be extracted and placed in a directory on your computer’s hard drive. Upon first invoking Energia Sketch as a project option within Code Composer Studio, you will be prompted to provide the path to this Energia directory.

14 Alternatively, going to the Project drop-down menu and selecting the “New Energia Sketch” option.
Among the setup “Wizard” options in the “Other” category you will find “Energia Sketch” listed:
Selecting this option presents a host of pre-written example programs in the Sketch language, as well as the option for an “Empty Sketch” to allow programming starting with a blank page. In this “New Energia Sketch” window you must select the correct microcontroller development board so Code Composer Studio will be aware of the proper target device, just as in assembly-language and C-language project setup:

A common source of trouble for people new to microcontroller programming is failing to select the proper target (board, in the case of Energia Sketch) when configuring a new project. Since development software such as Code Composer Studio is designed to output executable code for a wide range of microcontroller models, each having its own limitations and programming idiosyncrasies, the user must specify which microcontroller model the code will be sent to and run on. Failing to get this step correct will result in code that “compiles” but will not install correctly on the microcontroller!
3.11 Debugging tools

Code Composer Studio (CCS), like all modern integrated development environments (IDEs), offers a rich set of debugging tools useful for diagnosing errors in your written code. When debugging code written to the memory of a microcontroller, a second programmable device functions as the debugging host to provide the developer (you) with access to the microcontroller’s internal states. With this host device acting as an interface between the microcontroller and the IDE, you are able to single-step the program (i.e. execute line-by-line pausing each time) as well as monitor the states of the microcontroller’s internal registers, program counter, memory map, etc. If you are programming your MSP430 microcontroller using the LaunchPad development kit, the debugging host is another microcontroller (surface-mount) located on the printed circuit board that is pre-programmed for this task.

To select the “debug” option when assembling/compiling your code, first choose “Debug” under the “Build” (hammer-symbol) icon:

Then to fully build and download the debuggable code to your microcontroller, click on the bug-shaped icon (note the large arrow symbol I added to the screenshot pointing to this icon in the CCS toolbar):
Next, a window will appear showing the debugger configuration in progress. This typically requires just a few seconds of time:

Meanwhile, the CCS workspace switches automatically to the “debug” view, and within the “Debug” pane you will see the name of your project followed by “Code Composer Studio – Device Debugging”. If this does not appear, it generally means there is an old debugging session still active that was not properly shut down. Closing and re-starting Code Composer Studio will remedy this state.
Once the debugging session finally starts you will see additional text within the “Debug” pane declaring the USB connection to the microcontroller and underneath that text declaring the source code listing that will be animated during the session. One line in the source code will be color-highlighted to show the next step in execution, and a set of buttons in the toolbar will be available that were not before: a red square (stop) and yellow arrows (“step into” and “step over”). Clicking on either yellow arrow causes the program to “single-step” one instruction at a time in the source code listing, allowing you to witness the program’s execution at a pace you manually decide. A large arrow symbol (added to the screenshot) points to the “step into” icon:
As useful as it is to be able to “single-step” the microcontroller program, this functionality becomes even more useful when you combine it with live viewing of the microcontroller’s internal state. Under the “View” menu option you may select multiple features to add to the workspace, including viewing registers (shown below):
3.11. DEBUGGING TOOLS

Clicking on this option will open a new pane in the workspace with a view of all the microcontroller’s internal registers, allowing you to view their states as you single-step the program. Here we see the Register pane open, with a list of registers shown:
Expanding the “Core Registers” item in the registers list shows all the central registers to the microcontroller’s CPU, including the Program Counter (PC), Stack Pointer (SP), Status Register (SR), and general-purpose registers (R3 through R15):
Clicking the “step into” arrow button once advances the program’s execution from the **RESET** line in the assembly-language source code listing to the **StopWDT** line, and with this advance we see the Program Counter register has incremented from 0xC000 to its present value of 0xC004:

This Program Counter value change is reflected in the fact that it shows up as yellow-highlighted text within the “Registers” viewing pane.

It is important to realize that the color-highlighted line is the instruction that will be executed on the next click of the “step into” button. For example, they way we see it in this screenshot, the microcontroller has just executed the **RESET** instruction but it has not yet executed the **StopWDT** instruction.
After the StopWDT instruction executes, the Program Counter advances to 0xC00A and the color highlighting rests on the move (mov) instruction awaiting execution at the next “step into” click.

An alternative to “stepping into” an instruction in debug mode is to “step over” the instruction. This “step over” option skips the instruction, advancing the Program Counter without actually executing that instruction, which is useful if you are single-stepping a program and wish to see what will happen if the instruction’s execution does not happen.
When the `mov` instruction executes, it moves the eight-bit value `0x22` into register R5:

As with the incrementing Program Counter, the “Registers” viewing pane shows all changed values color-highlighted in yellow. Previously, register R5 contained the value `0x0000` but now it contains `0x0022` due to the `mov` instruction’s action.

The color-highlighted line in the source code listing tells us the subtract (`sub`) instruction will be next to execute when we click again on the “step into” button.
Following the \texttt{sub} instruction’s execution, we see a few changes in the register states. The Program Counter of course incremented, but along with that we see the result of the mathematical subtraction now stored in register R5 ($0x0022$ minus $0x0023$ equals negative one, which in 16-bit signed binary form is $1111\ 1111\ 1111\ 1111$ or $0xFFFF$ in hexadecimal):

Another change is the Status Register (SR), which has updated following the arithmetic subtraction operation. This is very typical for microprocessors, that the status register gets refreshed following any mathematical or logical operation. Bits within the Status Register will indicate such things as whether the result was zero, whether an overflow occurred, etc.

Something else changed if you look at the source code listing. Instead of showing \texttt{main.asm} as before, a new tab has appeared for a listing named \texttt{isr\_trap.asm}. The reason for this change is that our simple subtraction program came to a halt after executing the \texttt{sub} instruction, and this triggered an automatic interrupt. In a real program we would have an infinite loop to cause the program to execute continually. For this simple demonstration we need not concern ourselves with practicality,
and this sudden halt also serves to show how the debugging session directs your attention to such an interruption.

Another viewing option we have within the “Debug” environment is the “Memory Browser” showing a *hex dump* view of the microcontroller’s memory contents. In order to demonstrate how this works, I edited the assembly-language program to store and subtract to the absolute address 0x0200 rather than register R5, then reassembled it and started up a new debugging session:

This new program is now ready for single-step execution.
After the `mov` instruction executes, we see the moved value 0x22 now stored in memory address 0x0200, appearing as red text in the “Memory Browser” pane:
Following the `sub` instruction’s execution, we see the result (0xFFFF) stored in memory address 0x0200 just where it should be:
A useful feature during debugging sessions is the ability to manually overwrite data between program steps. Here we see another debugging session with the same subtraction program writing to memory address 0x0200, following execution of the `mov` instruction. However, instead of letting address 0x0200 contain the moved value of 0x22, we instead double-click on that cell of the hex dump to edit that value:
Here we see the value 0x25 residing in memory address 0x0200 following our hand-edit of that data:

Note that we have not yet executed the `sub` instruction. This will happen when we next click on the “step into” button.
Now we see the result 0x0002 stored in memory address 0x0200, the proper result of subtracting 0x23 from 0x25:

One final note: it is important to properly close a debugging session when you are finished, prior to editing the source code or doing anything else. A good way to do this is to right-click on the text within the “Debug” pane and select the option to “Terminate and Remove”. When this is done, the descriptive text within the “Debug” pane will disappear. If this is not done properly, the unterminated debugging session will still be running when you go to download a new version of your program or try to start another debugging session, and will interfere with your new intent.
Chapter 4

Derivations and Technical References

This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.
4.1 Introduction to assembly language programming

Microprocessors only understand *machine code* – instructions and data encoded as binary values, often written in hexadecimal “shorthand” form for better human-readability – but even in hexadecimal these codes are non-intuitive and confusing for human programmers to manage. *Assembly code* improves upon this situation by representing each machine-code instruction as a word-abbreviation called a *mnemonic*. The convenience of mnemonics, though, comes at a price: since the mnemonics themselves are nonsense to the microprocessor, a special software package called an *assembler* must be used to translate these mnemonics into machine code that the microprocessor can understand. Just as microprocessors each have their own unique instruction set defining which codes perform which functions, assembler software has its own unique “vocabulary” and “grammar” one must abide by in order to write functioning programs. Assembly programming, therefore, is its own *language* and like all languages has specific rules.

Furthermore, assembly-language lacks the strict standardization found in higher-level programming languages such as C, C++, Java, and Python. Just as machine-code programming is specific to the model of microprocessor, assembly-language programming rules are often specific to the assembler software version, which in turn is often closely coupled to the microprocessor model. It is thus impossible to write a generic tutorial on assembly programming, just as it is impossible to write a generic tutorial on machine-code programming. What we will explore here are features of assembly code common to *most* assemblers.

We will use a specific hardware application as the foundation for this lesson on assembly language programming, in order to have a practical context for understanding what the program does and how it works. This means the examples given here will not work with any microprocessor other than the system described here, but that is okay. Many of the principles learned here find general application to other systems.

Our hypothetical computer is shown below, based on a Motorola model 6502 8-bit microprocessor. A single output port mapped to memory address `0x4000` provides the means for our program to turn LEDs on and off, by writing bit-states to that byte located at `0x4000`. Another port at address `0x0401` provides inputs, where our program may read bit-states of the byte stored there to detect logic signals applied to those pins by external circuitry (not shown). ROM begins at address `0x8000` and extends through address `0xFFFF`. RAM begins at address `0x0000` and extends through address `0x3FFF`:

The 6502 provides an Accumulator register plus two general-purpose registers (named X and Y) for temporary data storage, each one eight bits wide.
4.1. INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

4.1.1 Machine code to blink an LED

Suppose we wish to make the LED connected to output pin P0 on the computer blink on and off. This means writing a 1 and then a 0 to bit 0 of the byte located at \(0x4000\) while clearing (making zero) all the other bits at that address. Our program will execute the following steps:

1. Write the binary value \(0b00000001\) to address \(0x4000\)
2. Toggle the least-significant bit of that byte in address \(0x4000\)
3. Go back to second instruction and repeat indefinitely

The same steps, using instructions available in the 6502’s instruction set:

1. **Load** the binary value \(0b00000001\) into the Accumulator register
2. **Store** the Accumulator’s value to address \(0x4000\)
3. Apply the **Exclusive-OR** function to the LSB in the Accumulator using the **mask**\(^1\) \(0b00000001\).
4. **Jump** to the second instruction and repeat indefinitely

Researching opcodes and operand formats for the model 6502 microprocessor, we find the following:

- The opcode for “Load Accumulator with immediate value” is \(0xA9\) followed by the desired value to load (\(0x01\))
- The opcode for “Store Accumulator value to absolute memory address” is \(0x8D\) followed by two bytes specifying the destination address in little-endian order (low byte first, high byte last: \(00\ 40\) for address \(0x4000\))
- The opcode for “Exclusive-OR with immediate value” is \(0x49\) followed by the mask value (\(0x01\))
- The opcode for “Unconditional Jump to absolute memory address” is \(0x4C\) followed by the target address in little-endian order

\(^1\)In programming, a **mask** is a set of bits intended to apply a bit-wise logical operation to bits within a larger word of data. Here, our mask of \(0b00000001\) means the LSB will be XOR’d with 1 (i.e. toggled, to make it switch states from whatever it was before to the opposite of that) while all other bits within the Accumulator’s 8-bit word will be XOR’d with 0 (i.e. left alone).
If we write all these opcodes and operands in order, one line per complete instruction, we get the following *hand-assembled* machine code:

```
A9 01
8D 00 40
49 01
4C ?? ??
```

The question-marks are there in our code because we need to determine where our program will begin in the computer’s ROM memory space in order to know which address we need to “jump” to in the last instruction. Let’s assume our program starts at the very first address in ROM (0x8000) and re-write the program showing the starting address of each line:

```
8000 A9 01
8002 8D 00 40
8005 49 01
8007 4C ?? ??
```

Recall that the purpose of our “Jump” instruction (last line, 0x4C) was to go back to the “Store Accumulator” instruction (0x8D) so that our recently XOR’d data will be re-written to the output port at address 0x4000. Therefore, the address we need to jump to is 0x8002. Knowing this, we may edit our machine code listing to include this address in the last instruction, in “little-endian” byte order because the model 6502 microprocessor happens to be a little-endian machine:

```
8000 A9 01
8002 8D 00 40
8005 49 01
8007 4C 02 80
```

If we were to write this program to the computer’s ROM and then read it back to display in conventional “hex dump” format, it would look like this:

```
8000 A9 01 8D 00 40 49 01 4C 02 80
```

If the ROM IC(s) in our computer are socketed and therefore easily removed for programming by an external device, we could use a PROM programmer\(^3\) to write this short program into a programmable ROM memory chip and then re-insert it into our computer’s board to run.

\(^2\)This is beginning to resemble the common “hex dump” memory display format, except that each line of text is limited to just one instruction

\(^3\)Commercially-available PROM programming tools consist of a unit connected to a personal computer with a zero-insertion-force (ZIF) IC socket to facilitate easy plugging and unplugging of memory ICs. Software provided
4.1.2 Assembly code to blink an LED

Now that we have seen the “hand-assembly” method of creating a simple LED-blinking program for our 6502-based computer, let us explore how we could do the same using assembly language. Starting with the original program specification telling us what we need the computer to do:

1. Write the binary value 0b00000001 to address 0x4000
2. Toggle the least-significant bit of that byte in address 0x4000
3. Go back to second instruction and repeat indefinitely

Next, we refer to instructions available in the 6502’s instruction set, but this time we write the mnemonic abbreviations given in that instruction set instead of opcodes:

1. LDA value 0b00000001
2. STA to 0x4000
3. EOR mask 0b00000001
4. JMP to second instruction

We are almost done with our assembly-language program! All we need to do now is write it using the proper syntax\textsuperscript{4} expected by our assembler software, being sure to include a directive\textsuperscript{5} to the assembler to begin at address 0x8000.

```
.LORG $8000
LDA #$01
LOOP
STA $4000
EOR #$01
JMP LOOP
```

This program you see above is completely functioning, ready to be assembled into machine code and written to the computer’s ROM.

\textsuperscript{4}We are assuming here that our assembler does not understand binary notation and expects all numerical values to be expressed in hexadecimal instead.

\textsuperscript{5}A “directive” is an instruction given not to the microprocessor, but rather to the assembler software. It tells the assembler to translate the assembly “source” code into machine code in some particular manner.
Optional *comments* help make our code easier to read:

```assembly
.ORG $8000 ; Begin at address 0x8000
LDA #$01 ; Load 0x01 into Accumulator
LOOP ; Mark the beginning of the loop
STA $4000 ; Store Accumulator value to Output port
EOR #$01 ; XOR the least-significant bit
JMP LOOP ; Jump to the beginning of the loop
```

Take note of some important details of this assembly-language program:

- Order of execution is left-to-right, top-to-bottom in the same order as reading English-language text.
- Any left-justified text (e.g. `LOOP`) is considered a *label* rather than an *instruction*, whether on its own line or preceding an instruction.
- Instructions (e.g. `LDA`, `STA`, etc.) must be preceded by whitespace. The number of space or tab characters doesn’t matter.
- Operands to those instructions (e.g. `$4000`, etc.) must be separated to the right of those instructions by some whitespace as well. The number of space or tab characters doesn’t matter. In this listing I’ve adjusted the number of spaces to make the columns neatly align.
- Any text to the right of a semicolon (`;`) character is considered a comment, ignored by the assembler and not included in the machine code at all.
- Any instruction preceded by a dot (e.g. `.ORG`) is a *directive* to the assembler, telling how to do some aspect of the translation into machine code, but not appearing in the final machine code itself.
- Note how the Jump instruction’s target address was resolved by the assembler, with no need for us to count address numbers to figure out where it should jump to. We simply place a label and let the assembler figure out those details for us.
- This assembler uses a dollar-symbol ($) to denote any hexadecimal value.
- The pound symbol (#) denotes an *immediate* value, meaning the literal number value specified. Otherwise, the operand value is considered to be a memory address location.

It should be noted that some of these conventions vary from assembler to assembler. For example, some assemblers require all labels to contain a colon at the end (e.g. `LOOP:` rather than `LOOP`). Some assemblers allow C-style hexadecimal notation (e.g. `0x4000`) while others insist on the `$` character. Some assemblers allow binary notation (e.g. `0b00000001` or `%00000001`) while others don’t. Some assemblers require directives be preceded by a dot (e.g. `.ORG`) while others insist directives *not* be preceded by any character. As always when using software, refer to the manufacturer’s documentation for details!
4.1.3 Slowing down the blinking

If we were to actually assemble and write this program to ROM, then start the computer to initiate program execution, we would likely find the LED blinking on and off so fast that it appeared to be steadily lit (albeit dimmer than usual). The reason for this is the fast fetch/execute cycle time of a typical microprocessor. A model 6502 running at a clock speed of 1 MHz would blink the LED on and off at a rate far too quick for the human eye to see.

In order to make the blinking rate slow enough to see, we must somehow delay the loop’s repetition. One easy way to do this is to insert a “counting” loop inside of our program’s blinking loop. This counting loop keeps the microprocessor occupied by doing nothing but sequentially counting, in order to purposely waste time and thereby delay its toggling of the LED output bit.

Here is a section of assembly code using common 6502 instructions to perform this delay task by forcing the microprocessor to count backwards from 255 (0xFF) until it reaches zero, complete with explanatory comments:

```assembly
LDX #$FF ; Load value 0xFF into register X

DELAY_LOOP
  DEX ; Decrement (subtract 1 from) value stored in X
  CPX #$00 ; Compare that value to zero
  BNE DELAY_LOOP ; If unequal, "branch" to DELAY_LOOP to repeat
```

One way to incorporate this delay code into our program is to simply insert it “in-line” with the original code between the EOR and JMP instructions, like this:

```assembly
.ORG $8000 ; Begin at address 0x8000
LDA #$01 ; Load 0x01 into Accumulator
LOOP
  ; Mark the beginning of the loop
  STA $4000 ; Store Accumulator value to Output port
  EOR #$01 ; XOR the least-significant bit
  LDX #$FF ; Load value 0xFF into register X

DELAY_LOOP
  DEX ; Decrement (subtract 1 from) value stored in X
  CPX #$00 ; Compare that value to zero
  BNE DELAY_LOOP ; If unequal, "branch" to DELAY_LOOP to repeat
  JMP LOOP ; Jump to the beginning of the loop
```

According to the model 6502 manual, the STA instruction requires 4 clock cycles, the EOR instruction 2 clock cycles, and the JMP instruction 3 clock cycles. This means 9 clock cycles would be required for every pass through the program, with two passes required for a full cycle of the LED’s blink (i.e. on and off). Dividing 1 MHz by the 18 clock cycles necessary to fully cycle the LED gives an LED blinking frequency of 55.556 kHz!
While this solution works quite well, there is a more sophisticated way to achieve the same “loop within a loop” structure, and that is to place the delay-time code within its own subroutine. A “subroutine” is a section of code that stands apart from the rest, ready to be called by the main portion of the program whenever needed.

Subroutines are particularly useful when that code must be invoked at multiple points within the main program. Instead of copying-and-pasting the necessary code repeatedly in-line where needed, we simply insert a “call” or “jump to subroutine” instruction where it’s needed and the microprocessor will jump to that new address (its Program Counter being preset as needed). Then, at the end of the subroutine we place a “return” instruction that tells the microprocessor to resume where it left off in the main program.

The following shows a listing of the assembly code for the slow-blinking program using a subroutine called DELAY:\footnote{The blank line between JMP LOOP and DELAY is there for esthetic purposes only, to help our eyes see the distinction between the main program and the subroutine. We could eliminate this blank line (or add more!) and the program would execute just as well.}:

```
.ORG $8000 ; Begin at address 0x8000
LDA #$01 ; Load 0x01 into Accumulator
LOOP ; Mark the beginning of the loop
STA $4000 ; Store Accumulator value to Output port
EOR #$01 ; XOR the least-significant bit
JSR DELAY ; Call the DELAY subroutine
JMP LOOP ; Jump to the beginning of the loop

DELAY
LDX #$FF ; Load value 0xFF into register X
DELAY_LOOP
DEX ; Decrement (subtract 1 from) value stored in X
CPX #$00 ; Compare that value to zero
BNE DELAY_LOOP ; If unequal, "branch" to DELAY_LOOP and repeat
RTS ; Return to the main program
```

Admittedly the use of subroutines doesn’t appear to be any better than simply inserting the DELAY code in-line with the original program. However, if we had a need to call this subroutine multiple times within our program, all we would need to add is another JSR DELAY instruction. Thus, the subroutine strategy becomes more efficient than the in-line strategy proportional to how many times that routine must execute.
An useful tool provided by most assemblers is a *disassembly* option. This takes the assembled machine code and translates it “backwards” into assembly code, displaying the memory addresses, machine code hex dump, and equivalent assembly side-by-side for comparison:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hexdump</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8000</td>
<td>A9 01</td>
<td>LDA #$01</td>
</tr>
<tr>
<td>$8002</td>
<td>8D 00 40</td>
<td>STA $4000</td>
</tr>
<tr>
<td>$8005</td>
<td>49 01</td>
<td>EOR #$01</td>
</tr>
<tr>
<td>$8007</td>
<td>20 0D 00</td>
<td>JSR $000D</td>
</tr>
<tr>
<td>$800A</td>
<td>4C 02 80</td>
<td>JMP $8002</td>
</tr>
<tr>
<td>$800D</td>
<td>A2 FF</td>
<td>LDX #$FF</td>
</tr>
<tr>
<td>$800F</td>
<td>CA</td>
<td>DEX</td>
</tr>
<tr>
<td>$8010</td>
<td>EO 00</td>
<td>CPX #$00</td>
</tr>
<tr>
<td>$8012</td>
<td>D0 FB</td>
<td>BNE $800F</td>
</tr>
<tr>
<td>$8014</td>
<td>60</td>
<td>RTS</td>
</tr>
</tbody>
</table>

Note how the “jump” and “branch” instructions all specify address locations in one way or another, but not the “return instruction” at the end of the subroutine. How does the microprocessor know which memory address to return to after completing the subroutine? The answer lies in a feature of the microprocessor called the *stack*: a section of volatile memory used by the processor to remember such things as previous Program Counter values. A jump-to-subroutine instruction causes the current memory address value held in the Program Counter to be “pushed” onto the stack before jumping to the subroutine’s starting memory address. A “return” instruction causes the microprocessor to “pop” the former address value off the stack and into the Program Counter again, so that execution resumes right where it left off.

If you examine this disassembled code closely, you will notice something strange with the “branch-if-not-equal” instruction: the disassembled code says `BNE $800F` but the machine code does not actually contain the `0x800F` address. Instead, it only contains the opcode for `BNE` (D0) and a byte with a value of `0xFB`. This is an example of *relative addressing*, where the operand to the instruction declares not the address itself, but rather how many addresses to skip, either forward or backward. As an eight-bit signed number, `0xFB` is equal to negative five. This tells the microprocessor to decrement its Program Counter by five to repeat the `DELAY LOOP`. If you count from address `0x8013` where the `0xFB` operand was resides to `0x800F` where the delay loop begins, you count five addresses (inclusive). Relative addressing is more efficient than absolute addressing because we only need one byte telling the instruction how far to jump instead of two bytes to specify a 16-bit address. Interestingly, the disassembler opted to show us an absolute address even though that’s not really how the 6502’s `BNE` instruction works.

Note how the “jump” and “branch” instructions all specify address locations in one way or another, but not the “return instruction” at the end of the subroutine. How does the microprocessor know which memory address to return to after completing the subroutine? The answer lies in a feature of the microprocessor called the *stack*: a section of volatile memory used by the processor to remember such things as previous Program Counter values. A jump-to-subroutine instruction causes the current memory address value held in the Program Counter to be “pushed” onto the stack before jumping to the subroutine’s starting memory address. A “return” instruction causes the microprocessor to “pop” the former address value off the stack and into the Program Counter again, so that execution resumes right where it left off.

If you examine this disassembled code closely, you will notice something strange with the “branch-if-not-equal” instruction: the disassembled code says `BNE $800F` but the machine code does not actually contain the `0x800F` address. Instead, it only contains the opcode for `BNE` (D0) and a byte with a value of `0xFB`. This is an example of *relative addressing*, where the operand to the instruction declares not the address itself, but rather how many addresses to skip, either forward or backward. As an eight-bit signed number, `0xFB` is equal to negative five. This tells the microprocessor to decrement its Program Counter by five to repeat the `DELAY LOOP`. If you count from address `0x8013` where the `0xFB` operand was resides to `0x800F` where the delay loop begins, you count five addresses (inclusive). Relative addressing is more efficient than absolute addressing because we only need one byte telling the instruction how far to jump instead of two bytes to specify a 16-bit address. Interestingly, the disassembler opted to show us an absolute address even though that’s not really how the 6502’s `BNE` instruction works.

---

Stacks are analogous to a pile (stack) of paper notes. If a person is reading a book and they suddenly get told to turn to a different chapter to read a passage there, they may write the current page number on a note and “push” that note to the top of the stack so they won’t forget it while turning to the new passage. After reading the new passage, the person retrieves their note from the top of the stack (i.e. “popping” it off the stack) and references it to return to the page where they left off. This may occur more than once, and the stack will “remember” not only the page numbers but also keep everything in the right order as the person eventually returns to their original place in the book.
4.1.4 Simplifying with symbols

Another technique useful for making our assembly-language programs easier to read and to maintain is the use of symbols to represent numerical values. Consider the last version of our LED-blinking program re-written to incorporate three symbols, defined at the very beginning of the code listing:

```
DTIME .EQU $FF    ; Create a symbol "DTIME" for the delay time parameter
OUTPT .EQU $4000  ; Create a symbol "OUTPT" for the Output port address
LED .EQU $01      ; Create a symbol "LED" for the LED's bit number

.ORG $8000        ; Begin at address 0x8000
LDA #LED          ; Load 0x01 into Accumulator
LOOP              ; Mark the beginning of the loop
    STA OUTPT     ; Store Accumulator value to Output port
    EOR #LED      ; XOR the least-significant bit
    JSR DELAY     ; Call the DELAY subroutine
    JMP LOOP      ; Jump to the beginning of the loop

DELAY
    LDX #DTIME    ; Load value 0xFF into register X
DELAY_LOOP
    DEX           ; Decrement (subtract 1 from) value stored in X
    CPX #$00      ; Compare that value to zero
    BNE DELAY_LOOP ; If unequal, "branch" to DELAY_LOOP and repeat
    RTS           ; Return to the main program
```

The .EQU directive tells the assembler to treat the symbol (on the left) as an alias of the value (on the right). This lets us use lettered symbols within our code rather than numerical values for important parameters such as I/O addresses, delay time, etc. These symbols may be used as many times as desired, and they will always mean the same thing (e.g. the LED symbol is used twice in this program, and it means $01 both times).
4.1.5 Using the stack

Previously we mentioned the microprocessor’s stack, a section of RAM used to hold data in sequential order. Stacks may be thought of as a Last-In First-Out shift register, where data retrieved from the stack is in reverse order of how data is placed onto the stack. The analogy of a microprocessor’s stack being a literal stack of paper sheets is helpful here: if we pull papers from the top of the stack, we will find their sequence is in reverse order of how we placed those sheets on the stack.

Microprocessors use their stack to manage subroutine calls, “pushing” the last Program Counter memory address to the stack prior to jumping to the subroutine’s address, then “popping” that old address off the stack when the subroutine completes so it knows where to resume its previous place in the main program. This form of stack usage is automatic, being built-in to the finite state machine sequence as part of each subroutine “call” instruction and each subroutine “return” instruction. Interrupts also use a stack to remember where to jump back in the main program after completing the interrupt service routine (ISR).

Certain instructions exist to make use of the stack in ways that are not necessarily related to subroutines or interrupts, and these can be very useful. Here we will explore one such practical use of the stack. Consider this simple “chasing LED” program using the 6502’s “Rotate Right” instruction to shift the place of a single “1” bit in the Accumulator byte, the goal being to create a “chasing” LED display on our computer where the light appears to repeatedly sweep along the row of LEDs:

```
.ORG $8000 ; Begin at address 0x8000
LDA #$01 ; Load 0x01 into Accumulator
LOOP ; Mark the beginning of the loop
STA $4000 ; Store Accumulator value to Output port
ROR ; Rotate Accumulator bits one place right
JMP LOOP ; Jump to the beginning of the loop
```

This code is every bit as simple as our original LED-blinking program. When run, it produces the following pattern of light (sequence shown chronologically from top to bottom):
During the step where no LEDs are lit, the “1” bit resides in the Carry bit of the microprocessor’s Status register, a special register used to store the results of certain mathematical and logical operations.

This pattern of light is what we expect the ROR instruction to produce after the Accumulator is initially loaded with 0x01. All is well, except for the same problem we had with our original “blinking LED” program: the sequence runs too fast for our eyes to discern. It just looks like a blur of eight LEDs all (dimly) lit!

We already know how to slow programs down, by inserting a counting loop that “wastes” the microprocessor’s time, so let’s modify this program accordingly:

```
.ORG $8000 ; Begin at address 0x8000
LDA #$01 ; Load 0x01 into Accumulator
LOOP ; Mark the beginning of the loop
STA $4000 ; Store Accumulator value to Output port
ROR ; Rotate Accumulator bits one place right
JSR DELAY ; Call the DELAY subroutine
JMP LOOP ; Jump to the beginning of the loop

DELAY
LDX #DTIME ; Load value 0xFF into register X
DELAY_LOOP
DEX ; Decrement (subtract 1 from) value stored in X
CPX #$00 ; Compare that value to zero
BNE DELAY_LOOP ; If unequal, "branch" to DELAY_LOOP and repeat
RTS ; Return to the main program
```

However, when we run this program we get a different light sequence:

```

<table>
<thead>
<tr>
<th>P7 P6 P5 P4 P3 P2 P1 P0</th>
<th>Accumulator value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 1</td>
<td>0x01</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>0x00</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0x80</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0xC0</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0xE0</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0xF0</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0xF8</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0xFC</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0xFE</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0xFF</td>
</tr>
</tbody>
</table>
```

For some reason, the “0” states rotated off the LSB-end of the byte are becoming “1” states to fill the MSB. Recall that the ROR instruction draws from the Carry bit of the Status register to fill
the MSB at each iteration. A reasonable hypothesis is that something other than the \texttt{ROR} is setting the Carry bit.

Indeed something \textit{does} act to set the Carry bit when we don’t want it to: the “compare X” \texttt{CPX} instruction inside our \texttt{DELAY} subroutine. According to the 6502 instruction set manual, the \texttt{CPX} sets the Carry bit if ever the X register’s value is equal to or greater than the value it’s being compared against. In fact, this is how the \texttt{BNE} instruction knows when to branch: it checks the Status register which is updated by all mathematical, logical, and comparison instructions. Given the design of our time-delay subroutine where the X register begins at a large value and counts down toward the comparison value of zero, we are \textit{guaranteed} to return from that subroutine with the Carry bit set.

This causes problems for our \texttt{ROR} instruction, which takes the “1” value left in the Carry bit from the subroutine’s \texttt{CPX} instruction and adds it to our chasing light sequence, which we do not want. Somehow we need the \texttt{ROR} to act on the Carry bit \textit{it} generated, not the \textit{new} Carry bit generated by the subroutine’s \texttt{CPX} instruction.

Our stack ends up being a simple solution to this problem. All we need to do is “push” the Status register’s state to the stack prior to calling the subroutine, then “pop” that old data back off the stack and into the Status register again before the \texttt{ROR} instruction reads the Carry bit. In other words, we can use the stack as temporary storage for the Status bits, and recall those bits after the subroutine is done using the Status register for its own purposes.

All this requires is the addition of two new instructions surrounding the “jump to subroutine”: a \texttt{PHP} instruction (“push processor status on stack”) prior to the jump, and \texttt{PLP} instruction (“pull processor status from stack”) after the jump:

```
.LORG $8000 ; Begin at address 0x8000
LDA #$01 ; Load 0x01 into Accumulator
LOOP ; Mark the beginning of the loop
  STA $4000 ; Store Accumulator value to Output port
  ROR ; Rotate Accumulator bits one place right
  PHP ; Push Status register to stack
JSR DELAY ; Call the DELAY subroutine
  PLP ; Pop Status register off stack
  JMP LOOP ; Jump to the beginning of the loop
.DEFLY
  LDX #DTIME ; Load value 0xFF into register X
  DELAY_LOOP ; Load value 0x00 into output port
    DEX ; Decrement (subtract 1 from) value stored in X
    CPX #0x00 ; Compare that value to zero
    BNE DELAY_LOOP ; If unequal, "branch" to DELAY_LOOP and repeat
    RTS ; Return to the main program
```

The stack is a very useful feature of any microprocessor, but it does have its limitations. If we push far more data onto the stack than we pop off, we can get a \textit{stack overflow} where the oldest
data gets overwritten and is lost. Also, we need to be very careful that we push from the correct sources and pop to the correct destinations. For example, in the above program we pushed the Status register to the stack, then immediately after that the JSR instruction pushed the Program Counter value to the stack before going to the subroutine. When the subroutine completed, it popped the old Program Counter value off the stack, and then immediately after that we popped the old Status register off the stack. This works because those sources and destinations came in the correct sequence, and so the data going on and off the stack went where it should.
Chapter 5

Questions

This learning module, along with all others in the ModEL collection, is designed to be used in an inverted instructional environment where students independently read the tutorials and attempt to answer questions on their own prior to the instructor’s interaction with them. In place of lecture, the instructor engages with students in Socratic-style dialogue, probing and challenging their understanding of the subject matter through inquiry.

Answers are not provided for questions within this chapter, and this is by design. Solved problems may be found in the Tutorial and Derivation chapters, instead. The goal here is independence, and this requires students to be challenged in ways where others cannot think for them. Remember that you always have the tools of experimentation and computer simulation (e.g. SPICE) to explore concepts!

The following lists contain ideas for Socratic-style questions and challenges. Upon inspection, one will notice a strong theme of metacognition within these statements: they are designed to foster a regular habit of examining one’s own thoughts as a means toward clearer thinking. As such these sample questions are useful both for instructor-led discussions as well as for self-study.

---

1. Technical reading is an essential academic skill for any technical practitioner to possess for the simple reason that the most comprehensive, accurate, and useful information to be found for developing technical competence is in textual form. Technical careers in general are characterized by the need for continuous learning to remain current with standards and technology, and therefore any technical practitioner who cannot read well is handicapped in their professional development. An excellent resource for educators on improving students' reading prowess through intentional effort and strategy is the book *Reading For Understanding – How Reading Apprenticeship Improves Disciplinary Learning in Secondary and College Classrooms* by Ruth Schoenbach, Cynthia Greenleaf, and Lynn Murphy.

2. Lecture is popular as a teaching method because it is easy to implement: any reasonably articulate subject matter expert can talk to students, even with little preparation. However, it is also quite problematic. A good lecture always makes complicated concepts seem easier than they are, which is bad for students because it instills a false sense of confidence in their own understanding; reading and re-articulation requires more cognitive effort and serves to verify comprehension. A culture of teaching-by-lecture fosters a debilitating dependence upon direct personal instruction, whereas the challenges of modern life demand independent and critical thought made possible only by gathering information and perspectives from afar. Information presented in a lecture is ephemeral, easily lost to failures of memory and dictation; text is forever, and may be referenced at any time.
GENERAL CHALLENGES FOLLOWING TUTORIAL READING

- **Summarize** as much of the text as you can in one paragraph of your own words. A helpful strategy is to explain ideas as you would for an intelligent child: as simple as you can without compromising too much accuracy.

- **Simplify** a particular section of the text, for example a paragraph or even a single sentence, so as to capture the same fundamental idea in fewer words.

- Where did the text **make the most sense** to you? What was it about the text’s presentation that made it clear?

- Identify where it might be easy for someone to **misunderstand the text**, and explain why you think it could be confusing.

- Identify any **new concept(s)** presented in the text, and explain in your own words.

- Identify any **familiar concept(s)** such as physical laws or principles applied or referenced in the text.

- Devise a **proof of concept** experiment demonstrating an important principle, physical law, or technical innovation represented in the text.

- Devise an experiment to **disprove** a plausible misconception.

- Did the text reveal any **misconceptions** you might have harbored? If so, describe the misconception(s) and the reason(s) why you now know them to be incorrect.

- Describe any useful **problem-solving strategies** applied in the text.

- **Devise a question** of your own to challenge a reader’s comprehension of the text.
**General follow-up challenges for assigned problems**

- Identify where any **fundamental laws or principles** apply to the solution of this problem, especially before applying any mathematical techniques.

- Devise a **thought experiment** to explore the characteristics of the problem scenario, applying known laws and principles to mentally model its behavior.

- Describe in detail your own **strategy** for solving this problem. How did you identify and organized the given information? Did you sketch any diagrams to help frame the problem?

- Is there **more than one way** to solve this problem? Which method seems best to you?

- **Show the work** you did in solving this problem, even if the solution is incomplete or incorrect.

- What would you say was the **most challenging part** of this problem, and why was it so?

- Was any important information **missing** from the problem which you had to research or recall?

- Was there any **extraneous** information presented within this problem? If so, what was it and why did it not matter?

- Examine **someone else’s solution** to identify where they applied fundamental laws or principles.

- **Simplify** the problem from its given form and show how to solve this simpler version of it. Examples include eliminating certain variables or conditions, altering values to simpler (usually whole) numbers, applying a **limiting case** (i.e. altering a variable to some extreme or ultimate value).

- For quantitative problems, identify the **real-world meaning** of all intermediate calculations: their units of measurement, where they fit into the scenario at hand. Annotate any diagrams or illustrations with these calculated values.

- For quantitative problems, try approaching it **qualitatively** instead, thinking in terms of “increase” and “decrease” rather than definite values.

- For qualitative problems, try approaching it **quantitatively** instead, proposing simple numerical values for the variables.

- Were there any **assumptions** you made while solving this problem? Would your solution change if one of those assumptions were altered?

- Identify where it would be easy for someone to go astray in attempting to solve this problem.

- **Formulate your own problem** based on what you learned solving this one.

**General follow-up challenges for experiments or projects**

- In what way(s) was this experiment or project **easy to complete**?

- Identify some of the **challenges you faced** in completing this experiment or project.
• Show how thorough documentation assisted in the completion of this experiment or project.

• Which fundamental laws or principles are key to this system’s function?

• Identify any way(s) in which one might obtain false or otherwise misleading measurements from test equipment in this system.

• What will happen if (component X) fails (open/shorted/etc.)?

• What would have to occur to make this system unsafe?
5.1 Conceptual reasoning

These questions are designed to stimulate your analytic and synthetic thinking. In a Socratic discussion with your instructor, the goal is for these questions to prompt an extended dialogue where assumptions are revealed, conclusions are tested, and understanding is sharpened. Your instructor may also pose additional questions based on those assigned, in order to further probe and refine your conceptual understanding.

Questions that follow are presented to challenge and probe your understanding of various concepts presented in the tutorial. These questions are intended to serve as a guide for the Socratic dialogue between yourself and the instructor. Your instructor’s task is to ensure you have a sound grasp of these concepts, and the questions contained in this document are merely a means to this end. Your instructor may, at his or her discretion, alter or substitute questions for the benefit of tailoring the discussion to each student’s needs. The only absolute requirement is that each student is challenged and assessed at a level equal to or greater than that represented by the documented questions.

It is far more important that you convey your reasoning than it is to simply convey a correct answer. For this reason, you should refrain from researching other information sources to answer questions. What matters here is that you are doing the thinking. If the answer is incorrect, your instructor will work with you to correct it through proper reasoning. A correct answer without an adequate explanation of how you derived that answer is unacceptable, as it does not aid the learning or assessment process.

You will note a conspicuous lack of answers given for these conceptual questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your conceptual answers, where applicable, is to use circuit simulation software to explore the effects of changes made to circuits. For example, if one of these conceptual questions challenges you to predict the effects of altering some component parameter in a circuit, you may check the validity of your work by simulating that same parameter change within software and seeing if the results agree.

---

3 Analytical thinking involves the “disassembly” of an idea into its constituent parts, analogous to dissection. Synthetic thinking involves the “assembly” of a new idea comprised of multiple concepts, analogous to construction. Both activities are high-level cognitive skills, extremely important for effective problem-solving, necessitating frequent challenge and regular practice to fully develop.
5.1.1 Reading outline and reflections

“Reading maketh a full man; conference a ready man; and writing an exact man” – Francis Bacon

Francis Bacon’s advice is a blueprint for effective education: reading provides the learner with knowledge, writing focuses the learner’s thoughts, and critical dialogue equips the learner to confidently communicate and apply their learning. Independent acquisition and application of knowledge is a powerful skill, well worth the effort to cultivate. To this end, students should read these educational resources closely, write their own outline and reflections on the reading, and discuss in detail their findings with classmates and instructor(s). You should be able to do all of the following after reading any instructional text:

√ Briefly OUTLINE THE TEXT, as though you were writing a detailed Table of Contents. Feel free to rearrange the order if it makes more sense that way. Prepare to articulate these points in detail and to answer questions from your classmates and instructor. Outlining is a good self-test of thorough reading because you cannot outline what you have not read or do not comprehend.

√ Demonstrate ACTIVE READING STRATEGIES, including verbalizing your impressions as you read, simplifying long passages to convey the same ideas using fewer words, annotating text and illustrations with your own interpretations, working through mathematical examples shown in the text, cross-referencing passages with relevant illustrations and/or other passages, identifying problem-solving strategies applied by the author, etc. Technical reading is a special case of problem-solving, and so these strategies work precisely because they help solve any problem: paying attention to your own thoughts (metacognition), eliminating unnecessary complexities, identifying what makes sense, paying close attention to details, drawing connections between separated facts, and noting the successful strategies of others.

√ Identify IMPORTANT THEMES, especially GENERAL LAWS and PRINCIPLES, expounded in the text and express them in the simplest of terms as though you were teaching an intelligent child. This emphasizes connections between related topics and develops your ability to communicate complex ideas to anyone.

√ Form YOUR OWN QUESTIONS based on the reading, and then pose them to your instructor and classmates for their consideration. Anticipate both correct and incorrect answers, the incorrect answer(s) assuming one or more plausible misconceptions. This helps you view the subject from different perspectives to grasp it more fully.

√ Devise EXPERIMENTS to test claims presented in the reading, or to disprove misconceptions. Predict possible outcomes of these experiments, and evaluate their meanings: what result(s) would confirm, and what would constitute disproof? Running mental simulations and evaluating results is essential to scientific and diagnostic reasoning.

√ Specifically identify any points you found CONFUSING. The reason for doing this is to help diagnose misconceptions and overcome barriers to learning.
5.1.2 Foundational concepts

Correct analysis and diagnosis of electric circuits begins with a proper understanding of some basic concepts. The following is a list of some important concepts referenced in this module’s full tutorial. Define each of them in your own words, and be prepared to illustrate each of these concepts with a description of a practical example and/or a live demonstration.

???

???

???

???

???

???

???
5.1.3 Terminal assignments and limits

Reference the datasheet for the MSP430G2553 20-terminal microcontroller and identify some of the alternate functions of its package terminals (i.e., those functions other than digital I/O port assignments such as P1.0-P1.7 for the various pins in the IC chip).

Reference the datasheet for the MSP430G2553 20-terminal microcontroller and identify some of the electrical limits:

- Maximum and minimum power supply voltage =
- Maximum and minimum analog input voltage =
- Maximum I/O pin current =

Challenges

- Suppose you needed to have the microcontroller turn a discrete load on and off, with an expected load current of 1 Ampere. Identify ways to interface the microcontroller output pin to this (relatively) high-current load.
5.1.4 Interrupt capabilities

An important concept in microprocessor and microcontroller operation is the notion of an *interrupt*. This is a signal, either received from some external source or generated internally, that causes the processor to halt its regular execution of the program and “jump” to a new location in program memory where other instructions exist to tell it how to *service* (i.e., deal with) this interrupting event.

Interrupts are often classified as being either *maskable* or *non-maskable*. This simply refers to whether the user is able to disable interrupts or not. A “maskable” interrupt is one that can be disabled by setting certain bits in a register of the processor, and a “non-maskable” is one that is always alert and cannot be disabled. The *reset* function on a microprocessor or microcontroller is a typical example of a non-maskable interrupt.

The way this usually works is that an interrupt event causes the Program Counter to go to a pre-designated address in memory called a *vector*, which contains the memory address where the first instruction of the service routine (typically called an “Interrupt Service Routine” or *ISR*) exists. These interrupt vectors are often hard-coded into the processor at the time of manufacture, but the user (programmer) is free to place their ISR code wherever they wish in memory and to write the appropriate starting addresses for those routines into the appropriate interrupt vector location(s).

In the MSP430 microcontroller series, an *interrupt enable* bit determines whether a maskable interrupt is able to function, while an *interrupt flag* bit records if an interrupt event actually happens.

Reference the datasheet for the MSP430G2553 microcontroller and identify some of the available interrupt sources, priorities, flags, and vectors.

**Challenges**

- Describe a practical application for interrupting a microcontroller from its regular program execution.
5.1.5 Disabling the watchdog timer

When you open a new assembly-language project in Code Composer Studio (CCS) the .asm source code file does not begin in an empty state. Instead, it is pre-populated by several directives, labels, and instructions. One of those is as follows (complete with its own comment):

```
StopWDT mov.w #WDTPW|WDTHOLD,&WDTCTL ; Stop watchdog timer
```

Reference the datasheet\(^4\) for the MSP430G2553 20-terminal microcontroller to identify the internal watchdog timer’s function in the microcontroller, and also explain why you think CCS defaults to disabling it.

<table>
<thead>
<tr>
<th>Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Identify a practical application for using the microcontroller’s watchdog timer.</td>
</tr>
</tbody>
</table>

\(^4\)If you are interested in diving deeper into this topic, you will find much more information on the watchdog timer in the *MSP430x2xx Family User’s Guide* than in the datasheet.
5.2 Quantitative reasoning

These questions are designed to stimulate your computational thinking. In a Socratic discussion with your instructor, the goal is for these questions to reveal your mathematical approach(es) to problem-solving so that good technique and sound reasoning may be reinforced. Your instructor may also pose additional questions based on those assigned, in order to observe your problem-solving firsthand.

Mental arithmetic and estimations are strongly encouraged for all calculations, because without these abilities you will be unable to readily detect errors caused by calculator misuse (e.g. keystroke errors).

You will note a conspicuous lack of answers given for these quantitative questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. My advice is to use circuit simulation software such as SPICE to check the correctness of quantitative answers. Refer to those learning modules within this collection focusing on SPICE to see worked examples which you may use directly as practice problems for your own study, and/or as templates you may modify to run your own analyses and generate your own practice problems.

Completely worked example problems found in the Tutorial may also serve as “test cases” for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely on an answer key!

---

5In other words, set up the circuit simulation software to analyze the same circuit examples found in the Tutorial. If the simulated results match the answers shown in the Tutorial, it confirms the simulation has properly run. If the simulated results disagree with the Tutorial’s answers, something has been set up incorrectly in the simulation software. Using every Tutorial as practice in this way will quickly develop proficiency in the use of circuit simulation software.

6This approach is perfectly in keeping with the instructional philosophy of these learning modules: teaching students to be self-sufficient thinkers. Answer keys can be useful, but it is even more useful to your long-term success to have a set of tools on hand for checking your own work, because once you have left school and are on your own, there will no longer be “answer keys” available for the problems you will have to solve.
5.2.1 Miscellaneous physical constants

Note: constants shown in bold type are exact, not approximations. Values inside of parentheses show one standard deviation ($\sigma$) of uncertainty in the final digits: for example, Avogadro’s number given as $6.02214179(30) \times 10^{23}$ means the center value $(6.02214179 \times 10^{23})$ plus or minus $0.00000030 \times 10^{23}$.

Avogadro’s number ($N_A$) = $6.02214179(30) \times 10^{23}$ per mole (mol$^{-1}$)

Boltzmann’s constant ($k$) = $1.3806504(24) \times 10^{-23}$ Joules per Kelvin (J/K)

Electronic charge ($e$) = $1.602176487(40) \times 10^{-19}$ Coulomb (C)

Faraday constant ($F$) = $9.64853399(24) \times 10^4$ Coulombs per mole (C/mol)

Magnetic permeability of free space ($\mu_0$) = $1.25663706212(19) \times 10^{-6}$ Henrys per meter (H/m)

Electric permittivity of free space ($\epsilon_0$) = $8.8541878128(13) \times 10^{-12}$ Farads per meter (F/m)

Characteristic impedance of free space ($Z_0$) = $376.730313668(57)$ Ohms (Ω)

Gravitational constant ($G$) = $6.67428(67) \times 10^{-11}$ cubic meters per kilogram-seconds squared (m$^3$/kg-s$^2$)

Molar gas constant ($R$) = $8.314472(15)$ Joules per mole-Kelvin (J/mol-K) = $0.08205746(14)$ liters-atmospheres per mole-Kelvin

Planck constant ($h$) = $6.62606896(33) \times 10^{-34}$ joule-seconds (J-s)

Stefan-Boltzmann constant ($\sigma$) = $5.670400(40) \times 10^{-8}$ Watts per square meter-Kelvin$^4$ (W/m$^2$·K$^4$)

Speed of light in a vacuum ($c$) = **299792458 meters per second** (m/s) = $186282.4$ miles per second (mi/s)

Note: All constants taken from NIST data “Fundamental Physical Constants – Extensive Listing”, from http://physics.nist.gov/constants, National Institute of Standards and Technology (NIST), 2006; with the exception of the permeability of free space which was taken from NIST’s 2018 CODATA recommended values database.
5.2. QUANTITATIVE REASONING

5.2.2 Introduction to spreadsheets

A powerful computational tool you are encouraged to use in your work is a spreadsheet. Available on most personal computers (e.g. Microsoft Excel), spreadsheet software performs numerical calculations based on number values and formulae entered into cells of a grid. This grid is typically arranged as lettered columns and numbered rows, with each cell of the grid identified by its column/row coordinates (e.g. cell B3, cell A8). Each cell may contain a string of text, a number value, or a mathematical formula. The spreadsheet automatically updates the results of all mathematical formulae whenever the entered number values are changed. This means it is possible to set up a spreadsheet to perform a series of calculations on entered data, and those calculations will be re-done by the computer any time the data points are edited in any way.

For example, the following spreadsheet calculates average speed based on entered values of distance traveled and time elapsed:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Distance traveled</td>
<td>46.9</td>
<td>Kilometers</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Time elapsed</td>
<td>1.18</td>
<td>Hours</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Average speed</td>
<td>= B1 / B2</td>
<td>km/h</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Text labels contained in cells A1 through A3 and cells C1 through C3 exist solely for readability and are not involved in any calculations. Cell B1 contains a sample distance value while cell B2 contains a sample time value. The formula for computing speed is contained in cell B3. Note how this formula begins with an “equals” symbol (=), references the values for distance and speed by lettered column and numbered row coordinates (B1 and B2), and uses a forward slash symbol for division (/). The coordinates B1 and B2 function as variables\(^7\) would in an algebraic formula.

When this spreadsheet is executed, the numerical value 39.74576 will appear in cell B3 rather than the formula = B1 / B2, because 39.74576 is the computed speed value given 46.9 kilometers traveled over a period of 1.18 hours. If a different numerical value for distance is entered into cell B1 or a different value for time is entered into cell B2, cell B3’s value will automatically update. All you need to do is set up the given values and any formulae into the spreadsheet, and the computer will do all the calculations for you.

Cell B3 may be referenced by other formulae in the spreadsheet if desired, since it is a variable just like the given values contained in B1 and B2. This means it is possible to set up an entire chain of calculations, one dependent on the result of another, in order to arrive at a final value. The arrangement of the given data and formulae need not follow any pattern on the grid, which means you may place them anywhere.

---

\(^7\)Spreadsheets may also provide means to attach text labels to cells for use as variable names (Microsoft Excel simply calls these labels “names”), but for simple spreadsheets such as those shown here it’s usually easier just to use the standard coordinate naming for each cell.
Common arithmetic operations available for your use in a spreadsheet include the following:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Powers (^)
- Square roots (sqrt())
- Logarithms (ln(), log10())

Parentheses may be used to ensure proper order of operations within a complex formula. Consider this example of a spreadsheet implementing the quadratic formula, used to solve for roots of a polynomial expression in the form of $ax^2 + bx + c$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x_1$ = $(-B4 + \text{sqrt}((B4^2) - (4<em>B3</em>B5))) / (2*B3)$</td>
</tr>
<tr>
<td>2</td>
<td>$x_2$ = $(-B4 - \text{sqrt}((B4^2) - (4<em>B3</em>B5))) / (2*B3)$</td>
</tr>
<tr>
<td>3</td>
<td>$a$ = 9</td>
</tr>
<tr>
<td>4</td>
<td>$b$ = 5</td>
</tr>
<tr>
<td>5</td>
<td>$c$ = -2</td>
</tr>
</tbody>
</table>

This example is configured to compute roots of the polynomial $9x^2 + 5x - 2$ because the values of 9, 5, and -2 have been inserted into cells B3, B4, and B5, respectively. Once this spreadsheet has been built, though, it may be used to calculate the roots of any second-degree polynomial expression simply by entering the new $a$, $b$, and $c$ coefficients into cells B3 through B5. The numerical values appearing in cells B1 and B2 will be automatically updated by the computer immediately following any changes made to the coefficients.

---

8Modern spreadsheet software offers a bewildering array of mathematical functions you may use in your computations. I recommend you consult the documentation for your particular spreadsheet for information on operations other than those listed here.

9Spreadsheet programs, like text-based programming languages, are designed to follow standard order of operations by default. However, my personal preference is to use parentheses even where strictly unnecessary just to make it clear to any other person viewing the formula what the intended order of operations is.

10Reviewing some algebra here, a root is a value for $x$ that yields an overall value of zero for the polynomial. For this polynomial ($9x^2 + 5x - 2$) the two roots happen to be $x = 0.269381$ and $x = -0.82494$, with these values displayed in cells B1 and B2, respectively upon execution of the spreadsheet.
Alternatively, one could break up the long quadratic formula into smaller pieces like this:

\[ y = \sqrt{b^2 - 4ac} \quad z = 2a \]

\[ x = \frac{-b \pm y}{z} \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x_1</td>
<td>= (-B4 + C1) / C2</td>
<td>= sqrt((B4^2) - (4<em>B3</em>B5))</td>
</tr>
<tr>
<td>2</td>
<td>x_2</td>
<td>= (-B4 - C1) / C2</td>
<td>= 2*B3</td>
</tr>
<tr>
<td>3</td>
<td>a =</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b =</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c =</td>
<td>-2</td>
<td></td>
</tr>
</tbody>
</table>

Note how the square-root term \((y)\) is calculated in cell \(C1\), and the denominator term \((z)\) in cell \(C2\). This makes the two final formulae (in cells \(B1\) and \(B2\)) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary\(^{11}\) – all that matters is that they properly reference each other in the formulae.

Spreadsheets are particularly useful for situations where the same set of calculations representing a circuit or other system must be repeated for different initial conditions. The power of a spreadsheet is that it automates what would otherwise be a tedious set of calculations. One specific application of this is to simulate the effects of various components within a circuit failing with abnormal values (e.g. a shorted resistor simulated by making its value nearly zero; an open resistor simulated by making its value extremely large). Another application is analyzing the behavior of a circuit design given new components that are out of specification, and/or aging components experiencing drift over time.

\(^{11}\)My personal preference is to locate all the “given” data in the upper-left cells of the spreadsheet grid (each data point flanked by a sensible name in the cell to the left and units of measurement in the cell to the right as illustrated in the first distance/time spreadsheet example), sometimes coloring them in order to clearly distinguish which cells contain entered data versus which cells contain computed results from formulae. I like to place all formulae in cells below the given data, and try to arrange them in logical order so that anyone examining my spreadsheet will be able to figure out how I constructed a solution. This is a general principle I believe all computer programmers should follow: document and arrange your code to make it easy for other people to learn from it.
5.2.3 Integer conversion table

Complete this table, performing all necessary conversions between numeration systems of these unsigned integer quantities:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Octal</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>10010</td>
<td>134</td>
<td>92</td>
<td>12</td>
</tr>
<tr>
<td>110011</td>
<td>67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100101</td>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111101000</td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011010110</td>
<td>336</td>
<td></td>
<td>DE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2D6</td>
<td></td>
</tr>
</tbody>
</table>

**Challenges**

- Which type of conversion do you find most difficult to perform, and why is that?

5.2.4 Setting up Port 1 I/O

Suppose you wish to use pins P1.0 through P1.5 as inputs, and use pins P1.6 and P1.7 as outputs. Write the necessary assembly-language instructions to configure these pins as such, and also an instruction to **set** P1.6 and **clear** P1.7.

**Challenges**

- Explain why we don’t need to write any data to P1SEL or P1SEL2.
- Explain why we would have to write data to P2SEL if this were Port 2 we were using.
5.2.5 Setting up Timer A

The MSP430 microcontroller series offer multiple timer functions, which are counters driven by clock pulse sources. Not being dependent upon the CPU’s fetch/execute cycle means these timers will track true time independently of the CPU’s programming, which makes them very useful for real-time control and precision timing applications.

Reference the *MSP430x2xx Family User’s Guide* document and determine how to configure Timer A0 to be driven by the microcontroller’s Auxiliary Clock (ACLK) signal at one-quarter that signal’s frequency, in up/down mode, with interrupt capabilities disabled.

**Challenges**

- Identify the clock divider options for this counter, and how you think these are obtained in hardware.

5.2.6 Selecting sub-main clock source

The MSP430 microcontroller series offer multiple oscillator signal sources for its internal clock signals. One of these signals is SMCLK, the Sub-Main Clock signal used to drive certain peripheral functions (i.e. other than the CPU).

Reference the *MSP430x2xx Family User’s Guide* document and determine how to configure the SMCLK signal to be driven by the VLOCLK on-board oscillator at one-eighth that oscillator’s frequency.

**Challenges**

- Identify the criteria important to determining proper clock frequency in a microcontroller.
5.2.7 MSP430 header file

Locate and view the header file for your particular microcontroller (e.g. msp430g2553.h), and locate within that file the various symbolic constants used as human-readable bit labels for various registers within the microcontroller. Then, answer the following questions:

- Identify the binary equivalent of BIT4
- Identify the hexadecimal and binary equivalents for the Watchdog Timer Password (WDTPW)
- Show how the Watchdog Timer interval may be set with a single symbolic constant (e.g. WDT_ADLY_1000) that is itself comprised of other symbolic constants
- How exactly are these symbolic constants useful to us when writing programs for the MSP430 microcontroller?

[Challenges]

- Explain how you could create your own symbolic constant(s) customized for a particular program you are writing.
5.3 Diagnostic reasoning

These questions are designed to stimulate your deductive and inductive thinking, where you must apply general principles to specific scenarios (deductive) and also derive conclusions about the failed circuit from specific details (inductive). In a Socratic discussion with your instructor, the goal is for these questions to reinforce your recall and use of general circuit principles and also challenge your ability to integrate multiple symptoms into a sensible explanation of what’s wrong in a circuit. Your instructor may also pose additional questions based on those assigned, in order to further challenge and sharpen your diagnostic abilities.

As always, your goal is to fully explain your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor’s responsibility is to probe and challenge your understanding of the relevant principles and analytical processes in order to ensure you have a strong foundation upon which to build further understanding.

You will note a conspicuous lack of answers given for these diagnostic questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your diagnostic answers, where applicable, is to use circuit simulation software to explore the effects of faults placed in circuits. For example, if one of these diagnostic questions requires that you predict the effect of an open or a short in a circuit, you may check the validity of your work by simulating that same fault (substituting a very high resistance in place of that component for an open, and substituting a very low resistance for a short) within software and seeing if the results agree.
5.3.1 Poor interface design

Explain why this attempt at interfacing the microcontroller to a high-current load is flawed, and propose a better solution:

Challenges

- Write a snippet of assembly-language code to turn on the load.
5.3.2 Improving a debug session

A very helpful feature of any software development application is *debugging*, where you get to interrupt the normal flow of execution and “single-step” the code to see what the microprocessor is doing at each line of code.

Suppose a student is developing a program to blink eight LEDs connected to the eight pins of Port 1 on a MSP430G2553 microcontroller, a partial\(^\text{12}\) listing of their code shown below:

```
MOV #0x00, P1OUT ; Initializes Port 1 pin states to low (0)
MOV #0xFF, P1DIR ; Sets all Port 1 pin directions to outputs (1)

LOOP
    XOR #0xFF, P1OUT ; toggle all bits in P1OUT using XOR function
    CALL #DELAY
    JMP LOOP

DELAY
    MOV #0xFFFF, R4

DELAYLOOP
    SUB #0x01, R4 ; Decrement R4 by one
    JNZ DELAYLOOP
    RET
```

When the student assembles this code and tries to run a debug session, she finds herself “stuck” in the *DELAYLOOP* for a great many steps. Identify a way for her to alter this program so that it does not linger in the “time delay” loop as long as it normally would when running at full speed.

**Challenges**

- Identify how to use a *comment* to very quickly eliminate the delay loop entirely from this code’s execution.

---

\(^{12}\) Assembler directives and other “boilerplate” code has been removed for simplicity.
5.3.3 Incorrect sum

A student writes a simple program to add two numbers together on a MSP430G2553 microcontroller, a partial listing of their code shown below:

```
MAIN
    mov #0xF200, R5
    add #0xFF00, R5
```

When run, the sum in register R5 is 0xF100 which is an incorrect sum. Identify the correct sum, and explain why this one is wrong.

**Challenges**

- Convert these quantities to decimal and show their proper addition.

---

\[13\text{Assembler directives and other “boilerplate” code has been removed for simplicity.}\]
Appendix A

Problem-Solving Strategies

The ability to solve complex problems is arguably one of the most valuable skills one can possess, and this skill is particularly important in any science-based discipline.

- **Study principles, not procedures.** Don’t be satisfied with merely knowing how to compute solutions – learn why those solutions work.

- **Identify** what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any “missing pieces” to a solution. **Annotate** all diagrams with this data.

- **Sketch a diagram** to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function before constructing it.

- **Follow the units of measurement and meaning of every calculation.** If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don’t understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.

- **Perform “thought experiments”** to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform diagnostic tests rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.

- **Simplify the problem** until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.

- **Check for exceptions** to see if your solution is incorrect or incomplete. A good solution will work for all known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to challenge that new idea to see if it holds up under a battery of tests. The philosophical
principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

- **Work “backward”** from a hypothetical solution to a new set of given conditions.
- **Add quantities** to problems that are qualitative in nature, because sometimes a little math helps illuminate the scenario.
- **Sketch graphs** illustrating how variables relate to each other. These may be quantitative (i.e. with realistic number values) or qualitative (i.e. simply showing increases and decreases).
- **Treat quantitative problems as qualitative** in order to discern the relative magnitudes and/or directions of change of the relevant variables. For example, try determining what happens if a certain variable were to increase or decrease before attempting to precisely calculate quantities: how will each of the dependent variables respond, by increasing, decreasing, or remaining the same as before?
- **Consider limiting cases.** This works especially well for qualitative problems where you need to determine which direction a variable will change. Take the given condition and magnify that condition to an extreme degree as a way of simplifying the direction of the system’s response.
- **Check your work.** This means regularly testing your conclusions to see if they make sense. This does *not* mean repeating the same steps originally used to obtain the conclusion(s), but rather to use some other means to check validity. Simply repeating procedures often leads to *repeating the same errors* if any were made, which is why alternative paths are better.
Appendix B

Instructional philosophy

“The unexamined circuit is not worth energizing” – Socrates (if he had taught electricity)

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits.

The following principles inform the instructional and assessment philosophies embodied in these learning modules:

• The first goal of education is to enhance clear and independent thought, in order that every student reach their fullest potential in a highly complex and inter-dependent world. Robust reasoning is always more important than particulars of any subject matter, because its application is universal.

• Literacy is fundamental to independent learning and thought because text continues to be the most efficient way to communicate complex ideas over space and time. Those who cannot read with ease are limited in their ability to acquire knowledge and perspective.

• Articulate communication is fundamental to work that is complex and interdisciplinary.

• Faulty assumptions and poor reasoning are best corrected through challenge, not presentation. The rhetorical technique of \textit{reductio ad absurdum} (disproving an assertion by exposing an absurdity) works well to discipline student’s minds, not only to correct the problem at hand but also to learn how to detect and correct future errors.

• Important principles should be repeatedly explored and widely applied throughout a course of study, not only to reinforce their importance and help ensure their mastery, but also to showcase the interconnectedness and utility of knowledge.
These learning modules were expressly designed to be used in an “inverted” teaching environment where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student’s understanding to ensure they are truly reasoning and not just memorizing. This is why “Challenge” points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn’t been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

An inspection of these learning modules reveals certain unique characteristics. One of these is a bias toward thorough explanations in the tutorial chapters. Without a live instructor to explain concepts and applications to students, the text itself must fulfill this role. This philosophy results in lengthier explanations than what you might typically find in a textbook, each step of the reasoning process fully explained, including footnotes addressing common questions and concerns students raise while learning these concepts. Each tutorial seeks to not only explain each major concept in sufficient detail, but also to explain the logic of each concept and how each may be developed.

---

1In a traditional teaching environment, students first encounter new information via lecture from an expert, and then independently apply that information via homework. In an “inverted” course of study, students first encounter new information via homework, and then independently apply that information under the scrutiny of an expert. The expert’s role in lecture is to simply explain, but the expert’s role in an inverted session is to challenge, critique, and if necessary explain where gaps in understanding still exist.

2Socrates is a figure in ancient Greek philosophy famous for his unflinching style of questioning. Although he authored no texts, he appears as a character in Plato’s many writings. The essence of Socratic philosophy is to leave no question unexamined and no point of view unchallenged. While purists may argue a topic such as electric circuits is too narrow for a true Socratic-style dialogue, I would argue that the essential thought processes involved with scientific reasoning on any topic are not far removed from the Socratic ideal, and that students of electricity and electronics would do very well to challenge assumptions, pose thought experiments, identify fallacies, and otherwise employ the arsenal of critical thinking skills modeled by Socrates.

3This rhetorical technique is known by the Latin phrase reductio ad absurdum. The concept is to expose errors by counter-example, since only one solid counter-example is necessary to disprove a universal claim. As an example of this, consider the common misconception among beginning students of electricity that voltage cannot exist without current. One way to apply reductio ad absurdum to this statement is to ask how much current passes through a fully-charged battery connected to nothing (i.e. a clear example of voltage existing without current).
from “first principles”. Again, this reflects the goal of developing clear and independent thought in students' minds, by showing how clear and logical thought was used to forge each concept. Students benefit from witnessing a model of clear thinking in action, and these tutorials strive to be just that.

Another characteristic of these learning modules is a lack of step-by-step instructions in the Project and Experiment chapters. Unlike many modern workbooks and laboratory guides where step-by-step instructions are prescribed for each experiment, these modules take the approach that students must learn to closely read the tutorials and apply their own reasoning to identify the appropriate experimental steps. Sometimes these steps are plainly declared in the text, just not as a set of enumerated points. At other times certain steps are implied, an example being assumed competence in test equipment use where the student should not need to be told again how to use their multimeter because that was thoroughly explained in previous lessons. In some circumstances no steps are given at all, leaving the entire procedure up to the student.

This lack of prescription is not a flaw, but rather a feature. Close reading and clear thinking are foundational principles of this learning series, and in keeping with this philosophy all activities are designed to require those behaviors. Some students may find the lack of prescription frustrating, because it demands more from them than what their previous educational experiences required. This frustration should be interpreted as an unfamiliarity with autonomous thinking, a problem which must be corrected if the student is ever to become a self-directed learner and effective problem-solver. Ultimately, the need for students to read closely and think clearly is more important both in the near-term and far-term than any specific facet of the subject matter at hand. If a student takes longer than expected to complete a module because they are forced to outline, digest, and reason on their own, so be it. The future gains enjoyed by developing this mental discipline will be well worth the additional effort and delay.

Another feature of these learning modules is that they do not treat topics in isolation. Rather, important concepts are introduced early in the series, and appear repeatedly as stepping-stones toward other concepts in subsequent modules. This helps to avoid the “compartmentalization” of knowledge, demonstrating the inter-connectedness of concepts and simultaneously reinforcing them. Each module is fairly complete in itself, reserving the beginning of its tutorial to a review of foundational concepts.

This methodology of assigning text-based modules to students for digestion and then using Socratic dialogue to assess progress and hone students' thinking was developed over a period of several years by the author with his Electronics and Instrumentation students at the two-year college level. While decidedly unconventional and sometimes even unsettling for students accustomed to a more passive lecture environment, this instructional philosophy has proven its ability to convey conceptual mastery, foster careful analysis, and enhance employability so much better than lecture that the author refuses to ever teach by lecture again.

Problems which often go undiagnosed in a lecture environment are laid bare in this “inverted” format where students must articulate and logically defend their reasoning. This, too, may be unsettling for students accustomed to lecture sessions where the instructor cannot tell for sure who comprehends and who does not, and this vulnerability necessitates sensitivity on the part of the “inverted” session instructor in order that students never feel discouraged by having their errors exposed. Everyone makes mistakes from time to time, and learning is a lifelong process! Part of the instructor's job is to build a culture of learning among the students where errors are not seen as shameful, but rather as opportunities for progress.
To this end, instructors managing courses based on these modules should adhere to the following principles:

- Student questions are always welcome and demand thorough, honest answers. The only type of question an instructor should refuse to answer is one the student should be able to easily answer on their own. Remember, the fundamental goal of education is for each student to learn to think clearly and independently. This requires hard work on the part of the student, which no instructor should ever circumvent. Anything done to bypass the student’s responsibility to do that hard work ultimately limits that student’s potential and thereby does real harm.

- It is not only permissible, but encouraged, to answer a student’s question by asking questions in return, these follow-up questions designed to guide the student to reach a correct answer through their own reasoning.

- All student answers demand to be challenged by the instructor and/or by other students. This includes both correct and incorrect answers – the goal is to practice the articulation and defense of one’s own reasoning.

- No reading assignment is deemed complete unless and until the student demonstrates their ability to accurately summarize the major points in their own terms. Recitation of the original text is unacceptable. This is why every module contains an “Outline and reflections” question as well as a “Foundational concepts” question in the Conceptual reasoning section, to prompt reflective reading.

- No assigned question is deemed answered unless and until the student demonstrates their ability to consistently and correctly apply the concepts to variations of that question. This is why module questions typically contain multiple “Challenges” suggesting different applications of the concept(s) as well as variations on the same theme(s). Instructors are encouraged to devise as many of their own “Challenges” as they are able, in order to have a multitude of ways ready to probe students’ understanding.

- No assigned experiment or project is deemed complete unless and until the student demonstrates the task in action. If this cannot be done “live” before the instructor, video-recordings showing the demonstration are acceptable. All relevant safety precautions must be followed, all test equipment must be used correctly, and the student must be able to properly explain all results. The student must also successfully answer all Challenges presented by the instructor for that experiment or project.
Students learning from these modules would do well to abide by the following principles:

- No text should be considered fully and adequately read unless and until you can express every idea *in your own words, using your own examples*.

- You should always articulate your thoughts as you read the text, noting points of agreement, confusion, and epiphanies. Feel free to print the text on paper and then write your notes in the margins. Alternatively, keep a journal for your own reflections as you read. This is truly a helpful tool when digesting complicated concepts.

- Never take the easy path of highlighting or underlining important text. Instead, *summarize and/or comment* on the text using your own words. This actively engages your mind, allowing you to more clearly perceive points of confusion or misunderstanding on your own.

- A very helpful strategy when learning new concepts is to place yourself in the role of a teacher, if only as a mental exercise. Either explain what you have recently learned to someone else, or at least *imagine* yourself explaining what you have learned to someone else. The simple act of having to articulate new knowledge and skill forces you to take on a different perspective, and will help reveal weaknesses in your understanding.

- Perform each and every mathematical calculation and thought experiment shown in the text on your own, referring back to the text to see that your results agree. This may seem trivial and unnecessary, but it is critically important to ensuring you actually understand what is presented, especially when the concepts at hand are complicated and easy to misunderstand. Apply this same strategy to become proficient in the use of *circuit simulation software*, checking to see if your simulated results agree with the results shown in the text.

- Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied\(^4\) effort, and never give up!

Students interested in using these modules for self-study will also find them beneficial, although the onus of responsibility for thoroughly reading and answering questions will of course lie with that individual alone. If a qualified instructor is not available to challenge students, a workable alternative is for students to form study groups where they challenge\(^5\) one another.

To high standards of education,

Tony R. Kuphaldt

\(^4\)As the old saying goes, “Insanity is trying the same thing over and over again, expecting different results.” If you find yourself stumped by something in the text, you should attempt a different approach. Alter the thought experiment, change the mathematical parameters, do whatever you can to see the problem in a slightly different light, and then the solution will often present itself more readily.

\(^5\)Avoid the temptation to simply share answers with study partners, as this is really counter-productive to learning. Always bear in mind that the answer to any question is far less important in the long run than the method(s) used to obtain that answer. The goal of education is to empower one’s life through the improvement of clear and independent thought, literacy, expression, and various practical skills.
Appendix C

Tools used

I am indebted to the developers of many open-source software applications in the creation of these learning modules. The following is a list of these applications with some commentary on each.

You will notice a theme common to many of these applications: a bias toward code. Although I am by no means an expert programmer in any computer language, I understand and appreciate the flexibility offered by code-based applications where the user (you) enters commands into a plain ASCII text file, which the software then reads and processes to create the final output. Code-based computer applications are by their very nature extensible, while WYSIWYG (What You See Is What You Get) applications are generally limited to whatever user interface the developer makes for you.

The GNU/Linux computer operating system

There is so much to be said about Linus Torvalds’ Linux and Richard Stallman’s GNU project. First, to credit just these two individuals is to fail to do justice to the mob of passionate volunteers who contributed to make this amazing software a reality. I first learned of Linux back in 1996, and have been using this operating system on my personal computers almost exclusively since then. It is free, it is completely configurable, and it permits the continued use of highly efficient Unix applications and scripting languages (e.g. shell scripts, Makefiles, sed, awk) developed over many decades. Linux not only provided me with a powerful computing platform, but its open design served to inspire my life’s work of creating open-source educational resources.

Bram Moolenaar’s Vim text editor

Writing code for any code-based computer application requires a text editor, which may be thought of as a word processor strictly limited to outputting plain-ASCII text files. Many good text editors exist, and one’s choice of text editor seems to be a deeply personal matter within the programming world. I prefer Vim because it operates very similarly to vi which is ubiquitous on Unix/Linux operating systems, and because it may be entirely operated via keyboard (i.e. no mouse required) which makes it fast to use.
Donald Knuth’s \TeX\ typesetting system

Developed in the late 1970’s and early 1980’s by computer scientist extraordinaire Donald Knuth to typeset his multi-volume magnum opus *The Art of Computer Programming*, this software allows the production of formatted text for screen-viewing or paper printing, all by writing plain-text code to describe how the formatted text is supposed to appear. \TeX\ is not just a markup language for documents, but it is also a Turing-complete programming language in and of itself, allowing useful algorithms to be created to control the production of documents. Simply put, \TeX\ is a programmer’s approach to word processing. Since \TeX\ is controlled by code written in a plain-text file, this means anyone may read that plain-text file to see exactly how the document was created. This openness afforded by the code-based nature of \TeX\ makes it relatively easy to learn how other people have created their own \TeX\ documents. By contrast, examining a beautiful document created in a conventional WYSIWYG word processor such as Microsoft Word suggests nothing to the reader about how that document was created, or what the user might do to create something similar. As Mr. Knuth himself once quipped, conventional word processing applications should be called WYSIAYG (What You See Is All You Get).

Leslie Lamport’s \LaTeX\ extensions to \TeX\

Like all true programming languages, \TeX\ is inherently extensible. So, years after the release of \TeX\ to the public, Leslie Lamport decided to create a massive extension allowing easier compilation of book-length documents. The result was \LaTeX, which is the markup language used to create all ModEL module documents. You could say that \TeX\ is to \LaTeX\ as C is to C++. This means it is permissible to use any and all \TeX\ commands within \LaTeX\ source code, and it all still works. Some of the features offered by \LaTeX\ that would be challenging to implement in \TeX\ include automatic index and table-of-content creation.

Tim Edwards’ \Xcircuit\ drafting program

This wonderful program is what I use to create all the schematic diagrams and illustrations (but not photographic images or mathematical plots) throughout the ModEL project. It natively outputs PostScript format which is a true vector graphic format (this is why the images do not pixellate when you zoom in for a closer view), and it is so simple to use that I have never had to read the manual! Object libraries are easy to create for \Xcircuit, being plain-text files using PostScript programming conventions. Over the years I have collected a large set of object libraries useful for drawing electrical and electronic schematics, pictorial diagrams, and other technical illustrations.
Gimp graphic image manipulation program

Essentially an open-source clone of Adobe’s PhotoShop, I use Gimp to resize, crop, and convert file formats for all of the photographic images appearing in the ModEL modules. Although Gimp does offer its own scripting language (called Script-Fu), I have never had occasion to use it. Thus, my utilization of Gimp to merely crop, resize, and convert graphic images is akin to using a sword to slice bread.

SPICE circuit simulation program

SPICE is to circuit analysis as TeX is to document creation: it is a form of markup language designed to describe a certain object to be processed in plain-ASCII text. When the plain-text “source file” is compiled by the software, it outputs the final result. More modern circuit analysis tools certainly exist, but I prefer SPICE for the following reasons: it is free, it is fast, it is reliable, and it is a fantastic tool for teaching students of electricity and electronics how to write simple code. I happen to use rather old versions of SPICE, version 2g6 being my “go to” application when I only require text-based output. NGSPICE (version 26), which is based on Berkeley SPICE version 3f5, is used when I require graphical output for such things as time-domain waveforms and Bode plots. In all SPICE example netlists I strive to use coding conventions compatible with all SPICE versions.

Andrew D. Hwang’s ePiX mathematical visualization programming library

This amazing project is a C++ library you may link to any C/C++ code for the purpose of generating PostScript graphic images of mathematical functions. As a completely free and open-source project, it does all the plotting I would otherwise use a Computer Algebra System (CAS) such as Mathematica or Maple to do. It should be said that ePiX is not a Computer Algebra System like Mathematica or Maple, but merely a mathematical visualization tool. In other words, it won’t determine integrals for you (you’ll have to implement that in your own C/C++ code!), but it can graph the results, and it does so beautifully. What I really admire about ePiX is that it is a C++ programming library, which means it builds on the existing power and toolset available with that programming language. Mr. Hwang could have probably developed his own stand-alone application for mathematical plotting, but by creating a C++ library to do the same thing he accomplished something much greater.
Another open-source tool for mathematical visualization is gnuplot. Interestingly, this tool is not part of Richard Stallman’s GNU project, its name being a coincidence. For this reason the authors prefer “gnu” not be capitalized at all to avoid confusion. This is a much “lighter-weight” alternative to a spreadsheet for plotting tabular data, and the fact that it easily outputs directly to an X11 console or a file in a number of different graphical formats (including PostScript) is very helpful. I typically set my gnuplot output format to default (X11 on my Linux PC) for quick viewing while I’m developing a visualization, then switch to PostScript file export once the visual is ready to include in the document(s) I’m writing. As with my use of Gimp to do rudimentary image editing, my use of gnuplot only scratches the surface of its capabilities, but the important points are that it’s free and that it works well.

Python programming language

Both Python and C++ find extensive use in these modules as instructional aids and exercises, but I’m listing Python here as a tool for myself because I use it almost daily as a calculator. If you open a Python interpreter console and type from math import * you can type mathematical expressions and have it return results just as you would on a hand calculator. Complex-number (i.e. phasor) arithmetic is similarly supported if you include the complex-math library (from cmath import *). Examples of this are shown in the Programming References chapter (if included) in each module. Of course, being a fully-featured programming language, Python also supports conditionals, loops, and other structures useful for calculation of quantities. Also, running in a console environment where all entries and returned values show as text in a chronologically-ordered list makes it easy to copy-and-paste those calculations to document exactly how they were performed.
Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License (“Public License”). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. Adapter’s License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or
limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

**Section 2 – Scope.**

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

   A. reproduce and Share the Licensed Material, in whole or in part; and
   
   B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.
For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated):

ii. a copyright notice;
iii. a notice that refers to this Public License;
iv. a notice that refers to the disclaimer of warranties;
v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter’s License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,
whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully
be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.
Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the “Licensor.” Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark “Creative Commons” or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.
Appendix E

References


lnk_msp430g2553.cmd, linker command file, version 1.198, Texas Instruments Incorporated, 2016.


“MSP430G2553 LaunchPad Development Kit” user’s guide, document number SLAU772, Texas Instruments Incorporated, June 2018.


Appendix F

Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

7 September 2022 – added a Case Tutorial section showing the use of pointers in a simple add/subtract C program.

21 March 2022 – added some clarifications to the first Case Tutorial example regarding bitwise operators, where C is used to read an input pin.

4 March 2022 – minor edits to the Case Tutorial section on blinking two LEDs using C.

2 February 2022 – minor edits to some instructor notes.

30 November 2021 – corrected a LaTeX typo, where I failed to include an escape character before a pound symbol in one of the instructor notes. This error went undetected when compiling the document for student use, but failed when I tried to compile an instructor version.

26 November 2021 – minor edit to a footnote.

14-15 November 2021 – eliminated reference to memory address locations for variables in the “C example: bitwise operations” Case Tutorial because they were really not germane to the topic at hand. Also added another section to the Tutorial on bit-level input instructions, another section on the watchdog timer, and a new Case Tutorial showing use of the watchdog timer in a trivial program.

9 November 2021 – corrected typographical errors in image_5445 and image_5446. In both cases I mistakenly used the 0x prefix when I should have used 0b because the port control register values were in binary and not hexadecimal.

7 September 2021 – minor typographical error corrections.
26 August 2021 – minor edits to the “Elementary output and input” section of the Tutorial. Also, moved the “Writing to output pins” subsection to become a new section following “Elementary output and input” called “Bit-level output instructions”.

22 August 2021 – added commentary to some of the Case Tutorial examples. Also added “Integer conversion table” Quantitative Reasoning question, and made some minor typographical error corrections.

20 June 2021 – added more content to the Tutorial on starting new projects, especially the importance of selecting the proper “target” microcontroller.

14 June 2021 – added to one of the existing Case Tutorial sections on PWM, showing how an RC network may be used to produce a pseudo analog output channel.

13 June 2021 – added another Case Tutorial section, on crude analog input demonstration.

10 May 2021 – commented out or deleted empty chapters.

3 May 2021 – added BJT-only solution to the “Example: interposing MCU to a heavy load” Case Tutorial.

28 April 2021 – added more explanatory text to the Case Tutorial example “C example: Boolean SOP expression”.

12 April 2021 – added two new Case Tutorial examples. One shows analog-digital conversion of a voltage signal and reporting of the ADC count value using the UART. The other takes this concept further by showing how the microcontroller may be used as a simple datalogger, storing 128 sampled analog voltage signals in an array and transmitting that stored data in comma-separated variable (CSV) format using the UART.

11 April 2021 – corrected code error in two Case Tutorial examples showing how to use the UART. The error was a single ampersand (&) rather than a double-ampersand (&&).

6 April 2021 – edited the Case Tutorial example on interposing heavy loads to show commutating diodes and optoisolators.

5 April 2021 – added a new Case Tutorial section showing MCU driving two BJTs as a sine-wave DC-to-AC converter.

29 March 2021 – added a reference to the intrinsic function `delay_cycles()` in the “C example: alternating LED blink” Case Tutorial. Also fixed a code error in the “C example: pushbutton-triggered timer” where the value of `tenths` was being limited to 0xFF rather than 0xFFFF as it should have been. Also added a footnote to the timer Case Tutorial explaining how to adjust the `TACCR0` register for super-fine adjustment of the $\frac{1}{10}$ second interrupt period.

25 March 2021 – improved and expanded upon code examples in the Case Tutorial section “C example: pushbutton-triggered timer”. Also added a new Case Tutorial section showing how to transmit both letter and number characters in ASCII format via the UART.
17 March 2021 – added more commentary to the Tutorial chapter regarding the default functions of pins P2.6 and P2.7.

12 March 2021 – added a new Case Tutorial section showing MCU driving BJT driving MOSFET.

6-7 March 2021 – added a Case Tutorial section showing UART transmission of ASCII text.

22-25 February 2021 – added a Case Tutorial section showing pushbutton-based interrupt in C. Also added some clarifying text in the Tutorial chapter regarding the same topic. Additionally, added some explanatory text about the use of BIT macros pre-defined in the msp430.h header file.

19 February 2021 – added a Case Tutorial section showing simple stack operations, and also added some content to a couple of existing Case Tutorial examples.

3 February 2021 – added instructor notes, and also a pagebreak prior to the "Improving a debug session". Also added decoupling capacitors to the motor start/stop Case Tutorial.

30 January 2021 – added new Case Tutorial section showing how to use the microcontroller to synthesize a sine wave, based on code written by Jason Wahl on 29 January 2021.

29 January 2021 – divided part of the I/O subsection into its own subsection, discussing in detail how to set and clear output pin bits.

26 January 2021 – modified schematic diagram in Case Tutorial example showing how to drive an electric motor. I added a relay to the transistor interposing in order to provide greater electrical isolation between the motor and the MCU.

11 December 2020 – modified the Case Tutorial section on start-stop control, to include a schematic showing how a transistor could interpose between the MCU and an electric motor.

22 September 2020 – corrected typographical error in the C source code for the bitwise operations Case Tutorial example, courtesy of Tysha Eisele.

18 September 2020 – corrected errors on pin diagrams for all Case Tutorial schematics, where I failed to show the active-low Reset pin tied high!

15 September 2020 – moved interrupt section bullet-list into a table.

8 September 2020 – minor typographical correction in “Setting up Port 1 I/O” question.

7 September 2020 – added Case Tutorial sections on pulse-width modulation output using Timer A, analog-digital conversion using Timer A interrupt, and motor start-stop control (all written in C).

2 September 2020 – fixed typographical error in the “subtracting two numbers” assembly-language example within the Case Tutorial chapter. Also, added a new “Debugging tools” section. Fixed another typographical error in the I/O subsection as well (wrong hex representation for a binary number).
30 August 2020 – added screenshots showing set-up of C project in Code Composer Studio.

29 August 2020 – significantly edited the Introduction chapter to make it more suitable as a pre-study guide and to provide cues useful to instructors leading “inverted” teaching sessions.

25 August 2020 – added screenshots showing set-up of Energia project in Code Composer Studio.

18 August 2020 – added Case Tutorial example for Boolean SOP function written in C.

18 July 2020 – added Case Tutorial examples written in both C and Sketch to add two numbers.

8 July 2020 – added Case Tutorial example for a program coded in Sketch rather than C or assembly.

26 June 2020 – added Case Tutorial example some programs coded in C rather than assembly.

23-24 June 2020 – added Case Tutorial example showing external (Port 1) interrupt, and a new Tutorial section on interrupt handling for the MSP430 microcontroller series.

12-17 June 2020 – added more Case Tutorial examples, Tutorial content, and questions.

11 June 2020 – document first created.
Index

Absolute addressing mode, 8, 34, 171
Accumulator, 178
Adding quantities to a qualitative problem, 214
Address resolution, 182
Addressing mode, 8, 34, 171
Addressing, relative, 185
ALU, 136
Annotating diagrams, 213
Arduino, 124
Arithmetic Logic Unit, 136
Assembly code, 178

Bitwise operation, 131

Calling a subroutine, 184
CCS, 124
Checking for exceptions, 214
Checking your work, 214
Clock, 183
Code Composer Studio, 124
Code, computer, 221
Comment, 182
Commutating diode, 43

Debug mode, 152, 155
Debug session, properly closing, 162, 176
Delay, 183
Dimensional analysis, 213
Diode, commutating, 43
Directive, 181
Disassembler, 185

Edwards, Tim, 222
Endianness, 137, 179
Energia, 124
eZ-FET, 124

Fetch/execute cycle, 183

General Interrupt Enable bit, 141, 142
GIE bit, 141, 142
Graph values to solve a problem, 214
Greenleaf, Cynthia, 191

Harvard architecture, 135
Hex dump, 180
How to teach with these modules, 216
Hwang, Andrew D., 223

I/O port, 125
Identify given data, 213
Identify relevant principles, 213
Immediate addressing mode, 34
Inductive kickback, 43
Instructions for projects and experiments, 217
Intermediate results, 213
Interposing, 43
Interrupt, 140, 141, 187, 199
Interrupt service routine, 187, 199
Interrupt vector, 141
Inverted instruction, 216
ISR, 140, 141, 187, 199

Jump instruction, 179

Kickback, inductive, 43
Knuth, Donald, 222

Label, 182
Lamport, Leslie, 222
LaunchPad, 124
Limiting cases, 214
Little-endian, 137, 179

Machine code, 178
Mask, 132, 140, 142, 199
Maskable interrupt, 142
Metacognition, 196
Microcontroller, 119
Microprocessor, 119
Moolenaar, Bram, 221
MOS 6502 microprocessor, 178
Murphy, Lynn, 191

NMI, 142
Non-maskable interrupt, 142

Opcode, 179
Open-source, 221
Operand, 179
Optimization, compiler, 157

P2SEL, 140
Pin assignments, MSP430G2553IN20, 123
Popping from the stack, 185, 187
Port, I/O, 125
Problem-solving: annotate diagrams, 213
Problem-solving: check for exceptions, 214
Problem-solving: checking work, 214
Problem-solving: dimensional analysis, 213
Problem-solving: graph values, 214
Problem-solving: identify given data, 213
Problem-solving: identify relevant principles, 213
Problem-solving: interpret intermediate results, 213
Problem-solving: limiting cases, 214
Problem-solving: qualitative to quantitative, 214
Problem-solving: quantitative to qualitative, 214
Problem-solving: reductio ad absurdum, 214
Problem-solving: simplify the system, 213
Problem-solving: thought experiment, 213
Problem-solving: track units of measurement, 213
Problem-solving: visually represent the system, 213
Problem-solving: work in reverse, 214
Processor, 119
Program counter, 184
Pushing to the stack, 185, 187

Qualitatively approaching a quantitative problem, 214

Reading Apprenticeship, 191

Reductio ad absurdum, 214–216
Register, 178
Relative addressing, 185
Release mode, 152, 155
Return from subroutine, 184

Schoenbach, Ruth, 191
Scientific method, 196
setup(), 12
Shift register, 187
Simplifying a system, 213
Socrates, 215
Socratic dialogue, 216
SPICE, 191
Stack, 185, 187
Stallman, Richard, 221
Status register, 188, 189
Subroutine, 141, 184
Symbolic addressing mode, 34
Symbols, 42, 73, 80, 133
System on a chip, 148

Thought experiment, 213
Time delay, 183
Torvalds, Linus, 221

Unconditional jump, 179
Units of measurement, 213

Visualizing a system, 213
Von Neumann architecture, 135

Work in reverse to solve a problem, 214
WYSIWYG, 221, 222