

# MSP430 Microcontroller Engineering Guide

## Getting Started

By Tomislav N. Krnich  
B. Sc. in Physics

A book about the basic knowledge you need to have for developing programs for the MSP430. It is written in a bright, clear, and down-to-earth language for hobbyists, inventors, technicians, engineers, and product managers. It begins with presenting the MSP430's basic set of systems, their purposes, and how it starts, runs, goes to sleep, interrupted from sleep, performs work, and then goes back to sleep. Focus is on code and program development: accessing registers, the basic approach for developing a program, a programming reference model for getting oriented, the two basic patterns of program development, the most common programming routines and practices, the various types of input interruption signals which tell this microcontroller which interrupt service routine to use for carrying out work and producing output signals, and how to write the code for those routines. It is fully illustrated, indexed, and presents numerous programming examples. Included are many helpful tips. Ideal for self-paced, individualized learning. All examples are written in the C Programming Language.



---

# **MSP430 Microcontroller Engineering Guide**

---

## **Getting Started**

By Tomislav N. Krnich  
B.Sc. in Physics

Written, illustrated, edited, printed, bound, and published by the author.

Internetpress<sup>®</sup>

TOMISLAV N. KRNICH (1961- ) was born in Canada and then immigrated to the United States of America where he earned a Bachelors of Science degree in Physics with a minor in U.S. National Security Policy from Georgia State University. He has worked in the civil and defense aviation industries as an engineer, and in the telecom-munications, internet, information systems, and automotive industries as a solution architect, systems analyst, and technical writer. Embedded computing systems are one of his special interests.

Internetpress, Los Gatos, California 95032

<http://internetpress.com>

© 2022 by Tomislav N. Krnich

All rights reserved. Published August 2022

The author is grateful for your purchase. Please obey international copyright laws since the sale of this book provides an income for the author. Producing copies of this book for distribution is not permitted. Only the author has that right. You may contact the author through e-mail at [REDACTED].

ISBN (vol 1): 978-0-9985736-0-1 (paperback)

- 81 Program Examples Written in the C Programming Language
- 90 Diagrams
- 29 Chapters
- Includes a Comprehensive Index

This book was produced in the United States of America.

While every precaution has been taken in the preparation of this book, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained in this book.

Internetpress® is a registered trademark that is owned by the author.

This book is printed on acid free paper.

*Do not treat others in ways that you would not like to be treated - the golden rule.*

---

<b>1</b>	<b>Introduction</b>	1
	Data Processing as the Highest View of Handling Digital Data	1
	Microcontrollers are Built of System Modules and Peripheral Modules	2
	The Historical Emergence of the Microcontroller	2
	Appearance of the MSP430 Microcontroller	3
	Why use the MSP430?	3
	Purpose of this Book	3
	Firmware is the Program of Instructions We Develop for a Microcontroller	4
	Bits, Bytes, and the Native Word are the Basic Units of Data	4
	How the Microcontroller Views Data	5
	The MSP430 is Offered with Two Types of Processors: CPU and the CPUX	6
	How We [as Developers] may View and Express Data	6
	Registers are Used for Configuring and Controlling the Microcontroller	6
	Main Memory Registers	7
	What is a Main Memory Register used for?	8
	CPU Registers	8
	Summary	9
<b>2</b>	<b>Visualizing How the MSP430 Operates</b>	11
	Basic View	12
	Power-up View	12
	Event-Driven Views	13
	View of an Internally Occurring Event	13
	Steps 1 and 2	14
	CPU Interrupt System Behavior	14
	Step 3	15
	Steps 4, 5, 6, and 7	15
	C, Assembly, and the Final ISR Instruction	15
	Step 8	15
	View of an Externally Occurring Event	16
	Steps 1 and 2	16
	Steps 3, 4, and 5	17
	Steps 6 and 7	17
	Steps 8, 9, 10, and 11	17
	Step 12	17
	Functional Block Diagram View	17
	Memory Modules	18
	Memory Buses	19
	Power Management Module	19
	CPU, EEM, JTAG, and SBW Modules	20
	Clock System Module	21
	ADC Module	21
	MPY32 Module	22
	I/O Ports	22
	Digital I/O Module	24
	SYS Module	24
	CRC16 Module	24
	Timer Modules	25



<i>...continuation of the Functional Block Diagram View</i>	
Serving as a Frequency Dependent Timer	26
Serving as a Frequency Independent Counter	26
Serving as a Module for Measuring Rates	26
Serving as a Module for Producing PWM Voltage Signals	27
How the Functional Diagram Describes the Timer Module	28
eUSCI Module	28
LPM3.5 Domain	29
RTC Counter Module	30
BAKMEM Module	30
Pin Designation View	31
Module Functional View	32
<hr/>	
<b>3 Visualizing the Main Memory</b>	35
Main Memory Structure	35
CPU Memory Structure	36
Introduction to Register Tables	37
<hr/>	
<b>4 The Reset System and its Subsystems</b>	39
Power-Up	39
Reset	39
The BOR, POR, and PUC Sequence	40
Register Table Bitfields	40
Reset Signals	40
BOR Signals	40
False BOR Signals	41
POR Signals	41
PUC Signals	41
<hr/>	
<b>5 How to Read and Use the Register Tables</b>	43
The Conventional Register Table	43
Register Variable and Register Name	43
Register Variable	44
Opening the Header File where the Register Variables are Declared	45
Register Bitfields	45
Bitfield Mask	46
Bitfield Mask Suffix	46
The Standard Bits	47
Register Bit Accessibility and Initial Condition	47
Bitfield Descriptions	48
Interrupt System Bitfields	49
Using a Register Table and Functional View to Help Develop Code	49
Distinguishing between a Digital I/O Module and Port	50
Digital I/O Module	50
Port	51
Port Register Tables	52
Port Channels, Port Register Bitfields, and Port Register Bitfield Masks	52
First Type of Port Register Table	52
Second Type of Port Register Table	54
Third Type of Port Register Table	56

<b>6</b>	<b>Code Composer Studio Usage Tips</b> .....	59
	Forcing the CCS Debugger to Step through Each Instruction .....	59
	Configuring the Variables View for a Different Numbering Format .....	60
<b>7</b>	<b>How to Write into a Register</b> .....	61
	Our Model Register .....	61
	Masking Concepts .....	62
	Overview of the Setting, Clearing, and Toggling Operations .....	64
	Setting Bits in a Register .....	65
	Setting a Single Bit .....	65
	Combining Masks to Create a Single Mask .....	66
	Setting Multiple Bits .....	66
	Clearing Bits in a Register .....	67
	Clearing a Single Bit .....	67
	Clearing Multiple Bits .....	68
	Simultaneously Setting and Clearing Bits in a Register .....	69
	Toggling Bits in a Register .....	70
	Toggling a Single Bit .....	70
	Toggling Multiple Bits .....	71
	Just Simply Writing a Number into a Register .....	71
	Writing into Password Protected Registers .....	72
<b>8</b>	<b>How to Declare a Storage Variable</b> .....	75
	A Description for the Storage Variable .....	75
	Declaring Storage Variables .....	76
<b>9</b>	<b>How to Read a Register</b> .....	77
	The Process .....	77
	The Code .....	77
<b>10</b>	<b>Background for Testing the Contents of a Register</b> .....	79
	Integer Constants .....	79
	The MSP430 Relaxed Compiler .....	79
	Using Binary Notation .....	80
	Enabling CCS Support for GCC Extensions .....	80
<b>11</b>	<b>How to Test the Contents of a Register</b> .....	81
	The Process .....	81
	The Code .....	81
<b>12</b>	<b>How to use a Pointer to Read and Write into Main Memory</b> .....	83
	Pointer and Pointer Variable .....	83
	Indirection Operator .....	83
	Converting an Address Number into a Pointer: the Pointer Expression .....	84
	Declaring a Pointer Variable .....	85
	Declaring a Pointer and Assigning it to a Pointer Variable .....	85
	Reading Data .....	86
	Using a Pointer .....	86
	Using a Pointer Variable .....	87
	Writing Data .....	87
	Using a Pointer Macro .....	88

<b>13 Watchdog Timer and Putting it on Hold</b> .....	89
Purpose .....	89
Basic Operation .....	89
Interval Reset Instruction .....	90
Watchdog Control Register Table .....	90
Stopping the Watchdog Timer .....	94
Writing an Active Watchdog Timer Handler .....	95
Stopping the Watchdog Timer during the Boot Process .....	95
Using Boot Hook Functions to Stop the Watchdog and Execute other Instructions ..	95
Using the <code>_system_pre_init()</code> function .....	96
Using the <code>_system_post_cinit()</code> function .....	96
Reading the Watchdog Timer Register .....	96
<hr/>	
<b>14 main() Function</b> .....	99
Purpose .....	99
How the main() Function is Called .....	99
Syntax and Format for a C Language Function .....	99
The Two Standard Syntaxes for the main() Function .....	100
First Syntax and Format .....	100
main() is Void of Parameters .....	100
The return Statement .....	100
Second Syntax and Format .....	101
<hr/>	
<b>15 Program Development Nomenclature</b> .....	103
Routines .....	103
Subroutines .....	103
Block of Instructions .....	103
Logic Circuits as Routines and Subroutines .....	103
Service .....	103
<hr/>	
<b>16 Structures for Program Development</b> .....	105
Sequence Structure .....	105
Selection Structures .....	106
Repetition Structures .....	106
<hr/>	
<b>17 Basic Approach for Developing a Microcontroller Solution</b> .....	107
Task 1: Conceptualize the Problem and its Solution .....	107
Task 2: Design the Power Supply Interface Circuit .....	107
Task 3: Design the Signal Input Interface Circuit .....	107
Task 4: Develop Instructions which Configure the Signal Input Path .....	107
Task 5: Develop Instructions which Configure the Input Module .....	108
Task 6: Develop Instructions which Make Decisions .....	108
Task 7: Develop Instructions which Act on the Result of a Decision .....	108
Task 8: Develop Instructions which Configure the Output Module .....	108
Task 9: Develop Instructions which Configure the Signal Output Path .....	108
Task 10: Design the Signal Output Interface Circuit .....	108

<b>18 MSP430 Reference Model</b> .....	109
Structural Overview .....	109
Input Signal Stack .....	110
Externally Occurring Events .....	111
Internally Occurring Events .....	112
Output Signal Stack .....	112
Power Supply Stack .....	113
<hr/>	
<b>19 Patterns for Program Development</b> .....	115
Repetitive-Driven Pattern of Program Development .....	115
Configure and Setup Sequence .....	116
Watchdog Timer Handler .....	116
Oscillator Settling Handler .....	116
Signal I/O Multiplexing .....	117
Configure the System and Peripheral Modules .....	117
Unlock Digital I/O Port Channels .....	117
Reset Fault Handler .....	117
The Repetitive Sequential Routine .....	118
The Repetitive Selection Routine .....	119
Event-Driven Pattern of Program Development .....	120
System Configuration and Setup Sequence .....	121
Watchdog Timer Handler .....	121
About the Next six Routines .....	122
Oscillator Settling Handler .....	122
Signal I/O Multiplexing .....	122
Configure the System and Peripheral Modules .....	122
Unlock the Digital I/O Port Channels .....	123
Port Channel Interrupt Flag Handler .....	123
System Reset Fault Handler .....	123
Enable Maskable Interruptions .....	124
Volatile Data Handler .....	124
Enter a Low Power Operating Mode .....	125
Input Signal Sequence .....	125
Decision and Output Signal Sequence .....	125
<hr/>	
<b>20 Placing the Event-Driven Pattern into a Larger Context</b> .....	127
Physical Interfacing and Power-Up .....	129
Supply Voltage Supervision .....	130
Operating Mode Diagram .....	131
Reset Routines .....	134
Brownout Reset (BOR) .....	135
Power-On Reset (POR) .....	136
Power-Up Clear (PUC) .....	137
MSP-BSL and boot.c .....	138
MSP-BSL .....	139
Boot Program .....	139
Boot Program Execution .....	140
Initializing the Program Execution Stack .....	141
Initializing the Memory Protection Unit .....	141
Execute a Pre-Initialization Function .....	141
Initialize Global Variables .....	142
Execute a Post Initialization Function .....	142
Call the main() Function .....	142

main() Function	142
CPU Interruption	142
Preprocessing Translation Units	143
Conventional C Translation Units	144
#include Preprocessor Directives	144
#define Preprocessor Directives	145
Define Global Variables	145
Define Conventional Functions	146
MSP430 Translation Units	146
#include <msp430.h>	146
#include Specialized MSP430 Library Headers	146
Define a Pre-initialization Boot Hook Function	147
Define a Post initialization Boot Hook Function	147
Define an Interrupt Service Routine (ISR)	147
Define #pragma Directives	147
<b>21 Repetitive-Driven Programming Examples</b>	<b>149</b>
Development Tools	149
Repetitive Sequential Pattern	149
Pseudo Code Template	149
Using a Repetitive Sequence to Produce an Output Signal	150
Model Use Case	150
Interfacing Circuit	150
Output Signal Pathway	151
The Program	153
Block of System Setup Instructions	153
Block of Repetitive Sequential Instructions	154
Repetitive Selection Pattern	155
Pseudo Code Template	155
Using an External Input Signal for Selecting an Output Signal	155
Model Use Case	155
Interfacing Circuits	156
Switch Circuit and Its Operation	157
First State Operation: Switch is Open	158
Second State Operation: Switch is Closed	159
LED Circuit	159
The Program	159
Block of System Setup Instructions	160
Block of Repetitive Selection Instructions	161
Using an Internal Input Signal for Selecting an Output Signal	161
Model Use Case	161
Interfacing Circuits	162
Input Interfacing Circuit	162
Output Interfacing Circuits	162
Signal Pathways	162
Input Path from the Temperature Sensor	164
PMM Section	164
ADC Section: Input Signal Multiplexing	165
ADC Control Signals	166
Voltage Measurement Scale	166
Clock Conversion Signal	166
SAMPCON and SHI Signals	166
ADCENC and ADCON Signals	167
Output Paths to LED1 and LED2	167

*...continuation of Using an Internal Input Signal for Selecting an Output Signal*

The Program	168
Block of System Setup Instructions	168
Write a Watchdog Timer Handler	168
Setup the Sensor to Create Input Signals	168
Setup the ADC as the Input Signal Peripheral Module	169
Setup the Digital I/O as the Output Signal Peripheral Module	170
Declare Variables and Constants	170
Block of Repetitive Selection Instructions	172
<b>22 Event-Driven Programming Routines and Practices</b>	<b>175</b>
Boot Initialization	175
Pre-Initialization	175
Post Initialization	175
Manipulating Bits in Password Protected Registers	176
Password that Protects a Single Register	176
Password that Protects a Set of Registers	177
Watchdog Timer Handlers	178
Placing the Watchdog on Hold	178
Using Watchdog Mode	178
The Counter Interval	179
The Pattern	179
The Code Examples	180
Oscillator Settling Handler	182
Signal Path from an External Oscillator to the Fault Detector	183
Code Example for the Oscillator Fault Handler	184
Configuring a Port Channel	187
Configuring as a GPIO Input for Sensing a Signal Changing from Low to High	190
Configuring as a GPIO Input for Sensing a Signal Changing from High to Low	191
Configuring a Channel as a Non-GPIO Function	192
Configuring a Port Channel as Unused	197
Accessing Protected Registers	197
Watchdog Registers	197
Power Management Module Registers	197
Clock System Registers	198
Memory Protection Unit (MPU) Registers	198
Configuring Unused Port Channels	199
Code Example for Putting a Port Channel into a High Impedance State	198
Unlocking Modules & Digital Port Channels which are in the LPMx.5 Domain	201
Port Channel Interrupt Flag Handler	202
Clearing a Port Channel Flag from Inside of an ISR	202
Determining the Source of an Interrupt Flag	203
Interrupt Vector	203
Flow of Execution from a Set Flag to an ISR or an RFH	203
Basic Flow for the Non-Maskable and Maskable Interruptions	204
Basic Flow for the Reset Interruption	204
Flag to Routine Relationships	205
Flag to Reset Fault Handler (RFH) Relationship	205
Flag to ISR Relationships	206
Flag Determining Code Examples	207
Using the if() Statement	207
Using the switch() Statement	208
The <code>__even_in_range()</code> Function	208

...continuation of *Using the switch() Statement*

Code Example for using the <code>switch()</code> to Determine which Flag is Set . . . . .	209
The PxIV Register Table . . . . .	209
The <code>switch()</code> Code . . . . .	210
Conventional Register Scenario . . . . .	211
Enabling and Disabling Maskable Interruptions . . . . .	212
Unlocking and Locking FRAM . . . . .	213
Review of Volatile and Non-Volatile Memory . . . . .	213
FRAM Access Control . . . . .	213
Example for Unlocking and Locking FRAM . . . . .	214
Register which Controls the FRAM . . . . .	214
Code Example . . . . .	216
Volatile Data Handler . . . . .	217
Using the <code>PERSISTENT() #pragma</code> to Protect Volatile Data . . . . .	218
Code Example for Protecting a Single Variable . . . . .	218
Code Example for Protecting the Variables in an Array . . . . .	219
Using the Backup Memory Registers . . . . .	220
Determining How Much Memory is Consumed . . . . .	221
Entering a Low Powered Operating Mode . . . . .	222
Conventional Lower Powered Operating Modes . . . . .	222
Fractional Lower Powered Modes (LPMx.5) . . . . .	223
Delay Function . . . . .	225
<hr/>	
<b>23 Interrupt Handling and Interrupt Vectors . . . . .</b>	<b>227</b>
CPU Interruptions are Event-Driven . . . . .	227
Event Monitoring Blocks . . . . .	227
Conventional Flag Registers and Interrupt Vector Registers . . . . .	227
The Interrupt Service Routine and Vector . . . . .	228
Interrupt Vectors . . . . .	228
Block of Interrupt Control Logic . . . . .	229
Reset, Non-Maskable (NMI), and Maskable Types of Interruptions . . . . .	229
How the Interruption is Processed . . . . .	230
Transfer of Program Execution to the ISR . . . . .	231
Execution while inside of the ISR . . . . .	231
Transfer of Program Execution from the ISR back to the Low Powered State . . . . .	232
Interrupt Prioritization . . . . .	232
Interrupt Compare Controller (ICC) . . . . .	233
<hr/>	
<b>24 How to Determine which are the Multi-Flagged Vectors . . . . .</b>	<b>235</b>
Determining which Vectors are Bound to More than one Flag . . . . .	235
Finding the Name of the Register where the Flag is Located . . . . .	235
<hr/>	
<b>25 The Reset Interruption . . . . .</b>	<b>237</b>
Flow for the System Reset Interruption . . . . .	238
The Reset Fault Handler (RFH) . . . . .	239
Reset Fault Handler Based on <code>if()</code> Statements . . . . .	241
Reset Fault Handler Based on a <code>switch()</code> Statement . . . . .	242
Reset Caused by a Watchdog Timer Overflow . . . . .	244

---

<b>26 How to Write an Interrupt Service Routine (ISR)</b> .....	245
The Conventional ISR .....	245
Built-in Default Interrupt Service Routine (ISR) .....	247
Customized Default Interrupt Service Routine (ISR) .....	248
<hr/>	
<b>27 Non-Maskable Interruption (NMI)</b> .....	249
Flow for the Non-Maskable Interruption (NMI) .....	250
Non-Maskable ISR Examples .....	252
main() Function for ISR Code Examples 76 and 77 .....	253
Putting the RST/NMI Pin into NMI Mode .....	254
Configuring P1.0 to Drive the LED .....	256
Final Instructions for main() .....	256
ISR which uses the if() Selection Statement to Determine the NMI Flag .....	257
The ISR's Behavior .....	257
Writing the ISR .....	257
Getting the Vector's Name .....	257
Binding the Vector to the ISR .....	258
The ISR's Signature .....	258
The First if() Selection Statement .....	258
The Second if() Selection Statement .....	259
Returning the Flow of Execution Back to where it was Interrupted .....	259
ISR which uses the switch() Selection Statement to Determine the NMI Flag .....	260
The ISR's Behavior .....	260
Writing the ISR .....	260
Getting the Vector's Name .....	260
Binding the Vector to the ISR .....	260
The ISR's Signature .....	261
Getting the IVR Register Variable and its Codes .....	261
The switch() Statement .....	263
Returning the Flow of Execution Back to where it was Interrupted .....	264
<hr/>	
<b>28 Maskable Interruption</b> .....	265
Flow for the Maskable Interruption .....	266
About this Chapter's Examples .....	268
ISR using the if() Selection Statement to Determine which Maskable Flag is Set .....	269
The main() function .....	270
Configuring P2.3 and P2.7 to Sense Input Signals .....	270
Configuring P1.0 and P1.1 to Produce Output Signals .....	271
Final Instructions for main() .....	271
The ISR's Behavior .....	272
Writing the ISR .....	272
Port Channel Flag Names .....	272
Getting the Interrupt Vector Name .....	272
Binding the Vector to the ISR .....	273
The ISR's Signature .....	273
How Many if() selection Statements to Use .....	273
The First if() Selection Statement .....	273
The Second if() Selection Statement .....	273
Returning the Flow of Execution Back to where it was Interrupted .....	274



ISR using the switch() Selection Statement to Determine which Maskable Flag is Set	274
The main() Function	274
The ISR's Behavior	275
Writing the ISR	275
Getting the Port Channel Flag Names	275
Getting the Interrupt Vector Name	276
Binding the Vector to the ISR	277
The ISR's Signature	277
How Many case selection Statements to Use	277
The First Case	277
The Second Case	278
Returning the Flow of Execution Back to where it was Interrupted	278
<hr/>	
<b>29 Interruption from Fractional Low Powered Mode (LPMx.5)</b>	279
Flow for the LPMx.5 Interruption	282
Flow for the LPMx.5 Interrupt Service Routine (ISR)	284
Program Example	285
Circuit Schematic for the Program	286
How the Program Example Works	286
Structure of the Program Example	287
Program Example	288
<hr/>	
<b>Index</b>	293

The MSP430 microcontroller is typically used for monitoring and controlling devices, but it is not limited to just that type of work. It is a general purpose data processing machine, and such a machine is called a digital computer. It reads data from a device, it then processes the data to make decisions, and the results of those decisions produces data which then is used for controlling, driving, or communicating with a device. Furthermore, a single MSP430 is not limited to interacting with a single device. It can simultaneously handle many devices.

Data which flows into and out of the MSP430 are in the form of voltage signals which flow at low rates of amperes. The voltages range from 0 to 5 volts of direct current (VDC). The amperes range from 0 to about 10 milliamperes.

There are a very wide variety of devices which the MSP430 can monitor and control, and all those devices can produce and receive a wide variety of analog and digital voltage signals. Therefore, we can perceive the MSP430 as being a type of computer called a mixed signal processor. But not just a mixed signal processor, it is a stored program mixed signal processor. That means it will handle and produce many different types of analog and digital signals while under the control of a computer program.

Signals which enter the MSP430 are converted to binary numbers so the signals can be understood by it and processed, in other words, so decisions can be made. The results of the decisions are also in the form of and stored as binary numbers. The result of a decision is typically used for producing an output signal. Binary numbers, which form a decision, are then converted to the appropriate type of analog or digital output signal.

The binary numbers which are processed (meaning, analyzed, manipulated, and moved around inside of the microcontroller) are referred to as binary data or digital data. Electronic circuits which handle digital data are called digital logic circuits. The work carried out by such circuits is called data processing.

---

### **Data Processing as the Highest View of Handling Digital Data**

Data processing involves computation and movement of digital data. The set of circuits which handle computation is called the arithmetic logic unit (ALU). The set of circuits which handle the movement of data, so the ALU can access it, is called control logic. The combination of these computation and movement circuits is called a data processing unit, and when built into a digital computer, they are called a central processing unit (CPU).

A CPU needs instructions in order to know where to get data, to know which computations to perform on the data, and where to put the resulting data. Those instructions

are in the form of a program which is loaded in the computer's memory. The CPU uses its control logic for locating the beginning of the program, and then sequentially reads and executes each instruction in the program. These steps are called the fetch and execute cycle. A CPU which is under the control of a program is called a microprocessor.

Besides fetching and executing instructions, the CPU control logic also handles the work needed for transferring data back into memory and for controlling blocks of digital logic circuits which support the microprocessor in specific ways. Those blocks are called modules. There are two types of modules: system modules and peripheral modules.

---

### **Microcontrollers are Built of System Modules and Peripheral Modules**

A system module supports the overall operation of the CPU. For example, two essential types of system modules are called the clock module and the memory module. The clock provides a timing signal which the CPU uses for cycling through its fetch and execution cycles, and the memory module stores the program.

A peripheral module acts as a data interface in between the microprocessor and a device which the microcontroller is monitoring or controlling. In other words, it handles voltage signals coming from or going out to some type of device. The signals are called microprocessor input and output signals.

For data flowing into the microcontroller, the peripheral module converts input signals into digital data and then makes it available to the microprocessor by placing it in a specific location in memory. For data flowing out of the microcontroller, the module gets digital data from some specific location in memory, and then converts it to some type of voltage output signal. The output signal is in the form or type which can be accepted and used by the device. The device is referred to as a peripheral device, and such a device maybe in the form of a switch, relay, sensor, actuator, display, application specific integrated circuit (ASIC), or another microcontroller.

---

### **The Historical Emergence of the Microcontroller**

During the middle 1960's, the ALU, control logic, and the peripheral modules were fabricated into separate silicon chips of integrated circuits to form a chip set. The individual chips had to be connected together with wires to form what was called back then a microcomputer. Although there were various technologies used for creating memory modules, memory was typically fabricated of toroidal magnets to form non-volatile program storage that was called magnetic core memory. Non-volatile means the program will not be lost after the computer is turned off. The Apollo space flight guidance computer was built of core memory.

Around 1970, the set of chips which formed the ALU and control logic were integrated into a single silicon chip that was called a one-chip microcomputer. This book will often refer to the ALU and control logic as the central processing unit (CPU) or

microprocessor. Memory modules and peripheral modules, however, were still externally connected components. And it was around this time when memory began to be available as practical integrated circuits built on silicon. Peripheral modules made of integrated circuits also grew in sophistication.

During the middle 1970's, the memory module and the general purpose input and output peripheral module began to be integrated into the one-chip microcomputer. And during the 1980's, this type of one-chip microcomputer began to be called a microcontroller.

---

### **Appearance of the MSP430 Microcontroller**

During 1992, Texas Instruments began to offer the MSP430, where MSP is an acronym for mixed signal processor. Mixed signal means that various types of input signals could be fed into it, and various types of output signals could be produced by it. That ability was created by integrating the microprocessor with a variety of specialized peripheral modules into a single integrated circuit. Since that time, Texas Instruments has continually improved the MSP430 microprocessor and also to continually develop different types of peripheral modules. Today, there are over four hundred different models of the MSP430, and the models branch out into families. Each model is distinguished by their unique set of system and peripheral modules.

---

### **Why use the MSP430?**

Whatever can be done by a microcontroller can also be done with circuits built of ordinary discrete devices such as resistors, transistors, capacitors, diodes, and wires. And like any other circuits, that approach takes time to design, build, and test. Such circuits also need a lot of room.

In contrast, the MSP430 contains a set of general purpose and application specific circuits, which are in the form of built in peripheral modules. The modules can be directed to behave in the ways we need through the control of a program. The ability to quickly make changes to the program also dramatically reduces product development time. So when using the MSP430, we can save a lot of time, effort, space, and cost. And have some fun during the process.

---

### **Purpose of this Book**

The MSP430 is quite literally a remarkable product. It is a computing platform that provides a whole lot of complex data processing capabilities which are abstracted out to a single and far less complex abstract layer in the form of a computer program. We use the program to quickly tailor the processing capabilities to fit our needs.

The modules, especially the peripheral modules, and the ability to program them are the two primary reasons why we choose to use the MSP430 for building products which can monitor and control devices. Therefore, in order to incorporate the MSP430 into a product, knowing about how its modules operate and how to program

them to do work are the most important types of knowledge and skills we must develop.

This book has three purposes. The first one is to explain how to write programming instructions which can read and write into the MSP430's registers. The second purpose is to explain the fundamental structure of a program. And the third purpose is to explain how to write programming instructions which prepare the MSP430 for work.

Later volumes will go into far greater detail about individual modules and how to program them.

---

### **Firmware is the Program of Instructions We Develop for a Microcontroller**

The computer program that we develop and load into the microcontroller is also called a firmware program.

Firmware should not be confused with software because of the way they are put into service and used is not the same. And since computer architecture is based upon different levels of abstraction, making a distinction between the two allows us to better explain and design computing systems.

The characteristics which distinguish the two are their installation and runtime behavior. A firmware program is permanently set into the nonvolatile storage fields of memory, and when needed, it is executed directly from those fields. A software program is semi-permanently stored in some type of nonvolatile memory, but when needed, it is loaded into volatile memory and executed from there. When the software program is exited or closed, it is removed from volatile memory.

Arguments can easily be made which dispute these characteristics. But the point here is to have concrete definitions so there will be no confusion about what type of program is being discussed and so that proper abstractions can be utilized for making explanations and descriptions. For example, such distinctions become important in scenarios where an MSP430 is communicating and interacting with software which is running on conventional personal computers, or local servers, or remote servers. Be aware that Texas Instruments documentation will typically refer to firmware as user software. This book will often refer to the firmware program as just simply the program.

---

### **Bits, Bytes, and the Native Word are the Basic Units of Data**

The binary, decimal, and hexadecimal are positional numbering systems which are used for developing programs. They use a base set of symbols, called numerals or digits, which are combined into a sequence of positions to express a single number of numerical value. Binary numbers are expressed with only two different digits, while decimals use ten different digits, and hexadecimal numbers use sixteen. All use an infinite number of positions, or place values, to express a single number.

A binary bit is a single position or place value that represents a single digit. A binary digit can be expressed as either a 0 or 1. The field of computing organizes bits into sets called nibbles, bytes, and words which have a standard quantity of bits.

A nibble is made of four bits, and a byte is made of eight bits. As for the size of a word, the size depends on the microcontroller's central processing unit (CPU), since it works on words of a fixed length, and that length varies from one type of CPU to another. It's called the native word size. Historically, CPU design has progressed through being able to handle four, eight, sixteen, thirty two, and now sixty four bit words.

As will be elaborated upon later, there are two types of words which an MSP430 CPU processes. One type is meant to express the number to an address in memory, and the other type is meant to express the data at the address. When developing a program, we will not be interested in the length of an address number. But we will be interested in the length of data at an address. A single address in MSP430 memory can only store a byte, but the CPU can simultaneously process two bytes at a time. This means that the native word for the MSP430 is sixteen bits wide, or in other words, it is a sixteen bit CPU.

---

### **How the Microcontroller Views Data**

This really depends on where the data appears. When the data is located on the outside of the microcontroller, and it is flowing into or out of a peripheral module, the data will appear as analog or digital voltage signals. The type of signal which actually appears will depend on the peripheral device which is being monitored or controlled. The voltage will typically range from ■■■ to ■■■ VDC. And we must take into account that the actual voltage signal produced by the peripheral device may have to be properly conditioned with circuits which act as the interface between the microcontroller and the peripheral device.

The MSP430 does not accept nor produce alternating current (AC) voltage signals. And AC signals must not be confused with analog signals, since AC is typically not used for transporting information. However, for an MSP430 to accept an AC signal, we have to design an interfacing circuit which will convert it to one that fluctuates between zero and 3.6 VDC. And for an MSP430 to produce an AC signal, we have to design a circuit that will convert a direct current voltage signal that fluctuates from ■■■ to ■■■ volts to the appropriate AC voltage signal. But keep in mind that the MSP430 is not intended to be used in that way.

Since the peripheral module acts as a signal interface in between the microprocessor and the peripheral device, it is the point where data under goes conversion. All data which flows from the outside and into the peripheral module is converted by the module to digital data, and vice versa.

Within the microcontroller, the MSP430 microprocessor perceives and handles all data as either bytes or words. However, when developing firmware, our code may

handle data which is a bit, byte, or word in width. And using firmware to manipulate just a single bit in memory is not uncommon.

---

### **The MSP430 is Offered with Two Types of Processors: CPU and the CPUX**

The MSP430 is offered with two types of CPUs and two sizes of memory. The first type of CPU is just simply called the CPU, and it uses the smaller memory module. The second type of CPU is called the CPUX, and it uses the larger memory module.

The smaller module has [REDACTED] addresses of storage, while the larger module has [REDACTED] addresses of storage. Both modules provide the same amount of storage at each address in memory; meaning, each address can store only eight bits. The single characteristic which distinguishes the two CPUs is their ability to handle address numbers. The CPUX can handle a larger set of addresses. Other than that, there is no difference between the two CPUs. Also, just because a particular type of CPU may be able to access [REDACTED] or [REDACTED] addresses of storage, that characteristic does not mean the memory module has that amount of space. The memory module comes in different sizes, depending on the model of MSP430.

---

### **How We [as Developers] may View and Express Data**

We will be writing code in the C programming language. So from that point of view, meaning, as we view our firmware code while developing it, the data can appear in any valid notation. It can be written as decimal numbers, binary numbers, hexadecimal numbers, or as alphanumeric characters; whichever notation is most convenient and appropriate from our point of view. When loading the program into the microcontroller, the MSP430 C Language Compiler will convert the entire program to machine code, which is expressed in binary form.

Decimal numbers can be used in our firmware code without any special notation. A binary number must be denoted with the prefix [REDACTED] for example, [REDACTED]. A hexadecimal number must be denoted with the prefix [REDACTED] for example, [REDACTED] and [REDACTED]. However, data sheets and users guides will often denote a hexadecimal number with the suffix [REDACTED] for example [REDACTED] and [REDACTED].

When stepping through our code, while our code development tool is in debugging mode, we can configure the tool to present the firmware data, such as numerical variables and constants, to us in binary, decimal, or hexadecimal notation. The instructions to do that are described by a later chapter.

---

### **Registers are Used for Configuring and Controlling the Microcontroller**

The word register has multiple documented origins, which go back to about 1259 of the current epoch, and which are partly borrowed from French (registre) and partly borrowed from Latin (registrum). The undocumented usage most certainly goes back further in time. Its concise meaning varies with context, but it basically means that it

is documentation used for recording facts. In the context of an MSP430, here is what it means.

A microcontroller register is a sort of fact storing electronic circuit. It can be defined as having physical characteristics and logical uses. It is physically a set of storage fields where each field can store only a single bit. The storage field is called a bit-field. The bit in a bitfield can be read and manipulated by our firmware.

As for its uses, a register has two logical uses. The first use is for putting data into a module, and the second use is for getting data from a module. In other words, from the microprocessor's point of view and our firmware's view, they are data inputs and outputs for modules.

Every module has its own dedicated set of registers. To configure, control, and monitor a module, our firmware must read and write data into their registers. Data which is read from the register is used for making a decision. Data which is written into a register is used for configuring or driving a module. This is the most fundamental concept in programming the MSP430.

The MSP430 has two locations for registers. One is inside of the CPU, and the other is inside of the main memory module.

---

### **Main Memory Registers**

As will be elaborated upon later, locations in main memory are denoted by address numbers. For the MSP430, the set of address numbers will range from [redacted] to either [redacted] or [redacted] depending on the CPU type. That set of addresses is called an [redacted], and keep in mind that each address can store a single byte of data, and no more.

And up to now, main memory has been referred to as a single memory module, but in reality, it is two modules. One module is made of non-volatile storage technology called read only memory (ROM), and the other is made of volatile storage technology called random access memory (RAM). The latest models of MSP430 are built of ferro-electric RAM (FRAM). When power is removed from the microcontroller, data in the volatile sections of memory is lost, but data is not lost from the nonvolatile sections. The program is stored in the nonvolatile sections of main memory.

Registers which are located in main memory are built of RAM, so when in service, they can be changed by our firmware and the CPU. The quantity of main memory registers is partly dependent on the quantity of modules in the microcontroller, which varies from one model of MSP430 to another. The amount of RAM is also dependent on pricing, meaning that some microcontrollers have more RAM than others in order to accommodate larger amounts of volatile data which our program may create and use.

Registers are typically distributed across the lower addresses in memory. The size of a main memory register will be either [redacted] or [redacted] bits wide. Do not get the size of a register confused with the fixed size of storage at an address; they are not the same

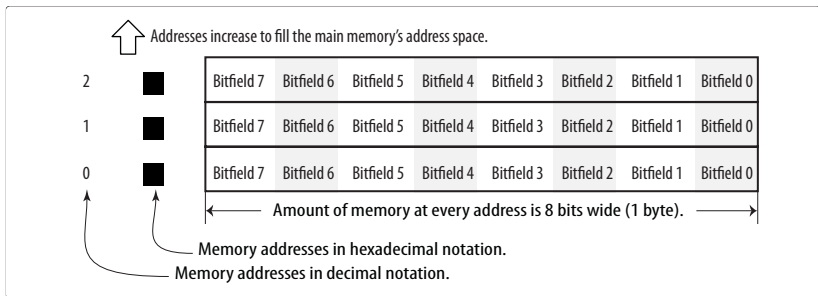


thing. Register size is a logical expression of storage space, while address storage size is a physical expression of storage space. This definition will become clearer when you start learning how to use registers.

Many registers are 8 bits wide; therefore, they have a single address. But if the register is 16 bits wide, it's comprised of two adjacent addresses in memory. So technically, those wider registers have a lower and higher address, but from a programming point of view, only the lower address is needed for reading and writing into those 16-bit registers. Those technical details will be abstracted away with programming symbols called register variables, as will be explained later.

Since registers are located in the volatile sections of main memory, when the microcontroller starts or restarts, our program will typically have to configure those registers to some extent before the modules can be put into service. To some extent means that most registers will automatically be loaded with default data which may not need to be changed and represents an acceptable or preferred configuration for operating the module. The reset system handles the loading of that data, and it is called system initialization data.

**Diagram 1:** This is the structure of individual storage places in main memory. The first three addresses in memory are shown here. Every address in main memory can store one byte of bits. The lower addresses in main memory are where the registers are located.



## What is a Main Memory Register used for?

Registers located in main memory are used for setting up, configuring, monitoring, and operating various systems and modules. They provide the means for our program to get and put data into modules. Most registers are completely dedicated to an individual module, while a few are dedicated to handling two or more types of modules.

## CPU Registers

Registers located inside of the CPU are called CPU Registers, and they are sometimes referred to as machine registers. The CPU is built with many registers which have specific purposes, but in the MSP430, only sixteen of them can be accessed with our firmware. And out of those sixteen, we will be concerned with only one of them. It's called the status register.

Fourteen of the CPU registers are built of volatile type storage technology. The remaining two nonvolatile registers are called constant generator registers. The data

in them do not change. They contain numbers, such as `0x0000` and `0x0001` which are commonly used in firmware instructions. This mitigates the need for the CPU to fetch them from main memory where our firmware is located. If using the C programming language, instead of the Assembly language, the MSP430 compiler sees these numbers in our firmware, and it will automatically edit our code to use the data supplied by the constant generator register.

The status register contains data about the microcontroller's operating state. From a programming perspective, we are mostly concerned with the status register bitfields which control the clock module and which can block CPU interruption signals, also called interrupt requests (IRQ). The clock is used for producing accurate square wave voltage signals which are used by all systems and modules so they can run and be synchronized with each other. As for the interrupt signals, they ask the CPU to stop what it's doing and execute a specific instruction in our program called an interrupt service routine (ISR). Requests from some modules can be blocked by manipulating a specific bit in the status register.

And finally, except for the constant generator register and status register, the CPU registers are designed to accommodate the width of any address number in the memory module. Since these registers handle the addresses to data, and not the actual data itself. For an MSP430 which is built of the conventional MSP430 CPU, it must be able to access an address space having address numbers ranging from `0x0000` to `0xFFFF` bits in width, while CPUX be able to access a space which reaches up to `0x0000` bit address numbers.

Other types of CPU registers, which are of no immediate concern to us as beginners, will handle the actual data and manipulate it, based upon our program's instructions. After we become more experienced with developing firmware, those unoccupied CPU registers can be utilized for storing data which can be accessed very quickly.

---

## Summary

Data can be understood as electrical signals and binary numbers. When entering and leaving the microcontroller, they are in the form of analog and digital electrical signals. When they are inside of the microcontroller, they are in the form of binary numbers so the CPU can process them.

The MSP430 is built of components called system modules and peripheral modules. The system modules are used for carrying out the program instructions and supporting the program in some way. For example, the CPU, memory, power management, and timer are system modules. Peripheral modules are interfaces in between the CPU and devices which the microcontroller is monitoring and controlling. Meaning, they are used for monitoring and controlling peripheral devices. A peripheral device can be in the form of, for example, a switch, sensor, actuator, display, or some application specific integrated circuit.

This book will refer to the program that we develop and load into the MSP430 as firmware or just the program. It will not be called software.

The native data word size for the MSP430 is [redacted] bits wide. The MSP430 may come with either a standard CPU or the CPUX. The standard CPU can handle a memory address space that ranges from [redacted] to [redacted] addresses, while the CPUX can handle an address space that ranges from [redacted] to [redacted]. Both CPUs have the same native data word size.

A register is a place of volatile storage. It can be [redacted] [redacted] or [redacted] two bits in width. There are two types of registers. The first type of register is called the main memory register. They are located in the [redacted] addresses of the memory module, and they act as the data interfaces to a modules. A register is the interface where our program can put data into a module or get data from a module. When putting data into a module, we are either configuring or driving the module. When getting data from the register, we are either monitoring the module or having it return some type of information which our program needs for some purpose. Although an individual address in main memory will physically have one [redacted] of space, a single register may be constructed of [redacted] [redacted] or even [redacted] adjacent addresses in memory.

The second type of register is called the CPU register, and it is located in the CPU. There are [redacted] CPU registers which our program can read and write into, but we are basically interested in one of them. It's called the status register, and it is used for controlling the microcontroller's operating mode and for blocking CPU interruption signals. Such interruption signals are used for triggering interrupt service routines (ISR). CPU interruptions and ISRs allow us to develop event-driven programs. The MSP430 is designed to be under the control of an event-driven program.

This book has three purposes. The first one is to explain how to write programming instructions which can read and write into the MSP430's registers. The second purpose is to explain the fundamental structure of an event-driven program. And the third purpose is to explain how to write programming instructions which prepare the MSP430 for work.

## Visualizing How the MSP430 Operates

Within the context of computing, a view is either a picture of a process, a structure, or both. It contains elements and their relationships which describe a process or structure.

Views can be produced within a range of abstractions. Meaning, views of greater abstraction show less detail, while views of less abstraction show greater detail. When learning about something new, we typically start with views which show high amounts of abstractions and then progress to views which show lower amounts of abstractions.

We are concerned about learning how to write a program which instructs the MSP430 to handle one or more event-driven processes. That is the firmware program's purpose and objective. Therefore, we are interested in viewing progressively different amounts of abstractions which show how it carries out that process. This chapter presents six different views which are arranged from high to lower amounts of abstractions. And most of the views are within an event-driven context.

That context is the fundamental concept which we must view the operation and application of the MSP430. It is designed and engineered to handle processes in that specific context.

The first view, called the basic view, is the most abstract picture of the microcontroller's operation. It shows a simple and easy to understand picture of its purpose and operation. It provides a starting point for visualizing how the MSP430 can be applied to create a product, but in a very abstract way. The second view, called the power-up view, goes into a little more detail about its operation. This is an important phase which every MSP430 goes through in order to prepare itself for work. The third view is a detailed view of how the MSP430 handles event-driven processes, the work which it is meant to do. There are actually two event-driven views. One handles internally occurring events, and the other handles externally occurring events.

The first three views are published nowhere else except by this book. They provide the visual foundation we need for understanding how a proper program basically instructs the MSP430 to handle event-driven processes. The remaining three views are published by Texas Instruments through their user guides and data sheets.

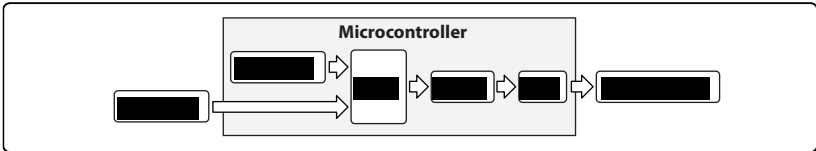
The fourth view is called a functional block diagram. It only appears in data sheets, and it shows the all elements which make up a specific model number of MSP430. The fifth view is called a pin designation, and it shows us which functions are available at every pin on the microcontroller's case. It is also a physical representation of the case. The sixth, and last view, is called the module functional view. It is a detailed picture of how an individual module is configured and operates. All module func-

tional views, except for one type, are published by the microcontroller's user guide. The remaining type of functional view is of the microcontroller's ports, and they are published by its data sheet.

## Basic View

When beginning to imagine or visualize the operation of an MSP430, this basic view provides a place to start. The operation is understood as being within an event-driven context.

**Diagram 2:** Basic view of how the MSP430 operates.



Events are categorized as external and internal. External events are in the form of

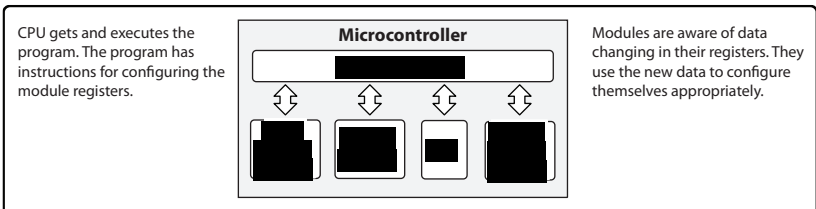
During power-up, the firmware program had configured the microcontroller to sense for some type of event. When the event is sensed,

The purpose of the work it to produce some specific type of output voltage signal. The signal communicates information to a peripheral device or drives it in some way.

## Power-up View

Power-up should be understood as a very short period of time or operating phase where the microcontroller is preparing itself for work. It does not accept any input signals, nor does it produce any output signals during this time.

**Diagram 3:** Power-up view of how the MSP430 operates.



Four microcontroller elements are involved during this phase: . Some peripheral modules will act as input modules, while others will act as output modules.

No [redacted] elements are involved, except for power. The purpose of the [redacted] module is to be a location where the [redacted] are stored. [redacted]

The power-up sequence begins when [redacted]  
[redacted]  
[redacted] This phase is called power ramp-up, and it typically takes about ten microseconds.

Once all systems are properly energized, the CPU control logic loads the first instruction of the program into the CPU and then puts the microcontroller into the active operating mode. The CPU can then begin to execute the instructions in the program.

During power-up, the program has a single objective: [redacted]  
[redacted] The modules are always aware of data changing in their own memory registers, so when a change occurs, they read and use the register data to configure themselves.

The last instruction in the program puts the microcontroller into some low powered operating mode called sleep. Only specific events, which the modules are configured to sense and monitor, will interrupt the microcontroller from its low powered state.

---

## Event-Driven Views

After the power-up scenario has completed, the microcontroller is typically in some low powered operating mode. It is characterized as some level of sleep when it is monitoring for events which it was configured to handle. When an event occurs, the microcontroller enters the active operating mode to handle the event. And when finished, it returns to the same operating mode from where it was awakened. An event-driven view shows the MSP430's operation from the point in time when an event occurs to the point when the result of the event produces an output signal. This is the type of operation which the MSP430 was designed to carry out.

There are two different event-driven scenarios which we are concerned with. The first scenario involves an event which occurs from inside of the microcontroller, and the second scenario involves an event which occurs from outside of the microcontroller. Since it provides a framework from which the externally produced event is built upon, we'll begin with a view of an internally produced event.

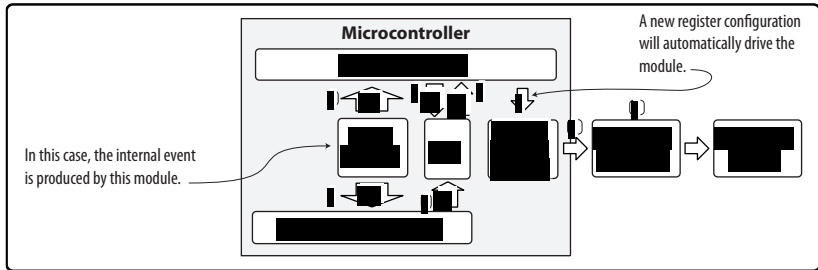
---

### View of an Internally Occurring Event

Internally occurring events are produced by modules which are located inside of the microcontroller. This view, of how an internally occurring event is handled, involves two modules and a peripheral device. One module is in the form of a [redacted] and the other is in the form of a [redacted] module. And the peripheral device is in the form of an [redacted]

An overview of the process goes like this. [REDACTED]

**Diagram 4:** View of how an internally occurring event is handled.



After the power-up, the program had configured the timer to count through a range of [REDACTED], called an interval, so when it reaches the end of the interval, a timer overflow occurs. An overflow means the timer has reached the end of the interval. Furthermore, the timer had also been configured to set a CPU interrupt flag (IFG) when that event occurs. An IFG is just simply a single bitfield in a particular timer register which triggers an interrupt request (IRQ) signal to be sent the CPU interrupt system.

Be aware that the various low powered operating modes are created by progressively removing the clock signal, and possibly power, from an increasing number of modules. The particular mode in this scenario (the exact one is irrelevant for now) will be able to provide a clock signal and power to the timer module.

### Steps 1 and 2

The process begins at step 1 when the timer overflows. That is our internally produced event. An overflow automatically [REDACTED]

### CPU Interrupt System Behavior

All IRQs have a priority level which is assigned by their [REDACTED] in a dedicated section of memory called the [REDACTED]. The various levels are published by the microcontroller's data sheet in a section which is typically named "[REDACTED]" The [REDACTED] as a list of ISR vectors. A vector is a cross-reference between [REDACTED]

The CPU interrupt system also acts as an IRQ filter. Meaning, some IRQs may be blocked from interrupting the CPU while others are not. The filter is activated by using a firmware instruction to set a bit in the CPU status register. That bit is typically named the general interrupt enable (GIE) bit. The set of requests which can be blocked are called maskable CPU interruptions, and such requests are produced by peripheral modules. The set of requests which cannot be blocked are called non-maskable interruptions (NMI), and such requests are produced by system modules. Non-maskable interrupts can be further categorized as a user NMI (UNMI) or as a system NMI (SNMI). The typical UNMI is caused by an input signal coming from outside of the microcontroller; for example, from a button used for restarting the microcontroller. An SNMI is caused by a system module located inside of the microcontroller; for example, a low voltage at the power supply pin will cause an SNMI.

A timer overflow flag will typically produce a maskable interrupt request.

---

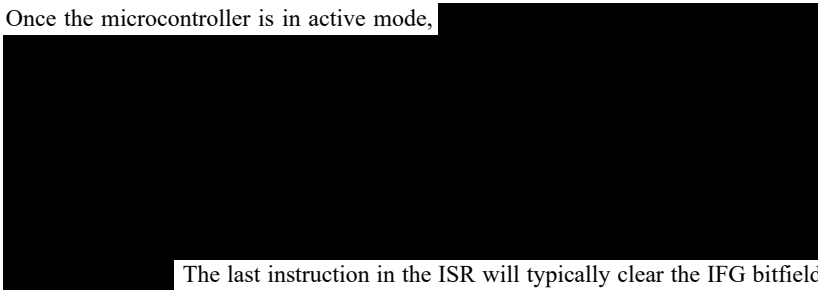
### Step 3

Now back to the view of how an internally occurring event is handled, as shown on page 14. Once the CPU interrupt system finds the vector (VTR) for the request (IRQ), it loads the ISR's memory address into the CPU (3), and then the interrupt system puts the microcontroller into active mode.

---

### Steps 4, 5, 6, and 7

Once the microcontroller is in active mode,



The last instruction in the ISR will typically clear the IFG bitfield to 0, and once that is done, the microcontroller is automatically put back to sleep.


---

### C, Assembly, and the Final ISR Instruction

If our firmware is written in the C programming language, the MSP430 compiler will automatically add a final ISR instruction that puts the microcontroller back into the operating mode from which it was in. If our firmware is written in the Assembly programming language, we would have had to write that last instruction. This book will be using the C language.

---

### Step 8

At step 8, the output signal is conditioned into a form which will properly drive the  peripheral device. This conditioning is carried out by a circuit, which is located

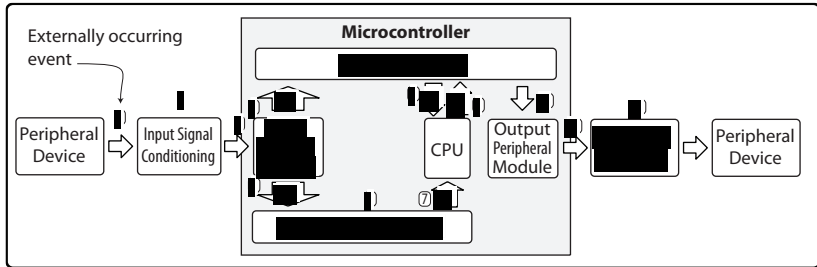


outside of the microcontroller, and which provides the interface between the microcontroller and the peripheral device.

### View of an Externally Occurring Event

Externally occurring events are produced by peripheral devices which are connected to the outside of the microcontroller.

**Diagram 5:** View of how an externally occurring event is handled.



This view, of how an externally occurring event is handled, contains two peripheral devices and a microcontroller. In this case, one device is in the form of a button that produces an input voltage signal. That signal is our externally occurring event. The other device is an

While this view shows the input module and output module as separate modules, a single digital I/O module could be, and typically is, configured to handle both signals.

During power-up, the program had configured the input module to watch for signals on a specific pin and trigger a specifically written ISR to handle the event.

And finally, like the internally occurring event, this scenario also begins with the microcontroller in some low powered operating mode of sleep. But it is a mode where the digital I/O module is continually active, or it becomes active, when it senses an incoming signal.

### Steps 1 and 2

When the button is pressed, it sends a

---

### Steps 3, 4, and 5

When the input signal appears at the input pin (3), the module senses the [REDACTED]

Then at step 5, the module then produces an interrupt request signal (IRQ).

---

### Steps 6 and 7

When the CPU interrupt system receives the IRQ, it first determines th [REDACTED]

[REDACTED] and then it puts the microcontroller into active mode. The term VTR refers to the CPU interrupt vector.

---

### Steps 8, 9, 10, and 11

Once the microcontroller is in active mode, the CPU fetches the first instruction of the ISR from main memory (8), and then it begins to execute the ISR. Instructions in the ISR tell the CPU to [REDACTED]

[REDACTED] The last instruction in the ISR will typically clear the IFG bit-field to 0, and once that is done, the microcontroller is automatically put back to sleep.

Since our firmware is written in the C programming language, the MSP430 compiler will automatically add that final ISR instruction which puts the microcontroller back into the operating mode from which it was in. Meaning, the specific mode it was in before the event occurred.

---

### Step 12

At step 12, the output signal is conditioned into a form which will properly drive the [REDACTED] peripheral device. This conditioning is carried out by a circuit, which is located outside of the microcontroller, and which provides the interface between the microcontroller and the peripheral device.

---

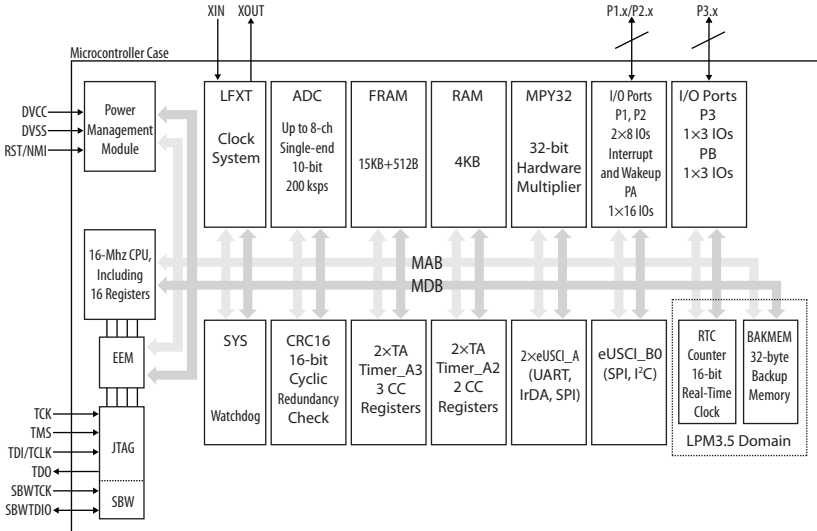
### Functional Block Diagram View

This and the following two views are what Texas Instruments will typically publish in their user guides and data sheets. A functional block diagram will appear in the microcontroller's data sheet. Its purpose is to show us which modules are built into the microcontroller, what are their basic capabilities, which modules are connected to the memory bus, the basic set of pins built into the case, and which modules the pins are connected with. It does not show how these modules operate and work together.

In this case, the diagram is about an MSP430FR2433, and it represents what is typically published as a functional block diagram. The only change to this type of view will be more or less modules, since that depends on the microcontroller model. However, this particular diagram does show the basic set of modules which we'll see across most models of MSP430 microcontrollers.

The user guide for a family of MSP430 microcontrollers will describe these modules in further detail, along with other modules which a family member may include. And the data sheet will go into specifics. Both documents can be found on the microcontroller's home page at [ti.com](http://ti.com).

**Diagram 6:** The functional block diagram as would be published by the microcontroller's data sheet.



## Memory Modules

The first matter to be aware of is the two memory modules which are named FRAM and RAM. They represent main memory, and they have often been referred to as a single module for two reasons. The first reason is to simplify the discussion, and the second reason is because a single memory address space actually spans across both of these modules. The lower addresses are built of volatile random access memory (RAM), while the higher addresses are built of non-volatile ferro-electric RAM (FRAM). Registers are located in RAM, while our program is located in FRAM. Data sheets will often refer to RAM as SRAM (static random access memory).

We learned earlier that the address space for the MSP430 CPU ranges from 0 to [redacted] while the CPUX goes to [redacted] and that each address will store a [redacted] of data. Every MSP430 uses one or the other of these address spaces, depending on which CPU is built into it. However, the amount of actual usable memory in an MSP430 will vary from model to model.

This means that for microcontrollers which are built with less than [REDACTED] kilobytes of memory, not every address in the space is used. Those unused addresses spaces are typically placed somewhere in the middle of the address space in order to maintain program code at the top of the space and registers at the bottom of the space.

The function block diagram for this MSP430FR2433 shows that RAM is 4KB (kilobytes) in size, in other words, it occupies four thousand addresses which are each a byte in size. FRAM occupies 15KB with an additional 512B (bytes). And this too can be expressed as fifteen thousand addresses which are a single byte in width, along with an additional five hundred and twelve addresses of the same width. To learn what that additional address space is used for, we refer to the data sheet. In this case, the section named Memory Organization, states that it's used for information memory, which means, that space stores factory information about the microcontroller. For example, calibration data that can be used for adjusting the accuracy of the real time clock (RTC) module.

---

## Memory Buses

Two memory buses interconnect all the modules, except for the JTAG and SBW modules.

The memory address bus (MAB) transports main memory addresses, and it is either sixteen or twenty bits wide, depending on the size of the overall address space.

The memory data bus (MDB) transports data to and from the addresses in memory. Although each address only stores a byte, the data bus is sixteen bits wide, so it can transport two bytes. That means a byte from two adjacent addresses. The data bus width accommodates the native word size of the CPU, which is sixteen bits.

Keep in mind that data basically flows along the buses like this: [REDACTED]

[REDACTED] and the modules write into their registers when they have to update them.

---

## Power Management Module

At the upper left corner of Diagram 6 is the power management module (PMM). It is responsible for [REDACTED] during operation. Inside of this mode are subsystems which help the module carry out its work. They are referred to as various types of supply voltage supervisors.

This PMM has two voltage supply connections coming from outside of the microcontroller. As was explained earlier, DVCC is a pin that provides a [positive] digital power supply, while DVSS is a pin that provides digital ground. However, sometimes a functional block diagram will show an AVSS and AVCC connection. They mean [positive] analog power supply and analog ground respectively.

The digital supply connections are designed to provide power to modules which process digital data, while analog supply connections are designed to provide power to modules which process analog data. This is based upon the rationale that analog modules will consume more current than digital modules. For example, a particular model of MSP430 may contain analog modules which consume more current than the digital modules. A digital to analog converter (DAC) is an example of such an analog module.

You may be led to think that the analog and digital terminals must use different sources of energy, but Texas Instruments will typically recommend that the same source be used for both connections. This is to allow us to simultaneously measure the current consumed by those two types of circuits. The data sheet will provide such information by searching for AVCC.

Also going into the PMM are the RST or NMI signals which typically use the same pin. These are binary voltage signals which are typically coming from a peripheral device such as a button or something else. The purpose of this signal is to produce a user non-maskable interrupt (UNMI) signal. And the pin which monitors for these signals must be configured to distinguish between the two. When the pin is configured to RST (reset) mode, the signal will force the microcontroller to restart. When the pin is configured to NMI mode, the signal will trigger an interrupt service routine.

---

### **CPU, EEM, JTAG, and SBW Modules**

A block that represents the CPU will typically show the CPU's maximum clock frequency (or operating speed), and the type of CPU. The clock frequency is produced and handled by the clock module, which can be set by firmware, but the frequency is ultimately dependent on the supply voltage. Lower voltages will produce lower frequencies. The CPU will be indicated as either being the standard CPU or the CPUX. Its job is to fetch programming instructions from main memory, execute the instructions, and write the results of those instructions into main memory registers.

The Spy-Bi-Wire (SBW), Joint Test Action Group (JTAG), and embedded emulation module (EEM) blocks are to be viewed as having a combined purpose. They are used for

while our firmware development tool is in debugging mode. The SBW block is a data interface that translates a Texas Instruments proprietary bi-direct communications protocol into a JTAG protocol. The JTAG block provides the ability to load and erase our program. The EEM block provides the ability to test our program. Every MSP430 has these blocks built into them. After the microcontroller has completely gone through the development process and put into service, these blocks no longer needed. But they can provide another feature while the microcontroller is in regular service.

Some families of MSP430 microcontrollers have the ability to use these blocks to protect the microcontroller from unauthorized access. Meaning, we can lock the microcontroller with an electronic fuse to prevent any access which is attempting to read the firmware, copy it, or takeover the microcontroller. An instruction in our program is used for blowing the fuse. So when the fuse is blown, access to the MSP430

through the JTAG interface is permanently disabled. These blocks can also be used for remotely updating the microcontroller after it has been put into service.

Furthermore, and what is not shown by the functional block diagram, is a programmer-debugger circuit which is needed as an interface between the SBW-JTAG blocks and our personal computer where our program is developed. The circuit is just a final external data interface between the MSP430 and our computer. When you purchase the MSP430 as part of a kit called a LaunchPad, a programmer-debugger is built into it. It enables us to communicate with the MSP430. When the microcontroller is put into service, this circuit is not needed.

---

## **Clock System Module**

Every microcontroller has a clock for producing a binary voltage signal that drives and synchronizes all activity in the microcontroller. They often go by a specialized name, but this one is just simply named the clock system.

Most, if not all clock modules, are connected with pins on the case to connect an external oscillating device for driving the clock. Although the clock module is able to produce its own clock signal, it may not be accurate enough for some applications. So those pins provide access for a more accurate oscillator, such as a standard quartz crystal which is built into many watches.

This particular microcontroller, shown by Diagram 6, has a real-time clock (RTC) module for keeping time which is based upon a calendar, so a watch crystal may be used for improving its accuracy.

The names for those external clock connections are shown as XIN and XOUT, and they interface with the LFXT port of the clock, which stands for low frequency external. The pins are actually labeled with those names.

Watch crystals are typically referred to as low frequency oscillators, and they oscillate at 32,768 Hertz.

---

## **ADC Module**

Next to the clock module, the functional diagram shows the analog to digital converter (ADC) peripheral module. Its job is to measure an analog voltage signal and then convert the measurement into a binary number. It's basically a sophisticated voltmeter that quickly converts a measurement into a binary number and writes it to a specific register in memory. The voltage signals are produced by some peripheral device, such as a thermometer or battery. The firmware program will typically use the individual measurements to make decisions.

This ADC block is shown as having 8-ch, Single-end, 10-bit, and 200 ksps. Here is what those specifications mean.

A circuit which consists of a single path from a pin on the case to the ADC module, and where a voltage can be placed into and be measured by the ADC, is called a voltage signal input channel. This ADC has eight channels. An ADC which compares the

voltage on a channel to ground is called a single-ended ADC, while an ADC which compares the voltage on one channel to another channel is called a differential ADC. This ADC is single-ended.


Although the resolution for an ADC can be expressed in several different ways, this one is expressed in bits. It is a 10-bit voltage resolution ADC. To understand what that means, we must visualize the ADC's voltage measurement scale from a particular point of view. We begin to develop that view by first determining the largest decimal number that can be expressed through ten bits. That number is 1,023. Next we visualize the range of voltages where the signal must be conditioned to swing. Conditioned means to use a circuit that adjusts the actual voltage signal magnitude to within an amount which the MSP430 can accept. For an MSP430, the range will typically be from zero to 2.5 volts, but it can be as high as the supply voltage. Next we divide the measurement range into 1,023 equal parts, which then makes each division 0.0024 volt in width. So in other words, the voltage resolution is 2.4 millivolts, and that is what ten bits of resolution means.


The last specification shown is the number of kilo samples per second (ksp/s). This is obviously a rate, and it means the maximum number of analog to digital conversions which can be executed during one second. For this ADC, it is 200 ksp/s. But do not let the unit of samples per second (sp/s) mislead you. It actually means the entire process of taking a sample of the voltage signal, measuring it, then converting it to a binary number, and then writing it into a register.

One last interesting characteristic about the image of this ADC block is the absence of the channel inputs. They are implied.

---

### **MPY32 Module**

The MPY32 system module is used for 


Here's how it basically works. 

---

### **I/O Ports**

Two I/O (input/output) port blocks are shown by the functional diagram. Ports are numbered in order to distinguish one from the other. And within the microcontroller industry, this type of port is often referred to as a general purpose input or output (GPIO) port. Every MSP430 has at least one port.

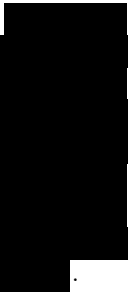
A port has two purposes.



A port will typically have either four or eight channels which interconnect a specific module with pins on the case of the microcontroller. But some models of MSP430 can have a port with odd number of channels, such as three. Input and output signals flow across those channels.

Inside of the port is a logic circuit which is called a multiplexer. It is situated in between the port channels and modules, and it is under the control of our program. For input signals, it connects a channel with a particular module. For output signals, it connects a particular module with a channel. The switching process is called signal multiplexing.

Now let's take a closer look at what the I/O port specifications mean,



Below the block names (I/O Ports) are the labels P1, P2, and P3. That tells us which port numbers are built into the block. Combining two ports into a single block is only a logical representation of the port's function, not a physical representation. Ports 1 and 2 are logically combined to show that they can be programmed as a single port called PA, which means Port A. And notice that the other block is labeled PB, which means Port B. This is related to a programming concept, which will be explained later, that uses a programming symbol called a register variable. Register variables give our program direct access to the contents of a register so data can be written into it and read from it. Therefore, the register variable named as PA will give us simultaneous access to ports 1 and 2. Notice that block PA includes the notation 1x16 IOs. That means a single bus which is sixteen channels wide. It obviously takes into account the buses for P1 and P2. However, each port does have a unique register variable for us to use, and that is how we'll typically read and write into a register.

Ports can be configured to monitor for voltage signal events which trigger interrupt service routines (ISR). The block for P1 and P2 shows these ports as being able to interrupt and wake-up. The word interrupt means that ports 1 and 2 can be configured to monitor for events which will trigger an ISR. The word wake-up means that the same events can wake the microcontroller from a low powered operating mode to trigger an ISR.



---

## Digital I/O Module

A single port channel is not just simply a single channel. It is actually made of two sub-channels, and both sub-channels are connected to the same terminal pin on the case of the microcontroller. One sub-channel is used for handling input signals, and the other is used for output signals. They cannot be used at the same time, so we have to switch from one to the other as needed.

Control over those sub-channels is handled by a block of logic called a Digital I/O Module. So an eight channel port has eight of those modules. They're not shown by the functional block diagram, but are located inside the port blocks.

Be aware that while one side of those sub-channels are connected to a single pin, the other sides are connected to a switch called a [REDACTED] and a single Schmitt Trigger. The multiplexor is used for connecting a sub-channel to a specific system or peripheral module. Analog signals typical go through the [REDACTED]. The trigger is used for sensing digitally high or low voltage signals, so its used for sensing digital input signals. Both sub-channels can be configured to set an interrupt flag when a specific signal occurs, but this feature is typically limited to the first two ports.

When speaking about those sub-channels, we always refer to them as either the input or output channel for a digital I/O module, or just simply as the channel. A diagram of such a module is shown by diagram 8 on page 33.

---

## SYS Module

This block represents a module called System Control. It basically represents a set of different system modules.

The work which this block carries out is done by all MSP430's. It is responsible for handling the [REDACTED]

[REDACTED] Their operation will be elaborated upon by later chapters.

Be aware that for many models of MSP430, these individual capabilities are built into their own separate modules. To configure and operate these modules, you write data into their registers. And to get data from them, you read their registers.

---


## CRC16 Module

The Cyclic Redundancy Check (CRC) system module is used fo [REDACTED]

The CRC module can also be used for [REDACTED]

Checking the integrity of data is a common task which is performed by data transmission equipment. It is needed because the messages could be corrupted when passing through strong electromagnetic fields or faulty connections. Those are common scenarios which occur in many different types of usage environments.

Here's how it basically works.



When the time comes to send a message, our program instructs the CPU to initialize the CRC module by writing a word into the initialization and result register. The word can be any combination of bits, but a word that is completely made of 1 digits is typically used. The program then tells the CPU to start writing the data into the data in register one byte at a time. Every time a byte is written into the register, a new word of data appears in the initialization and result register. That word represents the CRC checksum for all the data which has been entered up to that time. When the last byte of data has been entered into the data in register, the program instructs the CPU to form the data into a message by appending the checksum to the end of the data. The program then tells the CPU to use some communications module to deliver the message. When the peripheral device receives the message, it enters the data into its own CRC module, or firmware algorithm, to produce its own checksum. The device also uses the same word to initialize its own CRC module. The delivered checksum is then compared with the produced checksum, and if they match, the message is uncorrupted and ready for use.

The functional diagram shows the CRC module as being a 16-bit module. That means the checksums it will produce are sixteen bits wide. Some models of MSP430 come with a 32-bit module or both.

---

## Timer Modules

A brief explanation about the timer module is needed before explaining how it is represented by the functional diagram.

A timer is a system module that provides four different services. It serves as a frequency dependent timer. It serves as a frequency independent counter. It serves as a module for measuring voltage signaling rates. And it serves as a module for producing pulse width modulated (PWM) voltage signals.

A timer module has several registers which are used for configuring and operating it, but for this brief explanation, we are only interested in two of them. One register is used for counting through an interval (range) of numbers, and it is called the timer [counter] register. The other register is used either for capturing the count number

associated with an event or for making comparisons between its own contents and that of the timer register; this register is called the capture-compare register. How this register is used will depend on whether the timer is configured into the capture operating mode or the compare operating mode.

An important feature of the timer is its ability to produce CPU interrupt flags (IFG). It can set flags for timer overflows and external events which it is monitoring. A flag is used for triggering an interrupt service routine (ISR).

---

### **Serving as a Frequency Dependent Timer**

When the timer serves as a frequency dependent timer, we use it for

s.

Here is how it works. The timer is first configured into the

---

### **Serving as a Frequency Independent Counter**

When the timer serves as a frequency independent counter, we use it for

s.


Here is how it works.

---

### **Serving as a Module for Measuring Rates**

When the timer serves as a module for measuring rates, we use it for

Here is how it works.

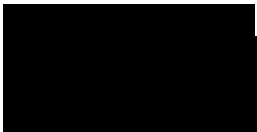
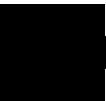


Now be aware that event signals can also be produced from inside of the microcontroller by our firmware. Therefore, firmware produced signals can be used for measuring the length of time which a set of programming instructions (called a process) had taken to execute.

---


### **Serving as a Module for Producing PWM Voltage Signals**

A pulse width modulated (PWM) voltage signal is a digital [square] wave which is characterized by its frequency and duty cycle. It is not an analog [sinusoidal] signal.

When the timer serves a module for producing PWM voltage signals, we use it for . For example, it can be used for controlling th 

Until now, each service which the timer provides had only needed to use a single timer register and a single capture-compare register. In this scenario, we'll need two capture-compare registers.

Here is how it basically works.



While the timer is counting through each period, it uses the two capture-compare registers to determine at which point in the period to produce a digital 1 signal and at which point to produce a digital 0 signal. For example, to produce a square wave having a 50% duty cycle over a 1,000 count period, the first capture-compare register is configured to 500 counts, and the second register is configured to 1,000 counts. So when the timer begins at zero, it produces a digital 0 voltage signal at its output until

it reaches the count of 499. When the timer counter reaches 500, that number equals to the number in the first capture-compare register, so the timer produces a digital 1 output voltage signal. The output signal remains at a digital 1 until the counter reaches 1,000, a number which equals the number in the second register and represents the end of the period. The process then repeats when the timer begins counting from the beginning of the period again.

---

### **How the Functional Diagram Describes the Timer Module**

A timer module is constructed of individual units of logic called blocks. There are two types of blocks. Every timer has one block, called a timer block, where the timer register is located. This register is used for counting through a range of numbers. The second type of block is called a capture-compare register (CCR) block. It is where a single capture-compare register is located, and along with inputs for sensing events and a single output for producing PWM signals. A timer module will typically have two or more CCR blocks.

The example functional block diagram, on page 18, shows two blocks which represent timer modules. These two blocks are not to be confused with the blocks of logic, just described, which make up an individual timer module.

Both blocks in the diagram contain three descriptions about their timers: the type of timer module and how many, their name, and the number of CCR blocks per timer. Let's look at the left block. The first description is 2xTA, and it means there are two Timer A modules. Texas Instruments produces at least three different timers (named A, B, and D), which vary slightly in capabilities. The second description is just simply Timer\_A3, which will lead to some confusion. There are two timers in this block, and they are distinguished as being named Timer0\_A3 and Timer1\_A3. So the word Timer\_A3 just refers to both timers. The third description is 3 CC Registers, and this means that each timer module has three CCR logic blocks per timer.

---

### **eUSCI Module**

An MSP430 will typically come with one or more modules which give it the ability to communicate with peripheral devices by using a specific type of communications protocol. For example, the universal asynchronous receiver-transmitter (UART) protocol, the inter-integrated circuit (I<sup>2</sup>C) protocol, and the serial peripheral interface (SPI) protocol. A protocol is a set of rules and procedures which specify how the communications module and peripheral device must transmit and receive messages.

A type of communications module which is currently built into the MSP430 is called the enhanced universal serial communication interface (eUSCI). It is constructed with a small set of different protocol capabilities, typically about two or three of them. Therefore, it has to be configured to use a specific protocol before going into service. Each configuration is called a protocol mode. And the module will only operate in one mode at a time.

The eUSCI module has many different registers which are used for configuring, monitoring, and operating it. For this short explanation, we are only interested in those registers which receive and transmit messages to a device. They are often called the transmit buffer and the receive buffer.

To basically use the eUSCI module, we first configure it into the protocol mode we need, which also includes that it be configured to set a CPU interruption flag (IFG) when a message is received. Then we write an interrupt service routine (ISR) that will respond to the flag. When it is time to transmit a message, our program writes the message into the transmit buffer register. When the module recognizes that the data in the buffer has changed, it automatically transmits the message. When the module recognizes that a message is being received, it automatically writes the message into the receive buffer and sets the IFG. The flag triggers the corresponding ISR, and the ISR contains instructions which tell the CPU to get the data by reading the receive buffer.

The example functional block diagram, on page 18, shows two blocks of eUSCI modules. They both state which type of communications module, the quantity, and which protocol modes are included. The block on the left says that it contains 2xeUSCI\_A. This means that the block represents two eUSCI modules, and both are type A modules. The eUSCI module comes in at least two types of modules, and they are distinguished by the protocol modes they provide. In the parenthesis are shown its protocol modes. IrDA was not mentioned earlier; it means the Infrared Data Association protocol. It uses infrared light as the physical medium for message transport.

---

### **LPM3.5 Domain**

The next two functional blocks are enclosed in a dashed lined box labeled as LPM3.5 Domain. That means that the blocks in this box will still be able to operate while the microcontroller is in low power operating mode 3.5.

The MSP430 is under the control of different operating modes. When all modules are energized, that configuration is called active operating mode (AM). To reduce the consumption of power, there are at least four low powered operating modes. They are typically referred to as 1, 2, 3, and 4. And each mode reduces the amount of power consumed by the microcontroller by removing the clock signal, and probably some actual power, which are supplied to individual modules. When the operating mode number increases, so does the quantity of modules which are deactivated. Our program can put the microcontroller into these modes by writing into specific bitfields of the CPU status register.

LPM3.5 is a special low powered operating mode which removes power from volatile memory (RAM). That means any data in RAM will be lost when this mode is entered. Modules which are not affected by this mode are enclosed by the LPM3.5 Domain as shown by the functional block diagram.

---

## RTC Counter Module

The purpose of the real-time clock (RTC) counter module is to incrementally count through a range of numbers which start at zero and end at some fixed number. We use it to calculate time, which is typically needed for displaying a time or for marking an event and its data with a time, also referred to as a time stamp.

Here is how it basically works. A clock signal is selected and fed into the module that will be used for a timing signal. The signal will drive the module to count through a selected range of numbers that will incrementally appear in a counter register. The accuracy of the counting frequency is dependent on the choice of clock signal. The frequency is then adjusted so that a specific number of counts will represent either a minute, a second, or a fraction of a second. Another register is used for adjusting the counting range to what is needed. When the counter reaches the end of the range, we call this event a counter overflow. An overflow forces the counter to immediately start counting from zero again. We may call that range a counting period.

To calculate a time, our program must read the counter register to get a count number. There are two basic scenarios which involve reading that register. Our program can just simply read the register when needed, or the module can be configured to set a CPU interrupt flag when an overflow occurs. The flag triggers an interrupt service routine which then reads the counter register.

The typical functional block diagram will show a box that represents a single RTC Counter module. The description is very clear. It will say that it's an RTC Counter, but one additional characteristic will be included. It describes the size of the counter register. In this case, the description says it a 16-Bit Real-Time Clock, which means that the counter register is sixteen bits wide. Therefore, the counting range is from zero to 65,535.

You may be wondering about the difference between the timer module and the RTC counter module, since they basically do the same work. The main difference is that the RTC counter can operate in low power operating mode 3.5 (LPM3.5), while the timer cannot, or at least, it typically does not operate in that mode.

---

## BAKMEM Module

Earlier, the LPM3.5 Domain was described. It was characterized as a low power operating mode where power is removed from volatile memory. Therefore, any data which is stored there will be lost when that mode is entered. That volatile data is in the form of storage variables and data storage structures (such as arrays). The backup memory (BAKMEM) module provides a place for storing that volatile data when the microcontroller enters LPM3.5 operation, and its availability is typically dependent on an active RTC Counter Module. It's typically not available without an active RTC.

This module is constructed of a set of nonvolatile registers. Each register has its own unique name which can be used by our program.

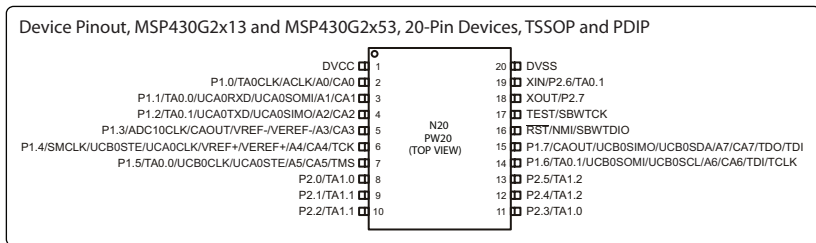
The strategy for backing up volatile data is not described by the microcontroller's user guide or data sheet, so it is left for us to design. However, the fundamental strategy is to use one or more instructions, which are appropriately placed in our program, which will copy data from storage variables and structures to their dedicated backup memory registers. And that work is done before entering LPM3.5.

Like the RTC Counter module, the functional block diagram which describes this module is also clear. It names the module as BAKMEM. And it also says that it contains 32-byte Backup Memory. That means the module is constructed of thirty two registers which are each a byte in width.

## Pin Designation View

Another and important view we use for learning about how the MSP430 works is through a pin designation view, which is also referred to as the device pinout view. Its purpose is to show us the physical shape of the microcontroller's case, the location of every pin, and the designated functions at each pin.

**Diagram 7:** The pin designation view of a microcontroller's case.



This view is published by at least two different types of documents. It can be found in the microcontroller's data sheet, and if it is built into a kit, like one of the LaunchPad kits, it will appear in the kit's user guide. Both documents can be found on the microcontroller's home page.

The example shown here is of the G2553. The point of view is from directly above the case, which is always used for pin designation views. Above the case is the title of the view, as it is typically printed by the data sheet. It states which models the diagram applies to, the number of pins, and the standard package types (TSSOP and PDIP) for this model.

At the center of the case is a description that says N20 and PW20. Those are Texas Instruments proprietary case type numbers (also referred to as package options). At the upper left corner is an orientation circle that indicates the location of pin number 1. Every pin is pictured, along with its number. Next to each pin is its function. Most pins on an MSP430 are able to provide more than one function, which is why we see so many functions at a single pin. Behind each of those pins is a multiplexer which our program may use for selecting a function.



Not shown here is the terminal functions table, which is found in the data sheet right after the pin designation view. That table is a list of every pin, along with their pin number, their designated functions, and descriptions for every function. We use that table to learn about the meaning of each function.

---

## Module Functional View

The functional view of a module is a schematic which shows the logical flow of data through a module, how a module is configured, how a module is controlled, and how a module is monitored.

Documentation for every module includes a functional view. Such a view is typically called a System Block Diagram. Every module, except for one, has its functional view published by their microcontroller's user guide. That remaining module has its functional view published by their microcontroller's data sheet, and that module is called the Digital I/O Module. Furthermore, the data sheet does not call it a system block diagram, it is called a *port input/output diagram*.

Diagram 8 is an example of a functional view. In this case it's the functional view of the digital I/O modules in a port. Only one module is shown, but it generically represents all eight modules (0 through 7) in port 1. On the right is an octagon that represents the terminal pin on the case of the microcontroller. All functional views use the same logic symbols as this one, but some views will include symbols which are not shown here. Texas Instruments publishes a free document that defines many such symbols. It's called an "Overview of IEEE Standard 91-1984: Explanation of Logic Symbols," and it can be downloaded as document number SDYZ00.

Here is how to interpret the view. The voltage signals will either flow from left to right or from the right to left, depending on how the module is configured.

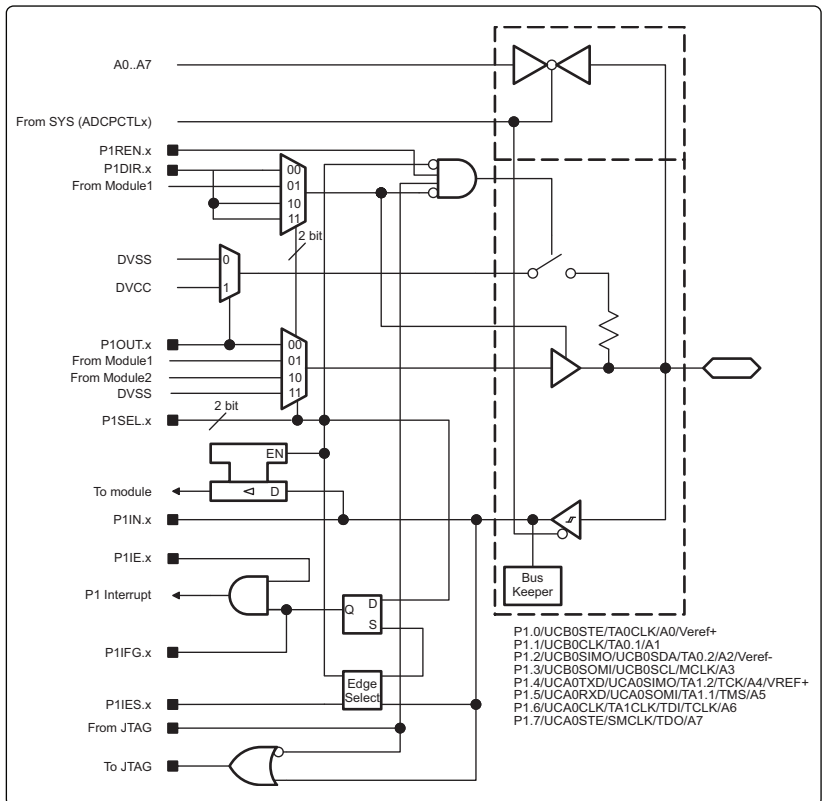
On the left side of the view, we see a vertical arrangement of symbols. They are in the form of filled squares, the end of lines having no decoration, and lines having an arrowhead. The filled squares represent one or more bitfields which our firmware may read and manipulate. Squares which represent more than one bitfield will be connected to a line that represents a bus of lines. The bus is represented by a single line having a short diagonal line across it with a label above. The label indicates the number of lines in the bus in terms of bits. The end of lines which have no decoration will represent paths for voltages signals to enter or exit the module. Lines which have an arrow will represent a path for a voltage signal and the direction which it must flow. On the right side of the view, we see a stretched hexagon. That represents a single physical pin on the microcontroller's case.

At the interior of the view we see several other symbols. Many of them represent logic gates, such as AND and OR gates. The two isosceles trapezoids represent port signal multiplexers, and the bits in a multiplexer represent the various bitfield patterns needed to switch the path. An empty triangle represents a buffer where the voltage signal is held to some low or high digital state. The bowtie, having two triangles facing each other with a circle in between them, represents a transmission gate. The

gate allows a signal to flow from left to right or from right to left, but the circle, which under the control of another signal, opens and closes the gate. The triangle which contains a parallelogram that looks like the letter S is a Schmitt trigger. It is a voltage level sensing circuit, so it is used for determining whether an inbound signal is a binary low or high signal. And the last remaining symbol which is worth mentioning is in the form of a square with the letters S, D, and Q in it. It is a set-reset block of logic, also referred to as a digital set-reset flip flop gate. It behaves as a primitive counter. The inputs S for set, and D for data (instead R for reset), will increment the output Q between a digital low and high signal.

Many functional views include notes, such as we see in the lower right corner of this example view. In this case, the notes are in the form of a list of pins and their functions. This is a generic functional view of port 1 and its eight pins: P1.0 through to P1.7. So these notes just simply tell us which functions are available at the pins depending on which port pin this view represents.

**Diagram 8:** The functional view of a Digital I/O Module as published by a microcontroller's data sheet. Most functional views are published by a microcontroller's user guide. Since a port is built with a set of these modules, this is a generic view of modules 0 to 7 in port 1. The terminal pin is shown on the right as an octagon. While the data sheet calls such a functional view a "port input/output diagram," the user guide will call the same type of functional view as a "system block diagram."





## Visualizing the Main Memory

The underlying concept for programming an MSP430, and any other microcontroller, is being able to write program instructions which can read and write into its memory. Although the memory is where our program is stored, what are of most concern to us are the registers in memory. They are used for configuring, controlling, and monitoring the microcontroller's systems and modules. They are also used for getting data from a module which our program needs for making decisions, and they are used for putting data back into a module which are the results of the decisions our program had made.

Before learning about how to read and write into those registers, we must know about the types of memory, their structures, and the tables which document and describe the registers in memory. After we understand these topics, we must then learn about the reset system, since that knowledge will prepare us to go into much greater detail about register tables and how to use them. Once we reach that point, we will be ready to learn how to write code that reads and writes data into a microcontroller's registers.

---

### Main Memory Structure

Having a basic knowledge about the structure of main memory is a prerequisite to understanding microcontroller documentation. We use that documentation for learning about modules, their registers, and the purpose of individual register bitfields. And that knowledge allows us to write program instructions which can read and manipulate those registers as needed.

Main memory is used for storing a firmware image and nothing else. The memory is also known as the program execution environment. The MSP430 compiler uses our program code to build an image, and a program development tool, such as Code Composer Studio, loads the image into main memory.

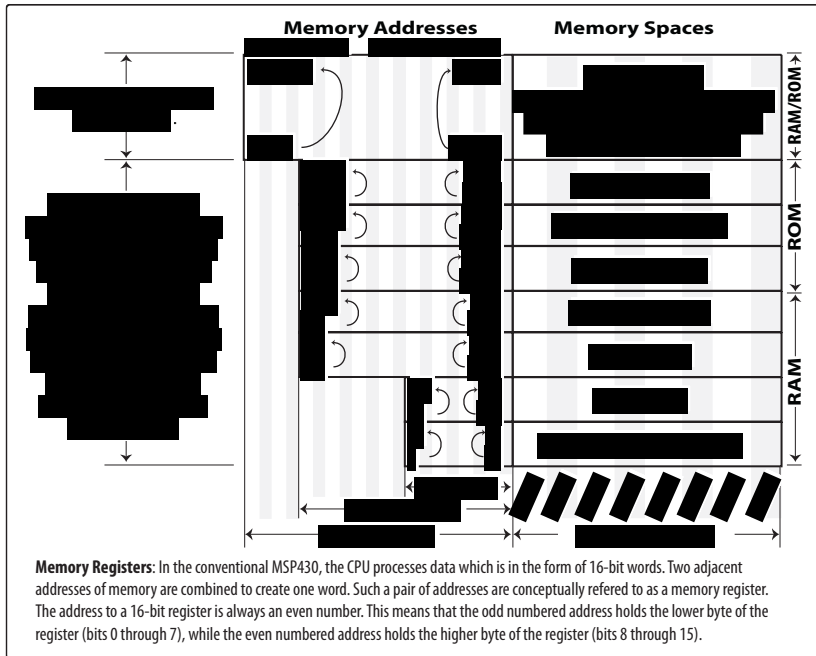
Main memory can be visualized as a stack of rows. Each row contains an address and its eight bitfields of storage. A single bitfield is a place used for storing a single bit of data.

For the conventional MSP430 CPU, the stack will be 65,535 addresses high, while microcontrollers built with the CPUX will have a stack that is 1,048,575 addresses high. The range of addresses is called the Main Memory Address Space.

At the lower address numbers, which start at zero, is a section dedicated to all the registers. Each register has a name which is documented in the microcontroller's user guide (found at the microcontroller's home page). Above the registers are the higher address spaces which are dedicated sections of memory used for storing various types

of volatile and non-volatile data. Volatile program data is data which is produced by the program during runtime. Information memory is data about the microcontroller itself, such as calibration data used by our program for calculating accurate times which vary under temperature. Non-volatile program code is our actual program. And the interrupt vector table is a cross-reference between interrupt flags (IFGs) and interrupt service routines (ISRs). The section of extended memory will be some combination of RAM and ROM, depending on the microcontroller model.

**Diagram 9:** A memory map shows the logical structure of the main memory address space. The actual physical address space is distributed across two memory modules: the ROM module and the RAM module. The actual usage of each section of memory will be different from one microcontroller to another, so refer to its data sheet for that information.



## CPU Memory Structure

Like the main memory structure, the CPU registers can also be visualized as a stack, but one that is only sixteen rows tall. Each row has a register name and contains bitfields of storage. The amount of bitfields is based on the address space which the CPU must access, since they are used for handling main memory addresses. If the address space is built into a conventional MSP430 CPU, each register contains sixteen bitfields. If the registers are built into a CPUX, they are twenty bitfields wide.

Two nomenclatures are used for naming these CPU registers. The first one labels the entire set of registers as R0 through R15, and those are the names which may be used in programming code. The second nomenclature labels them with conventional names: the Program Counter, Stack Pointer, Status, Constant Generator, and General Purpose registers. Our firmware has access to all these registers, but we'll only be

concerned with the status register because it controls the operating mode for the microcontroller and its interrupt system.

## Introduction to Register Tables

Although an in depth explanation of register tables will be described later, an introduction to them must be made now to provide some background for a topic about the reset system which immediately follow this chapter.

A register table describes the register's name, the bitfield positions, the purpose of each bitfield, how the reset system initializes them, their behavior during operation, and the mask name for each bitfield. It is the primary source of documentation that we need for learning about a register. These tables can only be found in the user guide for a microcontroller. A user guide is typically written for a single family of microcontrollers, since a family has many characteristics in common. The guide can only be found at a microcontroller's product home page on the Internet.

**Diagram 10:** An image of a conventional register table. In this case it is the Timer A Control Register.

TACTL, Timer_A Control Register							
15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLRL	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
<b>Unused</b>	Bits 15-10	Unused					
<b>TASSELx</b>	Bits 9-8	Timer_A clock source select					
		00	TACLK				
		01	ACLK				
		10	SMCLK				
		11	INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)				
<b>IDx</b>	Bits 7-6	Input divider. These bits select the divider for the input clock.					
		00	/1				
		01	/2				
		10	/4				
		11	/8				
<b>MCx</b>	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.					
		00	Stop mode: the timer is halted.				
		01	Up mode: the timer counts up to TACCR0.				
		10	Continuous mode: the timer counts up to 0FFFFh.				
		11	Up/down mode: the timer counts up to TACCR0 then down to 0000h.				
<b>Unused</b>	Bit 3	Unused.					
<b>TACLRL</b>	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLRL bit is automatically reset and is always read as zero.					
<b>TAIE</b>	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request.					
		0	Interrupt disabled				
		1	Interrupt enabled				
<b>TAIFG</b>	Bit 0	Timer_A interrupt flag					
		0	No interrupt pending				
		1	Interrupt pending				

Most register tables provide an explanation which shows a picture of all the bitfields, their arrangement within the register, and their descriptions. There are typically two types of descriptions. One is about the data, meaning, the pattern of bits which may be written into or read from the fields, and the other is about the behavior they cause or indicate.

A data description has two purposes.

Setting a bit means changing the numerical value for a bit to a one digit (1), while clearing a bit means changing it to a zero digit (0). The second purpose is to indicate some type of output from the module. Therefore, bitfields are the data inputs and outputs from an individual module.

Next to the data description is the behavioral description of the bitfield. It will typically have many explanations. They include, but are not limited to, the contents of each bitfield when the microcontroller starts or restarts, whether our firmware can read and write into the bitfield, and how does the system or module which the register belongs to will affect or change the contents of the bitfields.

The register nomenclature tells us the name of the register, its programming symbol, and the individual programming symbol for each bitfield. A programming symbol is a sequence of alphanumeric characters. When used in our firmware instructions, it gives us direct access to the register and its bitfields. Later we will distinguish between two types of programming symbols and how to use them: the register variable and its bitfield masks.

There are two types of register tables which we must be concerned about.



## The Reset System and its Subsystems

Since a register table provides information that describes how the reset system affects a register's bitfields, having some knowledge about the reset system is a prerequisite for the next chapter which is about register tables. Furthermore, this topic is later elaborated upon by the “Reset Routines” on page 134.

The reset system is a set of logic circuits which handle a sequence of processes from when the microcontroller powers up or signaled to reset to when the CPU begins executing the firmware. Those processes are called the BOR, POR, and PUC. They also have a direct affect on the registers. They will initialize or re-initialize the state of their bitfields. This is important to understand because a program's design must take into account of the register's state after a power-up or reset.

The reset system basically executes this sequence of operations: to assure the microcontroller powers up properly and begins stable operation, to initialize the microcontroller's registers, and then call the boot program so the firmware execution environment is initialized. The last instruction in the boot program calls the `main()` function so the CPU will begin executing our program.

---

### Power-Up

Power-up is a scenario where the microcontroller is in a completely unpowered state, what we call off, and then power is applied to it. The first system to begin operating is the reset system. It handles a controlled power-up by managing the amount of power supplied and then uses the BOR, POR, and PUC sequence to initialize the registers. Initializing the registers means writing bits into them so they are in a specifically configured state.

When the reset system is finished, it then loads the address to the first instruction of the boot program into the CPU, and then puts the microcontroller into the active operating mode so the CPU can begin executing the boot.

Details of the boot program are explained later, but you should know that the last instruction in boot program will transfer the flow of program execution to the `main()` function by calling it. That function contains the instructions which form the program we develop and load into the microcontroller.

---

### Reset

Reset is a scenario where the microcontroller is in some type of operating mode, and then an event occurs which signals the microcontroller to restart at a BOR, POR, or PUC. A reset will re-initialize the registers, then reboot the microcontroller, then load the `main()` function into the CPU, and then put the microcontroller in active mode.



The events which cause a reset are many, and they are described by sections which follow.

And finally, a reset event produces its own unique interrupt flag, which in turn, produces a reset signal in the form of a non-maskable CPU interruption (NMI). Therefore, depending on which flag is set, the reset system will begin at the BOR, POR, or PUC.

---

### The BOR, POR, and PUC Sequence

An important concept about the reset system is its phased approach to initializing the registers. Current designs of the reset system use a sequence of three phases which are called the Brownout Reset (BOR), the Power-On Reset (POR), and the Power-Up Clear (PUC).

A power-up will execute each phase and in this specific order: the BOR, POR, and then PUC. A reset will begin the sequence at a particular phase, and the chosen phase will depend on the type of reset event.

The microcontroller's user guide publishes a diagram that shows the power-up and every type of reset event, the reset phases, the operating modes, and the flow of execution through them all. The section is typically named Operating Modes. It gives you the overall picture of the microcontroller's operation. An image of it is shown on page 132.

---

### Register Table Bitfields

A register table will show which bitfields will be initialized or re-initialized during a BOR, POR, or PUC. It is important to take this into account when designing a program, so that a power-up or reset event can be properly handled by our program.

---

### Reset Signals

Events which cause the reset system to begin at will vary from one microcontroller to another, but not by very much. They can be classified as BOR, POR, and PUC signals.

---

### BOR Signals

These are the events which will typically produce a BOR signal.

- t
- 
- 
- 
- 
- 

- A program instruction manipulating a BOR signaling bit in some power monitoring module register

The reset pin is analogous to the restart button on a personal computer and many computer controlled devices. A fractional low power operating mode is referred to as LPMx.5, and it is an extremely low energy mode that removes power from main memory.

The voltage supply supervisor is typically responsible for monitoring the voltage level supplied to the microcontroller. If it senses the supply going below a specific level, a BOR signal is produced. That level is typically 1.8 volts. Low supply voltages are called power brownouts.

---

### False BOR Signals

[REDACTED]

[REDACTED]

---

### POR Signals

Current models of the MSP430 typically have two events which will produce a POR signal.

- [REDACTED]
- [REDACTED]

---

### PUC Signals

Five types of events will produce a PUC signal.

- [REDACTED]
- [REDACTED]
- [REDACTED]
- [REDACTED]
- [REDACTED]

An overflow means that the watchdog timer has counted up to a specific interval of clock cycles. As for the FRAM error, FRAM technology uses an algorithm that must write data back to the place where it was read and for checking for the quality of the written data. If there is an error, that error produces a FRAM uncorrectable bit error signal. For additional information, see page 134.



## How to Read and Use the Register Tables

A previous chapter had introduced us to the memory register table, and that was followed by an explanation about the reset system. We are now ready to go into greater detail about the register table, since that knowledge is needed for developing firmware code which can read and write into registers.

Registers are used for configuring, controlling, and monitoring a system or module. For example, a program writes data into a register to configure a module, or a program may read data from a register to make a decision, and the result of the decision is then written back into a register to produce a new module configuration. Registers are used for telling a system or module to work in a specified way, while a register table is documentation that tells us about the different configurations which a register can be put into or appear as.

There are basically two types of register tables. One type is the conventional register table, and it's used for describing all types of system and module registers. The other type is the port register table. It is only used for describing registers which belong to a digital I/O port module.

---

### The Conventional Register Table

Shown below, by Diagram 11, is a typical register table published by an MSP430 user guide. All modules, except for the digital I/O, will use this type of table. In this case, it's a table for the Timer A module. That module is used fo

Reaching the end of the interval is also known as . And the flag may be used as a trigger for executing an interrupt service routine (ISR).

As with many other modules, this is not the only register for Timer A. The purpose of this register is to provide

On the left side of Diagram 11 is a column of numbers that will not appear in a published table. It's printed there to help explain the contents of the table.

---

### Register Variable and Register Name

Shown at line 1 is the table's title. It is made of the register variable and the register's name. The variable is very important to recognize, since a program must use it for . The variable name appears as TACTL, meaning, Timer A Control.

## Register Variable

A register variable can be understood from two points of view. One is from a logical point of view, and the other is from a programming language point of view.

From a logical point of view, a register variable is a structural abstraction of all the bitfields in a single register. It provides our firmware with a

It is the primary identifier that we use in our firmware instructions for

**Diagram 11:** Example of a conventional register table. In this case, it is of the Timer A control register. The column of numbers which appear at the left side are for descriptive purposes and will not appear in a published table.

1 TACTL, Timer_A Control Register							
2	15	14	13	12	11	10	9 8
3	Unused					TASSELx	
4	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
5	7	6	5	4	3	2	1 0
6	IDx	MCx	Unused	TACLx	TAIE	TAIFG	
7	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
8	Unused	Bits 15-10	Unused				
9	TASSELx	Bits 9-8	Timer_A clock source select				
10			00 TACLK				
11			01 ACLK				
12			10 SMCLK				
13			11 INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)				
14							
15	IDx	Bits 7-6	Input divider. These bits select the divider for the input clock.				
16			00 /1				
17			01 /2				
18			10 /4				
19			11 /8				
20	MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.				
21			00 Stop mode: the timer is halted.				
22			01 Up mode: the timer counts up to TACCR0.				
23			10 Continuous mode: the timer counts up to 0FFFFh.				
24			11 Up/down mode: the timer counts up to TACCR0 then down to 0000h.				
25	Unused	Bit 3	Unused.				
26	TACLx	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLx bit is automatically reset and is always read as zero.				
27							
28	TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request.				
29			0 Interrupt disabled				
30			1 Interrupt enabled				
31	TAIFG	Bit 0	Timer_A interrupt flag				
32			0 No interrupt pending				
33			1 Interrupt pending				

From programming language point of view, a register variable is a C programming language preprocessor directive. The C language has several types of preprocessor directives. This type is a macro. A macro is a set of one or more lines of programming instructions. The instructions, in this case, handle all the pointer operations needed for accessing the contents of a register at a specific address in main memory. All register variables, are defined by their own preprocessor directive macro and declared in the microcontroller's header file. When we build a firmware image, instructions which form the macro will

Without the register variable, we would have to write complex pointer instructions to specific addresses in main memory in order to. And furthermore, those pointer instructions would have to be edited with the correct address whenever we had to port (meaning edit and prepare) our program to run on a different model of

MSP430. All this is handled automatically by [REDACTED] which are published by register tables and defined in each microcontroller's header file, and which are designed to be used across all models of MSP430 microcontrollers.

When writing into a register, we typically don't have to be concerned with the [REDACTED] data size. But when reading it, we do have to be concerned. If we ever need to read the entire contents of a register, and then assign that data to a storage variable, our storage variable must take into account the size, or width, of the register. A register may be [REDACTED], so the storage variable should be declared as either a [REDACTED] an [REDACTED]), or a [REDACTED] type of data respectively.

Here is why we must take size into account. Declaring a storage variable with too large of a data type will occupy unnecessary space in memory, while declaring it with a too small of a data type will amputate some bits from the data.

Also, be aware that all the bits in a register represent a [REDACTED]. Therefore, the data type for the variable must be unsigned. Registers do not typically store negative numbers. Unless it is some specialized type of data register, such as a register belonging to the [REDACTED] system module which is used for [REDACTED].

[REDACTED] data types (data which is typed as [REDACTED] are actually [REDACTED] bit binary numbers, so they are the perfect size for storing data which is read from [REDACTED] bit registers.

As for [REDACTED] and [REDACTED] types of data, they can vary in widths among computers. For the MSP430, [REDACTED] types are [REDACTED] bits wide, so that data type is good for reading [REDACTED] bit registers. Long [REDACTED] types are [REDACTED] bits wide, and that type is good for reading [REDACTED] bit registers.

---

## Opening the Header File where the Register Variables are Declared

When you create a new development project with Code Composer Studio (CCS), the appropriate header file is automatically added to the project. If we command CCS to create a main.c file in our project, the file will contain an #include preprocessor directive that declares a generic msp430.h file, which in turn, includes the specific header file which is written for the microcontroller. The file will be named with the model number of the microcontroller and use the H filename extension.

When a register variable or bitfield mask appears in your code, you may view the header file by [REDACTED]

---

## Register Bitfields

At lines 2 through 7 is a diagram of the entire register. Its bitfields are numerically indicated at lines 2 and 5. There are sixteen bitfields. Module registers are located inside of main memory where every address can only store eight bits. Therefore, the sixteen bits in this register must be distributed across two adjacent addresses. The bit-

fields at line 3 are located at the higher address in memory, while the bitfields at line 6 are at the next lower address. Keep in mind that addresses are of no concern to us, since we'll be using the [REDACTED] to access them. And if this was an eight bit register, the diagram would only present eight bitfields.

Notice that field 3 and fields 10 through 15 are labeled as Unused. That's exactly what they mean. The remaining bitfields are places which do provide services. Another conceptually important fact is that the name for each field represents a specific bitfield mask.

---

### Bitfield Mask

From a logical point of view, a bitfield mask is an abstraction of a specific bitfield or set of bitfields in a single register. It provides our firmware with an identifier that directly correlates with the specific field or sets of fields in a memory register. It is the primary identifier that we use with a register variable for choosing the specific bitfields in a register that we want to read or manipulate.

From a programming language point of view, a bitfield mask is a preprocessor directive in the form of a symbolic constant. Meaning, it is an identifier that literally symbolizes some binary number which is sixteen bits or less in width. The number must express the proper pattern of zeros and ones to act as a mask. The places in the number where a zero digit is located will indicate that the correlating bitfield must be ignored, while the places where a one digit is located will indicate that the correlating bitfield must be read or manipulated. Therefore, the symbolic constant must be a number that expresses a [REDACTED].

When the register variable and bitfield mask are used together in a firmware instruction, the operation is understood as [REDACTED]. This will be elaborated upon by a later section.

The underlying symbolic constant that the mask represents is declared and defined by its own [REDACTED] in the same header file as the register variables.

---

### Bitfield Mask Suffix

Notice that some bitfield masks include the letter x as a suffix. For example, the mask for fields 8 and 9 is TASSELx. Since the mask represents two bitfields, and two bitfields can provide four different patterns of bits, the suffix provides a place in the mask for distinguishing one pattern from another. At lines 10 through 13 you can see those four patterns, and each pattern provides us a different option. The replacement text for the suffix is defined by the header file for the microcontroller.

Be aware that the [REDACTED] for the suffix is somewhat intuitive, but until you have developed enough experience with using them, you will have to read the header file to obtain the concise replacement text. The replacement will typically be some [REDACTED] that will numerically begin at zero. For example, the replacement text could be [REDACTED]. But there is no standard. It might include an under-

score; for example,

or some masks the replacement text will be more elaborate, so reading the header file will confirm the precise suffix.

## The Standard Bits

If you are in a hurry, or just simply not interested determining the actual replacement text for the suffix, the header file provides a set of symbolic constants which can be used as generic masks. They are called the Standard Bits. By incorporating hexadecimal notation, they are named BIT0 through BIT9 for fields zero through nine, and BITA through BITF for fields ten through fifteen. So for example, instead of using the symbol TASSELx for fields 8 and 9, you could use the symbols BIT8 and BIT9 to mask those same bitfields. Their usage in a firmware instruction will be elaborated upon by later sections.

Standard bits are commonly used for masking digital I/O registers. However, using the standard bits instead of the bitfield masks which are designed for masking specific registers is not a good practice because of three reasons: readability, our comprehension of the mask is reduced, and most importantly, code portability across other MSP430 devices can be negatively affected.

**Code Example 1:** The standard set of masks for manipulating bits in GPIO registers. This is how those sixteen standard bits are defined in a microcontroller's header file. The actual symbolic constants which we would use in our code are seen as BIT0 through BITF.

---

```

1 #define BIT0 (0x0001) //in binary format: 0000 0000 0000 0001
2 #define BIT1 (0x0002) //in binary format: 0000 0000 0000 0010
3 #define BIT2 (0x0004) //in binary format: 0000 0000 0000 0100
4 #define BIT3 (0x0008) //in binary format: 0000 0000 0000 1000
5 #define BIT4 (0x0010) //in binary format: 0000 0000 0001 0000
6 #define BIT5 (0x0020) //in binary format: 0000 0000 0010 0000
7 #define BIT6 (0x0040) //in binary format: 0000 0000 0100 0000
8 #define BIT7 (0x0080) //in binary format: 0000 0000 1000 0000
9 #define BIT8 (0x0100) //in binary format: 0000 0001 0000 0000
10 #define BIT9 (0x0200) //in binary format: 0000 0010 0000 0000
11 #define BITA (0x0400) //in binary format: 0000 0100 0000 0000
12 #define BITB (0x0800) //in binary format: 0000 1000 0000 0000
13 #define BITC (0x1000) //in binary format: 0001 0000 0000 0000
14 #define BITD (0x2000) //in binary format: 0010 0000 0000 0000
15 #define BITE (0x4000) //in binary format: 0100 0000 0000 0000
16 #define BITF (0x8000) //in binary format: 1000 0000 0000 0000

```

---

## Register Bit Accessibility and Initial Condition

At lines 4 and 7, of diagram 11 page 44, are notations that describe the accessibility and initial condition for each bitfield.

Accessibility means whether our firmware is able to read or write into the field. For example, all bitfields in this register display the letters rw below them, which means, each field can be read or written into.

The initial condition means which numerical value automatically appears in the field after the microcontroller starts or restarts. That value is either a zero or one. The start



and restart processes are explained by the “The Reset System and its Subsystems” on page 39.

This accessibility and initial condition information is important for us to take into account while designing and developing code. If the conditions of a register at either a POR or PUC are not what our program requires or anticipates, we’ll have to write instructions which set or clear bits to meet the requirements of our program.

In the example shown by diagram 11, the initial condition for each bitfield is described with the notation  $-(0)$ . This means that a POR event will clear the field to a value of zero, and remain that way until our firmware or some other event changes it. If a PUC were to be the reset subsystem which initialized the bitfield, the notation would be  $-0$ . A parenthesis will distinguish the difference.

There are many other notations for us to be aware of, and they are all documented by the microcontroller’s user guide, in the section named “Register Bit Conventions,” in the guide’s preface.

**Table 1:** The typical register bit accessibility and initial condition table as published by the preface of a user guide.

<b>Key</b>	<b>Bit Accessibility and Intialization</b>
rw	read/write
r	read only
r0	read as 0
r1	read as 1
w	write only
w0	write as 0
w1	write as 1
(w)	No register bit implemented; writing a 1 results in a pulse. The register bit always reads as 0.
h0	cleared by hardware
h1	set by hardware
-0, -1	condition after PUC
-(0), -(1)	condition after POR
-[0], -[1]	condition after BOR
-{0}, -{1}	condition after brownout

## Bitfield Descriptions

Below the diagram of the register, at lines 8 through 33 of Diagram 11, is a table that describes the purpose and usage of each bitfield. It typically consists of three columns.

One column is for the field’s mask, a second column identifies which fields the mask handles, and a third column shows a name for the mask and the purpose of the bitfield or set of bitfields it handles.

In the third column, but below the mask name and purpose, are listed the different options for configuring the fields. For example, at line 10 we see `00 TACLK`. That means if bit 9 and bit 8 are both zero, then when the Timer A module reads those fields, it will configure TACLK as the timing signal. Although it is not described by this register table, the TACLK signal is explained by user guide sections that precede it. In this case, it is a clock signal that would be supplied from outside of the microcontroller, by a device also known as an external oscillator.

Let's look at one more example. At line 11 we see `01 ACLK`. That means if bit 9 is zero and bit 8 is one, then when the module reads those fields, it will select ACLK as the clock signal. ACLK is an acronym for the Auxiliary Clock, which is one of the clocks built directly into the microcontroller. Many acronyms are defined by the glossary located in the preface of the user guide.

---

### Interrupt System Bitfields

There are two `00000000` in this register (shown by diagram 11) which deserve some mention, since these types of `00000000` appear in registers belonging to many other systems and modules. One is of type interrupt enable (IE), and the other is of type interrupt flag (IFG). Their acronyms are typically prefixed with letters and numbers which distinguish them as belonging to a specific module or system. In this register one is called Timer A Interrupt Enable (TAIE), and the other is called Timer A Interrupt Flag (TAIF).

Those bitfields are associated with the CPU interrupt system. If the interrupt enable field is set, a type of event, which can occur in the module, will set the interrupt flag and be used for triggering an interrupt service routine (ISR). An ISR then carries out the instructions which you have written into it. This feature allows you to write event-driven firmware.

The description for these two interrupt related bitfields does not explain `00000000`, `00000000`, but a user guide section that precedes this table does tell us. In this case, the section is called Timer A Interrupts, and it says that three different events will set a flag. For this flag, named TAIFG, a timer overflow will trigger it. An overflow is just simply when the timer has counted through some specific interval of clock cycles. The interval is configured with another register.

---

### Using a Register Table and Functional View to Help Develop Code

We use a register table to learn about the bitfields which configure, control, and monitor for signals which are handled by a module. And we use a functional view (page 32) to learn about the paths where signals enter, flow through, and exit a module. Also keep in mind that signals include `00000000`. The paths are annotated with points where bitfields may `00000000` the signals. Those points appear as solid squares and are labeled with the specific bitfield mask which `00000000` that point.

With the knowledge about register tables and module functional view, we have at least a couple of approaches to developing code for configuring a module. One is a less abstract, while the other is more abstract. The two are distinguished by whether the module is documented by a functional view that shows a distinct signal path through it or not.

Here is how the less abstract, more orderly, approach works.

Use the more abstract approach when the module has a functional view which does not show a through it. This approach will

---

### Distinguishing between a Digital I/O Module and Port

If you are not familiar with digital I/O modules and ports, the way Texas Instruments has, and seems to continue with documenting information about them, can be a little confusing. Generally speaking, a user guide will treat them as , while a data sheet will treat them as .

Arguments can be made about whether they are the same thing or not. So in order to mitigate any confusion, and to create a better context which explains them, this book will treat the digital I/O module and a port as two different things.

---

### Digital I/O Module

An earlier chapter had introduced the concept that a port contains channels, and that each channel is actually made of two sub-channels. One sub-channel handles input signals, so it's called the input sub-channel. The other channel handles output signals,

so it's called the output sub-channel. The user guides and data sheets do not use the sub-channel nomenclature; they just refer to both sub-channels as a single channel that can be configured as an input path or an output path, but not simultaneously as an input and output. The module functional view of a port, as shown by diagram 8 on page 33, and as published by the microcontroller's data sheet clearly shows those paths.

The purpose of the input sub-channel is to [REDACTED]

).

The purpose of the output sub-channel is to [REDACTED]

Let's take a closer look at how digital signals are characterized by and for the MSP430. We know that a digital signal has two states: a low and a high voltage state. The exact amount of voltage that defines a low or high state is completely dependent on 1) the amount of voltage supplied to the microcontroller's power supply terminals, 2) whether the voltage is within the context of an input or output signal, and 3) the model of microcontroller. All that information can be found in the microcontroller's data sheet. But generally speaking, for digital input voltages, the low state is no higher than about 40% of the supply voltage, while the high state is no lower than about 75% of the supply. A filter, called a Schmitt Trigger, handles the input signals, which is also described by the microcontroller's data sheet. For digital output voltages, the low voltage state is about 0.3 volts above whatever ground is (typically zero volts), while the high state is about 0.3 volts below the supply voltage.

---

## Port

A port is defined as a set of channels and their supporting circuits where input or output signals may flow through. A set will typically contain eight channels, but some microcontrollers have ports with fewer channels. Keep in mind that a single channel is actually two sub-channels, one for input and one for output.

The supporting circuits provide services which maintain the flow of different types of digital and analog signals. Those services include, but are not limited to, signal switching, voltage signal buffering, and voltage signal comparing. Therefore, the port is at the center of a microcontroller's ability to handle different types of signals. So along with its set of modules, this is why the MSP430 is called a mixed signal processor (MSP).

When more than one port exists, they are alphanumerically identified; for example, P1, P2, P3, and so on. Microcontrollers which have many ports will combine adjacent ports into pairs and name them alphabetically; for example, PA, PB, PC, and so on.

## Port Register Tables

An earlier section had introduced, what this book calls, the conventional register table. The purpose of that name is to distinguish the typical register table from the port register tables, since the format is different. Furthermore, there are three different formats which are used for describing port registers.

Be aware that port register tables are

## Port Channels, Port Register Bitfields, and Port Register Bitfield Masks

A very important concept about programming port registers is the relationship between a single port channel and a port register's bitfield. This is not explained by the user guides, it is only implied. For example, port channel 0 is handled by register bitfield 0, while channel 1 is handled by bitfield 1, and so on with the remaining channels in the port.

Another matter which is not explained by the user guides is about port register bitfield masks. The bitfields in a port register

## First Type of Port Register Table

This may possibly be the first and original type of format used for describing the registers for a port. The obvious difference between this and a conventional table is that we see no register bitfields. Furthermore, we see no indication about the number of channels at the port. The data sheet tells us that information.

**Diagram 12:** The first type of port register table.

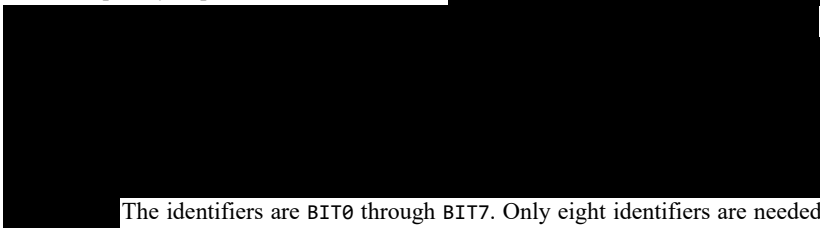
Port	Register	Short Form	Address	Register Type	Initial State
P1	Input	P1IN	020h	Read only	-
	Output	P1OUT	021h	Read/write	Unchanged
	Direction	P1DIR	022h	Read/write	Reset with PUC
	Interrupt Flag	P1IFG	023h	Read/write	Reset with PUC
	Interrupt Edge Select	P1IES	024h	Read/write	Unchanged
	Interrupt Enable	P1IE	025h	Read/write	Reset with PUC
	Port Select	P1SEL	026h	Read/write	Reset with PUC
	Port Select 2	P1SEL2	041h	Read/write	Reset with PUC
	Resistor Enable	P1REN	027h	Read/write	Reset with PUC

The example table shows a set of registers for a single port, named P1, as indicated by column one. Every line in the table represents a single port register. Column two lists all the registers which configure and control the port. Nine are listed here by name. Be aware that the first and second ports of an MSP430 will typically include registers for configuring CPU interruptions, while the remaining ports will not have that feature.

Column three lists the short form name for each register. The short form is actually the register variable. So the variable `P1IN` provides us direct access to the bitfields of the Port 1 Input register. When you read the descriptions for each register, which are published by user guide sections which precede this table, these variables will be described as `PxIN`, `PxOUT`, `PxDIR`, and so on. The letter `x` represents the port number, so if there was a Port 2 table shown here, the register variable for its input register would be `P2IN`, and so on. This allows the same naming format to be used across all ports, which makes them easier to remember.

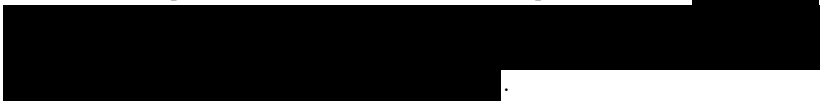
What is not apparent is the size of each register and the details about each bitfield. The typical port register is eight bitfields wide, where each bitfield handles a single port channel. The functional details of each register are explained by user guide sections that precede the table.

It is conceptually important to understand that



The identifiers are `BIT0` through `BIT7`. Only eight identifiers are needed since a typical port register has only eight fields, unless you have the four channel port.

Be aware that a port channel number and a terminal pin number are



Column four, named Address, lists the precise address to the register in main memory. If the name of the column was Offset, the number in the column would represent the amount of addresses which the register is located past a base address for the Port 1 module. The base address is the precise address number in main memory, which is published by the microcontroller's data sheet, in a section named Peripheral File Map. For example, if the offset address for the Output register is `02h`, then that register is located two addresses past the module's base address in main memory. The base address, in other words, is the actual address to the first register for Port 1. Keep in mind that when developing instructions which read and manipulate register bitfields, we'll be using the register variables. Therefore, these addresses will be of no use to us.

Column five lists the register's [access] type, meaning, whether our firmware can read or write into it. Notice that the input register is read only, which should make sense, because our firmware will only be used for reading input digital voltage signals placed onto a pin. As for the output register, it is used for producing digital voltage signals, which means, firmware will have to set bits in the register to do that, but

firmware can also read this bitfield to determine its state. Clearing an output register's bitfield removes any voltage output signal at the pin.

Column six lists the initial state of all the bitfields in the register after the microcontroller has started or restarted. For the input register, there is a dash, which means that the state of the bitfield depends on the external signal flowing into the pin. So during a start or restart, if the external voltage signal coming into the pin is high, then the initial state at that bitfield is a digital value of 1. As for the output register, its initial state is listed as Unchanged, which means the register acts like nonvolatile memory: it permanently stores the state. So when the microcontroller starts or restarts, the state at which the bitfield was before the start or restart will be that same state. The last initial state description is Reset with PUC (Power-Up Clear). This means when the microcontroller restarts at the PUC, the system reset circuitry clears the register bitfields to zero. This nomenclature can be confusing because, within the context of microcontrollers, the word set means to write a digital value of 1 into a bitfield, so reset could be interpreted as rewriting a 1 into the bitfields. If we have doubts about a nomenclature's meaning, it's best to write some code to test the behavior of the field during a start and restart.

## Second Type of Port Register Table

The second type of port register table is very similar to the first type.



**Diagram 13:** The second type of port register table.

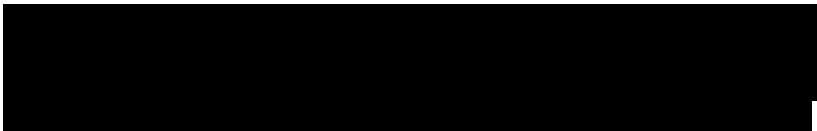
Offset	Acronym	Register Name	Type	Access	Reset
00h	P1IN or PAIN_L	Port 1 Input	Read only	Byte	
02h	P1OUT or PAOUT_L	Port 1 Output	Read/write	Byte	undefined
04h	P1DIR or PADIR_L	Port 1 Direction	Read/write	Byte	00h
06h	P1REN or PAREN_L	Port 1 Resistor Enable	Read/write	Byte	00h
08h	P1DS or PADS_L	Port 1 Drive Strength	Read/write	Byte	00h
0Ah	P1SEL or PASEL_L	Port 1 Port Select	Read/write	Byte	00h
18h	P1IES or PAIES_L	Port 1 Interrupt Edge Select	Read/write	Byte	undefined
1Ah	P1IE or PAIE_L	Port 1 Interrupt Enable	Read/write	Byte	00h
1Ch	P1IFG or PAIFG_L	Port 1 Interrupt Flag	Read/write	Byte	00h

The first column, named Offset, lists the number of addresses past a base address in main memory where the register is precisely located. Be aware that nomenclature may use names such as base, address, offset, or something like that to denote the offset addresses. Don't let this confuse you, because all the port register addresses will be listed in numerical sequence, starting at 00h (0x00), which is typically the first address in main memory. And keep in mind that these addresses will typically be of little or no use to us since we'll be using variables to access the registers.

The second column, named Acronym, is the register variable. Two variables are shown here, and their usage presents two points of view. The first view is of a single

register, and the second view is of a pair of registers. The first register variable, for example P1IN, is a view of a single register. When using this variable, it should be clear to us that when using this symbol we are accessing a single register named P1IN.

The second register variable, PAIN\_L, is a view of a pair of registers that belong to a port named A. Microcontrollers with many ports may logically (not physically) combine two ports into a pair in order to accommodate a different perspective of its ports. For example, a microcontroller having eight ports may logically combine them into sets of two: Port A for Ports 1 and 2, Port B for Ports 3 and 4, and so on. Register variables have been defined and declared in the microcontroller's header file to accommodate this perspective. So in this table we see PAIN\_L, which is an acronym for Port A Input Register Lower. This is literally the same identifier as P1IN, but redeclared with the identifier PAIN\_L. If the table for Port 2 were shown here, it would have the acronyms P2IN and PAIN\_H. The letters L and H denote the register's address in main memory. The letter L means the lower address in memory, and the H means the higher address.



As for the output register and interrupt edge select register, their initial states are listed as Undefined. Regardless of what the word undefined means, there is always an initial condition after a start or restart. We just have to figure it out on our own, so that involves some investigation. Documentation that precedes this table is the best place to start. Located in the Digital I/O chapter of the user guide, there may be a section named “Configuration After Reset.” If not, you may have to go to an earlier chapter which may go by the name of “System Resets, Interrupts, and Operating Modes.” In this case, the first section does exist, and it presents two initialization scenarios: one after a BOR and another after a POR or PUC. If the scenario is a BOR initialization, the documentation says all port channels are in a high impedance state with the port module's functions disabled to prevent any cross current. In other words, the port is not activated so that there is no voltage at the channel pins and no current can flow in or out of the pins. If the scenario is a POR or PUC initialization, the documentation states that the direction registers are set to configure all the channels as inputs, but the port will not be activated. So when you see a register table list-



ing the initial conditions as undefined, or something like that, it unfortunately means you have to determine that on your own. You may have to refer to user guide for another MSP430 microcontroller for some clues, or ask Texas Instruments.

### Third Type of Port Register Table

The third and last type of table format used for describing a port register is very much like the conventional register table format used for other types of modules.

The example here shows a diagram of the register along with a table that describes each bitfield as actually published by a user guide. The register is eight bits wide, its variable is PxDIR, firmware can read and write into each bitfield (rw), and the initial state of each field is zero (-0). The register is called the Port X Direction Register, and it is used for producing or sensing digital voltage signals at the pin which the channel services. Each bitfield has a one-to-one relationship with a port channel. For example, bitfield 2 will handle signals at port channel 2.

**Diagram 14:** The third type of port register table.

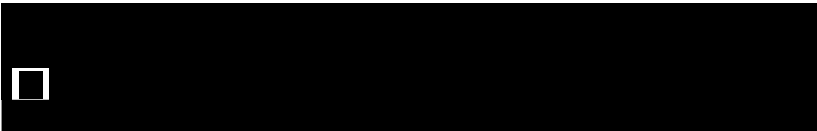
PxDIR Register							
7	6	5	4	3	2	1	0
PxDIR							
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

**P1DIR Register Description**

Bit	Field	Type	Reset	Description
7-0	PxDIR	RW	0h	Port x direction 0b = Port configured as input 1b = Port configured as output

What is not shown is the port number. That is implied, since this is a generic direction register diagram for all the ports. So if we were writing an instruction that configures the direction register for port 1, the variable for that register would actually be P1DIR.

Below the register diagram is the bitfield description table for the register. It is labeled as P1DIR Register Description. The column named BIT shows the number of bitfields, zero through seven (7-0), meaning, the following descriptions apply to each bitfield in the register. The next column, named Field, is incorrectly filled with the register variable, PxDIR. This is an actual published mistake, since a register variable is not a bitfield. The actual purpose of this column is unknown, so it reminds us that nobody is perfect.



(1b). The notation used there is binary (it would be less complicated to us if just decimal 0 and 1 were used by the description). As we will learn later, the standard bits are used for clearing and setting bits in these bitfields.



## Code Composer Studio Usage Tips

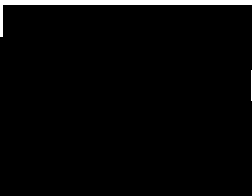
Our development tool will be Code Composer Studio (CCS). So we'll also be using it to load our firmware program into the microcontroller and removing bugs and testing the program. That work involves stepping through each instruction in the program to watch its behavior. And part of that behavior is characterized by the storage variables and registers it manipulates. When CCS is in that state of use, it is in [program] debug mode. So when in that mode, we'll often be interested in stepping through every single instruction, and we'll also be interested in having CCS present the contents of a variable or register in a desired number format.

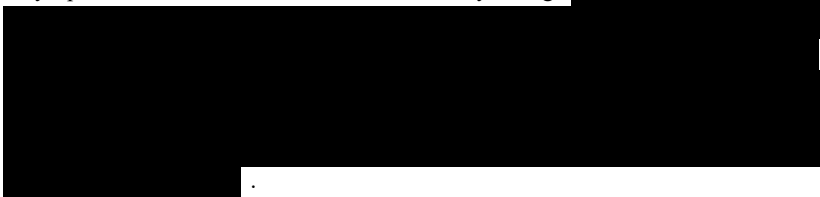
---

### Forcing the CCS Debugger to Step through Each Instruction

Forcing the debugger to step through every program instruction involves a setting which is completely dependant upon the compiler optimization.

Built into CCS is the MSP430 C/C++ compiler. It's used for converting our program from C or C++ into object code, also known as machine language, a language which the microcontroller can understand. The compiler is also designed so it can be used to optimize our program. For example, it will modify our code to improve its execution speed and reduce the size of it by simplifying loops, rearranging statements and expressions, and allocating variables into registers. But this has a trade-off when debugging a program.

Any optimization made to our code will actually change 



A Texas Instruments document titled as “Debug verses Optimization Trade-Off,” advises us to start at a high level of optimization, and if we are not able to effectively test and remove bugs from our program, then lower the optimization until we can. On the other hand, a seemingly more logical and effective strategy may be to start with no optimization when testing and removing bugs, and then when finished, raise the optimization. Neither of these strategies have any supporting evidence.

The compiler has optimization levels which range from zero to 4, with zero being the least amount. It can also be turned completely off. For a new development project, the compiler is automatically set to zero.

We have a choice of changing the optimization level for the entire development project or for an individual file of program code.

➔ **To change the compiler optimization for an entire project:**

1. [Redacted]
2. [Redacted]
3. [Redacted]
4. [Redacted] n

➔ **To the change compiler optimization for an individual file of code:**

1. [Redacted]
2. [Redacted]
3. [Redacted]
4. [Redacted]
5. [Redacted]

If you want to learn more about this topic, see the [Redacted]

2.

### Configuring the Variables View for a Different Numbering Format

While using debug mode for testing and removing bugs from our program code, we'll be watching the contents of storage variables and registers. We may want to view those contents in a specific type of numbering format. For example, we may want to view the data in the decimal, hexadecimal, or the binary numbering format.

If the data is of type float, double, or long double, and you want to see all parts of the number, meaning, the integer, decimal point, and the mantissa, then select the Default number format.

➔ **To change the displayed numbering format for debug mode:**

1. [Redacted]
2. If the *Variables View window* is not open, then do the following.
  - A. From the [Redacted]
  - B. Point to [Redacted]
3. At the top of the [Redacted] Click on the **triangle** to open a *pop-up menu*.
4. Point to [Redacted], and then select the format which you desire.

## How to Write into a Register

Writing into a register, manipulating the bits in it, and bitwise manipulation all mean the same thing. It is a program instruction that changes the binary value of a bit in one or more register bitfields.

Available to us are two methods for manipulating bits. One is called Direct Memory Access, and the other is called Symbolic Memory Access. The first one uses complicated code, pointers, and requires that we know and keep track of register addresses. It also does not allow us to easily develop firmware that can be ported to different models of the MSP430.

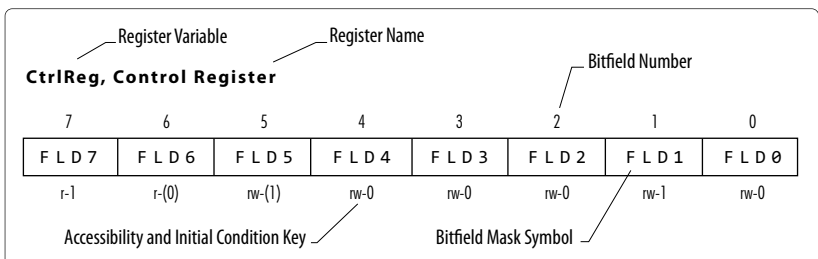
The symbolic method is the easy way to manipulate bits in a register, and it's highly recommended by Texas Instruments, since they put a lot of effort into producing programming tools, source code, and documentation which leverages the use of the symbolic method.

Setting a bit means to write a binary value of 1 into a bitfield. Clearing a bit means to write a binary value of 0 into a bitfield. Toggling a bit means to flip the field from 0 to 1, or from 1 to 0. These are the three programming operations we use for manipulating bits in a register.

### Our Model Register

To help explain the concepts used for writing into a register, we'll use a model register table. It is based upon the conventional register table, so it has a register variable, a register name, bitfields which are numbered, masks for each field, and under each field is a key that explains its accessibility and initial condition.

**Diagram 15:** The model register that will be used by this chapter to explain how to develop a programing instruction which can write into a register.



The eight hypothetical masks for this model register are defined as follows. If it were a real register, these masks would be defined in the microcontroller's header file.

**Code Example 2:** The eight bitfield masks for our model register. This how they would be defined in the microcontroller's header file. Each mask is #defined as a preprocessor directive symbolic constant having a single specific value. For example, the mask FLD0 is defined as the constant numerical value of 0x01, which in binary notation is 1.

---

```

1 #define FLD0 (0x01) //binary mask 0000 0001
2 #define FLD1 (0x02) //binary mask 0000 0010
3 #define FLD2 (0x04) //binary mask 0000 0100
4 #define FLD3 (0x08) //binary mask 0000 1000
5 #define FLD4 (0x10) //binary mask 0001 0000
6 #define FLD5 (0x20) //binary mask 0010 0000
7 #define FLD6 (0x40) //binary mask 0100 0000
8 #define FLD7 (0x80) //binary mask 1000 0000

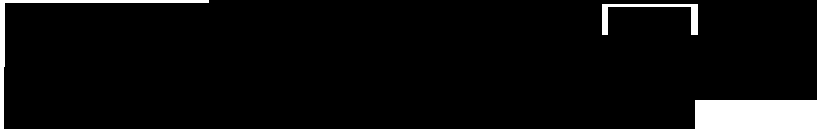
```

---

## Masking Concepts

Registers can be either eight or sixteen bits wide. Eight and sixteen bit binary numbers are used for indicating a specific place or bitfield in such registers. For example, the eight bit number 10000000 can be used for indicating the seventh bitfield in an eight bit register, and the four bit number 1000 can be used for indicating the third bitfield. Keep in mind that the first bitfield in a register is numerically labeled as 0 (zero), and the sequence always begins at the right side of the register diagram. A binary number is used for identifying one or more places in a register, and that number is called a bitfield mask.

In the C language, a



The typical microcontroller header file also defines a set of



Setting, clearing, and toggling a register bit are done with Boolean algebra operations. It is a simple form of math that we need not worry too much about. We only need to focus on three matters: selecting the register variable, selecting an appropriate mask, and selecting an operator that will set, clear, or toggle the bit.

From a mathematical point of view, the variable and mask are operands, and the Boolean operation is the operator. The operator is a C language identifier that represents and carries out a specific type of Boolean operation.

To visually understand how a bitwise operation works, the operands and operator are setup like a simple arithmetic problem. One operand is put on top of the other with

the operator placed to the left. The top operand is the register variable, and the bottom operand is the bitfield mask. The mask tells the operator which field to manipulate. For now, it doesn't matter which operation we use, so the operator is represented as a box. It could be any operator that sets, clears, or toggles a register field.

**Diagram 16:** To fundamentally understand how a bitwise Boolean operation is used for writing into a register, we set up the operation like a simple arithmetic problem.



Now let's say that we want to manipulate the bit in field 3 of our model register. To do that, we need the register variable and a mask which indicates field 3 as containing the bit we want to manipulate. Such a mask must represent the binary number 1000 (remember that the first field is 0). Our model register table conveniently gives us the variable and the symbolic constant, FLD3, as the mask. After we perform the operation, the resulting register bits will appear below the line.

**Diagram 17:** To setup the operation so it will write a bit into field 3 of our model register, we use the register variable (CtrlReg) and the mask for field 3 (FLD3).



This is how the mask works.



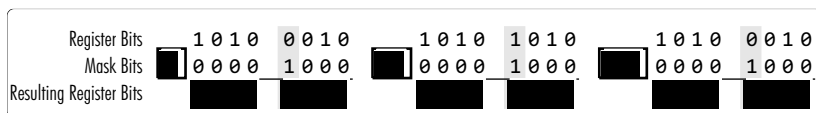
**Diagram 18:** Now we exchange the register variable with the actual contents in the register, since that is what the variable represents, and we exchange the bitfield mask with the constant binary number it represents.



As you now might see, a mask can be used for manipulating more than one field at a time just by having the digit 1 placed in one or more positions in the mask.

Let's now use this same register variable and mask to set, clear, and toggle a bit in field 3. The diagram below shows each operation. The operand at the top represents the contents of the register, as provided by the register variable. Below the register contents is a mask that represents the bitfield we want to manipulate. The 1 bit in the mask indicates which field to manipulate. After the operation is performed, we can see the resulting register bits below the line. Only the field that is identified by the mask is manipulated.

**Diagram 19:** Now we place text into the empty operator box to show which operation to carry out. The example has been duplicated three times in order to represent the three possible writing operations. The result of the operation is shown below each line, and it represents the contents of the register. So from left to right, a bitfield register is set, cleared, and toggled.





The setting operation writes a 1 digit to field 3, the clearing operation writes a 0 digit to field 3, and the toggling operation flips the contents of the field. For the toggling operation, only a single toggle is shown from 0 to 1, but if we toggle the bit again, it'll go back to 0.

If there had been a 1 in any other place in the mask, those corresponding fields would also be operated upon.

## Overview of the Setting, Clearing, and Toggling Operations

By using our model register and the C programming language, the syntax for setting, clearing, and toggling instructions are all shown here together. Three of the examples will manipulate a single bit in a register, while the other three will simultaneously manipulate multiple bits. The actual bitwise Boolean operators, as provided by the C language, are used in the examples. The word bitwise means the operator will operate on the operands bit by bit.

The syntax for writing into a register can be written in short form or long form. The examples shown here are in the long form and short form. The following sections, which explain how to write these instructions, will describe both forms.

**Table 2:** The **long form** of all three operations as used for manipulating bits in a register. Each operation is shown with two examples. One manipulates a single field, and the other simultaneously manipulates three fields. Also shown is the name of the operation as known in the C programming language.

Operation	Bitwise C Operation	Example Code in Long Form
Setting a Single Bit	Bitwise OR	<pre>REG  = 1 &lt;&lt; 3;</pre>
Setting Multiple Bits	Bitwise OR	<pre>REG  = 0x00000007;</pre>
Clearing a Single Bit	Bitwise NOT-AND	<pre>REG &amp;= ~1 &lt;&lt; 3;</pre>
Clearing Multiple Bits	Bitwise NOT-AND	<pre>REG &amp;= ~0x00000007;</pre>
Toggling a Single Bit	Bitwise XOR	<pre>REG ^= 1 &lt;&lt; 3;</pre>
Toggling Multiple Bits	Bitwise XOR	<pre>REG ^= 0x00000007;</pre>

**Table 3:** The **short form** of all three operations as used for manipulating bits in a register. Each operation is shown with two examples. One manipulates a single field, and the other simultaneously manipulates three fields.

Operation	Bitwise C Operation	Example Code in Short Form
Setting a Single Bit	Bitwise OR Assignment	<pre>REG  = 1 &lt;&lt; 3;</pre>
Setting Multiple Bits	Bitwise OR Assignment	<pre>REG  = 0x00000007;</pre>
Clearing a Single Bit	Bitwise NOT-AND Assignment	<pre>REG &amp;= ~1 &lt;&lt; 3;</pre>
Clearing Multiple Bits	Bitwise NOT-AND Assignment	<pre>REG &amp;= ~0x00000007;</pre>
Toggling a Single Bit	Bitwise XOR Assignment	<pre>REG ^= 1 &lt;&lt; 3;</pre>
Toggling Multiple Bits	Bitwise XOR Assignment	<pre>REG ^= 0x00000007;</pre>

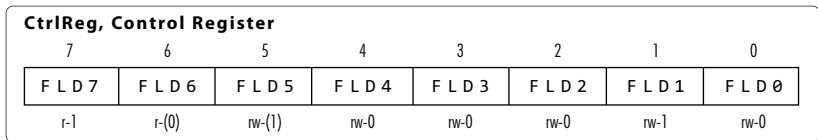
## Setting Bits in a Register

To set a bit means to write a 1 digit into a register bitfield. The operation for setting a bit is called a Boolean Bitwise Inclusive OR. In the C programming language, the symbol for this operation is the vertical bar ( | ).

### Setting a Single Bit


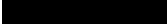
To set a single bit we need a register variable, an appropriate bitfield mask, the bitwise inclusive OR operator, and an assignment operator. Examples shown here will set a bit in field 3 of our model register. Therefore, the register variable will be `CtrlReg`, and the mask will be `FLD3`.

**Diagram 20:** Our model register.





These are the long and short syntax forms which will set a single bit in a register.

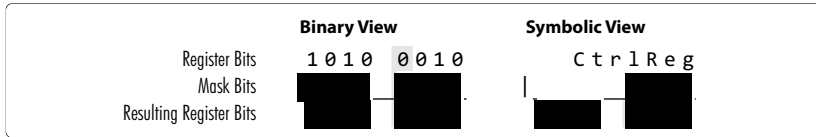
**Code Example 3:** Using the long-form and short-form to set Field 3.

```
1  //Long-form. Using the OR operation
2  //Short-Form. Using OR-Assignment operator
```

The register variable gives us the contents of the register (`10100010`), and the mask (`00001000`) tells the operator which field to manipulate. The bitwise inclusive OR (|) operation compares each register bitfield with their correlating place in the mask. Places in the mask which contain a 1 will set the bit in the correlating bitfield to 1, and places in the mask which contain a 0 will not affect the contents of the correlating bitfield.

Line 1 expresses the instruction in long form. 

Line 2 expresses the instruction in short form. 

**Diagram 21:** Using the OR Operation to set a bit in field 3.

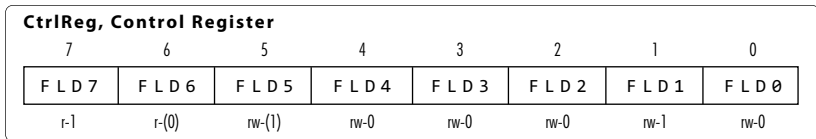
## Combining Masks to Create a Single Mask

Whenever more than a single register bit is being manipulated, we must combine their masks to form a single mask. That mask then can be operated upon by a bitwise operator.

To create a single mask of two or more masks, an addition operator is placed in between every mask, and then that sum is placed inside of parenthesis. For example, to combine the masks FLD3, FLD2, and FLD1 into a single mask, the masks are added together, and then placed in parenthesis (FLD3 + FLD2 + FLD1). By using our model register, the sum of those masks will create a single mask that expresses the binary number 1110.

## Setting Multiple Bits

To simultaneously set multiple bits, we need a register variable, a single mask for the bitfields we need to set, the inclusive OR operator, and an assignment operator. Examples shown here will set bits in fields 3, 2, and 1 of our model register. Therefore, the register variable is CtrlReg, and the masks are FLD3, FLD2, and FLD1.

**Diagram 22:** Our model register.

These are the long and short syntax forms which will simultaneously set multiple bits in a single register.

**Code Example 4:** Using the long and short-forms to set Fields 3, 2, and 1.

```
1 [REDACTED] //Long-Form sets fields 1, 2, and 3.
2 [REDACTED] //Short-Form using OR-Assignment operator.
```

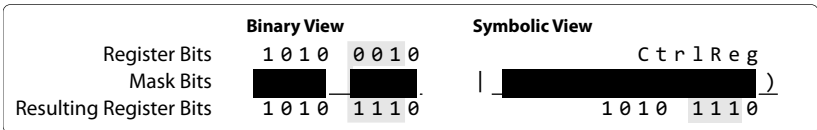
The register variable gives us the contents of the register (10100010), and the mask (00001110) tells the operator which fields to manipulate. We combined masks FLD3, FLD2, and FLD1 to create a single mask whose sum is 1110.

The bitwise inclusive OR operation (`|`) compares each register bitfield with their correlating place in the mask. Places in the mask which contain a 1 will set the bit in the correlating bitfield to 1, and places in the mask which contain a 0 will not affect the contents of the correlating bitfield.

Line 1 expresses the instruction in long form.

Line 2 expresses the instruction in short form.

**Diagram 23:** Using the OR Operation for setting a bit in fields 1, 2, and 3.



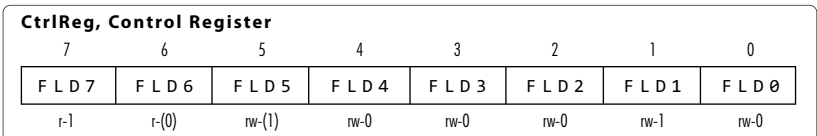
## Clearing Bits in a Register

To clear a bit means to write a 0 into a register bitfield. The operation for clearing a bitfield involves two operations. One is called the Bitwise NOT operation, and it is denoted as a tilde (~). The other is called the Bitwise AND operation and is denoted with an ampersand (&). It will be called a NOT-AND operation that clears a field to zero.

### Clearing a Single Bit

To clear a single bit we need a register variable, an appropriate bitfield mask, the bitwise NOT operator, the bitwise AND operator, and an assignment operator. Examples shown here will clear a bit in field 7 of our model register. Therefore, the register variable will be CtrlReg, and the mask will be FLD7.

**Diagram 24:** Our model register.



These are the long and short syntax forms which will clear a single bit in a register.

**Code Example 5:** Using the NOT-AND operator to clear Field 7.

```
1 [redacted] //Long-Form using the NOT-AND operation.
2 [redacted] //Short-Form using the AND-Assignment operator.
```

The operating sequence goes like this. The bitwise NOT (~) inverts each bit in the mask from 10000000 to 01111111. Next, the bitwise AND (&) compares each field in the register with each place in the inverted mask. Places in the mask which have a 1 will not affect the contents of the field, but places in the mask which have a 0 will

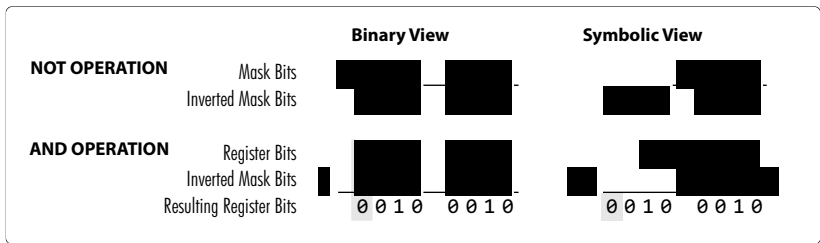
clear the correlating field to zero. The result is assigned (written) back to the register variable to immediately update the contents of the register.

Line 1 expresses the instruction in long form. [Redacted]

Line 2 expresses the instruction in short form. [Redacted]

The diagram shows the two step operation needed for clearing Field 7. Notice that the NOT operation is a unary operation, meaning, it operates on just a single operand. The AND operator works on two operands, the register variable and inverted mask.

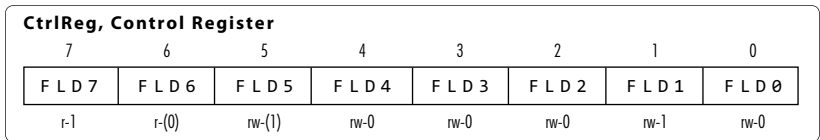
**Diagram 25:** Using the NOT-AND Operations for clearing field 7 to 0.



### Clearing Multiple Bits

To simultaneously clear multiple bits, we need a register variable, a single mask for the bitfields we need to clear, the bitwise NOT operator, the bitwise AND operator, and an assignment operator. Examples shown here will clear bits in fields 7, 5, and 1 of our model register. Therefore, the register variable is CtrlReg, and the masks are FLD7, FLD5, and FLD1.

**Diagram 26:** Our model register.



These are the long and short syntax forms which will clear multiple bits in a register.

**Code Example 6:** Using the NOT-AND operator to clear Fields 7, 5, and 1 to zero.

```

1 [Redacted] //Long-Form.
2 [Redacted] //Short-Form.

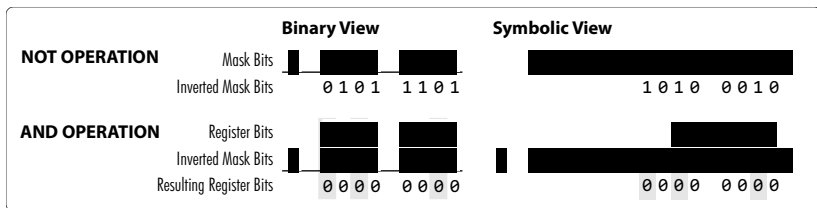
```

The register variable gives us the contents of the register (10100010), and the combined mask (10100010) tells the operator which fields to manipulate. We combined masks FLD7, FLD5, and FLD1 to create a single mask whose sum is 10100010.

Line 1 expresses the instruction in long form.

Line 2 expresses the instruction in short form.

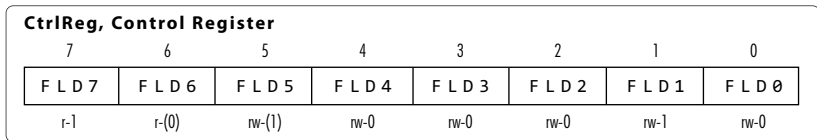
**Diagram 27:** Using the NOT-AND operators to clear Fields 7, 5, and 1 to zero. According to the C language precedence of operations, the NOT operation is performed first, and then it's followed by the AND operation that clears the fields.



### Simultaneously Setting and Clearing Bits in a Register

To simultaneously set and clear bits in a register means to use a single instruction for setting and clearing bits in a register. We use the same operators, but the syntax is different.

**Diagram 28:** Our model register.



By using our model register, two examples are shown here. The first example will clear a bit in field 7 and set a bit in field 6. The second example will clear bitfields 7, 5, and 1, and it will set bitfields 6, 4, 3, 2, and 0.

**Code Example 7:** The first instruction clears a single bit and sets a single bit. The second instruction clears three bits and sets five bits.

```
1 [redacted];
2 [redacted];
```

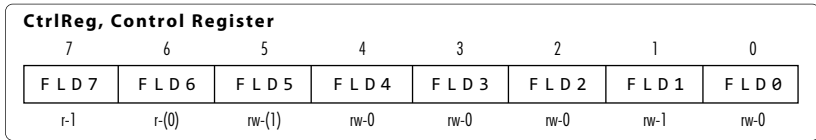
## Toggling Bits in a Register

To toggle a bitfield means to flip the bit so it changes from 0 to 1 or from 1 to 0, depending on which bit is already in the field. Toggling uses the Bitwise Exclusive OR operation (XOR). The programming symbol for XOR is the caret (^).

### Toggling a Single Bit

To toggle a single bit we need a register variable, an appropriate bitfield mask, the bitwise exclusive OR operator, and an assignment operator. Examples shown here will toggle a bit in field 3 of our model register. Therefore, the register variable will be `CtrlReg`, and the mask will be `FLD3`.

**Diagram 29:** Our model register.



These are the long and short syntax forms which will set a single bit in a register.

**Code Example 8:** Using the long and short forms for toggling field 3.

```

1 CtrlReg |= FLD3; //Long-Form using the exclusive OR operator.
2 CtrlReg ^= FLD3; //Long-Form. Another toggle.
3 CtrlReg ^= FLD3; //Short-Form using exclusive OR-Assignment operator.

```

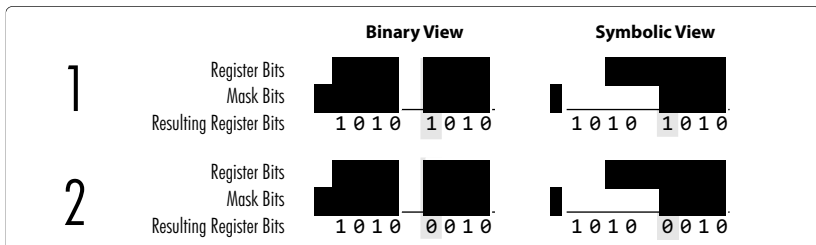
The register variable gives us the contents of the register (10100010), and the mask (00001000) tells the operator which field to manipulate.

The bitwise XOR operation compares each register field with their correlating place in the mask. Places in the mask which have a 1 will toggle the correlating field. Places in the mask which have a 0 will not affect the contents of the field.

Line 1 shows the operation with two operands separated by the



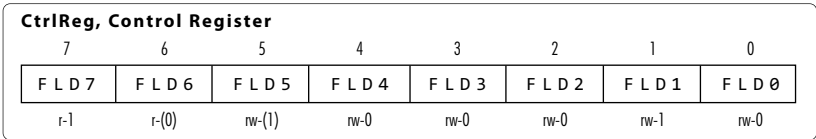
**Diagram 30:** Toggling the bit in field 3. The first operation toggles the field to 1, and the second operation toggles it back to 0.



## Toggle Multiple Bits

To toggle multiple bits we need a register variable, a single mask that combines all the bitfield masks we need to toggle, the bitwise exclusive OR operator, and an assignment operator. Examples shown here will toggle the bits in fields 3, 2, and 1 of our model register. Therefore, the register variable will be `CtrlReg`, and the mask will combine `FLD3`, `FLD2`, and `FLD1`.

**Diagram 31:** Our model register.



These are the long and short syntax forms which will toggle three fields in a register.

**Code Example 9:** Using the long and short forms for toggling field 3, 2, and 1.

```
1 CtrlReg |= 0x1110; //Long-Form.
2 CtrlReg ^= 0x1110; //Short-Form.
```

The register variable gives us the contents of the register (`10100010`), and the combined mask (`00001110`) tells the operator which fields to toggle. We combined masks `FLD3`, `FLD2`, and `FLD1` to create a single mask whose sum is `1110`.

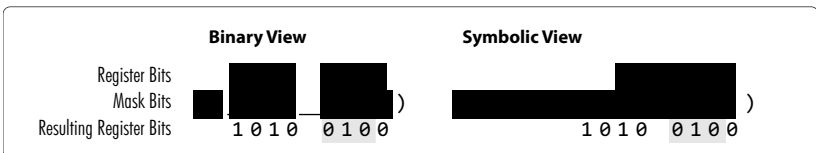
The bitwise XOR operation compares each register field with their correlating place in the combined mask. Places in the mask that have a 1 will toggle the correlating field. Places in the mask that have a 0 will not affect the contents of the field.

Line 1 shows the operation with an XOR operator separating the register variable and the combined mask.

Line 2 is the short form of the instruction. It uses

Here is how the operations appear in binary and symbolic views.

**Diagram 32:** Toggling fields 3, 2, and 1.



## Just Simply Writing a Number into a Register

All the writing methods which have been described will ultimately assign a number to a register which sets and clears its bitfields to our desired states. So by now, you might be wondering "why do we have to write a complicated instruction which uses



several operators and operands in order to write a number into a register?" Why don't we skip all that work and just assign an appropriate number to the register which will simultaneously set and clear all the register bitfields?

Well, sometimes during the execution of a program we do not know in advance which number must be written because some bitfields must be changed while others or not. So that's the main reason.

But there are scenarios where we know which state we want the entire register to be placed into. One of those scenarios is when the program is configuring or initializing module registers before they will be used. That work is typically done at the beginning of a program.

So for such scenarios, we often just use a single number that will properly set and clear the bitfields all at one time. The number can be written in the binary, hexadecimal, or decimal format. The MSP430 C compiler will automatically convert the number into binary format, and then build the program with that format.

**Code Example 10:** Assigning a number to an 8-bit register so it will clear fields 7, 5, 3, 1 and set fields 6, 4, 2, and 0. It can be written in the binary, hexadecimal, or decimal format.

---

```

1 // All three instructions write the same binary number into CtrlReg.
2 // Using the binary format.
3 // Using the hexadecimal format.
4 // Using the decimal format.
```

---

The one advantage the binary format has over the other two is that we can explicitly see which fields will be set and cleared.

A typical desktop computer operating system, such as Microsoft Windows, will include a calculator which can be used as a tool for converting a number from one format to another.

---

## Writing into Password Protected Registers

Manipulating bits in a password protected register is handled with methods which are similar to those described earlier in this chapter, but there are some matters which we must take into account with such registers. That topic is presented on page 176. This section briefly describes an alternative method for writing into a password protected register that simplifies the instruction by using the OR operator (|).

Writing into such a register involves an instruction which simultaneously unlocks the register and sets a bit in some other field which configures a system to behave in some way. For example, the instruction for putting the watchdog on hold is written that way. If additional fields must be set, then use the OR operator to include them.

**Code Example 11:** Using the OR operator in a password protected register.

```
████████████████████; // Typical instruction for stopping the watchdog.
```

WDTCTL is the watchdog timer control register. WDT PW is the bitfield mask for the password, and WDT HOLD is the mask for putting the watchdog on hold. For detailed information about the watchdog, see page 89.



## How to Declare a Storage Variable

A variable is used for holding data which changes. But unlike the software development scenario, MSP430 firmware development with variables must be handled differently. First we'll look at a little history about it, and then we'll learn how to declare them in firmware.

---

### A Description for the Storage Variable

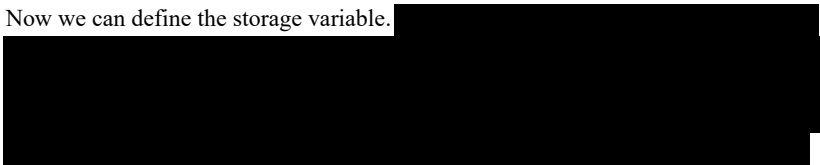
The international standard for the C Language does not define the word *variable*, nor does it define the compound form (portmanteau) of the words *storage variable*. The standard uses the word *variable* in just a few places, and its definition is only implied. Although this is a digression, a concrete definition of those words is made here in order to be thorough.

From the most abstract point of view, all the grammatical elements in the C language, which provide us the means for writing instructions to form a computer program, are called objects. This is not to be confused with objects of the C++ language which are created from classes.

The standard for C defines an object as a region of data storage in the execution environment, the contents of which can represent values. In our work, the environment is a firmware image which is loaded into main memory, and the region of data storage is the main memory address space.

Once again, from an abstract point of view, what is stored at an address may or may not represent a numerical value. If it does represent such a value, then what is stored at the address is a type of operand. If what is stored at the address does not represent a numerical value, then it represents a type of operator.

Now we can define the storage variable.



A *type* is a specific category of operator or operand. The different types of operators are used for performing different types of operations on operands. For example, there are types of operations which perform relations, bitwise Boolean, assignment, addition, subtraction, multiplication, and so on. As for operands, they are data in the form of types of numerical values. For example, there are integer, real floating, Boolean, and character types of numerical data.

So what we have is a hierarchy of abstractions that define a storage variable. An object is a storage location in memory. The content may or may not represent a

numerical value. If the content does represent a value, the content is an operand. If the content does not represent a value, the content is an operator. An operator is immutable, while an operand is either mutable or immutable. And finally, a type of operand which is mutable is called a variable or storage variable.

---

## Declaring Storage Variables

A storage variable is declared with one or more data type qualifiers, also referred to as specifiers. For example, a variable that is declared as an integer can be further specified as an unsigned integer, or further specified as a long unsigned integer.

In MSP430 firmware, storage variables must be specified as storing volatile type of data. So for example, a variable that is specified as storing an integer must be further specified as a `volatile`.

**Code Example 12:** To declare a storage variable, we must specify it as `volatile`. Here, the storage variable `x` is specified as `volatile int`.

---

```
1 volatile int x = 0x00; //A storage variable specified as type volatile int
```

---

Here is the rationale.

To mitigate code optimizations which involve variables, we must `volatile`

## How to Read a Register

### The Process

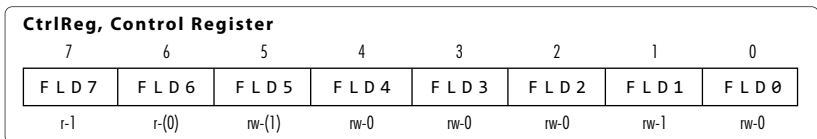
To read a register means to assign its contents to a storage variable or just use its contents as an operand (expression) in an instruction without an assignment. By using register variables and masks, we can read the entire register or specific bitfields in a register.

To read a protected register, meaning, a register which incorporates a password, a different technique is used. The watchdog timer and PMM registers are of that type. That technique is introduced by “Reading the Watchdog Timer Register,” on page 96, and by another example shown at the end of the last chapter.

When making an assignment, the width of the storage variable must match the width of the register. For example, declare the variable as a `uint8_t` for eight bit registers, or declare it as an `uint16_t` for sixteen bit registers. Furthermore, both types must be declared as `volatile` because registers are typically used for storing positive binary numbers. If the register is expected to handle signed numbers, then be aware that a bitfield is reserved for that sign, so your code must take that into account with a signed type of data declaration.

When more than one bitfield must be read,

**Diagram 33:** Our model register. The code examples shown below will read bits from our model register.



**Code Example 13:** The AND operator (&) is used for reading fields in a register.

```


1 uint8_t result; // Declaring our storage variable.
2
3 uint8_t reg = 0x00; // Reading a single field
4 uint8_t reg = 0x00 & 0x0F; // Reading two fields
5 uint8_t reg = 0x00 & 0x00FF; // Reading multiple fields
6 uint8_t reg = 0x00; // Reading the entire register

```

### The Code

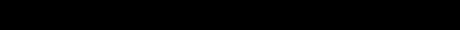
Our model register is eight bits wide, so the storage variable is declared as `uint8_t` with the symbol `result` as its identifier. And the variable is initial-

ized to zero ( $0x00$ ). Alternatively, the number zero could have been in the decimal or binary format. See page 6 for information about the notation for those formats.

The first instruction reads a single bitfield. 


).

The operator compares each place in the variable with its bitwise corresponding place in the mask. Places in the mask which have a zero will tell the operator to not read the corresponding place in the variable. Places in the mask which have a 1 will tell the operator to read the corresponding place in the variable. Places which are not read will result in a 0. The *entire eight bit result* is then assigned to the storage variable, in this case, it is named `result`.


You must be very careful about how 




It reads the value in `FLD3` and compares it to the standard bit `BIT3`. What is being compared is the result of the reading operation, which will be either  $0x0$  or  $0x4$ , to the standard bit `BIT3`, which is  $0x4$ . If they are equal the resulting decision is true, then the block of code inside of the curly brackets will be executed.

On line 4, the second instruction reads two bitfields. 



The third instruction, and line 5, 



While running this code in debug mode, and viewing the values assigned to the storage variable, the numerical notation might be in a format which is not convenient for you. For example, it might be presented in hexadecimal notation. For instructions about how to change the format, see 



## Background for Testing the Contents of a Register

Before the discussion about testing the contents of a register can be presented, we digress, yet again, with a review about integer constants, relaxed compilers, and using binary notation in our code.

---


### Integer Constants



Binary notation and testing the contents of a register depend on some understanding of the integer constant, which we have learned earlier on page 75, is an immutable numerical value. We all know about whole numbers. It's a system of numbers that starts at zero and goes to infinity {0, 1, 2, 3,...}. Whole numbers are not negative numbers, nor are they fractions. Integers are whole numbers including their negative values {..., -3, -2, -1, 0, 1, 2, 3,...}.

The C programming language refers to all integers as integer constants. Furthermore, the language allows us to use decimal or hexadecimal notation to express integers; meaning, in our code, we have the option of writing the integer ten as either 10 or as 0xA. It's our choice, and the choice only depends on our preference or need. Keep in mind that both notations will be converted by the compiler into binary notation, since from the microprocessor's point of view, all program code is handled in binary notation.

---

### The MSP430 Relaxed Compiler



A popular compiler is the  Compiler (). It's a relaxed compiler that has been extended so we may use binary notation to express a number in our code. The Texas Instruments Code Composer Studio (CCS) uses GCC, and they have relaxed it even further. In order to make it easier for us to write programs for their microcontrollers, they had to! For example, the language has been extended to include custom made functions and declarations for interrupt service routines (ISRs). The customized functions allow our code to easily access the CPU registers, and ISRs are essential for writing event-driven firmware.



---

## Using Binary Notation

Using binary notation in our code is helpful when it is used for making decisions. It allows us to immediately visualize a numerical value as a pattern of binary bits.

For example,



Such decision making code will typically use an unsigned integer, expressed in decimal or hexadecimal notation, to represent the pattern that will be compared with a pattern in a register or variable. For most of us, the binary representation of a decimal or hexadecimal integer cannot be quickly visualized without some extra mental effort. That makes our code harder to read. Therefore, using binary notation in such coding scenarios will facilitate the quick visualization of the pattern. Especially since the user guides show the contents of a register as a pattern of bits, and not as some decimal or hexadecimal representation of an integer.

Here's the rule for using binary notation in our code:



---

## Enabling CCS Support for GCC Extensions

Unlike older versions, the latest versions of Code Composer Studio will automatically include support for GCC extensions, and it is turned on by default. Therefore, we no longer have to worry about turning it on. But this information is here in case that changes again.

## How to Test the Contents of a Register

A firmware instruction can be used for performing an action, such as driving or monitoring some module, and it can be used for making a decision. An action quite literally means reading and writing into a register. A decision will be in the form of performing a calculation or comparing one piece of data to other pieces of data, and then using the decision to produce a result. Testing the contents of a register is one form of decision making.

### The Process

Testing the contents of a register means to read the register's contents and then compare the contents to an integer.

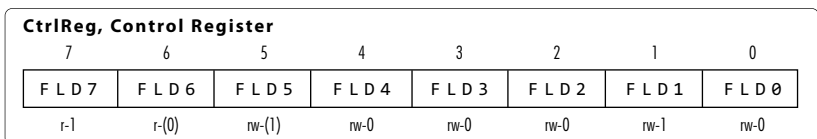
The result of the decision is a `bool`.

### The Code

The comparison is done through some type of selection statement and an equality operator (`==`). A selection statement is typically in the form of an if statement or similar type of selection statement (see “Structures for Program Development” on page 105“ for an elaboration).

By using a `mask`, we can test a specific bitfield in a register. When more than one bitfield is to be tested, the masks for those fields are combined to create a single mask (as described by “Combining Masks to Create a Single Mask” on page 66). If the entire register is to be tested, only the register variable is needed. Code examples shown below will test bits from our model register.

**Diagram 34:** Our model register, as introduced by page 61.



Since our model register is only eight bits wide, the comparison will be made with an eight bit test integer. The test integer is written in binary notation so we can quickly visualize the integer as a pattern of bits. The type of selection statement used in each example is the `if` statement. It makes a Boolean decision that results in a true or false result or 1 or 0 respectively. For the sake of brevity, the instruction block which follows the `if` statement, contains no code. The empty block is denoted with braces `{ }`.

**Code Example 14:** Using a read operation and a decision statement to test the contents of a register. An if statement is used for making the decision. For the sake of brevity, the instruction block following the decision has no code. The empty block is denoted with braces {}.

---

```
1 /** Testing a Single Field *****/
2 ██████████ // If result is true, execute the block.
3 {
4   ██████████
5 }
6
7
8 /** Testing Multiple Fields *****/
9 ██████████ // If true, execute block.
10 {
11   ██████████
12 }
13
14
15 /** Testing the Entire Register ***/
16 ██████████ // If result is true, execute the block.
17 {
18   ██████████
19 }
```

---

## How to use a Pointer to Read and Write into Main Memory

Earlier chapters which are about reading, writing, and testing the contents of a register clearly say that a pointer variable is not needed to do that work. So their code examples do not use pointers nor pointer variables. Instead of pointers, a register variable is used for accessing the register.

The register variable is just simply an `int` of an actual pointer variable. `int` meaning, we would have to know and use the actual address number to a register. And that would make our development work more complicated.

So why do we need to learn about using a pointer variable and the pointer which it stores? There are a few reasons.

---

### Pointer and Pointer Variable

A pointer is just simply a type of number. It is a type which represents an address number in main memory. A pointer variable is just simply a type of storage variable. It is a type of variable used for storing a pointer.

A pointer and a pointer variable are typically not useful by themselves because we are concerned with the data stored at the address, not the address number itself. A pointer or pointer variable are combined with a special type of operator that will access the data at the address. It is called the indirection operator.

Generally speaking, a pointer and pointer variable enable us to develop programs which can access data at a specific address in memory and to create data structures which can grow and shrink in memory. We are interested in their ability to access data in main memory, since that is the only method available for accessing data at a specific address.

---

### Indirection Operator

A special type of operator is used for working on pointers and pointer variables. It is called the indirection operator (`*`), and it has three purposes.

The first purpose is used for converting an address number into a pointer. The second purpose is used for declaring a storage variable as a pointer variable.

After a pointer and its pointer variable have been created, the indirection operator's third purpose then comes into play. It is used for accessing the data at a pointer. In other words, it is used for accessing the data at a specific address in memory.

---

### Converting an Address Number into a Pointer: the Pointer Expression

Although a pointer is a type of number which represents an address in memory, it is written and used as an expression. The expression is in the form of a unary operator and a number which represents an address in memory. The expression is then used in an instruction which reads or writes into the address.

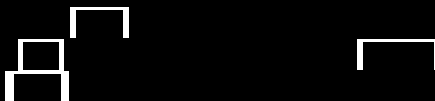
The unary operator is called a `*` and it is used for `*(int*)`. In this case, it is used for converting an address number into a specific type of pointer; meaning, to `*(int*)`.

The amount of data at any address is eight bits, but a pointer can be specified to point to eight, sixteen, thirty-two, or more bits of adjacent data in memory. For example, a pointer to eight bits is specified as a `char`, and a pointer to sixteen bits is specified as an `int`. A pointer to more than eight bits in memory is including bits which are at the next higher addresses in memory. For example, a pointer to address number `0x1A1A` and specified as a sixteen bit pointer is pointing to the bits at `0x1A1A` and the next higher address in memory, `0x1A1B`.

Here is how the pointer expression is written. `*(int*)`

Be aware that this expression is not a declaration. The pointer, which appears as the address-number, cannot be used by itself in other instructions or in other places in the same instruction. The cast is a temporary operation which lasts only during the life of the operation.


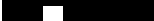
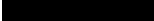
Three types of pointer expressions are shown below. `*(int*)`



**Code Example 15:** Shown are three pointer expressions which use the same address for pointing to different amounts of data in memory. A pointer to more than eight bits in memory is including bits which are at the next higher addresses in memory. The address number is the pointer.

---

```

1  // Expression which points to 8 bits
2  // Expression which points to 16 bits
3  // Expression which points to 32 bits

```

---

## Declaring a Pointer Variable

To declare a pointer variable, we choose an identifier for the variable, then we use an indirection operator to specify it as a variable which stores a pointer, and then we further specify it as storing a pointer which points to a specific amount of data and being of type volatile.

After a pointer variable has been declared, it can be used for reading and writing data into a specific address in memory.


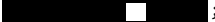
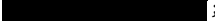
Shown by the following code example are three instructions which declare pointer variables.



**Code Example 16:** Declaring a pointer variable. The pointer variables are aPtr, bPtr, and cPtr. The indirection operator (\*) specifies the variable as one that stores a pointer.

---

```



1  ; // Stores a pointer to 8 bits of memory
2  ; // Stores a pointer to 16 bits of memory
3  ; // Stores a pointer to 32 bits of memory

```

---

The pointer variable is further specified as storing a pointer which points to a specific amount of data in memory.



The final specifier types the variables as being . That is in accordance with all storage variables we declare in a program for the MSP430. The  specifier tells the MSP430 compiler to not include this variable when it uses algorithms to optimize our code which may adversely affect the variable.

## Declaring a Pointer and Assigning it to a Pointer Variable

The example shown here just simply show the pointer expressions developed earlier as being assigned to pointer variables. The variables were also explained by the previous section. Notice that the data type of the variable matches the pointer's type.

Use this type of instruction when you plan to use the pointer in more than one instruction, otherwise, all you will need is the pointer expression by itself.

**Code Example 17:** Declaring and assigning a pointer to a pointer variable.

```

1 [REDACTED]; // 8-bit pointer assigned to 8-bit variable
2 [REDACTED]; // 16-bit pointer assigned to 16-bit variable
3 [REDACTED]; // 32-bit pointer assigned to 32-bit variable

```

## Reading Data

Using a pointer or a pointer variable for reading data from a specific address in memory is like using a register variable to read a register. The same operators and syntax are used. The standard bits can also be used for reading individual bitfields at an address.

There are two matters which distinguishes the method of using a pointer as compared to using a register variable. When using a pointer, we need to know the address number, but when using a register variable, we don't need know it. And using a pointer provides the option of reading from more than one address, while a register variable limits us to just the addresses within the register itself. The main reasons we need to use a pointer is when the program must access data in memory which does not have a dedicated register variable, such as microcontroller calibration data, and to enable the program to create dynamic data structures.

The examples shown here are limited to just reading the entire byte at an address and the adjacent bytes. Refer to code example 13, on page 77, for techniques which can read individual bitfields.

## Using a Pointer

When using a pointer, it means using a pointer expression. To write an instruction which reads data from an address, we need the expression, the indirection operator, and a storage variable where the data will be assigned to.

The pointer expression involves a `*` operator that carries out two operations. It specifies the

At line 1 of the following code example, a byte will be read from address `0x1A1A`, and it will be assigned to the storage variable `a`.

At line 2, two bytes are read.

At line 3,

**Code Example 18:** Using a pointer for reading data from addresses in memory.

```

1 // Reading 8 bits from 0x1A1A
2 // Reading 16 bits fr 0x1A1A and 0x1A1B
3 ); // Reading 32 bits fr 0x1A1A to 0x1A1D

```

### Using a Pointer Variable

Before a pointer variable can be used for reading data from an address, it must have been declared and have a pointer assigned to it. That is shown earlier by code example 17.

To use a pointer variable for reading data at an address in memory, we use a pointer variable and an indirection operator to access the data, and then we assign the data to a storage variable.

**Code Example 19:** Using a pointer variable for reading data from an address in memory.

```

1 // Reading 8 bits from 0x1A1A
2 // Reading 16 bits from 0x1A1A and 0x1A1B
3 // Reading 32 bits from 0x1A1A to 0x1A1D

```

### Writing Data

A pointer variable is used for writing into an address in memory. Using a pointer variable is like using a register variable for writing data into a register.

Before a pointer variable can be used for reading data from an address, it must have been declared and have a pointer assigned to it. That is shown earlier by code example 17.

To use a pointer variable for writing data into an address in memory, we use a pointer variable and an indirection operator to access the data, and then we assign the data to the variable. The amount of data which we can write, or assign, to a pointer variable depends on how the variable's data type was declared. In the following code example, they were declared as a char, int, and a long respectively.






The examples only show how to write one, two, or four bytes to a pointer. Therefore, when writing more than one byte, the next byte is written into the next higher address past the pointer.

Since the pointer variable acts like a register variable, we can also use it for writing into specific bitfields. The syntax for those operations are explained by “Overview of the Setting, Clearing, and Toggling Operations” on page 64.

**Code Example 20:** Using a pointer variable for writing bytes of data into addresses in memory. The pointer is 0x1A1A.

---

```

1  // Writing 8 bits to 0x1A1A
2  // Writing 16 bits to 0x1A1A and 0x1A1B
3  // Writing 32 bits from 0x1A1A to 0x1A1D

```

---

## Using a Pointer Macro

A register variable is basically a C language macro. The macro is assigned a unique name, and it is defined by an indirection operator working on a pointer expression. The operator accesses the data at the pointer.

The benefit which a macro provides is a single identifier which can be used as a storage variable. We can read and write into it without the complex pointer notation and syntax.

The following code example shows how to use a pointer to create a macro which can be used for reading and writing into a specific address in main memory.

On line 1,



Since the macro acts like a register variable, we can also use it for writing into specific bitfields. The syntax for those operations are explained by “Overview of the Setting, Clearing, and Toggling Operations” on page 64.

**Code Example 21:** Creating a pointer macro that acts like a register variable.

---

```

1  // Defining the macro REG
2  // Reading a byte from address 0x1A1A
3  // Writing a byte to address 0x1A1A

```


---


## Watchdog Timer and Putting it on Hold

A firmware project will typically start with disabling the watchdog timer module so we may focus on developing code without constantly having our program interrupted by a watchdog timer overflow. Generally speaking, after the primary firmware code has been completed, we then develop instructions that configure the watchdog. A properly configured watchdog is important for commercially sold products which depend on microcontrollers. Unfortunately, a thorough explanation about watchdog strategies are beyond the scope of this book. This chapter will focus on the watchdog's purpose, its basic operation, and how to place it on hold. It is typically placed on hold until the end of program development when we can make decisions on where are the best places in the program to reset the watchdog's interval timer.

---


### Purpose

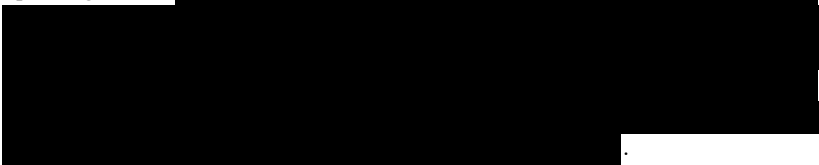
The primary purpose of the watchdog is to 



---

### Basic Operation

The watchdog basically operates as an interval timer. It can be put into three different operating modes: 



When the microcontroller is in an off state, a successful power-up involves the sequential execution of the BOR, POR, and PUC reset subsystems. If the module is in watchdog mode, a flag will cause a reset. Meaning, the microcontroller will restart at the Power-Up Clear (PUC) subsystem. A restart will hopefully clear the cause of the CPU crash.

If the watchdog is in timer mode, and interrupts are enabled (the General Interrupt Enable (GIE) bitfield in the CPU Status Register is set), a flag will cause the interrupt system to direct the CPU to execute a specific ISR. An ISR is used for carrying out a specific process (set of instructions). However, the conventional timer module, not the watchdog timer module, is the most appropriate module used for carrying out that type of ISR.

Regardless of which mode the watchdog is in, an interrupt flag will immediately clear the counter back to zero.

---

### Interval Reset Instruction

When in the watchdog mode, an interval reset instruction is used as a threshold to a timer overflow event.

A firmware program will typically have more than one instruction for clearing the timer interval counter. The placement and location (meaning, the program line number) of each reset instruction is a strategic decision which is based on the length of the interval.

The interval length is configured through a bitfield in a watchdog timer register. An instruction is used for choosing an interval length from a set of lengths. You cannot choose a length which is different from what the given set offers.

Longer intervals will

Deciding on an interval length (the number of clock cycles) is based on the number of instructions that will be executed during an interval. Depending on the clock frequency, the interval may range from 1 to 375,000 cycles.

In the assembly language, a single instruction takes from three to six cycles to execute. But a single C language instruction is an abstraction for many assembly instructions, so it will be longer.

For information about writing an interval reset instruction, see “Watchdog Timer Handlers” on page 178.

---

### Watchdog Control Register Table

Since we’ll be writing into the watchdog’s control register to disable it, we need to learn about its register variable and which bitfield masks are needed for disabling it. Shown below, by diagram 35, is a view of the typical table for that register.

Not shown are

On the left side of the table are a column numbers that will not appear with a real table. It’s used as a cross-reference between the explanations in this section and the lines in the table.

For disabling the watchdog, we are only concerned with the register variable (WDTCTL), the password mask (WDTPW), and the watchdog timer hold (WDTHOLD) mask. However, short explanations about the other fields are provided here for informational purposes. Keep in mind that these masks are the actual identifiers as would be used in real firmware code.

**Diagram 35:** A view of the typical watchdog timer control register. The column of line numbers are not part of the table. They are used here to help explain the table.


1	<i>WDTCTL, Watchdog Timer+ Register</i>							
2	15	14	13	12	11	10	9	8
3	<b>WDTPW</b>							
5	7	6	5	4	3	2	1	0
6	<b>WDTHOLD</b>	<b>WDTNMIES</b>	<b>WDTNMI</b>	<b>WDTTMSSEL</b>	<b>WDTCNTCL</b>	<b>WDTSSSEL</b>	<b>WDTISx</b>	
7	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r0(w)	rw-(0)	rw-(0)	rw-(0)
8	<b>WDTPW</b>	Bits 15-8	Watchdog timer+ password. Always read as 069h. Must be written as 05Ah, or a PUC is generated.					
9	<b>WDTHOLD</b>	Bit 7	Watchdog timer+ hold. This bit stops the watchdog timer+. Setting WDTHOLD = 1 when the WDT+ is not in use conserves power.					
10			0 Watchdog timer+ is not stopped					
11			1 Watchdog timer+ is stopped					
12	<b>WDTNMIES</b>	Bit 6	Watchdog timer+ NMI edge select. This bit selects the interrupt edge for the NMI interrupt when WDTNMI = 1. Modifying this bit can trigger an NMI. Modify this bit when WDTIE = 0 to avoid triggering an accidental NMI.					
13			0 NMI on rising edge					
14			1 NMI on falling edge					
15	<b>WDTNMI</b>	Bits 5	Watchdog timer+ NMI select. This bit selects the function for the RST/NMI pin.					
16			0 Reset function					
17			1 NMI function					
18	<b>WDTTMSSEL</b>	Bit 4	Watchdog timer+ mode select					
19			0 Watchdog mode					
20			1 Interval timer mode					
21	<b>WDTCNTCL</b>	Bit 3	Watchdog timer+ counter clear. Setting WDTCNTCL = 1 clears the count value to 0000h. WDTCNTCL is automatically reset.					
22			0 No action					
23			1 WDTCNT = 0000h					
24	<b>WDTSSSEL</b>	Bit 2	Watchdog timer+ clock source select					
25			0 SMCLK					
26			1 ACLK					
27	<b>WDTISx</b>	Bits 1-0	Watchdog timer+ interval select. These bits select the watchdog timer+ interval to set the WDTIFG flag and/or generate a PUC.					
28			00 Watchdog clock source /32768					
29			01 Watchdog clock source /8192					
30			10 Watchdog clock source /512					
31			11 Watchdog clock source /64					

Line 7 shows the accessibility and initial conditions of the lower half of the register. All the bitfields are labeled as rw-(0), which means

Line 8 is a description for the register's password bitfields (WDTPW). The password is just simply a specific 16-bit binary number and does not change. Whenever an

instruction writes into the register, the instruction must simultaneously include the password mask.

Be careful about



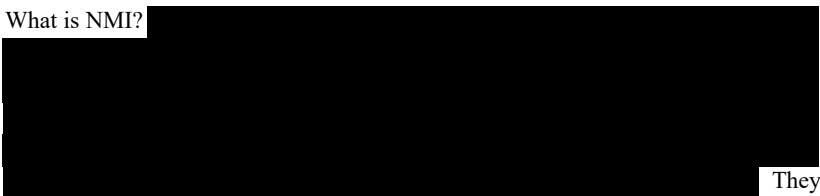
If we ever want to read the upper register's password at Line 3, we'll always get 069h. But reading the lower register at Line 6 will return the actual state of each bitfield. And furthermore, we do not need the password to read the lower, higher, or both registers. Code examples which follow will show how to properly use this mask.

Line 9 begins to describe the Watchdog Timer Hold bitfield (WDTHOLD) used for starting and stopping the watchdog. If we set the bit, the watchdog will be disabled. If cleared, it will be enabled and run.

Line 12 describes the NMI Edge Select bitfield (WDTNMIES). As will be elaborated upon by the next bitfield, if the  $\overline{\text{RST/NMI}}$  Pin is configured to the NMI function, it will sense logical high and low digital voltage states which are sent to the pin from peripheral devices, such as buttons and switches. This bitfield controls whether an interrupt service routine (ISR) is triggered by either a voltage rising from a digitally low to a high state or from a digitally high to a low state. This is conceptually important to take into account when designing circuits which interface with the  $\overline{\text{RST/NMI}}$  pin.

Line 15 describes the NMI Select bitfield (WDTNMI). A feature provided by the watchdog module is an ability to sense digital voltage signals at the  $\overline{\text{RST/NMI}}$  pin that is located on the case of the microcontroller. This bitfield configures the  $\overline{\text{RST/NMI}}$  pin to be used for restarting the microcontroller or for triggering a non-maskable interrupt (NMI) service routine. If the pin is configured to the reset function, an external peripheral device, such as a button, can be used for manually restarting the microcontroller. If the pin is configured to the NMI function, an external peripheral device can be used for triggering a specific ISR.

What is NMI?



They

come from the  $\overline{\text{RST}}/\text{NMI}$  pin, from the clock module, and from the memory module. Maskable interrupts, which are of many types, are general purpose in nature. They all can be blocked by disabling interruptions (setting the General Interrupt Enable (GIE) bitfield in the CPU's status register). Be aware that to temporarily disable interrupts is an important action to occur while configuring the registers for some peripheral modules and systems.

Line 18 describes the Watchdog Timer Mode Select mask (WDTMSEL). This field controls the result of a timer interval overflow. If the field is cleared, an overflow will trigger a restart. If the field is set, an overflow will set a flag that triggers an ISR.

Line 21 describes the Timer Counter Clear bitfield (WDCNTCL). This bitfield is used for clearing the timer counter to zero. When using the watchdog for monitoring CPU crashes, this is the field that our firmware will use

Line 24 describes the Timer Clock Source Select bitfield (WDTSSSEL). Use this bitfield for

Line 27 describes the Timer Interval Select bitfield (WDTISx). The actual masks as defined by the header file are WDTIS0 and WDTID1 which are symbolic constants for 0x01 and 0x02 respectively. Use these bitfields for

The format used for showing us the number of cycles can be confusing. Here is how you interpret

The rationale for using that format is so we can quickly calculate the length of time for the interval. For example, if the watchdog is driven at 16 MHz, then the calculation is  $[(16\text{MHz})/(32768\text{cycles})]= 488.3\text{ms}$ , and if it is driven at 1 MHz, then the calculation is  $[(1\text{MHz})/(32768\text{cycles})]= 30.5\text{ms}$ .

About three to six cycles are needed by the CPU for executing a single line of instruction in the form of assembly code, but more cycles are typically consumed for executing an instruction in the form of C language code. This is because a single line of assembly is typically built of a single operator and two operands, while a single line of C can have many more operators and operands. However, the difference in the amount of object code produced by both languages is typically none or very little.

Code Composer Studio has a tool for counting the number of clock cycles consumed for every instruction. It's called the Profile Clock. When Code Composer is in debug mode, we can open the tool from the *Run Menu*. From there, we point to **Clock** and then select **Enable and Show**. The Profile Clock will appear in the lower right-hand corner of the Main Window.

## Stopping the Watchdog Timer

Stopping the watchdog is also referred to as putting it on hold or disabling it. But permanently disabling the watchdog is a bad practice, so we temporarily disable it in order to focus on developing a program without constantly having our program interrupted by a watchdog overflow.

The control register for the watchdog is a password protected register. Manipulating bits in a password protected register is not done in the same way as a conventional register. The password mask and the fields which we want to set are added together with the additive operator (+) to create a 16-bit wide binary number, and then the number is assigned to the 16-bit register variable. In other words, we sum up all the fields which must be set, and then write them into the register. Masks which are included in the summation will have their respective bitfields set, while masks which are not included will have their respective fields cleared. And finally, the password is always included; otherwise, a reset at PUC will be caused.

Diagram 36 shows the register for the following codes examples. It is a portion of the entire table shown by diagram 35, on page 91. The password (WDTPW) in binary notation is 0101101000000000, which is 0x5A00 in hexadecimal notation. The hold bit-field mask (WDTHOLD) is the symbolic constant for 10000000 (0x80). The mask and password are defined by the microcontroller's header file, and the password value is typical for all MSP430 watchdog registers.

**Code Example 22:** Placing the watchdog timer on hold.

```

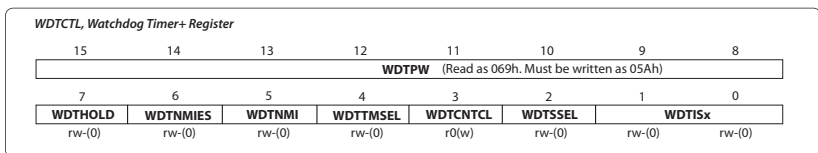
[REDACTED]; // Placing the watchdog timer on hold.

```

Now be aware that when this register is read, [REDACTED]



**Diagram 36:** Watchdog register for code example 22.



---

## Writing an Active Watchdog Timer Handler

For information about how to write an instruction that uses an interval reset instruction, see “Watchdog Timer Handlers” on page 178.

---

## Stopping the Watchdog Timer during the Boot Process

One important characteristic of the MSP430 boot program is its responsibility for initializing global variables. However, too many variables can adversely affect the execution of an instruction which configures the watchdog register.

In a scenario where your firmware involves instructions which initialize many global variables, and the instruction for holding the watchdog timer is located inside of the `main()` function, the watchdog timer may

---

## Using Boot Hook Functions to Stop the Watchdog and Execute other Instructions

After the reset system has managed the powering up of the microcontroller and initializing the registers, it loads the CPU program counter register with the address to the first instruction of the boot program. The program has its own set of instructions which initialize the firmware execution environment (the firmware image). A boot hook is a function which is intrinsic (built into) to the MSP430 compiler. It gives us direct access to two points in the boot program which are near the beginning and near the end of the program.

The point where a boot hook function gives us access near the beginning of the program is called system pre-initialization, and the function’s name is `__pre_init`. Use this function for containing instructions which are specifically needed for initializing a register before the `main()` function is called. It will be executed after `__pre_init`.

The point where a boot hook function gives us access near the end of the program is called system post-initialization, and this function’s name is `__post_init`. It will be executed during firmware initialization, but before any global variables are initialized. After the post-initialization function has been executed, the next instruction in the boot program will call our `main()` function.

For more information, see the `__pre_init` of the *MSP430 Optimizing C/C++ Compiler User’s Guide* (SLAU132).



## Using the `_system_pre_init()` function

Use this function for inserting firmware instructions near the beginning of the boot program.

This function is declared as returning an integer, and it is void of any parameters. Within its code block, there are two instructions. The first one puts the watchdog on hold. The second one returns an integer to the instruction in the boot program that calls this function. If the integer is 1, these instructions will be executed. If the integer is 0, these instructions will not be executed.

This function can only be used outside of the `main()` function.

**Code Example 23:** Using the system pre-initialization function.

```

1 int _system_pre_init(void)
2 {
3     wdog_hold();           //Put the watchdog timer on hold.
4     return 1;             //Return 1 to execute, or 0 to not execute.
5 }
```

## Using the `_system_post_cinit()` function

Use this function for inserting firmware instructions near the end of the boot program. It is declared as being void of a `void` value and void of any parameters. It contains an instruction, for example, that puts the watchdog timer on hold.

This function can only be used outside of the `main()` function.

**Code Example 24:** Using the system post-initialization function. It must appear

```

1 void _system_post_cinit(void)
2 {
3     wdog_hold();           //Put the watchdog timer on hold.
4 }
```

## Reading the Watchdog Timer Register

To read a watchdog register means to assign its contents to a storage variable. It is based on the technique used for reading a conventional register (not having a password), but it is slightly modified to take into account for a password. By using a password and masks, we can read specific register bitfields, or we can avoid using masks so the entire register can be read. Once the data is in the variable, it's available for use as needed.

. Therefore, our storage variable only needs to

be eight bits wide to hold the contents from the lower register. For the storage variable, we declare it as storing `uint8_t` unsigned char type of data.

**Code Example 25:** Reading the bitfields in a watchdog register.

---

```

1 uint8_t wdr; //Declare our storage variable.
2 uint8_t wdr_val; //Read the entire lower register.
3 uint8_t wdr_hold; //Read the WDT_HOLD field.
4 uint8_t wdr_nmies; Read WDT_HOLD and WDTNMIES.

```

---

When our instruction reads the watchdog register, the upper eight bits will always return as `0xFF`. And since we are only interested in the lower `0xFF`, we use binary subtraction to remove those upper register bits. Binary subtraction is an operation similar to decimal subtraction, meaning, each place in the operands are operated upon individually. Therefore, we must convert `0xFF` from an eight bit number to a sixteen bit number by appending two zeros (`0xFF00`). That new number will be able to subtract `0xFF00` from the upper eight bits, and leave us a result in the form of the lower eight bits of the watchdog register.

The first instruction reads the `WDT_REG`

```

uint8_t wdr_val;
uint8_t wdr_hold;
uint8_t wdr_nmies;
uint8_t wdr;
uint8_t wdr_val;
uint8_t wdr_hold;
uint8_t wdr_nmies;
uint8_t wdr;

```

The second instruction, which is based on the syntax of the first instruction, just reads a `uint8_t`

```

uint8_t wdr_val;
uint8_t wdr_hold;
uint8_t wdr_nmies;
uint8_t wdr;
uint8_t wdr_val;
uint8_t wdr_hold;
uint8_t wdr_nmies;
uint8_t wdr;

```

The third instruction reads `WDT_REG`

```

uint8_t wdr_val;
uint8_t wdr_hold;
uint8_t wdr_nmies;
uint8_t wdr;
uint8_t wdr_val;
uint8_t wdr_hold;
uint8_t wdr_nmies;
uint8_t wdr;

```



The `main()` function can easily be under appreciated, so that perception can make us overlook some important details about it. From the perspective of an MSP430, let's briefly look at its purpose, its two syntaxes, how it's called, whether data is passed to it, and about the data it returns.

---

### **Purpose**

The purpose of the `main()` function is to contain instructions which form the core program component of our firmware. Program components such as preprocessor directives, function definitions and their prototypes, interrupt service routines, and global variables are located outside of the `main()` function. Those external components are what enable us to develop modular programs.

---

### **How the main() Function is Called**

Other than not requiring a function prototype to be declared, the `main()` function is no different from any other C programming language function, so it can be called by another instruction which is located outside of `main()`. And that is what actually happens. During startup, and at the end of the `main()` function.

**Code Example 26:** Here is the exact instruction, along with its comment, in the `main()` function to be executed.

---

```
main() // Call main() function.
```

---



---

### **Syntax and Format for a C Language Function**

The format for the `main()` function follows the conventional rules for any other function. The standard format of a function definition is shown below. From that format, two syntaxes can be derived.

**Code Example 27:** Format for a C language function.

---

```
1 main()
2 {
3   return 0;
4 }
5
```

---

The `main()` can be any valid identifier. The `return` is the type of data which this function will return back to the instruction that calls this function.

The [redacted] identifies each variable and their type where data can be put or fed into the function. Within the function's code block are lines of instructions in the form of *definitions* and *statements*.

---

## The Two Standard Syntaxes for the main() Function

The international standard for the C language provides two `main()` function syntaxes from which to choose. What distinguishes one from the other is the parameter list. The first syntax is void of parameters, and the second syntax has parameters. Although the second one can be used, the first one is typically, if not exclusively, used in writing a `main()` function that will be executed by an MSP430.

---

### First Syntax and Format

This first form is used for writing the `main()` function for an MSP430.

**Code Example 28:** The syntax for writing a `main()` function.

---

```

1 [redacted]
2 [redacted]
3 [redacted]
4 [redacted]
5 [redacted]

```

---

The function is declared as producing a [redacted] in the form of an integer (`int`) and as being [redacted] of any [redacted]. This means that `main()` is expected to [redacted]), and that no data can be put into the `main()` function. The function's block of code or instructions are enclosed by curly brackets.

---

### `main()` is Void of Parameters

Notice that the instruction which calls `main()`, as shown by Code Example 26, will attempt to pass the integer zero to `main()`, but `main()` is declared as being void of any parameters. Why do these two instructions contradict each other?

[redacted]

---

### The return Statement

The last instruction in the block is a [redacted] statement. It passes data back to the instruction which calls `main()`. In this case, the data type is integer, having the value of zero, and all in accordance with the `main()` function's declaration.

As will be described by a later chapter, `main()` should be developed so that it will not be [redacted] executed. Meaning, the [redacted] instruction, at the end of `main()`,

should never be [REDACTED]. That method of development is called event-driven, and it is introduced on page 120.

So why bother to include a [REDACTED] instruction? Although the `main()` function will be designed to never allow the flow of execution to return back to the instruction that called `main()`, the [REDACTED] instruction is not necessary because [REDACTED]

If for some unintended reason the flow of execution does go back to the boot program, the next instruction in the boot is `exit(1)`. That puts the microcontroller into some low powered operating mode.

---

## Second Syntax and Format

This second syntax form is unconventional and is not used in developing a program for commercial production. Here is its syntax.

**Code Example 29:** The second syntax and form which is not used for developing a `main()` function.

---

```

1 int main(int argc, char *argv[])
2 {
3     /* */
4     return 0;
5 }
```

---

This syntax allows us to pass data to `main()` so the data can be used in some way. That syntax is most probably used for running test cases during development. Many of us who have used the C language for writing programs that run on a personal computer will recognize this format, since we used it for passing data from the operating system command line to the program. The Code Composer Studio scripting console can be used for passing command line data to `main()`.

For more information, see the section named “Passing Arguments to `main()`,” in the *MSP430 Optimizing C/C++ Compiler User’s Guide* (SLAU132).



## Program Development Nomenclature

---

### Routines

In the field of programming, a routine is understood as a set of program instructions which describe how to perform a task. The context here is an MSP430. Therefore, most tasks are triggered by a CPU interruption flag. One such task is in the form of interrupt service routine (ISR). The routine reads registers to get information, then uses the information to make a decision, and then uses the result of the decision to put data into a register, or to manage a set of stored program data, or both.

---

### Subroutines

A routine may be divided into subroutines, where each subroutine is a subset of instructions used for performing a subtask. Therefore, since we are viewing an ISR as a routine, the ISR then can be viewed as a set of subroutines.

---

### Block of Instructions

A routine or a subroutine may also be referred to as a block of instructions or code. Blocks are just abstractions we use for referring to and distinguishing between sets of instructions. A block is often delimited by brackets or parenthesis.

---

### Logic Circuits as Routines and Subroutines

But also keep in mind of another perspective of routines. They are not limited to program instructions. They may also refer to microcontroller logic and control circuits which carry out tasks. For example, the entire reset system of circuits can be viewed as a routine, and its three reset subsystems (BOR, POR, and PUC) can be viewed as subroutines.

---

### Service

Service is also another concept that has special meaning within the context of a microcontroller, and it's from the CPU's point of view.

A routine, subroutine, or single instruction tells the CPU what to do. In other words, when the memory address to an instruction is loaded into the CPU's program counter register, the CPU executes that instruction. The instruction will typically be for executing an ISR. And an ISR will typically be used for getting data from a module, using the data to make a decision, and then putting the resulting data of the decision back into a module. We may call that work performed by the CPU as servicing an ISR or servicing a module.

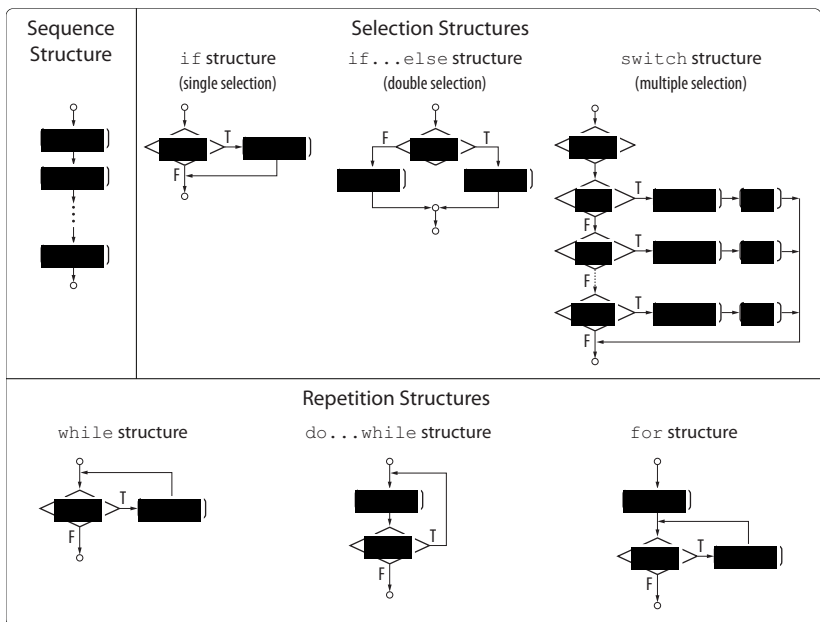




## Structures for Program Development

A flow of execution is defined as a sequence of program instructions. And for a program to carry out the result of a decision, the flow must be able to branch away from the decision so an appropriate sequence of instructions will be executed. The C programming language offers three categories of structures which control program flow. They are called the sequence structure, the selection structures, and the repetition structures. The sequence structure is used for creating a linear, non-branching, flow of execution. Decisions are not made in such a structure. The selection and repetition structures are used for making decisions. The result of the decision may transfer control of execution to an alternate flow.

**Diagram 37:** The fundamental flow control structures which we use for developing a program.



### Sequence Structure

The sequence structure is a natural form that is built into all programming languages. It can be visualized as a sequence of [redacted] where the program execution flows from one line of [redacted] to the next and from one [redacted] to the next. Such a sequence can be presented graphically with shapes which represent an instruction or routine. Arrows point from one shape to another. The shapes are typically in the form of rectangles, diamonds, and ovals which contain a caption that describes

the instruction or routine. The arrows are called flow lines, and they represent the flow of program execution. Sometimes the arrows may not appear because the shapes are butted against each other in a way that represents the flow.

---

## Selection Structures

This is the first type of structure which can be used for making a program decision. The selection structure is used for specifying that the next instruction to be executed may be other than the next one in the sequence. That is called transfer of execution flow control.

The C programming language provides three types of selection structures which are referred to as selection statements. All use a Boolean expression as a condition for making a decision.

The `if()` selection statement will

. The `if...else` selection statement will

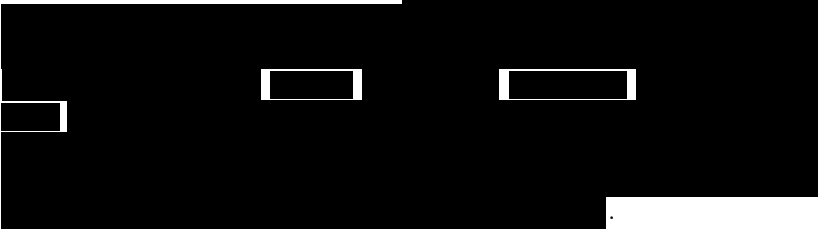
The `switch()` selection statement begins with making a decision, and then uses the result of the decision for transferring the flow to one of many different cases. Each case contains a subroutine of instructions for handling the case.

The `if()` statement is known as a single selection statement because it selects or ignores a single instruction. The `if...else` statement is known as a double selection statement because it selects between two instructions. And the `switch()` statement is known as a multiple selection statement because it selects one instruction among many different instructions.

---

## Repetition Structures

This is the second type of structure which can be used for making a program decision. The repetition structure is used for



## Basic Approach for Developing a Microcontroller Solution

When approaching a firmware and hardware development project for the MSP430, having a plan will help us organize our work so we can divide the project into manageable tasks and subtasks which together accomplish an objective.

An initial plan provides a framework that can be elaborated upon as the project progresses. Described here is such a plan. Modify it as needed. But what are not shown by this plan are tasks which carry out quality assurance, reliability, testing, and release to market or for going into service.

Be aware that these tasks do not have to be done in the order which they are shown. Furthermore, they can be grouped into subsets of common tasks.

---

### Task 1: Conceptualize the Problem and its Solution

To conceptualize a solution,

---

### Task 2: Design the Power Supply Interface Circuit

---

### Task 3: Design the Signal Input Interface Circuit

If the solution involves an

---

### Task 4: Develop Instructions which Configure the Signal Input Path

If the solution involves an input signal, instructions which configure the input signal path from the microcontroller's terminal to the appropriate module are to be developed. Meaning, when signals enter the microcontroller, their path will typically need

to be directed toward the proper module. The instructions will typically configure an input signal multiplexer, since most terminals on the case are multiplexed with more than one module. That is a job which is typically handled by the I/O Module. Some additional instructions may also be needed for conditioning the signals while they are on their way to the module.

---

**Task 5: Develop Instructions which Configure the Input Module**

If the solution involves [REDACTED]

---

**Task 6: Develop Instructions which Make Decisions**

After the input signals have been determined, develop instructions which [REDACTED]

---

**Task 7: Develop Instructions which Act on the Result of a Decision**

After the decision has been made, [REDACTED]

---

**Task 8: Develop Instructions which Configure the Output Module**

Develop instructions which configure the module that will handle the output signal.

---

**Task 9: Develop Instructions which Configure the Signal Output Path**

The output signal flows on a path from the signal output module to a pin on the case. Therefore, instructions which configure the output signal path from the module to the appropriate terminal on the microcontroller's case must be developed.

---

**Task 10: Design the Signal Output Interface Circuit**

The output signal must flow [REDACTED]

A use case is the act of putting something to work within a particular situation. The field of engineering utilizes them to visually show the configuration of a product while in use, since it provides a starting point where engineering development may proceed.

An MSP430 is typically used for monitoring and controlling devices. An abstract reference model of the MSP430 is presented here that can be utilized to visually show its configuration while in use. It is generic, so it can apply to the monitoring case, the controlling case, or both. It provides a starting point where engineering development may proceed. And it can be elaborated upon during the course of a development project to provide less abstraction and more details.

A well designed reference model can serve as an engineering specification that guides the program development process, a topic covered by the next chapter. If not used for creating a specification, the model will at least provide a visual guide that will help us get oriented with our development efforts.

---

### **Structural Overview**

As shown by diagram 38, the MSP430 can be visualized as three stacks of elements which stand on a single program element.

The stack on the left handles input signals, while the one on the right handles output signals. Situated at the base is the core element, the program. It contains all the instructions which the CPU will carry out. Elements which are shaded contain registers which the CPU can read and write into. The middle stack represents electrical power that supplies energy to the microcontroller.

There are scenarios where data or signals may go both ways within a stack of elements. For example

The focal points in this model are the

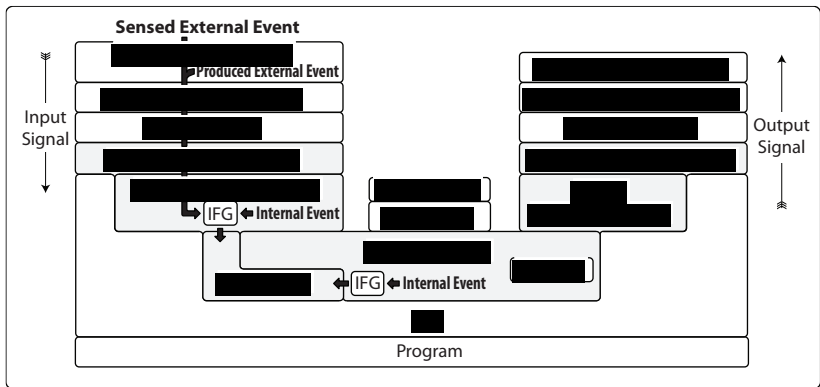
The physical boundary between the microcontroller and its environment is represented by two elements: the input terminals and the output terminals. Elements which

are above the terminals are located outside of the microcontroller and are part of its environment.

Two layers of elements are able to

As for the CPU, it steps through our program to execute the instructions. Some instructions tell the CPU to read data from a register and use it to make a decision. Other instructions use the result of the decision to write data into registers, into volatile storage variables, or into both. Furthermore, parts of the program are in the form of interrupt service routines. So depending on which flag was set by the event, the interrupt system tells the CPU to execute the routine linked to that specific flag.

**Diagram 38:** The reference model.



## Input Signal Stack

An event is something that happens, takes place, or occurs. The MSP430 is designed to monitor and react to events. They may occur from outside or inside of the microcontroller. Although an event can be characterized as any physical phenomena, the result of the event ultimately becomes a voltage signal which enters a terminal pin on the microcontroller's case.

In a properly configured microcontroller, an event will

When the CPU is finished with the ISR, it resumes with what it was doing before it

was interrupted, which is typically some operating mode of sleep. The ISR is designed to handle the event and produce an appropriate output signal.

## Externally Occurring Events

An external event occurs at the outside of the microcontroller. An input peripheral device is used for sensing or producing such events. If sensed, it is a passive device such as a sensor. If produced, it is an active device such as a switch.

Passive devices will convert a sensed event into an electrical signal. While on the other hand, active devices do not sense events, they produce them, and they are in the form of electrical signals. Regardless of the type of device, we call the signal it produces a microcontroller input signal. The signal is then supplied to the next lower element in the stack, a conditioning circuit.

The input signal will typically need to be

When the signal appears on a terminal, it typically flows into an I/O module where it is switched to a specific destination by a signal multiplexer. Most, if not all terminals on the MSP430 are

Meaning, a single terminal has access to

During power-up, the program had configured the input peripheral module to monitor for input signals. So when it senses a signal, the raw signal is placed into a storage buffer or converted into a binary number and placed into a register, and then the module sets an interrupt flag. The flag is in the form of a register bitfield, which is continuously being monitored by the CPU interrupt system, at the next lower layer.

When the interrupt system notices a set flag, it

The purpose of the ISR is to provide instructions to the CPU about how to handle and react to the event. The instructions will typically specify five tasks. 1) Read any relevant data provided by the input peripheral module that characterizes the signal. 2) Read any relevant data stored in main memory. 3) Use all the data to make a decision. 4) The result of the decision is then used for updating any relevant (volatile) data stored in main memory and writing data into the output peripheral module registers. Data changing in a module's registers will automatically drive it to carry out the work which it was designed and configured to do. That work produces a signal which



then drives the output peripheral device. And 5) the final instruction clears the interrupt flag.

---

### Internally Occurring Events

An internal event occurs at the inside of the microcontroller. They are produced and sensed by [REDACTED]. For example, if it is an analog to digital (ADC) peripheral module, it could be monitoring a built-in temperature sensor. If it is a real time clock module, it could be producing and monitoring for timer counter overflows. The types of internal events which an MSP430 can sense and produce will vary from one model to another.

An internally occurring event is handled like [REDACTED]. Meaning, [REDACTED]

[REDACTED] When the CPU has finished with the ISR, it resumes with what it was doing before it was interrupted., which typically is a low powered operating mode we call sleep.

---

### Output Signal Stack

The source of an output signal is typically, if not always, from an interrupt service routine. If it isn't from an ISR, the program was probably developed to execute a programming example or test, since the proper way to produce output signals are with an ISR. Therefore, the scenario here involves an ISR being the source of an output signal.

An ISR is part of the program. Its purpose is to provide instructions to the CPU about how to handle and react to an event. After it has made a [REDACTED]

[REDACTED]. The module then produces a signal which drives or sends a message to an output peripheral device. Therefore, an output signal can be defined as either updating volatile data, driving a peripheral device, sending a message to a peripheral device, or all three. The signal begins as binary data, and it ends as some specific form of voltage signal.

If the output signal is used for updating a set of volatile data, the scenario may involve saving the stored data until a time when it must be sent to a peripheral device.

[REDACTED], and then it writes the data into a volatile storage array.

If the output signal is used for driving a peripheral device, the scenario may involve configuring the registers of a peripheral module to produce a special type of output

signal that drives a peripheral device. For example, the ISR writes into timer module registers so the module will produce a pulse-width modulated signal that drives a brushless electric motor.

And if the output signal is used for sending a message to a peripheral device, the scenario may involve configuring the registers of a peripheral module to produce a signal in accordance with a standard protocol so a peripheral device can read it. For example, the ISR writes some words into the registers of an enhanced universal serial communication interface (eUSCI) module which is operating in the inter-integrated circuit (I2C) mode. Once the module senses a change in its message input registers, it automatically sends the message out a port channel or some other dedicated pin.

The three elements in between the output peripheral module and output peripheral device act in the same way as those same elements in the input signal stack, but in reverse. A multiplexer (in an I/O module) switches the circuit from a peripheral module to an output terminal. And a conditioning circuit is used for converting the output signal into one which can be accepted by the peripheral device. It adjusts the voltage, current, and noise to acceptable levels, and then supplies the signal to the next higher element, the peripheral itself.

---

### Power Supply Stack

This stack represents electrical energy flowing into the microcontroller. The energy is used for giving power to the microcontroller.

At the top of this stack is the [REDACTED]

The supply [REDACTED] layer is an abstraction of the [REDACTED] on [REDACTED] [REDACTED] on the microcontroller, and the electrical circuit that interconnects the two sets of terminals. The circuit is designed to condition the electricity so it can be within the specified range of voltage and current.

Below the interface is the system modules layer. The MSP430 has at least two system modules which handle the power supplied. They are known as the [REDACTED]



## Patterns for Program Development

A pattern is something shaped or designed to serve as a model from which a thing is to be made. In this context, the pattern serves as a model from which a program is to be made.

While the reference model, as presented by the last chapter, provides a broad abstract view of the microcontroller's entire operation, these patterns are narrower in view. They provide abstract views of just the program's operation. Therefore, they are a starting point from where the actual program development process may begin. They show the basic set of instructions, routines, and their relationships which are needed for developing a program for the MSP430.

Two program patterns are presented here. The first one is called the repetitive pattern, and it is appropriate for developing programming examples and tests. The second one is called the event-driven pattern, and it is used for developing commercial grade programs.

Be aware of what is not provided by these patterns. They are not complete pictures of a program. A program written in the C programming language, especially its `main()` function, will depend on instructions and routines which support it. For example, they are instructions which include specific C language and MSP430 libraries, instructions which define preprocessor directives, global variables, global functions and their prototypes, and boot initialization functions. Therefore, those supporting instructions are not shown by the patterns. That is a topic that will be covered and elaborated upon by the next chapter, and *completely covered* by the last chapters.

---

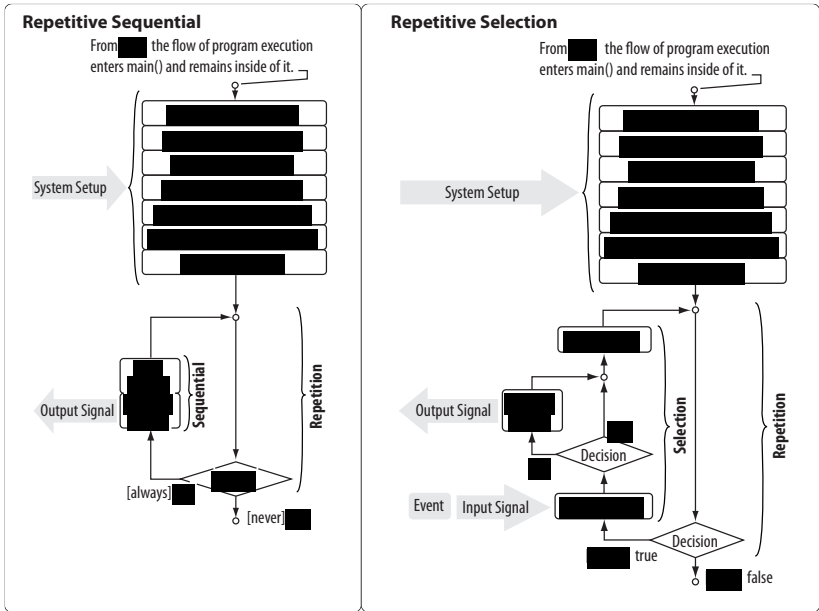
### Repetitive-Driven Pattern of Program Development

This pattern should be limited to projects which present a programming example or possibly for firmware testing scenarios. Unless a rationale argument can be made for using it, this pattern should not be considered for producing commercial grade products.

As shown by diagram 39, two forms of the repetitive pattern are presented here: the sequential and selection.

Both patterns are written entirely inside of the `main()` function. They begin with a sequence of instructions which configure and setup the microcontroller for operation. Following that sequence is a single repetitive loop of instructions. That loop is the core characteristic of the repetitive pattern, and the instructions inside of it are what distinguishes the pattern as being sequential or selection. The sequential pattern does not depend on an input signal to produce an output signal, while the selection routine depends on an input signal to produce an output signal.

**Diagram 39:** The repetitive-driven patterns of program development.



**Configure and Setup Sequence**

Before doing any actual work, a sequence of instructions which configure and setup the microcontroller for operation must be executed.

**Watchdog Timer Handler**

This sequence of configuration routines begins with a watchdog timer handler. The typical usage scenario, such as a programming example or test case, will have the watchdog disabled. Once it is disabled, an oscillator settling handler is executed.

**Handler**

An oscillator is used for producing a timing signal which drives the clock module,

So to mitigate that scenario, we use an oscillator settling handler. It puts the flow of execution into a small loop where the condition of the flag is read, and if the flag is set, it is then cleared, and the flow of execution goes back to the beginning of the loop. If the flag is read as being cleared, then the flow exits the loop and continues with the next system setup instruction.

---

### Signal I/O Multiplexing

The signal multiplexers (in the IO/ modules) are now to be configured. Since the sequential pattern only produces output signals, a multiplexer is configured to create an outbound circuit from an output peripheral module to a terminal on the microcontroller's case. The servicing routine, located inside the sequential routine, is the source of the output signal.

While on the other hand, the selection pattern uses an input signal to produce an output signal. Therefore, two multiplexers are configured. One creates an inbound circuit from a terminal to an input peripheral module, while the other creates an outbound circuit from an output peripheral module to a terminal.

---

### Configure the [REDACTED] Modules

---

### Unlock [REDACTED]

---

### Reset Fault Handler

A reset fault is an event produced by a system module or a signal at an input terminal which is configured to restart the microcontroller (called reset mode).

If the reset fault is produced by a module, the event is in the form of a voltage brown-out, security violation, watchdog timer overflow, or some other system fault. If produced by a terminal in reset mode, the event is in the form of a signal produced by, for example, a switch. Such a switch is used on computers to restart them from an unstable operation or a crash.

These are all non-maskable CPU interruptions (NMI) which set their own particular interrupt flag. Therefore, the intent of this reset fault handler is to determine which flag was set, and use that information in some way. Possibly, to produce an output signal that illuminates an LED. The handler would then clear the flag.

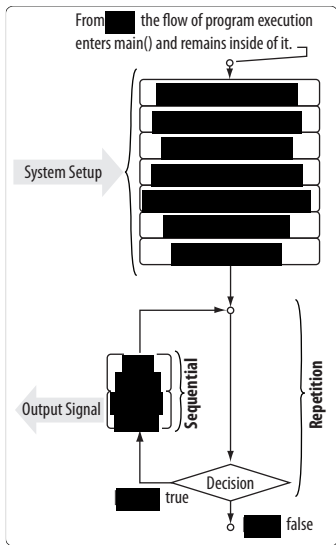
### The Repetitive Sequential Routine

The repetitive sequential routine does not depend on an input signal to produce an output signal. Use it for producing a repetitive output signal which does not depend on making a decision.

**Diagram 40:** The repetitive-driven pattern of development using a sequential routine.

This routine is built of two program execution flow control structures. One is a repetition structure, and the other is a sequence structure.

The repetition structure is in the form of a



Inside of the loop is a sequence of instructions. The first instruction is a [redacted]. This represents a microcontroller output signal.

The second instruction is a [redacted]. It occupies the CPU for a specific number of clock cycles, so it will not execute the next instruction. Otherwise, the servicing routine would be executed again without any delay in time. [redacted]

[redacted]

Once the delay handler is executed, the flow of execution returns to the decision at the beginning of the repetition structure.

A programming scenario that will use this repetitive sequential routine, for instance, is the blinky programming example. It is typically loaded into the microcontroller at the factory. The servicing routine is a short sequence of instructions which configure the digital I/O module to produce a repetitive cycle of signals at a pin that flashes an LED.

## The Repetitive Selection Routine

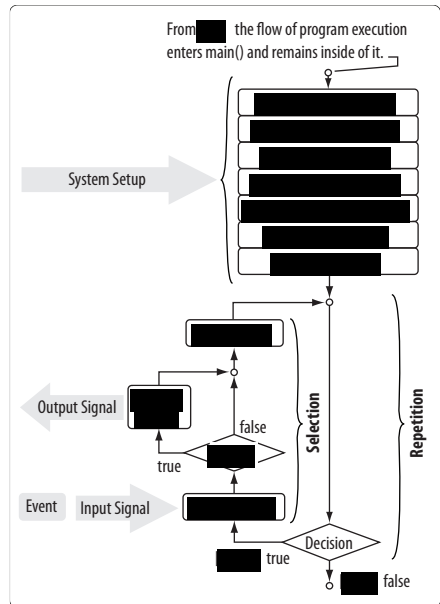
The repetitive selection routine will depend on an input signal to produce an output signal. Use it for producing a repetitive signal which depends on making a decision.

**Diagram 41:** The repetitive-driven pattern of program development using the selection routine. The selection routine depends on [REDACTED].

This structure is similar to the [REDACTED]. It is also an infinite loop, in the form of a [REDACTED] statement, that contains a selection routine instead of a sequential routine. The selection routine is also referred to as a [REDACTED] routine. Let's begin with the first instruction inside of this loop.

The first instruction in the loop is a [REDACTED].

The next instruction is a [REDACTED].



The decision is typically in the form of an `if()` selection statement. The decisions are made on conditions built of relational or equality operators. If the condition evaluates to zero or false, the [REDACTED] routine is skipped. If the condition evaluates to a non-zero number or true, the [REDACTED] routine is executed. In this case, the condition is built of an operand in the form of the storage variable, an operator, and another operand which the variable is compared with.

The last instruction in the [REDACTED] routine is a [REDACTED]. It occupies the [REDACTED].



Once the delay handler is executed, the flow of execution returns to the decision at the beginning of the repetition structure.

A programming scenario that will use this repetitive selection routine, for example, is an electronic thermometer. The input device is peripheral in the form of a built-in temperature sensor with an input to an analog-to-digital converter (ADC) module. The sensor monitors the temperature of the microcontroller's case, sends the reading as analog voltage signal to the ADC where its converted to a binary number, and then the ADC writes the number into a register. That data represents the microcontroller input signal. To get the data, a polling handler reads the register and then writes it into a storage variable. A decision is then made about

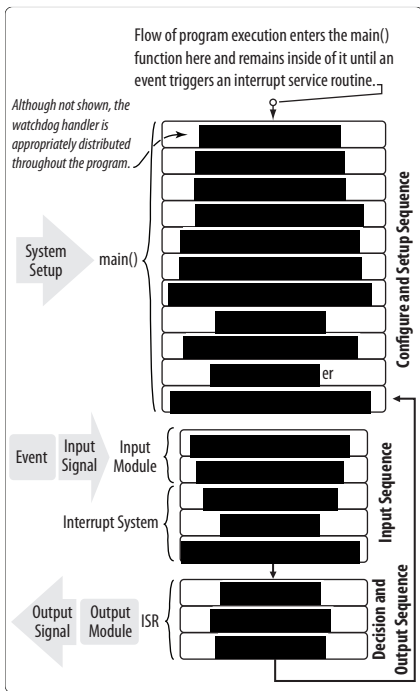
The display then graphically presents the new temperature. A delay handler then occupies the CPU for a minute before the flow of execution returns to the decision at the beginning of the repetition routine.

### Event-Driven Pattern of Program Development

**Diagram 42:** The event-driven pattern of program development. Use this pattern for developing commercially sold products.

Use this pattern for developing commercially sold products. It is constructed of three basic sequences of routines: a sequence that configures and sets up the microcontroller, an input signal sequence, and a decision and output signal sequence. Input signals are referred to as events.

The flow of execution begins with the sequence of routines which configure and setup the microcontroller for work. It's like the corresponding sequence of routines in the repetitive-driven pattern, but elaborated upon.



The first of those three routine enables the microcontroller to

Once it is asleep, the microcontroller is ready to handle events. An event produces an input signal. The flow of execution resumes when an input signal triggers the

Once the signal input sequence has been executed, the decision and signal output sequence is now executed. Meaning,

---

### System Configuration and Setup Sequence

This sequence of routines is entirely contained by . The purpose of this sequence is to prepare the microcontroller for work and finish with putting it into a low powered operating mode of sleep.

---

### Watchdog Timer Handler

During program development, the watchdog timer is typically disabled so it will not interfere with the development process. But near the end of the process, a proper watchdog timer handler must be developed. A watchdog is an essential part of a program which is destined for commercial use. Its purpose is to handle CPU crashes.

The final program will typically

One obvious place to put a handler is at the end of the configuration sequence, but right before the instruction that puts the microcontroller to sleep. That would ensure a cleared timer counter before an ISR is executed.

Be aware that if a program has many global storage variables to declare, the counter may overflow before that process is completed. Such a delay may result in triggering a microcontroller restart before the flow of execution can reach `main()`. In that case, a timer handler would have to be placed in the boot program. Placing the handler inside of the boot is done with an MSP430 intrinsic function. For more information,

see “Stopping the Watchdog Timer during the Boot Process,” on page 95. By the way, using global variables in event-driven firmware is a risky practice.

---

### About the Next six Routines

The following routines, from the oscillator settling handler to the reset fault handler, with exception to multiplexing and channel interrupt flag handling, are just like those routines in the repetitive-driven pattern of development. Therefore, those sections are repeated here.

---

	<b>Handler</b>

---

### Signal I/O Multiplexing

Signal multiplexers are now to be configured. Two circuit paths are configured with them. One forms an inbound circuit from a terminal on the case to an input peripheral module, while the other forms an outbound circuit from an output peripheral module to a terminal.

---

### Configure the System and Peripheral Modules

Routines are now used for configuring the modules. The system modules, such as the clock, are configured first, since peripheral modules will probably depend on them in some way. The peripheral modules are then configured.

Not all modules are configured. Just those which XXXXXXXXXX  
XXXXXXXXXX.

Although the I/O port channels can be configured now, their interrupt flags (IFGs) cannot be cleared to zero before the port channels are unlocked. Therefore, that routine is placed after unlocking the channels.

---

## Unlock the Digital I/O Port Channels

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

---

## Port Channel Interrupt Flag Handler

[REDACTED]

---

## System Reset Fault Handler

A system reset fault is an event produced by a system module, or it is a signal at an input port channel which is configured to tell the microcontroller to reset itself.

If the fault is produced by a module, [REDACTED]

[REDACTED]

A reset fault causes a System Reset Interruption. Each type of interruption will set their own particular interrupt flag.

The intent of this reset fault handler is to determine which reset interrupt flag was set, and use that information in some way. Possibly, to produce an output signal that illuminates an LED. The handler would then clear the flag.

To repeat an important concept, keep in mind that every channel's interrupt flag can only be cleared after all channels have been unlocked.

## Enable Maskable Interruptions

A non-maskable CPU interruption (NMI) is caused by an unplanned event which makes the microcontroller unstable or inoperable. Such events are categorized as System NMIs (SNMI) and User NMIs (UNMI). A system NMI is caused by a system reset fault. A user NMI is caused by an oscillator fault or a signal sensed at the  $\overline{\text{RST}}$ /NMI pin. To include a user reset which is triggered by the  $\overline{\text{RST}}$ /NMI pin, the port channel which is connected to the pin is configured into NMI mode; meaning, it will cause an interrupt service routine to be executed, instead of resetting the microcontroller. An NMI cannot be stopped or blocked from interrupting the CPU.

A maskable CPU interruption is caused by a planned event which occurs at a peripheral module. It is the type of event which the program is designed around and meant to handle. In other words, these types of interruptions, quite literally, represent the input signals which the microcontroller is intended to handle. Such events are sensed and produced by peripheral modules and some system modules. Interruptions caused by these events must be enabled, otherwise they are blocked from interrupting the CPU.

So when a module senses an event, it sets a bit in a particular register which belongs to the module. The bitfield is called an interrupt flag (IFG), and these flags are monitored by the interrupt system. If it is a maskable IFG, it can be blocked from the interrupt system's view. To allow these maskable flags to be seen, or unmasked, a bitfield in the CPU Status Register is set. It's called the General Interrupt Enable (GIE) bitfield. We typically use an MSP430 intrinsic function to set that bit.

When the microcontroller powers-up or resets, the GIE bitfield is in a cleared state. Therefore, maskable interrupt flags cannot be sensed by the interrupt system.

At this point in the system configuration and setup sequence, after the reset fault handler, is where we place an instruction that sets the GIE bit. Otherwise, the microcontroller will not become event-driven. It will not execute an ISR to handle an event.

Also assure that the

## Volatile Data Handler


sections. Therefore, that data is called volatile. Those modes are referred to as fractional low powered operating modes (LPMx.5); for example, LPM3.5. In order to mitigate the loss of data during a fractional low powered operating mode, we currently have two options: a special function that places such data into non-volatile memory or a backup memory register. The function is called PERSISTENT(), and the registers are typically called BACKMEM. The function is always available with microcontrollers built of FRAM, but not the registers.

If your microcontroller will be entering a low powered operating mode that will lose volatile data, this is the point in the configuration sequence when that data must be saved.

A volatile data handler is just simply one or more instructions which reads data from volatile storage and then writes it into the BAKMEM registers or the PERSISTENT() function.

---

### Enter a Low Power Operating Mode


 in the configuration sequence is a single instruction that places the microcontroller into a specific low power operating mode.

In other words, after a successful power-up, all the preceding routines will be executed, and then the flow of execution will come to this instruction, which tells the CPU to put the microcontroller to sleep.

Only a CPU interruption will wake up the microcontroller and put it into the active operating mode.

---

### Input Signal Sequence

The input sequence is the  block of routines in the event-driven pattern as shown by diagram 42 on page 120. It starts with an event which produces a signal.






---


### Decision and Output Signal Sequence

The decision and output sequence is the last block of routines in the event-driven pattern. With the CPU having the first instruction to the ISR and the microcontroller in active mode, the ISR is now executed.

Inside the ISR is a sequence of three basic routines. The servicing routine carries out the decision making instructions and then the result of the decision produces an output which tells a peripheral module to act in some way. If the microcontroller will be going back into a fractional low-powered operating mode, and the resulting data from the decision must be saved, then a volatile data handler will be executed. If a conventional low powered mode will be re-entered, there is no need to protect volatile data.

 in the sequence is an instruction which clears the flag bit to zero. A concept called an interrupt vector will be introduced later in this book. It binds one or more flags to a single ISR. If the vector binds just a single flag to an ISR, then such flags are automatically cleared. If the vector binds more than one flag to an ISR, the ISR will have to determine which flag was set, and then transfer the flow of execution to the proper subroutine in the ISR to handle the event and then clear the flag.

The latest models of MSP430 are now being built with a special type of register called an 

 .

Once the last instruction in the ISR has been executed, the microcontroller is automatically put back into the same low-powered mode of sleep from where it was interrupted.

## Placing the Event-Driven Pattern into a Larger Context

The event-driven pattern, which was introduced by the previous chapter, can be better understood when placed into a larger context. That means viewing the pattern as being surrounded by all the supporting firmware routines, built-in microcontroller routines, physical connections, and the relationships between them all. As shown by diagram 43, it is the big picture of what we must take into account when developing a program for the MSP430.



The firmware image we develop and load into the microcontroller is built of four components: the boot program, the `main()` function, the ISR, and the preprocessing translation units. A standard template, which is automatically included in a Code Composer Studio development project, is used by the MSP430 compiler to build the boot program. But we may insert instructions which augment the template. For example, instructions which handle the watchdog timer or change some registers before `main()` is called. As for the `main()` function, we must develop the entire function.

After `main()`, we are then concerned with the preprocessing translation units. These units do not represent an ordered sequence. We have to develop some of these units, while others are pre-developed.

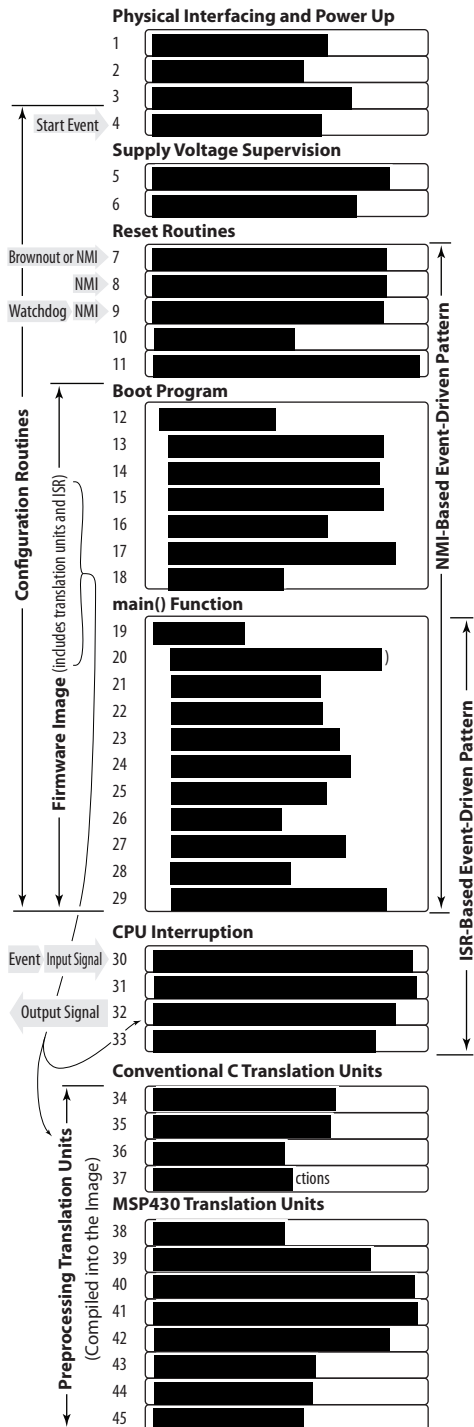


**Diagram 43:** The event-driven pattern as it is placed into a larger context.

The block of conventional C language translation units enable us to produce and use modular blocks of instructions for developing the program. The block of MSP430 translational units enable us to use and produce proprietary modular blocks of MSP430 code for developing the program. During the program preprocessing and compilation process, those translation units are expanded and inserted into the firmware image. Some of the units are absolutely needed, while others are not. Their usage depends on the needs of the program.

There are two event-driven patterns of development in the diagram. The first is of the ISR-based pattern that was introduced on page 120. It is built with the block of `main()` function routines and the block of CPU interruption routines. The microcontroller's primary use case is focused on this pattern. The second is of the NMI-based pattern. It supports the microcontroller when something goes wrong. Therefore, the microcontroller's secondary use case is focused on that pattern. It is built of the reset routines, boot program, and `main()` function. As you can see, the two patterns have some overlap.

Except for the ISR-based event-driven pattern, since it was explained by the previous chapter, the remaining sections of this



chapter will elaborate on the items shown by diagram 43. Also, a special topic is inserted prior to the section about the reset routines. It's about the operating mode diagram that can be found in every MSP430 user guide. It explains important information which puts the reset system and the remaining routines into yet another operating context.


---

## Physical Interfacing and Power-Up

The physical interfacing tasks involve connecting a peripheral device to the microcontroller. The power-up event is the application of electricity that energizes the microcontroller.

At line 1, the circuits which interconnect the peripheral device with the microcontroller are designed, fabricated, and then installed. Depending on the type of peripheral device, either an input circuit, an output circuit, or both circuits are needed. The circuits condition the signals to be within the microcontroller's operating specifications. For input signals, conditioning means to bring and maintain the voltage and current to within the specifications which the microcontroller can accept. For output signals, it means to convert the voltage and current to the specifications which are needed by the peripheral device.

Line 2 represents the



There is one last matter about connecting power to the microcontroller. It has to do with controlling noise at the VSS and VCC pins. Those pins are typically very close to each other, if not right next to each other. Fluctuations in power demand can act to create a small noise signal across those pins, and that noise may affect the operation of the microcontroller. So to mitigate that noise, we use a circuit to decouple those two pins. The circuit will typically involve a couple of capacitors. One circuit is used for the digital power pins (DVSS and DVCC) and another is used for the analog power pins (AVSS and AVCC). The microcontroller's data sheet will tell you what is specifically needed. The section is called "power supply decoupling" or something like that.

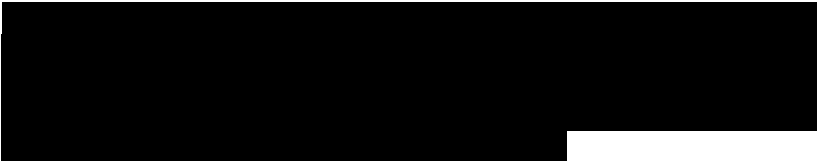
Line 3 represents all unused pins on the microcontroller and how they must be properly terminated. Although each microcontroller has its own requirements, this is what you can expect. For unused port pins, they should be configured to the signal output direction. Pins which supply JTAG services are used for loading firmware into the microcontroller and for running the microcontroller in debug mode. They are typically multiplexed with other services, so they should be switched to the I/O port function and then put into the signal output direction. For the microcontroller which has pins where analog voltages are supplied (typically for an analog to digital converter), the AVCC pin should be connected to DVCC, and AVSS should be connected to DVSS. For complete information about these matters, see the microcontroller's user guide and data sheet. Both documents have sections which are named "Connection of Unused Pins," or a similar title.

A start event points directly into line 4, the power-up. The power supplying event can be caused by any type of device which directly applies the required amount of power to the DVSS and DVCC pins on the microcontroller. The power could be from a switch, a voltage regulator, or just simply a direct connection to a battery. Once the required amount of energy is reached, 1.8 to 3.6 volts DC, the microcontroller begins to operate.

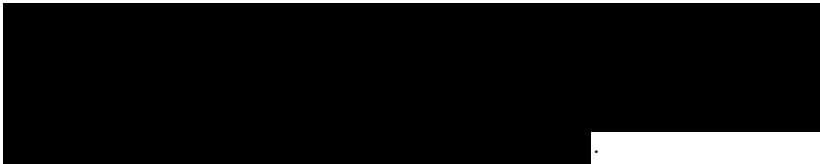
---

### Supply Voltage Supervision

This discussion follows the path through the digital voltage supply (DVCC) pin. The analog supply (AVCC) has its own separate but similar path into the microcontroller.



The supervisor is made of at least two logic circuits. One circuit monitors the voltage as it rises through the power-up event threshold, called VSVSH+. The other circuit monitors the voltage as it falls through the power down event threshold, called VSVSH-. Those acronyms are used in the data sheets, while the user guides typically use slightly different acronyms.



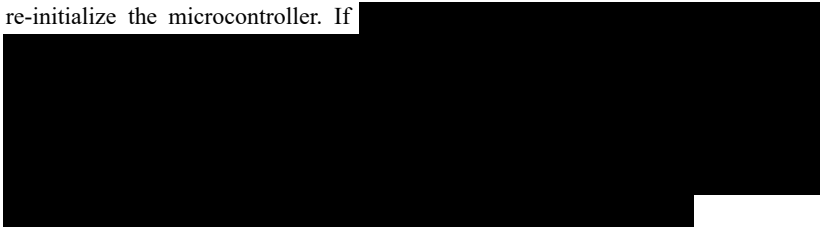
From a program development point of view, this is what we need to know (with some creative license). During a power-up, when the power-up supervisor recognizes that the voltage has risen above and remains above the power-up event threshold (VSVSH+), the supervisor then creates a delay of about 10 milliseconds to allow the microcontroller to become properly energized. After that amount of time has passed, the supervisor produces a signal that results in a BOR signal. On the other hand, during a voltage brownout scenario, when the power down supervisor recognizes that the voltage has fallen below and remains below the power down event threshold (VSVSH-), the supervisor produces a signal that tells the power management module to properly remove power supplied to the microcontroller. When power comes back, the power-up supervisor handles that event. User guides will not explicitly name these two types of supervisors, but instead they refer to them as the brownout circuit.

---

### Operating Mode Diagram

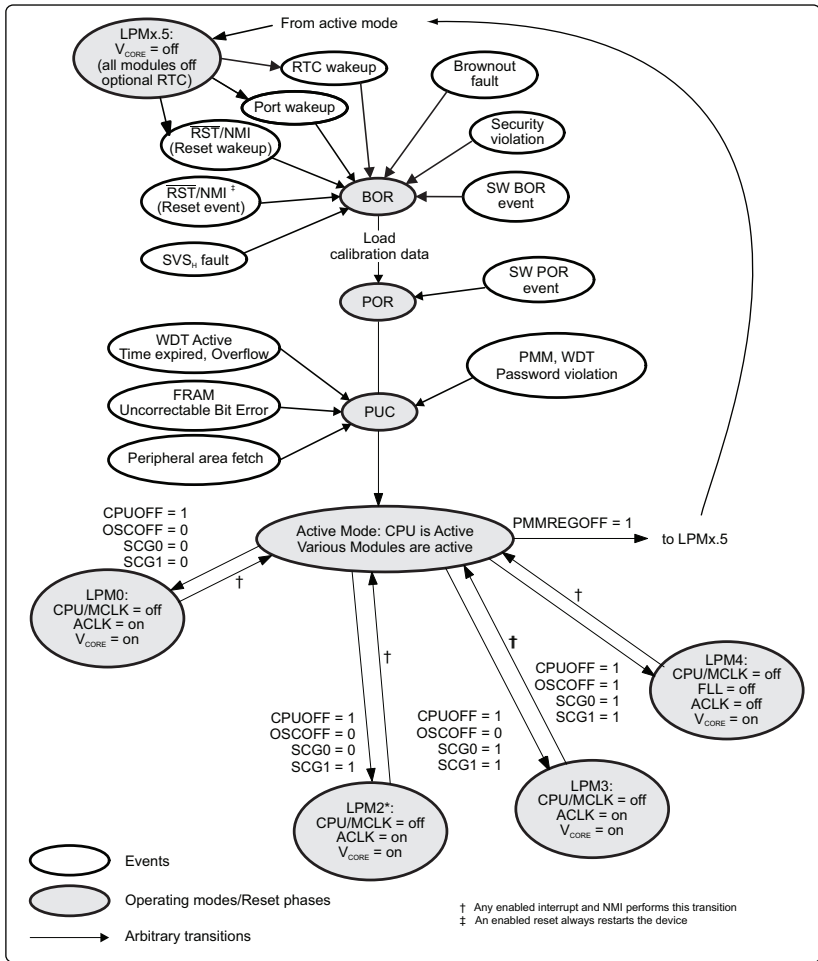
Before going onto the reset system, an introduction to the operating mode diagram is in order. Having knowledge about this diagram will help you to better understand the event-driven pattern and its larger context, which the remaining sections in this chapter will describe.

The reset system's purpose is to carry out a sequence of routines which initialize or re-initialize the microcontroller. If



When compared to the initialization scenario, re-initialization has a larger context since NMIs, brownouts, and operating modes are involved. The microcontroller's user guide publishes a diagram which shows the operating modes, the reset phases, the events which cause NMIs, and their relationships. It's called the operating mode diagram, and an example of it is shown by diagram 44. Unfortunately, it does not explicitly take into account the maskable CPU interruptions. Otherwise, it would be a complete view of the microcontroller's operating modes.

**Diagram 44:** The typical operating mode diagram as published by an MSP430's user guide. It shows the operating modes, the reset phases, the events which cause NMIs, and their relationships.



The operating mode diagram uses

The power-up event is not shown. It is a phase where the microcontroller begins unpowered and reaches the required operating voltage. If it were shown, it would be located above the BOR with an arrow from itself to the BOR.

All the reset phases and their release to the active operating mode, which appear in the operating mode diagram, are shown on lines 4 through 11 inside of the larger context presented by diagram 43, on page 128. And the low powered modes of sleep are all represented by line 29, including the fractional low powered mode.

NMIs are produced by events which typically occur at system modules. Such reset events are in the form of operating faults, security faults, real time clock (RTC) counter overflows, and an external reset (RST) signal at a digital I/O pin which is specifically configured to produce a reset NMI.

Not explicitly shown by the operating mode diagram are

Unlike the transitions from a conventional low powered mode to the active mode, the transitions from a fractional low power operating mode do

At the bottom of the diagram are four low powered operating modes. Next to each mode is a list of four items. The items represent the bitfields in the CPU status register and whether they are set or cleared for entering a mode.

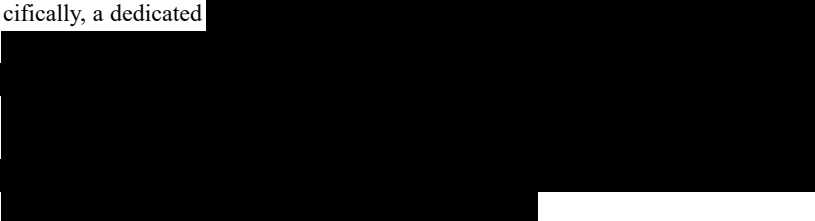
If the CPUOFF bitfield is set, that turns off the CPU. When the OSCOFF field is set, that disconnects the clock module from an external oscillator, such as a watch crystal. The two SCG fields are used for turning off the various clock signal buses. So depending on the combination of the two bitfields, some combination of the main clock bus (MCLK), the sub-main clock bus (SMCLK), or the auxiliary clock bus (ACLK) can be turned off. The CPU status register table gives the complete details. Also refer to the microcontroller's data sheet for a full characterization of each operating mode. The section is called "operating modes," obviously.

To place a microcontroller into a specific operating mode,

The remaining matters which are of concern to us are the SW BOR event, the SW PUC event, and Vcore. The SW means software, but in this book it means firmware

(FW). So what is pointed out here is an instruction in our program that sets an interrupt flag in some register that forces a restart at a BOR or a PUC. Setting the flag is the event.

As for Vcore, that refers to core voltage. The power management module is responsible for managing and distributing power throughout the microcontroller. More specifically, a dedicated







For a complete list of maskable and non-maskable CPU interruptions, see the microcontroller's data sheet. The section is usually called the "Interrupt Vector Table."

---

## Reset Routines

The reset system's purpose is to use one or more phases in a sequence to initialize or re-initialize registers, then load the first instruction of the boot program into the CPU, and then put the microcontroller into the active operating mode. Initialization means to clear or set bitfields to an initial state of operation. Each phase is responsible for handling their set of bitfields. Therefore, we can say that a phase is initializing to a BOR, POR, or PUC state. However, the phases do overlap with their initialization work. Meaning, each phase may initialize the same register, but they will not initialize the same bitfields.



A power-up or supply voltage brownout event starts a process that properly energizes or re-energizes the microcontroller, and when finished, to produce a signal that tells

the first reset phase (BOR) to begin working. The energizing process is typically handled by a supply voltage supervisor (SVS), or the power management module (PMM), or by both.

There are several types of events which will produce a system reset (re-initialization) signal. They are referred to as system reset interruptions. Each type of reset interruption has a dedicated block of event monitoring logic to produce a BOR, POR, or PUC reset signal. The common name for a reset signal is just simply its event name.

From a program development point of view, we are interested in

- [REDACTED],
- [REDACTED],
- [REDACTED],
- [REDACTED],
- [REDACTED],
- [REDACTED].

Taking that information into account will help us develop a program that will properly configure the microcontroller, but it will also help us to develop a program that properly handles events which cause the microcontroller to be reset.

The microcontroller's data sheet will tell us all the events which will initialize and reset the registers along with their interrupt flag names. The section is typically called the "Interrupt Sources, Flags, and Vectors or the Interrupt Vector Table and Signatures."

Every register table will show which bitfields are initialized, by which phase, and to which binary state. Under the register diagram is initial condition notation that identifies which bitfields in a register will be set or cleared by a BOR, POR, or PUC. The preface in the microcontroller's user guide shows a table that describes all the notations and their meanings, while an example of it can be found on page 48.

An explanation of every register's initial state is beyond the scope of this book, so only an overview and the most significant characteristics of each reset phase are presented here. For complete information, refer to the individual register tables, which are published by the microcontroller's user guide.

---

### **Brownout Reset (BOR)**

The first generation of MSP430 microcontrollers did not have a brownout reset (BOR) phase, at least not explicitly published by their user guides. Later generations all explicitly come with this reset subsystem.

[REDACTED]


Three types of events will produce a [REDACTED]

the




set of events which will cause a BOR is typically the largest, but not by very much. And the latest generation of microcontrollers will set a flag to indicate the event.

The type of events which will produce a BOR



As an aid for programmatically handling those events,



The set of bitfields which the BOR will initialize are also limited. In models which have a richer set of features, the BOR subsystem will, for example, configure registers for the system control module, the power management module, the FRAM controller, and the memory protection unit.

When referring to a register table, the bitfield initial condition notation for a BOR is in the form of a zero or 1 placed in square brackets. For example, a [0] or [1] means the field is cleared or set by a BOR.




From a fault handling perspective, keep in mind that an overflowed watchdog timer will only restart the microcontroller at the PUC subsystem, so being able to use an instruction to force a restart at a BOR provides us with an alternate point to restart when a PUC does not clear an operating problem.

Once the BOR has finished its work, it produces a POR signal that tells the next reset subsystem to start.

---

### **Power-On Reset (POR)**

As compared to the BOR and PUC, the set of events which will cause a POR is typically the smallest. Current generations of microcontrollers only have

In contrast to a BOR, a POR will configure a significantly larger set of registers to a POR state. When reading the register diagram, the POR initial condition notation will appear as a zero or 1 in parenthesis, for example, (0) or (1), means it is cleared or set by the POR subsystem.

A watchdog timer overflow will not trigger a POR. It will only cause a PUC. Therefore, using an instruction to force a restart at the POR or a BOR provides us with an alternate point to restart when a PUC does not resolve an operating problem.


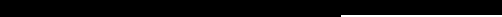
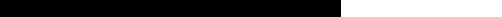
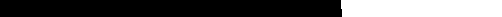
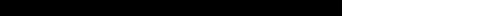
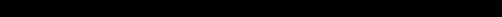
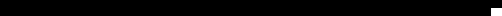
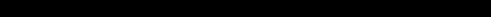
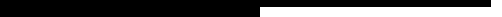
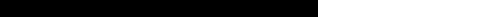


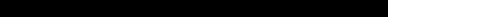
Once the POR has finished its work, it produces a PUC signal that tells that reset subsystem to start.


---

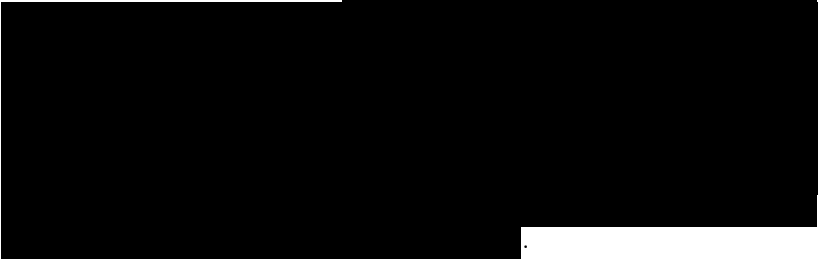
### Power-Up Clear (PUC)

The PUC is the , and it will configure many registers to a PUC state.

These are the events which will cause a PUC to occur:

- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

Here is what those events mean. 



Although the memory address space for an MSP430 ranges from zero to 65,536 (and higher for the CPUX), that does not mean the entire space is filled with memory. Microcontrollers come with different amounts of memory. Some come with large amounts of memory while others come with small amounts of memory. Therefore, if the program reads from any address which is vacant of memory, the value 3FFFh (16,383) is returned. Or if the program executes an instruction which fetches an address from vacant memory or a segment of protected memory, that causes a PUC NMI, also called a fetch from a peripheral area [in memory]. A fetch is an Assembly language term for an instruction that redirects the flow of execution from the next instruction in the flow to another point in the program. In the C language a decision or the result of a decision is an abstraction of the fetch.

The last type of event which will produce a PUC is caused by the FRAM controller. FRAM is nonvolatile memory, and it is used for storing the program. It is able to consume less energy than conventional nonvolatile memory technologies, such as

SRAM. However, one of its characteristics requires that when an address is read, its data must be written back into the address. The writing work is done automatically by the FRAM controller, and it includes a process which detects uncorrectable writing errors. If such an error occurs, it is called an uncorrectable FRAM bit error, and that error produces a PUC.

One of the most important initializations which the PUC makes involves the

Also of significance during a PUC is what it does to the reset ( $\overline{\text{RST}}/\text{NMI}$ ) pin, the digital I/O pins, and the CPU status register. The reset pin is

The register initial condition notation is simple for this phase. To show that a bitfield is configured by the PUC, the bitfield initial condition is just simply denoted with a zero or 1, without any punctuation.

A single interrupt flag is typically associated with a PUC, but that can change with later generations of the MSP430. That flag is set to indicate a watchdog timer overflow. The PUC does not clear it, nor does anything else, so we must include an instruction somewhere in order to clear it, typically inside of an ISR. To determine if other flags are associated with a PUC or set by it, see the operating mode diagram or the reset circuit logic diagram in the "System Reset and Initialization" chapter or section of the microcontroller's user guide.

When the PUC has finished its work, it loads the memory address to the first instruction of the boot program into the CPU program counter register (line 10 of diagram 43 on page 128), and then the microcontroller is released to the active operating mode (line 11). The program counter tells the CPU which instruction (address) in the firmware image to execute next. With the program counter loaded, and the microcontroller in active operating mode, the CPU starts to execute our firmware at the boot program.

---

### **MSP-BSL and boot.c**


The typical MSP430 has two programs which include the word boot in its name, and that can lead to confusion. So before going any further with the event-driven pattern, a brief explanation about what distinguishes one program from the other is in order. The first program is called MSP-BSL, and the second program is called the Boot Program.

---


## MSP-BSL

This program's name is technically an acronym for the Multi-Signal Processor Bootstrap Loader (MSP-BSL), but user guides and data sheets will typically call it something else. And that is where the confusion may occur. It was originally called Bootstrap Loader, but now it is often called the Bootloader, the BSL, or the Boot Code. This book will call it MSP-BSL. During MSP430 production, the factory loads a copy of MSP-BSL into a dedicated section in main memory, and it stays there permanently.

Its purpose is to



Tools are needed for sending commands and data, such as a firmware image, to the MSP-BSL program. The tools connect to



And finally, there is a BSL program for models of the MSP430 which are built of Flash memory and another one which is for models of the MSP430 which are built of FRAM memory. Further explanations about the MSP-BSL are beyond the scope of this book. But for more information, use the following sources of information.

- MSP-BSL home page at [www.ti.com/tool/mspbsl](http://www.ti.com/tool/mspbsl)
- USER'S GUIDE: MSP430 Flash Device Bootloader (BSL) - SLAU319
- USER'S GUIDE: MSP430 FRAM Device Bootloader (BSL) - SLAU550

---

## Boot Program

The Boot Program's purpose is to execute a set of instructions immediately before our `main()` program, as will be explained by the next section. Every time we use Code Composer Studio (CCS) to build and load our program into the microcontroller, it is automatically built into our firmware image. The name of this file is `boot.c`, and it's written in the C language, so we may edit it to include additional instructions.

There is one single `boot.c` template file that is shared among every MSP430 CCS project we create. In other words, when we build our `main()` program, CCS copies that file into our firmware image. The original `boot.c` file remains unchanged.

Now about its location. CCS has a subdirectory named `Compiler`. It contains subdirectories for different MSP430 compilers. Every time CCS is updated, it typically includes a later version of the MSP430 compiler, which is added to the compiler

directory. Older compilers are not deleted so we may choose to use them with a specific development project. Within each compiler directory is a subdirectory named SCR. The boot.c file is located in the SCR directory. Therefore, a single boot.c file is shared among all projects which use the same compiler version.

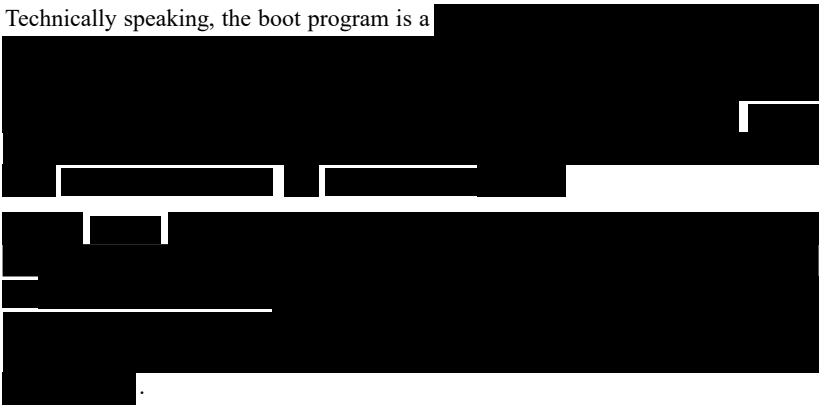
This book will refer to the boot.c file and the instructions it contains as the Boot Program.

---

## Boot Program Execution

The reset system completes its work with loading the address to the first instruction in the boot program into the program counter, and then the microcontroller is released to the active operating mode so the boot can be executed. At the end of the boot program is an instruction that calls the `main()` function so it can be executed.

Technically speaking, the boot program is a



The templates are actually sets of boot function parameters. During a build, the compiler chooses the appropriate set from among the templates. Then the set is placed into the parameter list of the boot function. That chosen set of parameters will be used to initialize the firmware image in a way which is appropriate for the image and microcontroller model. The choice is automatically made by the compiler, which is configured and set by the project build properties.

The properties are automatically made and configured when we create our firmware project. After the project build settings are configured, we may manually make adjustments to those settings, but that topic is beyond the scope of this discussion. However, it is explained by the “MSP430 Optimizing C-C++ Compiler User’s Guide” (SLAU132).

The chosen suffix for the boot function name represents the parameter template that was selected. When we put Code Composer Studio into debug mode (essentially loading the image into the microcontroller and running it), we can see that function’s name displayed in the Debug window pane. We can also find that name inside of the image by displaying the Disassembly window pane. As for the templates, they can be found in the boot.c file.

Let's now take a closer look at the sequence of routines in the boot function.

---

### Initializing the Program Execution Stack

The first routine in the boot function (line 13 of diagram 43 on page 128), will initialize the program execution stack. The stack is an essential data structure needed by the program in order to

The stack is automatically created by the MSP430 compiler, and the top of it is typically placed at the highest address in RAM.

---

### Initializing the Memory Protection Unit

After initializing the stack, and if the microcontroller includes a memory protection unit (MPU), then the MPU is automatically initialized (line 14 of diagram 43 on page 128). The purpose of the MPU is to protect the FRAM sections of main memory. Registers are used for configuring the MPU to divide the FRAM into variable sized segments, and then to place access control over those segments. For example, read, write, and execute access control over the instructions and data in each segment. You would want to use the MPU if your program contains intellectual property which you want to hide and protect.

---

### Execute a Pre-Initialization Function

If a pre-initialization boot hook function has been defined, then at line 15 it is executed. That function name is `_system_pre_init()`, and it is defined as a preprocessing translation unit at line 40.

Pre-initialization means

Here is the rationale. If our program has man

To mitigate that scenario, we insert an instruction into this pre-initialization boot hook function that will put the timer on hold or setup the timer to use an appropriate interval.

---

## Initialize Global Variables

As we know, a global storage variable is class of data storage in the C language. It is simply created by writing an instruction that declares an identifier as a specified type of variable, but the instruction must be outside of any function, meaning, outside of the `main()` function and all global functions. Having that global scope allows the variable to retain its value throughout the execution of the program.

The compiler places storage variables in a section of addresses located in volatile main memory, so data in that section is lost when the microcontroller is off. Off is not to be confused with most low power operating modes which do not remove power from the memory module. However, the fractional low power modes (LPMx.5) do remove power from main memory.

So the initialization of global variables (at line 16 of diagram 43 on page 128) means that we put instructions in the boot which write data into those variables so they will be ready (initialized) with data before `main()` is called.

For global variables which are not assigned or initialized to a specific value, the compiler will [REDACTED].

If writing your firmware in the C++ language, all object constructors will also be initialized at this time.



---

## Call the `main()` Function

Line 18 is just simply a single instruction that calls the `main()` function. It is the last instruction in the boot program, so it tells the CPU to load the address to the first instruction of `main()` into the CPU program counter register so it will be executed.

---

## `main()` Function

The `main()` function starts at line 19. It represents the main sequence of routines in our program. All the routines and their sequence which form `main()` were explained earlier by the “Event-Driven Pattern of Program Development” on page 120, so they will not be repeated here.

---

## CPU Interruption

Events which occur at peripheral modules are what we develop the program around. They are the focus of the event-driven pattern. And keep in mind that the typical

event-driven scenario begins with the microcontroller in some low powered operating mode of sleep.

The event produces an input signal, and then on line 30 the signal is sensed and sets an interrupt flag. On line 31 of diagram 43 (page 128), the interrupt system loads the address to the first instruction of the proper interrupt service routine (ISR) into the CPU, and then the system puts the microcontroller into the active operating mode so the ISR can be executed. At line 32, the ISR is executed.

Inside of the ISR are routines which get data to make a decision, and then use the result of the decision to produce an output signal (line 32). The data is obtained by reading it from the input module's registers, and to possibly help with the decision, data is also read from storage variables. The ISR will also include routines for saving volatile data and for clearing the interrupt flag.

Also keep in mind

When finished with the ISR, the microcontroller is automatically put back into the operating mode from where it was interrupted. The actual ISR is defined later on line 42. For more information, this entire asynchronous sequence was introduced and elaborated on by the "Input Signal Sequence" on page 125.

---

## Preprocessing Translation Units

A preprocessing translation unit is code that we write in the C language, which is meant to be outside of the `main()` function, but it is referred to or used by one or more instructions from inside of `main()`. The units are written into the same file that contains the `main()` function. However, some types of units may refer to the actual translation units which are located in another file.

Translation units are handled

For example,

There are two significant types of translation units: directives and intrinsics. A preprocessor directive begins with a hash mark (#). An intrinsic function will begin with either one ( `_` ) or two ( `__` ) low lines, but later versions of the preprocessor will recognize some intrinsic functions without using the low lines. Intrinsic functions are only



recognized by the MSP430 preprocessor, since they are intrinsic to it and not part of the standard C language. Use the "MSP430 Optimizing C-C++ Compiler User's Guide" (SLAU132) to learn about all functions which are intrinsic to the MSP430.

To help explain the translation units shown by diagram 43, they are all listed after the `main()` function, but in practice, some will partially or completely precede it. For example, `#include` directives and function prototypes must be placed before `main()`, while function definitions are placed after `main()`.

We have two categories of translation units. The first category is the conventional translation units which are recognized by the standard C language compiler. The second category is the MSP430 translation units which are only recognized by the MSP430 C preprocessor.

---

### **#include Preprocessor Directives**

At line [redacted] of diagram 43 on page 128, is a preprocessing translation unit called an `#include` preprocessor directive. It causes a copy of the specified file to be included in place of the directive; for example, `#include <math.h>`. The type of header files which are discussed here are from the standard C and C++ language library of files, but they can also be programmer defined library files. They are also referred to as header files. The programmer type of file is one that we created to provide specialized portable code which can be included in any of our projects.

When Code Composer Studio (CCS) is installed, the installation process creates several [redacted]

CCS provides a Project Explorer window pane that organizes all our project files into individual project directories. Every project directory has an *includes* subdirectory that contains two links which point to those two subdirectories of libraries which are physically located in the root directory of CCS.

The first link points to the [redacted] subdirectory. Its purpose is to hold a mixture [redacted]

Within the project root directory, we may add our own custom written library header files as preprocessing units.

When writing an `#include` preprocessor directive, we have two different syntaxes.

**Code Example 30:** The different two syntaxes which we may use for including external files into a firmware image build.

---

```
1 [REDACTED] > //first syntax
2 [REDACTED] //second syntax
```

---

The difference between these is the location where the preprocessor begins searching for files to be included. If the filename is enclosed with angle brackets, the preprocessor starts looking in the *include* directory. If the filename is enclosed with quotation marks, the preprocessor starts looking in the project root directory.

---

## #define Preprocessor Directives

```
[REDACTED]
```

**Code Example 31:** The syntax for a `#define` directive that declares a constant value.

---

```
1 [REDACTED]
```

---

When this line appears in a code file, the preprocessor replaces all subsequent occurrences of the identifier with the *replacement-text*. For example, `#define PI 3.14159` will replace all occurrences of `PI` which follow the declaration with `3.14159`.

A macro is a custom identifier, like the symbolic constant, which will be replaced in our firmware program with the *replacement-text* before the program is compiled. But the *replacement-text* is code that processes data, it's not a constant. For example, the *replacement-text* could be an instruction that calculates the area of a circle. Macros can be defined with or without arguments which get passed into the instruction where the macro appears. When defining the macro, its identifier is appended with parentheses that enclose a parameter list which facilitates the passing of data into the instruction. Just like a typical C function.

**Code Example 32:** The syntax for a `#define` directive that declares a macro.

---

```
1 [REDACTED] // macro that defines a formula for a circle
2 [REDACTED] // macro in an instruction that assigns a value to the variable area
```

---



---

## Define Global Variables

At line [REDACTED] we may declare one or more global storage variables. When declared outside of the `main()` function and all other functions, which are called by instructions

inside of `main()`, these global variables will be available for use, or accessed, by every function in the program.

You may initialize a variable, having global scope or not, to a specific value, but if you don't, the compiler will automatically initialize it to zero.

---

## Define Conventional Functions

[REDACTED]

[REDACTED]

---

## MSP430 Translation Units

We have eight types of MSP430 translation units which interest us: [REDACTED]

[REDACTED]

---

### **#include <msp430.h>**

Within the library of MSP430 header files is `msp430.h`. It is distinguished from all the others, since it is essential for every development project. Its purpose is to act as a cross-reference between our firmware project and the base header file for the specific microcontroller model which we're developing code for.

Every model of MSP430 has its own base header file which is named [REDACTED] where the word [REDACTED] is replaced by the microcontroller's model number. It contains all the identifiers and their definitions which are unique to a specific model of MSP430. These identifiers, which are published by the user guides and data sheets, represent register variables, register bitfield masks, symbolic constants, and at least one other `#include` directive. That directive includes `intrinsics.h`, a file that defines all the MSP430 intrinsic functions. One of the functions in that file allows us to write interrupt service routines.

---

### **#include Specialized MSP430 Library Headers**

These header files will be written by either Texas Instruments or a third party. They typically have to be copied into our project where the `#include` directive can find them.

These libraries are typically part of a specialized [REDACTED]. Such libraries are written by Texas Instruments (TI), and they are made available through the TI Resource Explorer at [dev.ti.com](http://dev.ti.com). TI calls them software development kits (SDK). Third party kits can be found at the home page for the individual microcontroller or at a third party's web site.

---

**Define a Pre-initialization Boot Hook Function**

```
[REDACTED]
```

---

**Define a Post initialization Boot Hook Function**

```
[REDACTED]
```

---

**Define an Interrupt Service Routine (ISR)**

```
[REDACTED]
```

---

**Define #pragma Directives**

```
[REDACTED]
```



## Repetitive-Driven Programming Examples

The repetitive pattern is used for developing program code examples and program code for carrying out tests. It is not typically used for developing programs for commercial use. Two forms of the repetitive pattern are presented here: the sequential pattern and the selection pattern.

Both patterns are written entirely inside of the `main()` function. They begin with a sequence of instructions which configure and setup the microcontroller for operation. Following that sequence is a single repetitive loop of instructions. The loop itself is what characterizes the pattern as being repetitive. The instructions inside the loop are what distinguish the pattern as being sequential or selection. The sequential pattern does not depend on an input signal to produce an output signal, while the selection routine depends on an input signal to produce an output signal.

---

### Development Tools

All program examples are developed with Code Composer Studio. As for the hardware platform, an imaginary generic development kit will play the role. It will have a built-in power supply, microcontroller, peripheral devices, and peripheral interfacing circuits. Imagine it as a typical LaunchPad development kit that Texas Instruments offers. You should be able to use the example bitfield masks and register variables for programming any MSP430, which have the same system and peripheral modules, with little or no changes.

And finally, to help guide us, we use the development approach shown on page 103 and the reference model on page 110.

---

### Repetitive Sequential Pattern

LaunchPad development kits always have a program example loaded in them by the factory. It is in the form of this repetitive sequential pattern. It just simply repetitively flashes an LED. The example is called Blinky. A template of the pattern is first presented in pseudo code, and then the template is used as a guide for developing Blinky into a program written in C. For an elaborate introduction about this pattern, see page 118.

---

### Pseudo Code Template

Program code is a sequence of instructions which form an algorithm. Development of an algorithm often starts with using English words to describe what each instruction must do. Those words are called pseudo code. They are an abstraction of the program code that will be written in C. Shown here is a pseudo code template of the repetitive sequential pattern.

**Program Example 1:** Pseudo code template for using the repetitive sequential pattern.

---

```

1  ██████████
2  ██████████
3  {
4      /** SYSTEM SETUP **/
5          // ██████████
6          // ██████████
7          // ██████████
8          // ██████████
9          // ██████████
10         // ██████████
11         // ██████████
12
13        /** REPETITIVE SEQUENCE **/
14            // ██████████
15            // ██████████
16            // ██████████
17            // ██████████
18
19        ██████████ // Flow of execution never reaches this instruction
20    }
```

---

## Using a Repetitive Sequence to Produce an Output Signal

To collect the requirements we need for developing the program, we write a model use case, examine the interfacing circuits, and then examine the port channel pathway.

### Model Use Case

A model use case is a written explanation about the form and function of the hardware and firmware needed to flash an LED without an input signal. The model is conceptualized as just simply to flash an LED for approximately every second without depending on an input signal. The microcontroller has a built-in oscillator for producing a clock signal. A power-up or reset configures the signal to oscillate at very close to 1 megahertz.

The kit already has a power supply, an LED, and interfacing circuits, so only a program is needed. The program must set up the microcontroller and then execute a repetitive loop containing a sequence of instructions which toggle an output signal at a pin. A circuit interfaces the pin with an LED.

### Interfacing Circuit

Although we are not concerned with building any circuits, we need to learn about which pin on the microcontroller's case is connected to the LED. Knowing the pin number will help us determine the port channel, which in turn tells us which bitfields must be configured to enable and use the channel.

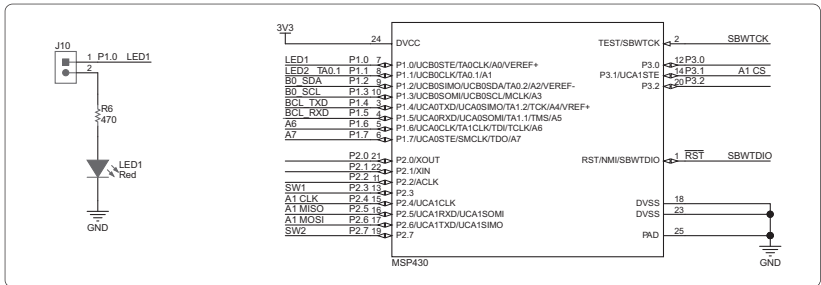
There are two ways to learn about the pin. ██████████

Diagram 45 shows a couple of circuits which are part of the schematic for our kit. The circuit on the left is the interface circuit for the LED, and the other is the micro-

controller's case with its pins, connections, and port functions. Together, the two images show that the interface circuit connects with pin 7 on the case and the output signal must flow through channel 0 of port 1 (P1.0).

The component J10 refers to a two pin strip header where a shunt jumper connects the pins. The jumper is removed when we want to use pin 7 with another interface circuit.

**Diagram 45:** Shown are parts of a schematic which is published by a typical user guide for an MSP430 microcontroller kit.



## Output Signal Pathway

After we know which port and channel the output signal will be using, then we determine which bitfields are needed for enabling and using a path through the channel. We use the port input/output diagram to get that information, and it is published by the microcontroller's data sheet. Shown by diagram 46 is the diagram for port 1.

The path through the channel is highlighted. The LED interfacing circuit and pin number are added, so they do not appear on the original port diagrams. Keep in mind that port I/O diagrams are generic in nature, meaning, a single diagram is used for all eight port channels. All the channels and their functions are listed at the lower right corner.

The solid squares in the diagram represent one or more bitfields in a register. The register variable with a suffix appears next to the bitfield. The suffix is an abstraction that represents the channel number; it is not part of the actual variable we use for reading and writing into the register. For example, for a bitfield labeled as P1DIR.x, the prefix P1DIR is the actual register variable name, and the suffix .x denotes some channel number in the port. In other words, the suffix denotes some bitfield channel number in the register, but it is not part of the actual variable name.

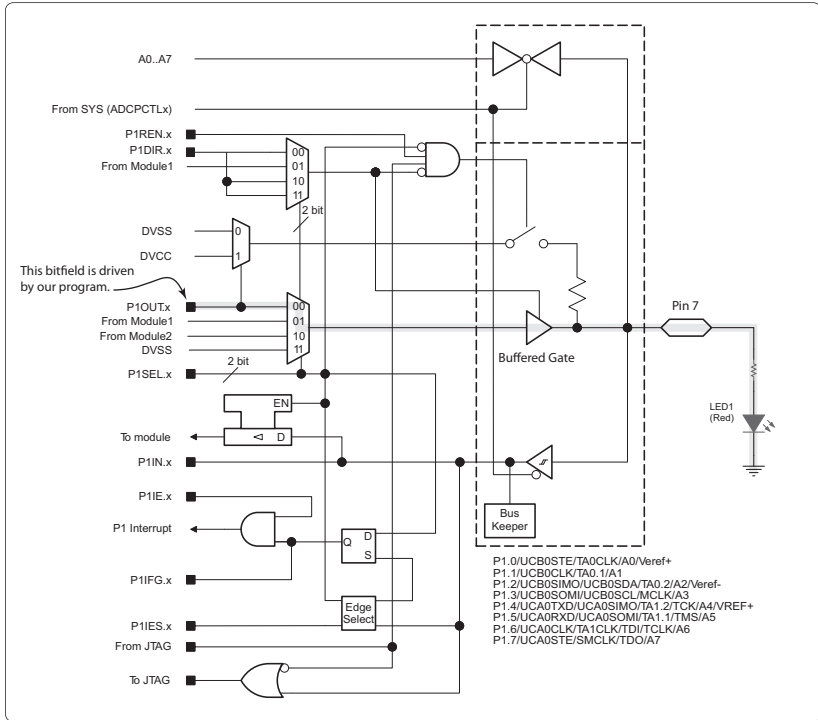
To elaborate on what was just said, each bitfield in a port register handles a specific channel. So for reading or writing into a port register, we only need the register variable and a standard bit (introduced on page 47). The bit is used as the mask for accessing the specific channel in the port register. For example, to set a bit in channel 0 of the port 1 direction register, we use the variable P1DIR with the standard bit BIT0.



Let's now take a closer look at the path and the bitfields needed for enabling it.

The path is highlighted. It starts at the port 1 output register (P1OUT). It specifically starts at the channel 0 bitfield of the register, as suggested by the suffix (.x). When an instruction sets a bit in that field, a high logic [voltage] level is placed on the channel.

**Diagram 46:** Port 1 input/output diagram with the highlighted output signal from channel 0 out to LED1. This is the typical diagram published by the microcontroller's data sheet. It is a generic view of a single port channel. All the channels which it represents and their functions are listed at the lower right corner. This diagram is also referred to as the functional diagram for the digital I/O modules in a port.



A zero or 1 bit signal flows into a channel function multiplexer. The multiplexer is used for switching between peripheral module services supplied to this same channel. It is controlled by two bits in the port 1 function selection register (P1SEL). As described by the microcontroller's user guide, those bitfields are cleared during a power-up or reset. Therefore,

. Furthermore,

. The two bits in that multiplexer are also cleared to 00, which switches that multiplexer to P1DIR.

The port 1 direction register is used for enabling the buffered gate. The buffer uses the zero or 1 from the P1OUT.x bitfield to decide on producing a low or high voltage signal which is sent out to the pin. The port 1 direction register table says that after a power-up or restart, its bitfields are cleared. Meaning, the buffer is closed and the

channel is configured as an input. So to enable the buffer, we'll need an instruction that sets a bit in field 0 of P1DIR.

After examining the path through the port 1 input/output diagram, we have determined that channel 0 of the P1DIR and P1OUT registers will have to be configured. We'll set a bit in the direction register to open the gate, and we'll toggle a bit in the output register to flash the output signal going to the LED.

---

## The Program

After we learn about which bitfields are needed for configuring port 1 and using channel 0, we can start writing instructions. We start with the block of system setup instructions and then write the block of repetitive routines.

**Program Example 2:** The repetitive sequential pattern in C that will flash an LED without depending on an input signal. It is based upon the pseudo code of program example 1.

---

```

1  ██████████
2  ██████████
3  {
4  // ** SYSTEM SETUP **/
5  ██████████ // Watchdog Timer Handler
6  ██████████ // Set P1.0 to output direction
7  ██████████ // Unlock Digital I/O Channels
8
9  // ** REPETITIVE SEQUENCE **/
10 ██████████ // Repetitive Loop - beginning
11 ██████████ // OUTPUT Servicing Routine: toggle P1.0
12 ██████████ // Intrinsic Function Delay Handler
13 } // Repetitive Loop - end
14
15 ██████████ // Never reached
16 } // End of ██████████

```

---

## Block of System Setup Instructions

There is very little to develop for setting up the system to prepare it for executing the repetitive block of instructions. We will need a watchdog timer handler for halting the timer, otherwise, it will force a system reset before the flow of program execution can properly run the program. Nor do we need the timer for this application. Therefore, we write an instruction, ██████████ that simultaneously writes the password (█████████ and timer hold (█████████ masks into the timer control register (█████████

The repetitive sequence depends on only one peripheral module. It's the digital I/O module, and it will be used for producing a periodic output signal on channel 0 of port 1 (P1.0). Diagram 46 is also referred to as the Functional Diagram for the Digital I/O Modules in a Port. The buffered gate must be enabled so signals can flow out the channel. In other words, the channel signaling direction must be configured to produce output signals. The instruction just simply involves setting a bit in the port 1 direction register (P1DIR). Since the direction is for channel 0, we use the standard bit BIT0 as shown by the instruction on line 6.

Most, if not all, microcontrollers come with a built-in oscillator which is used for producing a clock signal. It's called the reference oscillator (█████████ and its signal is fed

into a sort of timing signal conditioning block of logic called the digitally controlled oscillator (DCO). The reset system configures the DCO so it will produce a 1 MHz clock signal. We will take that into account when developing the delay handler.

Since an external oscillator signal will not be used, we do not need an oscillator settling handler. Nor do we need to configure any system modules. And since the reset system will configure the channel 0 output function multiplexer to use the port 1 output function (P1OUT), no output signal multiplexing instruction is needed.

Since this microcontroller is built of [REDACTED] the reset system locks all digital I/O channels to a high impedance state. This effectively turns off the channels so signals cannot flow in or out of the channels. But we need to use one of those channels. Therefore, on line [REDACTED] we use the [REDACTED] mask for clearing a bit in a power management control register ([REDACTED]) to unlock the channels.

And finally, since this is a non-critical application, any fault which will cause the reset system to run, is not of importance to us. Therefore, no [REDACTED] handler is needed.

### Block of Repetitive Sequential Instructions

A `while()` loop and the sequence of instructions inside of it characterize the behavior of the repetitive sequential pattern. The loop begins at line [REDACTED] and ends at line [REDACTED] of program example 2.

The loop uses a Boolean expression as a condition for making a decision. The expression is just simply the digit [REDACTED]. Therefore, the condition always evaluates to true, so the flow of execution is transferred into and always back into the loop.

The first of two sequential instructions is the output signal servicing instruction, shown on line [REDACTED].

It toggles bitfield 0 of the port 1 output register (P1OUT). That bitfield handles the signaling on channel 0. [REDACTED]

[REDACTED] Every time the instruction is executed, it will place onto or remove the signal from the channel, depending on the current state of the bitfield.

The second instruction, on line [REDACTED] is a delay handler in the form of an MSP430 intrinsic function. All this function does is force the CPU to count through an interval of clock cycles, nothing else. The length of the interval is configured by the [REDACTED] placed into the function's input parameter.

We want the delay to be [REDACTED]. Therefore, a little bit of research into the microcontroller's user guide will help us choose an interval number. The clock system chapter has a section that describes its operation. It says that a PUC configures the DCO to produce a 1 megahertz signal. So an interval of [REDACTED] cycles will create a delay of [REDACTED].

In reality, that interval will typically not produce exactly one second, but very close to it. For example, it might be 0.999 seconds. Adjusting the number of clock cycles, or trimming the DCO, or both can bring the delay to exactly one second. Clock registers can be used for trimming the DCO, while using measurement equipment and different amounts of cycles can bring the delay closer to one second. But there is a caveat. All oscillators produce small amounts of jitter every so often in the signal. That will make the period of the delay drift about a cycle every few seconds or so. Beyond the loop, on line 10 is a return statement. It will never be reached by the flow of execution.

---

## Repetitive Selection Pattern

This is the second type of repetitive pattern. Like the repetitive sequential pattern, it has two blocks of instructions. The first one prepares the microcontroller for executing the second block of repetitive instructions.

It is the patterns inside of the repetitive blocks which distinguish the sequential pattern from the selection pattern. Basically, the sequential pattern does not include a selection control structure, such as an `if()`, `if()...else`, or `switch()` control structure for producing an output signal. The selection pattern will depend on one of those structures to produce an output signal.

Two programming examples are presented. The first one uses an external input signal for producing an output signal that flashes an LED. The external signal is produced by a push button switch. The second example uses an internal input signal for producing an output signal that flashes an LED. The internal input signal is produced by a temperature sensor that is built into the microcontroller. Both program examples are designed and expected to run, with little or no changes, on the typical MSP430 development kit produced by Texas Instruments, such as the so called LaunchPad kits.

---

## Pseudo Code Template

Shown by program example 3 is our pseudo code template. It is derived from diagram 41, on page 119. It provides a framework for developing a repetitive selection routine in the C language.

---

## Using an External Input Signal for Selecting an Output Signal

To collect the requirements for developing this program, we write a model use case, examine the interfacing circuits, and then examine the port channel pathway.

---

## Model Use Case

This model is conceptualized as a power supply, button switch, microcontroller, LED, and their interfacing circuits which are all built into a single kit. When the switch is closed, the microcontroller will illuminate the LED. When the switch is

opened, the LED goes dark. A power-up or reset will automatically configure the clock signal to run at 1 Mhz.

**Program Example 3:** Pseudo code template for using the repetitive selection pattern.

```

1 #
2
3 {
4     /** SYSTEM SETUP **/
5     //
6     //
7     //
8
9     /** REPETITIVE SELECTION **/
10    //
11    //
12    //
13    //
14    //
15    //
16    //
17    //
18
19    // Flow of execution never reaches this instruction
20 }

```

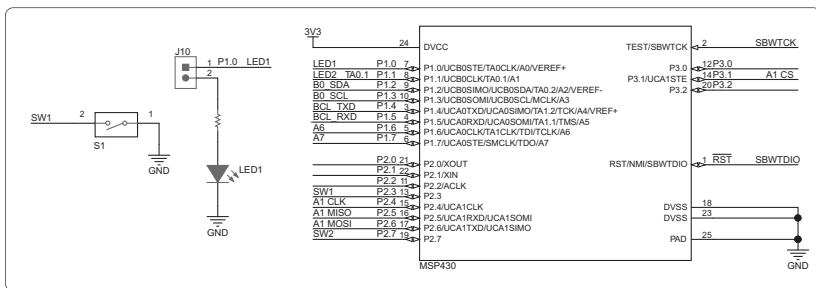
## Interfacing Circuits

Although we will not be developing any interfacing circuits, we need to use our kit's schematics to learn about which port channels the switch and LED will be using. Diagram 47 shows those schematics.

We'll be using switch SW1. Notice that it is a single-pole single-throw (SPST) type of switch. Terminal 1 is connected to ground, while terminal 2 is connected to pin 13 of the microcontroller, and the schematics show that pin 13 is serviced by channel 3 of port 2 (P2.3). The physical switch is in the form of a tactile button. It will be used for producing an input signal.

The LED interface circuit is the same circuit used in the last programming example. Therefore, channel 0 of port 1 (P1.0) will be used for driving that LED.

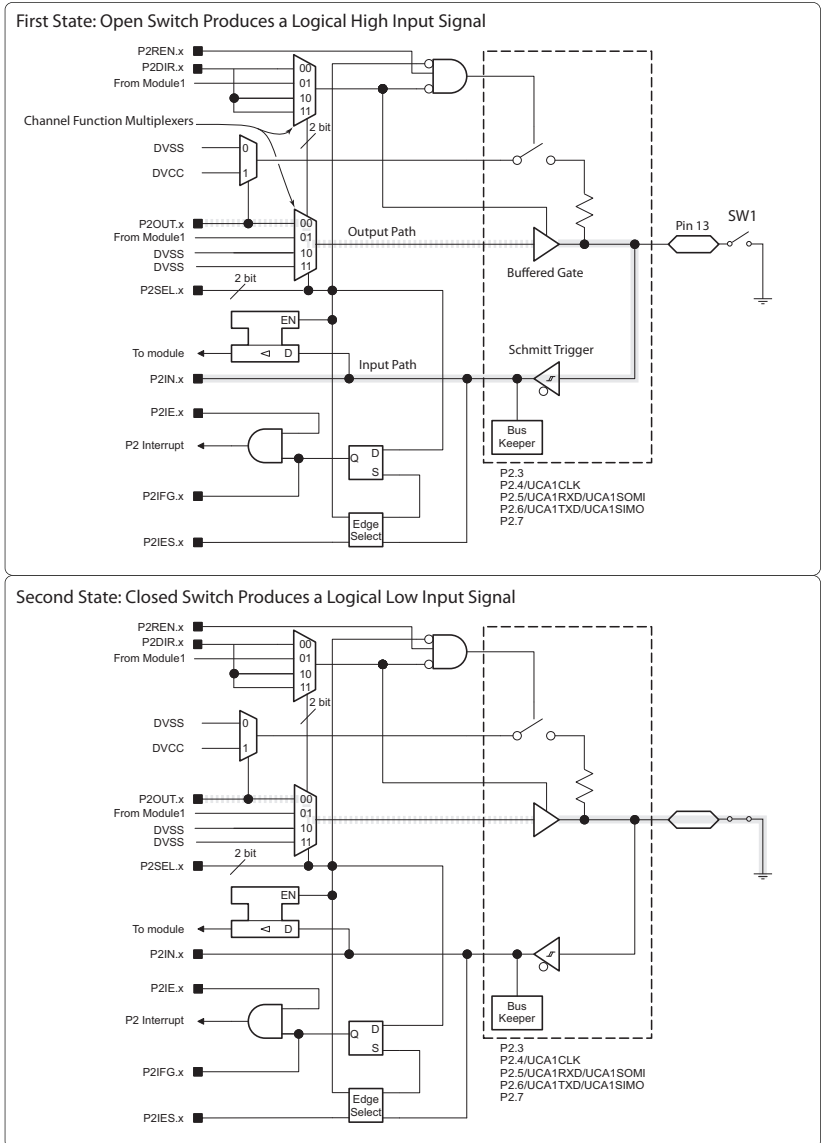
**Diagram 47:** LED and switch Interfacing circuits.



## Switch Circuit and Its Operation

Diagram 48 shows two images of the port 2 input/output diagram. It is published by the microcontroller's data sheet. These are slightly modified with additional labels and highlighting to help illustrate the circuit's operation.

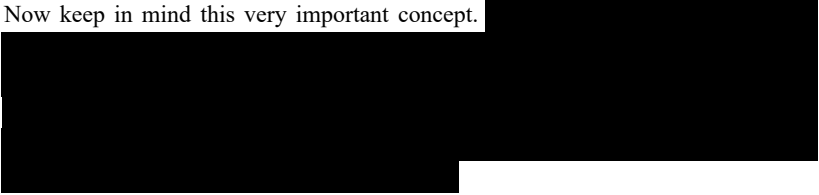
**Diagram 48:** Port 2 input/output diagrams showing the highlighted input signal path handled by SW1.



To understand the operation of the switch circuit, we must visualize both of its operating states. The top image shows the state of the circuit when the switch is open and producing a high input signal. The bottom image shows the state when it is closed and producing a low input signal.



Now keep in mind this very important concept.



### **First State Operation: Switch is Open**

In the first state, when switch SW1 is open, the circuit is producing a logical high input signal. A high signal is produced at P2OUT.3, and a high signal is sensed at P2IN.3. Five conditions must be present for that to happen.



The lower multiplexer allows signals to flow from P2OUT.3 to the buffered gate, while the upper multiplexer allows a logical high signal to flow from P2DIR.3 to the gate and enable it. The gate allows channel signals to flow out towards the pin. But the switch is open, so instead, the signals flow down to the Schmitt Trigger. That trigger is a block of logic that distinguishes the signal as being a logical high or low signal.



Setting a bit in the channel 3 bitfield of the port direction register (P2DIR), produces the third condition. The result sends a high signal to the gate.

Setting a bit in the channel 3 bitfield of the port 2 output register (P2OUT), produces the fourth condition. It places a high signal on the output path of channel.

An open switch creates the fifth and last condition. After those five conditions are created, channel 3 of the port 2 input register (P2IN) can be read to determine the input signal. In this state, with the switch open, the signal will be high.

## Second State Operation: Switch is Closed

In the second state, when switch SW1 is closed, the circuit is producing a logical low input signal. Meaning, a high signal is still produced at P2OUT.3, but a low input signal is now sensed at P2IN.3. Five conditions must be present for that to happen.

The first four conditions are identical to those of the first state.

When the switch is closed, the voltage signal flows out to the pin and into the ground instead of flowing back onto the input path to the Schmitt Trigger. Absence of voltage at the trigger produces a logical low signal that clears the channel 3 bitfield in the port 2 input register (P2IN). That bitfield then can be read to determine the input signal.

## LED Circuit

Since the LED interface circuit is the same one as used for the previous programming example, the path through port 1 is the same. For those details, see “Output Signal Pathway” on page 151.

## The Program

After we learn about which bitfields are needed for configuring channels P1.0 and P2.3, we can start writing code. We start with the system setup instructions and then write the repetitive routines.

**Program Example 4:** Code for using the repetitive selection pattern.

```

1
2
3 {
4   /** SYSTEM SETUP **/
5   // Disable the watchdog timer
6   // Set P2.3 direction outward to SW1
7   // Set P1.0 direction outward to LED1
8   // Clear signal at P1.0 to darken LED1
9   // Unlock the digital I/O channels
10
11  /** REPETITIVE SELECTION **/
12  // Repetitive loop - beginning
13  // Input Polling handler: read P2 inputs
14  // If input is 1, SW1 is closed, then
15  //   // 1st Output Svc routine: LED on
16  // Else, SW1 is open
17  //   // 2nd Output Svs routine: LED off
18  } // Repetitive loop - end
19
20 //
21 }
```



## Block of System Setup Instructions

Setting up the system to prepare it for the repetitive block of instructions is just simply on elaboration on program example 2 on page 153. Only two additional instructions are needed. One configures the signal input channel for switch SW1, and the other assures that the LED is initialized to a darkened state.

\_\_\_\_\_

\_\_\_\_\_).

Now to be organized, we use a systematic approach for developing the setup instructions. Meaning, we write instructions which first setup the input signal path, and then we write instructions which setup the output signal path.

The signal input is handled by channel 3 of port 2 (P2.3). We use \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_ When closed, it's removed.

The reset system has already setup most of the channel's output path, while the input path needs no setup. The output path only \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_). That bitfield handles channel 3.

For the output signal that flashes the LED, it depends on the same module that was used by program example 2 on page 153. It's the digital I/O module (as shown by diagram 46 on page 152), and it will be used for producing a periodic output signal on channel 0 of port 1 (P1.0). \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_ 7.

Since the reset system will configure the channel 0 output function multiplexer to use the port 1 output function (P1OUT), no output signal multiplexing instruction is needed.

Although not necessary, we use an instruction for initializing the output signal to the LED. We want to assure that no signal is produced before the repetition block is entered. So on line 8, an instruction uses the standard bit BIT0 to clear field 0 in the port 1 output register (P1OUT).

This microcontroller is built of \_\_\_\_\_ so a reset locks all digital I/O channels to a high impedance state. But we need to use two of those channels. Therefore, on line 7 we use the LOCKLPM5 mask for clearing a bit in a power management control register (PM5CTL0) to unlock the channels.

There is really no other set up needed. The reset system configures the digitally controlled oscillator (DCO) to produce a 1 MHz clock signal. Since an external oscillator

signal will not be used, we do not need an oscillator settling handler. Nor do we need to configure any system modules. And finally, since this is a non-critical application, any fault which will cause the reset system to run, is not of importance to us. Therefore, no reset fault handler is needed.

### Block of Repetitive Selection Instructions

A `while()` loop and the selection instructions inside of it characterize the behavior of the repetitive selective pattern. The loop begins at [REDACTED] and ends at [REDACTED] as shown by program example 4 on page 159.

The loop uses a Boolean expression as a condition for making a decision. The expression is the digit 1. Therefore, the condition always evaluates to true, so the flow of execution is transferred into and always back into the loop.

Inside of the loop, [REDACTED]

At line [REDACTED] the [REDACTED] handler reads the entire port 2 input register (P2IN) and assigns it to a storage variable named `state`. Although eight bits are written into the variable, the input signal is in the form a bit in field 3 of the register.

The double selection structure, which is in the form of an [REDACTED]

The return instruction at line 20 is never reached because the flow of execution remains inside of the loop.

### Using an Internal Input Signal for Selecting an Output Signal

To collect the requirements we need for developing this program, we write a model use case, examine the interfacing circuits, and then examine the signal pathways. After we have that information, program development can begin.

#### Model Use Case

This model is physically conceptualized as a power supply, temperature sensor, microcontroller, two LEDs, and their interfacing circuits which are all built into a single kit. When the temperature is below 20°C, green LED1 illuminates while red LED2 goes dark. When the temperature is above 20°C, green LED1 goes dark while

red LED2 is illuminated. This code is not optimized, so when the temperature is 20°C, both LEDs will fluctuate. And finally, a power-up or reset will automatically configure the clock signal to run at 1 Mhz.

---

### Input Interfacing Circuit

A temperature sensor that is built into the microcontroller provides the input signal. The output of the sensor is directly connected to one of the ADC input channels. Therefore, no input circuit needs to be built.

In this case, the input channel is A12. The circuit is shown by diagram 49 on page 163, and it is explained later by the “Input Path from the Temperature Sensor” on page 164.

---

### Output Interfacing Circuits

Two LEDs will be driven by the microcontroller. A green one will be driven to illuminate when the temperature is below a specific temperature. A red LED will be driven to illuminate when the temperature is above a specific temperature. Since this project is built into a development kit, the LEDs and their interfacing circuits already exist.

By referring to the kit’s printing circuit board (PCB), or its user guide, we see that the red LED is labeled as LED1. It uses the same interfacing circuit as used by a project described earlier. That circuit is shown by diagram 45 on page page 151. It interconnects LED1 with channel 0 of port 1 (P1.0).

The green LED is labeled as LED2. LED1 and LED2 are using circuits of the exact same design, except that LED2 is connected to a different pin and port channel. It is connected to pin 8, and that pin is connected to channel 1 of port 1 (P1.1).

---

### Signal Pathways

The input signal will originate at the temperature sensor, located inside of or handled by the power management module (PMM), and it will end at the ADC conversion memory register (ADCMEM). The program uses the

The output signal begins at the digital I/O module and ends at one of the LEDs, depending on the result of the decision.

Diagram 50 shows a portion of the typical block diagram for a PMM as published by a user guide, and diagram 49 shows a typical ADC block diagram. The first diagram shows the thin highlighted path where the input signal originates and flows out of the PMM, while the second diagram shows the entire input path under a thick highlight.

Diagram 49: A typical ADC block diagram as published by a user guide.

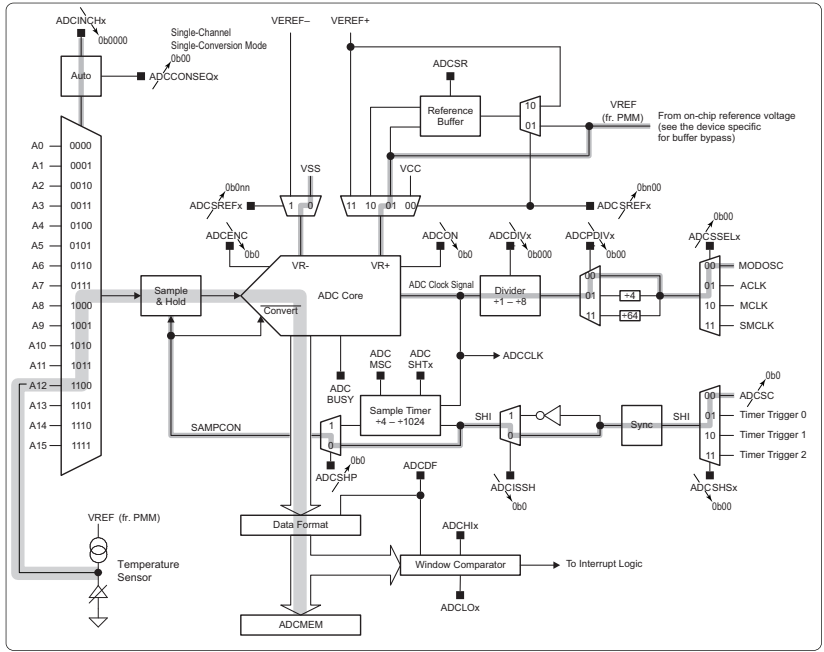


Diagram 50: This is a portion of a typical PMM block diagram as published by a user guide.

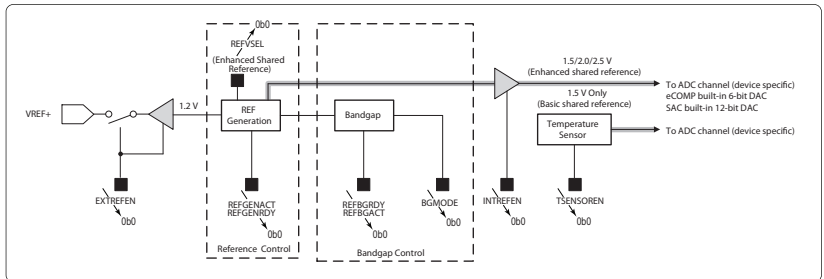


Diagram 46, on page 152, is of the typical digital I/O port block diagram as published by the microcontroller's data sheet. It shows the highlighted output signal path from

the port to one of the LEDs. Although this project drives two LEDs, that same diagram can be used for both LEDs by replacing the pin and LED numbers.

Those three preceding diagrams help us determine which bitfields will be of concern to us. They configure the paths so the signals can reach their destinations. That is a prerequisite for preparing the microcontroller to execute the block of repetitive instructions.

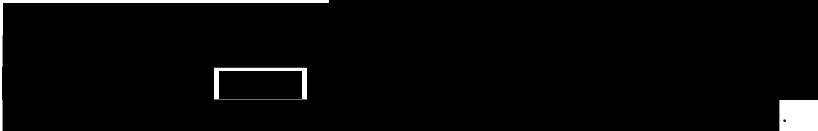


Although the register names are not shown, the initial state for every bitfield can be found by referring to their register's table, which is published by the microcontroller's user guide.

### Input Path from the Temperature Sensor

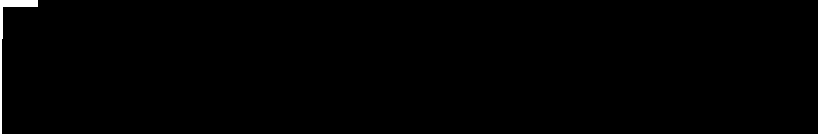
The sensor is typically built into or managed by the PMM. It's powered by the PMM, and its output is connected to an ADC input channel. The path has a PMM section and an ADC section which are of concern to us. Furthermore, we are concerned with the ADC control signals which carries out the sampling, measuring, and converting process. There are four controls signals which are also of concern to us.

**PMM Section.** We first begin by



The user guide also makes two statements which concerns us. The sensor must be energized by an internal voltage reference signal called VREF, and before writing to any PMM registers, they must be unlocked.

The path for the shared voltage reference is highlighted by diagram 50. It starts from the



For our microcontroller, it says that only the basic 1.5 volt shared reference is built into the PMM.

Six bitfields are used for producing that basic shared reference. After searching for the registers which they are in and reading about their control over the path, we learn that only one bitfield is of concern to us. It is the Enable Internal Reference Voltage (INTREFEN) field, in the PMMCTL2 register. It must be set (0b1) to enable the reference.

A bandgap control block of circuits is shown. It is type of power management integrated circuit technology. It uses the electrical bandgap in silicon to produce a fixed voltage during power supply variations, temperature changes, and circuit loadings.

Current generations of the PMM utilize a password for accessing their registers. Although one of its register tables will tell us that it is responsible for handling the password, so does the introductory paragraph to the section about PMM registers. Either way, we learn that most of those registers are password protected, including the ones we need to configure. That means an instruction which unlocks the registers must precede the PMM configuration instructions. To unlock the registers, a password mask is written into the proper register.

In this case, a sixteen bit register, named `PMMCTL0`, provides bitfields at the higher address where the password is written into. The introduction to the PMM registers also tells us about two options. We may use the 16-bit conventional register variable with the 16-bit password mask for setting bits in those upper fields, or we may use the higher 8-bits register variable, `PMMCTL0_H`, and assign the higher 8-bits password mask, `PMMPW_H`, to it. The operation is what distinguishes one instruction from the other. We'll be using the later technique.

**ADC Section: Input Signal Multiplexing.** The typical ADC block diagram, as shown by diagram 50, shows the entire temperature sensor signal path into the ADC. The internal shared voltage reference (VREF) energizes the sensor. The output from the sensor is connected to input channel A12 where the multiplexer must switch that channel to the sample and hold block.

For making that switch, two bitfields are involved. The first one is called

In this case, `ADCINCHx` is a

As for `ADCCONSEQx`, it is a mask in ADC Control Register 1 (`ADCCCTL1`), and its table says the initial state is `0b00`, which is the Single-Channel Single-Conversion Sequence mode. We want that mode, so those bitfields can be used as is.

A signal called

The second period converts the measurement to a binary number, and then places the number into the ADC Conversion Memory register (`ADCMEM`). When the signal is logically high, the entire process begins. When the signal is driven low, the first period is stopped, and the second period starts, and it ends automatically with the data written into `ADCMEM`. The

single-channel single-conversion sequence mode directs the ADC to carry out the process for one channel and do it just once.

## ADC Control Signals

These signals control the operation of the ADC. We are concerned with four types of signals.

### Voltage Measurement Scale.

For the lower endpoint, diagram 49, on page page 163, shows a highlighted signal path from VSS to VR-, and for the upper endpoint, it shows a path from VREF to VR+. Two multiplexors handle the switching along those paths, and both are configured with the 3-bit ADC Select Reference mask (ADCSREFx). The first bit of that mask (0b0nn) configures the VR- multiplexor, while the last two bits (0bn00) will configure the VR+ multiplexor.

By searching in the user guide for that mask name, we can find its register table. In this case, it is ADCMEMCTL0, the same register where ADCINCHx is located.

Although the register table shows the options for configuring the endpoints, it does not show us the mask suffix for each option (ADCSREFx). It can be found in the microcontroller's header file as ADCSREF\_1.


Keep in mind that this is a 10-bit ADC. That means


**Clock Conversion Signal.** A clock signal is needed for driving the ADC to work. It's called the clock conversion signal.

The source for this signal is the built-in Module Clock (MODCLK) oscillator. According to the user guide, it oscillates at a frequency of 5 Megahertz. Furthermore, it does not get divided before entering the ADC core.


**SAMPCON and SHI Signals.** Recall that the SAMPCON signal tells the ADC to carry out a sample, measure, and convert process. The process is divided into a period for sampling and measuring and a period for conversion. When the signal is

driven high, the first period is started. When the signal is driven low, the first period is stopped, and the second period automatically starts and ends with the conversion written into the ADCMEM register.

SAMPCON is actually driven by 

As shown by diagram 49, the SAMPCON-SHI signal has two sources and two destinations. One source is the ADC Sample-and-Hold Source Select multiplexor. It is used for selecting between the ADC Start Conversion (ADCSC) bitfield and three different timer module triggers. 

As for the two SAMPCON-SHI destinations, one goes into the Sample and Hold block and the other goes into the ADC Core. They simultaneously tell those two blocks to start and stop the sample and measuring period.

The highlighted signal path from the source select multiplexor to the sample and hold block is the 

Since the default configuration for the SAMPCON-SHI signal path can be used as is, no configuration instructions need to be developed.

**ADCENC and ADCON Signals.** Two signals were added to the ADC block diagram, as shown by diagram 49 on page 163. They do not appear in the original. One comes from the ADC Enable Conversion bitfield (ADCENC), and the other comes from the ADC On bitfield (ADCON). In this case, both are located in the ADCCTL0 register.

ADCON just simply turns on the ADC, while ADCENC has two purposes. The first one enables or allows the ADC to work. The second purpose locks most of the ADC register so they cannot be changed.

After a power-up or reset, ADCENC is cleared, so ADC registers are unlocked, but the ADC is not enabled to work. Therefore, setting a bit in this field to enable the ADC and lock ADC registers is typically the last ADC configuration instruction. The ADCMEM register where the converted data is written into is not affected by ADCENC.

### Output Paths to LED1 and LED2

There are two paths. One from P1.0 to LED1, and the other is from P1.1 to LED2. Both paths originate at a port channel and end at an LED. The paths are the same, except where they originate and terminate are different.

The signal path from P1.0 to LED1 is shown by diagram 46 on page 152. Bitfield 0 (**BIT0**) of the port 1 output register (P1OUT) is used for driving the signal. The path to



LED2 is exactly the same, except bitfield 1 (BIT1) of the same port is used for driving that signal. BIT1 and BIT2 are two of the standard masks used for manipulating bits in port registers.

---

## The Program

As shown by code example 33, two basic blocks of program code are used for developing this program. The first block (lines 3 to 25) is made of system setup instructions. It prepares the microcontroller for executing the next block of instructions. That next is block (lines 28 to 43) is called the repetitive selection pattern.

The repetitive selection pattern is in the form of an infinite loop. It gets the raw analog sample from the sensor output, measures it, converts it to a digital number, and then writes the number to the ADC conversion memory register. The number is then converted to degrees Celsius, and then used for making a decision. If the number (the temperature) is below 20°C, then the green LED is on. If it is above 20°C, then the red LED is on. The flow of program execution then goes back to the beginning of the repetitive pattern and executes it again.

---


### Block of System Setup Instructions




Four modules must be setup to prepare the microcontroller for executing the repetitive block of instructions. They are the watchdog timer, the power management module (PMM), the analog to digital converter (ADC), and the digital I/O module.

**Write a Watchdog Timer Handler.** The watchdog timer will not be needed, so it will be turned off, as shown at line 3.

**Setup the Sensor to Create Input Signals.** The temperature sensor is handled by the PMM. Four instructions are needed for setting up the PMM. The first instruction, on line 6, assigns the higher eight bits password (PMPW\_H) to the higher eight bits of the PMM Control 0 register (PMMCTL0\_H). That allows the following instructions to write into their respective PMM registers,

As a reminder, 

The second instruction, on line  enables the PMM's shared voltage reference (VREF). That instruction uses the Internal Reference Enable mask (INTREFEN) for setting bits in the PMMCTL2 register to enable the reference.

The third instruction, on  enables the temperature sensor. The mask  is used for setting the sensor enabling bits in the same register (.

**Code Example 33:** Using the repetitive selection pattern for developing a program where the internal temperature sensor creates an input signal and a decision is made about selecting between two output signals.

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45 }

```

On line 9 is an MSP430 intrinsic function that produces a four hundred clock cycle delay in the flow of execution. Apparently, the sensor may need some time to settle after it is enabled, but that specification could not be found by this author. So it may not be needed. But since it was used in some programming examples produced by Texas Instruments, it is also used here. It ends the instructions needed for configuring the PMM.

**Setup the ADC as the Input Signal Peripheral Module.** Four instructions are needed for setting up the ADC. The first one, on line 12, switches the ADC input


multiplexer to channel A12, since the temperature sensor output is connected to that channel. The mask `ADCINCH_12` is used for setting the appropriate bits in ADC Memory Control 0 register (`ADCMCTL0`).

As a reminder, the register table for `ADCMCTL0` tells us that the mask `ADCINCHx` is used for selecting a specific channel, but it does not give us the suffix `x`. To get the exact mask name which includes the suffix, we search for it in the microcontroller's header file. While in Code Composer Studio, the file is opened by typing the register variable name into the program code file, selecting the variable, right mouse button click to open a popup menu, and then select **Open Declaration** to get the file. Near the `ADCMCTL0` declaration will be the declarations and definitions for all of its masks. It shows that `ADCINCH_12` is the mask for channel 12.

The second instruction, on line 13, configures the ADC analog voltage measurement scale to use specific voltages for its endpoints. The scale is used for measuring the temperature samples (in volts). We want the lower endpoint (`VR-`) to use `VSS` (ground), and the upper endpoint (`VR+`) to use `VREF`, the shared voltage reference produced by the PMM. We want to use those endpoints because the sensor output will swing between those two points.

The `ADCMCTL0` register is used for configuring those two references, and its table tells us to use the mask `ADCSREFx`. Once again, we need the proper suffix to replace `x`. The header file, along with the register table, tells us that the mask `ADCSREF_1` can be used for simultaneously selecting `VSS` for `VR-` and `VREF` for `VR+`. Therefore, as shown by line 13, we use `ADCSREF_1` to set the proper bits in `ADCMCTL0`.

The last two instructions will



**Setup the Digital I/O as the Output Signal Peripheral Module.** The last module to be configured is the digital I/O. It will be used for creating two channels where output signals will flow across to drive the LEDs. Both channels will be in port 1. Channel 0 will drive LED1, while channel 1 will drive LED2.

On line 18, channel 0 (`BIT0`) of the port 1 direction register (`P1DIR`) is set to the signal output direction. On the following line, channel 1 (`BIT1`) of the same register is also set to the output direction. Notice that we use the standard bits as masks.

On line 20 is yet another unlocking instruction. Current generations of MSP430 microcontrollers must have their port channels turned on before use. This is the typical instruction used for unlocking the channels.

**Declare Variables and Constants.** The block of repetitive instructions will need a couple of variables and a constant.

The first instruction, on line 23, declares the variable [REDACTED] as storing integer data. Integers are expressed with sixteen bit binary numbers. The 10-bit measurements, which are written into the ADC conversion memory register, will be copied into this variable.

The second instruction, on line 24, uses a pointer operation for reading a number from an address in main memory and assigns it to a constant named [REDACTED]. That address does not have a register variable for it. The data at that address is a sixteen bit number that will be needed when the program converts the measurement from millivolts to degrees Celsius.

To write the instruction shown on line 24, we depend on knowing about the ADC Calibration Transfer Function, the ADC Device Descriptors, and a pointer operation. The following paragraphs will cover those concepts.

A transfer function, in the form of a linear equation, will be used for [REDACTED]

One number adjusts the calculation for measurements at the lower end of the measurement range, and the other is applied to the upper end of the range. The lower adjustment applies to measurements around 30°C, while the upper applies to measurements around 85°C.

The microcontroller's data sheet publishes a table showing the addresses to those adjustments. That table is called the Device Descriptors table, and it categorizes those adjustments as ADC Calibration device descriptors. A device descriptor is one or more bytes of data that provides information about a specific microcontroller characteristic. There are many descriptors, but we are only interested in those used for adjusting ADC measurements. Those two descriptors are called the ADC 1.5 Reference Temperature descriptors. That section of the table is shown by diagram 51. There is an adjustment for handling measurements around 30°C, and there is one for handling measurements around 80°C. We're interested in the 30°C adjustment.

**Diagram 51:** The ADC calibration section of the microcontroller's device description table. It is published by the microcontroller's data sheet.

ADC Calibration	ADC calibration tag	1A14h	Per unit
	ADC calibration length	1A15h	Per unit
	ADC gain factor	1A16h	Per unit
		1A17h	Per unit
	ADC offset	1A18h	Per unit
		1A19h	Per unit
	ADC 1.5-V reference temperature 30°C	1A1Ah	Per unit
		1A1Bh	Per unit
	ADC 1.5-V reference temperature 85°C	1A1Ch	Per unit
		1A1Dh	Per unit

There are two addresses for the 30 degree constant, but the data sheet does not say which address to use. Both addresses are needed. Here is the rational. The ADC converts its measurements into ten bit numbers. Since the measurement will never be

wider than ten bits, the constant which adjusts the measurement will be ten bits wide or narrower. And since each address in memory is eight bits wide, two addresses are needed for storing the largest constant, which will be ten bits wide.

Let's now go back to the instruction on line 24. It

. This pointer operation is explained by code example 16 on page 83.

Now for the last variable, the transfer function converts the reading from . That variable is used for making a decision on line 35. Therefore, on line 25, that variable is declared as a floating point number named Temp. We use that type of data because the result will be expressed in decimal fractions. For example, the result might be calculated to 20.25, and that is of course in degrees Celsius.

### Block of Repetitive Selection Instructions

The repetitive block of instructions occupies lines 28 through 43. It reads the raw temperature sensor output in millivolts, converts millivolts to Celsius, and then uses the result to make a decision about which LED to illuminate. This entire block is contained within a `while()` iteration statement. Its condition expression is just simply the number 1, which always evaluates to a Boolean value of true. Therefore, it creates an infinite loop.

On line 29 is the first instruction in this pattern. It sets a bit in the ADC Start Conversion bitfield to start the sample-measurement period. On line 31 that bitfield is cleared to stop that period. In other words, when ADCSC is set, a sample is held and measured. When ADCSC is cleared, the sample-measurement process is stopped.

That one cycle, from low to high and then back to low, is called a sample and measurement period, or just simply the sampling period. Directly after the sampling period, the ADC automatically enters the conversion cycle.

Following those start and stop instructions are delays which allows the ADC enough time to carry out the sample-measure and then conversion processes. Here is how we calculate those delays.

Let's start with the first delay shown on line 30. It is used for allowing the ADC enough time for measuring the sample. The measurement is carried out by a technique called successive approximation register (SAR). It is basically a trial and error measuring technique that converges upon the measurement after a few ADC clock cycles. To calculate the number of clock cycles needed for a proper delay, we need to take into account the number of cycles the sampling period needs, whether the sensor

may create an addition delay, which clock signal is driving the ADC, and whether any ADC dividing circuits are slowing down the clock signal.

The ADC “Sample and Conversion Timing” section of the user guide says that when the ADC is in 10-bit conversion mode, it needs 4 ADC clock cycles to sample and measure an ADC input signal. A later section, about “Using the Integrated Temperature Sensor,” says that when sampling and measuring the sensor output, the sampling period must be greater than 30 microseconds ( $\mu\text{s}$ ). Diagram 49, on page 163, shows the clock signal going into the back of the ADC core. At power-up or reset, the ADC is configured to be driven by the Module Oscillator (MODOSC), and the signal is not divided anywhere to slow it down.



Here's where that 1 MHz comes from. During power-up or a reset, the reset system selects the digitally controlled oscillator (DCO) as the clock system module's timing source signal and adjusts it to produce a 1 MHz signal. That signal is then placed onto the master clock signal bus (MCLK). The CPU is connected to that bus, so it executes the program at that same frequency. Which means the `__delay_cycles()` intrinsic function will run at that speed.

The calculation for converting the 4 ADC clock cycles (needed for a conventional sampling period) to master clock cycles (MCLK) is shown by calculation 1. It uses the conversion factors 5 MHz for the ADC clock speed and 1 MHz for the master clock speed.

**Calculation 1:** For the conventional sampling period, this converts the 4 ADC cycles running at 5 MHz to master clock cycles (MCLK) running at 1 MHz.

$$4\text{ADC cycles} \times \frac{1\text{second}}{5000000\text{ADC cycles}} \times \frac{1000000\text{MCLK cycles}}{1\text{second}} = 0.8\text{MCLK cycles} \quad (1)$$

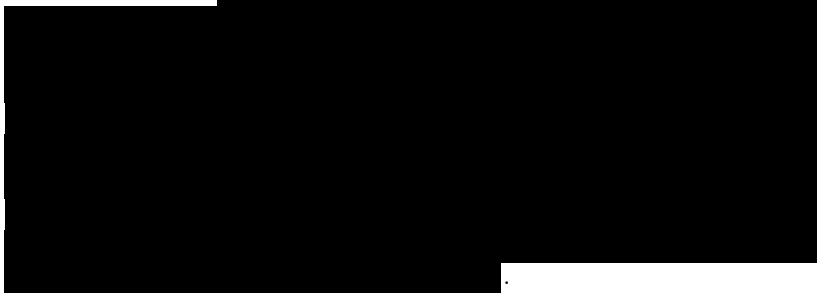
The calculation for converting the additional 30  $\mu\text{s}$  (needed for sampling the temperature sensor output) to master clock cycles is shown by calculation 2. It uses the conversion factors of 1 Hz for the 30  $\mu\text{s}$  delay and 1 MHz for the master clock speed.

**Calculation 2:** For the extra delay needed for sampling the temperature sensor output, this converts the additional 30  $\mu\text{s}$  to master clock cycles (MCLK) running at 1 MHz.

$$30\mu\text{s} \times \frac{1\text{second}}{1000000\mu\text{s}} \times \frac{1000000\text{MCLK cycles}}{1\text{second}} = 30\text{MCLK cycles} \quad (2)$$

Now we sum together the results of calculations 1 and 2 to get 30.8 MCLK cycles, and then round it up to 31. This is the total delay needed for the sampling period that is inserted into the delay function, as shown on line 30.

Now we calculate the



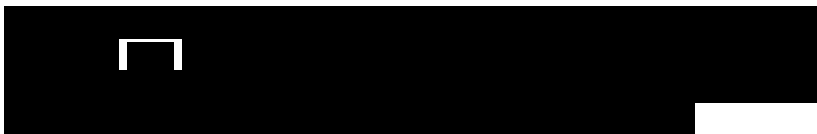
The calculation for converting 13 ADC cycles, needed for converting the measurement into a 10-bit binary number, to master clock cycles is shown by calculation 3. It uses the conversion factors 5 MHz for the ADC clock speed and 1 MHz for the master clock speed.

**Calculation 3:** For converting the measurement into a 10-bit binary number, this converts the needed 13 ADC clock cycles running at 5 MHz to master clock cycles (MCLK) running at 1 MHz.

$$13 \text{ ADC cycles} \times \frac{1 \text{ second}}{5000000 \text{ ADC cycles}} \times \frac{1000000 \text{ MCLK cycles}}{1 \text{ second}} = 2.6 \text{ MCLK cycles} \quad (3)$$

On line 33 an instruction just simply reads the contents of the ADC memory conversion register (ADCMEM0) and assigns it to the storage variable Voltage. This instruction quite literally represents the input signal to our program.

On line 34 is the transfer function that converts the measured voltage sample to degrees Celsius. It's just simply a linear equation in slope-intercept form ( $y=mx+b$ ), where  $m$  is the slope,  $x$  is the measurement minus the adjustment, and  $b$  is 30. The details about this function are typically published in the “Temperature Sensor Calibration” section of the user guide. The slope is calculated by using the two ADC calibration descriptors: the ADC voltage references at 30°C and 85°C. For our microcontroller the slope is calculated to be 0.3929.



After the LEDs have been driven to their proper states, the flow of program execution goes back to the beginning of the loop, at line 28, and then executes the sequence again.

The return instruction on line 44, of code example 33, on page page 169, is never reached.

## Event-Driven Programming Routines and Practices

The MSP430 is meant to run event-driven programs. Such programs share common instructions, routines, and practices. This chapter presents and explains them.

Code Composer Studio (CCS) was used for writing the programming examples. They are generic enough to be used as instructions for any MSP430 with little or no changes. And they are presented in an order which they might appear in a program. To simplify the microcontroller's environment, it is assumed to be built into a development kit, like a Texas Instruments MSP430 LaunchPad.

---

### Boot Initialization

Two MSP430 intrinsic functions are available for modifying the boot program. One will be executed early in the boot, while the other will be executed later in the boot.

---

### Pre-Initialization

This function is an MSP430 translation unit (explained on page 143), so it is placed outside of and before the `main()` function.

Use the pre-initialization boot function for executing one or more instructions during boot-up and before global variables are initialized. For more information about the context of this function, see “Execute a Pre-Initialization Function” on page 141 and “Define a Pre-initialization Boot Hook Function” on page 147.



**Code Example 34:** Format and syntax for the pre-initialization function.

---

```

1 [redacted]
2 [redacted]
3 [redacted] // To not execute this function, return 0
4 } // End of function

```

---



---

### Post Initialization

Like the previous function, this is also an MSP430 translation unit, so it is placed outside of and before the `main()` function.

Use the post initialization function for executing instructions after [redacted]. For more information about the context of this function see “Execute a Post Initialization



Function” on page 142 and “Define a Post initialization Boot Hook Function” on page 147.

The header for this function is declared as void of any return value, and it is void of any input parameters. Line 2 is where you can place one or more instructions. If written in your program, this function will always be executed.

**Code Example 35:** Format and syntax for the post initialization function.

---

```

1 [REDACTED]
2 [REDACTED] */
3 } // end of function

```

---

## Manipulating Bits in Password Protected Registers

A password protected register is typically a sixteen bit register that has its upper address or upper eight bits dedicated to a password. The password is just simply a bit-field mask that represents a specific pattern of eight bits. Although the password mask varies from one register to another, it typically represents the byte  $0xA5$ .

There are basically two types of protected registers. [REDACTED].

### Password that Protects a Single Register

This is how such a password works. Any change to the lower byte of the register must be done by taking into account which fields must be cleared and which must be set and then forming a byte that implements that pattern. That byte is then added to the password to form a single sixteen bit word, and that word is then written into the register. The watchdog register is designed to be used in that way.

To write such an instruction, the password mask and the fields which we want to clear and set are [REDACTED].

In the following example, an imaginary mask for the password is denoted as `PASSWORD`, the standard bits are used as masks for the remaining fields in the register, and an imaginary variable for the entire sixteen bit register is denoted as `REGISTER`. Fields 7, 5, 3, and 1 of the lower register will be set, while fields 6, 4, 2, and 0 will be cleared with this example. For an explanation about the standard bits, see page 47.

**Code Example 36:** Password that protects a single register. [REDACTED].

---

```

[REDACTED]
[REDACTED];

```

---

Be aware that the password must be sixteen bits, meaning, the upper eight bits are the

password and the lower eight bits are zeros. So if the password is  $0xA5$ , then we must convert it to sixteen bits by appending eight more zero bits to it. For example,  $0xA500$ .

### Password that Protects a Set of Registers

This password works very much like the first type, except it is used for unlocking a set of registers. After those unlocked registers have been configured as needed, the password is used for locking them again.

The password is written into the upper eight bits of a sixteen bit register, and the lower eight bits must be taken into account. Just like how the protection of a single register is used. Such an instruction unlocks a set of registers. The set typically belongs to a single module; for example, the power management module (PMM) and the clock system (CS) module.

Relocking the set of registers uses a different instruction.

This topic was introduced earlier on page 54, and it will be elaborated upon now. The byte mode method involves a sixteen bit register which has a sixteen bit register variable, but it also has two eight bit register variables. One variable is for the upper eight bits, and the other is for the lower eight bits.

**Diagram 52:** As published by a microcontroller's user guide, a power management module (PMM) register and a portion of the PMM Registers table. The later is placed in front of all the register tables.

#### *PMMCTL0 Register (offset = 00h) [reset = 9640h]*

15		14		13		12		11		10		9		8	
PMPW															
rw-1		rw-0		rw-0		rw-1		rw-0		rw-1		rw-1		rw-0	
7		6		5		4		3		2		1		0	
Reserved		SVSHE		Reserved		PMMREGOFF		PMMSWPOR		PMMSWBOR		Reserved			
rw-[0]		rw-[1]		r0		rw-[0]		rw-(0)		rw-[0]		r0		r0	

#### *PMM Registers*

Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	PMMCTL0	PMM control register 0	Read/write	Word	9640h	Section 2.3.1
00h	PMMCTL0_L		Read/write	Byte	40h	
01h	PMMCTL0_H		Read/write	Byte	96h	
02h	PMMCTL1	PMM control register 1	Read/write <sup>(1)</sup>	Word	9600h	Section 2.3.2
02h	PMMCTL1_L		Read <sup>(1)</sup>	Byte	00h	

Shown above, as published by a microcontroller's user guide, is a power management module (PMM) register and a portion of the PMM Register table. Such a table is always published at the back of a module's chapter, but precedes all the module's registers. We see that the PMM password (PMPW) is located in the upper eight bits of PMM Control 0 (PMMCTL0) register. And we see that the register table shows the mask

for those upper eight bits as `PMMCTL0_H`. The register variables and masks are listed in the Acronym column.

The following code example shows how that register and its upper register mask is used for unlocking the PMM registers and then relocking them.

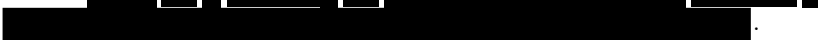


**Code Example 37:** Password that Protects a Set of Registers.



```
1 PMUCTL0 = 0x00000000; // unlocking the PMM registers w. respect to SVSHE
2 PMUCTL0 = 0x00000000; // relocking the PMM registers
```

On line 2, an instruction relocks all the PMM registers. We cannot use the same instruction as seen on line 1. That will not work. Byte mode, as directed by the user guide, along with the wrong password, must be used. Therefore, we write an instruction that



## Watchdog Timer Handlers

Two routines are presented here for handling the watchdog timer. The first just simply disables the watchdog, and the second routine sets up the watchdog into the watchdog mode and provides an interval reset instruction. The register used for the code examples are shown by the “Watchdog Control Register Table” on page 90.

### Placing the Watchdog on Hold

The 16-bit password mask is denoted as `WDTPW`, the 8-bit mask for setting the hold bit-field is denoted as `WDTHOLD`, and the 16-bit register variable is denoted as `WDTCTL`. A single unary addition operation is used for creating a 16-bit word that is written (assigned) into the register.

**Code Example 38:** Placing the watchdog timer on hold.

```
WDTCTL = WDTPW + WDTHOLD; // placing the watchdog timer on hold
```

### Using Watchdog Mode

When the watchdog is placed into watchdog mode, it is used for handling CPU crashes. A power-up or reset automatically puts it into watchdog mode.

**Diagram 53:** Program design pattern for implementing a watchdog timer.

The diagram to the right is an elaboration on two parts of the event-driven pattern shown by diagram 43 on page 128. One part is the `main()` function, and the other part is the interrupt service routine. Placed inside of this diagram and highlighted is a set of watchdog instructions. They conceptually show a basic pattern for implementing the watchdog.

To develop the pattern, you need basic knowledge about the counter interval, details about the pattern, and code examples.

---

### The Counter Interval

The entire pattern is visualized within the context of the watchdog timer counter interval.

The length of the interval, which is in units of clock cycles, will control the placement of the counter reset instruction. It clears the counter back to zero. An instruction may be placed anywhere in the program that resets the counter and changes the interval as needed.

Keep in mind that the main clock signal (MCLK) drives the

Now back to the interval. The length of the counter interval is based upon the number of main clock cycles which are needed for executing a block of instructions. The reset instruction is placed near the end of the block, just before the counter overflows.

The number of clock cycles which are needed to execute a block of instructions is measured with the

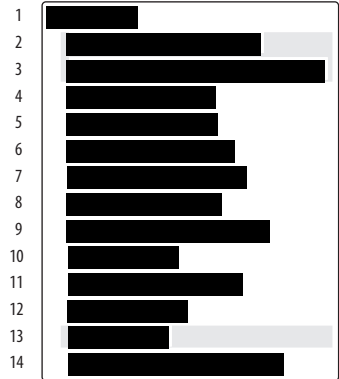
When measuring an interval, we must know whether or not the

---

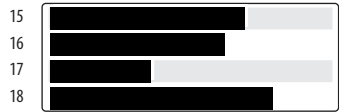
### The Pattern

On line █ of diagram 53, the watchdog timer interval is set and the timer counter is cleared to zero. The length of the interval must be long enough to go past the counter reset instruction shown on line █

#### main() Function



#### Interrupt Service Routine



On line `1` a decision is made about whether or not a timer overflow had caused the microcontroller to reset. If it did, a routine may be executed which carries out some corrective action process. That routine may be used as a subroutine in the reset fault handler, described by a later section. However, the action may not necessarily be corrective. It can be in the form of an illuminated LED or message sent out of a communications peripheral module. Furthermore, one of the instructions in this routine should be used for clearing the watchdog interrupt flag.

At the end of `2` is an instruction that places the microcontroller into a low-powered operating mode where it waits to be interrupted. When an event interrupts the CPU, the interrupt system loads the correlating interrupt service routine (ISR) into the CPU so it can be executed. That ISR is shown on lines `3` through `4`.

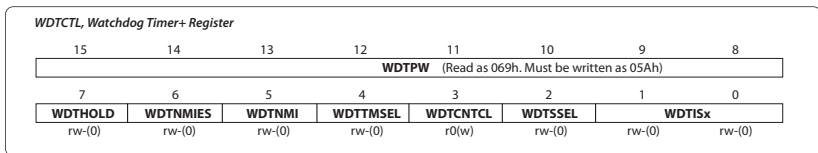
The first instruction in the ISR sets `5`. The interval is made long enough for the CPU to execute the entire block of ISR instructions. The last line of instruction is at `6`, but that instruction is automatically added to the end of the ISR by the MSP430 compiler.

## The Code Examples

The examples are shown by code example 39 and 40. The first example instructs the CPU to clear the counter, set an interval, and handle a reset caused by a counter overflow. The second example instructs the CPU to just simply clear the counter. The line numbers which appear in the code examples directly correlate with those in diagram 53 on page 179.

The watchdog register which is used for the programming examples has its table shown by diagram 35 on page 91, but the register descriptions are not shown. The register's variable name is the Watchdog Timer Control register (WDTCTL).

**Diagram 54:** This watchdog register is used for the following code examples.



**Code Example 39:** Setting the interval, clearing the counter, and checking the watchdog timer interrupt flag. Line 2 sets a watchdog timer interval and clears the counter to zero, while line 3 is the fault handling routine. It checks for a counter overflow. If one has occurred, the following block handles it. The IFG1 register is shown by diagram 55.

```

2 WDTCTL = WDTPW & WDTHOLD; // Set interval & clear the counter
3 if (WDTIFG & IFG1IFG) { // If WDTIFG is set, then
       WDTCTL = WDTPW & WDTISX; // execute corrective action

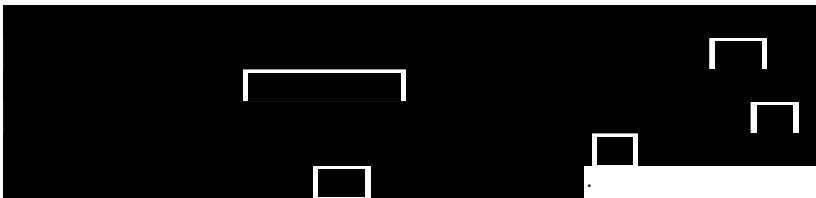
```

The instruction on line `2` simultaneously sets an interval and clears the counter. The two fields which configure the interval are shown by the mask WDTISx. A search in the header file for this microcontroller shows that a single mask does not exist for

both fields. In this case, it has two masks. One mask is for field 0, and the other is for field 1. They are [REDACTED] and [REDACTED] respectively. Publishing a single mask for a couple of fields, but providing two masks for actually doing the job is atypical for a microcontroller's header file. A single mask with a distinguishing suffix is typically supplied for multi-field masks. So just be aware of that.

After some analysis and choices provided by the register's table, we have determined that the interval shall be [REDACTED] clock cycles. Since the microcontroller will be automatically configured to the default clock speed of [REDACTED], that interval equals to [REDACTED] ms ( $[REDACTED]$  MHz / [REDACTED] cycles).

As shown by the watchdog register's table (page 91), to configure the watchdog timer interval select bitfields (WDTISx), field 0 must be set and field 1 must remain cleared. Therefore, mask WDTIS0 will be used for setting that field, and WDCNTL will be used for setting a bit in field 3 that will also clear the counter to zero. The instruction is constructed as described by "Manipulating Bits in Password Protected Registers" on page 176. We add WDTIS1, WDCNTCL, and the password WDPW together, and then we assign (write) that sum to the register variable WDCNTL, as shown by line 2 of the code example.



If the state of the flag is zero, the reset was not caused by a counter overflow, and the block of corrective action instructions are by-passed. If the state is 1, the block of corrective action is executed.

**Diagram 55:** The Interrupt Flag 1 register (IFG1), where, in this case, the watchdog timer interrupt flag is located [REDACTED].

IFG1, Interrupt Flag Register 1		7	6	5	4	3	2	1	0
					NMIIFG				WDTIFG
					rw-0				rw-0)
	Bits 7-5	These bits may be used by other modules. See device-specific data sheet.							
NMIIFG	Bit 4	NMI interrupt flag. NMIIFG must be reset by software. Because other bits in IFG1 may be used for other modules, it is recommended to clear NMIIFG by using BIS.B or BIC.B instructions, rather than MOV.B or CLR.B instructions.							
		0	No interrupt pending						
		1	Interrupt pending						
	Bits 3-1	These bits may be used by other modules. See device-specific data sheet.							
WDTIFG	Bit 0	Watchdog timer+ interrupt flag. In watchdog mode, WDTIFG remains set until reset by software. In interval mode, WDTIFG is reset automatically by servicing the interrupt, or can be reset by software. Because other bits in IFG1 may be used for other modules, it is recommended to clear WDTIFG by using BIS.B or BIC.B instructions, rather than MOV.B or CLR.B instructions.							
		0	No interrupt pending						
		1	Interrupt pending						

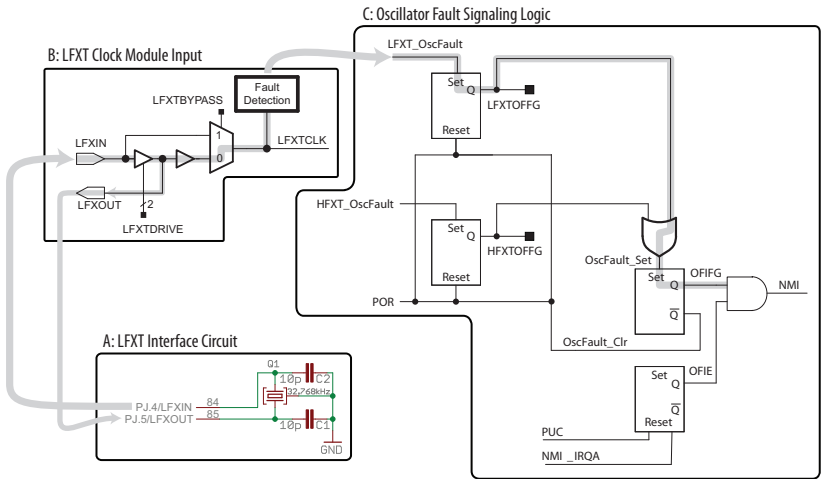
As said earlier, the corrective action taken depends on the expected behavior of your program and what you think must be done. It can be as simple as illuminating an LED or sending a message out a communication peripheral module, or just clearing



## Signal Path from an External Oscillator to the Fault Detector

The path from an oscillator to the fault detection logic will typically involve three sections. Shown by diagram 56 is that path. We're dealing with a low frequency external oscillator (LFXT), so the path will flow along circuits which are dedicated to that type of signal.

**Diagram 56:** Typical Path for external oscillator fault signal detection.



The first section is the interface circuit. That circuit, shown by section A, is part of a schematic typically published by a development kit's user guide. It shows the oscillator terminals connected to pin numbers 84 and 85, along with the signal names and the port channels. In this case, they are channels 4 and 5 of port J (PJ.4 and PJ.5).

The second section of the path, labeled as B, is the signal going into the back end of the clock module. This section is part of the block diagram for a clock module, which is published by the microcontroller's user guide.

Notice the loop in the circuit. It accommodates the voltage swings of the oscillator, which is typically from 0.1 to 4.9 volts.

The first bitfield in section B is labeled as [REDACTED] and it is a two bit field. When we search the user guide for this field, it says that it is used for amplifying the oscillator driving current. Lower frequencies need lower amounts of current while higher frequencies need more, and the user guide tells us about our choices. The second bitfield is called LFXTBYPASS, and it controls the signal by-pass multiplexer. If the signal comes from an oscillator in the form of a crystal, then it must be cleared to zero so the signal can flow through the amplifiers. If the external signal is from a device which is



producing a conventional clock signal, the field is set so the signal can pass around the amplifiers. Within the context of the settling handler, neither one of these bitfields are a concern to us.

Once the signal exits the multiplexer,



Upon entering section C, a fault signal drives a flip-flop to set the low frequency external oscillator fault flag (LFXTOFFG). The outputs of a flip-flop are denoted with letters Q and  $\bar{Q}$ , where one is a complement of the other; meaning, when one output is set the other is cleared and vice versa. The signal then



In the context of the fault handler, we are interested in the LFXTOFFG and OFIFG bitfields. LFXTOFFG will be used for making a decision that results in the flow of execution remaining in our fault handler or not, while the body of the routine will clear both flags.

### Code Example for the Oscillator Fault Handler

The handler is placed inside the system configuration block of instructions of the `main()` function, and before the clock system is configured. As shown by the following code example, lines 5 through 8 form the handler itself. The preceding instructions prepare the microcontroller for the handler.

**Code Example 41:** The oscillator fault handler.

```

1  PJ4 |= 0x00000004;           // Set PJ.4 to LFXIN function
2  PJ5 |= 0x00000008;         // Set PJ.5 to LFXOUT function
3  CS |= 0x00000001;         // Clear to unlock the port channels
4  CS |= 0x00000002;         // Set to unlock CS registers
5  while(LFXTOFFG) {          // While LFXTOFFG is 1, then enter loop
6      OFIFG = 0;             // Clear the OFIFG fault flag
7      OFIFG = 0;             // Clear the OFIFG fault flag
8  }                          // End of while() loop

```

At lines 1 and 2 the port channels are configured to use the external low frequency oscillator (LFXT). These channels interconnect the signals from the oscillator to the fault detection logic. So they need to be setup before the handler is executed.

Diagram 56, on page 183, shows the signal path from the oscillator to the fault signaling logic, but it does not show the paths through channels 4 and 5 in port J. We

need to know which channel bitfields must be configured for creating the path through the port.

Earlier in this book, port pin diagrams were used for learning which bitfields are involved in setting up the channel through a port. This time, a *port pin functions table*, as published by the microcontroller's data sheet, will be used for learning which bitfields are involved. Diagram 57 shows the table we need.

**Diagram 57:** The pin functions table for channels 4 and 5 of Port J (PJ.4 and PJ.5) as published by our microcontroller's data sheet.

**Port PJ (PJ.4 and PJ.5) Pin Functions**

PIN NAME (PJ.x)	x	FUNCTION	CONTROL BITS AND SIGNALS <sup>(1)</sup>					LFXT BYPASS
			PJDIR.x	PJSEL1.5	PJSEL0.5	PJSEL1.4	PJSEL0.4	
PJ.4/LFXIN	4	PJ.4 (I/O)	I: 0; O: 1	X	X	0	0	X
		N/A	0					
		Internally tied to DVSS	1	X	X	1	X	X
		LFXIN crystal mode <sup>(2)</sup>	X	X	X	0	1	0
		LFXIN bypass mode <sup>(2)</sup>	X	X	X	0	1	1
PJ.5/LFXOUT	5	PJ.5 (I/O)	I: 0; O: 1	0	0	0	0	0
						1	X	
		N/A	0	see <sup>(4)</sup>	see <sup>(4)</sup>	X	X	1 <sup>(3)</sup>
						0	0	0
		Internally tied to DVSS	1	see <sup>(4)</sup>	see <sup>(4)</sup>	1	X	0
						X	X	1 <sup>(3)</sup>
		LFXOUT crystal mode <sup>(2)</sup>	X	X	X	0	0	0
						1	X	0
				X	X	1 <sup>(3)</sup>		
				0	1	0		

(1) X = Don't care

(2) Setting PJSEL1.4 = 0 and PJSEL0.4 = 1 causes the general-purpose I/O to be disabled. When LFXTBYPASS = 0, PJ.4 and PJ.5 are configured for crystal operation and PJSEL1.5 and PJSEL0.5 are don't care. When LFXTBYPASS = 1, PJ.4 is configured for bypass operation and PJ.5 is configured as general-purpose I/O.

(3) When PJ.4 is configured in bypass mode, PJ.5 is configured as general-purpose I/O.

(4) With PJSEL0.5 = 1 or PJSEL1.5 = 1 the general-purpose I/O functionality is disabled. No input function is available. When configured as output, the pin is actively pulled to zero.

The first column at the left shows the pin names (PJ.x), and the second column shows the port channel number (x). The third column shows the functions supplied to the pins. We are interested in the LFXIN and LFXOUT functions in crystal mode. Bypass mode is used when a conventional clock signal is supplied to those channels. Under the heading Control Bits and Signals are all the register variables which configure these two channels. They include the .x suffix to denote the channel number, in other words, it is the register bitfield number. But unfortunately, some of the suffixes actually denote the specific channel number, which is misleading. Lastly, registers which are marked with an X are no concern to us.

Not shown by the table are the initial states of

For the LFXIN function in crystal mode, the table shows that bitfield 4 must be set in the Port J Select 0 register (PJSEL0). Field 4 configures channel 4. Therefore, on line 1 of the code example, the instruction sets a bit in field 4 of PJSEL0.

For the LFXOUT function, the table incorrectly shows that bitfield 4 must be set in PJSEL0. It is bitfield 5 that must be set. Field 5 configures channel 5. Therefore, on line 2 of the code example, the instruction sets a bit in field 5 of PJSEL0.

Before signals can flow through any port channel, they must be unlocked. Therefore, on line `1` an instruction uses the `0x00000000` password mask for clearing bits in `0x00000000` to unlock the channels.

On line `4` is another unlocking instruction.

Starting at line `10` and ending at line `15` is the oscillator fault handler. It's a loop in the form of a `while()` repetition statement.

If the expression returns `0b0`, that evaluates to false, but if it returns `0b1000`, that also evaluates to false.

For our microcontroller, `LFXTOFFG` is a single bitfield in the Clock System Control 5 register (`CSCTL5`). Its initial state is `1`, and it remains at that value until the oscillator settles or an instruction clears it to zero. The condition `CSCTL5 & LFXTOFFG` just simply reads that flag.

At line `10` the `LFXTOFFG` flag is cleared. If the oscillator has not settled, the fault detection logic automatically sets the flag back to `1`. At line `15`


The `OFIE` bit must remain cleared at this point in the flow of program execution. Meaning, this oscillator fault handler must be executed before any instructions begin to configure the clock system. Here's why. A well configured clock system will have this bit set so oscillator faults which occur after a power-up can be handled by a dedicated interrupt service routine (ISR). This oscillator fault handler is not meant to cause an ISR. It is meant to allow the external oscillator to have enough time to ramp-up in order to produce a stable signal.


Be aware that the clock system will automatically use a built-in oscillator if the external oscillator is not producing a good signal.

---


## Configuring a Port Channel

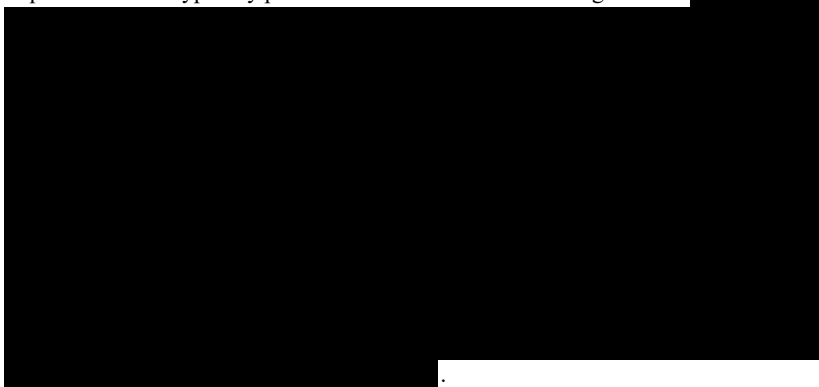
The signaling pathways into and out of a microcontroller are concentrated onto the port channels. Although some pathways, which are typically limited to reset signals, program loading, and program debugging signals do not go through port channels, practically all signaling paths are carried through port channels. Each channel will typically have its own dedicated pin on the microcontroller's case.

Most channels provide more than one service. 



A single channel provides one separate path for handling input signals and another separate path for handling output signals. The diagram for a port channel is published by the microcontroller's data sheet; it is referred to as a Port Input/Output Diagram. An example is shown by diagram 58. At the lower right hand corner of the diagram is a list which outlines all the services which the channel can provide. The data sheet also publishes a table of pin functions (diagram 59), typically on the page following the diagram. It shows which registers and bits are used for selecting a function.

A port channel is typically put into one of four different configurations. 

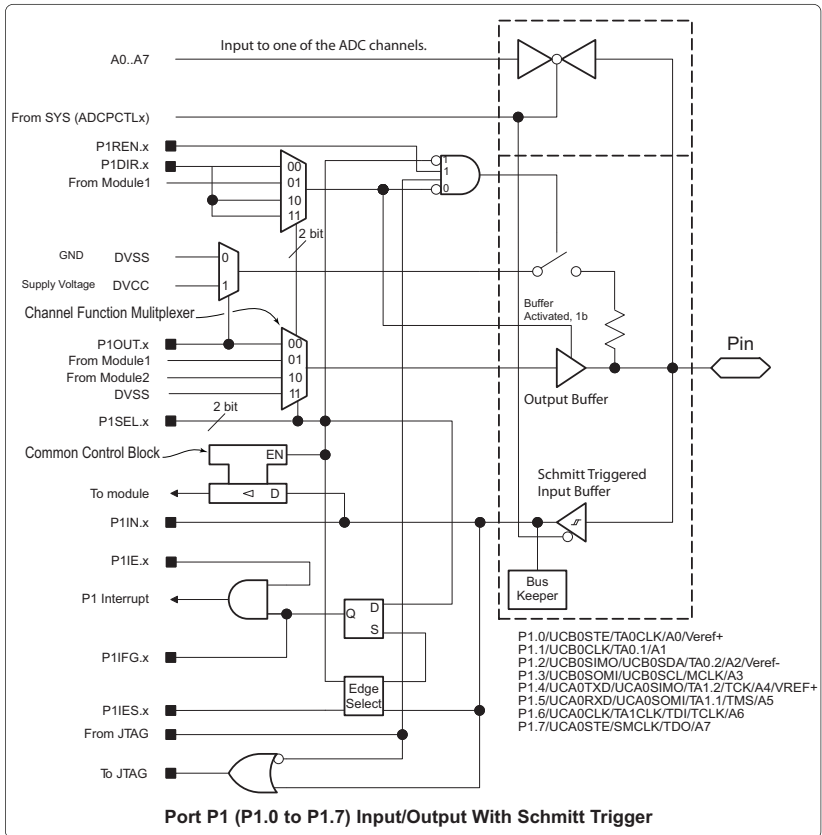


Nine different registers are typically used for configuring a port channel into the GPIO, non-GPIO, or unused function. And those register configurations use a common set of programming instructions. Those instructions are presented by this section. Once those registers have been used to configure the channel into a GPIO or unused function, the channel is then ready for work. But if the channel is configured to a non-GPIO function, then you will probably have to go the peripheral's registers and then configure them as needed.

The code examples in this section have comments which include the register's bitfield accessibility and the initial condition after a power-up or reset event. Most bitfields in the examples will be denoted as rw, which means our program has permission to read

or write bits into the bitfield. The accessibility will typically be followed by the initial condition -0 or -undefined. The -0 means the field is initialized to zero by the PUC reset system, and -undefined means a reset subsystem does not initialize the bitfield, and that the field will hold its state through a reset event. These and all the other notations are defined and published by the preface of the microcontroller's user guide. They are also used as label under each bitfield in a register diagram. The undefined notation (-undefined) is my own notation. A hypothetical register diagram is shown by diagram 15 on page 61.

**Diagram 58:** Port 1 Input/Output diagram for an MSP430FR2433, as published by its data sheet.



An important register that is not shown in the code examples is the Port x Input register (PxIN):

. Undefined means it is not initialized and typically holds its state through a reset event.

You'll notice that all the code examples will include instructions which just simply clear a bit in a field which was already initialized to zero by a PUC. The rationale behind this action is improved reliability. The environment which the microcontroller

is operating within may produce strong electromagnetic noises which on the rare occasion may edit the bit, so even though the reset initializes the bitfield, the program code assures it remains that way before use. If the use case is for a toy which just simply flashes an LED, you probably can avoid the extra initialization.

**Diagram 59:** Port 1 Pin Functions table for an MSP430FR2433, as published by its data sheet. It is typically located on the page after the port input/output diagram.

Port P1 (P1.0 to P1.7) Pin Functions

PIN NAME (P1.x)	x	FUNCTION	CONTROL BITS AND SIGNALS <sup>(1)</sup>			
			P1DIR.x	P1SELx	ADCPCTLx <sup>(2)</sup>	JTAG
P1.0/UCB0STE/ TA0CLK/A0	0	P1.0 (I/O)	I: 0; O: 1	00	0	N/A
		UCB0STE	X	01	0	N/A
		TA0CLK	0	10	0	N/A
		A0/Verif+	X	X	1 (x = 0)	N/A
P1.1/UCB0CLK/TA0.1/ A1	1	P1.1 (I/O)	I: 0; O: 1	00	0	N/A
		UCB0CLK	X	01	0	N/A
		TA0.CCI1A	0	10	0	N/A
		TA0.1	1			
		A1	X	X	1 (x = 1)	N/A
P1.2/UCB0SIMO/ UCB0SDA/TA0.2/A2	2	P1.2 (I/O)	I: 0; O: 1	00	0	N/A
		UCB0SIMO/UCB0SDA	X	01	0	N/A
		TA0.CCI2A	0	10	0	N/A
		TA0.2	1			
		A2/Verif-	X	X	1 (x = 2)	N/A
P1.3/UCB0SOMI/ UCB0SCL/MCLK/A3	3	P1.3 (I/O)	I: 0; O: 1	00	0	N/A
		UCB0SOMI/UCB0SCL	X	01	0	N/A
		MCLK	1	10	0	N/A
		A3	X	X	1 (x = 3)	N/A
P1.4/UCA0TXD/ UCA0SIMO/TA1.2/TCK/ A4 /VREF+	4	P1.4 (I/O)	I: 0; O: 1	00	0	Disabled
		UCA0TXD/UCA0SIMO	X	01	0	Disabled
		TA1.CCI2A	0	10	0	Disabled
		TA1.2	1			
		A4, VREF+	X	X	1 (x = 4)	Disabled
		JTAG TCK	X	X	X	TCK
P1.5/UCA0RXD/ UCA0SOMI/TA1.1/TMS/ A5	5	P1.5 (I/O)	I: 0; O: 1	00	0	Disabled
		UCA0RXD/UCA0SOMI	X	01	0	Disabled
		TA1.CCI1A	0	10	0	Disabled
		TA1.1	1			
		A5	X	X	1 (x = 5)	Disabled
		JTAG TMS	X	X	X	TMS
P1.6/UCA0CLK/ TA1CLK/TDI/TCLK/A6	6	P1.6 (I/O)	I: 0; O: 1	00	0	Disabled
		UCA0CLK	X	01		Disabled
		TA1CLK	0	10	0	Disabled
		A6	X	X	1 (x = 6)	Disabled
		JTAG TDI/TCLK	X	X	X	TDI/TCLK
P1.7/UCA0STE/SMCLK/ TDO/A7	7	P1.7 (I/O)	I: 0; O: 1	00	0	Disabled
		UCA0STE	X	01	0	Disabled
		SMCLK	1	10	0	Disabled
		A7	X	X	1 (x = 7)	Disabled
		JTAG TDO	X	X	X	TDO

(1) X = don't care

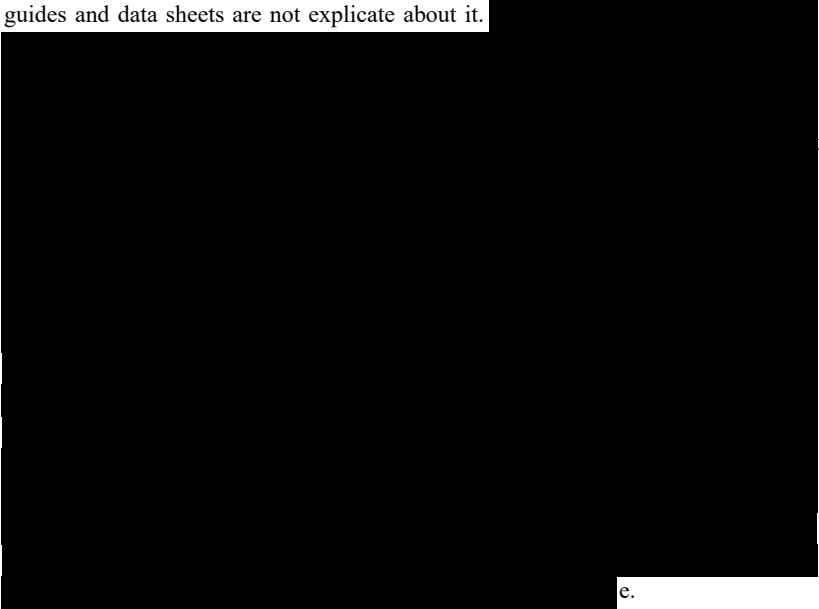
(2) Setting the ADCPCTLx bit in SYSCFG2 register disables both the output driver and input Schmitt trigger to prevent leakage when analog signals are applied.

When configuring the channel for the GPIO function, be aware that

. So when you clear the field to zero to produce a logical low signal out the channel, the channel's logic automatically connects DVSS (ground) to one end of the resistor, but the other end of resistor will not be connected to the channel. It's connected by setting a bit in the Port x Resistor Enable register (PxREN). In this case it acts as a pull-down resistor; the resistor will be pulling the voltage level in the channel to DVSS in order to prevent the voltage

from floating away from that level. This is to assure that the microcontroller's operating environment does not have an electromagnetic affect on the channel's signal. On the other hand, if the bitfield in PxOUT were set to produce a logical high, one end of the resistor would be automatically connected to DVCC, so when the resistor is enabled, its other end will be connected to the channel in order to keep the output signal from floating away from the logical high voltage level.

And finally, there is one matter which we should be aware about because the user guides and data sheets are not explicate about it.



A channel also has two other buffers. One is the signal output buffer, and that buffer holds the output signal state through a reset. The other is a signal input buffer, referred to as a Bus Keeper, and that buffer holds the output signal state through a reset. This is the reason why PxIN and PxOUT are characterized with an initialization of undefined after a reset event.

The following subsections will present code examples that will configure Channel 0 of Port 1 on an MSP430FR2433, but the same code can be used on any other MSP430 with very little or no changes.

To view the bitfields in these registers during operation, put Code Composer Studio into debugging mode (press F11), and then from the **View** menu, select **Register** to open the *Registers window*.

---

### **Configuring as a GPIO Input for Sensing a Signal Changing from Low to High**

Use these instructions for configuring a port channel into a general purpose input or output (GPIO) channel which senses input signals changing from low to high. In other words, the channel will be configured as an input, then initialized to a logically low voltage state (equal to DVSS), then configured to monitor for a voltage state

which changes from low to high (close to or equal to DVCC), and then set a flag when a change is sensed. The example will configure channel 0 of port 1 (P1.0).

**Code Example 42:** Negative Logic GPIO Input Function. Configuring a port channel to sense an input signal which changes from

```

1 [REDACTED] // rw-0. Clear P1SEL0.0 for GPIO function.
2 [REDACTED]; // rw-0. Clear P1SEL0.0 for GPIO function.
3 [REDACTED]; // rw-undefined. Clear P1OUT.0 to produce a low signal,
4 [REDACTED] // which automatically connects DVSS with the built-in
5 [REDACTED] // resistor to pull (hold) the signal low, then
6 [REDACTED]; // rw-0. set P1REN.0 to enable the resistor, and then
7 [REDACTED]; // rw-0. [REDACTED].
8 [REDACTED]; // r-undefined. Clear to set a flag on a rising signal edge.
9 [REDACTED]; // rw-0. Clear the flag before interrupts are enabled. The
10 [REDACTED] // state of this flag will be held through a wake up from LPMx.5.
11 [REDACTED]; // rw-0. Set to enable channel 0 to request an [REDACTED]

```

On lines 1 and 2 of code example 42, fields zero of function selection registers 0 and 1 (P1SEL0 and P1SEL1) are cleared to select the GPIO function for channel 0. Although the registers were already initialized to zero by a PUC (rw-0), we do it again for reliability.

On line 7, field zero in the Port 1 Direction register (P1DIR) is [REDACTED]. On line 8, field zero of the Port 1 Interrupt Edge Select register (P1IES) is cleared to [REDACTED]. And then on line 9, field zero of the Port 1 Interrupt Flag register (P1IFG) is cleared to assure that an unintentional flag will not trigger an interruption. [REDACTED] we clear those bitfields again for reliability.

And finally on line [REDACTED] we set field zero of the [REDACTED] to request an interruption when a flag is set. But since port channels are maskable interruptions, the request will not reach the interrupt system until masked interruptions are enabled (see page 212).

### Configuring as a GPIO Input for Sensing a Signal Changing from High to Low

Use these instructions for configuring a port channel into a GPIO channel which senses input signals changing from high to low. In other words, the channel will be configured as an input, then initialized to a logically high voltage state (equal to DVCC), then configured to monitor for a voltage state which changes from high to



low (DVSS), and then set a flag when a change is sensed. The example will configure channel 0 of port 1 (P1.0).

**Code Example 43:** Positive Logic GPIO Input Function. Configuring a port channel to sense an input signal which changes from a high to low logical state to set a flag which requests an interruption.

```

1 // rw-0. Clear P1SEL0.0 for GPIO function.
2 // rw-0. Clear P1SEL1.0 for GPIO function.
3 //
4 // which automatically connects DVCC with the built-in
5 // resistor to pull (hold) the signal high, then
6 // rw-0. set P1REN.0 to enable the resistor, and then
7 //
8 // r-undefined. Set to set flag on a falling signal edge.
9 // rw-0. Clear the flag before interrupts are enabled. The
10 // state of this flag will be held through a wake up from LPMx.5.
11 // rw-0. Set to enable channel 0 to request an interruption.

```

On lines 1 and 2 of code code example 43, fields zero of function selection registers 0 and 1 (P1SEL0 and P1SEL1) are cleared to select the GPIO function for channel 0. Although the registers were already initialized to zero by a PUC (rw-0), we do it again for reliability, and that will be done for other registers.

On line 3, field zero of the Port 1

On line 7, field zero in the Port 1 Direction register (P1DIR) is cleared to configure the channel to use its input path. On line 8,

And finally on line 11 we set field zero of the Port 1 Interrupt Enable register (P1IE) to give this channel the permission to request an interruption when a flag is set. Since port channels are maskable interruptions, the request will not reach the interrupt system until masked interruptions are enabled (see page 212).


### Configuring a Channel as a Non-GPIO Function


Use these instructions for configuring a port channel to provide a function other than the GPIO function.


Port channels are typically multi-functional. Meaning, they can provide more than one type of functional service. Such channels have a built in multiplexer that can be used for connecting the channel to a particular service provided by a peripheral mod-

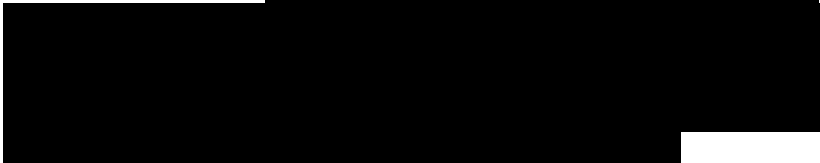
ule. But the channel's multiplexer cannot connect the channel to any module because it has been designed and made to connect with only a specific subset of the microcontroller's peripheral modules.

Which functional services a channel may connect with is published by the microcontroller's data sheet. The port input/output diagram, published by the data sheet, is the first place to go for getting that information. Typically at the lower right corner of the diagram is a list which shows all the functions that each port channel can provide, as shown by diagram 58 on page 188. And typically the page which follows the diagram is a port pin functions table, as shown on page 189. It lists each port channel, the pin which they are connected with, the functions they provide, and the register bitfields which must be configured to select the function. Many of the signal names are in the form of acronyms, and those names are described by a Signal Descriptions table, that can be found earlier in the microcontroller's data sheet.

Configuring a channel to serve as a non-GPIO function typically involves 



Code example 44 shows how channel 0 of port 1 is configured for a non-GPIO function. The routine selects a 



We begin by searching for information about configuring the channel function selection registers. So we open the microcontroller's data sheet, and then turn to the port 1 input/output diagram, as shown on page 188. Near the center of the diagram is the channel function selection multiplexer. Below it is a square, labeled as P1SELx, that represents the two fields in the channel function selection registers (P1SEL0 and P1SEL1), which control the multiplexer. Two bits will control the multiplexer, one from each register. And specifically in this case, bits from fields 0 of both registers. Remember that each register bitfield corresponds with a channel, so bitfield 0 is for channel 0.

At the lower right corner of the port input/output diagram is a list that itemizes every channel in the port and the functions they provide, or more specifically, the function signal names. At P1.0, we see signal name TA0CLK. Now when we go to the front of the data sheet and search for TA0CLK in the Signal Descriptions table we will find it described as the "Timer clock input TACLK for TA0." In other words, it is the input signal for driving the Capture/Compare Timer Block 0 of Timer type A. If you open the microcontroller's user guide, and go to the chapter about Timer A, you'll see a

Timer A Block Diagram. And that diagram shows all the different clock signals which can be used for driving a single block. TA<sub>x</sub>CLK is one of those signals, where x denotes the block number. In this case, it is the TA0CLK input signal at P1.0.

**Code Example 44:** Non-GPIO Function. Channel 0 of port 1 is configured in an MSP430FR2433 to use an external clock signal to drive Timer Module A0.

```

1 [redacted] 0; // rw-0. Set channel 0 bitfield.
2 [redacted] ; // rw-0. Clear channel 0 bitfield.
3 [redacted] ; // Setting a bit in any P1SELx register typically disables
4 [redacted] // the channel's ability to request a CPU interruption.
5 [redacted] ; // rw-undefined. Leave at any state.
6 [redacted] ; // rw-0. Clear P1REN.0 to disable the built-in resistor.
7 [redacted] ; // rw-0. Clear as specified by the Port Pin Functions table.
8 [redacted] ; // [redacted]e.
9 [redacted] ; // rw-0. Clear the channel's flag.
10 [redacted] ; // rw-0. Clear to disable interruptions from channel 0.

```

We must be careful about interpreting a port input/output diagram, because not all signaling goes through the channel function multiplexor. Only output signals go through it. The signal output buffer, shown to the right and near the pin, provides a clue that it is the output path for the port channel. On the other hand,

Remember that a GPIO input is just simply a bitfield in the P1IN register. The block is located between the P1SEL.x and P1IN.x register bitfields. There are some other input paths which provide signaling for interruptions and JTAG signals, but we're ignoring those. That block handles the signaling going into the module which provides the non-GPIO service, in this case, Timer A. Therefore, the channel 0 bits from the P1SEL0 and P1SEL1 registers will control the multiplexor and they can also enable (EN) the common control block to direct an input signal to Timer A, which is named TA0CLK, but not explicitly shown. It is just generically labeled as "To module."

Now we go back to the microcontroller's data sheet and turn to the Port Pin Functions table, which is typically located directly after the Port Input/Output diagram.

The remaining four columns of the Port Pin Functions table tell us which registers and bitfield settings are used for configuring the channel to the function we want. The fourth column, **P1DIR.x**, tells us which signaling direction to put the channel into. It says a zero bit for our function, which is the input direction. The configuration for the P1.0 (IO) function clues us in by showing that the input direction is zero (I:0), and 1

is for the output direction (O:1). So in bitfield 0 of the port 1 direction register (P1DIR), we'll have to clear that field to zero.

The fifth column, **P1SELx**, tells us how to configure the channel function selection bits for registers P1SEL0 and P1SEL1. It shows the bits as 10, which means bitfield 0 of register P1SEL0 must be set to 1, and bitfield 0 of P1SEL1 must be cleared to zero. Keeping in mind that bitfield zero of a port register correlates to channel 0. Therefore, that will send bits 1 and 0 to the channel multiplexer so it will connect the input to Capture/Compare Timer A Block 0 with Channel 0 of Port 1.

The sixth column is labeled as **ADCPCTLx**. Table note (2) tells us that it is a field in the System Configuration 2 register (SYSCFG2). The x tells us that there are more than one of those bitfields. So we open the microcontroller's user guide, and then search for the register named SYSCFG2. Once we have found the table for that register, we then learn that those are bitfields which control the signals going into an analog to digital converter (ADC), and that each of those bitfields are read/write and initialized to zero by a reset from PUC (rw-0). The port 1 Pin Functions table says that each one of those bitfields must be cleared to zero, but since the PUC has already done that for us, we don't have to write instructions to do that.

We have collected all the channel function selection configuration information we need, so now let's step through code example 44 on page 189. We want to configure channel 0 of port 1 as an input to the Timer A Block 0 Clock (TA0CLK) module so we can feed it an external clock signal. So on line 1 we set field zero of P1SEL0, and then on line 2 we clear field zero of P1SEL1 to connect channel 0 with the TA0CLK input signal function. Just as shown by the P1 Pin Functions table.

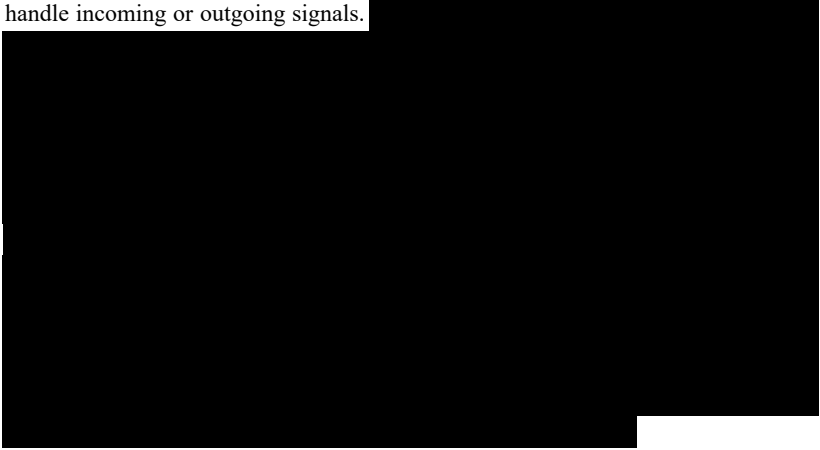
It's worth our time to open the microcontroller's user guide, then turn to the Digital I/O chapter, and then read the channel Function Select Registers section to learn about any side effects which those settings produce. That section tells us that setting a bit in any of those registers to select a non-GPIO function will disable the corresponding port channel's interrupt ability. Meaning, the channel will not be able to request an interruption if its interrupt flag is set, regardless of the state in which the channel 0 bit of the Port 1 Interrupt Enable register (P1IE) is in. That's OK, and that's what we want because that is a GPIO related function.

Now onto line 5, recall that the PxOUT register is used for producing output signals. That is a GPIO function. But when we used the function selection registers to switch the channel away from the GPIO function, that action had disconnected P1OUT from the port channel. So channel 0 of the port 1 output register (P1OUT) can be left at any state we desire. However, it might be a good idea to put an unused bitfield in the PxOUT register into its lowest state of energy in order to mitigate any possible consumption of energy. Therefore, on line 5, the output signal is cleared to zero.

Now onto line 6, recall that when the channel is configured as GPIO, the built-in pulling resistor will be automatically connected to DVCC or DVSS, depending on the state of channel 0 output (P1OUT.0). Back on line 5 we had cleared the output to zero, which automatically connected the resistor to ground (DVSS). But we also want

to assure that there is no influence coming from DVSS which could be placed onto the channel's pin (as shown by the port input/output diagram on page 184). Therefore, we cleared channel 0 of the port 1 resistor enable register (P1REN) to disconnect the resistor from the channel.

On line 7 is an instruction which configures the signaling direction in the channel to handle incoming or outgoing signals.



The remaining three port registers are all used for monitoring input signals which could be used for producing a request for interrupt signal (IRQ). They are the Interrupt Edge Select (PxIES), the Interrupt Flag (PxIFG), and the Interrupt Enable (PxIE) registers. Except for some type of esoteric and sophisticated clock input signal diagnostic interrupt service routine (ISR), we're not going to use these input signals for producing interruptions which will run an ISR. Therefore, we want to disable this feature.

So on line 8, we clear channel 0 of the interrupt edge select register (PxIES), and on line 9 we clear channel 0 of the interrupt flag register (PxIFG). We could have left both of those registers as is, since on line 10 we're going to disable interruptions from channel 0, but we clear them for good programming practice. Then on line 10, we clear channel 0 of the interrupt enable register (PxIE) to disable interruptions from channel 0. Although a PUC will automatically clear registers PxIFG and PxIE to zero, as a good programming practice we assure they remain cleared.

So what have we learned about configuring a port channel to provide a non-GPIO function? We need configuration information from the data sheet and from the Digital I/O chapter of the user guide, and we need to decide on whether or not the non-GPIO signals will be used for producing interruptions. And finally, configuring the port channel may just be the first step in using a non-GPIO function, we typically then have to configure the registers belonging to the peripheral module which is supplying the non-GPIO service.

---

## Configuring a Port Channel as Unused

This topic is covered by “Configuring Unused Port Channels” on page 198.

---

## Accessing Protected Registers

Although the examples in previous sections have already shown how to use protected registers, this section elaborates upon an earlier section called “Manipulating Bits in Password Protected Registers” on page 176.

There are two types of protected registers. The first type is in the form of a built-in password, typically at the higher eight bits of a sixteen bit register, which controls access to the lower eight bits. The watchdog control register is built that way.

The second type is in the form of one or more bits in a register that controls access to other registers. The bits do not typically control access to their own register. Those other registers will belong to the same module, such as the CSKEY bits used for unlocking clock system registers, or to other modules, such as the LOCKLPM5 bit that controls signal flow across port channels.

Here are some commonly used instructions for accessing protected registers. The actual register variable names may be slightly different with your microcontroller.

---

## Watchdog Registers

The watchdog instructions are repeated here so they can be all seen together.

**Code Example 45:** Putting the watchdog on hold. It is put on hold during a scenario when it is not needed, or so it will not interfere with code development. `XXXXXXXXXX` is the password mask, and it must be used every time the register is written into.

---

```
XXXXXXXXXX; // Putting the watchdog on hold
```

---

The next example shows how the watchdog counter is cleared and an interval is set. The first instruction configures the watchdog by

`XXXXXXXXXX` the password mask, and it must be used every time the register is written into.

**Code Example 46:** Clearing the counter and setting an interval.

---

```
1 XXXXXXXXXXXX // Set interval and clear the counter
2 WXXXXXXXXXX // Clear the counter
```

---



---

## Power Management Module Registers

These registers are typically used for setting bits which will force a BOR, POR, or PUC, enabling an internal temperature sensor, and choosing a reference voltage that peripheral modules may use for making decisions. In this example, a password is used for enabling the High-Side Supply Voltage Supervisor (SVSHE).

**Code Example 47:** Unlocking the Power Management Module (PMM) registers. [REDACTED]

```
[REDACTED]
[REDACTED]
[REDACTED]; // Unlock the PPM registers
```

## Clock System Registers

Some microcontrollers have their clock system registers locked after a power-up event. [REDACTED]

**Code Example 48:** Unlocking the clock system registers.

```
[REDACTED]; // Unlock the clock system registers
```

## Memory Protection Unit (MPU) Registers

The memory protection unit acts as a barrier to specific segments of main memory. When enabled, it will not let the CPU write data into segments which store non-volatile data, such as our program. It can be disabled so the program can be upgraded and additional non-volatile data can be placed into those segments and be protected. It does not stop the CPU from reading data in those protected segments.

**Code Example 49:** Unlocking the Memory Protection Unit (MPU) registers. [REDACTED]

```
[REDACTED]; // Unlock the MPU registers
```

## Configuring Unused Port Channels

If a port channel will not be used, it must be configured to mitigate unwanted voltage signals and current drains. Those signals are often referred to as uncontrolled voltages or floating voltages and they may unintentionally cause an interrupt flag to be set. As for the drains, they are also referred to as parasitic current drains and they unintentionally consume power.

Every microcontroller's [REDACTED]

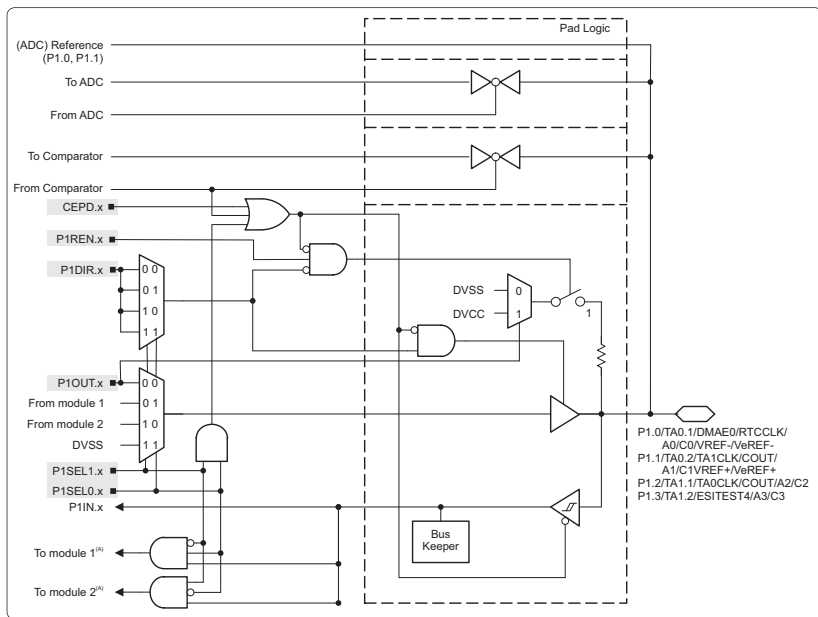
[REDACTED]. That configuration puts the channel into a high impedance state, meaning, it blocks floating voltages from entering the channel and mitigates drains.

But there can be additional measures which may not be mentioned. For example, we may clear the channel output signal to zero, enable the channel's signal pull-up/pull-down resistors, and if the comparator function is present, disable the comparator's voltage buffer in the channel.

To be thorough about this, go to the microcontroller's data sheet and inspect the port channel diagrams to determine which functions may appear at each channel. And then go to the user guide to read about the feature in order to determine if the guide provides additional advice.

The points in the flow of program execution where we want to put unused channels into a high impedance state are when system and peripheral modules are configured. That point is located before unlocking the channels and certainly before maskable interruptions are enabled, as shown by diagram 43, on page 128.

**Diagram 60:** A typical port 1 pin diagram as published by a microcontroller's data sheet. Although it shows a



### Code Example for Putting a Port Channel into a High Impedance State

As a reference model, the code example will put channels 2 and 3 of a port into a high impedance state. In this case, it is port 1, as shown by diagram 60. It is an image of a typical port 1 pin diagram, as published by the microcontroller's data sheet.

All port channels provide the digital I/O function, but they also provide an additional set of functions. Your particular channels may provide a different set of functions than shown here. That only means its port pin diagram may be different from the one seen here.

The basic strategy for putting one or more channels into a high impedance state can be outlined like this. Determine which port channel pins will not be used. Open the



user guide and read the section about the Connection of Unused Pins to learn what should be done (there may be more than one of those sections). Open the data sheet and read the port pin diagram to determine which bitfields control the channel and which functions are provided through the channel. Go back to the user guide and read the chapters or sections about those functions to learn about any voltage drain mitigation advice.

We already know that channels 2 and 3 of port 1 will not be used in our project. They are referred to as P1.2 and P1.3. Let's begin by reading the section about unused pins. It tells us to switch the channels to the digital I/O function and put them into the signal output direction.

So now we go to the Function Selection Registers section of the Digital I/O chapter of the user guide. It tells us that two registers are used for selecting a function, P1SEL0 and P1SEL1. And the tables for those registers tell us that they are both cleared to zero after a power-up or reset; meaning, the General Purpose I/O (GPIO) function is selected. Although they are already initialized to zero, lines 1 and 2 of code example 50 show how to clear the corresponding channel bitfields in those registers to zero.

**Code Example 50:** Putting channels P1.2 and P1.3, of diagram 60, into a high impedance state. Instructions which have the additional comment "default," may not be needed because those bitfields are initialized to those states during a power-up or system reset.

---

```

1  [REDACTED] ); // To select as i/o, clear fields 2 and 3 (default)
2  [REDACTED] ); // To select as i/o, clear fields 2 and 3 (default)
3  [REDACTED] ); // To [REDACTED] 1
4  [REDACTED] ); // To [REDACTED]
5  [REDACTED] ); // To enable resistors, set fields to 1
6  [REDACTED] ); // To disable comparator buffers, set fields to 1

```

---

Now we put the signal direction of those two channels to be outward. To be in that state, the port Direction Registers section of the user guide tells us to set a bit in the corresponding channel bitfield of the direction register (PxDIR). The register table also tells us that information. The variable name for that register is P1DIR, and its table tells us it is initialized to zero, the input direction. [REDACTED]

We now have done what the user guide tells us for putting the channel into a high impedance state, now we may take some extra measures to mitigate any parasitic current drain.

We open the data sheet to the port 1 pin diagram. This particular port has two diagrams. One covers channels 0 through 3, and the other covers 4 through 7. The first one is shown by diagram 60. We read it to determine what other functions and circuits may cause some drain at the channel. Just below the pin is a list that shows precisely which functions are provided by the channels. We also see a signal conditioning (voltage pull-up/pull-down) resistor and three function circuits which are of concern to us.

In order to stop the possibility of any floating voltage, we



The first of the three functions which are of concern to us is the input to the channel (P1IN.x). Although it is not selected, it will not be affected by a floating voltage because the enabled resistor draws the voltage to DVSS (ground). This assures the channel 2 and 3 bitfields of P1IN remain cleared to zero. Therefore, as expected, it is of no concern to us.

The Schmitt trigger (shown as a triangle, enclosing a parallelogram, and pointing to P1IN.x) has an inverted control signal applied to its bottom. When the signal on the line is high, the circle inverts it to low, and vice versa. That signal turns the trigger on and off. Although a trigger is an active circuit, as opposed to a passive circuit that does not need operating power, it apparently does not draw enough current to be of concern, otherwise, the guide would have said so.

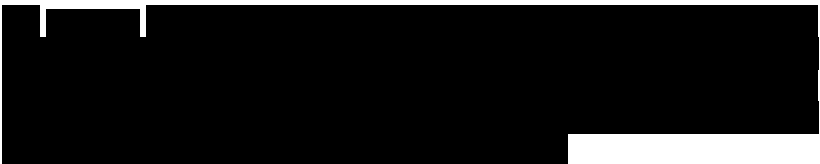
Next is the analog to digital converter function (ADC). It's shown at the top of the port pin diagram. By reading what the user guide says about this function, we learn that it is turned off after a power-up or reset. Meaning, its tri-state buffer is in the off state. So it is also of no concern to us.

The voltage comparator is a different matter. By reading what the user guide says about it, we learned that it is initialized to an on state. The guide also says that some levels of voltage will cause the comparator to draw a parasitic current. Those levels are at the transition voltage levels of the logic gates. If not used, it advises us to disable the comparator by turning off its tri-state voltage buffer, shown in the pad logic of the port pin diagram. Therefore, on line 6 of the code example, the corresponding channel bitfields (CEPD3 and CEPD2) are cleared in the Comparator Control 3 register (CECTL3). In this case, it's comparator E, one of several types of comparators.

---

### **Unlocking Modules & Digital Port Channels which are in the LPMx.5 Domain**

All modules and port channels which are located in the LPMx.5 domain must be unlocked before the flow of execution reaches the port channel interrupt handler. The handler depends on the channels to be unlocked, otherwise, the individual channel interrupt flags cannot be cleared.



However, all unused channels, meaning, those which are serving pins which are not connected to anything, must be put into a high-impedance state to mitigate the flow of current across those pins and into their channels. That is done earlier in the flow of execution when the ports are configured.

**Code Example 51:** Unlocking all the port channels.

---

```

[REDACTED]; // Unlock all port channels
[REDACTED]
[REDACTED]).

```

---

## Port Channel Interrupt Flag Handler

Use this routine for clearing all the digital I/O port channel interrupt flags. Otherwise, they'll cause ISRs to be unintentionally loaded into the CPU after maskable interruptions are enabled. The result could be unexpected or corrupted program behavior.

Do not get these maskable interrupt flags confused with non-maskable flags. The non-maskable flags will be handled by the reset fault handler, described later. And keep in mind that other peripheral modules have their one flags, which can be cleared in similar, if not the same ways.

The example shown here is in the form of a global function. [REDACTED]

The typical microcontroller comes with its first two ports able to set interrupt flags, and the register variables are generically referred to P<sub>x</sub>IFG. They are typically eight bits wide, so the number 0xFF is used to clear all eight bitfields.

**Code Example 52:** A port channel interrupt flag handler in the form of a function.

---

```

1 [REDACTED]; // function prototype
2 [REDACTED]; // function call
3 [REDACTED] { // function definition
4 [REDACTED]; // Clear P1 channel IFGs
5 [REDACTED]; // Clear P2 channel IFGs
6 } // end portChannelIfgHandler()

```

---

## Clearing a Port Channel Flag from Inside of an ISR

Another place where a channel IFG must be cleared is from inside of an ISR that services an event at a port channel. The example in this case, [REDACTED]

**Code Example 53:** Clearing a port channel IFG. [REDACTED]

---

```

[REDACTED]; // clearing the interrupt flag for P1.3

```

---

Be aware two related matters. When reading the flag code from an interrupt vector register (IVR), a topic explained later, the act of reading the register will automatically clear the flag. When a register contains a single flag bitfield, reading that register will typically clear that flag too.

---

### Determining the Source of an Interrupt Flag

Following this section is a topic that explains how to write a reset fault handler, and the next chapter explains how to write a conventional interrupt service routine (ISR) and a fractional low powered interrupt handler. Handlers and ISRs have two characteristics in common. They respond to flags which have been set, and sometimes they must determine which event had set the flag. The former characteristic is often dependant on the later. The later also involves vectors and some slightly complicated relationships which must be known before attempting to write a handler or ISR. Therefore, those topics are first introduced here.

---

### Interrupt Vector

A vector is a fundamental concept used in programming ISRs and fault handlers. It is an address in main memory which stores the address number to the first instruction in an ISR or to the boot program. In other words, a vector is a cross-reference between an interrupt flag and its ISR or to the reset system.

---

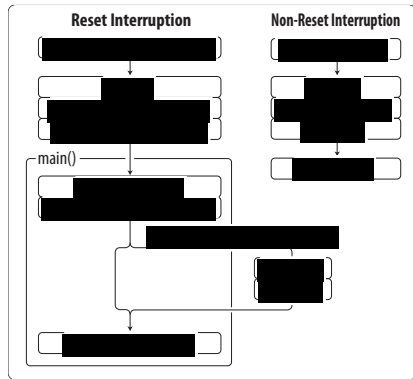
### Flow of Execution from a Set Flag to an ISR or an RFH

Most, if not all, events which cause an interruption will also involve setting an interrupt flag (IFG) which identifies the event. Once a flag is set, it produces a request for interruption signal (IRQ), then the interrupt system places the IRQ in a queue based on its priority with other requests.

The event may cause a non-reset interruption or a reset interruption. The former does not force a system reset, while the later does. Therefore, we have two different flows for handling those classes of events. Also, both classes contain members which will set maskable or non-maskable flags.

**Diagram 61:** Two interrupt driven flows. One is driven by non-reset interruptions, which do not force a system reset. The other is driven by reset interruptions, which do force a system reset.

The flow for a non-reset interruption is handled by an ISR, while the flow for a reset interruption is managed by an ISR or a reset fault handler (RFH). The flows are shown by diagram 60, and their context is within the operating mode diagram on page 132.



### Basic Flow for the Non-Maskable and Maskable Interruptions

Interruptions which do not force a reset are caused by non-maskable and maskable interruptions. Before the interruption occurs, the microcontroller may be in the active mode while executing an instruction in `main()` or in an ISR, but it is typically in some low powered operating mode (LPM 0, 1, 2, 3, 4, or LPMX.5). Interrupting an ISR is a special case called a nested interruption, and it typically involves an instruction in the ISR which re-enables maskable interruptions.

When the event occurs, it

Flow of execution for the non-maskable interruption is explained in detail by Chapter 27 on page 249, while the flow the maskable interruption is covered by Chapter 28 on page 265.

### Basic Flow for the Reset Interruption

Interruptions which force a reset are caused by system faults, processing faults, a signal from the  $\overline{\text{RST}}/\text{NMI}$  pin, or signals which wake up the microcontroller from some fractional low powered operating mode (LPMx.5).

Before the interruption, the microcontroller is in one of three possible operating modes.

When an event which causes a reset occurs, it sets a unique flag.

If the microcontroller was in the active or a conventional low powered mode, then

If the microcontroller was in a fractional low powered mode, then the

At the end of the reset, which is always a PUC, the address to the boot program is loaded into the CPU, the microcontroller is released to active mode, and that program is executed (not shown by diagram 61). The last instruction in the boot calls

Once in , all the configuration routines (also not shown by diagram 61) are executed, and the flow reaches the reset fault handler. If the reset was caused by a fault or a signal from the  $\overline{\text{RST}}/\text{NMI}$  pin, the handler determines which flag caused the interruption and transfers the flow to the instructions for handling that flag. Otherwise, the reset fault handler is by-passed.

The flow then reaches the instruction that enables maskable interruptions.

If the reset was caused by an event at a peripheral module which was active in a fractional low powered mode, its flag has been pending in the flag interrupt queue. These are maskable interrupt flags, so when such interruptions are enabled, the CPU is immediately interrupted, the flow leaves `main()`, the flag's vector is loaded into the CPU, and its ISR is executed. After it is executed, the flow returns to `main()`.

The next, and last, instruction in `main()`, puts the microcontroller into some conventional low powered or fractional low powered operating mode.

Flow of execution for the reset interruption is explained in detail by Chapter 25 on page 237.

---

## Flag to Routine Relationships

The routines which we are concerned with here are the reset fault handler (RFH) and the interrupt service routine (ISR). Before writing any one of these routines we must know about their relationships with the flags which cause them to be executed.

---

### Flag to Reset Fault Handler (RFH) Relationship

The reset fault handler is typically placed inside of `main()`, after the microcontroller is configured, but before maskable interruptions are enabled.

There are many types of events which will set a flag that causes a reset. They are all listed by

The event is referred to as the interrupt source, and each source has its own unique

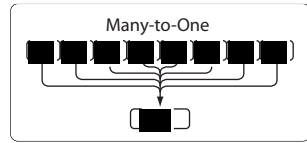
All those

the inter-

rupt system uses it to determine a point where to then transfer the flow of execution. That point is either at the beginning of the BOR, POR, or PUC.

This all means that the reset vector has a one-to-many relationship with all sources which will cause a reset. Keep in mind that the reset vector is not like all the other vectors. It is not a cross-reference between a flag and its interrupt service routine. It is a sort of cross-reference between a type of reset flag and a BOR, POR, or PUC.

**Diagram 62:** The reset fault handler has a relationship with many reset interrupt flags.



Once the reset system is finished and releases the microcontroller to active mode, the boot program is executed. The last instruction in boot calls [REDACTED].

Once the flow of execution reaches the [REDACTED] handler, instructions in it have to distinguish which flag is set and then use that information for transferring the flow to the proper subroutine that will properly disposition the reset event, if needed.

## Flag to ISR Relationships

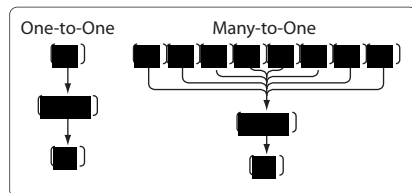
In contrast to the reset flag which forces a reset, a conventional flag will cause an ISR to be executed. Meaning, it tells the interrupt system to stop the CPU if it's running, to load the corresponding vector into the CPU, and then release CPU to execute the vector's ISR.

The exception is an interruption from some fractional low powered mode (LPMx.5), which is a maskable interruption. When the microcontroller emerges from LPMx.5, it must go through [REDACTED].

There are two basic relationships between these types of flags and their ISR vectors: one-to-one and many-to-one. There [REDACTED].

[REDACTED] That means such ISRs will have to distinguish which flag had caused the interruption and then transfer the flow to the proper subroutine.

**Diagram 63:** For conventional interrupt flags (which do not force a reset), they can have a one-to-one or a many-to-one relationship with a single ISR vector.



For example, let's look at the vector for a port. A single vector will act as the [REDACTED] between eight port channel flags at a port and a single ISR. That

means the ISR must distinguish which flag was set and transfer the flow to the subroutine for that specific channel.

## Flag Determining Code Examples

All interrupt flags are located inside of registers. Some are located inside of conventional registers and are in the form of individual bitfields. Others are located inside of interrupt vector generator registers. A generator is dedicated to doing one job: it uses the entire register to present a code which indicates the pending interrupt flag (IFG).

Flags which are located in conventional registers are typically

Keep in mind that some conventional flag registers are password protected. So before an instruction may clear a flag,

Instructions for determining which flag had caused the interruption is typically placed inside of reset fault handlers (RFH) and interrupt service routines (ISR). So once the flow of execution enters them, they will typically have to decide which flag had caused the interruption, then based on the decision, transfer the flow to the proper subroutine to handle the interruption and its flag.

A control structure is used for making the decision and its resulting transfer. It is in the form of an `if()` or `switch()` statement, which were introduced on page 105.

The international standard for the C Programming Language defines a statement as specifying an action to be performed.

### Using the `if()` Statement

Use the `if()` selection statement for reading a single flag in a . If there are more than one flag which must be read, an individual `if()` statement must be used for each one.

A `switch()` statement should not be used because the number of bitfield patterns would far exceed the number of flags. The set of patterns would have to include every pattern of flag bitfields, since multiple flags could be set. That set is calculated by the factorial of the number of flags in the register.

The `if()` statement is built of a condition which is followed by a block of instructions which belong to the statement. That block is called the body of the statement. When the body has more than one line of instruction, it is delimited with curly brackets.

The *condition* is in the form of an expression which reads a flag's bitfield. The *expression* is built of a register variable and a mask to read the field's state. The result of the expression must be a zero or 1. That's a Boolean result which the statement



uses for making a decision. If the expression results to 1, the flag is in a set state, so the flow of execution is then transferred to the statement's body. Otherwise, the body is by-passed.

The following example reads Special Function Interrupt Flag 1 register (SFRIFG1) to determine the state of the Watchdog Timer Interrupt Flag (WDTIFG). If the flag is set, the flow is transferred to the body of the statement, and its instructions are executed. The empty curly brackets represent the body. It would have instructions which handle the timer overflow event in some way, and finish with clearing the flag.

**Code Example 54:** Using the `if()` statement to read the Watchdog Timer Interrupt Flag (WDTIFG). If it is set, the flow will be transferred to the statement's body.

```
// If WDTIFG is set, then enter the statement body.
```

**Diagram 64:** In this case, the WDTIFG mask is located in the Special Function IFG 1 register (SFRIFG1). Its description table is not shown. Do not let

SFRIFG1 Register

15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
JMBOUTIFG	JMBINIFG	Reserved	NMIIFG	VMAIFG	Reserved	OFIFG	WDTIFG
rw-(1)	rw-(0)	r0	rw-0	rw-0	r0	rw-(1)	rw-0

## Using the `switch()` Statement

Use the `switch()` multiple selection statement for

The statement is built of a controlling expression which is followed by a block of subroutines. The expression reads the register to get the flag code.

The statement then uses the code for transferring the flow of execution to the proper subroutine where the event is handled. Each subroutine is distinguished with a case label, where the label is a specific flag code number.

An important behavior which characterizes a vector generator register is the . The set only contains zero and even numbers. Zero represents no outstanding flag, while an even number identifies a specific flag.

Although we may just simply use the register variable as the controlling expression for a `switch()`, there exists an MSP430 intrinsic function that may be used instead. It leverages that even number characteristic for improving the transfer speed to the proper case. The name of the function is .

## The Function

When reading an interrupt vector generator register, use this function as the controlling expression in a `switch()`. It will force the compiler to create an efficient data

structure out of the `switch()` that will be faster to execute. The following example shows its syntax.

**Code Example 55:** Syntax for the `switch()` function.

This function takes two parameters, and it returns a single value.

The first parameter (`reg`) is the `switch()`, and the second variable (`range`) is the end `switch()`. It represents the range of codes from zero to a specific even integer. In other words, the end code is the last or largest number in the generator's set of codes. The range parameter can be in the form of an integer or the mask for the flag of the last code number.

This function returns an integer that represents the flag code presented by the IVR. And that number tells the `switch()` which is the proper case to execute.

### Code Example for using the `switch()` to Determine which Flag is Set

This example involves a `switch()` that reads the interrupt vector generator register for Port 1 to determine which channel had set the flag, and then it transfers the flow of execution to the proper case where the event is handled. The flag is automatically cleared when the register is read.

**The P<sub>x</sub>IV Register Table.** The IVR and table for the port is shown by diagram 65. This is a generic IVR and table for a digital I/O port which is typically published by the microcontroller's user guide. Meaning, we use it for all digital I/O ports having channels which can interrupt the CPU. It's called the Port `x` Interrupt Vector (P<sub>x</sub>IV) Register, where the letter `x` is replaced with the port number of interest to us. In this case, it's the P1IV Register. Notice that all sixteen bits of the register are dedicated to presenting a single flag code, but only the lower eight are needed.

**Diagram 65:** A port interrupt vector register (P<sub>x</sub>IV).

PxIV Register							
15	14	13	12	11	10	9	8
PxIV							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
PxIV							
r0	r0	r0	r-0	r-0	r-0	r-0	r0

PxIV Register Description				
Bit	Field	Type	Reset	Description
15-0	PxIV	R	0h	Port <code>x</code> interrupt vector value 00h = No interrupt pending 02h = Interrupt Source: Port <code>x</code> .0 interrupt; Interrupt Flag: PxIFG.0; Interrupt Priority: Highest 04h = Interrupt Source: Port <code>x</code> .1 interrupt; Interrupt Flag: PxIFG.1 06h = Interrupt Source: Port <code>x</code> .2 interrupt; Interrupt Flag: PxIFG.2 08h = Interrupt Source: Port <code>x</code> .3 interrupt; Interrupt Flag: PxIFG.3 0Ah = Interrupt Source: Port <code>x</code> .4 interrupt; Interrupt Flag: PxIFG.4 0Ch = Interrupt Source: Port <code>x</code> .5 interrupt; Interrupt Flag: PxIFG.5 0Eh = Interrupt Source: Port <code>x</code> .6 interrupt; Interrupt Flag: PxIFG.6 10h = Interrupt Source: Port <code>x</code> .7 interrupt; Interrupt Flag: PxIFG.7; Interrupt Priority: Lowest

The register's table, referred to as the PxIV Register Description, has five columns. The first column is labeled as Bit, while the second column is labeled as Field. They tell us that the bitfield mask is PxIV and is sixteen bits wide (15-0). The mask is generic, so in this case it is actually P1IV, which also happens to be the register variable! The next column shows the field's type as being R, meaning it is a field that can only be read, not written into. The Reset column shows the field as 0h, which is an alternative hexadecimal notation for 0x0. That means after a reset, the bitfield is cleared to zero. The last column is labeled as Description. It lists the flag code number for each port channel (02h to 10h). Notice that every code number is even. When no flag is pending, the code is zero. Also notice that every item in the list is prioritized, with channel 1 having the highest priority. The way these codes are listed and their priority is a common design practice for interrupt vector generating registers.

**The switch() Code.** Code example 56 shows a switch() statement that reads the IVR for Port 1 to determine which channel had set the flag, and then it transfers the flow of execution to the proper case where the event is handled. When the flag is read, it is automatically cleared.

**Code Example 56:** Using the switch() to determine which channel interrupt flag had caused the interruption in Port 1.

```

1  [REDACTED] // Get code, then switch to proper case
2  [REDACTED] : [REDACTED] // Case for channel 0 (P1IFG.0)
3  [REDACTED] // Handle the event and [REDACTED]
4  [REDACTED] ; // Exit switch()
5  [REDACTED] // Case for channel 1 (P1IFG.1)
6  [REDACTED] // Handle the event and [REDACTED]
7  [REDACTED] ; // Exit switch()
8  [REDACTED] : // Case for channel 2 (P1IFG.2)
9  [REDACTED] // Handle the event and [REDACTED]
10 [REDACTED] // Exit switch()
11 [REDACTED] : // Case for channel 3 (P1IFG.3)
12 [REDACTED] // Handle the event and [REDACTED]
13 [REDACTED] // Exit switch()
14 [REDACTED] // Case for channel 4 (P1IFG.4)
15 [REDACTED] // Handle the event and [REDACTED]
16 [REDACTED] // Exit switch()
17 [REDACTED] : // Case for channel 5 (P1IFG.5)
18 [REDACTED] // Handle the event and [REDACTED]
19 [REDACTED] // Exit switch()
20 [REDACTED] : // Case for channel 6 (P1IFG.6)
21 [REDACTED] // Handle the event and [REDACTED]
22 [REDACTED] // Exit switch()
23 [REDACTED] : // Case for channel 7 (P1IFG.7)
24 [REDACTED] // Handle the event and [REDACTED]
25 [REDACTED] // Exit switch()
26 } // End switch()

```

On line 1, instead of just using the register variable P1IV as the controlling expression, the [REDACTED] function is used. The function will tell the MSP430 compiler to create a data structure out of the switch() that is quicker to traverse, and to return the code in the P1IV Register. Its parameters are the register variable and the

code for the last flag in the range of codes. When the flow of execution enters the `switch()`, it will use the code to transfer the flow to the proper case.

The cases can be understood as subroutines. There are eight, one for each port channel's flag.

One line 2 is the first case, and it handles

On line 4 is a `Break` statement.

The remaining seven cases work in the same way.

### Conventional Register Scenario

An example of a conventional register which is completed dedicated to flags is the port channel interrupt flag (IFG) register. It's shown by the following diagram. The `switch()` code for determining which flag is set in the register is shown by code example 57.

**Diagram 66:** A PxlFG register and its description table.

PxIFG Register							
7	6	5	4	3	2	1	0
PxIFG							
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

PxIFG Register Description					
Bit	Field	Type	Reset	Description	
7-0	PxIFG	RW	Undefined	Port x interrupt flag 0b = No interrupt is pending. 1b = Interrupt is pending.	

**Code Example 57:** Using a `switch()` to search for a set flag in a conventional which is fully occupied with bitfields of flags.

```

1  [REDACTED] {
2  [REDACTED] // Read the register, then switch to proper case
3  [REDACTED] // Case for channel 0 (P1IFG.0)
4  [REDACTED] // Handle the event and [REDACTED]
5  [REDACTED] // Exit switch()
6  [REDACTED] :
7  [REDACTED] // Case for channel 1 (P1IFG.1)
8  [REDACTED] // Handle the event and [REDACTED]
9  [REDACTED] // Exit switch()
10 [REDACTED] // Case for channel 2 (P1IFG.2)
11 [REDACTED] // Handle the event and [REDACTED]
12 [REDACTED] // Exit switch()
13 [REDACTED] // Case for channel 3 (P1IFG.3)
14 [REDACTED] // Handle the event and [REDACTED]
15 [REDACTED] // Exit switch()
16 [REDACTED] // Case for channel 4 (P1IFG.4)
17 [REDACTED] // Handle the event and [REDACTED]
18 [REDACTED] // Exit switch()
19 [REDACTED] // Case for channel 5 (P1IFG.5)
20 [REDACTED] // Handle the event and [REDACTED]
21 [REDACTED] // Exit switch()
22 [REDACTED] // Case for channel 6 (P1IFG.6)
23 [REDACTED] // Handle the event and [REDACTED]
    [REDACTED] // Exit switch()
    [REDACTED] // Case for channel 7 (P1IFG.7)

```

```

24   ██████████ // Handle the event and ██████████
25   ██████████ // ██████████
26 } // End switch()

```

---

The difference between a `switch()` that reads an IVR and a `switch()` that reads a conventional IFG register involves the ██████████ function and the cases. The IVR scenario uses the function, and the cases numerically increment by two integers. In this scenario, the function is not used, and the cases increment by the flag's place value in the registers word. In this case, it's an eight bit word. Here's how it works.



Once the `switch()` has the register contents, it transfers the flow of execution to the proper case. Let's assume the contents is `0x2`, the flow is transferred there. The sub-routine in the case is executed, and the `break` statement transfers the flow out of the `switch()`.

---

## Enabling and Disabling Maskable Interruptions

Events at modules cause interrupt flags to be set. A flag signals the interrupt system to interrupt the CPU and load an interrupt service routine in it or cause a system reset.

A maskable interruption is one that is produced by a peripheral module. Its signal can be blocked from telling the interrupt system to handle an interruption. In contrast, a non-maskable interrupt cannot be blocked, and it is typically produced by a system module.

The bitfield that controls access to the interrupt system is called the General Interrupt Enable bit (GIE), and it is located inside of the CPU's status register. Only assembly language code can directly access that register. But two MSP430 intrinsic C functions can be used for manipulating the GIE. One can set it to allow maskable signals to reach the interrupt system. The other can clear it to block the signals.

An instruction that enables maskable interruptions is typically placed after system and peripheral modules are configured and right after the reset fault handler.

An instruction that disables maskable interruptions is placed before instructions that depend on disabled maskable interruptions. For example, some modules require maskable interruptions to be disabled before they are reconfigured.

**Code Example 58:** Enabling maskable interruptions. This

```
// Enable maskable interrupts
```

**Code Example 59:** Disabling maskable interruptions. This

```
// Disable maskable interrupts
```

## Unlocking and Locking FRAM

Before going into an example about unlocking and locking FRAM, a review of volatile and non-volatile memory and what controls access to FRAM is in order. The information presented in this section is a prerequisite to the next section that presents the volatile data handler.

### Review of Volatile and Non-Volatile Memory

Main memory is constructed of volatile and non-volatile sections of memory. Volatile sections are typically made of a semiconductor technology called Static RAM (SRAM), while non-volatile segments will be constructed of either a technology called Flash, or Ferro-Electric RAM (FRAM), or both. Non-volatile segments were initially built only of Flash, but the latest generations are typically built of FRAM and Flash. The word Flash is not an acronym; it's just simply a name that the inventor gave it.

In order to work, memory is energized with electrical power. When the power is removed, data which is stored in volatile memory will vanish, while data which is stored in non-volatile memory is retained. Our program is stored in non-volatile segments of memory, but storage variables will automatically be placed in non-volatile segments.

However, data which is stored in non-volatile memory can actually be changed. For example, we can erase the existing program in it, and replace the program with a newer one. And that work can also be done while the microcontroller is in-service.

### FRAM Access Control

Non-volatile memory will typically have some controls for protecting its contents. The controls for protecting FRAM grow more sophisticated with newer generations of the MSP430.

At least one system module is dedicated to controlling the CPU's access to FRAM. The control is divided into providing permissions for

That means some microcontrollers will leave the FRAM unlocked after a reset, while others will lock it.

The modules which control access to FRAM are typically called the [REDACTED] Registers for those modules contain the bitfields which can be configured to control the type of access we desire.

All FRAM microcontrollers will come with a system controller, but they will also come with either a FRAM controller, or an MPU, or both. The registers which control access to FRAM will belong to one of those modules, but control over FRAM will not typically be shared across them. Therefore, to determine which registers provide access control, we have to refer to the microcontroller's Functional Block Diagram, as published by its data sheet, to determine which modules our microcontroller has, and then we have to refer to the microcontroller's user guide to learn about them. An example of that diagram is shown on page 18.

The system controller, the FRAM controller, and the MPU will carry out work other than access control, but it is the MPU that focuses on specific types of access controls, such as dividing FRAM into variable sized sections of security with their own read, write, and execute permissions.

---

### Example for Unlocking and Locking FRAM

This example uses the MSP430FR2433. When it emerges from reset, the FRAM remains unlocked to allow the CPU to read and execute what's inside of it, but it's locked to prevent any writing into it.

Be aware that in this case our microcontroller (an MSP430FR2433) has two separate sections in main memory which are constructed of FRAM. The user guide and data sheet will tell us that. One section is called Program FRAM, and it's where our program is stored. The other section is called Data FRAM or Information FRAM. The purpose of that section is to provide some non-volatile space in memory for us to use for whatever want to put in there.

The example involves three lines of instructions. The first line [REDACTED]

[REDACTED]

---

### Register which Controls the FRAM

Keep in mind that different families and models within those families of microcontrollers will use different registers, different options, and different bitfield masks for controlling access to FRAM, so this register is just an example to provide one point of view. The example's main purpose is to point out the pattern we'll always use when accessing data in FRAM: *first unlock it, then access it, then relock it.*

This microcontroller depends on its system control module for controlling access to its FRAM. And the control bits are located in the System Configuration Register 0 (SYSCFG0), as shown by diagram 67. Let's take a closer look at that register.

It's a sixteen bit register, with the upper eight bitfields used for a password having the mask FRWPPW. Every time we write into that register, the instruction must include the password.

This microcontroller has some ability to partially protect the Program FRAM by locking a range of addresses in it so the CPU will not be able to read or write into them, but it will be able to execute the instructions in them. Fields 7 through 2, which are identified with the mask FRWPOA, are used for [REDACTED]. For example, if we were to set the bit in field 3 to 1, then [REDACTED]

**Diagram 67:** The System Configuration Register 0 (SYSCFG0) for an MSP430FR2433.

SYSCFG0 Register							
15	14	13	12	11	10	9	8
FRWPPW <sup>(1)</sup>							
rw-1	rw-0	rw-0	rw-1	rw-0	rw-1	rw-1	rw-0
7	6	5	4	3	2	1	0
FRWPOA <sup>(2)</sup>						DFWP	PFWP
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1	rw-1

<sup>(1)</sup> The password must be written with the FRAM protection bits in a word in a single operation.

<sup>(2)</sup> These bits are valid only in the MSP430FR235x and MSP430FR215x devices.

#### SYSCFG0 Register Description

Bit	Field	Type	Reset	Description
15-8	FRWPPW <sup>(1)</sup>	RW	96h	FRAM protection password FRAM protection password. Write with 0A5h to unlock the FRAM protection registers. Always reads as 096h
7-2	FRWPOA <sup>(2)</sup>	RW	0h	Program FRAM write protection offset address from the beginning of Program FRAM. The offset increases by 1KB resolution. 000000b = The write protection starting from the beginning of Program FRAM; the entire Program FRAM under PFWP protection 000001b = The FRAM program memory is unprotected (read/write) between the beginning of program FRAM and the beginning of program FRAM + 1024 B. The remainder of the program FRAM is protected by the PFWP protection. 000010b = The FRAM program memory is unprotected (read/write) between the beginning of program FRAM and the beginning of program FRAM + 2048 B. The remainder of the program FRAM is protected by the PFWP protection. 000011b = The FRAM program memory is unprotected (read/write) between the beginning of program FRAM and the beginning of program FRAM + 3072 B. The remainder of the program FRAM is protected by the PFWP protection. 000100b = The FRAM program memory is unprotected (read/write) between the beginning of program FRAM and the beginning of program FRAM + 4096 B. The remainder of the program FRAM is protected by the PFWP protection. : 111111b = The FRAM program memory is unprotected (read/write) between the beginning of program FRAM and the beginning of program FRAM + 64512 B. The remainder of the program FRAM is protected by the PFWP protection.
1	DFWP	RW	1h	Data (Information) FRAM write protection 0b = Data (Information) FRAM write enable 1b = Data (Information) FRAM write protected (not writable)
0	PFWP	RW	1h	Program (Main) FRAM write protection 0b = Program (Main) FRAM write enable 1b = Program (Main) FRAM write protected (not writable)

<sup>(1)</sup> The password must be written with the FRAM protection bits in a word in a single operation.

<sup>(2)</sup> These bits are valid only in the MSP430FR235x and MSP430FR215x devices.

Bitfield 1, which has the mask DFWP, will protect the section of Data FRAM by controlling the CPU's ability to write into that section. Bitfield 0, having the mask PFWP, will protect the Program FRAM. When these fields are cleared to zero, the CPU will be able to write into those sections.



## Code Example

The data we will access is a storage variable located in FRAM, and the FRAM is locked when the microcontroller emerges from reset. The MSP430 compiler typically puts storage variables in volatile memory built of SRAM, but an earlier instruction (not shown) had told the compiler to put it into non-volatile memory built of FRAM. That instruction had used the `PERSISTENT()` `#pragma` to put it into FRAM, and that will be explained by the next section about the volatile data handler.

This microcontroller has two sections of FRAM: Program and Data FRAM. Since we need to know which section of FRAM to unlock, we then ask ourselves which section did the `PERSISTENT()` `#pragma` put it into? We need that answer in order to unlock the correct section of FRAM. It will typically be put it into the lowest addresses of Program FRAM. But how do we know that? We can check the memory to verify that assumption.

To carry out that check, we begin by using our programming integrated development environment (IDE), in this case it's Code Composer Studio (CCS), to load the program into the microcontroller, in other words, put it into debug mode. Then we go to the Main Menu and click on **View**, and then select **Memory** to open the *Memory Browser window*. In that window we type into its search textbox the name of the variable, and then press Enter. The browser will search for the variable and then highlight its address location and contents in memory. Sometimes the browser may not be able to locate a `PERSISTENT()` variable, so instead you can go directly to your program code, select the variable, then mouse right-button click on it to open a pop-up menu, and then select **Add Watch Expression**. Now go back to the Main Menu and click on **View**, and then select **Expressions** to open the *Expressions window*. The variable and its address number will appear in the window. Now we know the address to the variable, so we open the data sheet for the microcontroller and go to the Memory section. That section will tell us the address boundaries of every section in main memory. With all that information we are able to determine that our variable is located in Program FRAM, not Data/Information FRAM.

**Code Example 60:** Unlocking Program FRAM to update a variable in FRAM, and then relocking it.

```

1  [REDACTED] ; // Unlock Program FRAM,
2  [REDACTED] ; // then increment counter variable, and
3  [REDACTED] ; // then relock Program FRAM.

```

So on line 1 of the code example, the instruction unlocks the Program FRAM. Be very careful about this instruction. The password mask is actually equal to [REDACTED] and we want to clear the Program FRAM Write Protection ([REDACTED] field to zero; therefore, we want to combine the password and the [REDACTED] mask, not the [REDACTED] mask, to form a sixteen bit word, and then assign it to the register. On line 2, an instruction accesses the [REDACTED] variable named [REDACTED] and then increments it by 1. On line 3 the instruction relocks the Program FRAM. Once again be careful with the

relocking instruction. We want the [REDACTED] bits to remain cleared to zero, and we want the [REDACTED] and [REDACTED] fields set to 1. Therefore, we have to combine the password mask, the [REDACTED] mask, and the PFWP mask to form a sixteen bit word, and then assign it to the register.


---

## Volatile Data Handler

As described earlier on page 76, a storage variable is used for holding data which changes. Furthermore, variables can be organized into a set of variables called a storage structure. Structures are typically made of a type of storage structure called an array. When the CPU executes a program instruction which creates a storage variable or structure, it typically places them in a section of main memory called random access memory (RAM). In contrast, our program is stored in non-volatile read only memory (ROM).

RAM is volatile. Meaning, when power is removed from main memory, all the data which is stored in RAM vanishes. Therefore, any data stored in RAM is called volatile data. Power will be removed from memory when the microcontroller is put into a fractional low powered operating mode (LPMx.5), or when the microcontroller goes through a reset, and obviously, when it is disconnected from power. When the microcontroller is put into a conventional low powered operating mode, such as LPM 1, 2, 3, or 4, power is not removed from main memory.

If we need to protect variables and storage structures from volatility, we have some options.



If FRAM is used for main memory, we have at least one method for protecting volatile data, but depending on the microcontroller's design, we may have two methods. The first method involves a preprocessing directive called a `PERSISTENT()` `#pragma`. It's basically an MSP430 intrinsic function that can copy volatile data into a segment of ROM which is built of FRAM, the same place where our program is located. The second method involves a system module called Backup Memory (BAKMEM), but not all FRAM microcontrollers have this module. It's basically a set of registers where we can copy volatile data into. The registers are sixteen bits wide, and the amount of registers available to us depends on which microcontroller model we choose. This section will describe how to use both of those methods.

## Using the PERSISTENT() #pragma to Protect Volatile Data

Use the PERSISTENT() #pragma for converting a variable or data structure from a volatile class of storage to a non-volatile class of storage. So when the microcontroller is put into a fractional low powered operating mode (LPMx.5), or goes through a reset, or when it is disconnected from power, that data will not be lost.

Two programming examples are presented. One example protects a single variable, and the other example protects the variables in an array. Keep in mind that the code is for an MSP430FR2433, and the examples are an elaboration on code example 60, on page 216, that shows how to unlock and relock FRAM in that specific microcontroller. That microcontroller has two sections of FRAM, one is dedicated to program storage, and the other is dedicated to data storage. Those two FRAM sections were introduced and explained on page 216, and we learned how to determine which one of them were used by the PERSISTENT() #pragma for storing non-volatile variables and at which address in memory. The following two examples provide a view of how this specific microcontroller is used, while other microcontrollers, especially of other families of MSP430, will most probably use different registers and may have just a single section of FRAM or they may have more than two, but the MSP430 compiler will probably still place the non-volatile variables in the lowest addresses of program FRAM to protect them.

### Code Example for Protecting a Single Variable

In this example, the storage variable we want to protect is named counter.

**Code Example 61:** Using the PERSISTENT() #pragma to create a non-volatile variable. It will be saved during a fractional low powered mode (LPMx.5), a reset, and when power is lost. The FRAM unlocking and locking instructions are specific to the MSP430FR2433.

---

```

1  PERSISTENT(counter) // Make the counter variable non-volatile
2  volatile char counter = 0; // Declare and initialize the variable
3  }
4  FRAM_UNLOCK; // Unlock Program FRAM
5  counter++; // Increment counter variable
6  FRAM_LOCK; // Relock Program FRAM
7  z = counter; // Read counter and assign to z
8  }

```

---

On line 1 we place the variable as a parameter for the PERSISTENT() #pragma. Notice that there is no `volatile`. That is a requirement.

On line 2 we declare the variable, and we do it just as we would declare any other variable. It is specified as `volatile`, and in this case, further specified as a `char` and initialized to zero.

On line 3 is the `volatile`, and on line 4 is the instruction that unlocks the `FRAM`. As a reference, the System Configuration Register 0 (SYSCFG0) table is shown by diagram 67 on page 215. Be careful about the syntax for unlocking the FRAM, because it uses a register password (`0x00000000`).

On line 5, the number stored by the variable is read and incremented by one, and then on line 6, the FRAM is `locked`. Once again, pay attention to how the password, the `0x00000000` and the PFWP masks are combined into a sixteen bit word and assigned to `0x00000000` to lock the FRAM.

On line 7 is just a simple instruction that reads `0x00000000`.

### Code Example for Protecting the Variables in an Array

In this example, three storage variables are organized into an array named `r`. Keep in mind that the unlocking and locking instructions are specific to the MSP430FR2433, so the register variable and masks will probably be different for other microcontrollers.

**Code Example 62:** Using `PERSISTENT()` `#pragma` to make a data structure non-volatile.

```

1 volatile uint8_t r [3] ; // make r non-volatile
2 uint8_t r [0] = 0 ; // declare and initialize the data structure
3 uint8_t r [1] = 1 ;
4 uint8_t r [2] = 2 ; // Unlock Program FRAM
5 uint8_t r [2] ++ ; // Increment variable at r[2]
6 uint8_t r [2] ++ ;
7 } // End main()

```

On line 1 we place the identifier for the array as a parameter for the `PERSISTENT()` `#pragma`. Notice that there is no semicolon that terminates the instruction, and that the instruction is located outside of and before the `main()` function. That is a requirement.

On line 2 we

6. The `char` specifier just simple makes every variable in the array to be eight bits wide. And the array must always be declared directly after the `PERSISTENT()` `#pragma`.

On line 3 is the signature for our `main()` function, and on line 4 is the

One line 5, the number stored by the array at position `r[2]` is read and incremented by one, and then on line 6, the FRAM is relocked. Once again, pay attention to how the password, the DFWP, and the PFWP masks are combined into a sixteen bit word and assigned to `SYSCFG0` to lock the FRAM. And finally, on line 7 is the

## Using the Backup Memory Registers

If your microcontroller comes with backup memory registers, then use them for copying data which are located in volatile variables and saving the data in non-volatile registers. They will remain non-volatile during a fractional low powered mode (LPMx.5), a reset, and when powered is disconnected.

The following diagram shows a typical register table for a set of backup registers, as published by the microcontroller's user guide. It shows two registers that provide us with four bytes of non-volatile storage, but other microcontrollers may have many more. The first column, named Offset, tells us where the register is located in main memory. The microcontroller's data sheet tells us at which address the registers begin, and the offset tells us how many address numbers the register is away from the first address in memory. Since we're developing in C, we'll be using the register variable names to access those registers, so those address numbers are of no concern to us.

**Diagram 68:** A register table for some backup memory registers, as published by the microcontroller's user guide. In this case, four bytes of backup memory is available.

**BAKMEM Registers**

Offset	Acronym	Register Name	Type	Access	Reset
00h	BAKMEM0	Backup Memory 0	Read/write	Word	Undefined
00h	BAKMEM0_L		Read/write	Byte	
01h	BAKMEM0_H		Read/write	Byte	
02h	BAKMEM1	Backup Memory 1	Read/write	Word	Undefined
02h	BAKMEM1_L		Read/write	Byte	
03h	BAKMEM1_H		Read/write	Byte	

The next column, named Acronym, is of concern to us. It lists the register variable names. Each register has three variable names. For example, `BAKMEM0` is used for reading and writing into the entire sixteen bits of the register. That's called word mode access. The remaining two variables, which have the suffixes `_L` and `_H`, are used for accessing the lower or upper eight bits of the register respectively. When using those names, the process is called byte mode access.

The last column is named Reset. It just simply tells us what is in the register after a reset. It's undefined because what is stored in the register depends on what our program had written into it before the reset or power-up event.

Shown by code example 63 is a variable that has been declared and initialized on line 1, and on line 2. Byte mode is used for

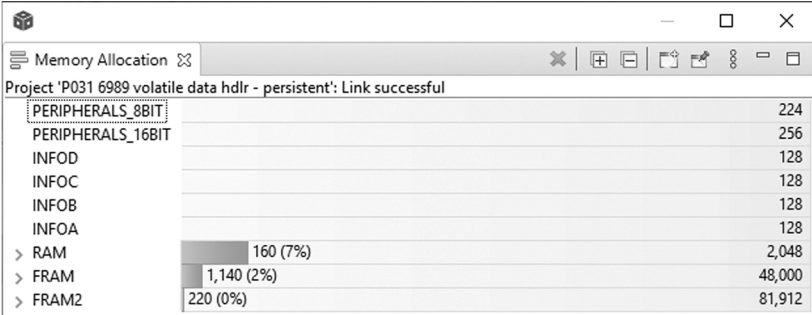
**Code Example 63:** Using a BAKMEM register to protect data stored in a volatile variable.

```
1 volatile uint8_t x; // declare and initialize x as an 8-bit variable
2 BACKMEM_L = x; // write the value of x into BACKMEM_L
```

## Determining How Much Memory is Consumed

There will come a time when you will need to know how much memory your program has consumed and how much remains available. Use the Memory Allocation tool for viewing that information. To open this window while in CCS, go to the *Main Menu*, click on *View*, and then select **Memory Allocation**. It's shown by diagram 69.

**Diagram 69:** Memory Allocation window. Use it for determining how much storage the `PERSISTENT() #pragma` is consuming in bytes. It will appear under FRAM as `.TI.persistent`. The backup registers will not be explicitly shown, because they will be implied within the sixteen bit peripherals.



Section	Memory (bytes)	Percentage
PERIPHERALS_8BIT	224	
PERIPHERALS_16BIT	256	
INFOF	128	
INFOC	128	
INFOB	128	
INFOA	128	
> RAM	2,048	160 (7%)
> FRAM	48,000	1,140 (2%)
> FRAM2	81,912	220 (0%)

Main memory is divided into sections, where each has its own purpose. The tool will itemize them into a list. Some sections can be expanded to view their subsections. In one of the FRAM sections, you will see the subsection where the `PERSISTENT() #pragma` had placed its code. The subsection is named `.TI.persistent`. Another subsection of concern to us is named `.text`; it is where our program is stored. On the right side of the window is the total amount of memory allocated, or available, in bytes for each section.

The `PERIPHERALS_8BIT` section contains all the , while `PERIPHERALS_16BIT` contains all the . The info sections contain data such as the microcontroller device identification number, a CRC value, and silicon chip hardware versions. Look in the device descriptor table of the data sheet to learn what's exactly in there. RAM stores volatile data, such as the program execution stack and variables, while FRAM stores non-volatile data, like our program. The

allocated sizes will not change, but may be different for other microcontrollers. The subsections which fill RAM and FRAM are described, to some extent, by the MSP430 Optimizing C/C++ Compiler User Guide (SLAU132), but the MSP430 Assembly Language Tools User Guide (SLAU131) goes into more detail.

---

## Entering a Low Powered Operating Mode

The MSP430 offers a set of operating modes to choose from. Each member of the set is different. The set shown here is what is currently offered. But depending on the microcontroller, it may contain fewer modes.

The primary mode is called

The microcontroller's user guide tells us exactly which systems and peripherals are cutoff, while the data sheet gets into specifics. The typical operating mode diagram, as published by a user guide, is shown on page 132.

The set of modes can be divided into a subset of conventional low powered operating modes and a subset of fractional low powered operating modes. Here is what distinguishes one from the other. The fractional modes practically turn off everything except for the real-time clock module (RTC) and the ability for some ports to sense an incoming signal from an event. This means all volatile data is lost, and when the microcontroller interrupted from such a mode, the flow of execution begins at the brown-out reset (BOR).

Putting the microcontroller into a low powered operating mode is typically the

---

## Conventional Lower Powered Operating Modes

When the microcontroller is interrupted from a conventional low powered operating mode, it will immediately enter the active mode (AM) and the flow of execution will begin at the proper interrupt service routine (ISR). After the ISR is executed, the microcontroller is automatically put back into the mode from which it was interrupted.

Code example 64 lists five instructions. Each one puts the microcontroller into a specific low powered mode. The instruction is typically placed at the end of `main()`, right before the return statement.

**Code Example 64:** The set of conventional low powered operating modes.

```
1 // enter mode 0
2 // enter mode 1
3 // enter mode 2
4 // enter mode 3
5 // enter mode 4
```

### Fractional Lower Powered Modes (LPMx.5)

Earlier generations of microcontrollers do not have this feature. But if your microcontroller is built of FRAM technology, it most probably does have it.

When the microcontroller is interrupted from a fractional low powered operating mode (LPMx.5), it will immediately enter the active mode where the flow of execution will begin at the brownout reset (BOR). An instruction in `main()` will enable maskable CPU interruptions. So directly after they are enabled, the interrupt system can load the proper ISR into the CPU. When finished with the ISR, the flow of execution is transferred back to the instruction which put the microcontroller into LPMx.5.

Putting the microcontroller into a fractional low powered mode is a two step routine. First the microcontroller is prepared for it, and then a conventional low powered operating mode instruction completes the job.

Preparing the microcontroller just simply means to turn off the power management module (PMM) voltage regulator. It supplies power to everything in the microcontroller. Once off, power will typically remain flowing only to the real-time clock module (RTC) and to the input gate of every port channel which can provide an interruption service. Including the channel that can signal the microcontroller to reset from a BOR or load an ISR into the CPU (that particular channel provides the  $\overline{\text{RST}}$ /NMI function). Cutting off the power also means losing all data stored in volatile memory, which is a section of memory referred to as RAM.

Therefore, from an interrupt perspective, we have two sources of interrupt signals offered to us, at least for now, while in a fractional low powered mode. The first is an internal signal from the RTC. It can produce a signal when a specific calendar time is reached, or when the RTC is in timer mode and causes a timer overflow event. The second signal may come from an external source produced by some peripheral device.

Code example 65 shows the instructions for preparing the microcontroller for entering a fractional mode.



**Code Example 65:** Preparing a microcontroller to enter a fractional low powered operating mode.

```

1  // function prototype
2  ; // function call
3  // start of function definition
4  // unlock PMM registers w. regard to SVSHE
5  // set to turn off PMM voltage regulator
6  // lock the PMM registers
7  } // end prepareForLPMx5()

```

On line 1 is the function prototype. Place it before `main()`. On line 2 is the function call. Place it right before the instruction which puts the microcontroller into a low powered mode, as shown by code example 64. On lines 3 to 7 is the function definition that contains instructions which prepare for a fractional low powered mode; it is typically placed outside of and after `main()`.

Let's now take a closer look at the body of the function.

On line 5, a byte mode instruction will set a bit in the lower eight fields of the register to turn off the voltage regulator. In this register, that field's mask is `PMMREGOFF`. The last instruction, on line 6, uses a byte mode instruction to relock the PMM registers. According to the user guide, we just need to assign the wrong password to do the job.

Code examples 66 and 67 show how to put the microcontroller into a fractional low powered mode. They both use the function defined by code example 65 for preparing the microcontroller for the fractional mode. At the time this book was published, the MSP430 can only be put into LPM3.5 or LPM4.5. To do that, we prepare the microcontroller for the fractional mode, as shown on lines 1. Then on lines 2, we either use the instruction for LPM3 or LPM4.

**Code Example 66:** Placing the microcontroller into LPM3.5.

```

1  // prepare for LPMx5, see code example 65
2  // enter mode 3

```

**Code Example 67:** Placing the microcontroller into LPM4.5.

```

1  // prepare for LPMx5, see code example 65
2  // enter mode 4

```

---

## Delay Function

A program may have points in the flow of execution where it must be paused before proceeding to the next instruction. The pause may be needed for one period of time, or it may be needed over several periods of time. For example, a pause can be used by a routine for toggling a port channel from low to high to illuminate an LED for a period of cycles.

A pause can be inserted into the flow of execution with a signal from a timer module, or it can be inserted with the `delay_cycles` function.

That function is intrinsic to the MSP430 compiler. It does not return any value, but it does take a single value in the form of an integer. That integer represents clock cycles, meaning, the number of clock cycles which the CPU must count through before it proceeds to the next instruction. Therefore, this function forces the CPU to be occupied with counting cycles, and that is an energy consuming process.

Use this function in a routine when CPU availability and energy consumption do not have a high priority or no negative consequences. Use an interrupt signal from a timer overflow event for creating pauses when CPU availability and energy consumption is a high priority.

The following example shows the function as occupying a single line of code with `delay_cycles` as the number of cycles it must count through. If the CPU is driven by a 1 MHz clock signal, this number of cycles is very close to producing a 1 second delay.

If a routine does not depend on a delay in units of time, but purely on the number of clock cycles needed to execute some instructions or process, then use the index to lookup “clock cycles for an instruction” for information about determining the number of clock cycles you need.

**Code Example 68:** Use the `delay_cycles` function for inserting a pause into the flow of program execution. If the clock signal is running at 1 MHz, a value of `1000000` will produce about a 1 second delay. The default clock speed after a power-up is 1 MHz.

---

```
delay_cycles(1000000); // For a 1 MHz clock, this produces a delay of ~1 second
```

---

The parameter for this function is an unsigned long integer. Meaning, the number of cycles can be within the range from 0 to 4,294,967,295.



## Interrupt Handling and Interrupt Vectors

When properly programmed, the MSP430 is a microcontroller which can be put into an appropriate operating mode from where it will monitor for events, react to those events, and then return back to the mode from where it began to continue monitoring for events. It is designed and manufactured to be an event driven microcontroller.

Earlier chapters had introduced the interruption and its vector. This chapter goes into that topic with more detail...the basic details we need for understanding how an interrupt flag is bound to an ISR.

---

### CPU Interruptions are Event-Driven

The MSP430's ability to be event driven is provided by its CPU Interruption System. It is built with a distributed set of event monitoring blocks of logic and a single block of control logic.

---

### Event Monitoring Blocks

The monitoring blocks are built into the various systems and peripheral modules. Each block is unique to the system or peripheral module, so they can monitor for events which are unique to their system or module. When the logic senses an event, it sets a specific bit (to 1) in a register to indicate that the event has been sensed. That bit is located inside of a bitfield called an interrupt flag (IFG). Each type of event monitoring logic has its own dedicated IFG. Therefore, the IFG distinguishes which specific event had occurred. Keep in mind that at this point of the interrupt process, the interrupt system views the IFG as an interrupt request (IRQ) signal.

---

### Conventional Flag Registers and Interrupt Vector Registers

A conventional register of flags is just simply an eight or sixteen bit register where each bitfield is dedicated to a single flag. Some registers may include a mixture of fields where some are dedicated to flags while others are dedicated, for example, to configuring a peripheral module.

An interrupt vector register (IVR) is an eight or sixteen bit register where all the fields are used for presenting a single code number. The number represents a specific flag. When a program instruction reads such a register, typically, that action will also automatically clear the flag. If more than one flag has been set and are pending (waiting to be serviced), the IVR will automatically present the next flag code. That next flag will typically be serviced after the flow of execution has executed the ISR for the current flag and returned back to the operating mode from where it was interrupted. The pending flag will then put the interrupt system back into action.

## The Interrupt Service Routine and Vector

An interrupt service routine (ISR) is a function (set of instructions) we develop for handling an event, and an interrupt vector is a unique number which we use for binding a flag to an ISR. Although that number begins as an integer that represents the vector's priority among all the other vectors, when our program is built and loaded into memory, the MSP430 compiler converts the priority number to the main memory address number where the first instruction in the ISR is located.

### Interrupt Vectors

All vectors are defined as symbolic constants by a section located at the end of the microcontroller's `msp430.h` file. For brevity, diagram 70 shows the beginning of that section for an MSP430FR2433, and it is very typical for all MSP430 microcontrollers. The first two definitions can be seen, and both are for the `PORT_2_VECTOR`. That is the actual vector name we use for binding port 2 IFGs to an ISR. The first definition is for Assembly language usage, while the second definition is for C language usage. Both `#define` directives create the same symbolic constant: `PORT2_VECTOR`.

**Diagram 70:** Shown here is the beginning of the interrupt vector definitions as listed at the end of typical `msp430.h` header file.

```

/*****
 * Interrupt Vectors (offset from 0xFF80 + 0x10 for Password)
 *****/

#pragma diag_suppress 1107
#define VECTOR_NAME(name)          name##_ptr
#define EMIT_PRAGMA(x)             _Pragma(#x)
#define CREATE_VECTOR(name)        void * const VECTOR_NAME(name) = (void *) (long) &name
#define PLACE_VECTOR(vector,section) EMIT_PRAGMA(DATA_SECTION(vector,section))
#define PLACE_INTERRUPT(func)      EMIT_PRAGMA(CODE_SECTION(func, ".text:_isr"))
#define ISR_VECTOR(func,offset)    CREATE_VECTOR(func); \
                                     PLACE_VECTOR(VECTOR_NAME(func), offset) \
                                     PLACE_INTERRUPT(func)

#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define PORT2_VECTOR      ".int41" /* 0xFFDA Port 2 */
#else
#define PORT2_VECTOR      (41 * 1u) /* 0xFFDA Port 2 */
#endif

```

Shown by diagram 71 are all the `PORT2_VECTOR` vector definitions in the C language. The Assembly definitions are not shown, since we're not interested in them. The second column shows the symbolic name of the vector. That symbol is what we use for binding the vector to an ISR, or more precisely, binding the flags which are connected to that vector to an ISR. The third column shows the vector's interrupt priority number. For example, `(41 * 1u)` is a C language expression that produces the unsigned integer 41. The product of 41 and `1u`, where the suffix `u` converts

t. That will break the ability to

use the same program in other MSP430s. The last column is a comment which describes the vector. For example, `/* 0xFFDA Port2 */` means the vector is located at the address `0xFFDA` in main memory, but stored at that address is the address number to the first instruction in the actual interrupt service routine (ISR) for that vector. That translation work is handled by the MSP430 compiler.

**Diagram 71:** The typical list of interrupt vector definitions as found at the end of a microcontroller's header file.

```
#define PORT2_VECTOR      (41 * 1u)    /* 0xFFDA Port 2 */
#define PORT1_VECTOR      (42 * 1u)    /* 0xFFDC Port 1 */
#define ADC_VECTOR        (43 * 1u)    /* 0xFFDE ADC */
#define USCI_B0_VECTOR    (44 * 1u)    /* 0xFFE0 USCI B0 Receive/Transmit */
#define USCI_A1_VECTOR    (45 * 1u)    /* 0xFFE2 USCI A1 Receive/Transmit */
#define USCI_A0_VECTOR    (46 * 1u)    /* 0xFFE4 USCI A0 Receive/Transmit */
#define WDT_VECTOR        (47 * 1u)    /* 0xFFE6 Watchdog Timer */
#define RTC_VECTOR        (48 * 1u)    /* 0xFFE8 RTC */
#define TIMER3_A1_VECTOR  (49 * 1u)    /* 0xFFEA Timer3_A2 CC1, TA */
#define TIMER3_A0_VECTOR  (50 * 1u)    /* 0xFFEC Timer3_A2 CC0 */
#define TIMER2_A1_VECTOR  (51 * 1u)    /* 0xFFEE Timer2_A2 CC1, TA */
#define TIMER2_A0_VECTOR  (52 * 1u)    /* 0xFFF0 Timer2_A2 CC0 */
#define TIMER1_A1_VECTOR  (53 * 1u)    /* 0xFFF2 Timer1_A3 CC1-2, TA */
#define TIMER1_A0_VECTOR  (54 * 1u)    /* 0xFFF4 Timer1_A3 CC0 */
#define TIMER0_A1_VECTOR  (55 * 1u)    /* 0xFFF6 Timer0_A3 CC1-2, TA */
#define TIMER0_A0_VECTOR  (56 * 1u)    /* 0xFFE8 Timer0_A3 CC0 */
#define UNMI_VECTOR       (57 * 1u)    /* 0xFFFA User Non-maskable */
#define SYSNMI_VECTOR     (58 * 1u)    /* 0xFFFC System Non-maskable */
#define RESET_VECTOR      (59 * 1u)    /* 0xFFFE Reset [Highest Priority] */
```

## Block of Interrupt Control Logic

Let's now go back to the block of interrupt control logic. It carries out several processes. It monitors the state to every IFG bit. When it sees a set flag, it determines whether or not the

, then it loads the vector into the CPU, and then it puts the microcontroller into the active operating mode so it can execute the ISR.

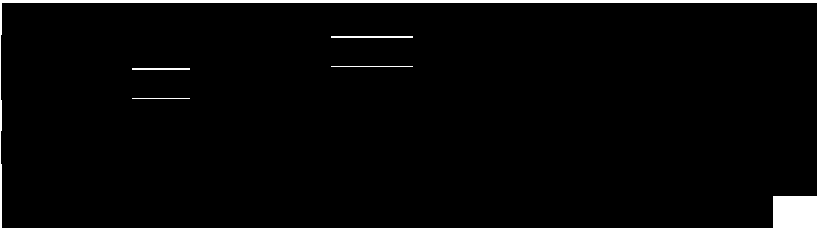
## Reset, Non-Maskable (NMI), and Maskable Types of Interruptions

Interruptions are divided into reset, non-maskable, and maskable types of interruptions. All reset flags are bound to the `RESET_VECTOR`, as shown in diagram 71, and it has the highest priority. It is only used for resetting the microcontroller, and its request cannot be stopped from interrupting the microcontroller. Furthermore, it cannot be used for creating and executing an ISR. Since more than one flag is bound to it, the interrupt system logic uses the reset flag to decide on whether to produce a BOR, POR, or PUC signal. A routine, inside of `RESET_VECTOR`, is used for distinguishing which `RESET_VECTOR` had caused the reset and then transfers the flow of execution to the proper subroutine to handle it. That routine is called the `RESET_VECTOR` handler (see page 211). A reset is typically caused by an event which has stopped or may possibly stop the microcontroller from doing its work.

The non-maskable interruption is referred to as the NMI. Like the reset interruption, its request cannot be blocked from interrupting the microcontroller, but it does not cause any type of reset. It causes a specific ISR to be executed.

Most events which cause an NMI are operating errors and violations which will not stop the microcontroller from doing its work, but should be diagnosed and properly dispositioned with an ISR.

The NMI is divided into system non-maskable interruptions (SNMI) and user non-maskable interruptions (UNMI). An SNMI is typically caused by an instruction trying to access (read) a vacant address in memory, a JTAG mailbox error, or an error in reading or writing data into a section of memory made of FRAM technology. A UNMI is typically caused by an oscillator fault (producing a bad clock signal) or a signal at the  $\overline{\text{RST}}/\text{NMI}$  pin, while the pin is in NMI mode.



The maskable interruption, or just simply referred to as an interruption, causes a specific ISR to be executed. It is the type of interruption which our program is primarily concerned with. The ISR uses the input signals (located in registers) from peripheral modules for making decisions, and the results of those decisions are used for producing output signals (which are driven by registers). The output signals are handled by the same modules, or other modules, or both. The vectors for these ISRs are shown by diagram 71 as having a priority of 56 and lower. While the ISR is being executed, the interrupt system will block requests from all maskable interruptions, but requests for reset and NMIs will be accepted. It is possible to place an instruction in the ISR to enable maskable interruptions, and that technique is called nested interruptions.

---

### How the Interruption is Processed

The process for the non-maskable interruption is described here, and it is elaborated upon by a later chapter. And the specific reset and non-maskable types of interruptions are described later by their own separate chapters.

The status register, program counter, and program execution stack are all involved in processing an interruption. The actual details, as described here, are not a prerequisite to writing an ISR, but having the knowledge gives you better contextual understanding.

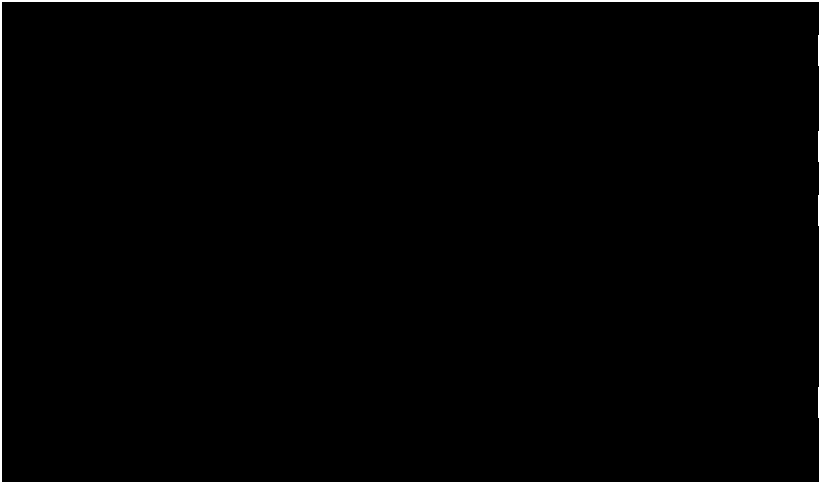
The status register and program counter register are CPU registers. The status register contains bitfields used for putting the microcontroller into the various operating modes and for enabling maskable interruptions. The program counter is a block of

CPU logic whose job is to keep track of which instruction is being executed and which instruction to execute next. It carries out its work by handling the addresses to those instructions. It uses an instruction register to hold the instruction which the CPU is executing. It uses the program counter register for storing the address to the next instruction which the CPU must execute, and it updates the register with the next address as the CPU steps through our program. For example, the reset system places the first instruction in the boot program into the program counter before releasing the microcontroller to the active operating mode.

The program execution stack is a data structure created and placed into main memory by the MSP430 compiler. When the flow of execution transfers away from the main flow of instructions, such as to a function, it basically keeps track of the address to the next instruction in the main flow. That address is where the flow of execution resumes or returns to after completing the function.

---

### **Transfer of Program Execution to the ISR**



---

### **Execution while inside of the ISR**

If the flag is the only one bound to the vector, meaning, it is the only possible source of interruptions for the vector, then typically the interrupt system will automatically clear it.

If more than one flag is bound to the vector, then the ISR will have to decide which flag had caused the interruption, and then transfer the flow to the proper ISR subroutine to handle the event and clear the flag. For example, when distinguishing a channel flag from other channel flags at a port.

If the ISR had read an interrupt vector register (IVR) to learn which flag had caused the interruption, the reading process will typically clear the flag.



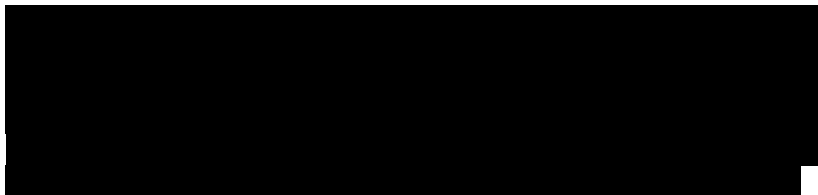
If the objective of the ISR includes being interrupted by maskable interruptions, then it may include an instruction that enables such interruptions. That ability is referred to as nested interruptions.

A subroutine then handles the event. This means the subroutine gets the input data from some system or peripheral module, uses it to make a decision, the decision produces a result, and the result is used to produce an output signal. The signal will typically drive a peripheral module, or update a set of data, or both.

If the objective of the ISR includes changing the operating mode which the flow of execution returns to after executing the ISR, then it may include an instruction that changes the status register bits which are stored on the stack.

---

### **Transfer of Program Execution from the ISR back to the Low Powered State**



---

### **Interrupt Prioritization**

While the CPU is executing an ISR, the interrupt system is still monitoring for set flags. If a flag for an NMI gets set, the CPU will be interrupted. If a flag for a maskable interruption gets set, it will be prioritized with the other flags and handled after the current ISR has been executed. All interruptions are prioritized by their vector number.


When a signal sets the flag for a maskable interruption, signals for the same flag will be ignored until that same flag is cleared. For example, when the flag for channel 0 of port 1 (PIIFG.0) is set, additional signals which continue in attempting to set that same flag will be ignored until that flag is cleared.


Flags codes which are presented by an interrupt vector register (IVR) are prioritized by the register. The priority is published by the register's description table. For example, an IVR for the flags in a digital I/O port will prioritize the channel flags from zero to 7 with channel zero being first.

---

### **Interrupt Compare Controller (ICC)**

The interrupt system is responsible for monitoring interrupt flags, accepting their requests, and prioritizing those requests. All flags are bound to their own interrupt vectors. Those vectors are located in the microcontroller's base header file, and they explicitly define their own priority number. Therefore each flag's priority is defined by the vector which they are bound to.

Some microcontrollers now include a 





## How to Determine which are the Multi-Flagged Vectors

The MSP430 and the programs we develop for it are event-driven. If the event had set an interrupt flag which is bound to a single flag vector, then typically that vector's ISR will not need to include instructions which determine which flag had caused the interruption. On the other hand, if the event had set a flag which is bound to a multi-flag vector, then that vector's ISR will need to include instructions which determine which flag had caused the interruption.

For example, let's look at a digital I/O port. The first two ports on a microcontroller are typically able to interrupt the CPU. They are typically built of eight channels, and each channel is able to set a flag when it senses an event. Furthermore, a port is bound to just a single vector, and a vector can only be bound to a single ISR. So we must take into account that eight flags are bound to a single vector, and the vector is bound to a single ISR. This means the ISR must include a routine which can determine which flag had caused the interruption, and then transfer the flow of execution to the proper subroutine of instructions for clearing the flag and handling its event.

To develop code which searches for the flag and transfers flow to the proper subroutine, we need to know how to determine which are the multi-flagged vectors and how to find the variable name of the register where the flag is located.

---

### Determining which Vectors are Bound to More than one Flag

To learn which vectors are bound to more than one flag,

---

### Finding the Name of the Register where the Flag is Located

We know that a flag will be in the form of a single bitfield, or a code number, or both. When it is in the form of a bitfield, it appears as a field in a conventional register. When it is in the form of a code number, it appears as a set of bitfields in an interrupt vector register (IVR). A microcontroller will have conventional registers, or interrupt vector registers, or both for presenting which flag had interrupted the CPU.

The following table shows all the interrupt flag names for each vector, but it does not provide the register name where the flag is located. A name will typically end with IFG. To find the name of the register where the flag is located, open the microcontroller's user guide, and then search for the flag name. If you are familiar with the microcontroller, you will probably already know which system or peripheral module has the register where the flag is located, then you can start your search from there. If viewing a PDF of the guide, it's a simple task to use the PDF viewer's search tool.

**Diagram 72:** A typical interrupt vector addresses table as published by a microcontroller's data sheet. In this case, it's for the MSP430FR2433. Although the vector's name is not shown, its [word] address in main memory is shown. It can be used as a cross-reference for looking up the vector symbolic constant in the microcontroller's header file.

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
<b>System Reset</b> Power up, Brownout, Supply supervisor External reset RST Watchdog time-out, Key violation FRAM access time error FRAM uncorrectable bit error detection Software POR, BOR FLL unlock error	PMMPORIFG, PMMBORIFG, SVSHIFG PMMRSTIFG WDTIFG ACCTEIFG UBDIFG SYSRSTIV FLLUNLOCKIFG	Reset	FFFEh	59, Highest
<b>System NMI</b> Vacant memory access JTAG mailbox FRAM bit error detection	VMAIFG JMBINIFG, JMBOUTIFG CBDIFG, UBDIFG	Nonmaskable	FFFCh	58
<b>User NMI</b> External NMI Oscillator fault	NMIIFG OFIFG	Nonmaskable	FFFAh	57
Timer0_A3	TA0CCR0 CCIFG0	Maskable	FFF8h	56
Timer0_A3	TA0CCR1 CCIFG1, TA0CCR2 CCIFG2, TA0IFG (TA0IV)	Maskable	FFF6h	55
Timer1_A3	TA1CCR0 CCIFG0	Maskable	FFF4h	54
Timer1_A3	TA1CCR1 CCIFG1, TA1CCR2 CCIFG2, TA1IFG (TA1IV)	Maskable	FFF2h	53
Timer2_A2	TA2CCR0 CCIFG0	Maskable	FFF0h	52
Timer2_A2	TA2CCR1 CCIFG1, TA2IFG (TA2IV)		FFEEh	51
Timer3_A2	TA3CCR0 CCIFG0	Maskable	FFEEh	50
Timer3_A2	TA3CCR1 CCIFG1, TA3IFG (TA3IV)		FFEAh	49
RTC	RTCIFG	Maskable	FFE8h	48
Watchdog timer interval mode	WDTIFG	Maskable	FFE6h	47
eUSCI_A0 receive or transmit	UCTXCPTIFG, UCSTTIFG, UCRXIFG, UCTXIFG (UART mode) UCRXIFG, UCTXIFG (SPI mode) (UCA0IV)	Maskable	FFE4h	46
eUSCI_A1 receive or transmit	UCTXCPTIFG, UCSTTIFG, UCRXIFG, UCTXIFG (UART mode) UCRXIFG, UCTXIFG (SPI mode) (UCA1IV)	Maskable	FFE2h	45
eUSCI_B0 receive or transmit	UCB0RXIFG, UCB0TXIFG (SPI mode) UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG (I <sup>2</sup> C mode) (UCB0IV)	Maskable	FFE0h	44
ADC	ADCIFG0, ADCINIFG, ADCLOIFG, ADCHIIFG, ADCTOVIFG, ADCOVIFG (ADCIV)	Maskable	FFDEh	43
P1	P1IFG.0 to P1IFG.7 (P1IV)	Maskable	FFDCh	42
P2	P2IFG.0 to P2IFG.7 (P2IV)	Maskable	FFDAh	41 Lowest
Reserved	Reserved	Maskable	FFD6h to FF88h	
Signatures	BSL Signature 2		0FF86h	
	BSL Signature 1		0FF84h	
	JTAG Signature 2		0FF82h	
	JTAG Signature 1		0FF80h	

## The Reset Interruption

The highest priority interruption is the reset. It is caused by operating faults, errors, violations, a reset instruction, or just simply a signal from the  $\overline{\text{RST}}/\text{NMI}$  pin while it is in reset mode. The word “system” refers to all microcontroller systems and peripheral modules. A reset instruction is a line of code in our program, and it is just simply an instruction that sets a specific reset interruption flag. Keep in mind that an interrupt flag's ability to request an interruption must be enabled by setting the interrupt enable bit for the flag.

A fault, error, or violation event will or might stop the microcontroller from doing its work. Therefore, they are of enough concern to justify a system reset.

There are three types of reset interruptions, and they are distinguished by

Although a reset interruption flag has its own vector, and unlike all other vectors, this vector is not the address to an ISR. It is the address to the first instruction in the boot program. Consequently, a reset interruption is not used for executing an interrupt service routine (ISR). Instead, we use a reset handler, which is written into `main()`, to handle the interruption. The interrupt logic uses the reset flag for determining which reset signal to produce (a BOR, POR, or PUC). After reset, the boot is then executed.

Probably the most well known reset interruption is caused by a

Another well known reset interruption is caused by a signal at the  $\overline{\text{RST}}/\text{NMI}$  pin, while that pin is in reset mode. The purpose for the  $\overline{\text{RST}}/\text{NMI}$  pin is to provide a manual method for a person or peripheral device to produce a BOR signal. If, instead, the pin is in NMI mode, it will request a non-maskable interruption which triggers an ISR. Since that later mode does not request a reset, the flow of execution for that method is explained by a later chapter.

Systems, not peripheral modules, are the primary components which monitor for events which may cause a system reset. And there are many different types of events

which will set a reset interruption flag. All those events can occur during active operating mode, because everything is on and working. As you progress through the hierarchy of low powered operating modes, there will be a growing number of those events which will not be able to occur, because power had been removed from the system or peripheral module. And as you progress through the hierarchy of fractional operating modes (LPMX.5), there are much less, and typically only three events will set a reset interruption flag that was caused by of some type of operating fault or error: the  $\overline{\text{RST}}/\text{NMI}$  pin in reset mode, the supply voltage supervisor, and the brown-out monitoring system. For events which are not caused by faults and can interrupt a fractional low powered operating mode, their flow is described separately and by a later chapter. To learn which specific events will cause a reset, and from which modes they are enabled, see the Operating Modes (page 131) and the Interrupt Vector Addresses (page 236) sections of the microcontroller's data sheet.

And as a reminder, a reset interruption flag must be enabled so it may request to interrupt the CPU. Each flag has its own enabling bit which can be set or cleared.

### Flow for the System Reset Interruption

**List 1:** The flow of execution for system reset interruption. It may occur while the microcontroller is in any operating mode, but most may only occur during the active mode. Events which cause this interruption are in the form of operating faults, errors, violations, a watchdog timer overflow, or a signal from the  $\overline{\text{RST}}/\text{NMI}$  pin while in reset mode. Depending on the event, its reset signal forces the microcontroller to start a reset from the BOR, POR, or PUC.

1. [REDACTED]
2. [REDACTED]





tion faults. Power-up and reset events will typically set an interrupt flag showing what had caused the reset.

The purpose of a reset fault handler (RFH) is to determine which reset flag was set and then use that information to decide which subroutine must be used to disposition the event.

If the reset had followed a power-up, typically, no disposition is needed, and the RFH is by-passed. If the reset was caused by an interruption, typically, a disposition is needed, so the RFH is entered and a subroutine is executed to handle it.

The action which the disposition causes will depend on what you think must be done, if anything at all. For example, an LED could be illuminated for some period of time, or a message could be sent out a communications module, or a routine can be executed that carries out sophisticated corrective action, or the disposition could be nothing at all. It really depends on the microcontroller's mission and the features you develop to carry out that mission.

Be aware that a few reset interruptions will not set a flag. For example, a

Also be aware of this. For microcontrollers which offer a fractional low power operating mode (LPMx.5), a reset flag, typically named `IFLPMx5` is set when exiting that mode. This is not a flag that a fault handler should check and disposition, since an exit from some fractional low powered mode is not caused by a fault. When the MSP430 is interrupted from such a mode, it always emerges through a full reset (BOR, POR, and PUC). Events which cause such an interruption are expected to

immediately after they are enabled, the ISR for the interruption is loaded into the CPU and executed. An instruction in the ISR then clears the flag. The LPMx.5 interruption is described by a later chapter.

Two reset fault handlers are presented here. The first one uses `if()` statements for reading the state of every reset flag in a conventional register. The second one uses a `switch()` statement for reading the flag code in an interrupt vector generator (IVG) register.

To modularize your program, these handlers can be placed inside of a function. It would not need to return a value, nor would it need to have parameters for taking values. For example, it could be defined as `void resetFaultHandler(void){}`, have its prototype as `void resetFaultHandler(void)`, and be called by an instruction as

`resetFaultHandler()`. The prototype would be placed before `main()`. The function definition would be placed after `main()`. And the function call would be placed inside of `main()`, but before maskable interruptions are enabled.

The following reset fault handlers will be placed inside of that `resetFaultHandler()` function.

### Reset Fault Handler Based on `if()` Statements

Use this handler when your microcontroller does not offer an interrupt vector generator register for getting the code for a reset flag.

When compared to the handler based on a `switch()`, this one involves more work to develop because you have to find which flags are set by a system fault, then determine which registers hold those flags, and then write decision making instructions that reads every one of those registers. Every statement will also have to clear its own flag. And if the flag is in a protected register, the register will have to be unlocked.

To learn about which faults cause a reset, see the “System Module Interrupt Vector Registers” section of the data sheet, and the “System Resets, Interrupts, and Operating Modes” chapter of the user guide.

Code example 69 defines a reset fault handling function. It is called from inside of `main()`. At line 1 is the function's header. It is void of returning data, and it is void of any parameters.

**Code Example 69:** A reset fault handler function that is based on `if()` statements.

```

1  void resetFaultHandler() { // reset fault handler function header
2      PMM_UNLOCK; // unlock the PMM registers
3      MPU_UNLOCK; // unlock the MPU registers
4      if (BOR == 1) { // if BOR caused by RST/NMI pin in reset mode
5          BOR = 0; // clear the flag
6      } //
7  } // end of first disposition routine
8      if (BOR == 1) // if BOR caused by
9      } // if BOR caused by
10     if (POR == 1) // if POR caused by
11     } // if PUC caused by
12     if (PUC == 1) // if PUC caused by
13     } // if PUC caused by
14     if (PUC == 1) // if PUC caused by
15     } // if PUC caused by
16     if (PUC == 1) // if PUC caused by
17     } // if PUC caused by
18 } // end of resetFaultHandler()

```

The function begins with a couple of instructions which unlock registers. They hold system reset flags, and they are typically locked from writing access. If a flag was set, the register must be unlocked to clear it. In this case, the microcontroller has four registers which contain all the flags we intend to read. Two of them are locked.

On line 2, a power management module control register (`PMM_UNLOCK`) is used for unlocking all the PMM registers. After reading about this register in the user guide,

we learn that one bitfield (SVSHE) is set by the reset system, so the unlocking instruction must take that into account. On line 3, an instruction uses the memory protection unit control register (MPUCTL0) for unlocking all the MPU registers.

On lines 4 through 7 is the first subroutine, in the form of an `if()` statement, which uses an operation for reading a flag and uses the result to decide what to do. The operation is called the controlling expression. If the result of the expression is 1, the flow of execution is transferred to the body of the statement. If the result is zero, the flow is transferred to the next routine on line 8.

The controlling expression is in the form of an operation that reads a single bit for a flag ( [REDACTED] ) in a sixteen bit register (PMMIFG). If the flag is set, the expression will return 1, and the body will be executed.

At line 5 of the body, an operation clears [REDACTED]

The remaining routines on lines 8 through 17 follow the same pattern of development for each possible reset flag. The bodies of those routines are empty to keep the example brief. And the actual register variables and masks may be different for your microcontroller.

### Reset Fault Handler Based on a `switch()` Statement

Use this handler when your microcontroller does offer a single register for determining which flag had caused the reset. It is referred to as an interrupt vector register (IVR). The register is typically called the System Reset Interrupt Vector (SYSRSTIV).

When compared to the previous method, [REDACTED]

The code for each flag is published by the microcontroller's data sheet in a table named System Module Interrupt Vector Registers. A portion of such a table is shown by diagram 73. It typically has five columns and many rows. The first column shows the register's variable name. In this case, its name is `SYSRSTIV`. The second column shows the address in main memory where the register is located, but we do not need it. The third column shows a list of reset events, their masks, and a brief description. The fourth column, titled Value, is the code that the interrupt system will write into the register to identify the flag which caused the reset. That is the code, in hexadecimal notation, that our reset handler will be reading. Important: notice that the set of codes are all even numbers, except for zero. That means we can use the `__even_in_range()` intrinsic function for reading the register.

**Diagram 73:** Portion of a System Module Interrupt Vector Registers table as published by the microcontroller's data sheet. Use it for [REDACTED]

INTERRUPT VECTOR REGISTER	ADDRESS	INTERRUPT EVENT	VALUE	PRIORITY
SYSRSTIV, System Reset	019Eh	No interrupt pending	00h	
		Brownout (BOR)	02h	Highest
		RSTIFG RST/NMI (BOR)	04h	
		PMMSWBOR software BOR (BOR)	06h	
		LPMx.5 wakeup (BOR)	08h	
		Security violation (BOR)	0Ah	
		Reserved	0Ch	
		SVSHIFG SVSH event (BOR)	0Eh	
		Reserved	10h	
		Reserved		

Code example 70 defines a reset fault handling function that is based on a `switch()`. It is used for reading the reset flag codes in the [REDACTED] register, an interrupt vector register.

**Code Example 70:** A single register reset fault handler. Use this handler when your microcontroller does offer an interrupt vector register (IVR) for determining which interrupt flag had caused a system reset. The register is typically [REDACTED]

```

1 [REDACTED] // Reset fault handler function header
2 [REDACTED] {
3     [REDACTED] // Brownout (BOR)
4     [REDACTED]
5     [REDACTED]
6     [REDACTED] // RSTIFG RST/NMI (BOR)
7     [REDACTED]
8     [REDACTED]
9     [REDACTED] // Security violation (BOR)
10    [REDACTED] utine
11    [REDACTED]
12    [REDACTED] E: [REDACTED] // SVSHIFG SVSH event (BOR)
13    [REDACTED]
14    [REDACTED]
15    [REDACTED] // See description on page 244
16    } // End of switch statement
17 } // End of resetFaultHandler()

```

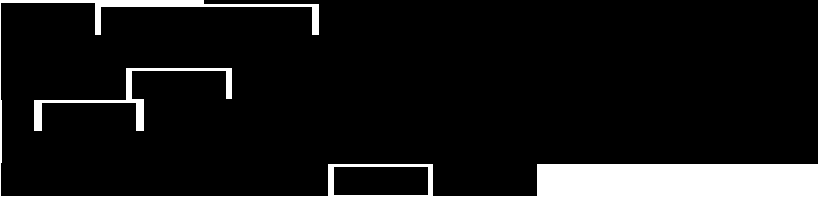
This reset fault handling function would be called by an instruction from inside of `main()`, and this definition would be placed outside of and after `main()`. The function's prototype would be placed before `main()`.

At line 1 is the function's header. It is void of returning data, and it is void of any parameters. The body of the function is in the form of a multiple selection `switch()` statement.

At line 2 is the beginning of the `switch()`. Instead of just simply using the register's variable name (`SYSRSTIV`) as the controlling expression to get the code, we use the [REDACTED] function, as explained on page 208. It will tell the MSP430 compiler to create an optimized data structure out of the `switch()`. For the second parameter (`range`), we use a code number [REDACTED] which is the [REDACTED] number in the range of code numbers, but we could have used the flag's mask.

Reading the register will automatically clear it and the flag to zero. The remaining lines of instructions are in the form of different subroutine cases, where each case handles the code for a specific reset flag. If the case code is equal to the controlling expression, the case is selected and then executed, and when finished, the flow of execution breaks out of the `switch()`.

On line 15 is another



For brevity, the code example presents only four cases which make up the `switch()`.

---

### **Reset Caused by a Watchdog Timer Overflow**

The watchdog can be configured to produce a reset or a non-maskable interruption (NMI). When it is configured to be in reset mode, it will produce a reset at PUC signal. When in NMI mode, an overflow can be used for triggering an ISR.

The watchdog is introduced by Chapter 13 on page 89, and the watchdog timer handler is described on page 178.

## How to Write an Interrupt Service Routine (ISR)

Syntax is the set of rules and principles in a programming language which governs the arrangement of operators and operands to create well formed instructions. In this case, we are interested in the syntax for an interrupt service routine (ISR). We'll need to know that syntax before progressing to the next two chapters, which go into detail about NMIs and maskable interruptions. Interruptions are designed and intended to execute ISRs.

There are three types of ISRs which we must be aware of. The first one is the conventional ISR, and it is used for handling NMIs and maskable interruptions. The second ISR is the built-in default ISR. It has the same syntax as the conventional ISR, but it is one which we do not write; it is automatically written for vectors which do not have a conventional ISR, as will be explained. The last one is the custom default ISR. Its purpose is to bind more than one vector to a single ISR. Those last two ISRs are also referred to as trap ISRs.

---

### The Conventional ISR

When a system module or peripheral module is enabled to produce CPU interruptions, a specific type of event at that module will then be able to set an interrupt flag (IFG) which is unique to that event. The flag then produces a request for interruption signal (IRQ), which is sensed by the interrupt system. The system then uses the flag to select the proper ISR to service the event. A relationship between the flag and the ISR must be established to make that happen. That relationship is created with an interrupt vector and a vector `#pragma` preprocessor directive.



The format for an ISR function is similar to any other conventional C language function, but the function's signature (a combination of the function's name and its parameters) is slightly different.

An ISR's signature starts with the MSP430 intrinsic key word `__interrupt`, and it is followed by the function's return type. Older MSP430 compilers require the double lower line prefix, while newer compilers do not require any prefix. An ISR does not return data, so its return type is always `void`. Following the return type is the function's name. For the name, you may use any name as an identifier which complies with the C language and the MSP430 compiler. And finally, an ISR function is always void of parameters. We cannot pass data into an ISR.

Following the ISR signature is a block of ISR instructions which form the body of the ISR. These instructions clear the flag and handle the event which had caused the interruption.

Every ISR ends with the return from interrupt (RETI) instruction (that was introduced on page 232). The MSP430 C compiler will automatically insert it as the last instruction in the body of the ISR, so we do not have to write it. It puts the microcontroller back into the operating mode from where it was interrupted and reloads the next program instruction into the CPU program counter register. RETI will always become the last instruction of the body.

The following code example shows the syntax for the conventional ISR.

**Code Example 71:** Syntax for the conventional ISR.

---

```

1 ISR_NAME __interrupt void ISR_NAME //Bind this vector to the following ISR
2 ISR_NAME __interrupt void ISR_NAME //Signature for the ISR function
3
4 return __no_operation __no_operation // End of ISR

```

---

One line 1 are two operands and an operator. The first operand is the

On line 2 is the ISR function signature. It is specified (declared) with

The name of the ISR function is any name you choose. In the example, it appears as `ISR_NAME`. It only has to comply with the C language and MSP430 compiler syntax.

Line 2 ends with the [REDACTED]

Line 3, and any other needed lines, is the [REDACTED]

The following example shows code for an ISR which will handle an interruption at a digital I/O port. It's for all the channels in Port 1. The body of the ISR will have to determine which channel had set the flag (page 207), then clear it, and then execute a subroutine which handles the event.

**Code Example 72:** The syntax for an interrupt service routine that will handle interruptions from Port 1.

```

1 [REDACTED] //Bind [REDACTED]
2 [REDACTED] //Declaring the ISR: any valid name can be used
3 [REDACTED] //Opening bracket to the ISR block
4 [REDACTED]
5 [REDACTED] //Closing bracket to the ISR block

```

### Built-in Default Interrupt Service Routine (ISR)

In a scenario where one or more modules have been enabled to produce CPU interruptions, and we have not written ISRs to handle those interruptions, the MSP430 compiler will automatically create a default ISR function for them. This is a feature built into the MSP430 compiler.

But keep in mind that this default ISR will only execute a single instruction. It puts the microcontroller into Low Power Operating Mode 0 (LPM0). That mode will typically just turn off the main clock signal to all peripheral modules, systems, and the CPU.

To verify the actual operating mode, [REDACTED]

[REDACTED] It contains a comment that will state which operating mode the default ISR will put the microcontroller into.



## Customized Default Interrupt Service Routine (ISR)

The alternative to the built-in default ISR is a customized default ISR. There are two types. The first type is used for handling a specific set of interrupt vectors. The second type is used for handling all interrupt vectors which are not handled by an ISR.

When written in C, both types will automatically include the RETI instruction which returns the flow of execution back to the operating mode from where the ISR was interrupted.

From a coding perspective, the syntax for these two types of ISR functions is just like the `ISR(VECTOR)`.

Use the first type of ISR, shown below, to handle a specific set of vectors. You just simply use a comma separated list of vectors to form the set. There is no comma after the last vector. The first line of the example ends with a comma and ellipses (...), meaning, additional vectors may follow. The ellipses is not part of the final syntax which completes the instruction. The name of the ISR is `ISR(VECTOR)`, but it can be `ISR(VECTOR)`.

**Code Example 73:** The custom default ISR that will handle a specific set of interrupt vectors.

```

1 ISR(VECTOR)
2 { //declaring the custom default ISR
3 { //opening bracket to the ISR block
4     //block of ISR instructions
5 } //closing bracket to the ISR block

```

Use the second type of ISR for collecting all unused vectors into a single ISR, we just use the `unused_interrupts` intrinsic keyword as the name for the vector.

**Code Example 74:** Declaring a custom default ISR that will handle all interrupt vectors which are not handled by an ISR.

```

1 ISR(unused_interrupts)
2 {
3 { //opening bracket to the ISR block
4     //block of ISR instructions
5 } //closing bracket to the ISR block

```

Like the conventional ISR function, you may choose any `ISR(VECTOR)`.

## Non-Maskable Interruption (NMI)

Next in priority, after the reset interruption, is the non-maskable interruption (NMI). The events which cause an NMI are similar to the events which cause a reset interruption. Those events are operating faults, errors, and program execution violations. But what distinguishes the NMI events from reset events is one characteristic. Events which cause an NMI will typically not stop the microcontroller from doing its work. They are most probably recoverable without any reset. But they are of enough concern to justify the second highest priority interruption which executes an ISR to properly disposition the fault causing event.

One type of recoverable fault event, for example, is an oscillator fault. It will probably cause errors in flow of execution, so you are going to want to use an ISR to pause the microcontroller until a stable clock signal returns (see page 182).

The  $\overline{\text{RST}}$ /NMI pin is a physical terminal on the microcontroller's case. It is used for producing a non-maskable interruption signal. But that signal can be used for triggering a brown-out reset (BOR) or for executing a specific ISR. When the pin is configured to reset mode, it produces an NMI that causes a BOR. When it is configured to NMI mode, it produces an NMI that triggers a specific ISR. The pin provides an external signaling point of high priority where the microcontroller can be manually restarted or for triggering an ISR.

The interrupt flag for an NMI will typically be used for executing a conventional interrupt service routine (ISR). However, there are a few NMI flags which can trigger either an ISR or a reset. For example, the NMI flag for a FRAM uncorrectable bit error. That flag can be the trigger for an ISR or a reset. For it to cause a reset, another bitfield must be set; it's called the Enable Power-Up Clear (PUC) Reset bit (UBDRSTEN).

There are two classes of NMIs.

System NMIs are in the form of

The flow of execution for an NMI is like the flow for a maskable interruption, the only distinguishing differences are the NMIs have higher priorities, and they cannot be blocked from interrupting the CPU. It can only be interrupted by another NMI

with a higher priority or a reset interruption. Typically, like all other interrupt flags, a flag for an NMI must be enabled through its own unique interrupt enable bit.

And finally, an NMI can

It contains a list of all the NMI flags for a specific microcontroller.

### Flow for the Non-Maskable Interruption (NMI)

The sequence of events which lead up to and which occur during and after a non-maskable interruption (NMI) are shown by List 2. Some of the events are produced by systems built into the microcontroller. Other events are caused by program instructions. In either case, the source of the event is implied when not stated.

The process presents a scenario where the flow of execution begins when it enters the `main()` function because of a power-up event. The last instruction in `main()` puts the microcontroller into a low powered operating mode.

NMIs are predominately caused by operating faults and violations, so they typically occur while the microcontroller is in the active operating mode (AM). Some NMIs may occur during a low powered operating mode, and fewer may occur during a fractional low powered mode. To keep the process example simple, the NMI will wake the microcontroller from some low powered mode. It doesn't matter which one. But it will not be a fractional low powered mode (LPMx.5) because that is a slightly more complicated process. An interruption from a fractional mode will be explained by Chapter 29 on page 279.

Once the NMI interrupts the microcontroller, the interrupt system goes to work; it selects the proper ISR and then loads it in the CPU where it is executed. After the CPU has gone through the ISR, the microcontroller is automatically put back into the low powered mode from where it was interrupted.

**List 2:** The flow of execution for the non-maskable interruption (NMI).

1. [REDACTED]
2. While the flow of execution is in the `main()` function, the following instructions are executed:

- [REDACTED]

- [Redacted]

3. While the microcontroller is in a low powered mode:

- [Redacted]

4. When an event occurs, it

5. [Redacted]

6. The flag produces a

7. Since the interruption is non-maskable, the request is

8. The interrupt system then executes five basic routines.

- [Redacted]

9. The microcontroller is now in the

10. The CPU now executes the ISR. While inside of the ISR:

- [Redacted]

11. When the flow of execution reaches the last instruction in the ISR, the

[Redacted]

12. The last instruction in the ISR is called the return from interrupt (RETI). It simultaneously executes two stack operations (page 232).

• [REDACTED]

### Non-Maskable ISR Examples

Presented here are non-maskable interruptions (NMIs) that will wake up the microcontroller from low powered mode zero (LPM0) and then tell the CPU to execute a specific interrupt service routine (ISR). Once the CPU is finished with the ISR, the microcontroller is automatically put back to sleep in LPM0.

As a reminder, what distinguishes a non-maskable from a maskable interruption is that a flag set by a non-maskable interrupt request (IRQ) signal cannot be blocked by the general interrupt enable (GIE) bit. That bit is located in the CPU's status register, and it will only block requests from maskable interruptions.

Three examples are presented. One is of a `main()` function. It configures the microcontroller so it will be able to execute the non-maskable ISR. The remaining two examples are of the ISRs. Since the flag for this particular interruption is in a register that has other flags, the ISR will have to determine which flag had made the interruption and then run the subroutine for that flag. This means two programming techniques are available to us for determining the flag and for directing the flow of execution into the proper ISR subroutine. One involves the `if()` selection statement, and the other involves the `switch()` selection statement. Therefore, one ISR example uses the `if()` selection and the other uses the `switch()` selection.

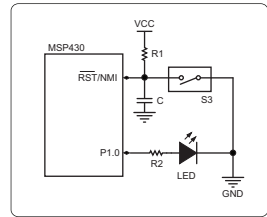
Here's how the set of examples work.

[REDACTED]

Although the examples can be used on all MSP430 microcontrollers with little or no changes, we're going to use the MSP430FR2433 which is built into the MSP-EXP430FR2433 development kit. It provides a push button switch, labeled as S3, that is connected to the  $\overline{RST}/NMI$  pin, and it provides an LED connected to P1.0.

Shown by diagram 74 is the circuit schematic for the NMI input signal and output signal that will be handled by the two ISR examples, described later in this section.

**Diagram 74:** Schematic for examples 75, 76, and 77. The supply voltage (VCC) and a push button switch (S3) are connected to the  $\overline{\text{RST}}/\text{NMI}$  pin of an MSP430. The pin is configured to NMI mode and to set a flag when the voltage signal at the pin falls from VCC to zero. An LED is connected to channel 0 of port 1 (P1.0). When the switch is closed, it pulls the voltage at the pin to zero. That sets the NMIIFG. An ISR then toggles the signal at P1.0. R1, R2, and C are 47k  $\Omega$ , 470  $\Omega$ , and 1,000 pF respectively.



The microcontroller appears with only the two relevant pins. One is the  $\overline{\text{RST}}/\text{NMI}$  pin, and the other is the P1.0 pin. A voltage potential is sent through a resistor-capacitor (RC) signal interface circuit and appears at the  $\overline{\text{RST}}/\text{NMI}$  pin. The potential is at the same level as the supply voltage (VCC). A thorough explanation of the interface circuit is beyond the scope of this discussion, but its purpose isn't. A later volume will go into the design details.

The RC circuit acts as a filter that converts an abrupt change in voltage level into a gradual change. In other words,

After a power-up, the voltage potential at the  $\overline{\text{RST}}/\text{NMI}$  pin becomes equal to VCC, and it will be held there by the power supply without draining current into the channel. When the switch is closed, it creates a short circuit to ground that creates a falling potential difference experienced by the pin. Inside of the microcontroller and behind the pin, the input signal monitoring circuit senses the change of voltage from VCC to ground, so it sets a flag that sends an interrupt request signal (IRQ) to the interrupt system. The flag's request can't be masked, so it's immediately accepted. The system then uses the flag's ISR vector to select the proper ISR, then loads the ISR's first instruction into the CPU, and then puts the microcontroller into active mode so the ISR will be executed. The ISR then tells the CPU to toggle the output signal at P1.0 to change the state of the LED. R2 reduces the output voltage signal to protect the LED.

### main() Function for ISR Code Examples 76 and 77

The purpose of `main()` is to put the  $\overline{\text{RST}}/\text{NMI}$  pin into NMI mode, then configure P1.0 to drive the LED, and then put the microcontroller into LPM0. Both ISR code examples, 76 and 77, use this same `main()` function. Only their IRSSs are different, since they present two different flag determining techniques. Therefore, the `main()` function is separately described here and shown by code example 75.

The routine in `main()` is made with just enough instructions to prepare the microcontroller to execute the ISR, no more and no less. But it still follows the event-driven

pattern of program development that was introduced on page 120. The routine begins on line 3 with an instruction that puts the watchdog on hold.

**Code Example 75:** The `main()` function for code examples 76 and 77. It puts the watchdog on hold, puts the RST/NMI pin into NMI mode, configures P1.0 to drive an LED, unlocks port channels, and then puts the microcontroller into low power mode 0.

---

```

1 // Base header file (see page 146)
2 // Begin main() (see page 99)
3 // Stop the watchdog timer (page 94)
4 // Set RST/NMI pin into NMI mode
5 // Set NMI interrupt to a falling edge
6 // Set to enable NMI flag interruptions
7 // Set P1.0 to output direction
8 // Clear P1.0 output to logical low
9 // Unlock port channels
10 // Put in low powered operating mode 0
11 // Never reached
12 } // End main()

```

---

### Putting the RST/NMI Pin into NMI Mode

Instructions on lines 4, 5, and 6 put the  $\overline{\text{RST}}$ /NMI pin into NMI mode. We first begin by reading the microcontroller's user guide to learn how the pin works, how to configure it, and which registers and bitfields are involved. It can be found on the microcontroller's home page at [www.ti.com](http://www.ti.com), and in this case, its document number is SLAU445. The guide has a dedicated section to this topic called *Reset Pin ( $\overline{\text{RST}}$ /NMI) Configuration*. Two registers are involved. The Special Function Register Reset Pin Control Register (SFRRPCR), as shown by diagram 75, and the Special Function Register Interrupt Enable Register 1 (SFRIE1), as shown by diagram 76. Both names are clumsily recursive, but that's what you can expect. We also learn that most of the bitfields can be used as is. Meaning, they do not have to be changed.



After searching in the adjacent register tables, we will find the bitfield that enables our flag (NMIIFG). The field is called the NMI Pin Interrupt Enable Flag (NMIIE), and it is located in the Special Function Register Interrupt Enable Register 1 (SFRIE1). That register also has bitfields for other flag enabling bits. So on line 6 of `main()`, an instruction sets the NMIIE bit in the SFRIE1 register to enable the NMI interrupt flag (NMIIFG).

**Diagram 75:** Special Function Register Reset Pin Control Register as published by the user guide.

SFRRPCR Register							
15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
Reserved			SYSFLTE	SYSRSTRE	SYSRSTUP	SYSNMIIES	SYSNMI
r0	r0	r0	rw-1	rw-1	rw-1	rw-0	rw-0

SFRRPCR Register Description				
Bit	Field	Type	Reset	Description
15-5	Reserved	R	0h	Reserved. Always reads as 0.
4	SYSFLTE	RW	1h	Reset pin filter enable 0b = Digital filter on reset pin is disabled 1b = Digital filter on reset pin is enabled
3	SYSRSTRE	RW	1h	Reset pin resistor enable 0b = Pullup or pulldown resistor at the RST/NMI pin is disabled 1b = Pullup or pulldown resistor at the RST/NMI pin is enabled
2	SYSRSTUP	RW	1h	Reset resistor pin pullup/pulldown 0b = Pulldown is selected 1b = Pullup is selected
1	SYSNMIIES	RW	0h	NMI edge select. This bit selects the interrupt edge for the NMI when SYSNMI = 1. Modifying this bit can trigger an NMI. Modify this bit when SYSNMI = 0 to avoid triggering an accidental NMI. 0b = NMI on rising edge 1b = NMI on falling edge
0	SYSNMI	RW	0h	NMI select. This bit selects the function for the RST/NMI pin. 0b = Reset function 1b = NMI function

**Diagram 76:** Special Function Register Interrupt Enable Register 1 as published by the user guide.

SFRIE1 Register							
15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
JMBOUTIE	JMBINIE	Reserved	NMIIIE	VMAIE	Reserved	OFIE	WDTIE
rw-0	rw-0	r0	rw-0	rw-0	rw-0	rw-0	rw-0

SFRIE1 Register Description				
Bit	Field	Type	Reset	Description
15-8	Reserved	R	0h	Reserved. Always reads as 0.
7	JMBOUTIE	RW	0h	JTAG mailbox output interrupt enable flag 0b = Interrupts disabled 1b = Interrupts enabled
6	JMBINIE	RW	0h	JTAG mailbox input interrupt enable flag 0b = Interrupts disabled 1b = Interrupts enabled
5	Reserved	RW	0h	Reserved.
4	NMIIIE	RW	0h	NMI pin interrupt enable flag 0b = Interrupts disabled 1b = Interrupts enabled
3	VMAIE	RW	0h	Vacant memory access interrupt enable flag 0b = Interrupts disabled 1b = Interrupts enabled
2	Reserved	R	0h	Reserved. Always reads as 0.
1	OFIE	RW	0h	Oscillator fault interrupt enable flag 0b = Interrupts disabled 1b = Interrupts enabled
0	WDTIE	RW	0h	Watchdog timer interrupt enable. This bit enables the WDTIFG interrupt for interval timer mode. It is not necessary to set this bit for watchdog mode. Because other bits in SFRIE1 may be used for other modules, it is recommended to set or clear this bit using BIS.B or BIC.B instructions, rather than MOV.B or CLR.B instruction. 0b = Interrupts disabled 1b = Interrupts enabled

As will be elaborated upon later, a multi-flagged vector will be used for binding the NMIIFG flag to the ISR. In this case, there are two flags which are bound to that vec-



tor. The other one is the oscillator fault interrupt flag (OFIFG). To keep the `main()` example brief, an instruction that enables that flag (OFIE) is omitted.

## Configuring P1.0 to Drive the LED

Lines 7 and 8 of `main()`, on page 254, configure channel 0 of port 1 to drive the LED. We again refer to the user guide, but this time to learn how to produce an output signal at P1.0. The chapter we need to read is called Digital I/O. We learn that most registers and their bitfields can be used as is, and that we are only concerned with two registers. One is the Port 1 Direction register (P1DIR), and the other is the Port 1 Output register (P1OUT). Since the fields in port registers do not have bitfield names, we'll be using the standard bits for setting and clearing bits in those registers (page 47).



**Diagram 77:** The Port x Direction Register as published by the user guide. The name PxDIR is just simply an abstraction for all port direction registers. Each bitfield controls a channel in port number x. When using the register variable in our program, we just simply substitute the port number for x. For example, P1DIR is the variable for the port 1 direction register.

PxDIR Register							
7	6	5	4	3	2	1	0
PxDIR							
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

P1DIR Register Description				
Bit	Field	Type	Reset	Description
7-0	PxDIR	RW	0h	Port x direction 0b = Port configured as input 1b = Port configured as output

**Diagram 78:** The Port x Output Register as published by the user guide. The name PxOUT is just simply an abstraction for all port output registers. Each bitfield controls a channel in the port. Like the port direction register variable, when using this register variable in our program, we just simply substitute the port number for x. For example, P1OUT is the variable for the port 1 output register.

PxOUT Register							
7	6	5	4	3	2	1	0
PxOUT							
rw	rw	rw	rw	rw	rw	rw	rw

PxOUT Register Description				
Bit	Field	Type	Reset	Description
7-0	PxOUT	RW	Undefined	Port x output When I/O configured to output mode: 0b = Output is low. 1b = Output is high. When I/O configured to input mode and pullups/pulldowns enabled: 0b = Pulldown selected 1b = Pullup selected

## Final Instructions for `main()`

While reading the Digital I/O chapter, we also learned that this microcontroller must have its port channels unlocked before usage. Therefore, on line 9 of `main()` the instruction unlocks the port channels (page 201).

On line 10, the instruction puts the microcontroller into a low powered operating mode zero (page 222), and on line 11 is the `return` statement. In our programming model, the return instruction should never be reached by the flow of execution. But it is there as a good program development practice (page 100).

---

### ISR which uses the `if()` Selection Statement to Determine the NMI Flag

Use the `if()` selection statement in the ISR when the flags for an NMI vector are only available as bitfields in a conventional register. This technique was first introduced on page 207.

---

#### The ISR's Behavior

This example is a continuation of the `main()` function in code example 75, on page 254. That function prepares the microcontroller for work. It

Here's what we want the ISR to do.

---

#### Writing the ISR

Here's how to write that ISR. We begin with knowing that the flag for the  $\overline{RST}/NMI$  pin, while in NMI mode, is `NMIIFG` and it is inside of the `SFRIFG1` register. We learned about all that from the *Reset Pin (RST/NMI) Configuration* section of the microcontroller's user guide.

---

#### Getting the Vector's Name

Now we need to get the vector's name for that `NMIIFG` flag.

Remember that this pin can be put into a mode that triggers a system reset or a mode that triggers a non-maskable request for an ISR. In the Word Address column is the address to the vector in main memory. It shows address number `FFFAh` for the flag's vector. But it's for two flags: the non-

maskable interrupt flag (NMIIFG) and the oscillator fault interrupt flag (OFIFG). Next, we open the microcontroller's base header file (described on page 45), and at the end of that file is a section that defines all the microcontroller's vectors. An image of that section is shown by diagram 71 on page 229. By using the address number `0xFFFFA` as a cross-reference, we can look up the symbolic name for the vector. In this case, it is `UNMI_VECTOR`.

## Binding the Vector to the ISR

We can now write the ISR. It's shown by code example 76. It starts at line 13, since it is the continuation of the program started in code example 75, on page 254.

On line 13 we assign

**Code Example 76:** The non-maskable ISR which uses `if()` selection statements. It is preceded by the `main()` function shown by code example 75. A signal at the RST/NMI pin, while in NMI mode, sets a flag. Since the vector for this ISR is bound to two different flags, the statements determine which flag is set, and then transfers the flow into their body where the flag is cleared and the event is handled. If the NMIIFG is set, its flag is cleared and the output signal at P1.0 is toggled once. If the OFIFG is set, an oscillator fault handler (page 184) manages the fault and clears the flag.

```

13 UNMI_VECTOR // Bind this vector to the ISR
14 void UNMI_ISR // Signature for the ISR
15 { // If NMIIFG is set, then
16     P1_0 ^= 1; // Toggle P1.0 (LED)
17     NMIIFG = 0; // Clear NMIIFG
18 } // End if()
19 if (OFIFG) // If OFIFG is set, then
20     OFIFG = 0; // Clear OFIFG
21 } // End if()
22 } // End of ISR

```

## The ISR's Signature

On line 14 is the signature for our ISR. It is specified with the `void` keyword, and it is further specified as being `void` of any return statement and `void` of any parameters. ISRs must always have those voids. The name for this ISR function is `UNMI_ISR`. It can be any name we choose, but in compliance with the C language. The line ends with the open bracket that delimits the beginning of the function's body.

## The First if() Selection Statement

On line 15 is the beginning of the first `if()` selection statement. It

On line 17 the NMIIFG bitfield in the `SFRIFG1` register is cleared. On line 18 is the closing bracket for this `if()` statement.

**Diagram 79:** The Special Function Register Interrupt Flag 1 register as published by the user guide.

SFRIFG1 Register							
15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
JMBOUTIFG	JMBINIFG	Reserved	NMIIFG	VMAIFG	Reserved	OFIFG	WDTIFG
rw-(1)		rw-(0)		r0		rw-(1)	rw-0

**SFRIFG1 Register Description**

Bit	Field	Type	Reset	Description
15-8	Reserved	R	0h	Reserved. Always reads as 0.
7	JMBOUTIFG	RW	1h	JTAG mailbox output interrupt flag 0b = No interrupt pending. When in 16-bit mode (JMBMODE = 0), this bit is cleared automatically when SYSJMBO0 has been written with a new message to the JTAG module by the CPU. When in 32-bit mode (JMBMODE = 1), this bit is cleared automatically when both SYSJMBO0 and SYSJMBO1 have been written with new messages to the JTAG module by the CPU. This bit is also cleared when the associated vector in SYSUNIV has been read. 1b = Interrupt pending, SYSJMB0x registers are ready for new messages. In 16-bit mode (JMBMODE = 0), SYSJMBO0 has been received by the JTAG module and is ready for a new message from the CPU. In 32-bit mode (JMBMODE = 1), SYSJMBO0 and SYSJMBO1 have been received by the JTAG module and are ready for new messages from the CPU.
6	JMBINIFG	RW	0h	JTAG mailbox input interrupt flag 0b = No interrupt pending. When in 16-bit mode (JMBMODE = 0), this bit is cleared automatically when JMBIO is read by the CPU. When in 32-bit mode (JMBMODE = 1), this bit is cleared automatically when both JMBIO and JMBI1 have been read by the CPU. This bit is also cleared when the associated vector in SYSUNIV has been read 1b = Interrupt pending, a message is waiting in the SYSJMBIx registers. In 16-bit mode (JMBMODE = 0) when JMBIO has been written by the JTAG module. In 32-bit mode (JMBMODE = 1) when JMBIO and JMBI1 have been written by the JTAG module.
5	Reserved	R	0h	Reserved. Always reads as 0.
4	NMIIFG	RW	0h	NMI pin interrupt flag 0b = No interrupt pending 1b = Interrupt pending
3	VMAIFG	RW	0h	Vacant memory access interrupt flag 0b = No interrupt pending 1b = Interrupt pending
2	Reserved	R	0h	Reserved. Always reads as 0.
1	OFIFG	RW	1h	Oscillator fault interrupt flag 0b = No interrupt pending 1b = Interrupt pending
0	WDTIFG	RW	0h	Watchdog timer interrupt flag. In watchdog mode, WDTIFG self clears upon a watchdog timeout event. The SYSRSTIV can be read to determine if the reset was caused by a watchdog timeout event. In interval mode, WDTIFG is reset automatically by servicing the interrupt, or can be reset by software. Because other bits in SFRIFG1 may be used for other modules, it is recommended to set or clear WDTIFG by using BIS.B or BIC.B instructions, rather than MOV.B or CLR.B instructions. 0b = No interrupt pending 1b = Interrupt pending

## The Second if() Selection Statement

On line 19 is the second `if()` statement which is meant to handle an oscillator fault, and is the second flag for this vector. It makes a decision

Be aware that for brevity, the `main()` example does not include an instruction that enables the oscillator fault interrupt flag (OFIFG).

## Returning the Flow of Execution Back to where it was Interrupted

On line 22 is the closing bracket for the ISR function. When the flow of execution passes through this point, the microcontroller is automatically put back into the low

powered mode from where it was interrupted. That return code is automatically added by the MSP430 compiler (see RETI on page 232).

---

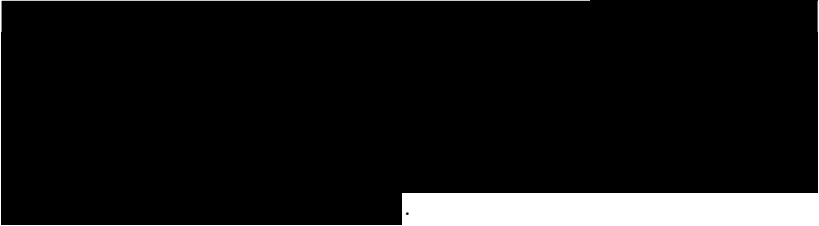
### ISR which uses the switch() Selection Statement to Determine the NMI Flag

Use the switch() selection statement in an ISR when the flags for its vector are available as code numbers in an interrupt vector register (IVR). This technique was first introduced on page 208.

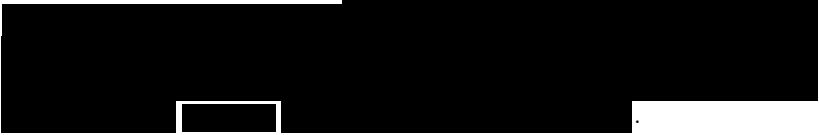
---

#### The ISR's Behavior

Like the previous ISR example, this one is also a continuation of the main() function in code example 75 on page 254. That function prepares



Here's what we want the ISR to do.




---

#### Writing the ISR

By reading the *Reset Pin ( $\overline{RST}/NMI$ ) Configuration* section of the microcontroller's user guide, we learned that the flag for the  $\overline{RST}/NMI$  pin, while in NMI mode, is NMIFG and it is inside of the SFRIFG1 register (shown on page 259).

---

#### Getting the Vector's Name

The same technique that was described earlier (on page 257) is used to get vector's name. We basically used interrupt vector addresses table (diagram 72 on page 236) as published by a microcontroller's data sheet to get the vector's address, and then we used the address and the microcontroller's base header file (page 45) to look up the definition for the vector. So just like the previous example, its name is UNMI\_VECTOR.

---

#### Binding the Vector to the ISR

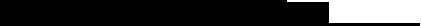




On line 13 we assign `UNMI_VECTOR` to the `#pragma vector` preprocessor directive. That instruction tells the MSP430 C compiler to bind this vector to the following function. That function is specified by the MSP430 intrinsic keyword `__interrupt`.

**Code Example 77:** The non-maskable ISR which uses the `switch()` selection statement. It is a continuation of the `main()` function shown by code example 75 on page 254.

---

```


13  // Bind this vector to the ISR
14  // ISR signature
15  // Begin switch() statement
16  // Case for a set NMIIFG
17  // Toggle P1.0 (LED)
18  // Exit switch()
19  // Case for a set OFIFG
20 
21  // Exit switch()
22  // See description on page 244
23  // End of switch() statement
24 } // End of ISR

```

---


### The ISR's Signature

On line 14 is the signature for our ISR. Like all other ISR signatures, it is specified with the `__interrupt` keyword, and it is further specified as being void of any return statement and void of any parameters. ISRs must always have those voids.

 The line ends with the open bracket that indicates the beginning of the function's body.

### Getting the IVR Register Variable and its Codes

To write the `switch()`, we are going to need the IVR's register variable and the flag code numbers it presents.

We begin at the microcontroller's user guide: 



But all that section tells us is that there are three IVRs, their register variable names, when our program reads the registers, when read, the registers automatically clear the pending flag to zero, and that those registers present flags for a reset, system NMI, and for a user NMI. We now have to refer to other sections of the user guide to determine which IVR keeps track of the user NMIs. It turns out, for our microcontroller, that the User NMI Vector Register (`SYSUNIV`) is the one we are looking for, as shown below by diagram 80. That is the register variable name we'll need for writing the `switch()` statement for our ISR.

Notice that the register description for `SYSUNIV` does not tell us the flag code numbers. Instead, it tells us to read the microcontroller's data sheet to get those numbers.

So we open up the data sheet and search the PDF for the word SYSUNIV, and we find it inside of a table about interrupt vector registers. That table is shown by diagram 81.

**Diagram 80:** The SYSUNIV register table and description as published by a user guide. It is the interrupt vector register that presents user NMI flag code numbers for the MSP430 FR2xx and FR4xx family of micro-controllers.

SYSUNIV Register							
15	14	13	12	11	10	9	8
SYSUNIV							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
SYSUNIV							
r0	r0	r0	r-0	r-0	r-0	r-0	r0

SYSUNIV Register Description				
Bit	Field	Type	Reset	Description
15-0	SYSUNIV	R	0h	User NMI vector. Generates a value that can be used as address offset for fast interrupt service routine handling. Writing to this register clears all pending user NMI flags. See the device-specific data sheet for a list of values.

**Diagram 81:** Current generations of microcontrollers use a system module for handling interruptions. The module typically uses a set of interrupt vector registers for presenting the pending interrupt flag. Shown here is a table of the interrupt vector registers, as published by the microcontroller's data sheet.

System Module Interrupt Vector Registers				
INTERRUPT VECTOR REGISTER	ADDRESS	INTERRUPT EVENT	VALUE	PRIORITY
SYSRSTIV, System Reset	015Eh	No interrupt pending	00h	
		Brownout (BOR)	02h	Highest
		RSTIFG RST/NMI (BOR)	04h	
		PMMSWBOR software BOR (BOR)	06h	
		LPMx.5 wake up (BOR)	08h	
		Security violation (BOR)	0Ah	
		Reserved	0Ch	
		SVSHIFG SVSH event (BOR)	0Eh	
		Reserved	10h	
		Reserved	12h	
		PMMSWPOR software POR (POR)	14h	
		WDTIFG watchdog time-out (PUC)	16h	
		WDTPW password violation (PUC)	18h	
		FRCTLPW password violation (PUC)	1Ah	
		Uncorrectable FRAM bit error detection	1Ch	
		Peripheral area fetch (PUC)	1Eh	
		PMMPW PMM password violation (PUC)	20h	
FLL unlock (PUC)	24h			
Reserved	22h, 26h to 3Eh		Lowest	
SYSSNIV, System NMI	015Ch	No interrupt pending	00h	
		SVS low-power reset entry	02h	Highest
		Uncorrectable FRAM bit error detection	04h	
		Reserved	06h	
		Reserved	08h	
		Reserved	0Ah	
		Reserved	0Ch	
		Reserved	0Eh	
		Reserved	10h	
		VMAIFG Vacant memory access	12h	
		JMBINIFG JTAG mailbox input	14h	
		JMBOUFIG JTAG mailbox output	16h	
		Correctable FRAM bit error detection	18h	
		Reserved	1Ah to 1Eh	
SYSUNIV, User NMI	015Ah	No interrupt pending	00h	
		NMIIFG NMI pin or SVS <sub>H</sub> event	02h	Highest
		OFIFG oscillator fault	04h	
		Reserved	06h to 1Eh	

The table shows that for the `SYSUNIV` register there are two interrupt events and each has their own value. In other words, the events are the flags and the values are their code numbers. When there is no flag pending, there is no interruption waiting to be handled, so the register will present the number zero (`00h`). The address `015Ah` is to the `SYSUNIV` register in main memory. We do not need that information because we'll be using the register's variable name to access the register. The code number for the `NMIIFG` is `02h`, and the code for the `OFIFG` is `04h`. Also notice the Priority column. The `NMIIFG` has a higher priority. That means that when `OFIFG` is being handled, it can be interrupted by a request from an `NMIIFG`.

So now we have what we need for writing the `switch()` statement. We have the name for the IVR, which is `SYSUNIV`, and we have the two codes numbers which the IVR will present for those flags.

### The `switch()` Statement

The `switch()` for an IVR was introduced on page 209. It is constructed of a `switch()` identifier, a decision, and cases. The result of the decision will transfer the flow of execution to the proper case to handle the event. Since the program is going to read an IVR to get the flag code, the register will automatically clear the flag when read. As the decision making expression, the `__even_in_range()` intrinsic function is used (as explained on page 210). Then the statement ends with the `__never_executed()` function (explained on page 244).

The `switch()` starts on line 15. To make the

```

15  switch( __even_in_range(SYSUNIV, 0x02, 0x04) )
16  {
17  case 0x02:
18  case 0x04:
19  break;
20  }
21  __never_executed();

```

On line 16

```

16  {
17  case 0x02:
18  case 0x04:
19  break;
20  }
21  __never_executed();

```

On line 19 is the second case. It handles the

```

19  break;
20  }
21  __never_executed();

```

. Then on line 22, the `break` keyword transfers the flow to line 23 where it exits out of the `switch()`.

Notice that there are no flag clearing instructions. That's because when the IVR recognized it was being read, it automatically cleared the pending flag to zero. Also notice the `__never_executed()` function as the last instruction in the `switch()`. It just simply tells the MSP430 compiler to avoid some typical work that will not be necessary for this `switch()`.



---

**Returning the Flow of Execution Back to where it was Interrupted**

On line 24 is the closing bracket for the ISR function. When the flow of execution passes through this point, the microcontroller is automatically put back into the low powered mode from where it was interrupted. That return code is automatically added by the MSP430 compiler (see RETI on page 232).

## Maskable Interruption

Interruptions are organized by priority. Resets have the highest, which are followed by the non-maskable interruptions (NMIs), and then the maskable interruptions which have the lowest priority. Their priorities are typically shown by an interrupt vector address table, as published by the microcontroller's data sheet. An example of such a table is shown by diagram 72 on page 236. And as that table shows, within the set of maskable interruptions, their interrupt flags are further organized by priority. But what is often left out of that chart is the prioritization of the channels in a port. For example, channel 0 has the highest priority, while each following channel has a progressively lower priority until channel 7 is reached, having the lowest priority.

A maskable interruption is a request which the interrupt system can be configured to refuse. A single bitfield in the CPU status register will tell the system to refuse or accept all requests from maskable interruptions. That field is called General Interrupt Enable (GIE). A PUC will always initialize that field to zero so all maskable requests will not be accepted after a reset. One of the last instructions in `main()` must be used to set that bit so they can all be accepted.

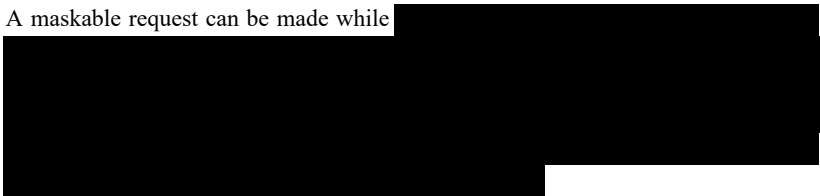
Although reset interruptions and NMIs have higher priorities than maskable interruptions, the maskables are of primary concern for us. Their flags are bound to interrupt service routines (ISRs) which carry out the microcontroller's primary objective.

That objective is to



Most input signals come from peripheral modules, while some come from system modules. And peripheral modules, which are driven by ISRs, are the source for output signals. This event-driven process is shown by diagram 38 on page 110, diagram 42 on page 120, diagram 43 on page 128, and diagram 44 on page 132.

A maskable request can be made while



Here's the rationale for implementing maskable interruptions in a microcontroller. All maskable interruptions are managed as a single set, so if the GIE bit is cleared to zero, their requests to interrupt the CPU will be blocked. Keep in mind that individual maskable interrupt flags still must be enabled if you want to use them. When the

interrupt system accepts a maskable interruption, it sets the GIE bit so other maskable requests will not interrupt the current ISR until its finished. When it's finished, the interrupt system sets the GIE bit to resume accepting maskable interruptions.

Within the set of maskable interrupt flags, each has its own priority number. So when one or more requests are made while the GIE bit is cleared, they will be organized by priority number into a queue and made to wait until GIE is set again. A waiting request is also referred to as a pending request.

We do have the option to interrupt an ISR while in progress. It can be done by placing the interrupt enable instruction inside of the ISR. It's the same instruction as used in the `main()` function. This means that any interrupt occurring during an interrupt service routine can interrupt the routine, regardless of the interrupt priorities. That technique is referred to as interrupt nesting.

---

### Flow for the Maskable Interruption

Use maskable interruptions for executing ISRs which carry out the microcontroller's primary objective. This flow is very much like the flow for the NMI, but NMIs are used for handling operating faults, errors, and violations.

**List 3:** The flow of execution for a maskable interruption.

1. In this scenario, the flow of execution first enters `main()` because of a power-up event.
2. While the flow of execution is in the `main()` function, the following routines are executed:

- [REDACTED]

3. While the microcontroller is in a low powered mode:

- [REDACTED]

4. When an event occurs, it [REDACTED]
5. The block of logic sets the flag for that event.
6. [REDACTED]
7. [REDACTED]
8. The interrupt system then executes four basic routines.
  - [REDACTED]
9. The microcontroller is now in the active operating mode with the program counter register containing the vector (address) which points to the first instruction in the ISR.
10. The CPU executes the ISR. While inside of it, these routines are executed:
  - [REDACTED]
11. When the flow of execution reaches the last instruction in the ISR, the [REDACTED]
12. The last instruction in the ISR is called the return from interrupt (RETI). It simultaneously executes two stack operations (page 232).
  - [REDACTED]

*Continuation of list item 12.* When using the C programming language, the MSP430 compiler automatically adds the RETI instruction to the end of the ISR.

**Diagram 82:** Table of operating modes as published by a microcontroller's data sheet. In this case, it is for the MSP430FR2433. The digital I/O module is shown as being On, Optional, or State Held while in those modes. On means that it is energized and available for use. Optional means that it can be energized or de-energized. State Held means when the microcontroller enters that mode, the port channel will not lose the state which it was in before that mode. For example, when the mode changes from active to LPM3, and the state of a port channel was high, then it will remain high in LMP3.

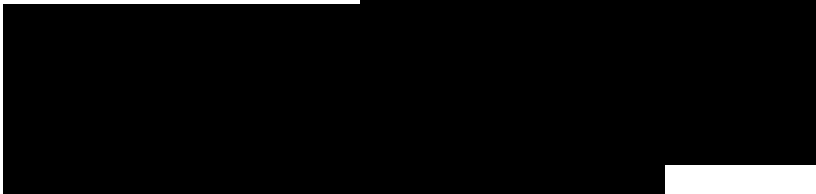
MODE		AM	LPM0	LPM3	LPM4	LPM3.5	LPM4.5
		ACTIVE MODE (FRAM ON)	CPU OFF	STANDBY	OFF	ONLY RTC	SHUTDOWN
Maximum system clock		16 MHz	16 MHz	40 kHz	0	40 kHz	0
Power consumption at 25°C, 3 V		126 µA/MHz	40 µA/MHz	1.2 µA with RTC counter only in LFXT	0.49 µA without SVS	0.73 µA with RTC counter only in LFXT	16 nA without SVS
Wake-up time		N/A	Instant	10 µs	10 µs	350 µs	350 µs
Wake-up events		N/A	All	All	I/O	RTC I/O	I/O
Power	Regulator	Full Regulation	Full Regulation	Partial Power Down	Partial Power Down	Partial Power Down	Power Down
	SVS	On	On	Optional	Optional	Optional	Optional
	Brownout	On	On	On	On	On	On
Clock	MCLK	Active	Off	Off	Off	Off	Off
	SMCLK	Optional	Optional	Off	Off	Off	Off
	FLL	Optional	Optional	Off	Off	Off	Off
	DCO	Optional	Optional	Off	Off	Off	Off
	MODCLK	Optional	Optional	Off	Off	Off	Off
	REFO	Optional	Optional	Optional	Off	Off	Off
	ACLK	Optional	Optional	Optional	Off	Off	Off
	XT1CLK	Optional	Optional	Optional	Off	Optional	Off
Core	VLOCLK	Optional	Optional	Optional	Off	Optional	Off
	CPU	On	Off	Off	Off	Off	Off
	FRAM	On	On	Off	Off	Off	Off
	RAM	On	On	On	On	Off	Off
Peripherals	Backup memory	On	On	On	On	On	Off
	Timer0_A3	Optional	Optional	Optional	Off	Off	Off
	Timer1_A3	Optional	Optional	Optional	Off	Off	Off
	Timer2_A2	Optional	Optional	Optional	Off	Off	Off
	Timer3_A2	Optional	Optional	Optional	Off	Off	Off
	WDT	Optional	Optional	Optional	Off	Off	Off
	eUSCI_A0	Optional	Optional	Off	Off	Off	Off
	eUSCI_A1	Optional	Optional	Off	Off	Off	Off
	eUSCI_B0	Optional	Optional	Off	Off	Off	Off
	CRC	Optional	Optional	Off	Off	Off	Off
ADC	Optional	Optional	Optional	Off	Off	Off	
I/O	RTC	Optional	Optional	Optional	Off	Optional	Off
	General-purpose digital input/output	On	Optional	State Held	State Held	State Held	State Held

## About this Chapter's Examples

This chapter presents two programming examples. Both utilize maskable interrupt flags which are bound to multi-flag interrupt vectors. Therefore, the ISRs will have to distinguish which flag had caused the interruption, and then transfer the flow of execution to the proper subroutine. One example is of an ISR that uses `if()` selection statements to make the decision, the other uses a `switch()` selection statement. And both examples use the event-driven pattern introduced on page 120, and elaborated on by Chapter 20, “Placing the Event-Driven Pattern into a Larger Context,” and Chapter 22, “Event-Driven Programming Routines and Practices.”

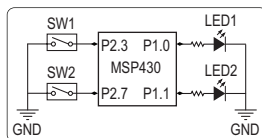
The rationale for using one method over another is based upon how the flag registers are read. The `if()` statement is used for reading conventional registers, while the `switch()` statement is used for reading interrupt vector registers (IVRs). To learn about IVRs, vectors, and multi-flagged vectors, see pages 227, 228, and 235 respectively.

This is how the programs will work.



Shown by diagram 83 is a circuit schematic of the microcontroller, the mechanical switches, and the LEDs. The program will be configuring channels P2.3 and P2.7 as signal inputs and channels P1.0 and P1.1 as outputs. This circuit is built into the MSP-EXP430FR2433 development kit, but the example programs will work on any MSP430 with those peripheral devices connected to those channels.

**Diagram 83:** Schematic for this chapter's programming examples. In this case, it is taken from the MSP-EXP430FR2433 development kit.



The nature of mechanical switches is to bounce when they are opened and closed. That behavior will produce a sequence of input signals that will cause the ISR to run multiple times for every switch actuation. Therefore, this is not a commercial grade circuit. A switch interface circuit, such as a resistor-capacitor (RC) filtering circuit, will produce a clean single input signal. But that is a topic for another volume.

The schematic published by that kit's user guide shows LED1 as red and conditioned with a 470  $\Omega$  resistor and LED2 as green with a 392  $\Omega$  resistor. The bill of materials (BOM), a file supplied with the kit and can be found at the kit's home page, specifies them as Lite-On part numbers LTST-C190CKT and LTST-C190GKT respectively. Their data sheets further specifies the red as drawing 40 mA and the green 30 mA. The MSP430FR2433 data sheet specifies that the typical output current at a channel when VCC is 3 volts is about 23 mA, so the resistors are there as current limiters to protect the microcontroller.

## ISR using the `if()` Selection Statement to Determine which Maskable Flag is Set

Use the `if()` selection statement in the ISR when the flags for a maskable interrupt vector are only available as bitfields in a conventional register. This technique was first introduced on page 207. Such an ISR is shown by code example 78, on page 270.

## The main() function

The purpose of `main()` will be to configure channels 3 and 7 of port 2 so they can sense input signals which will set a maskable flag, then configure channels 0 and 1 of port 1 to produce output signals, then unlock the port channels, then enable maskable interruptions, and then to put the microcontroller into low powered operating mode 0.

The routine in `main()` is made with just enough instructions to prepare the microcontroller to execute the ISR, no more and no less. But it still follows the event-driven pattern of program development that was introduced on page 120. The routine begins on line 3 with an instruction that puts the watchdog on hold.

**Code Example 78:** ISR which uses the `if()` selection statement to determine which maskable flag was set. Use the `if()` selection statement in the ISR when the flags are only available as bitfields in a conventional register. The ISR begins on line 17.

---

```

1 // Base header file (see page 146)
2 // Begin main() (see page 99)
3 // Stop the watchdog timer (page 94)
4 // Set P2.3 & P2.7 signal direction outwards
5 //
6 // Set P2.3 & P2.7 to enable channel interrupts
7 // Set P2.3 & P2.7 transition fr. high to low
8 // Clear P2.3 & P2.7 flags
9 // Set P1.0 & P1.1 direction outwards to LEDs
10 // Clear P1.0 & P1.1 output to darken LEDs
11 // Unlock all port channels (page 201)
12 // Enable maskable interruptions (page 212)
13 // Put in low powered operating mode 0 (page 222)
14 // Never reached
15 // End main()
16 // Bind this vector to the following ISR
17 // ISR signature
18 // If P2.3 IFG is set, then
19 // Toggle P1.0 (LED), and then
20 // Clear P2IFG.3 IFG
21 //
22 // If P2.7 IFG is set, then
23 // Toggle P1.1 (LED), and then
24 // Clear P2IFG.4 IFG
25 // End if
26 // End ISR

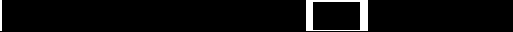
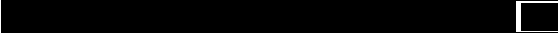
```

---

## Configuring P2.3 and P2.7 to Sense Input Signals

These instructions will configure the input paths and signals from the switches. By reading the Digital I/O chapter of the microcontroller's user guide, we know that five registers will have to be configured. And since the fields in port registers do not have bitfield names, we'll be using the standard bits for setting and clearing bits in those registers (page 47).

Diagrams 77 and 78, on page 256, show the Port x Direction (PxDIR) and Port x Output (PxOUT) registers. So on line 4, we set bitfields 3 and 7 in P2DIR to direct their signals outwards, and then on line 5 we set the same fields in P2OUT to produce a digital high output signal on those same channels. That will create the loop-back signals we need at those channels, as shown by diagram 48 on page 157.

Now we enable those channels to set interrupt flags when they sense a signal transition from high to low. As published by the same Digital I/O chapter, we have to make changes in the Port x Interrupt Edge Select (PxIES) register, the Port x Interrupt Enable (PxIE) register, and for house cleaning purposes, the Port x Interrupt Flag (PxIFG) register. So on line 6  . And to assure those flags are not set during a power-up, on line 8 we clear those flags in P2IFG

**Diagram 84:** PxIES register as published is the microcontroller's user guide.

PxIES Register							
7	6	5	4	3	2	1	0
PxIES							
rw	rw	rw	rw	rw	rw	rw	rw

PxIES Register Description				
Bit	Field	Type	Reset	Description
7-0	PxIES	RW	Undefined	Port x interrupt edge select 0b = PxIFG flag is set with a low-to-high transition 1b = PxIFG flag is set with a high-to-low transition

**Diagram 85:** PxIE register as published is the microcontroller's user guide.

PxIE Register							
7	6	5	4	3	2	1	0
PxIE							
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

PxIE Register Description				
Bit	Field	Type	Reset	Description
7-0	PxIE	RW	0h	Port x interrupt enable 0b = Corresponding port interrupt disabled 1b = Corresponding port interrupt enabled

**Diagram 86:** PxIFG register as published is the microcontroller's user guide.

PxIFG Register							
7	6	5	4	3	2	1	0
PxIFG							
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

PxIFG Register Description				
Bit	Field	Type	Reset	Description
7-0	PxIFG	RW	Undefined	Port x interrupt flag 0b = No interrupt is pending. 1b = Interrupt is pending.

## Configuring P1.0 and P1.1 to Produce Output Signals

These instructions will configure the output paths and signals for toggling the LEDs. On line 9, bits in fields 0 and 1 of P1DIR are set to configure the signal direction outward. On line 10, the same fields, but in P1OUT are set to produce a digitally high output signal.

## Final Instructions for main()

While reading the Digital I/O chapter, we also learned that this microcontroller must have its port channels unlocked before usage. This is common with microcontrollers built with FRAM. Therefore, on line 11 of main(), an instruction unlocks the port channels (page 201).

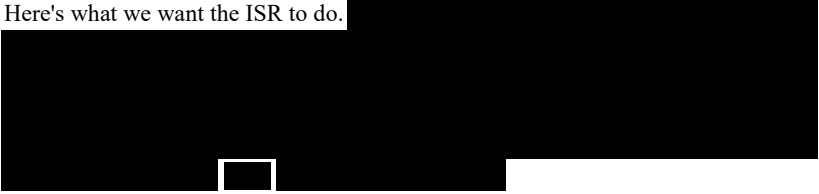


On line 12, maskable interruptions are enabled (page 212). On line 13, the instruction puts the microcontroller into low powered operating mode zero (page 222), and on line 14 is the return statement. In our programming model, the return instruction should never be reached by the flow of execution. But it is there as a good program development practice (page 100).

---

### The ISR's Behavior

Here's what we want the ISR to do.




---

### Writing the ISR

The work needed to write this ISR can be reduced to three steps: 1) getting the interrupt vector's name, 2) binding the vector to the ISR, and then 3) writing the `if()` selection statements.

---

### Port Channel Flag Names

Before proceeding with getting the vector's name for a port, we need to know the names for each port channel flag. This is a simple matter because,

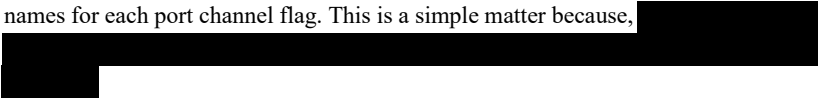
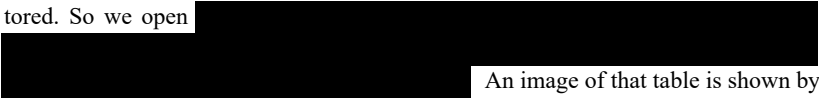


Diagram 86 on page 271, shows the `PxIFG` register. It generically represents any port IFG register. So for example, the flag names for channels 3 and 7 are `BIT3` and `BIT7` respectively. This technique was described by Chapter 9 on page 77.

---

### Getting the Interrupt Vector Name

We need to get the vector's name for all the port 2 interrupt flags which will be monitored. So we open



An image of that table is shown by diagram 72 on page 236, and in this case, it's for the MSP430FR2433.

In the first column, the table shows `P2`. That identifies the port 2 interrupt source. The second column shows the interrupt flag names, but the flag names are not as we would expect. They are shown as `P2IFG.0` to `P2IFG.7`. Those are not the names we use for reading a conventional interrupt flag register, as shown by diagram 86. They are used for reading an interrupt vector register (IVR), and the table shows the register name as `P2IV`. In the next ISR example, we'll be using the `switch()` statement to read the IVR, so ignore all that IVR stuff for now, since we'll be using the standard bits to read a conventional flag register.

In the Word Address column is the address to the vector in main memory. It shows address number FFDAh containing the vector to all the port 2 flags. The suffix h denotes the number as being in hexadecimal notation, but just focus on the number itself: FFDA.

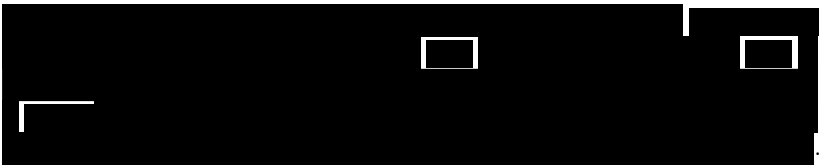
Next, we open the microcontroller's base header file (described on page 45), and in that file is a section that defines all the microcontroller's vectors. An image of that section is shown by diagram 71 on page 229. By searching for the address number FFDA, we can find the symbolic name for the vector. In this case, it is defined as PORT2\_VECTOR, and its address number is prefixed with 0x, which is another way to denote a hexadecimal number.

We can now start writing our ISR, as shown by code example 78 on page 270.

### Binding the Vector to the ISR



### The ISR's Signature



### How Many if() selection Statements to Use



### The First if() Selection Statement

On line 18 is the beginning of the first `if()` selection statement. It reads the state of channel 3's flag (BIT3) and then compares it to the standard bit BIT3 to determine if the flag is set (see page 77 about how that expression works). If the flag is set, the flow of execution enters the body of the statement.

The first instruction in the body, on line 19, toggles the bit in Field 0 of the P1OUT register. Those register fields produce output signals. If the signal is high, the instruction toggles it low, and vice versa. On line 20 the flag for channel 3 in the P2IFG register is cleared. And on line 21 is the closing bracket for this `if()` statement.

### The Second if() Selection Statement

This is very much like the first `if()` statement. On line 22 is the beginning of the second `if()` selection statement. It reads the state of channel 7's flag (BIT7) and then

compares it to the standard bit BIT7 to determine if the flag is set. If the flag is set, the flow of execution enters the body of the statement.

The first instruction in the body, on line 23, toggles the bit in Field 1 of the P1OUT register. On line 24 the flag for channel 7 in the P2IFG register is cleared. And on line 25 is the closing bracket for this `if()` statement.

---

### Returning the Flow of Execution Back to where it was Interrupted

On line 26 is the closing bracket for the ISR function. When the flow of execution passes through this point, the microcontroller is automatically put back into the low powered mode from where it was interrupted. That return code is automatically added by the MSP430 compiler (see RETI on page 231).

---

### ISR using the `switch()` Selection Statement to Determine which Maskable Flag is Set

Use the `switch()` selection statement in the ISR when the flags for a maskable interrupt vector are available as code numbers in an interrupt vector register (IVR). This technique was first introduced on page 208. Such an ISR is shown below.

**Code Example 79:** ISR which uses the `switch()` selection statement to determine which maskable flag was set. Use the `switch()` selection statement in the ISR when the flags are available as code numbers in an interrupt vector register (IVR). The ISR begins on line 17.

---

```

1  // Base header file (see page 146)
2  // Begin main() (see page 99)
3  // Stop the watchdog timer (page 94)
4  // Set signal direction outwards
5  // Set signal output to high
6  // Set to enable channel interrupts
7  // Set transition fr. high to low
8  // Clear P2.3 & P2.7 flags
9  // Set P1.0 & P1.1 direction outwards to LEDs
10 // Clear P1.0 & P1.1 output to darken LEDs
11 // Unlock all port channels (page 201)
12 // Enable maskable interruptions (page 212)
13 // Put in low powered operating mode 0 (page 222)
14 // Never reached
15 // End main()
16 // Bind this vector to the ISR
17 // ISR signature
18 // Begin switch() (see page 208)
19 // Case for a set P2.3 IFG
20 // Toggle P1.0 (LED)
21 // Exit switch()
22 // Case for a set P2.7 IFG
23 // Toggle P1.1 (LED)
24 // Exit switch()
25 // See description on page 244
26 // End of switch() statement
27 // End of ISR

```

---

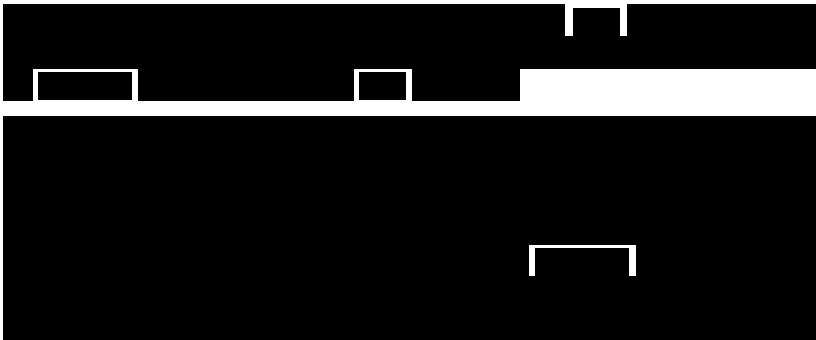
### The `main()` Function

The `main()` function for this example is written exactly as it is written for code example 78. There is no difference. So its purpose is the same. It configures channels 3 and

7 of port 2 so they can sense input signals which will set a maskable flag, then configure channels 0 and 1 of port 1 to produce output signals, then unlock the port channels, then enable maskable interruptions, and then to put the microcontroller into low powered operating mode 0. It also follows the event-driven pattern of program development.

---

### The ISR's Behavior




---

### Writing the ISR

The work needed to write this ISR can be reduced to four steps: 1) getting the port channel flag names, 2) getting the interrupt vector's name, 3) binding the vector to the ISR, and then 4) writing the `switch()` selection statement.

---

### Getting the Port Channel Flag Names

With a conventional port flag register, as shown by diagram 86 on page 271, we have to use the standard bits (page 47) as the channel flag names. With an interrupt vector register (IVR), as shown by diagram 87, we use the flag names as published by the microcontroller `msp430.h` header file. The IVR was introduced on page 209, and see page 45 for instructions about opening the header file.

Diagram 87 shows the Port 2 Interrupt Vector (P2IV) register. It is dedicated to all the port 2 flags. It presents a code number that will be either zero or even number in binary notation. If zero, there is no flag set or pending. If it presents an even code number within the range from 2 (02h) to 10 (10h), then there is a set (pending) flag waiting to be handled. The code number distinguishes the specific flag. The register description shows the flag code numbers in hexadecimal notation from 02h to 10h, but it does not give us the actual names that have to be used in our program. To get those names, we search the header file.

The header file has sections which define the flag names for each IVR. The names are defined as symbolic constants. In this case, it's the flags for P2IV.

Code Example 80 shows the symbolic constant definitions for P2IV section. The first column is the `#define` preprocessor directive. It tells the C compiler to assign the number shown in the third column to the symbolic constant in the second column.

**Diagram 87:** Port 2 Interrupt Vector (IV) register as published in the microcontroller's user guide. In the current example it is for the MSP430FR2433.

P2IV Register							
15	14	13	12	11	10	9	8
P2IV							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
P2IV							
r0	r0	r0	r-0	r-0	r-0	r-0	r0

P2IV Register Description				
Bit	Field	Type	Reset	Description
15-0	P2IV	R	0h	Port 2 interrupt vector value 00h = No interrupt pending 02h = Interrupt Source: Port 2.0 interrupt; Interrupt Flag: P2IFG.0; Interrupt Priority: Highest 04h = Interrupt Source: Port 2.1 interrupt; Interrupt Flag: P2IFG.1 06h = Interrupt Source: Port 2.2 interrupt; Interrupt Flag: P2IFG.2 08h = Interrupt Source: Port 2.3 interrupt; Interrupt Flag: P2IFG.3 0Ah = Interrupt Source: Port 2.4 interrupt; Interrupt Flag: P2IFG.4 0Ch = Interrupt Source: Port 2.5 interrupt; Interrupt Flag: P2IFG.5 0Eh = Interrupt Source: Port 2.6 interrupt; Interrupt Flag: P2IFG.6 10h = Interrupt Source: Port 2.7 interrupt; Interrupt Flag: P2IFG.7; Interrupt Priority: Lowest

The symbolic constant is a number which represents the flag name, and its value is the flag code number which the interrupt vector register will present to us.

**Code Example 80:** Channel flag name definitions for the Port 2 Interrupt Vector (P2IV) register as published by the microcontroller's msp430.h header file.

```

/* No Interrupt pending */
/* P2IV P2IFG.0 */
/* P2IV P2IFG.1 */
/* P2IV P2IFG.2 */
/* P2IV P2IFG.3 */
/* P2IV P2IFG.4 */
/* P2IV P2IFG.5 */
/* P2IV P2IFG.7 */

```

We'll be using the symbolic constants in our `switch()` statement for determining which flag is pending. We only need the symbols for channels 3 and 7 of port 2. They are `P2IV_P2IFG3` and `P2IV_P2IFG7` respectively.

## Getting the Interrupt Vector Name

We now have to get the interrupt vector name, since that will be the name we use for binding the flags to our ISR. So we open our microcontroller's data sheet and go to the Interrupt Vector Addresses section. It publishes a table of all the microcontroller's interrupt flags and the address in main memory to each flag's vector. An image of that table is shown by diagram 72 on page 236, and it's for the MSP430FR2433.

In the first column, the table shows P2. That identifies the interrupt source as being Port 2. The second column shows the interrupt flag names as being P2IFG.0 to P2IFG.7.

Unfortunately, those are not the actual flag names, but metaphors for the names. We know the actual names because in the last section we had found them in the micro-

controller's header file. And right next to those metaphors, and in parenthesis, is the name for the interrupt vector register (IVR).

In the fourth column is the information we need. It shows us the address in main memory where the interrupt vector is located. It shows address number FFDAh as where the vector to all the port 2 flags is stored. The suffix h denotes the number as being in hexadecimal notation, but just focus on the number itself: FFDA.

Next, we



We can now start writing our ISR, as shown by code example 79 on page 274.

---

### Binding the Vector to the ISR




---

#### The ISR's Signature

On line 17 is the function signature for our ISR. It is specified with the `__interrupt` keyword, and it is further specified as being `void` of any return statement and `void` of any parameters. ISRs must always have those `void`s. The name for this ISR function is `PORT2_ISR`. It can be any name we choose, but in compliance with the C language. The line ends with the open bracket that delimits the beginning of the function's body.

---

#### How Many case selection Statements to Use

The `switch()` selection statement is an intrinsic function to the C programming language. So the instruction on line 18 can be viewed as that function's signature.

Within the `switch()` are case statements. The purpose of the case statement is to determine if a specific number is equal to the number which was read or collected by the `switch()`'s signature. If the case number is equal to the number read by the signature, then the flow of execution enters the body of the case.


A case does not have to be written for flags which will not be used. Therefore, use one case per flag that we plan on using. In this example, two flags will be monitored, so only two cases are needed.

---

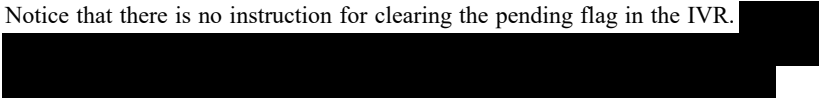
#### The First Case

This case determines whether or not the P2.3 flag is set, and if it is, the flow of execution is then transferred into the body of the case. So on line 19 is the beginning of

the case. It takes the number read by the `switch()` signature and compares it to the case number. For this case, it is `P2IV_P2IFG3`, which is the flag code number for channel 3 of port 2.



Notice that there is no instruction for clearing the pending flag in the IVR.



---

### The Second Case

This case determines whether or not the P2.7 flag is set, and if it is, the flow of execution is then transferred into the body of the case, just like the first case. So on line 22 is the beginning of the case. It takes the number read by the `switch()` signature and compares it to the case number. For this case, it is `P2IV_P2IFG7`, which is the flag code number for channel 7 of port 2.

The first instruction in the body, on line 23, toggles the bit in Field 1 of the P1OUT register. If the signal is high, the instruction toggles it low, and vice versa. Then on line 24 is the `break` statement. It transfers the flow of execution to line 26, out of the `switch()`. And once again, no flag clearing instruction is needed.

---

### Returning the Flow of Execution Back to where it was Interrupted

On line 27 is the closing bracket for the ISR function. When the flow of execution passes through this point, the microcontroller is automatically put back into the low powered mode from where it was interrupted. That returning program code is automatically added by the MSP430 compiler (see RETI on page 232).

## Interruption from Fractional Low Powered Mode (LPMx.5)

Use the interruption from fractional low powered mode (LPMx.5) when you need to put the microcontroller into an operating mode which consumes the least amount of energy. This type of operating mode was introduced on page 131, then by Chapter 22 on page 175, and then elaborated upon on page 223. It is a type of maskable interruption which power is removed from most modules and systems.

The most important characteristics about this operating mode is that 1) a limited set of systems and modules will remain active, 2) that power is removed from memory, therefore, all volatile data, such as data in storage variables and register settings will be lost, and 3) that an interruption will force the microcontroller to go through a complete system reset and then re-enter the `main()` function before the interrupt service routine (ISR) is executed. So we have to take all that into account when developing a program.

For a complete explanation about LPMx.5 and any changes made to it, we have to refer

In the data sheet, we would want to read the sections about the “Functional Block Diagram” and the “Operating Modes.” The diagram will show us which systems and modules are located in the LPMx.5 domain, meaning, which of them will remain active and able to interrupt the microcontroller. Diagram 6, on page 18, presents a typical functional block diagram that shows an LPMx.5 domain. The domain is a boundary that shows which systems and modules are active during LPMx.5. The operating modes section of the data sheet will publish a table that explicitly lists every operating mode and which systems remain active during an operating mode. Diagram 88, on the next page, shows such a table as published by the data sheet for the MSP430FR2344.

At this time, only three types of events are able to interrupt LPMx.5, but future designs may include more. Those events



Data held by a storage variable will be lost during LPMx.5 or when power is disconnected from the microcontroller because such variables are placed in volatile sections of main memory. So to protect a storage variable, we use a volatile data handler (page 217). Handlers can be written to use the `PERSISTENT()` `#pragma` or a backup memory register (BAKMEM). The `#pragma` must be written outside of and before the `main()` function, while the instruction which uses the backup memory register will typically be put inside of `main()` or an interrupt service routine (ISR). Be aware that BAKMEM registers are available only when the Real-Time Clock (RTC) is enabled, but that may change in the future, and they are not available in all microcontrollers.

**Diagram 88:** Table of operating modes as published by the data sheet for the MSP430FR2433. Notice the items for the fractional low powered modes LPM3.5 and LPM4.5, and that the Backup Memory registers (BAKMEM) are not available during LPM4.5. Furthermore, the table does not mention that BAKMEM registers are available only when the Real-Time Clock (RTC) is enabled, at least for now.

**Operating Modes**

MODE		AM	LPM0	LPM3	LPM4	LPM3.5	LPM4.5
		ACTIVE MODE (FRAM ON)	CPU OFF	STANDBY	OFF	ONLY RTC	SHUTDOWN
Maximum system clock		16 MHz	16 MHz	40 kHz	0	40 kHz	0
Power consumption at 25°C, 3 V		126 µA/MHz	40 µA/MHz	1.2 µA with RTC counter only in LFXT	0.49 µA without SVS	0.73 µA with RTC counter only in LFXT	16 nA without SVS
Wake-up time		N/A	Instant	10 µs	10 µs	350 µs	350 µs
Wake-up events		N/A	All	All	I/O	RTC I/O	I/O
Power	Regulator	Full Regulation	Full Regulation	Partial Power Down	Partial Power Down	Partial Power Down	Power Down
	SVS	On	On	Optional	Optional	Optional	Optional
	Brownout	On	On	On	On	On	On
Clock <sup>(1)</sup>	MCLK	Active	Off	Off	Off	Off	Off
	SMCLK	Optional	Optional	Off	Off	Off	Off
	FLL	Optional	Optional	Off	Off	Off	Off
	DCO	Optional	Optional	Off	Off	Off	Off
	MODCLK	Optional	Optional	Off	Off	Off	Off
	REFO	Optional	Optional	Optional	Off	Off	Off
	ACLK	Optional	Optional	Optional	Off	Off	Off
Core	XT1CLK	Optional	Optional	Optional	Off	Optional	Off
	VLOCLK	Optional	Optional	Optional	Off	Optional	Off
	CPU	On	Off	Off	Off	Off	Off
	FRAM	On	On	Off	Off	Off	Off
Peripherals	RAM	On	On	On	On	Off	Off
	Backup memory <sup>(2)</sup>	On	On	On	On	On	Off
	Timer0_A3	Optional	Optional	Optional	Off	Off	Off
	Timer1_A3	Optional	Optional	Optional	Off	Off	Off
	Timer2_A2	Optional	Optional	Optional	Off	Off	Off
	Timer3_A2	Optional	Optional	Optional	Off	Off	Off
	WDT	Optional	Optional	Optional	Off	Off	Off
	eUSCI_A0	Optional	Optional	Off	Off	Off	Off
	eUSCI_A1	Optional	Optional	Off	Off	Off	Off
	eUSCI_B0	Optional	Optional	Off	Off	Off	Off
I/O	CRC	Optional	Optional	Off	Off	Off	Off
	ADC	Optional	Optional	Optional	Off	Off	Off
	RTC	Optional	Optional	Optional	Off	Optional	Off
	General-purpose digital input/output	On	Optional	State Held	State Held	State Held	State Held

(1) The status shown for LPM4 applies to internal clocks only.

(2) Backup memory contains 32 bytes of register space in peripheral memory. See Table 6-24 and Table 6-43 for its memory allocation.

**NOTE**

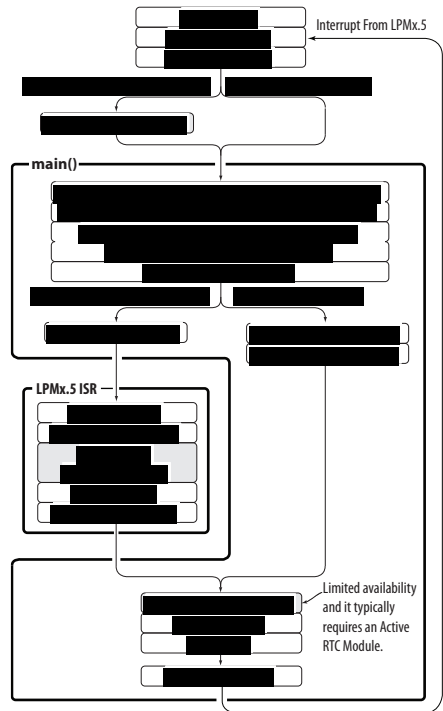
XT1CLK and VLOCLK can be active during LPM4 if requested by low-frequency peripherals, such as RTC or WDT.

Diagram 89, shown below, illustrates an abstract view of the flow of program execution in a program designed to handle an interrupt from LPMx.5. It shows the most fundamental routines and the relationships between them. In other words, a framework which we can build upon. It starts at the power-up event, then goes through a set of routines before entering the `main()` function. One of the last routines before entering `main()` involves the `PERSISTENT() #pragma`. Once inside `main()`, the structure of the routines are very much in line with the event-driven pattern shown by diagram 42 on page 120 and diagram 43 on page 128. The fundamental difference is that an LPMx.5 interrupt handler is used here. It will decide on whether or not the flow of execution had emerged from a power-up event or an interruption from LPMx.5.

**Diagram 89:** Basic flow of routines needed for creating and handling an interruption from a fractional low powered operating mode (LPMx.5). Since volatile data will be lost when put into this mode, it must be saved as non-volatile. That work can be done by using the `PERSISTENT() #pragma`, or the backup memory (BAKMEM) registers, or by writing to a peripheral data storage device. The BAKMEM registers typically depend on an active RTC.

The purpose of the LPMx.5 Interrupt Handler is to transfer the flow of execution to an ISR which handles the LPMx.5 interruption or to avoid it. A program selection structure, such as an `if()` statement (page 106), is used for making the decision. If the flow had emerged from an LPMx.5 interruption, then maskable interruptions are immediately enabled so the ISR which specifically handles interruptions from LPMx.5 can be executed. On the other hand, if the flow had emerged from a power-up event, then all enabled maskable interrupt flags, located outside of the LPMx.5 domain, which were inadvertently set are then first cleared before maskable interruptions are enabled. It is important to keep in mind that

An interruption from LPMx.5 is typically caused by a flag set by a port channel, or the Real-Time Clock (RTC), or the `RST/ANM1` pin in reset mode. See the Operating Modes section of the microcontroller's user guide for more information and updates.



An MSP430 typically provides at least two bitfields that can be read to determine which event had caused the flow of execution to enter `main()`. We must use the microcontroller's user guide to verify the exact names for those bitfields and registers, but here are the typical fields which are available to us for now. The first bitfield

is typically called the Power Management Module LPM5 Interrupt Flag (PMMLP5IFG), and it is typically located in the Power Management Module Interrupt Flag register (PMMIFG). When the microcontroller is interrupted from LPMx.5, the PMM will set this bit as an indicator for that type of event.

Alternatively, the second bitfield is located in an interrupt vector register (IVR), so it's not a single bitfield; all the fields in the register are read to obtain a code. That register is typically called the System Reset Interrupt Vector register (SYSRSTIV), and the codes which it presents are published by the microcontroller's data sheet. This should be our primary register for getting information about which event had caused a reset. See page 209 and page 227 for more information about IVRs.

After the ISR is executed,

When an event interrupts the microcontroller from LPMx.5, the flow of execution is immediately transferred into the BOR, the first phase in the reset system, and then the flow has to once again go through the entire reset system, then re-enter main, then reconfigure the systems and modules, and then enter the LPMx.5 interrupt handler. And that is where the flow is transferred towards the ISR so it can be executed.


The details of this flow are described by the remaining sections of this chapter.

### Flow for the LPMx.5 Interruption

Shown by List 4 is a step-by-step explanation about the flow for the LPMx.5 interruption that was illustrated by diagram 89 on page 281. As a convenient starting point, step 1 begins at where the flow of execution emerges from the boot program. Boot was introduced on page 139. What caused the flow to enter boot does not matter, it could have been caused by a conventional system reset event, a power-up event, or an interruption from LPMx.5.

Keep in mind that List 4 does not step through the entire flow of execution. It only describes the steps which go through the `main()` function. If an interruption from LPMx.5 had caused the flow to enter `main()`, then the interrupt handler will transfer the flow to the interrupt service routine (ISR), which is located outside of `main()`. The ISR is described by List 5 on page 284, in the next section. The flow then returns back to `main()` at the place where it had been transferred out.

**List 4:** Flow for the LPMx.5 interruption.

1. We begin with the flow of execution emerging from the boot program.
2. 

3. Flow now enters the `main()` function.

4. The watchdog timer handler is entered (page 178). Do one of the following.

- 

5.

6.

7.

8.

9.

10. Flow enters the *LPMx.5 Interrupt Handler*.

- 

11. If the LPMx.5 ISR was executed, the

12.

13. Prepare the microcontroller for LPMx.5.

-

14. The microcontroller is now in a fractional low powered operating mode. This means the power is removed from most systems and modules which results in the following conditions:

- 

- 

- 

- 

15. The microcontroller is now in a specific fractional low powered operating mode (LPMx.5).

16.

---

### Flow for the LPMx.5 Interrupt Service Routine (ISR)

As illustrated by diagram 89 on page 281, this section is a continuation of the flow that had been transferred out of `main()` and into this ISR. The transfer was carried out by the LPMx.5 Interrupt Handler at step 10 on page 283.

The handler had determined that the microcontroller had emerged from a low powered fractional operating mode (LPMx.5), so it transferred the flow to an instruction that enabled maskable interruptions. Immediately after they were enabled, the request for interruption (IRQ) signal, which was caused by the flag, was recognized by the interrupt system. Here is the sequence of routines which lead up to the ISR and then executes it.

**List 5:** Flow for the LPMx.5 ISR:

1. Once the interrupt system accepts the request, it then executes five routines.

- 

- 

- 

- 

-

- [REDACTED]
2. The microcontroller is now back in the active operating mode (AM) with the program counter register containing the vector (an address) which points to the first instruction in the ISR.
  3. [REDACTED]
  4. The RETI instruction transfers the flow of program execution back into `main()`.

---

### Program Example

Fractional low powered mode (LPMx.5) is typically limited to microcontrollers which are built of Ferro-Electric RAM (FRAM). They are distinguished by the letters FR in their part number; for example, MSP430FR2433. Those letters indicate the series, which in this case, means the FRAM series.

Code Example 81, on page 289, shows a program that configures the microcontroller for use before putting it into LPM3.5, a mode of sleep from where the microcontroller will wait to be interrupted to carry out an interrupt service routine (ISR). That program should work on any FR series of MSP430 with no or very little change made to it. It follows the event-driven pattern of programming, and that pattern is modified to incorporate the pattern of development for using fractional low powered operating modes. The event-driven process is shown by diagram 38 on page 110, by diagram 42 on page 120, by diagram 43 on page 128, and by diagram 44 on page 132. That

pattern has been modified to use LPMx.5 as described by earlier sections in this chapter and illustrated by diagram 89 on page 281.

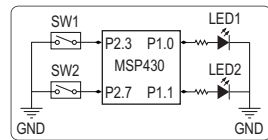
This program will execute in the LPM3.5 and LPM4.5 operating modes. It uses the `PERSISTENT()` `#pragma` for protecting (saving) data in a storage variable while in one of those modes. One characteristic of those modes is their extremely low amount of power consumption, while the other characteristic is the loss of volatile data held by storage variables.

To keep the program simple, it does not include

### Circuit Schematic for the Program

The circuit is very simple. It is built of an MSP430FR series microcontroller, two switches, two LEDs, a power supply, and wires for interconnecting them. To simplify the schematic, it does not include interface conditioning circuits, such as RC interface circuits for the switches and a decoupling interface circuit for conditioning the power supply; nor does the circuit show the power supply connections.

**Diagram 90:** Circuit schematic for the program example in this chapter.



As a convenience, this circuit is available as a low cost, prebuilt development kit called the MSP-EXP430FR2433. Therefore, that schematic takes into account the port channel numbers, the switch numbers, and LED numbers which are used in that kit. So Switch 1 (SW1) is connected to channel 3 of port 2 (P2.3), and SW2 is connected to P2.7. LED1 is connected with channel zero of port 1 (P1.0), and LED2 is connected to P1.1.

### How the Program Example Works

The program goes through a sequence of routines which configure the microcontroller for use and then puts it into LPM3.5, a deep level of sleep. When a switch is closed (this is the event), it provides a signal to the port channel. The channel

The ISR determines which channel had caused the interruption, and then transfers the follow of execution to the proper subroutine to carry out a specific set of instructions. If the interruption was caused by SW1, then the subroutine will increment a counter variable by 1, and then flash LED1 by the number of times equal to the number

stored in the variable. The development kit uses a green colored LED for LED1. If the interruption was caused by SW2, then the subroutine will clear the counter variable to zero and flash LED2. The kit uses a red colored LED for LED2. When the flow of execution finishes with the ISR, the microcontroller is automatically put back into LPM3.5. While the microcontroller is in LPM3.5, the contents of the counter variable are protected because the `PERSISTENT()` `#pragma` put it in non-volatile memory. So for example, if the counter contains the number 2, and SW1 is closed to produce a signal, then the ISR will increment the counter to the number 3 and then flash the green LED, at P1.0, three times. If SW2 is closed, then the ISR will clear the counter to zero and then flash the red LED once.

As a convenience, the ISR uses a `__delay_cycles()`. It is a built-in MSP430 intrinsic function that `__delay_cycles(1000)`.

`__delay_cycles()`. Its purpose is to mitigate false input signals caused by the bouncing mechanical switch contacts. It is a simple substitute for a proper switch interface circuit, so it should not be used in a commercial grade program to mitigate false signals caused by a bouncing switch.

---

### Structure of the Program Example

Functions are used for organizing the program into blocks of instructions which carry out a specific task. That technique will simplify the `main()` function so it will appear shorter and thus easier to comprehend. The functions are labeled with names which clearly state their purpose.

So preceding the `main()` function is a sequence of instructions which tell the MSP430 compiler to include the `msp430.h` header file, then declare the prototypes of each function we define after `main()`, and then a couple of instructions that use the `PERSISTENT()` `#pragma` to create our counter variable and store it in non-volatile memory.

The `main()` function contains a sequence of functions that will configure the microcontroller for use. The ports are unlocked, and then an LPMx.5 interrupt handler decides whether or not the microcontroller had emerged from reset because of an interruption or just from a conventional power-up event. The remaining instructions in `main()` prepare the microcontroller for LPMx.5 and then puts it to sleep.

Four functions and one ISR follow `main()`. The first one contains instructions which configure modules which are located outside of the LPMx.5 domain. The second one configures those which are inside the domain. The third one clears all interrupt flags which are enabled for use. And the fourth one contains instructions which prepare the microcontroller for a fractional low powered mode. The ISR just simply contains instructions which handle the interruption.

Be aware that the program is lean. Meaning, it does not contain all the instructions which are shown in the event-driven pattern of programming. This is intentional in order to focus on the basic structure of instructions for a program that will handle interruptions from LPMx.5. Furthermore, this is just one way to handle such an inter-



ruption. You may be able to design a better pattern, or one which better suits your needs.

---

### Program Example

The program is shown by code example 81. At line 1, it begins by including the standard header file (page 146). Then, as required by the C language, the function prototypes are listed before `main()`, while their definitions are placed after `main()`. So on line 2 is the prototype for configuring peripheral modules located outside of the LPMx.5 domain. Its definition begins on line 27. Since this program does not use any module outside of that domain, it only contains an MSP430 intrinsic function named `__no_operation()` that tells the CPU to just simply copy the contents in CPU Register 3 and write them back into the same register. This is just a place holder to show that this is where such configuration instructions would be located.

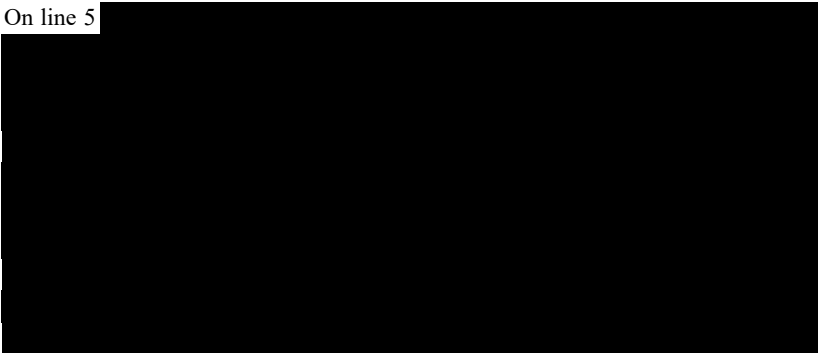
On line 3




Back to line 4 where



On line 5



Next, an instruction then turns off the PMM voltage regulator, which is what really reduces the power consumption in fractional mode. And then the PMM registers are relocked by writing an incorrect password into the upper eight bits of the `PMMCTL0` register. Those instructions were introduced earlier on page 222.

On line 6 is the  that we use for telling the MSP430 compiler to place our variable named `counter` into non-volatile memory, and then on the fol-

lowing line that variable is declared and initialized to zero. The usage of that #pragma was introduced on page 217.

**Code Example 81:** Program that uses an interruption from LPM3.5.

```

1 // base header file
2 // function prototype
3 // function prototype
4 // function prototype
5 // function prototype
6 // Make counter non-volatile, and then
7 // declare and initialize the variable.
8
9 // Put watchdog timer on hold.
10
11 // LPMx.5 Interrupt Handler.
12 // Unlock systems in LPMx.5 domain.
13 // LPMx.5 Interrupt Handler.
14 // If interrupted from LPMx.5,
15 // enable maskable interruptions.
16 // End if() block.
17 } // If not interrupted from LPMx.5,
18 // clear all enabled flags, and then
19 // enable maskable interruptions.
20 // End else block.
21 } // Prepare for fractional mode.
22 // Put into Low Powered Mode 3.
23 // Never reached.
24 // End main().
25 }
26
27 // Since there are no items, do nothing.
28 // End of function definition.
29 }
30
31 // Clear to set signal direction as inwards.
32 // Set signal transition from high to low.
33 // Set to enable resistors to pull high.
34 // Set to put output channel to logical high.
35 // Clear P1.0 & P1.1 output to darken LEDs.
36 // Set P1.0 & P1.1 direction outwards to LEDs.
37 // Set to enable interrupts by these flags.
38 // End of function definition.
39 }
40
41 // Clear P2.3 and P2.7 flags.
42 // End of function definition.
43 }
44
45 // Unlock PMM registers w. regard to SVSHE.
46 // Set to turn off PMM voltage regulator.
47 // Lock the PMM registers.
48 // End of function definition.
49 }
50
51 // End of function definition.
52 }
53 }
54 }

```

```

55 /***** This is our ISR *****/
56
57 // Bind this vector to the ISR
58 // ISR signature
59 // wait for switch to stop bouncing-150 ms
60 // Counter for while() loops
61
62 // Begin switch() statement
63 // If P2.3 IFG is set, then
64 // unlock Program FRAM,
65 // increment counter variable,
66 // then lock Data and Program FRAM
67 // while i < counter,
68 // Set to light P1.1 green LED,
69 // delay execution for .5 second,
70 // delay for .5 second, and then
71 // increment while() loop counter.
72 // End while() loop
73 // End of case. Exit switch().
74 // If P2.7 IFG is set, then
75 // unlock FRAM,
76 // clear counter,
77 // lock FRAM, and then
78 // Set to illuminate red LED
79 // delay execution for 2 seconds,
80 // delay execution for 2 seconds,
81 // End case. Exit switch().
82 // End of switch() statement
83 // End ISR
84 }
85 }

```

On line 9 begins the definition for our `main()` function. Then on line 10 we turn off the watchdog timer, but if this were going to be a commercial grade program, a proper watchdog would be configured, but turned off before the microcontroller is prepared for LPMx.5 on line 22. The watchdog timer was introduced on page 89, and then starting at page 178, it is elaborated upon. On lines 11 and 12 are the calls to the functions which configure the peripheral modules. And on line 13 we unlock the port channels and other functions which are in the LPMx.5 domain, otherwise, they will not be available for use.

Beginning on line 14 and ending on line 21 is our LPMx.5 interrupt handler. It decides on whether or not the flow of execution had emerged from a conventional power-up event or a reset caused by an interruption from LPMx.5. So on line 14, the System Reset Interrupt Vector register (SYSRSTIV) is read, and if what is read equals to the `SYSRSTIV_LPM5WU` constant, then the flow had emerged from an interruption from LPMx.5. The constant is defined by

, it is transferred to line 16 where maskable interruptions are enabled, and then immediately it is transferred to the first instruction in our ISR, that begins on line 58. When finished with the ISR, the flow is automatically transferred to line 22, back in `main()`, where the microcontroller is then prepared for LPMx.5. If the flow had not

emerged by an interruption from LPMx.5, it gets transferred to line 19 where the alternative path of execution is taken. All flags are cleared, and then interrupts are enabled.

The `main()` function ends with preparing the microcontroller for LPMx.5 (line 22), then it is placed into the low powered mode from where it waits to be interrupted. These instructions were explained on page 222. When interrupted, the flow of execution is transferred to the BOR, the first phase of the system reset, and then continues until it once again enters `main()`. The flow should never reach the `return` instruction on line 24.

The ISR begins on line 57.



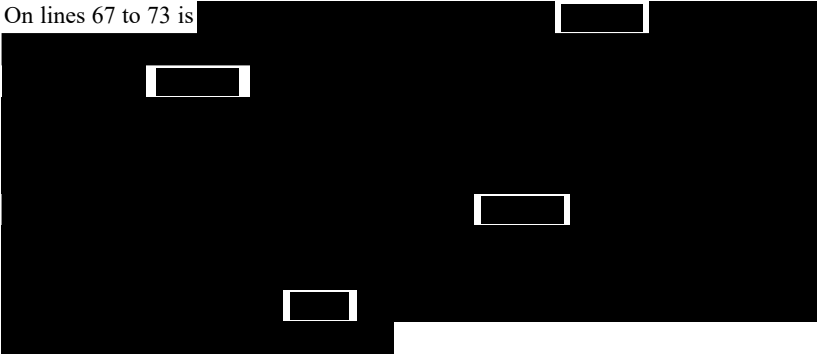
Since the interrupt vector is bound to every channel in port 2, the ISR must now distinguish which channel had set the flag and then transfer the flow of execution to the proper subroutine to handle that particular flag. To make that decision and the proper transfer, we use a `switch()` statement. The use of that statement was introduced on page 260, and then elaborated upon with a detailed explanation starting on page 274; therefore all those details will not be repeated here.

So in this scenario, our `switch()` needs two cases: one to handle the input signals produced by SW1 at P2.3, and one for SW2 input signals at P2.7. We could have just only used the register variable for the Port 2 Interrupt Vector register (P2IV) as the parameter for the `switch()` to make a decision, but instead we use the `__even_in_range()` function. Therefore, the parameters are P2IV and `0x10`. The first parameter, P2IV, is the register variable for the Port 2 Interrupt Vector register, and that means to read the contents of that register. The second parameter, `0x10`, means that number is the last code number in range of codes which that register can provide. The function will then optimize the `switch()` code to just use two codes for making the decision: one for the flag status at P2.3 and one for P2.7. The range is shown by the Port 2 Interrupt Vector register (P2IV) table in the microcontroller's user guide, as shown by diagram 87 on page 276.

Let's go to the first case of the `switch()`. The flow is transferred here if the code in the interrupt vector register is equal to `P2IV_P2IFG3`. That's the code number for the P2.3 flag, and it's defined in the header file. The first three instructions, beginning on line 64, unlocks the FRAM, then increments our counter variable (stored in FRAM),

and then relocks the FRAM. This sequence of instructions is explained by the code example on page 214.

On lines 67 to 73 is



Now for the second case in the `switch()`, it's used for flashing the red LED just once to visually tell us that the counter variable has been cleared to zero. So in order to clear the variable, we first have to unlock the FRAM (line 76) since it is in FRAM, then clear the variable (line 77), and then relock the FRAM (line 78). Then on lines 79 to 81 the red LED is flashed once. On line 82 the flow is transferred to the end of the ISR.

On line 83 is another MSP430 intrinsic function, the `__never_executed()` function. It is used as the default case, and it is specially designed for handling codes produced by a vector generator. It tells the compiler that our `switch()` can only take on values represented by the case labels within our `switch()` block. It provides information to the compiler when testing our program. In other words, the compiler will avoid generating test code for handling values which are not specified by the `switch()` case labels.

---

## Symbols

#define 145  
 #include 144, 146  
 #pragma Directives 147  
 #pragma, vector 245  
 \_\_delay\_cycles() 225  
 \_\_even\_in\_range() function 208  
 \_\_interrupt 246  
 \_\_never\_executed() 244, 292  
 \_\_no\_operation() 288  
 \_\_system\_post\_cinit() 96, 142, 147  
 \_\_system\_pre\_init() 96, 141, 147

## A

active mode 133  
 ADC  
   differential 22  
   module 21  
   single-ended 22  
 ADC calibration transfer function 171  
 ADCENC 167  
 ADCON 167  
 ADCSC 172  
 address space  
   as a region of storage 75  
   definition of 7  
   sizes 18  
 analog  
   ground 19  
   power supply 19  
 arithmetic logic unit 1  
 AVCC 19  
 AVSS 19

## B

backup memory registers, using 220  
 BAKMEM 217  
 BAKMEM Module 30  
 bandgap 165  
 basic view of a microcontroller 12  
 binary

  number notation 6  
   numbers 4

### bit

  clearing a 38  
   clearing, meaning of 61  
   definition of a 5  
   setting a 38  
   setting, meaning of 61  
   toggling, meaning of 61

### bitfield

  accessibility 47  
   description, simple 7, 35  
   initial condition 47  
   interrupt enable 49  
   mask suffix 46  
   mask, detailed description of a 46  
   mask, simple description of a 38  
   masking concepts 62

  ...continuation of bitfield

  register table, in a 45

  bitfield accessibility and the initial condition 187

  bitfield mask 62

### bits

  clearing 67  
   setting 65  
   simultaneously setting and clearing 69  
   toggling 70

### bitwise

  manipulation, simple explanation 46  
   operation 62

  block of instructions 103

  boot hook function

    post initialization 147

    pre-initialization 147

  boot hook functions 140

  boot initialization 175

  boot program 139, 140

    location of 139

    memory protection unit 141

    post initialization function 142

    pre-initialization function 141

    program execution stack 141

### BOR

  description 135

  events that produce a 135

  operating mode diagram, within 132

  BOR signal 40

  brownout 131

  brownout circuit 131

  brownout reset 135

  buffers, port channel 190

  Built 247

  byte 5

  byte mode 178

  byte mode access 220

## C

C/C++ software stack 141

  calibration data 19, 36

  capacitors, for the VCC pin 41, 129

### CCS

  compiler optimization for a file 60

  displayed numbering format 60

  project compiler optimization 60

  stepping through each instruction 59

  channel function multiplexor 194

  channel, port 23

  char data type width 45

  clearing bits 67

  clock cycles for an instruction 93

  clock frequency 20

  clock system

    module description 21

    password violation 137

  clock system registers, unlocking the 198

  clock, profile 94

  Code Composer Studio. *See* CCS

  Common Control Block 194

- compiler
  - MSP430 relaxed 79
  - strict C 79
- conditioning circuit 15, 16, 111
  - ADC example 22
- constant
  - generator registers 8
  - numerical value 75
  - symbolic 62
- control logic 1
- conventional C functions 146
- conventional register 235
- core voltage 134
- CPU
  - crash and watchdog timer overflow 137
  - definition of a 1
  - description 6
  - handling an ISR (internal event) 15
  - handling and ISR (external event) 17
  - in the reference model 110
  - interruption in larger context 142
  - memory structure 36
  - module, as a 20
  - supply voltage, effects from 20
- CPUX compared to CPU 6
- CRC16 module 24
- cyclic redundancy check (CRC) 24

## D

- data
  - how developers view it 6
  - how the microcontroller views it 5
  - processing, meaning of 1
  - storage 75
- data types
  - char 45
  - integer 45
  - long integer 45
- DCO 154
- debugging mode 20
- decimal notation 6
- decision and output signal sequence 125
- delay function 225
- delay handler 154
- device descriptors 171
- digital
  - ground 19
  - power supply 19
- digital I/O module
  - description, simple 50
  - port, as compared to a 50
- digital I/O pins
  - PUC affects 138
  - unlocking 123
- digital I/O port channels, unlocking the 201
- digitally controlled oscillator 154
- do...while() statement 106
- DVCC 19, 129
- DVSS 19, 129

## E

- EEM 20
- electronic fuse 20
- enable maskable interruptions 124
- environment 75

- eUSCI Module 28
- event
  - definition of an 110
  - externally occurring 111
  - internally occurring 112
  - start 130
- event monitoring blocks 227
- event-driven pattern
  - details 120
  - in a larger context 127
- execution, flow of 105
- Expressions window 216
- external oscillating device 21

## F

- fetch
  - and execute cycle 2, 15, 17
  - description 137
  - from peripheral area 137
- firmware
  - defined 4
  - image 35, 143
- flag signal buffer 190
- flow of execution 105
- for() statement 106
- fractional low power operating mode 132
- fractional lower powered modes 223
- FRAM 7, 18, 137, 217
  - access control 213
  - controller 214
  - data 214
  - information 214
  - memory password violation 137
  - program 214
  - unlocking and locking 213
- function
  - C language 99
  - frame stack 141
- functional block diagram 17

## G

- GCC
  - description 79
  - extensions, enabling 80
- general interrupt enable. *See* GIE
- general purpose input or output. *See* GPIO
- GIE 15, 93
- global
  - storage variables 145
  - variables, initializing 141
- GPIO 22, 187
  - ground
    - analog 19
    - digital 19

## H

- header file
  - base 146
  - C language 144
  - MSP430.h 45
- hexadecimal number notation 6

**I**

- I/O Port Modules 22
- I2C 28
- IE 49
- if() selection statement 106
- if...else selection statement 106
- IFG 14, 49, 245
  - flow of execution 203
  - its state through LPMx.5 190
- image. *See* firmware image
- immutable 75
- indirection operator 83
- information memory 36
- initial condition table 48
- initialization, definition of 134
- initializing
  - global variables 141
  - operating mode diagram, as shown by 131
- input
  - signal 143
  - signal sequence 125
  - signal stack 110
- intrinsics.h 146
- instruction clock cycles 93
- instruction, definition of an 1
- integer
  - constant 79
  - data type width 45
- interrupt
  - compare controller 14
  - compare controller (ICC) 233
  - control logic 229
  - enable bit, general 15
  - flag bit 14
  - flag handler, port channel 202
  - maskable 15, 92
  - non-maskable. *See* NMI
  - port, at a 23
  - prioritization 232
  - requests 9
  - See* interrupt service routine
  - system 14
  - system bitfields 49
  - system reference model 111
  - vector 203, 228, 245
  - vector generator register (IVR) 207
  - vector register 227, 235
  - vector table 14, 36
- interrupt request signal 125
- interrupt service routine 9, 49
  - customized default 248
  - defining 147
  - final instruction 15
  - purpose 111, 112
  - triggered by an external event 17
  - triggered by an internal event 15
  - within the reference model 109
- interrupt system 125
- interruption
  - determining the source of an 203
  - how it is processed 230
  - non-reset 203
  - reset 203, 237
- interruptions
  - complete list of 134
  - enable maskable 124
- intrinsic functions 133

- IrDA 29
- IRQ 9, 125
- IRQA 184
- ISR
  - LPMx.5 interruption, for the 284
  - purpose of an 9
  - See* interrupt service routine
  - syntax 245
  - syntax for the conventional 245
  - syntax for the custom default 248
  - vector 14
- IVR 235

**J**

- Joint Test Action Group (JTAG) 20
- JTAG 20, 130

**L**

- LDO 130
- LFXT 21
- locking the microcontroller 20
- LOCKLPM5 202
- long integer data type width 45
- loop, structure 106
- low power operating mode 125, 132, 133
- low powered operating mode, entering a 222
- LPM. *See* low power operating mode
- LPM3.5 domain 29
- LPMx.5 132, 223
  - bitfields for determining the interruption 281
  - domain 279
  - flow of execution diagram 281
  - Interrupt Handler 283
  - interrupt handler 281
  - interrupting from 279
  - types of interruptions 279
  - volatile data handler 280

**M**

- MAB 19
- machine registers 8
- macro 88, 145
- main memory
  - address space 35
  - map 36
  - memory modules 18
  - purpose 35
  - registers in 35
  - sizes 6
  - structure 35
- main() function 99, 142
- maskable interrupt 15, 124, 133, 229
  - enabled with PUC 138
  - enabling and disabling 212
- masking concepts, bitfield 62
- MDB 19
- memory
  - access, direct 61
  - access, symbolic 61
  - buses 19
  - consumed, determining how much is 221
  - map 36
  - modules 18
  - non-volatile 213



- ...*continuation of memory*
  - See also* main memory
  - segment violation 137
  - volatile 213
  - Memory Browser window 216
  - Memory Protection Unit 141, 214
    - registers, unlocking the 198
    - violation of 137
  - microcomputer, definition of 2
  - microcontroller
    - history 2
    - series 285
    - when the name appeared 3
  - microprocessor, definition of 2
  - MODCLK 166
  - model register 61
  - MODOSC 173
  - module
    - ADC 21
    - BAKMEM 30
    - clock system 21
    - CPU, EEM, JTAG, and SBW 20
    - CRC16 24
    - embedded emulation 20
    - eUSCI 28
    - I/O Port 22
    - memory 18
    - MPY32 22
    - peripheral 2
    - power management 19, 134
    - RTC counter 30
    - SYS 24
    - system 2
    - timer 25
    - types of 2
  - MPU. *See* Memory Protection Unit
  - MPY32 Module 22
  - MSP, definition of 3
  - MSP430
    - C preprocessor 143
    - compiler 143
    - header file 45
    - input/output power 1
    - purpose and usage 1, 3
    - reference model 109
    - when it began to be marketed 3
    - why use the MSP430 3
  - msp430.h 146
  - MSP-BSL 139
  - multiplexer, port 23
  - mutable 75
- N**
- native word size 5, 19
  - Negative Logic GPIO Input Function 191
  - nibble 5
  - NMI 15, 20, 92, 124, 133, 229, 249
  - noise at the VSS and VCC pins 129
  - non-maskable interruption. *See* NMI
  - non-volatile program code 36
  - numbering format, CCS displayed 60
- O**
- object 75
  - operand 75
  - operating mode 222
    - diagram 131
    - how created 14
    - low power 125
  - operating modes, table of 268, 280
  - operator 75
  - oscillator
    - for the clock system 21
    - low frequency 21
    - settling handler 122
  - oscillator settling handler 182
  - output buffer 194
  - output signal 143
    - defined 112
    - sequence 125
    - stack 112
    - the proper way to produce an 112
  - overflow 26
    - in the RTC module 30
    - timer 26
- P**
- password protected registers 176
  - pattern
    - definition of a 115
    - ISR-based 128
    - NMI-based 128
  - patterns for program development
    - event-driven 120
    - repetitive-driven 115
  - peripheral device
    - active 111
    - definition of 2
    - passive 111
  - peripheral module 2
  - PERSISTENT() #pragma 217
  - physical interfacing 129
  - pins, unused 130
  - plan, development 107
  - PM5CTL register 123
  - PM5CTL0 202
  - PMM 19
    - password 165
    - register table 177
    - registers, unlocking the 197
  - pointer 83
  - pointer variable 83
  - POR 136
    - events which cause a 136
    - operating mode diagram, within 132
    - signal 41
  - port 22, 51
    - bitfield masks, register 52
    - channel 23
    - channel, configuring a 187
    - digital I/O module, as compared to a 50
    - Input/Output Diagram 187
    - Input/Output diagram description 194
    - multiplexer 23
    - pins 52
    - register table, first type of port 52
    - register table, second type of port 54
    - register table, third type of port 56
    - register tables 52
  - Positive Logic GPIO Input Function 192
  - post 142

- power management module 19, 134
- power management module violation 137
- power supply
  - analog 19
  - stack 113
- power-up 39, 129, 131, 132
- preprocessing translation unit 143
- process 89
- profile clock 94
- program
  - counter 230
  - development tool 35
  - execution 106
  - execution environment 35
  - execution stack 141, 230
  - storage location 7
    - within the reference model 109
- programmer-debugger 21
- programming symbol 38
- protected registers, accessing 197
- protocol 28
- PUC 137
  - events which cause a 137
  - operating mode diagram, within 132
  - signal 41
- pulse width modulated voltage signal 27
- PWM 27
- PxIFG 190
- PxIN 188
- PxOUT 189
- PxREN 189

## R

- RAM 7, 18, 36, 217
- random access memory. *See* RAM
- read only memory. *See* ROM
- reading a register 77
- real-time clock 30
- reference model 109
- reference oscillator 153
- REFO 153
- register
  - accessibility & initial conditions ex. 91
  - bitfield descriptions 48
  - bitfields 45
  - CPU registers 8
  - definition of 6
  - location of 7
  - main memory registers 7
  - model 61
  - purpose of main memory registers 8, 43
  - reading a 77
  - re-initialization of 131
  - See also* register variable
  - testing the contents of a 81
  - writing into a 61
- register table
  - basic introduction 37
  - behavioral description 38
  - bitfields 45
  - conventional register table 43
  - conventional type of table 38
  - data description 37
  - detailed explanation 43
  - functional view, used with a 49
  - port register tables 52

- ...*continuation of register table*
- types of 38
- unconventional type of table 38
- watchdog, for the 90
- register variable
  - description, detailed 44
  - description, short 23
  - reading and writing into, when 45
- remotely updating the microcontroller 21
- repetition structure 106
- reset 39
- reset (RST/NMI) pin 138
- reset fault handler 123, 211
- reset interruption 229, 237
- reset system 39
  - BOR signal 40
  - BOR, POR, and PUC 134
  - initialization signal 134
  - POR signal 41
  - power-up 39
  - PUC signal 41
  - purpose 131, 134
  - reset 39
  - reset signal 134
  - subsystems 134
- RESET\_VECTOR 203
- resistor, pull-up/pull-down 189
- RETI 232
- return statement 100
- ROM 7, 36, 217
- routine, definition of a 103
- RST 20
- RST/NMI Pin 92
- RST/NMI pin 230
- RTC
  - counter module 30
  - module 21
- run-time stack 141

## S

- SAMPCON 166
- SAR 172
- SBW 20
- Schmitt Trigger 51, 158
- selection structure 106
- sequence
  - decision and output signal 125
  - input signal 125
  - structure 105
    - system configuration and setup 121
- series, microcontroller 285
- service, definition of a 103
- servicing
  - a module 103
  - an ISR 103
- setting bits 65
- SHI 167
- shutdown mode 132
- SNMI 15, 230, 249
- software, definition of 4
- SPI 28
- Spy-Bi-Wire 20
- SRAM 18
- stack
  - input signal 110
  - output signal 112

- ...continuation of stack
- power supply 113
- program execution 141
- standard bits 47
- standby mode 132
- start event 130
- status register 133, 138, 230
  - operating modes 29
  - purpose 9
- storage
  - data 75
  - location 75
  - variable 75, 124
- structure
  - flow lines 106
  - repetition control 106
  - selection control 106
  - sequence control 105
- subroutine, definition of a 103
- supply voltage
  - effect on CPU 20
  - requirements 129
  - supervision 130
  - supervisors 19
- SVSH 130
- SW BOR 133
- SW PUC 133
- switch() selection statement 106
- symbolic constant 62, 145
- SYS Module 24
- system
  - configuration and setup sequence 121
  - module 2
  - module, in the reference model 110
  - NMI 15
  - post-initialization 95
  - pre-initialization 95
- System Control Module 214

## T

- terminal functions table 32
- timer
  - as a frequency dependent timer 26
  - as a frequency independent counter 26
  - as a module for measuring rates 26
  - as described by functional a diagram 28
  - for producing PWM signals 27
  - module overview 25
  - overflow definition 14
  - overflow example 26
- Timer A0 module 193
- translation unit
  - conventional C 144
  - directives 143
  - intrinsic 143
  - MSP430 units 146

## U

- UART 28
- uncorrectable FRAM bit error detection 137
- unlock the digital I/O pins 123
- UNMI 15, 20, 230, 249
- unused port channels, configuring 198
- use case, definition of 109
- user NMI 15

## V

- value, constant numerical 75
- variable
  - declaring a storage 76
  - immutable 75
  - mutable 75
  - storage 75
- Vcore 133
- vector 14
- vector #pragma 245
- vector generator 207
- view
  - basic view of a microcontroller 12
  - definition of 11
  - device pinout 31
  - event, view of externally occurring 16
  - event, view of internally occurring 13
  - event-driven pattern, of the 120
  - functional block diagram 17
  - module functional 32
  - pin designation 31
  - startup 12
- viewing register bitfields during operation 190
- volatile
  - data 121
  - data handler 124
  - memory 4, 29, 30
  - program data 36
- voltage
  - level effects 129
  - regulators 130
  - supply requirements 129
  - supply supervision 130
  - supply supervisors 19
- VSVSH- 130
- VSVSH+ 130

## W

- watch crystal 21, 116, 122
- watchdog timer 89, 136, 137, 138, 197
  - handler 121
  - password violation 137
  - reading its registers 96
  - stopping during boot 95
  - WDTHOLD bitfield 94
  - WDTPW bitfield 91
- while() statement 106
- word mode access 220
- word, definition of a 5

## X

- XIN 21
- XOUT 21

# The MSP430 Microcontroller Engineering Guide: Getting Started

**For anyone who wants to easily and quickly develop a basic knowledge about how to develop programs for the MSP430 microcontroller.**

This book provides an insight to this remarkably sophisticated, powerful, yet easy to use computing system on a chip (SoC) which non-engineers and engineers can understand. It presents the basic concepts and fundamental techniques needed for developing programs which the MSP430 can use for monitoring sensors, controlling actuators, driving displays, and sharing information with remote destinations.

## Contents

Preface

1. Introduction

2. Visualizing How the MSP430 Operates

3. Visualizing the Main Memory

4. The Reset System and its Subsystems

5. How to read and use the Register Tables

6. Code Composer Studio Usage Tips

7. How to Write into a Register

8. How to Declare a Storage Variable

9. How to Read a Register

10. Background for Testing the Contents of a Register

11. How to Test the Contents of a Register

12. How to use a Pointer to Read and Write into Main Memory

13. Watchdog Timer and Putting it on Hold

14. main() Function

15. Program Development Nomenclature

16. Structures for Program Development

17. Basic Approach for Developing a Microcontroller Solution

18. MSP430 Reference Model

19. Patterns for Program Development

20. Placing the Event-Driven Pattern into a Larger Context

21. Repetitive-Driven Programming Examples

22. Event-Driven Programming Routines and Practices

23. Interrupt Handling and Interrupt Vectors

24. How to Determine which are the Multi-Flagged Vectors

25. The Reset Interruption

26. How to Write an Interrupt Service Routine (ISR)

27. Non-Maskable Interruption (NMI)

28. Maskable Interruption

29. Interruption from Fractional Low Powered Mode (LPMx.5)

Index

This series of books form a library written for anyone who wants to easily and quickly learn about developing programs and connecting devices to what might be the very best 16-bit microcontroller.

\$134.30

ISBN 978-0-9985736-0-1

Written, Illustrated, Edited, and Printed in the U.S.A.



1 3 4 3 0 >



9 780998 573601

Internetpress®

<http://internetpress.com>

310 Pages