

# TI ARM Peripherals Programming and Interfacing

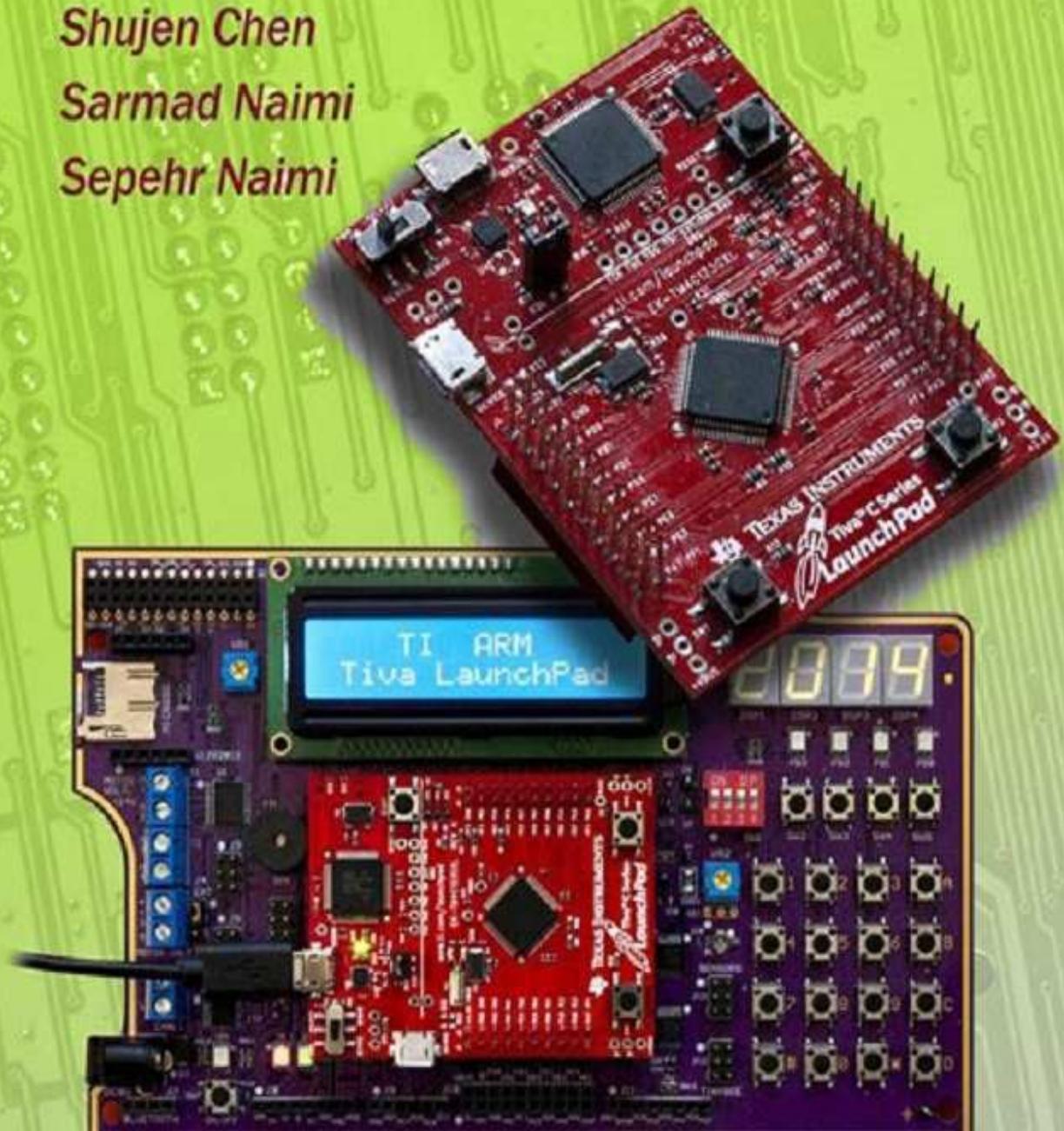
## Using C Language

*Muhammad Ali Mazidi*

*Shujen Chen*

*Sarmad Naimi*

*Sepehr Naimi*



# TI ARM Peripherals Programming and Interfacing

## Using C Language for ARM Cortex

Muhammad Ali Mazidi

Shujen Chen

Sarmad Naimi

Sepehr Naimi



**Copyright © 2014 Mazidi and Naimi**

**All rights reserved**

"Regard man as a mine rich in gems of inestimable value. Education can, alone, cause it to reveal its treasures, and enable mankind to benefit therefrom."

Baha'u'llah

## **Dedication**

*To the faculty, staff, and students of BIHE university for their dedication and steadfastness.*

# Preface

Since the early 2000s, hundreds of companies have licensed the ARM CPU and the number of licensees is growing very rapidly. While the licensee must follow the ARM CPU architecture and instruction set, they are free to implement peripherals such as I/O ports, ADCs, Timers, DACs, SPIs, I2Cs and UARTs as they please. In other words, while one can write an Assembly language program for the ARM chip, and it will run on any ARM chip, a program written for the I/O ports of an ARM chip for company A will not run on an ARM chip from company B. This is due to the fact that special function registers and their physical address locations to access the I/O ports are not standardized and every licensee implements it differently. We have dedicated the first volume in this series to the ARM Assembly language programming and architecture since the Assembly language is standard and runs on any ARM chip regardless of who makes them. Our ARM Assembly book is called "*ARM Assembly Language Programming and Architecture*" and is available from Amazon in Kindle format. See the following link:

[http://www.amazon.com/Assembly-Language-Programming-Architecture-ebook/dp/B00ENJPNTW/ref=sr\\_1\\_1](http://www.amazon.com/Assembly-Language-Programming-Architecture-ebook/dp/B00ENJPNTW/ref=sr_1_1)

For the peripheral programming of the ARM, we had no choice but to dedicate a separate volume to each vendor. This volume covers the peripheral programming of the TI (Texas Instruments) ARM Tiva chip. Throughout the book, we use C language to access the special function registers and program the TI ARM Tiva peripherals. We have provided a couple of Assembly language programs for I/O ports in Chapter 2 for those who want to experiment with Assembly language in accessing the I/O ports and their special function registers. These few Assembly language programs also help to see the contrast between the C and Assembly versions of the same program in ARM.

## Two approaches in programming the ARM chips

When you program an ARM chip, you have two choices:

1. Use the functions written by the vendor to access the peripherals. The vast majority of the vendors/companies making the ARM chip provide a proprietary device library of functions allowing access to their peripherals. These device library functions are copyrighted and cannot be used with another vendor's ARM chip. For students and developers, the problem with this approach is you have no control over the functions and it is very hard to customize them for your project.
2. The second approach is to access the peripheral's special function registers directly using C language and create your own custom library since you have total control over each function. Much of these functions can be modified and used with another vendor if you decide to change the ARM chip vendor. In this book, we have taken the second approach since our primary goal is to teach how to program the peripherals of an ARM

chip. We know this approach is difficult and tedious, but the rewards are great.

## Compilers and IDE Tools

For programming the ARM chip, you can use any of the widely available compilers from Keil ([www.keil.com](http://www.keil.com)), IAR ([www.IAR.COM](http://www.IAR.COM)) or any other one. Some vendors also provide their own compiler IDE for their ARM chips. TI provides [Code Composer Studio™ \(CCStudio\)](#) free of charge. For this book, we have used the Keil ARM compiler IDE to write and test the programs. They do work with other compilers including the TI CCStudio.

## TI ARM Trainer

The TI has many inexpensive trainers for the ARM Tiva C series. Among them are TI Launchpad Evaluation kits. Although we used the TI Launchpad Evaluation Kit to test the programs, the programs runs on other TI kits as long as they are based on Tiva™ C Series ARM® Cortex™-M4F-based microcontrollers series.

## Chapters Overview

In Chapter 1, we examine the C language data types for 32-bit systems. We also explore the new ISO C99 data types since they are widely used in IDE compilers for the embedded systems.

Chapter 2 examines the simple I/O port programming and shows sample programs on how to access the special function registers associated with the general purpose I/O (GPIO) ports.

Chapter 3 shows the interfacing of the ARM chip with the real-world devices: LCD and keypad. It provides sample programs for the devices.

In Chapter 4, the interfacing and programming of serial UART ports are examined.

Chapter 5 is dedicated to the timers in ARM. It also shows how to use timers as an event counter.

The Interrupt programming of the ARM is discussed in Chapter 6.

Chapter 7 examines the ADC concepts and shows how to program them with the ARM chip. It also examines the sensor interfacing and signal conditioning.

Chapter 8 covers the SPI protocol and shows DAC interfacing with sample programs in ARM.

The I2C bus protocol and interfacing of an I2C based RTC is discussed in Chapter 9.

Chapter 10 explores the relay and stepper motor interfacing with ARM.

The DC motor and PWM is examined in Chapter 11.

The Graphics LCD concepts and programming are discussed in Chapter 12.

Many high-end of ARM motherboards use DRAM memory. In Chapter 13, we examine the basic concepts of the DRAM memory chips.

The Cache memory concepts and organizations are discussed in Chapter 14. Although many low-end of ARM microcontrollers do not have on-chip cache, all the high-performance ARM chips come with on-chip cache.

The Virtual memory and memory management unit (MMU) features are available in the ARM R series. We explore the MMU of ARM in Chapter 15. Chapter 15 also covers the memory protection and MPU (memory protection unit) of ARM.

Appendix A provides an introduction to IC chip technology and IC interfacing along with the system design issues and failure analysis using MTBF.

## Online support for this book

All the programs in this book are available on our website:

[http://www.microdigitaled.com/ARM/ARM\\_books.htm](http://www.microdigitaled.com/ARM/ARM_books.htm)

Many of the interfacing programs such as LCD can be tested using the TI ARM Tiva Launchpad evaluation connected to an LCD on a breadboard. However, many courses use a system approach to the embedded course by using an ARM trainer. For this reason, we have modified the programs for the EduBase board using TI ARM Tiva board. See the following for the sample programs:

[http://www.microdigitaled.com/ARM/ARM\\_books.htm](http://www.microdigitaled.com/ARM/ARM_books.htm)

## Where to buy TI ARM Tiva Launchpad Evaluation kit?

See the link below for TI ARM Launchpad evaluation kit and TM4C123GXL datasheet.

<http://www.ti.com/tool/ek-tm4c123gxl>

The above Tiva Launchpad kit uses the TM4C123GH6PM chip. The TM4C123GH6PM chip is part of the ARM Cortex M4 C series from Texas Instruments' and is often called Tiva™ C Series. They are under Tiva C Series TM4C123G ARM® Cortex™-M4-based microcontroller products on TI website.

## Where to buy EduBase board?

See the link below for purchasing the EduBase board:

[http://www.microdigitaled.com/ARM/ARM\\_books.htm](http://www.microdigitaled.com/ARM/ARM_books.htm)

## End of the chapter problems

We are in the process of producing end-of-chapter problems for university courses. Please contact the authors if use this book for a university course.

[mdebooks@yahoo.com](mailto:mdebooks@yahoo.com)

# Table of Contents

## Chapter 1: C for Embedded Systems

Section 1.1: C Data types for Embedded systems

Section 1.2: Bit-wise Operations in C

Answer to Review Questions

## Chapter 2: TI ARM I/O Programming

Section 2.1: Texas Instruments Tiva TM4C123GH6PM Microcontroller

Section 2.2. GPIO (General Purpose I/O) Programming and Interfacing

Section 2.3: Seven-segment LED interfacing and programming

Answer to Review Questions

## Chapter 3: LCD and Keyboard Interfacing

Section 3.1: Interfacing to an LCD

Section 3.2: Interfacing the Keyboard to the CPU

Answers to Review Questions

## Chapter 4: UART Serial Port Programming

Section 4.1: Basics of Serial Communication

Section 4.2: Programming UART Ports

Answer to Review Questions

## Chapter 5: TI ARM Timer Programming

Section 5.0: Introduction to counters and timers

Section 5.1: System Tick Timer

Section 5.2: Generating delays using TI timers

Section 5.3: Pin Selection for Timers

Section 5.4: Using Timer for input edge-time capturing

Section 5.5: Using Timer As a Counter

Section 5.6: 32/64-bit Timer Programming (Case Study)

Answers to Review Questions

## Chapter 6: Interrupt and Exception Programming

Section 6.1: Interrupts and Exceptions in ARM Cortex-M

Section 6.2: ARM Cortex-M Processor Modes

Section 6.3: I/O Port Interrupt Programming

[Section 6.4: UART Serial Port Interrupt Programming](#)

[Section 6.5: Timer Interrupt Programming](#)

[Section 6.6: SysTick Programming and Interrupt](#)

[Section 6.7: Interrupt Priority Programming in TI ARM](#)

[Answer to Review Questions](#)

[Chapter 7: ADC, DAC, and Sensor Interfacing](#)

[Section 7.1: ADC Characteristics](#)

[Section 7.2: ADC Programming with the Tiva TM4C123G](#)

[Section 7.3: Sensor Interfacing and Signal Conditioning](#)

[Section 7.4: Interfacing to a DAC](#)

[Answers to Review Questions](#)

[Chapter 8: SPI Protocol and DAC Interfacing](#)

[Section 8.1: SPI Bus Protocol](#)

[Section 8.2: SPI programming in TI ARM Tiva](#)

[Section 8.3: LTC1661 SPI DAC](#)

[Answers to Review Questions](#)

[Chapter 9: I2C Protocol and RTC Interfacing](#)

[Section 9.1: I2C Bus Protocol](#)

[Section 9.2: I2C Programming in TI ARM Tiva](#)

[Section 9.3: DS1307 RTC Interfacing and Programming](#)

[Answers to Review Questions](#)

[Chapter 10: Relay, Optoisolator, and Stepper Motor Interfacing](#)

[Section 10.1: Relays and Optoisolators](#)

[Section 10.2: Stepper Motor Interfacing](#)

[Answers to Review Questions](#)

[Chapter 11: PWM and DC Motor Control](#)

[Section 11.1: DC Motor Interfacing and PWM](#)

[Section 11.2: Programming PWM in TI Tiva LaunchPad](#)

[Answers to Review Questions](#)

[Chapter 12: Programming Graphics LCD](#)

[Section 12.1: Graphics LCDs](#)

[Section 12.2: Displaying Texts on Graphics LCDs](#)

[Answers to Review Questions](#)

[Chapter 13: DRAM Memory Technology and DMA Controller](#)

[Section 13.1: Concept of Memory Cycle](#)

[Section 13.2: DRAM Technology](#)

[Section 13.3: Data Integrity in DRAM and ROM](#)

[Section 13.4: Concept of DMA](#)

[Answers to Review Questions](#)

[Chapter 14: Cache Memory](#)

[Section 14.1: Cache Memory Organizations](#)

[Section 14.2: Cache Memory and Multicore Systems](#)

[Answers to Review Questions](#)

[Chapter 15: MMU, Virtual Memory and MPU in ARM](#)

[SECTION 15.1: MMU and Virtual Memory in ARM](#)

[Section 15.2: Page Table Descriptors and Access Permission in ARM](#)

[Section 15.3: MPU and Memory Protection in ARM](#)

[Answers to Review Questions](#)

[Appendix A: IC Interfacing, System Design, and Failure Analysis](#)

[Section A.1: Overview of IC Technology](#)

[Section A.2: IC Interfacing and System Design Issues](#)

[Answers to Review Questions](#)



# Chapter 1: C for Embedded Systems

In reading this book we assume you already have some understanding of how to program in C language. In this chapter, we will examine some important concepts widely used in embedded system design that you may not be familiar with due to the fact that many generic C programming books do not cover them. In section 1.1, we examine the C data types for 32-bit systems. The bit-wise operators are covered in section 1.2.

## Section 1.1: C Data types for Embedded systems

In general C programming textbooks we see *char*, *short*, *int*, *long*, *float*, and *double* data types. The *float* and *double* data types standardized by the IEEE754 are covered in Volume 1 of this book series. We need to examine the size of C data types in the light of 32-bit processors such as ARM.

### char

The *char* data type is a byte size data whose bits are designated as D7-D0. It can be *signed* or *unsigned*. In the signed format the D7 bit is used for the + or - sign and takes values between -128 to +127. In the *unsigned char* we have values between 0x00 to 0xFF in hex or 0 to 255 in decimal since there is no sign and the entire 8 bits are used for the magnitude. (See Chapter 5 of Volume 1.)

The ARM microcontrollers use 4 bytes of memory space for 8-bit peripheral I/O ports. This is examined in the next chapter.

### short int

The *short int* (or usually referring as *short*) data type is a 2-byte size data whose bits are designated as D15-D0. It can be *signed* or *unsigned*. In the signed format, the D15 bit is used for the + or - sign and takes values between -32,768 to +32,767. In the *unsigned short int* we have values between 0x0000 to 0xFFFF in hex or 0 to 65,535 in decimal since there is no sign and the entire 16 bits are used for the magnitude. See Chapter 5 of Volume 1.

A 32-bit processor such as the ARM architecture reads the memory with a minimum of 32 bits on the 4-byte boundary (address ending in 0, 4, 8, and C in hex). If a *short int* variable is allocated straddling the 4-byte boundary, access to that variable is called an unaligned access. Not all the ARM processor support unaligned access. Those devices (including the TM4C123GH6PM used in the Tiva LaunchPad) supporting unaligned access pay a performance penalty by having to read/write the memory twice to gain access to one variable (see Example 1-1). Unaligned access can be avoided by either padding the variables with unused bytes (Keil) or rearranging the sequence of the variables (CCS) in allocation. By default, the compilers usually generate aligned variable allocation.

### Example 1-1

Show how memory is assigned to the following variables in aligned and unaligned allocation. Begin from memory location 0x20000000.

```
unsigned char a;  
unsigned short int b;  
unsigned short int c;
```

**Solution:**

## Unaligned allocation of variable c

a	b	b	c
c			

## Aligned allocation of variables by padding one byte between variable a and b

a		b	b
c	c		

## Aligned allocation of variables by rearranging the variable sequence

b	b	c	c
a			

---

### int

The *int* data type usually represents for the native data size of the processor. For example, it is a 2-byte size data for a 16-bit processor and a 4-byte size data for a 32-bit processor. This may cause confusion and portability issue. The C99 standard addressed the issue by creating a new set of integer variable types that will be discussed later. For now we will stick to the conventional data types.

The *int* data type of the ARM processors is 4-byte size and identical to *long int* data type described below.

### long int

The long int (or *long*) data type is a 4-byte size data whose bits are designated as D31-D0. It can be signed or unsigned. In the signed format the D31 bit is used for the + or - sign and takes values between  $-2^{31}$  to  $+2^{31}-1$ . In the unsigned long we have values between 0x00000000 to 0xFFFFFFFF in hex. See Chapter 5 of Volume 1. In the 32-bit microcontroller when we declare a long variable, the compiler sets aside 4 bytes of storage in SRAM. But it also makes sure they are aligned, meaning it places the data in locations with addresses ending with 0,4,8 and C in hex. This avoids unaligned data access performance penalty covered in Volume 1. The unsigned long is widely used in ARM for defining addresses since ARM address size is 32 bit long.

---

### Example 1-2

Show how memory is assigned to the following variables in aligned and unaligned allocation.  
Begin from memory location 0x20000000.

```
unsigned char a;  
unsigned short int b;  
unsigned short int c;  
unsigned int d;
```

### Solution:

Unaligned allocation of variable c

a	b	b	c
c	d	d	d
d			

Aligned allocation of variables by padding byte(s) between variable a and b

a		b	b
c	c		
d	d	d	d

Aligned allocation of variables by rearranging the variable sequence

d	d	d	d
b	b	c	c
a			

---

### long long

The *long long* data type is an 8-byte size data whose bits are designated as D63-D0. It can be signed or unsigned. In the signed format the D63 bit is used for the + or - sign and takes values between  $-2^{63}$  to  $+2^{63}-1$ . In the *unsigned long long* we have values between 0x0000000000000000 to 0xFFFFFFFFFFFFFF in hex. In the 32-bit microcontroller, when we declare a long long variable, the compiler sets aside 8 bytes of storage in SRAM and it makes sure they are aligned, meaning it places the data in locations with addresses ending with 0 and 8. This avoids unaligned data access performance penalty.

## Which data type to use?

It must be noted that while in the 8-bit microcontrollers we have to use the right data type for our variable, this is less of problem in 32-bit CPUs such as ARM. For example, for the number of days working in a month (or number of hrs in a day) we use unsigned char since it is less than 255. Using unsigned char in 8-bit microcontroller is important since it saves RAM space, memory access time, and computation clock cycles. If we use int instead, the compiler allocates 2 bytes in RAM and that is waste of RAM resource. The CPU will have to access the additional byte and perform arithmetic instructions with it even if the byte contains zero and has no effect on the result. This is a problem that we should avoid since an 8-bit microcontroller usually has few RAM bytes with slower clock speed for bus and CPU. In the case of 32-bit systems such as ARM, 1, 2, or 4 bytes of data will result in the same memory access time and computation time. Most of the 32-bit systems also have more generous amount of RAM to alleviate the concern of memory usage and allow padding for aligned access.

Data type	Size	Range
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
short int	2 bytes	-32,768 to 32,767
unsigned int	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295
long long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8 bytes	0 to 18,446,744,073,709,551,615

Table 1-1: ANSI C (ISO C89) integer data types and their ranges

### Notes

1. By default variables are considered as *signed* unless the *unsigned* keyword is used. As a result, *signed long* is the same as *long*; the *long long* is the same as *signed long long*, and so on with the exception of *char*. Whether *char* is signed or unsigned by default varies from compiler to compiler. In some compilers, including Keil, there is an option to choose if the *char* variable should be considered as *signed char* or *unsigned char* by default. (To choose this in Keil, go to *Project* menu and select *Options*. Then, in the *C/C++* tab, check or uncheck the choice *Plain char is signed*, as you desire.) It is a good practice to write out the *signed* keyword explicitly, when you want to define a variable as *signed char*.
2. In some compilers (including Keil and IAR) the *int* type is considered as *long int* while in some other compilers (including AVR and PIC compilers) it is considered as *short int*. In other words, the *int* type is commonly defined so that the processor can handle it easily. As we will see next, we can use *int16t* and *int32t* instead of *short* and *long* in order to prevent any kind of ambiguity and make the code portable between different processors and compilers.

## Data types in ISO C99 standard

While every C programmer has used ANSI C (ISO C89) data types, many C programmers are not familiar with the ISO C99 standard. In C standards, the sizes of integer data types were

not defined and are up to the compilers to decide. By conventions, char is one byte and short is two byte size. But int and long varies greatly among the compilers.

In ISO C99 standard, a set of data types were defined with number of bits and sign clearly defined in the data type names. (See Table 1-2.) The C ISO C99 standard is used extensively by embedded system programmer for RTOS (real time operating system) and system design. It is also supported by many C compilers. Notice the range is the same as ANSI C standard except it uses more descriptive syntax.

These integer data types are defined in a header file called *stdint.h*. You need to include this header file in order to use these data types.

Data type	Size	Range
<code>int8_t</code>	1 byte	-128 to 127
<code>uint8_t</code>	1 byte	0 to 255
<code>int16_t</code>	2 bytes	-32,768 to 32,767
<code>uint16_t</code>	2 bytes	0 to 65,535
<code>int32_t</code>	4 bytes	-2,147,483,648 to 2,147,483,647
<code>uint32_t</code>	4 bytes	0 to 4,294,967,295
<code>int64_t</code>	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<code>uint64_t</code>	8 bytes	0 to 18,446,744,073,709,551,615

Table 1-2: ISO C99 integer data types and their ranges

## Review questions

1. In an 8-bit system we use (char, unsigned char) for the number of months in a year.
2. For a system with 16-bit address, bus we use (int, unsigned int) for address definition.
3. For an ARM system the address is \_\_\_\_\_ bit wide and we use \_\_\_\_\_ data type for it.
4. True or false. In C programming of ARM, compiler makes sure data are aligned.

## Section 1.2: Bit-wise Operations in C

One of the most important and powerful features of the C language is its ability to perform bit manipulation. Because many books on C do not cover this important topic, it is appropriate to discuss it in this section. This section describes the action of bit-wise logic operators and provides some examples of how they are used.

### Bit-wise operators in C

While every C programmer is familiar with the logical operators AND (`&&`), OR (`||`), and NOT (`!`), many C programmers are less familiar with the bitwise operators AND (`&`), OR (`|`), EX-OR (`^`), inverter (`~`), shift right (`>>`), and shift left (`<<`). These bit-wise operators are widely used in software engineering for embedded systems and control; consequently, their understanding and mastery are critical in microprocessor-based system design and interfacing. See Table 1-3.

A	B	AND (A & B)	OR (A   B)	EX-OR (A^B)	Invert ~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

Table 1-3: Bit-wise Logic Operators for C

The following shows some examples using the C bit-wise operators:

```
0x35 & 0x0F results in 0x05      /* ANDing */  
0x04 | 0x68 results in 0x6C      /* ORing: */  
0x54 ^ 0x78 results in 0x2C      /* XORing */  
~0x55 results in 0xAA           /* Inverting 0x55 */
```

Examples 1-3 and 1-4 show how the bit-wise operators are used in C. Run the following programs on your simulator and examine the results.

### Example 1-3

Run the following program on your simulator and examine the results.

```
int main(void)  
{  
    volatile unsigned char temp; /* declare volatile otherwise  
the optimizer will remove it. */  
    temp = 0x35 & 0x0F;      /* ANDing : 0x35 & 0x0F = 0x05 */  
    temp = 0x04 | 0x68;      /* ORing   : 0x04 | 0x68 = 0x6C */  
    temp = 0x54 ^ 0x78;      /* XORing : 0x54 | 0x78 = 0x2C */  
    temp = ~0x55;          /* Inverting : ~0x55 = 0xAA */  
    while (1);  
    return 0;  
}
```

```
void SystemInit(void) /* required by the compiler */  
{  
}
```

---

## Setting and Clearing (masking) bits

As discussed in Volume 1 of the series, OR can be used to set a bit, and AND can be used to clear a bit. If you exam Table 1-3 closely, you will see that:

- Anything ORed with a 1 results in a 1; anything ORed with a 0 results in no change.
- Anything ANDed with a 1 results in no change; anything ANDed with a 0 results in a zero.
- Anything EX-ORed with a 1 results in the complement; anything EX-ORed with a 0 results in no change.

See Example 1-4.

### Example 1-4

The following program toggles only bit 4 of var1 continuously without disturbing the rest of the bits.

```
...  
int main(void)  
{  
    unsigned char var1;  
    while(1)  
    {  
        var1 = var1 | 0x10;      /* Set bit 4 (5th bit) of var1 */  
        var1 = var1 & 0xEF;    /* Clear bit 4 (5th bit) of var1 */  
    }  
  
    return 0;  
}  
...
```

Notice that we can also toggle the bit using XOR as shown below:

```
var1 = var1 ^ 0x10;
```

---

## Testing bit with bit-wise operators in C

In many cases of system programming and hardware interfacing, it is necessary to test a

given bit to see if it is high or low. For example, many devices send a high signal to state that they are ready for an action or to indicate that they have data. How can the bit (or bits) be tested? In such cases the unused bits are masked and then the remaining data is tested. See Example 1-5.

### Example 1-5

Write a C program to monitor bit 5 of var1. If it is HIGH, change value of var2 to 0x55; otherwise, change value of var2 to 0xAA.

#### Solution:

```
...
while(1)
{
    if (var1 & 0x20)      /* check bit 5 (6th bit) of var1 */
        var2 = 0x55;      /* this statement is executed if bit 5 is a 1 */
    else
        var2 = 0xAA;      /* this statement is executed if bit 5 is a 0 */
}
...
...
```

## Bit-wise shift operation in C

There are two bit-wise shift operators in C. See Table 1-4.

Operation	Symbol	Format of Shift Operation
Shift Right	>>	data >> number of bit-positions to be shifted right
Shift Left	<<	data << number of bit-positions to be shifted left

Table 1-4: Bit-wise Shift Operators for C

The following shows some examples of shift operators in C:

1. 0b00010000 >> 3 /\* it equals 00000010. Shifting right 3 times \*/
2. 0b00010000 << 3 /\* it equals 10000000. Shifting left 3 times \*/
3. 1 << 3 /\* it equals 00001000. Shifting left 3 times \*/

## Compound Operators

In C language, whenever the left-hand-side of the assignment operator (=) and the first operand on the right-hand-side are identical we can avoid repeating the operand by using the compound operators. As shown in Table 1-5, in compound operators, the operators are mentioned just on the left-hand-side of the equal sign and the first operand is omitted.

Instruction	Its equivalent using compound operators
a = a + 6;	a += 6;

a = a - 23;	a -= 23;
y = y * z;	y *= z;
z = z / 25;	z /= 25;
w = w   0x20;	w  = 0x20;
v = v & mask;	v &= mask;
m = m ^ togBits;	m ^= togBits.

Table 1-5: Some Compound Operator Examples

## Review Questions

1. What is result of 0x2F &0x27?
2. What is result of 0x2F | 0x27?
3. What is result of 0x2F ^ 0x27?
4. What is result of 0x2F >> 3?
5. What is result of 0x27 << 4?
6. In Example 1-5 what is stored in var2 if the value of var1 is 0x03?

Reading of the articles by Dan Saks and Michael Barr on embedded.com is strongly recommended:

[http://www.embedded.com/user/Dan\\_Saks#](http://www.embedded.com/user/Dan_Saks#)

<http://www.embedded.com/user/Michael.Barr>

## Answer to Review Questions

### Section 1.1: C Data types for Embedded systems

1. unsigned char
2. unsigned int
3. 32 – unsigned long (or uint32\_t)
4. True

### Section 1.2: Bitwise Operations in C

1. 0x27
2. 0x2F
3. 0x08
4. 0x05
5. 0x70
6. 0xAA



## Chapter 2: TI ARM I/O Programming

In microcontroller, we use the general purpose input output (GPIO) pins to interface with LED, switch (SW), LCD, Keypad, and so on. This chapter covers the programming of GPIO using LED, switches, and seven segment LEDs as examples. This is a very important chapter since the vast majority of embedded products have some kind of I/O. More importantly, this chapter sets the stage for understanding of peripheral I/O addresses and how they are accessed and used in ARM processors. Because some of the core materials covered in this chapter are widely used in subsequent chapters, we urge you to study this chapter thoroughly before moving on to other chapters. Section 2.1 examines the memory and I/O map of the TI ARM chip. Section 2.2 shows how to access the special function registers associated with the GPIO of TI ARM. In section 2.2, we also use simple LEDs and SWs to show the programming of GPIO. Section 2.3 examines the 7-segment LED connection to ARM and how to programming it.

## Section 2.1: Texas Instruments Tiva TM4C123GH6PM Microcontroller

The TI LaunchPad uses the TM4C123GH6PM microcontroller. The earlier version of the LaunchPad used the LM4F120 chip. The TM4C123GH6PM is an enhanced version of the LM4F120 and all the programs written for LM4F120 will run on TM4C123GH6PM without any modification but not the other way around. In other words, it is upward compatible. The TM4C123GH6PM chip has 256K bytes (256KB) of on-chip Flash memory for code, 32KB of on-chip SRAM for data, and a large number of on-chip peripherals as shown in Figures 2-1 and 2-2.

### Note

For other chips in this series see the following website:

[http://www.ti.com/lscds/ti/arm/arm\\_cortex\\_m\\_microcontrollers/arm\\_cortex\\_m4/stellaris\\_arm](http://www.ti.com/lscds/ti/arm/arm_cortex_m_microcontrollers/arm_cortex_m4/stellaris_arm)

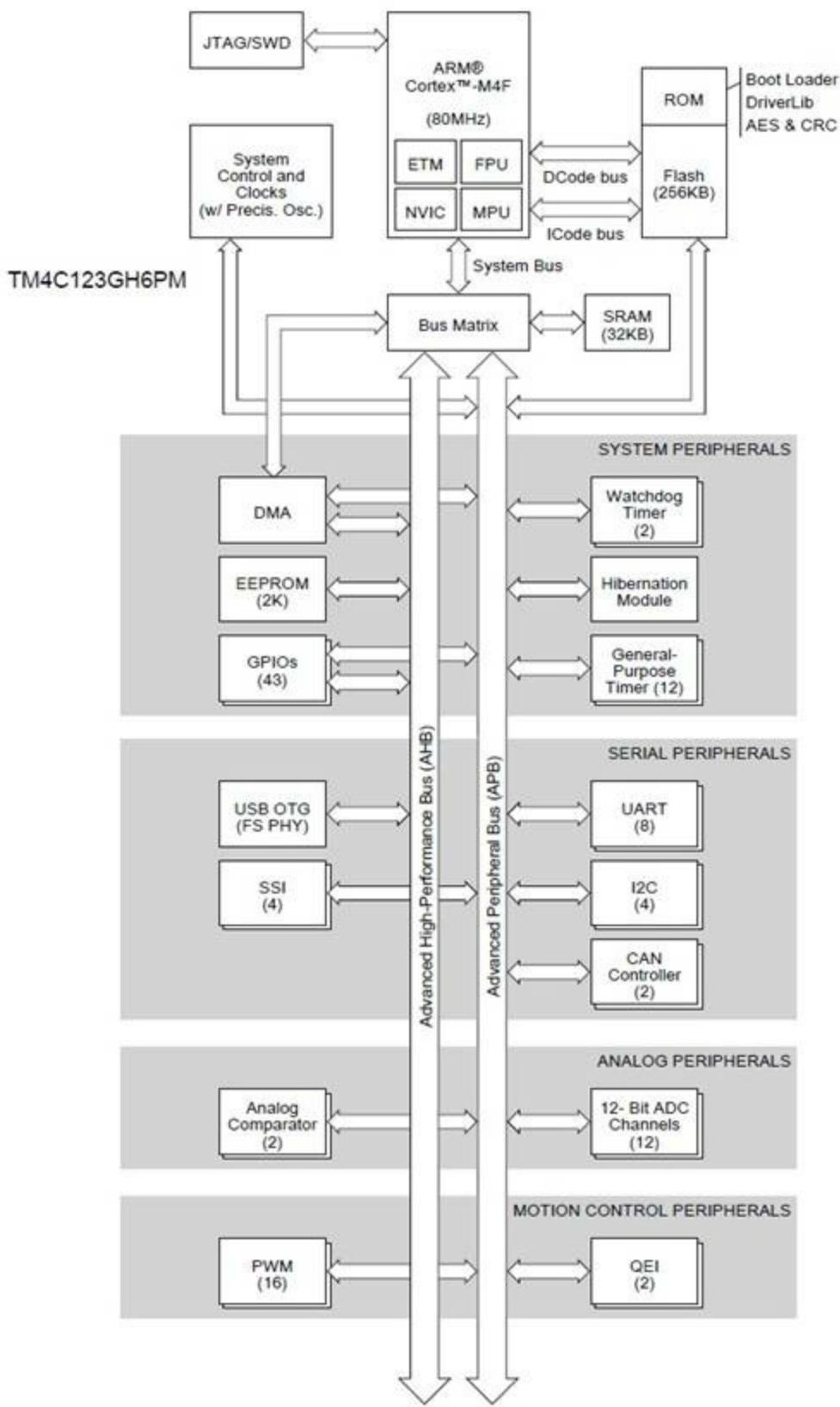


Figure 2-1: TI Tiva TM4C123GH6PM Microcontroller High-Level Block Diagram

# Tiva™ EK-TM4C123GXL LaunchPad

- ◆ ARM® Cortex™-M4F 64-pin 80MHz TM4C123GH6PM
- ◆ On-board USB ICDI (In-Circuit Debug Interface)
- ◆ Micro AB USB port
- ◆ Device/ICDI power switch
- ◆ BoosterPack XL pinout also supports existing BoosterPacks
- ◆ 2 user pushbuttons
- ◆ Reset button
- ◆ 3 user LEDs (1 tri-color device)
- ◆ Current measurement test points
- ◆ 16MHz Main Oscillator crystal
- ◆ 32kHz Real Time Clock crystal
- ◆ 3.3V regulator
- ◆ Support for multiple IDEs:

mentor  
embedded

ST  
IAR  
SYSTEMS

ARM  
KEIL

Composer  
Studio

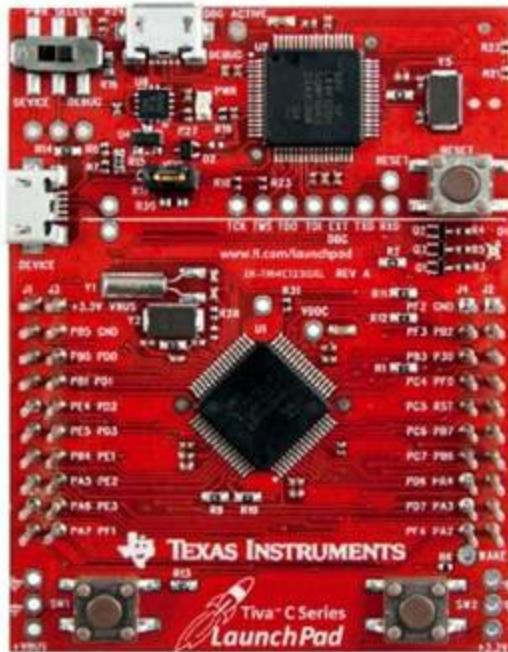


Figure 2-2: TI Tiva LaunchPad Trainer board

As we stated in Volume 1, the ARM has 4GB (Giga bytes) of memory space. It also uses memory mapped I/O meaning the I/O peripheral ports are mapped into the 4GB memory space. See Table 2-1 and Figure 2-3 for memory map for TM4C123GH6PM chip.

	Allocated size	Allocated address
Flash	256KB	0x00000000 to 0x0003FFFF
SRAM	32KB	0x20000000 to 0x20007FFF
I/O	All the peripherals	0x40000000 to 0x400FFFFFF

Table 2-1: Memory Map in TM4C123GH6PM

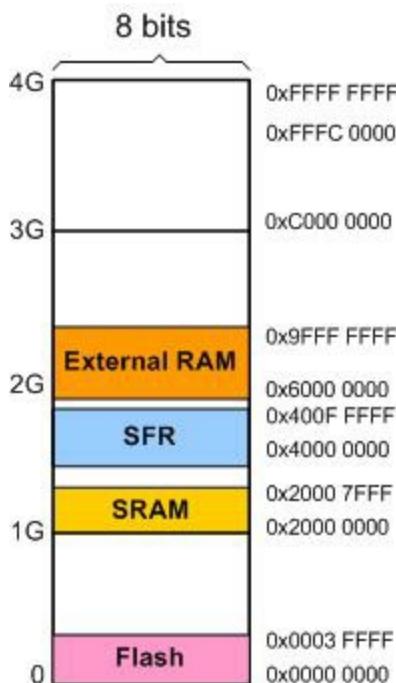


Figure 2-3: Memory Map

Regarding Figure 2-3, the following points must be noted:

- 1) 256KB of Flash memory is used for program code. One can also store in Flash ROM constant (fixed) data such as look-up table if needed. The Flash memory is organized in 1-KB block. Each block can be independently erased and written to.
- 2) The 32KB SRAM is for variables, scratch pad, and stack. It starts at address 0x20000000. Address aliases can be used for a portion of SRAM to allow individual bit-access. This is called *bit-banding* and is discussed in Volume 1.
- 3) The peripherals such as I/O, Timer, ADC are mapped to addresses starting at 0x40000000. In TM4C123GH6PM the upper limit is 0x400FFFFF. For details see Table 2-4 in TM4C123GH6PM datasheet. The upper limit address can vary among the family members of ARM chips depending on the number of peripherals the chip supports.

## GPIO

The general purpose I/O ports in TM4C123GH6PM ARM are designated as port A to port F. The following shows the address range assigned to each GPIO ports:

- GPIO Port A : 0x4000.4000 to 0x4000.4FFF
- GPIO Port B : 0x4000.5000 to 0x4000.5FFF
- GPIO Port C : 0x4000.6000 to 0x4000.6FFF
- GPIO Port D : 0x4000.7000 to 0x4000.7FFF
- GPIO Port E : 0x4002.4000 to 0x4002.4FFF
- GPIO Port F : 0x4002.5000 to 0x4002.5FFF

See Figure 2-4.

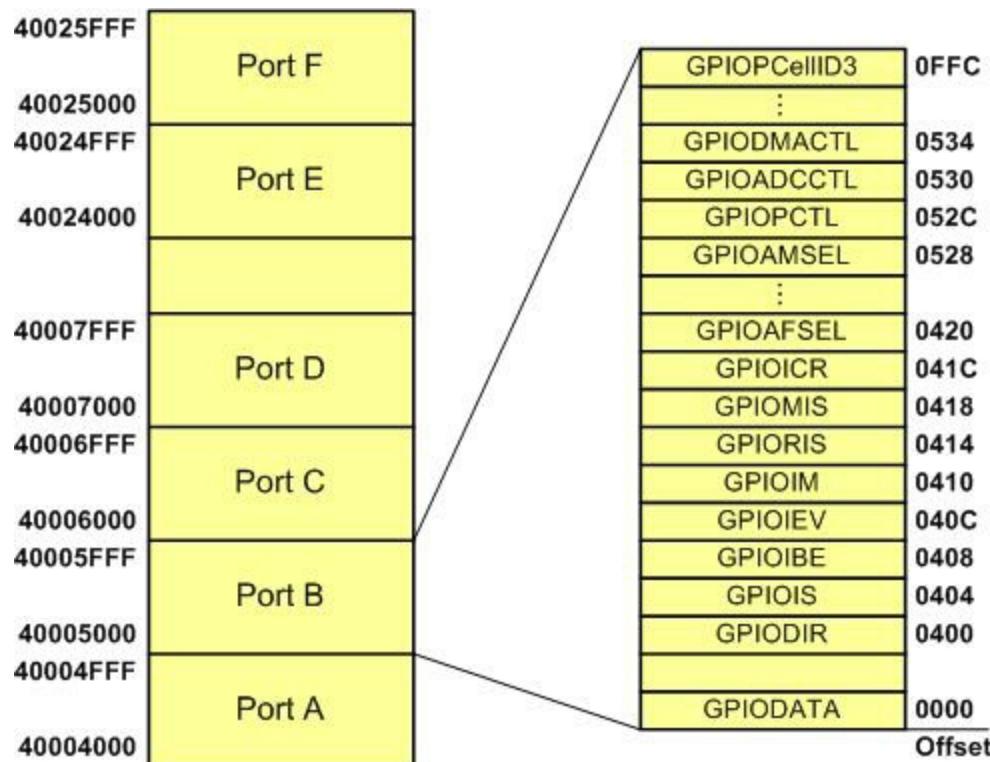


Figure 2-4: GPIO Memory Map

Notice that total of 4K bytes of memory space is assigned to each of the GPIO port. Why so many address locations? The reason is each GPIO has a large number of special function

registers associated with it and the GPIO DATA register supports bit-banding. The GPIO DATA register is 8-bit wide. With bit-banding, it will need 256 words (4 bytes each, 1 KB total). As we will see in the next section, these special function registers give the GPIO ports a very rich feature set.

## Review Questions

1. TM4C123GH6PM has \_\_\_\_\_ KB of on-chip Flash memory.
2. TM4C123GH6PM has \_\_\_\_\_ KB of on-chip SRAM memory.
3. TM4C123GH6PM Flash memory is used mainly for (program code, data).
4. TM4C123GH6PM SRAM memory is used for (program code, data).
5. Give the address space assigned to the Flash memory of TM4C123GH6PM.
6. Give the address space assigned to the peripheral IO of TM4C123GH6PM.

## Section 2.2. GPIO (General Purpose I/O) Programming and Interfacing

While memory holds code and data for the CPU to process, the I/O ports are used by the CPU to access input and output devices. In the microcontroller we have two types of I/O. They are:

- a. **General Purpose I/O (GPIO):** The GPIO ports are used for interfacing devices such as LEDs, switches, LCD, keypad, and so on.
- b. **Special purpose I/O:** These I/O ports have designated function such as ADC (Analog-to-Digital), Timer, UART (universal asynchronous receiver transmitter), and so on.

We have dedicated many chapters to these special purpose I/O ports. In this chapter, we examine the GPIO and its interfacing to LEDs, switches, and 7-segment LEDs and show how to access them using C programs.

### I/O Pins in TI Tiva LaunchPad

The ARM chip used in TI Tiva LaunchPad is Tiva C series TM4C123GH6PM and has ports A, B, C, D, E, and F. The pins are designated as PA0-PA7, PB0-PB7, PC0-PC7, PD0-PD7, PE0-PE5, and PF0-PF4. See Figure 2-5.

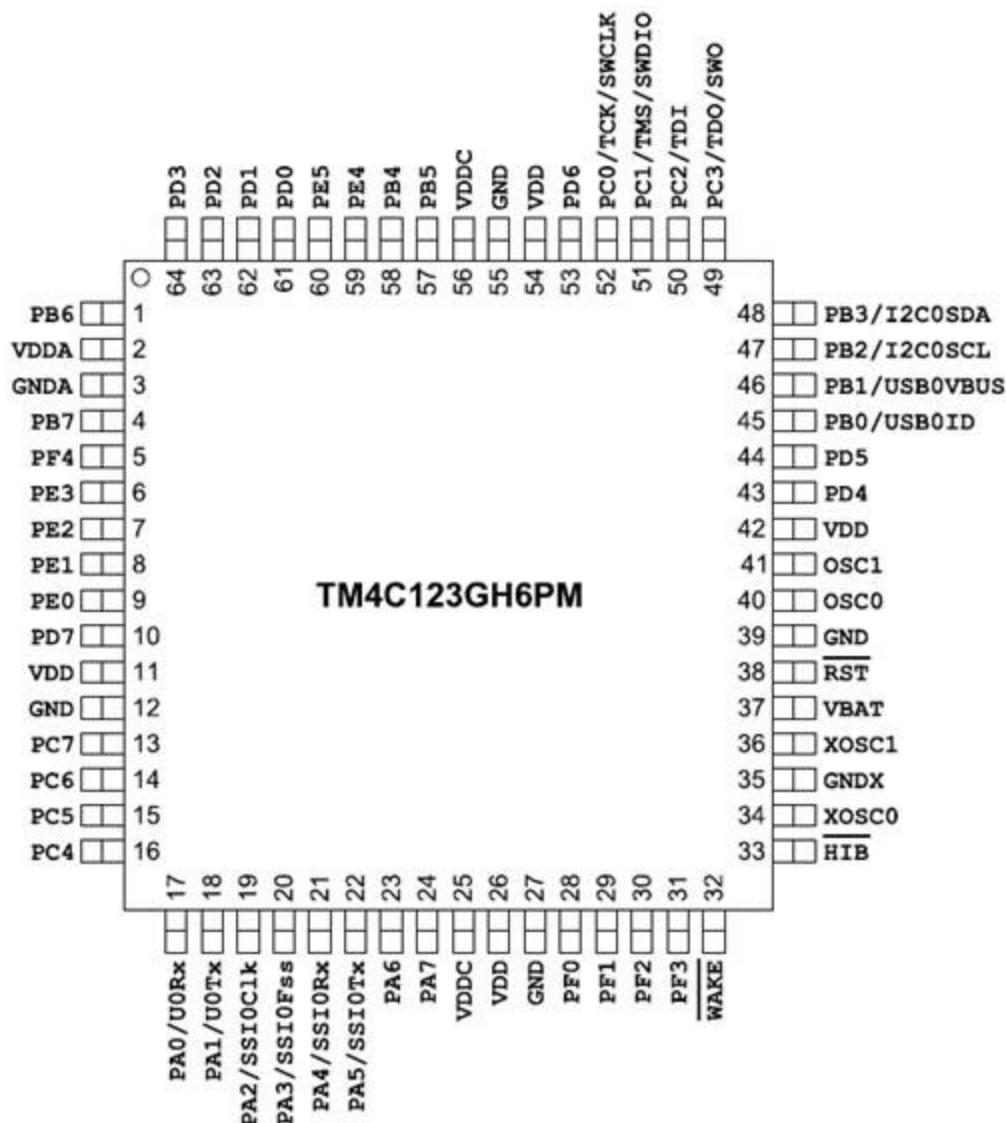


Figure 2-5: TM4C134GH6PM Pin-out

Notice that ports PE and PF do not have all the 8 pins implemented. PC0-PC3 is used for JTAG connections to the debugger on the LaunchPad. It also must be noted that the Tiva TM4C123GH6PM is upward compatible from the TI Stellaris LM4F120 ARM chip used in earlier version of the TI Stellaris LaunchPad Evaluation Kit. In other words, the programs written for LM4F120 chip will run on TM4C123GH6PM but not the other way around. The LM in LM4F120 is for Luminary Micro, a major producer of ARM chips, before it was acquired by Texas Instruments.

The ARM chips have two buses: *APB (Advanced Peripheral Bus)* and *AHB (Advanced High-Performance Bus)*. The AHB bus is much faster than APB. The AHB allows one clock cycle access to the peripherals. The APB is slower and its access time is minimum of 2 clock cycles.

The I/O ports addresses assigned to the PA-PF for APB are as follow:

- GPIO Port A (APB): 0x4000.4000
- GPIO Port B (APB): 0x4000.5000
- GPIO Port C (APB): 0x4000.6000
- GPIO Port D (APB): 0x4000.7000
- GPIO Port E (APB): 0x4002.4000
- GPIO Port F (APB): 0x4002.5000 (This port is connected to LED and SW on TI Tiva LaunchPad)

The Base addresses for the GPIOs of AHB is as follow:

- GPIO Port A (AHB): 0x4005.8000
- GPIO Port B (AHB): 0x4005.9000
- GPIO Port C (AHB): 0x4005.A000
- GPIO Port D (AHB): 0x4005.B000
- GPIO Port E (AHB): 0x4005.C000
- GPIO Port F (AHB): 0x4005.D000

For APB, AHB, and single cycle access time see Chapter 6 of ARM Assembly Language Programming and Architecture By Mazidi and Naimi book in this series.

There are many registers associated with each of the above I/O ports and they have designated addresses in the memory map. The above addresses are the Base addresses meaning that within that base address we have the registers associated with that port, as we will see next.

## Direction and Data Registers

Generally every microcontroller has minimum of two registers associated with each of I/O port. They are *Data Register* and *Direction Register*. As we discussed in Chapter 0 (available on <http://www.MicroDigitalEd.com>), for output we need a latch and for input we need a tri-state buffer. The Direction register is used to make the pin either input or output. After the Direction register is properly configured, then we use the Data register to actually write to the pin or read data from the pin. It is the Direction register (when configured as output) that allows the information written to the Data register to be driven to the pins of CPU. The same way, it is the Direction register (when configured as input) that allows the signal present at the CPU pin to be brought into the Data register. See Figure 2-6.

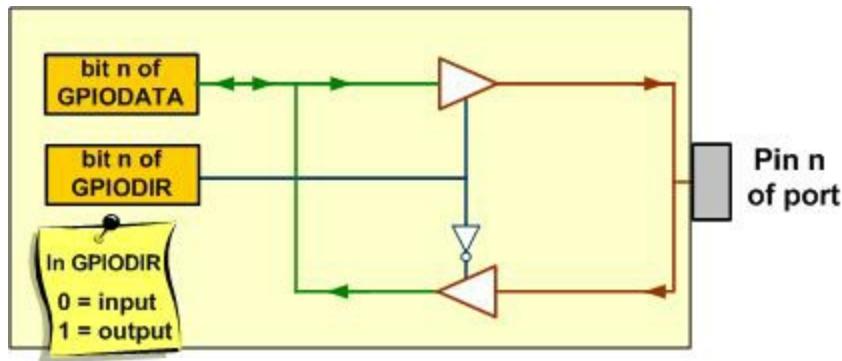


Figure 2-6: The Data and Direction Registers and a Simplified View of an I/O pin

## Data Register (GPIODATA) in TI ARM

The GPIO Data Register (GPIODATA) is located at the offset address of 0x0000 from the base address of that port. This is shown below. Because GPIODATA register supports bit-banding, it occupies 256 words (offset 0x000 – 0x3FC). In order to be able to write to all 8 bits of the GPIO Data Register at once, one must use the offset 0x3FC.



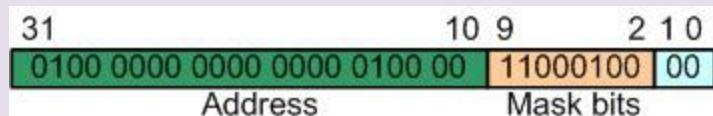
Figure 2-7: GPIOData Register

### Bit banding case study (For experts only)

You might want to know the reason 256 words (1024 bytes) are set aside for GPIODATA. As shown in the following figure, in order to make the addresses word aligned, bits 1 and 0 are always set to 0. Using bits 2 to 9 of the address, shows the bits that must be changed while writing to the GPIODATA register. For example, writing to address 0x34 (0000110100 in binary) means that pins 0, 2, and 4 of the port must be changed while the other pins remain unchanged. To change all the pins of the port, all bit masks (bits 2 to 9) must be set. This makes the offset address of 0x3FC (0011111100 in binary).



As an example, writing to address 0x40004310 means that bits 2, 6, and 7 of Port A must be changed since 0x40004000 is the base address of Port A. See the following figure.



## Direction Register in TI ARM

In the case of TI ARM chip, each of the Direction register bit needs to be a 0 to configure the port pin as input and a 1 as output. Notice this is different from some other microcontrollers in which 0 is for Output and 1 is for Input. The address of the GPIO Direction register is located at the offset address of 0x400 from the Base address of that port. This is shown below.



**Note:** D0 to D7 are used to set the direction for pins 0 to 7 of the port.

- 1: Output
- 0: Input

Figure 2-8: GPIO DIR Register

See Example 2-1 and Table 2-2 for clarification.

### Example 2-1

Find the physical address of the GPIO DATA and GPIO DIR registers for PORTF if the Base address of the PORTF is 0x40025000.

**Solution:**

The physical address location of the GPIO DATA for PORTF starts at  $0x40025000 + 0x000 = 0x40025000$ . In order to be able to write to all 8 bits of the data register at once you need to use  $0x40025000 + 0x3FC = 0x400253FC$ .

The physical address location of GPIO DIR for the PORTF is  $0x40025000 + 0x400 = 0x40025400$ .

Offset	Name	Description	Type	Reset Value
0x0000	GPIO DATA	Data register	R/W	0
0x0400	GPIO DIR	Direction register	R/W	0
0x0510	GPIO PUR	Pull-up Register	R/W	0*
0x051C	GPIO DEN	Digital Enable	R/W	0

**Note:** On reset, all pins are in input tri-state mode; except PA[1:0], PA[5:2], PB[3:2], PC[3:0]

Table 2-2: Some GPIO Registers

To access the I/O pins of TM4C123GH6PM ARM chip used in LaunchPad, we need to examine two more registers. They are GPIO DEN and RCGCGPIO.

### The GPIO Digital Enable Register

Each pin of the TI ARM chip can have multiple functions. We choose the function by programming an special function register (SFR).

Using a single pin for multiple functions is called *pin multiplexing* and is widely used by microcontrollers. In the absence of pin multiplexing a microcontroller will need several hundred pins to support all its on-chip features. For example, a given pin can be used as digital I/O, analog input, or I2C pin. Of course not all at the same time.

The GPIO DEN (Digital Enable) special function register allows us to enable the pin to be used as digital I/O pin instead of analog function. Each PORT of A-F has its own GPIO DEN register and one can enable the digital I/O for each pin of a given port. This is shown below.



**Note:** D0 to D7 are used to Enable the digital circuit for pins 0 to 7 of the port.

1: The digital functions for the corresponding pin are enabled.

0: The digital functions for the corresponding pin are disabled.

Figure 2-9: GPIODEN Register

## Example 2-2

Find the physical address of the GPIODEN register for PORTF if the Base address of the PORTF is 0x40025000.

### Solution:

According to Table 2-2, 0x51C is the offset for GPIODEN. So, the physical address location of the GPIODEN for PORTF is  $0x40025000 + 0x51C = 0x4002551C$ .

## The GPIO Clock enable for all the I/O ports

The RCGCGPIO is used to enable the clock source for the I/O port circuitry. If an I/O port is not used, the clock source to it can be disabled in order to save power. There is only one RCGCGPIO special function register for all the ports and each bit of that register is used to enable the clock source to one of the ports. In the case of TI TM4C123GH6PM since we have only ports A through F, the lower 6 bits of this register are used. The register bits are shown below.

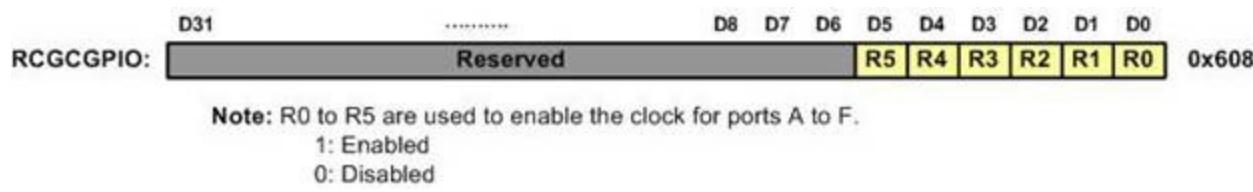


Figure 2-10: RCGCGPIO (Run mode Clock Gating Control) Register

In RCGCGPIO the base address is 0x400F.E000 and the offset is 0x608; as a result, the physical address is  $0x400FE000 + 0x608 = 0x400FE608$ . See Example 2-3.

## Example 2-3

Show how a) to enable the clock to PORTF and b) enable the digital I/O feature of pins for PORTF1, PORTF2, and PORTF3.

### Solution:

- To enable the clock source to PORTF we need to set HIGH the D5 of the RCGCGPIO register. We can OR hex value 0x20 (0b00100000) with RCGCGPIO register to make sure it leaves clock source to other ports unchanged.

- b) We need to write value 0b00001110=0x0E to PORTFDEN register located at address 0x4002551C

This is a very important register. Without clock the port will not work. Any access to the registers associated with the port before the clock is enabled will result in a hard fault and the program crashes. Also notice that in other TI Tiva ARM family members more bits of this register are used depending on how many I/O ports that chip supports. For example, in TI ARM TM4C123GH6GPE, the I/O ports are from A to P. In this case, 14 bits of this register are used, one for each port.

## LED connection in TI Tiva LaunchPad

In the TI Tiva LaunchPad we have a tri-color LED connected to PF2 (red), PF3 (blue), and PF4 (green) via DTC114EE1G NPN transistor. The transistor plays the role of a current switch. The tri-color (red, blue, green) LED is popular in many trainer kit for embedded systems.

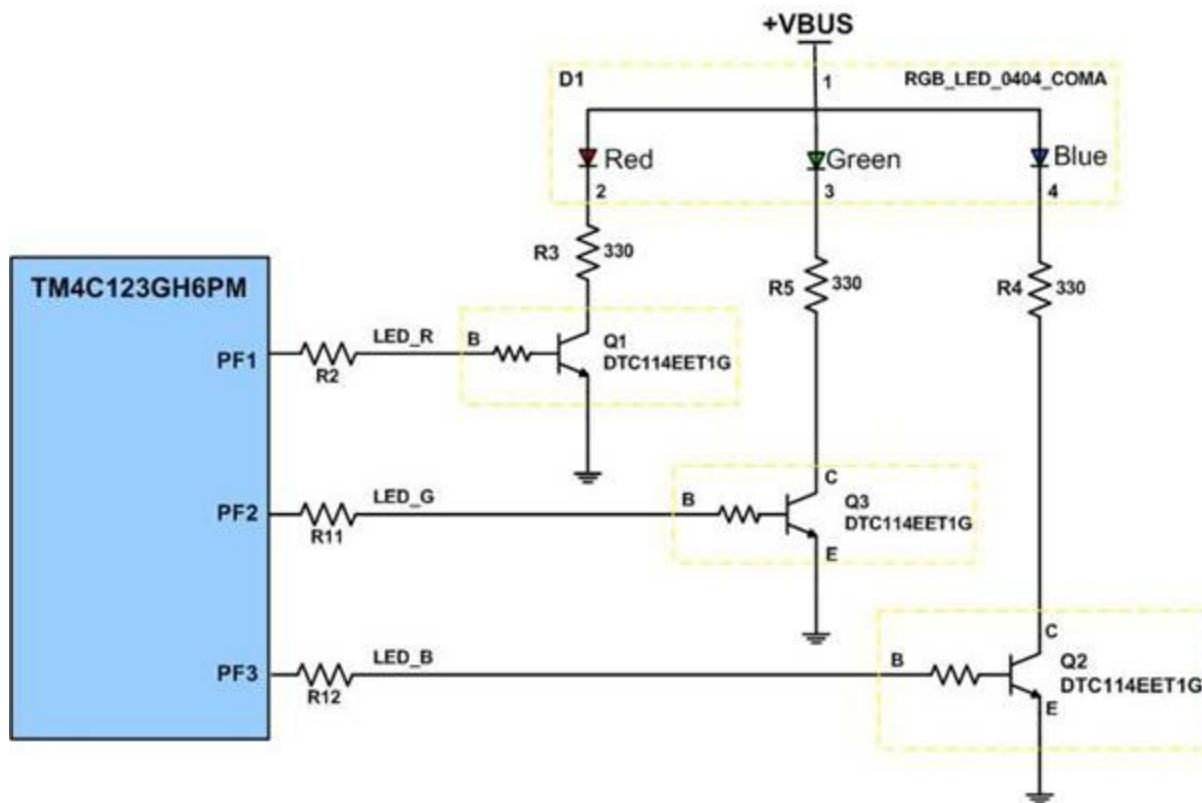


Figure 2-11: LED connection to PORTF in TI Tiva LaunchPad

## Toggling LEDs in TI Tiva LaunchPad in C

To toggle the RGB LEDs of the TI Tiva LaunchPad, the following steps must be followed.

- 1) enable the clock to PORTF, since we cannot do anything with the port registers until the clock is enabled,
- 2) set the Direction register bits of 1, 2, and 3 of PORTF as output,
- 3) enable the digital I/O feature of PORTF,
- 4) write HIGH to PORTF data register,
- 5) call a delay,

- 6) write LOW to PORTF data register,
- 7) call a delay,
- 8) Repeat steps 4 to 7.

Program 2-1 shows one way to toggle RGB LEDs continuously.

#### Program 2-1: Toggling LEDs in C (using special function registers by their addresses)

```

/* p2_1.c Toggling LEDs in C using registers by addresses */
/* This program toggles all three LEDs for 0.5 second ON and 0.5 second OFF.*/

/*PF1 - red LED
PF2 - blue LED
PF3 - green LED
They are high active (a '1' turns ON the LED).*/

/* PORTF data register */
#define PORTFDAT (*((volatile unsigned int*)0x400253FC))
/* PORTF data direction register */
#define PORTFDIR (*((volatile unsigned int*)0x40025400))
/* PORTF digital enable register */
#define PORTFDEN (*((volatile unsigned int*)0x4002551C))
/* run mode clock gating register */
#define RCGCGPIO (*((volatile unsigned int*)0x400FE608))
/* coprocessor access control register */
#define SCB_CPAC (*((volatile unsigned int*)0xE000ED88))

void delayMs(int n);      /* function prototype for delay */

int main(void)
{
    /* enable clock to GPIOF at clock gating register */
    RCGCGPIO |= 0x20;
    /* set PORTF pin3-1 as output pins */
    PORTFDIR = 0x0E;
    /* set PORTF pin3-1 as digital pins */
    PORTFDEN = 0x0E;

    while(1)
    {
        /* write PORTF to turn on all LEDs */
        PORTFDAT = 0x0E;
        delayMs(500);
        /* write PORTF to turn off all LEDs */
        PORTFDAT = 0;
        delayMs(500);
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)

```

```

        {} /* do nothing for 1 ms */

}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access*/
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB_CPAC |= 0x00F00000;
}

```

Program 2-2 shows the Assembly version of the Program 2-1.

**Program 2-2: Toggling LEDs in Assembly (using special function registers by their addresses)**

```

; p2_2.s
;
; Toggling LEDs in Assembly
; This program toggles all three LEDs
; for 0.5 second ON and 0.5 second OFF.
;
; THUMB

AREA      .text!, CODE, READONLY, ALIGN=2
ALIGN
EXPORT   __main

_main
    ; enable clock to GPIOF at clock gating register
    LDR      R0, =0x400FE608    ;RCGC register address
    LDR      R1, [R0]
    ORR      R1, #0x20
    STR      R1, [R0]

    ; set PORTF pin3-1 as output pins
    LDR      R0, =0x40025400
            ;DIR register address
    MOV      R1, #0x0E
    STR      R1, [R0]

    ; set PORTF pin3-1 as digital pins
    LDR      R0, =0x4002551C
            ; Digital Enable register address
    MOV      R1, #0x0E
    STR      R1, [R0]

loop
    ; write PORTF to turn on all LEDs
    LDR      R0, =0x400253FC
            ;PORTF address
    MOV      R1, #0x0E
    STR      R1, [R0]
    ; delay a little while
    LDR      R0, =500
    BL       delayMs    ; call delayMs

    ; write PORTF to turn off all LEDs

```

```

LDR      R0, =0x400253FC
MOV      R1, #0
STR      R1, [R0]
; delay a little while
LDR      R0, =500
BL       delayMs

; repeat the loop
B       loop

; This subroutine performs a delay of n ms.
; n is the value in R0
delayMs
    MOVS   R3, R0
    BNE    L1 ; if n = 0, return
    BX     LR ; return
L1
    LDR    R4, =5336
    ; do inner loop 5336 times (for 16 MHz CPU clock)
L2
    SUBS   R4, R4, #1      ; inner loop
    BNE    L2
    SUBS   R3, R3, #1      ; do outer loop n times
    BNE    L1
    BX     LR

; This subroutine grants access to
; floating point coprocessor.
; It is called by the startup code.
EXPORT SystemInit
SystemInit
    LDR    R0, =0xE000ED88
          ; Enable CP10,CP11
    LDR    R1, [R0]
    ORR    R1,R1,#(0xF << 20)
    STR    R1, [R0]
    BX     LR

; This variable is required for the
; startup code though not used.
EXPORT __use_two_region_memory
__use_two_region_memory EQU 1

ALIGN
END

```

In Program 2-1, notice how we define the physical address of the special function registers belonging to the I/O ports. This is tedious and error prone. Often the manufacturer of the device will provide these definitions in a C header file. In fact, there are two different header files available for TM4C123GH6PM.

With Keil uVision IDE, you can find <C:\Keil\ARM\INC\TI\TM4C123\TM4C123GH6PM.h> (In your C drive, look for Keil directory, then click on ARM, then click on INC, click on TI, click on TM4C123, and click on TM4C123GH6PM.h). In this file, each port is defined as a pointer to a *struct* with the registers as the members of the *struct*. For example, the Direction Register of Port F is referred to as `GPIOF->DIR` and the Data Register of Port F is referred to as `GPIOF->DATA`.

and so on. Program 2-1 is rewritten with this header file as Program 2-3a.

### Program 2-3a: Toggling LEDs in C (using special function registers by their names in header file)

```
/* p2_3a.c: Toggling LEDs using special function registers by their names defined in
the compiler header files */

#include "TM4C123GH6PM.h"

void delayMs(int n);

int main(void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2 1) as output */
    GPIOF->DIR = 0x0E;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN = 0x0E;

    while(1)
    {
        GPIOF->DATA = 0x0E;      /* turn on all LEDs */
        delayMs(500);

        GPIOF->DATA = 0;         /* turn off all LEDs */
        delayMs(500);
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}
```

If you have installed TivaWare (<http://www.ti.com/tool/sw-tm4c>), you may use the header file in the TivaWare install folder C:\ti\TivaWare\_C\_Series-1.1\inc\tm4c123gh6pm.h. In this header file, each register is defined with its own name. For example, the Direction Register of Port F is referred to as `GPIO_PORTF_DIR_R` and the Data Register of Port F is referred to as `GPIO_PORTF_DATA_R`. Program 2-3b is rewritten with this header file. For the rest of this book, we will use the header file in Keil install folder so that it is not necessary to install TivaWare.

## Program 2-3b: Toggling LEDs in C (using special function registers by their names in header file)

```
/* p2_3b.c: Toggling LEDs using special function registers by their names defined in
the TivaWare header file */

#include <stdint.h>
#include "C:\ti\TivaWare_C_Series-1.1\inc\tm4c123gh6pm.h"

void delayMs(int n);

int main(void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R5;
    /* enable the GPIO pins for the LED (PF3, 2 1) as output */
    GPIO_PORTF_DIR_R = 0x0E;
    /* enable the GPIO pins for digital function */
    GPIO_PORTF_DEN_R = 0x0E;

    while(1)
    {
        GPIO_PORTF_DATA_R = 0x0E;      /* turn on all LEDs */
        delayMs(500);

        GPIO_PORTF_DATA_R = 0;         /* turn off all LEDs */
        delayMs(500);
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor. */
    NVIC_CPAC_R |= 0x00F00000;
}
```

Program 2-4 shows how to toggle a single LED on TI Tiva LaunchPad.

## Program 2-4: Toggling a single LEDs in C

```
/* p2.4.c: Toggling a single LED

/* This program turns on the red LED and toggles the blue LED 0.5 sec on and 0.5 sec
off. */
```

```

#include "TM4C123GH6PM.h"

void delayMs(int n);

int main(void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2 1) as output */
    GPIOF->DIR = 0x0E;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN = 0x0E;
    /* turn on red LED only and leave it on */
    GPIOF->DATA = 0x02;

    while(1)
    {
        GPIOF->DATA |= 4;      /* turn on blue LED */
        delayMs(500);

        GPIOF->DATA &= ~4;    /* turn off blue LED */
        delayMs(500);
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor. */
    SCB->CPACR |= 0x00F00000;
}

```

Program 2-5 shows how to toggle a single LED on TI Tiva LaunchPad using Assembly language.

#### Program 2-5: Toggling a single LEDs in Assembly

```

; p2_5.s
; Toggling a single LED in Assembly
;
; This program turns on the red LED and toggles
; the blue LED 0.5 sec on and 0.5 sec off.
;
```

THUMB

```
AREA    |.text|, CODE, READONLY, ALIGN=2
ALIGN
EXPORT  __main
```

main

```
; enable clock to GPIOF at clock gating register
LDR    R0, =0x400FE608 ; RCGC reg. addr.
LDR    R1, [R0]
ORR    R1, #0x20
STR    R1, [R0]

; set PORTF pin3-1 as output pins
LDR    R0, =0x40025400
        ; DIR register address
MOV    R1, #0x0E
STR    R1, [R0]

; set PORTF pin3-1 as digital pins
LDR    R0, =0x4002551C
        ; Digital Enable reg. addr.
MOV    R1, #0x0E
STR    R1, [R0]

; turn on red LED only and leave it on
LDR    R0, =0x400253FC
        ; PORTF reg. address
MOV    R1, #2
STR    R1, [R0]
```

loop

```
; write PORTF to turn on blue LED
LDR    R0, =0x400253FC
LDR    R1, [R0]
ORR    R1, #4
STR    R1, [R0]
; delay a little while
LDR    R0, =500
BL     delayMs

; write PORTF to turn off blue LED
LDR    R0, =0x400253FC
LDR    R1, [R0]
AND    R1, #~4
STR    R1, [R0]
; delay a little while
LDR    R0, =500
BL     delayMs

; repeat the loop
B      loop
```

```
; This subroutine performs a delay of n ms.
; n is the value in R0
```

delayMs

```
MOVS   R3, R0
```

```

        BNE    L1           ; if n = 0, return
        BX     LR           ; return
L1      LDR    R4, =5336
        ; do inner loop 5336 times (for 16 MHz CPU clock)
L2      SUBS   R4, R4, #1  ; inner loop
        BNE    L2
        SUBS   R3, R3, #1  ; do outer loop n times
        BNE    L1
        BX     LR           ; return

; This subroutine grants access to
; floating point coprocessor.
; It is called by the startup code.
EXPORT SystemInit
SystemInit
        LDR    R0, =0xE000ED88
        ; Enable CP10,CP11
        LDR    R1, [R0]
        ORR    R1,R1,#(0xF << 20)
        STR    R1, [R0]
        BX    LR

; This variable is required for
; the startup code though not used.
EXPORT __use_two_region_memory
__use_two_region_memory EQU 1

        ALIGN
END

```

Program 2-6 shows toggling all the bits of PORTA on TI Tiva LaunchPad. Use your logic probe or an oscilloscope to see the pins going ON and OFF.

#### Program 2-6: Toggling all bits of PORTA in C

```

/* p2_6.c: Toggling all bits of Port A at 16 Hz (25 ms on, 25 ms off) */

#include "TM4C123GH6PM.h"

void delayMs(int n);

int main(void)
{
    /* enable clock to GPIOA at clock gating control register */
    SYSCTL->RCGCGPIO |= 1;
    /* enable the GPIOA pins as output */
    GPIOA->DIR = 0xFF;
    /* enable the GPIOA pins for digital function */
    GPIOA->DEN = 0xFF;

    while(1)
    {
        GPIOA->DATA = 0xFF; /* turn on all the pins of Port A */
        delayMs(25);

        GPIOA->DATA = 0;     /* turn off all the pins of Port A */

```

```

        delayMs(25);
    }

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## CPU clock frequency and time delay

Many microcontrollers have at least three clock sources fed to the CPU.

- 1) The on-chip RC oscillator circuit. This is the least precise clock source for the CPU. But it does not require an external oscillator.
- 2) The externally connected crystal (XTAL) oscillator. It offers the most precise clock but at high frequencies, such as above 100MHz, crystals are expensive.
- 3) PLL (phase lock loop). A compromise between precision and economy is to use an inexpensive low frequency crystal oscillator along with the on-chip PLL circuitry to generate a high frequency clock source for the CPU. This option is widely used for systems with CPU frequency of 20MHz and higher. Another added benefit of using the PLL is that the clock frequency is programmable. You may run high clock frequency for CPU intensive tasks and slow down the clock in other times to conserve energy.

TI Tiva LaunchPad is connected externally to a 16MHz XTAL oscillator and one can program its TM4C123GH6PM chip clock generator to implement options 2 and 3. The TM4C123GH6PM chip on TI Tiva LaunchPad also has a 16MHz on-chip RC circuitry as default.

## Measuring time delay in a C program loop

One simple way of creating a time delay is using a **for** loop in C language. The size of time delay loop for a given system is function of two factors: a) the CPU frequency and b) the compiler. It must be noted that a time delay C loop measured using a given compiler (e.g. Keil) may not give the same result if a different compiler such as Code Composer or IAR is used. Regardless of clock source to CPU and the C compiler used, always use oscilloscope to measure the size of time delay loop for a given system with a given compiler and compiler option setting. Measure the time delay in Programs 2-1 and Program 2-2 using an oscilloscope.

## Reading a switch in TI Tiva LaunchPad

In the TI Tiva LaunchPad there are two USB ports designated as Debug and Device. Both of them may be used to power the board. The Debug port is used to download and debug (trace) the program. There are also four switches on the four corners of the board. They are:

- 1) a slide switch by the Debug USB port is used to select the power source from either one of the USB ports,
- 2) a push-button switch designated as Reset,
- 3) SW1 push-button switch is connected to PF0 pin,
- 4) SW2 push-button switch is connected to PF4 pin.

These push-button switches are also called *momentary* switch. See Figure 2-12.

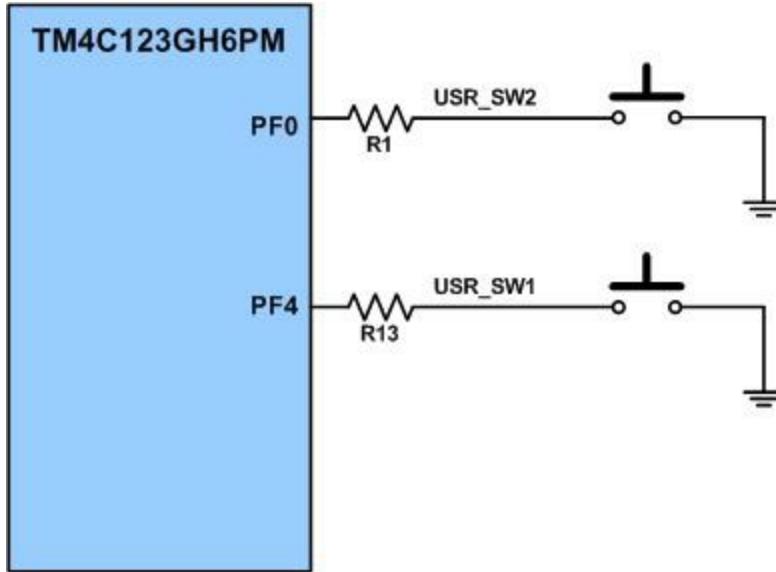


Figure 2-12: Push-button Switches Connected to the Microcontroller in the Tiva LaunchPad Board

Notice from the schematic, there is no pull-up resistor connected to SW1 or SW2. The PF0 and PF4 pins are connected to the push-button switches SW1 and SW2 directly (Actually they are connected through zero Ohm resistors so that they may be removed if other functions for these two pins are desired). To use the SW1 and SW2 we need to enable the internal pull-up resistor for PF0 and PF4 pins. The following shows the register for PUR (pull-up resistor). See Example 2-4.

GPIOPUR:	D31	.....	D8 D7 D6 D5 D4 D3 D2 D1 D0	0x510
		Reserved	PUE	

Note: D0 to D7 are used to enable/disable the pull-up resistor for pins 0 to 7 of the port.

- 1: Enable pull-up  
0: Disable pull-up

Figure 2-13: GPIOPUR (GPOIO Pull-Up Register)

### Example 2-4

Find the physical address of the GPIOPUR register for PORTF if the Base address of the PORTF is 0x40025000.

**Solution:**

Since the offset for GPIOPUR is 0x510, the physical address of the GPIOPUR for PORTF is 0x40025000+0x510=0x40025510.

To read SW1 and display it on the green LED, the following steps must be taken.

- 1) enable the clock to PORTF,
- 2) set the Direction register PF4 as input, and PF3 as output
- 3) enable the digital I/O feature of PORTF,
- 4) enable the pull up resistor option in PUR register since the switch circuit does not have pull-up resistor,
- 5) read SW1 on PORTF,
- 6) invert the value since the switch is active low and the LED is active high,
- 7) shift right the switch bit (PF4) to green LED bit(PF3) of the value,
- 8) write the value to green LED of PORTF,
- 9) Repeat steps 5 to 8.

See Programs 2-7.

#### Program 2-7: Reading SW1 and displaying it on the green LED

```
/* p2_7.c: Read a switch and write it to the LED */

/* This program reads SW1 of Tiva LaunchPad and writes the inverse of the value to the
green LED. SW1 is low when pressed (Normally High). LED is on when high. */

#include "TM4C123GH6PM.h"

int main(void)
{
    unsigned int value;
    SYSCTL->RCGCGPIO |= 0x20;      /* enable clock to GPIOF */
    GPIOF->DIR = 0x08;              /* set PORTF3 pin as output (LED) pin */
                                    /* and PORTF4 as input, SW1 is on PORTF4 */
    GPIOF->DEN = 0x18;              /* set PORTF pins 4-3 as digital pins */
    GPIOF->PUR = 0x10;              /* enable pull up for pin 4 */

    while(1)
    {
        value = GPIOF->DATA;        /* read data from PORTF */
        value = ~value;              /* switch is low active; LED is high active */
        value = value >> 1;          /* shift it right to display on green LED */
        GPIOF->DATA = value;         /* put it on the green LED */
    }
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */

void SystemInit(void)
```

```

    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}

```

To read SW2 and display it on the red LED, the program is similar to Program 2-7 except extra steps needed to take care of PORTF0 as described below. The SW2 is connected to PORTF0 pin, which is shared with NMI (non-maskable interrupt). To prevent accidental write to configuration registers and thus disables NMI, the configuration register bits for PORTF0 are normally locked. They may be unlocked by writing a passcode value of `0x4C4F434B` to the LOCK Register followed by setting bit 0 of the Commit Register.

*At the time this is written, the Commit Register of GPIOF is defined as read-only in the header file. To be able to modify the Commit Register, you need to change line 253 of TM4C123GH6PM.h from*

`_I uint32_t CR;`

to

`_IO uint32_t CR;`

### Program 2-8: Reading SW2 and displaying it on the red LED

```

/* p2_8.c: Read a switch and write it to the LED */

/* This program reads SW2 of Tiva LaunchPad and write the inverse of the value to the
red LED. SW2 is low when pressed. LED is on when high. */

/* SW2 is connected to PORTF0, which is an NMI pin. */
/* In order to use this pin for any function other than NMI, the pin needs be unlocked
first. */

#include "TM4C123GH6PM.h"

int main(void)
{
    unsigned int value;
    SYSCTL->RCGCGPIO |= 0x20; /* enable clock to GPIOF */

    GPIOF->LOCK = 0x4C4F434B; /* unlock commit register */
    GPIOF->CR = 0x01; /* make PORTF0 configurable */

    GPIOF->DIR = 0x02; /* set PORTF1 pin as output (LED) pin */
    /* and PORTF0 as input, SW2 is on PORTF0 */
    GPIOF->DEN = 0x03; /* set PORTF pins 1-0 as digital pins */
    GPIOF->PUR = 0x01; /* enable pull up for pin 0 */

    while(1)
    {
        value = GPIOF->DATA; /* read data from PORTF */
        value = ~value; /* switch is low active; LED is high active */
        value = value << 1; /* shift it left to display on red LED */
    }
}

```

```
    GPIOF->DATA = value;      /* put it on red LED */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}
```

## Review Questions

1. TM4C123GH6PM has \_\_\_\_\_ GPIO ports.
2. True or false. Every ARM microcontroller must have minimum of 3 memory spaces of Flash (for code), SRAM (for data), and I/O.
3. Port A in TM4C123GH6PM has \_\_\_\_ pins.
4. Give the address location assigned to Data register of PORTA if the Base address of the PORTA is 0x40004000.
5. Give the address location assigned to Direction register of PORTB if the Base address of the PORTB is 0x40005000.

## Section 2.3: Seven-segment LED interfacing and programming

Another popular output display is seven-segment LED. The 7-seg LED can have common anode or common cathode. With common anode, the anode of the LED is driven by the positive supply voltage and the microcontroller drives the individual cathodes LOW for current to flow through LEDs to light up. In this configuration, the sink current capability of the microcontroller is critical. With common cathode, the cathode of the LED is grounded and microcontroller drives the individual anodes HIGH to light up the LED. In this configuration, the microcontroller pins must provide sufficient source current for each LED segment. In either configurations, if the microcontroller does not have sufficient drive or sink current capacity, we must add a buffer between the 7-seg LED and the microcontroller. The buffer for the 7-seg LED can be an IC chip or transistors.

The seven segments of LED are designated as a, b, c, d, e, f, and g. See Figure 2-14.

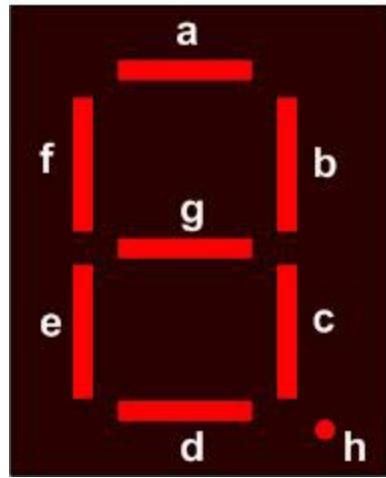


Figure 2-14: Seven-Segment

A byte of data should be sufficient to drive all of the segments. In the example below, segment a is assigned to bit D0, segment b is assigned to bit D1, and so on as shown below:

D7	D6	D5	D4	D3	D2	D1	D0
.	g	f	e	d	c	b	a

Table 2-3: Assignments of port pins to each segments of a 7-seg LED

The D7 bit is assigned to decimal point. One can create the following patterns for numbers 0 to 9 for the common cathode configuration:

Num	D7	D6	D5	D4	D3	D2	D1	D0	Hex value
0	0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	0	1	1	0	0x06
2	0	1	0	1	1	0	1	1	0x5B
3	0	1	0	0	1	1	1	1	0x4F
4	0	1	1	0	0	1	1	0	0x66
5	0	1	1	0	1	1	0	1	0x6D
6	0	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	0	1	1	1	0x07

8	0	1	1	1	1	1	1	1	0x7F
9	0	1	1	0	1	1	1	1	0x6F

Table 2-4: Segment patterns for the 10 decimal digits for a common cathode 7-seg LED

In Figures 2-15 and 2-16 the connection for 2-digit 7-seg LED and TI Tiva LaunchPad is shown. The program 2-9 shows the code.

Notice since the same segment for both digit 1 and digit 2 are connected to the same I/O port pin, the common cathode of each digit must be driven separately so that only one digit is on at a time. The two digits are turned on alternatively. For example, if we want to display number 14 on the 7-seg LED, the following steps should be used:

- 1) Configure Port B as output port to drive the segments,
- 2) Configure Port A as output port to select the digit,
- 3) Write the pattern of numeral 1 from Table 2-4 to Port B,
- 4) Write one bit of Port A to activate the tens digit,
- 5) Delay for some time,
- 6) Write the pattern of numeral 4 from Table 2-4 to Port B,
- 7) Write one bit of Port A to activate the ones digit,
- 8) Delay for some time,
- 9) Repeat from step 3 to 8.

At low frequency of alternating digits, the display will appear to be flickering. To eliminate the flickering display, each digit should be turned on and off at least 60 times each second. From the example above, the delay for steps 5 and 8 should be 8 milliseconds or less.

$$1 \text{ second} / 60 / 2 = 8 \text{ millisecond}$$

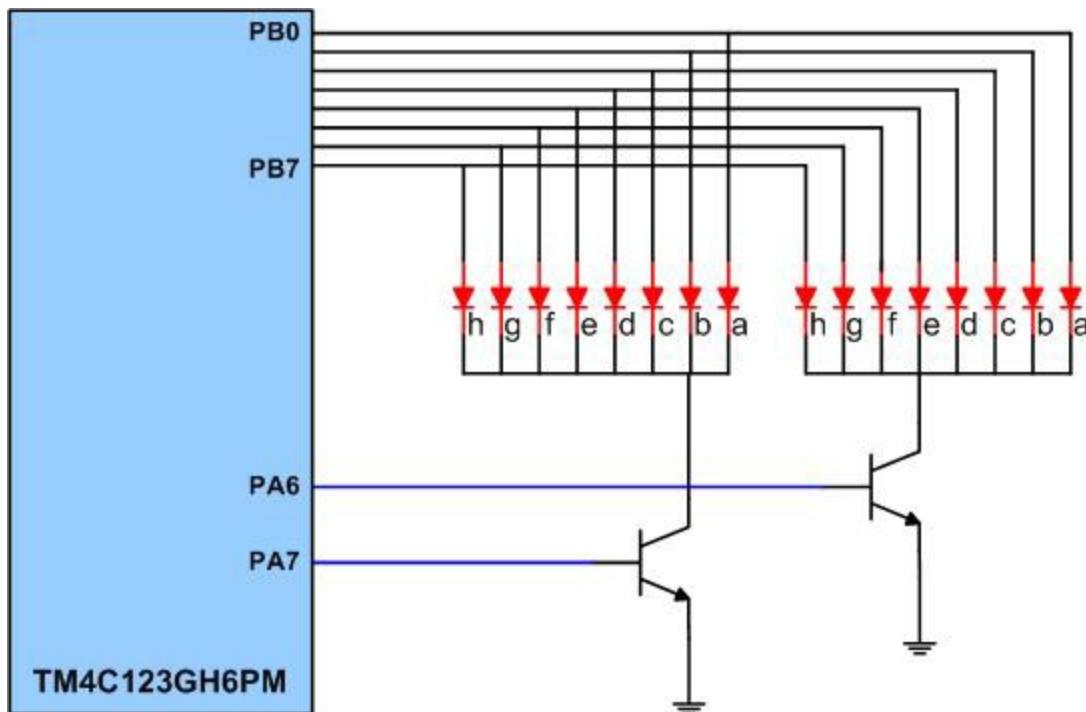
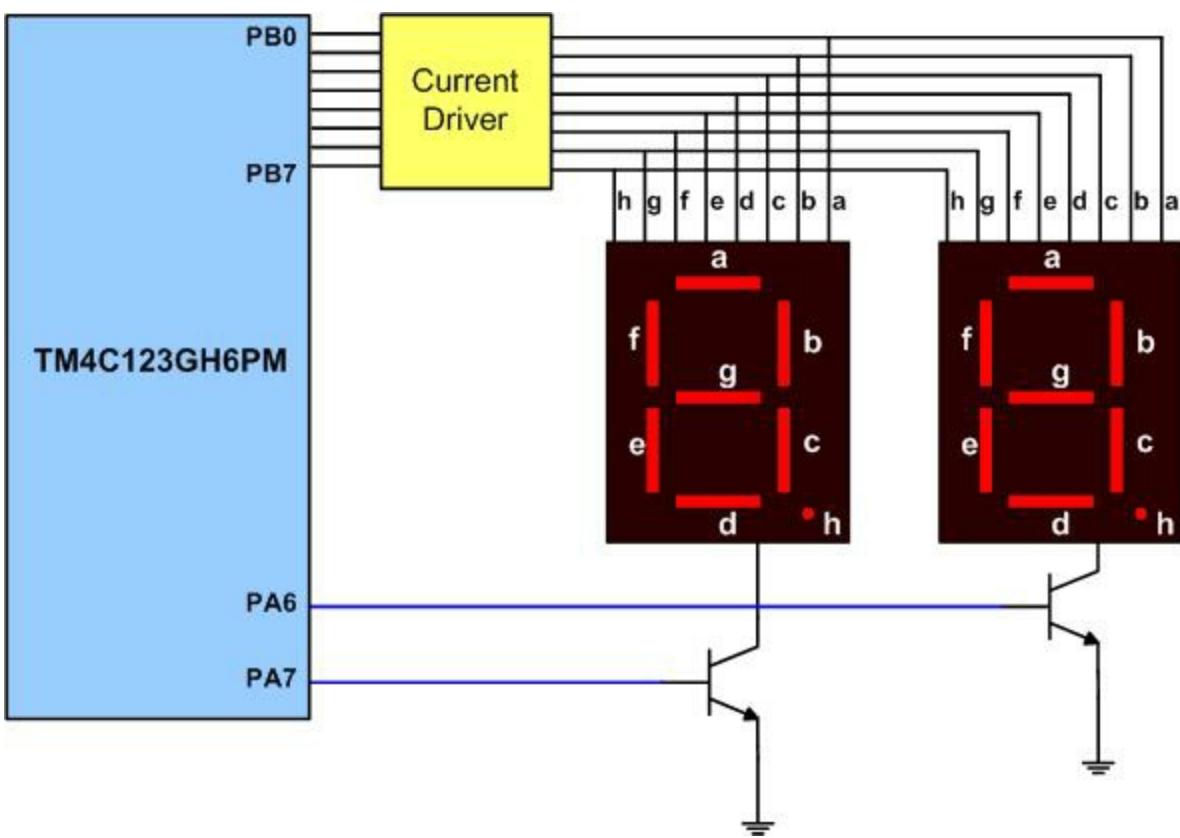


Figure 2-15: Microcontroller Connection to 7-segment LED



**Figure 2-16: Microcontroller Connection to 7-segment LED with Buffer Driver**

See Program 2-9.

#### Program 2-9: displaying "14" on 2-digit 7-seg LED display

```
/* p2_9.c: Display number 14 on a 2-digit 7-segment LED. */

#include "TM4C123GH6PM.h"

void delayMs(int n);

int main(void)
{
    unsigned char digitPattern[] =
        {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};
    /*from Table 2-4 */

    SYSCTL->RCGCGPIO |= 0x01; /* enable clock to GPIOA */
    SYSCTL->RCGCGPIO |= 0x02; /* enable clock to GPIOB */
    GPIOB->DIR = 0xFF;          /* set all PORTB pins as output to drive segments */
    GPIOB->DEN = 0xFF;          /* set all PORTB pins as digital pins */
    GPIOA->DIR = 0xC0;          /* set PORTA pin 7-6 as output to select digit */
    GPIOA->DEN = 0xC0;          /* set PORTA pin 7-6 as digital pins */

    for(;;)
    {
        GPIOB->DATA = digitPattern[1]; /* drive pattern of 1 on the segments */
        GPIOA->DATA = 0x80;             /* select left digit */
        delayMs(8);                   /* delay 8 ms will result in 16 ms per loop */
                                       /* or 62.5 Hz */
        GPIOB->DATA = digitPattern[4]; /* drive pattern of 4 on the segments */
    }
}
```

```

        GPIOA->DATA = 0x40;           /* select right digit */
        delayMs(8);
    }

}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}

```

Notice in Figure 2-16, a single pin is used to select each digit. That means if we want 4 digits we must use a total of 12 pins. That is 8 pins for the segments a through g, decimal point, and 4 pins to select each digit. This might not be feasible in applications in which we have a limited number of microcontroller pins to spare. One solution is to use a decoder for the digit selection. For example a 74LS138 decoder can be used for up to 8-digit 7-seg LED system with three select pins. Another approach is to use a 7-segment LED driver chip such as MAX 7221, which only uses two interface pins. An additional advantage of MAX7221 is that the refreshing of the segments is handled by the driver chip itself so the microcontroller does not have to spend time refreshing the display and can concentrate on other important tasks. The MAX7221 is an I<sup>2</sup>C device and the vast majority of microcontrollers come with on-chip I<sup>2</sup>C serial communication feature, which we will discuss in a separate chapter.

## Review Questions

1. In a common cathode 7-seg LED connection, to turn on a segment the microcontroller drives it (high , low).
2. True or false. In connecting the 7-seg LED directly to microcontroller, the refreshing of digits is done by microcontroller itself.
3. What is the disadvantage of letting microcontroller to do the refreshing of 7-seg LEDs?
4. List two advantages of using an IC chip such as MAX7221 chip?
5. In an application, we need 8 digits of 7-seg LED. How many pins of microcontroller will be used if we connect microcontroller to 7-seg directly (similar to Figure 2-16)? How about if we use 3-8 decoder for digit selection?

## Answer to Review Questions

## Section 2-1

1. 256KB
2. 32KB
3. Program code
4. Data
5. 0x00000000 to 0x0003FFFF
6. 0x40000000 to 0x400FFFFFF

## Section 2-2

1. 6
2. True
3. 8
4.  $0x40004000 + 0x00 = 0x40004000.$
5.  $0x40005000 + 0x0400 = 0x40005400.$

## Section 2-3

1. High
2. True
3. The time and pins of microcontroller is wasted to scan the 7-segments.
4. (1) It refreshes the 7-segments, (2) it is connected to the microcontroller using I<sup>2</sup>C which uses just 2 pins of the microcontroller.
5. 8 pins for data and 8 pins for selector; 8 pins for data and 3 pins for selector.



## Chapter 3: LCD and Keyboard Interfacing

In this chapter we show interfacing to two real-world devices: LCD and Keyboard. They are widely used in different embedded systems.

## Section 3.1: Interfacing to an LCD

This section describes the operation modes of the LCDs, then describes how to program and interface an LCD to the Tiva LaunchPad.

### LCD operation

In recent years the LCD is replacing LEDs (seven-segment LEDs or other multi-segment LEDs). This is due to the following reasons:

1. The declining prices of LCDs.
2. The ability to display numbers, characters, and graphics. This is in contrast to LEDs, which are limited to numbers and a few characters. (The new OLED panels are relatively much more expensive except the very small ones. But their prices are dropping. The interface and programming to OLED are similar to graphic LCD.)
3. Incorporation of the refreshing controller into the LCD itself, thereby relieving the CPU of the task of refreshing the LCD.
4. Ease of programming for both characters and graphics.
5. The extremely low power consumption of LCD (when backlight is not used).

### LCD module pin descriptions

For many years, the use of Hitachi HD44780 LCD controller dominated the character LCD modules. Even today, most of the character LCD modules still use HD44780 or a variation of it. The HD44780 controller has a 14 pin interface for the microprocessor. We will discuss this 14 pin interface in this section. The function of each pin is given in Table 3-1. Figure 3-1 shows the pin positions for various LCD modules.

Pin	Symbol	I/O	Description
1	VSS	--	Ground
2	VCC	--	+5V power supply
3	VEE	--	Power supply to control contrast
4	RS	I	RS = 0 to select command register, RS = 1 to select data register
5	R/W	I	R/W = 0 for write, R/W = 1 for read
6	E	I	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 4/8-bit data bus
12	DB5	I/O	The 4/8-bit data bus
13	DB6	I/O	The 4/8-bit data bus
14	DB7	I/O	The 4/8-bit data bus

Table 3-1: Pin Descriptions for LCD

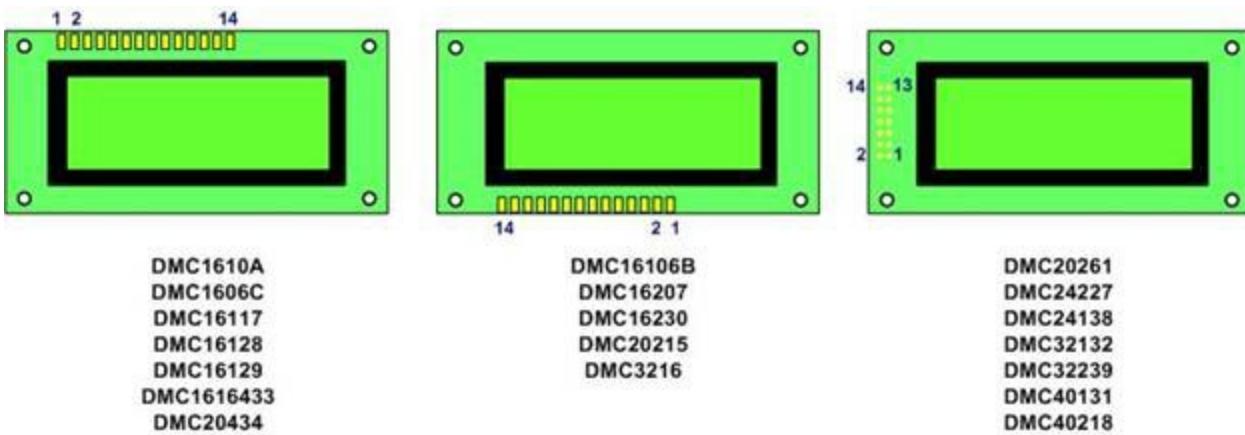


Figure 3-1: Pin Positions for Various LCDs from Optrex

**VCC, VSS, and VEE:** While VCC and VSS provide +5V power supply and ground, respectively, VEE is used for controlling the LCD contrast.

**RS, register select:** There are two registers inside the LCD and the RS pin is used for their selection as follows. If RS = 0, the instruction command code register is selected, allowing the user to send a command such as clear display, cursor at home, and so on (or query the busy status bit of the controller). If RS = 1, the data register is selected, allowing the user to send data to be displayed on the LCD (or to retrieve data from the LCD controller).

**R/W, read/write:** R/W input allows the user to write information into the LCD controller or read information from it. R/W = 1 when reading and R/W = 0 when writing.

**E, enable:** The enable pin is used by the LCD to latch information presented to its data pins. When data is supplied to data pins, a High-to-Low pulse must be applied to this pin in order for the LCD to latch in the data present at the data pins. This pulse must be a minimum of 230 ns wide, according to Hitachi datasheet.

**D0–D7:** The 8-bit data pins are used to send information to the LCD or read the contents of the LCD's internal registers. The LCD controller is capable of operating with 4-bit data and only D4-D7 are used. We will discuss this in more details later.

To display letters and numbers, we send ASCII codes for the letters A–Z, a–z, numbers 0–9, and the punctuation marks to these pins while making RS = 1.

There are also instruction command codes that can be sent to the LCD in order to clear the display, force the cursor to the home position, or blink the cursor. Table 3-2 lists some commonly used command codes. For detailed command codes, see Table 3-4.

Code (Hex)	Command to LCD Instruction Register
1	Clear display screen
2	Return cursor home
6	Increment cursor (shift cursor to right)
F	Display on, cursor blinking
80	Force cursor to beginning of 1st line
C0	Force cursor to beginning of 2nd line
38	2 lines and 5x7 character (8-bit data, D0 to D7)

Table 3-2: Some commonly used LCD Command Codes

## Sending commands to LCDs

To send any of the commands to the LCD, make pin RS = 0 and send a High-to-Low pulse on the E pin to enable the internal latch of the LCD. The connection of an LCD to the microcontroller is shown in Figure 3-2.

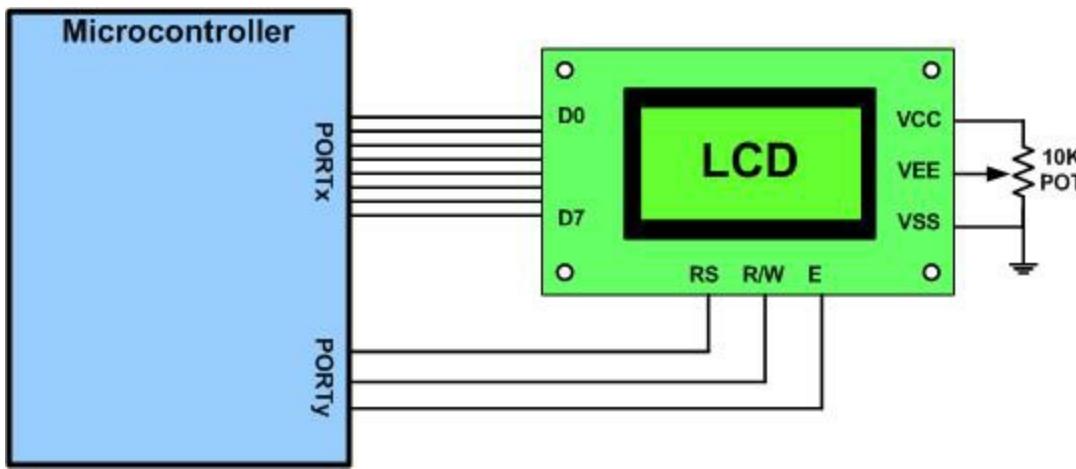


Figure 3-2: LCD Connection to Microcontroller

Notice the following for the connection in Figure 3-2:

1. The LCD's data pins are connected to PORTx of the microcontroller.
2. The LCD's RS pin is connected to Pin 5 of PORTy of the microcontroller.
3. The LCD's R/W pin is connected to Pin 6 of PORTy of the microcontroller.
4. The LCD's E pin is connected to Pin 7 of PORTy of the microcontroller.
5. Both Ports x and y are configured as output ports.

## Sending data to the LCD

In order to send data to the LCD to be displayed, we must set pin RS = 1, and also send a High-to-Low pulse to the E pin to enable the internal latch of the LCD.

Because of the extremely low power feature of the LCD controller, it runs much slower than the microcontroller. The first two commands in Table 3-2 take up to 1.64 ms to execute and all the other commands and data take up to 40 us. (At the highest clock speed, TM4C123GH6PM can execute more than 3,000 instructions in 40 us.) After one command or data is written to the LCD controller, one must wait until the LCD controller is ready before issuing the next command/data otherwise the second command/data will be ignored. An easy way (not as efficient though) is to delay the microcontroller for the maximal time it may take for the previous command. We will use this method in the following examples. All the examples in this chapter follow the original HD44780 datasheet (see Table 3-4). You may have to adjust the delay time for the LCD module you use.

**Program 3-1: This program displays a message on the LCD.**

```
/* p3_1.c: Initialize and display "Hello" on the LCD using 8-bit data mode. */
/* Data pins use Port B, control pins use Port A */

/* This program strictly follows HD44780 datasheet for timing. You may want to adjust
the amount of delay for your LCD controller. */

#include "TM4C123GH6PM.h"

#define LCD_DATA GPIOB
#define LCD_CTRL GPIOA
#define RS 0x20 /* PORTA BIT5 mask */
#define RW 0x40 /* PORTA BIT6 mask */
#define EN 0x80 /* PORTA BIT7 mask */

void delayMs(int n);
void delayUs(int n);
void LCD_command(unsigned char command);
void LCD_data(unsigned char data);
void LCD_init(void);

int main(void)
{
    LCD_init();
    for(;;)
    {
        LCD_command(1);          /* clear display */
        LCD_command(0x80);       /* lcd cursor location */
        delayMs(500);
        LCD_data('H');
        LCD_data('e');
        LCD_data('l');
        LCD_data('l');
        LCD_data('o');
        delayMs(500);
    }
}

void LCD_init(void)
{
    SYSCTL->RCGCGPIO |= 0x01; /* enable clock to GPIOA */
    SYSCTL->RCGCGPIO |= 0x02; /* enable clock to GPIOB */

    LCD_CTRL->DIR |= 0xE0;      /* set PORTA pin 7-5 as output for control */
    LCD_CTRL->DEN |= 0xE0;      /* set PORTA pin 7-5 as digital pins */
    LCD_DATA->DIR = 0xFF;       /* set all PORTB pins as output for data */
    LCD_DATA->DEN = 0xFF;       /* set all PORTB pins as digital pins */

    delayMs(20);                /* initialization sequence */
    LCD_command(0x30);
    delayMs(5);
    LCD_command(0x30);
    delayUs(100);
    LCD_command(0x30);

    LCD_command(0x38);          /* set 8-bit data, 2-line, 5x7 font */
    LCD_command(0x06);          /* move cursor right */
    LCD_command(0x01);          /* clear screen, move cursor to home */
    LCD_command(0x0F);          /* turn on display, cursor blinking */
}
```

```

}

void LCD_command(unsigned char command)
{
    LCD_CTRL->DATA = 0;      /* RS = 0, R/W = 0 */
    LCD_DATA->DATA = command;
    LCD_CTRL->DATA = EN;     /* pulse E */
    delayUs(0);
    LCD_CTRL->DATA = 0;
    if (command < 4)
        delayMs(2);          /* command 1 and 2 needs up to 1.64ms */
    else
        delayUs(40);         /* all others 40 us */
}

void LCD_data(unsigned char data)
{
    LCD_CTRL->DATA = RS;    /* RS = 1, R/W = 0 */
    LCD_DATA->DATA = data;
    LCD_CTRL->DATA = EN | RS; /* pulse E */
    delayUs(0);
    LCD_CTRL->DATA = 0;
    delayUs(40);
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* delay n microseconds (16 MHz CPU clock) */
void delayUs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3; j++)
            {} /* do nothing for 1 us */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## Checking LCD busy flag

The above programs used a time delay before issuing the next data or command. This allows the LCD a sufficient amount of time to get ready to accept the next data. However, the

LCD has a busy flag. We can monitor the busy flag and issue data when it is ready. This will speed up the process. To check the busy flag, we must read the command register ( $R/W = 1$ ,  $RS = 0$ ). The busy flag is the D7 bit of that register. Therefore, if  $R/W = 1$ ,  $RS = 0$ . When  $D7 = 1$  (busy flag = 1), the LCD is busy taking care of internal operations and will not accept any new information. When  $D7 = 0$ , the LCD is ready to receive new information.

Doing so requires switching the direction of the port connected to the data bus to input mode when polling the status register then switch the port direction back to output mode to send the next command. If the port direction is incorrect, it may damage the microcontroller or the LCD module. We leave this option to be explored by the reader.

## LCD 4-bit Option

To save the number of microcontroller pins used by LCD interfacing, we can use the 4-bit data option instead of 8-bit. In the 4-bit data option, we only need to connect D7-D4 to microcontroller. Together with the three control lines, the interface between the microcontroller and the LCD module will fit in a single 8-bit port. See Figure 3-3.

With 4-bit data option, the microcontroller needs to issue commands to put the LCD controller in 4-bit mode during initialization. This is done with command 0x20 and 0x28 in Program 3-2. After that, every command or data needs to be broken down to two 4-bit operations, upper nibble first. In Program 3-2, the upper nibble is extracted using **command & 0xF0** and the lower nibble is shifted into place by **command << 4**.

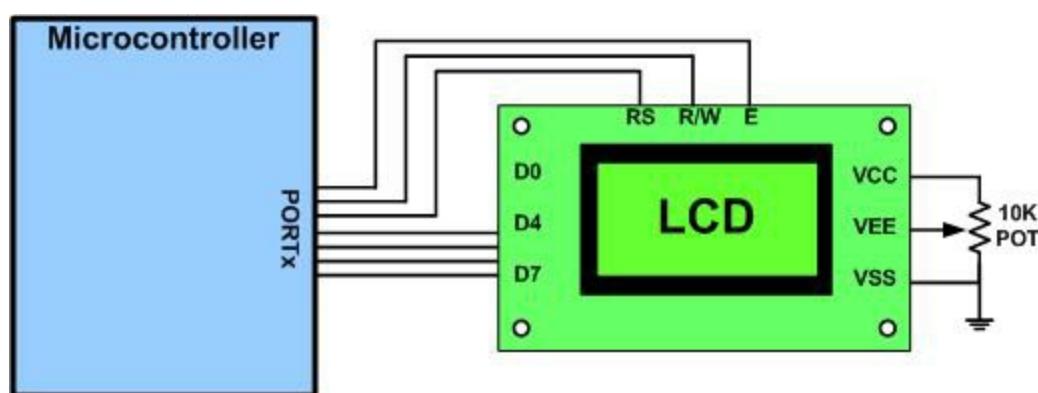


Figure 3-3: LCD Connection for 4-bit Data

**Program 3-2:** This program uses the 4-bit data option to show a message on the LCD.

```
/* p3_2.c: Initialize and display "Hello" on the LCD sing 4-bit data mode. All
interface uses Port B */

/* This program strictly follows HD44780 datasheet for timing. You may want to adjust
the amount of delay for your LCD controller. */

#include "TM4C123GH6PM.h"

#define LCD_PORT GPIOB
#define RS 1      /* BIT0 mask */
#define RW 2      /* BIT1 mask */
#define EN 4      /* BIT2 mask */
```

```

void delayMs(int n);
void delayUs(int n);
void LCD_nibble_write(unsigned char data, unsigned char control);
void LCD_command(unsigned char command);
void LCD_data(unsigned char data);
void LCD_init(void);

int main(void)
{
    LCD_init();
    for(;;)
    {
        LCD_command(1);           /* clear display */
        LCD_command(0x80);       /* LCD cursor location */

        delayMs(500);
        LCD_data('H');
        LCD_data('e');
        LCD_data('l');
        LCD_data('l');
        LCD_data('o');
        delayMs(500);
    }
}

void LCD_init(void)
{
    SYSCTL->RCGCGPIO |= 0x02;      /* enable clock to GPIOB */
    LCD_PORT->DIR = 0xFF;          /* set all PORTB pins as output */
    LCD_PORT->DEN = 0xFF;          /* set all PORTB pins as digital pins */

    delayMs(20);                  /* initialization sequence */
    LCD_nibble_write(0x30, 0);
    delayMs(5);
    LCD_nibble_write(0x30, 0);
    delayUs(100);
    LCD_nibble_write(0x30, 0);
    delayUs(40);

    LCD_nibble_write(0x20, 0);     /* use 4-bit data mode */
    delayUs(40);
    LCD_command(0x28);           /* set 4-bit data, 2-line, 5x7 font */
    LCD_command(0x06);           /* move cursor right */
    LCD_command(0x01);           /* clear screen, move cursor to home */
    LCD_command(0x0F);           /* turn on display, cursor blinking */
}

void LCD_nibble_write(unsigned char data, unsigned char control)
{
    data &= 0xF0;                /* clear lower nibble for control */
    control &= 0x0F;              /* clear upper nibble for data */
    LCD_PORT->DATA = data | control;      /* RS = 0, R/W = 0 */
    LCD_PORT->DATA = data | control | EN;  /* pulse E */
    delayUs(0);
    LCD_PORT->DATA = data;
    LCD_PORT->DATA = 0;
}

```

```

void LCD_command(unsigned char command)
{
    LCD_nibble_write(command & 0xF0, 0); /* upper nibble first */
    LCD_nibble_write(command << 4, 0); /* then lower nibble */

    if (command < 4)
        delayMs(2); /* commands 1 and 2 need up to 1.64ms */
    else
        delayUs(40); /* all others 40 us */
}

void LCD_data(unsigned char data)
{
    LCD_nibble_write(data & 0xF0, RS); /* upper nibble first */
    LCD_nibble_write(data << 4, RS); /* then lower nibble */

    delayUs(40);
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* delay n microseconds (16 MHz CPU clock) */
void delayUs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3; j++)
            {} /* do nothing for 1 us */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## LCD cursor position

In the LCD, one can move the cursor to any location in the display by issuing an address command. The next character sent will appear at the cursor position. For the two-line LCD, the address command for the first location of line 1 is 0x80, and for line 2 it is 0xC0. The following shows address locations and how they are accessed:

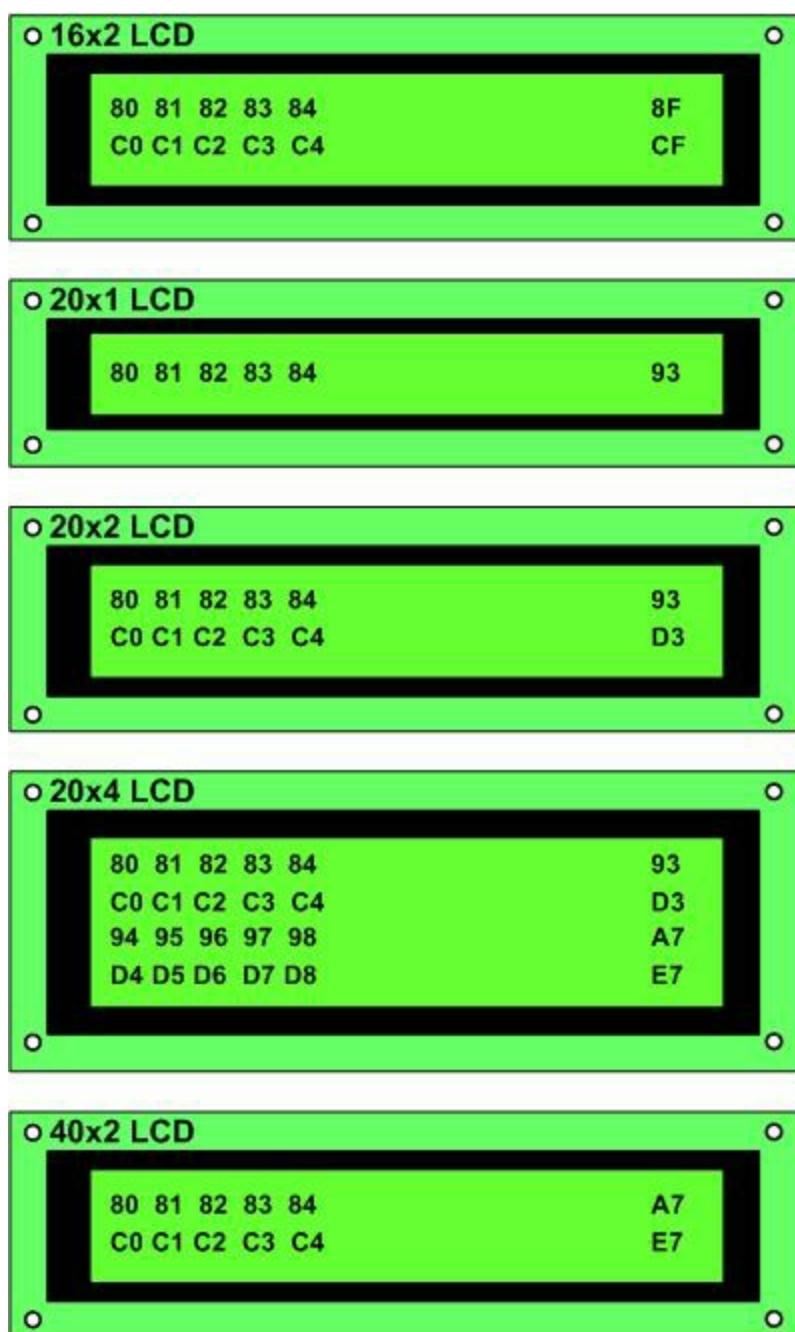
RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A6	A5	A4	A3	A2	A1	A0

where  $A_6A_5A_4A_3A_2A_1A_0 = 0000000$  to  $0100111$  for line 1 and  $A_6A_5A_4A_3A_2A_1A_0 = 1000000$  to  $1100111$  for line 2. See Table 3-3.

Table 3-3: LCD Addressing Commands

	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
<b>Line 1 (min)</b>	1	0	0	0	0	0	0	0
<b>Line 1 (max)</b>	1	0	1	0	0	1	1	1
<b>Line 2 (min)</b>	1	1	0	0	0	0	0	0
<b>Line 2 (max)</b>	1	1	1	0	0	1	1	1

The upper address range can go as high as  $0100111$  for the 40-character-wide LCD while for the 20-character-wide LCD the address of the visible positions goes up to  $010011$  (19 decimal = 10011 binary). Notice that the upper range  $0100111$  (binary) = 39 decimal, which corresponds to locations 0 to 39 for the LCDs of  $40 \times 2$  size. Figure 3-4 shows the addresses of cursor positions for various sizes of LCDs. All the addresses are in hex. Notice the starting addresses for four line LCD are not in sequential order.



**Figure 3-4: Cursor Addresses for Some LCDs**

As an example of setting the cursor at the fourth location of line 1 we have the following:

```
LCD_command(0x83);
```

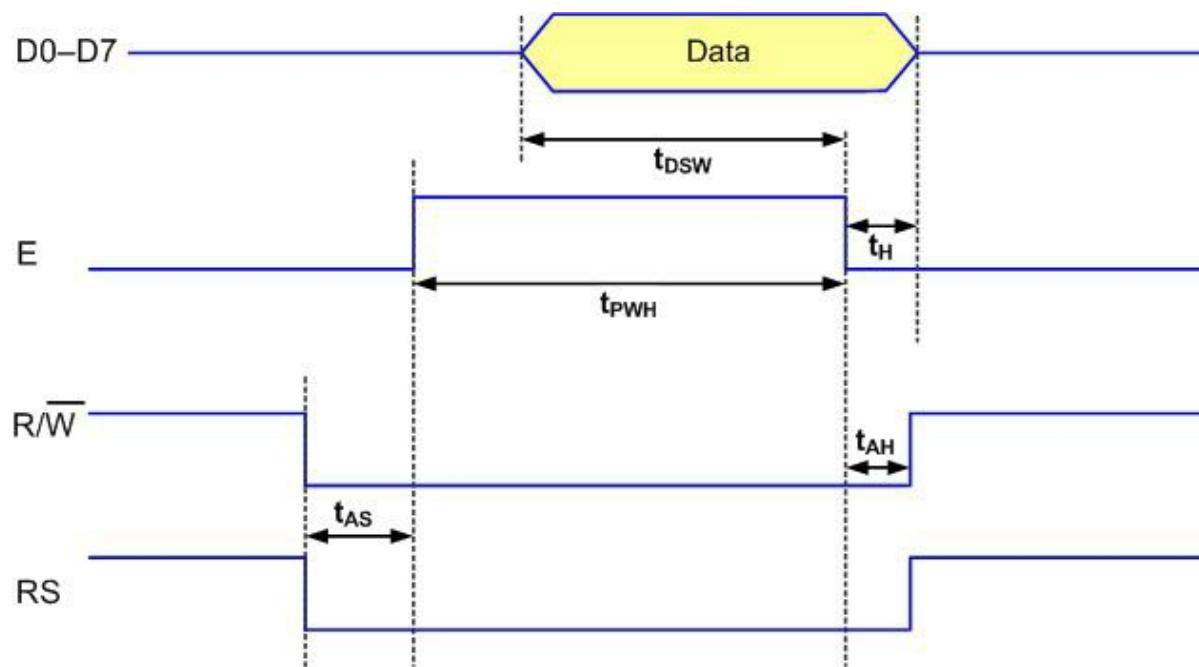
and for the sixth location of the second line we have:

```
LCD_command(0xC5);
```

Notice that the cursor location addresses are in hex and starting at 0 as the first location.

## LCD timing and data sheet

Figures 3-5 and 3-6 show timing diagrams for LCD write and read timing, respectively.



$t_{PWH} = \text{Enable pulse width} = 230 \text{ ns (minimum)}$

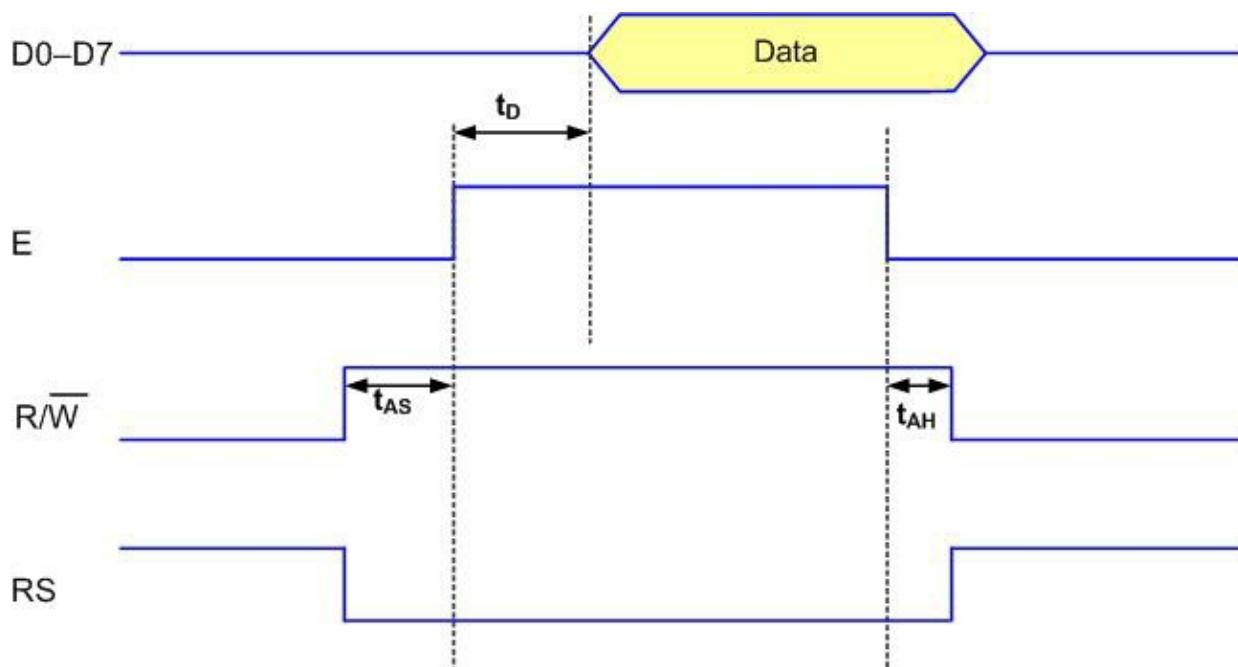
$t_{DSW} = \text{Data setup time} = 80 \text{ ns (minimum)}$

$t_H = \text{Data hold time} = 10 \text{ ns (minimum)}$

$t_{AS} = \text{Setup time prior to } E \text{ (going high) for both RS and R/W} = 40 \text{ ns (minimum)}$

$t_{AH} = \text{Hold time after } E \text{ has come down for both RS and R/W} = 10 \text{ ns (minimum)}$

Figure 3-5: LCD Write Timing



$t_D = \text{Data output delay time}$

$t_{AS} = \text{Setup time prior to } E \text{ (going high) for both RS and R/W} = 40 \text{ ns (minimum)}$

$t_{AH} = \text{Hold time after } E \text{ has come down for both RS and R/W} = 10 \text{ ns (minimum)}$

Note: Read requires an L-to-H pulse for the E pin.

Figure 3-6: LCD Read Timing

Notice that the write operation happens on the H-to-L transition of the E pin. The microcontroller must have data ready and stable on the data lines before the H-to-L transition of E to satisfy the setup time requirement.

The read operation is activated by the L-to-H pulse of the E pin. After the delay time, the LCD controller will have the data available on the data bus if the R/W line is high. The microcontroller should read the data from the data lines before lowering the E pulse.

Table 3-4 provides a more detailed list of LCD instructions.

Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (Max)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DD RAM address 0 in address counter	1.64 ms
Return Home	0	0	0	0	0	0	0	0	1	-	Sets DD RAM address to 0 as address counter. Also returns display being shifted to original positions. DD RAM contents remain unchanged.	1.64 ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies shift of display. These operations are performed during data write and read.	40µs
Display On/Off Control	0	0	0	0	0	0	1	D	C	B	Sets On/Off of entire display (D), cursor On/Off (C), and blink of cursor position character (B).	40µs
Cursor or Display shift	0	0	0	0	0	1	S/C	R/L	-	-	Moves cursor and shifts display without changing DD RAM contents.	40µs
Function Set	0	0	0	0	1	DL	N	F	-	-	Sets interface data length (DL), number of display lines (L), and character font (F)	40µs
Set CG RAM Address	0	0	0	1	AGC					Sets CG RAM address. CG RAM data is sent and received after this setting.		40µs
Set DD RAM Address	0	0	1	ADD					Sets DD RAM address. DD RAM data is sent and received after this setting.		40µs	
Read Busy Flag & Address	0	1	BF	AC					Reads Busy flag (BF) indicating internal operation is being performed and reads address counter contents.		40µs	
Write Data CG or DD RAM	1	0	Write Data					Writes data into DD or CG RAM.		40µs		
Read Data CG or DD RAM	1	1	Read Data					Reads data from DD or CG RAM.		40µs		

Abbreviations:

DD RAM: Display data RAM

CG RAM: Character generator RAM

AGC: CG RAM address

ADD: DD RAM address, corresponds to cursor address

AC: address counter used for both DD and CG RAM addresses

I/D: 1 = Increment, 0: Decrement

S =1: Accompanies display shift

S/C: 1 = Display shift, 0: Cursor move

R/L: 1: Shift to the right, 0: Shift to the left

DL: 1 = 8 bits, 0 = 4 bits

N: 1 = 2-line, 0 = 1-line

F: 1 = 5 x 10 dots, 0 = 5 x 7 dots

BF: 1 = Internal operation, 0 = Can accept instruction

Table 3-4: List of LCD Instructions

## Review Questions

1. The RS pin is an \_\_\_\_\_ (input, output) pin for the LCD.
2. The E pin is an \_\_\_\_\_ (input, output) pin for the LCD.
3. The E pin requires an \_\_\_\_\_ (H-to-L, L-to-H) transition to latch in information at the data pins of the LCD.
4. For the LCD to recognize information at the data pins as data, RS must be set to \_\_\_\_\_ (high, low).
5. Give the command codes for line 1, first character, and line 2, first character.

## Section 3.2: Interfacing the Keyboard to the CPU

To reduce the microcontroller I/O pin usage, keyboards are organized in a matrix of rows and columns. The CPU accesses both rows and columns through ports; therefore, with two 8-bit ports, an  $8 \times 8$  matrix of 64 keys can be connected to a microprocessor. When a key is pressed, a row and a column make a contact; otherwise, there is no connection between rows and columns. In a PC keyboards, an embedded microcontroller in the keyboard takes care of the hardware and software interfacing of the keyboard. In such systems, it is the function of programs stored in the ROM of the microcontroller to scan the keys continuously, identify which one has been activated, and present it to the main CPU on the motherboard. In this section, we look at the mechanism by which the microprocessor scans and identifies the key. For clarity some examples are provided.

### Scanning and identifying the key

Figure 3-6 shows a  $4 \times 4$  matrix connected to two ports. The rows are connected to an output port and the columns are connected to an input port. All the input pins have pull-up resistor connected. If no key has been pressed, reading the input port will yield 1s for all columns. If all the rows are driven low and a key is pressed, the column of that key will read back a 0 since the key pressed shorted that column to the row that is driven low. It is the function of the microprocessor to scan the keyboard continuously to detect and identify the key pressed. How it is done is explained next.

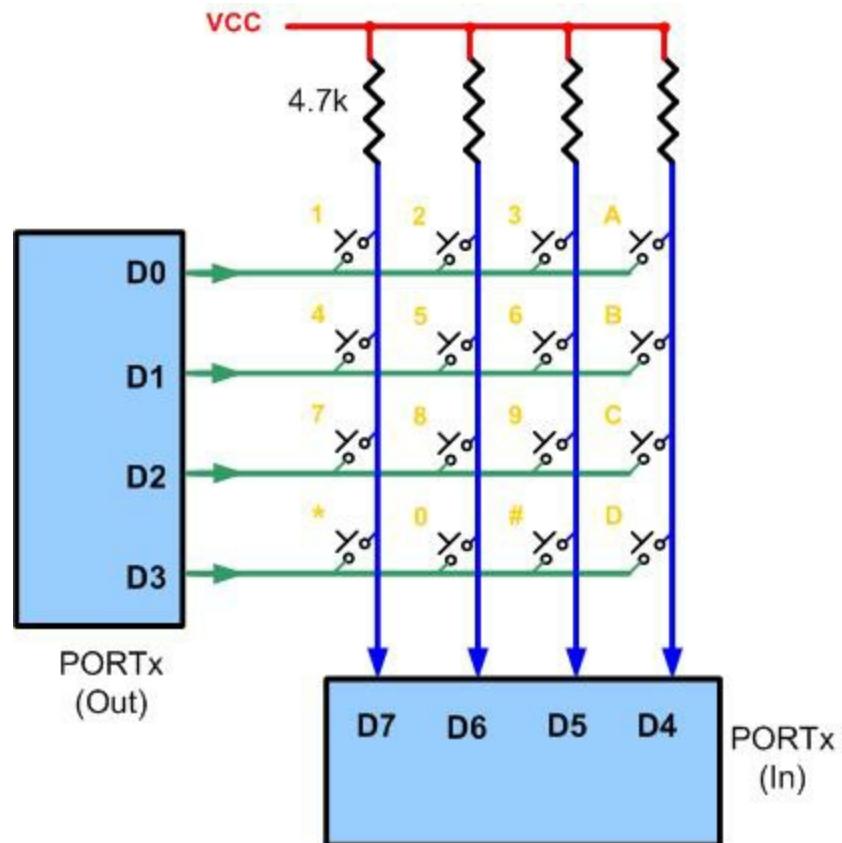


Figure 3-7: Matrix Keyboard Connection to Ports

### Key press detection

To detect the key pressed, the microprocessor drives all rows low, then it reads the columns. If the data read from the columns is  $D7-D4 = 1111$ , no key has been pressed and the

process continues until a key press is detected. However, if one of the column bits has a zero, this means that a key was pressed. For example, if D7-D4= 1101, this means that a key in the D5 column has been pressed.

The following program detects whether any of the keys is pressed.

**Program 3-3: This program displays 'P' on the LCD when a key is pressed. Otherwise, 'R' is displayed.**

```
/* to be used in conjunction with Program 3-2 */

void keypad_init(void);
unsigned char keypad_kbhit(void);

int main(void)
{
    unsigned char key;

    keypad_init();
    LCD_init();

    while(1)
    {
        LCD_command(0x80);          /* LCD cursor location */

        if (keypad_kbhit() != 0) /* if a key is pressed */
            LCD_data('P');      /* display 'P' */
        else
            LCD_data('R');      /* display 'R' */

        delayMs(10);             /* wait for a while */
    }
}

#define KEYPAD_ROW GPIOE
#define KEYPAD_COL GPIOC

/* this function initializes the ports connected to the keypad */
void keypad_init(void)
{
    SYSCTL->RCGCGPIO |= 0x04;    /* enable clock to GPIOC */
    SYSCTL->RCGCGPIO |= 0x10;    /* enable clock to GPIOE */

    KEYPAD_ROW->DIR |= 0x0F;    /* set row pins 3-0 as output */
    KEYPAD_ROW->DEN |= 0x0F;    /* set row pins 3-0 as digital pins */
    KEYPAD_ROW->ODR |= 0x0F;    /* set row pins 3-0 as open drain */

    KEYPAD_COL->DIR &= ~0xF0;   /* set column pin 7-4 as input */
    KEYPAD_COL->DEN |= 0x0F;    /* set column pin 7-4 as digital pins */
    KEYPAD_COL->PUR |= 0x0F;    /* enable pull-ups for pin 7-4
}

/* This is a non-blocking function. */
/* If a key is pressed, it returns 1. */
/* Otherwise, it returns a 0 (not ASCII 0).*/
unsigned char keypad_kbhit(void)
{
```

```

int col;

/* check to see any key pressed */
KEYPAD_ROW->DATA = 0;           /* enable all rows */
col = KEYPAD_COL->DATA & 0xF0;    /* read all columns */
if (col == 0xF0)
    return 0; /* no key pressed */
else
    return 1; /* a key is pressed */
}

```

## Key identification

After a key press is detected, the microprocessor will go through the process of identifying the key. Starting from the top row, the microprocessor drives one row low at a time; then it reads the columns. If the data read is all 1s, no key in that row is activated and the process is moved to the next row. It drives the next row low, reads the columns, and checks for any zero. This process continues until a row is identified with a zero in one of the columns. The next task is to find out which column the pressed key belongs to. This should be easy since each column is connected to a separate input pin. Look at Example 3-1.

### Example 3-1

From Figure 3-7, identify the row and column of the pressed key for each of the following.

- (a) D3–D0 = 1110 for the row, D7–D4= 1011 for the column
- (b) D3–D0 = 1101 for the row, D7–D4= 0111 for the column

### Solution:

From Figure 3-7 the row and column can be used to identify the key.

- (a) The row belongs to D0 and the column belongs to D6; therefore, the key number 2 was pressed.
- (b) The row belongs to D1 and the column belongs to D7; therefore, the key number 7 was pressed.

---

Figure 3-8 is the flowchart for the detection and identification of the key activation.

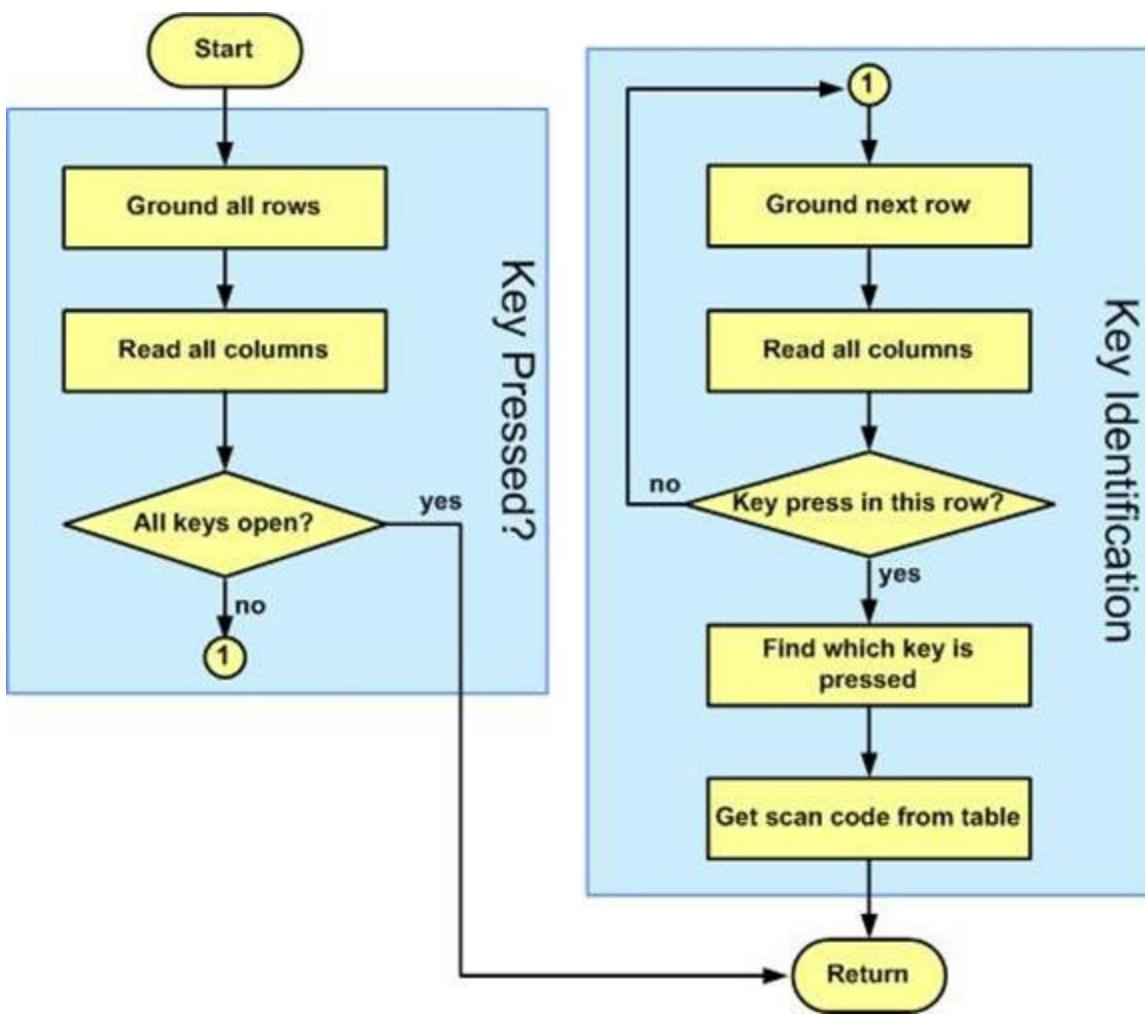


Figure 3-8: The Flowchart for Key Press Detection and Identification

Program 3-4 provides an implementation of the detection and identification algorithm in C language. Next, we will examine it in detail. First for the initialization of the ports, Port E pins 3-0 are used for rows and configured as output digital pin. These pins are also configured as open drain output, which means they are driven low actively but not driven high. Open drain output prevents port damage when two keys of the same column are pressed at the same time. If output pins are driven high and low and two keys of the same column are pressed, it will short the output low to output high of the adjacent pin and cause damages to the output pins. The Port C pins 7-4 are configured as input digital pin with the internal pull-up resistors enabled. This will ensure that the input pins read 1 when no key is pressed.

The key scanning function is a non-blocking function, meaning the function returns regardless of whether there is a key pressed or not. The function first drives all rows low and check to see if any key pressed. If no key is pressed, a zero is returned. Otherwise the code will proceed to check one row at a time by driving only one row low at a time and read the columns. If one of the columns is active, it will find out which column it is. With the combination of the active row and active column, the code will find out the key that is pressed and using the lookup table `keymap[ ]` return the ASCII key label.

**Program 3-4: This program displays the pressed key on the LCD.**

```
...
/* to be used in conjunction with Program 3-2 */
```

```

void keypad_init(void);
unsigned char keypad_getkey(void);

int main(void)
{
    unsigned char key;

    keypad_init();
    LCD_init();

    while(1)
    {
        LCD_command(0x80);          /* LCD cursor location */

        key = keypad_getkey();     /* read the keypad */
        if (key != 0)
        {   /* if a key is pressed */
            LCD_data(key);        /* display the key label */
        }
        else
            LCD_data(' ');

        delayMs(20);              /* wait for a while */
    }
}

#define KEYPAD_ROW GPIOE
#define KEYPAD_COL GPIOC

/* this function initializes the ports connected to the keypad */
void keypad_init(void)
{
    SYSCTL->RCGCGPIO |= 0x04;    /* enable clock to GPIOC */
    SYSCTL->RCGCGPIO |= 0x10;    /* enable clock to GPIOE */

    KEYPAD_ROW->DIR |= 0x0F;      /* set row pins 3-0 as output */
    KEYPAD_ROW->DEN |= 0x0F;      /* set row pins 3-0 as digital pins */
    KEYPAD_ROW->ODR |= 0x0F;      /* set row pins 3-0 as open drain */

    KEYPAD_COL->DIR &= ~0xF0;    /* set column pin 7-4 as input */
    KEYPAD_COL->DEN |= 0xF0;      /* set column pin 7-4 as digital pins */
    KEYPAD_COL->PUR |= 0xF0;      /* enable pull-ups for pin 7-4 */
}

/* This is a non-blocking function to read the keypad. */
/* If a key is pressed, it returns the key label in ASCII encoding. Otherwise, it
returns a 0 (not ASCII 0). */
unsigned char keypad_getkey(void)
{
    const unsigned char keymap[4][4] = {
        { '1', '2', '3', 'A' },
        { '4', '5', '6', 'B' },
        { '7', '8', '9', 'C' },
        { '*', '0', '#', 'D' },
    };

    int row, col;

```

```

/* check to see any key pressed first */
KEYPAD_ROW->DATA = 0;           /* enable all rows */
col = KEYPAD_COL->DATA & 0xF0;    /* read all columns */
if (col == 0xF0) return 0;        /* no key pressed */

/* If a key is pressed, it gets here to find out which key. */
/* Although it is written as an infinite loop, it will take one of the breaks or
return in one pass.*/
while (1)
{
    row = 0;
    KEYPAD_ROW->DATA = 0x0E;      /* enable row 0 */
    delayUs(2);                  /* wait for signal to settle */
    col = KEYPAD_COL->DATA & 0xF0;
    if (col != 0xF0) break;

    row = 1;
    KEYPAD_ROW->DATA = 0x0D;      /* enable row 1 */
    delayUs(2);                  /* wait for signal to settle */
    col = KEYPAD_COL->DATA & 0xF0;
    if (col != 0xF0) break;

    row = 2;
    KEYPAD_ROW->DATA = 0x0B;      /* enable row 2 */
    delayUs(2);                  /* wait for signal to settle */
    col = KEYPAD_COL->DATA & 0xF0;
    if (col != 0xF0) break;

    row = 3;
    KEYPAD_ROW->DATA = 0x07;      /* enable row 3 */
    delayUs(2);                  /* wait for signal to settle */
    col = KEYPAD_COL->DATA & 0xF0;
    if (col != 0xF0) break;

    return 0; /* if no key is pressed */
}

/* gets here when one of the rows has key pressed */
if (col == 0xE0) return keymap[row][0]; /* key in column 0 */
if (col == 0xD0) return keymap[row][1]; /* key in column 1 */
if (col == 0xB0) return keymap[row][2]; /* key in column 2 */
if (col == 0x70) return keymap[row][3]; /* key in column 3 */
return 0; /* just to be safe */
}
...

```

## Debounce

When a mechanical switch is closed or opened, the contacts do not make a clean transition instantaneously, rather the contacts open and close several times before they settle. This event is called contact bounce (see Figure 3-9). So it is possible when the program first detects a switch in the keypad is pressed but when interrogating which key is pressed, it would find no key pressed. This is the reason we have a return 0 after checking all the rows. Another problem manifested by contact bounce is that one key press may be recognized as multiple key presses by the program.

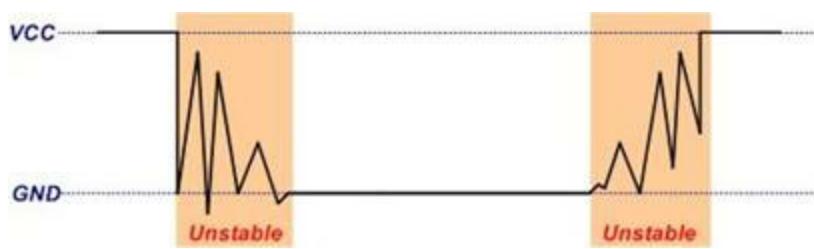


Figure 3-9: Switch contact bounces

We will see the software solution in Program 3-5. The sample program here demonstrates the use of keypad with the LCD. For every key pressed, one character representing the key is added to the display. In the infinite loop of main( ), keypad\_getkey() is repeatedly called to scan the keypad. If it returns something other than 0, a key is pressed and the key label is displayed on the LCD. It then calls delay for 20 ms to wait for the contact bounce to subside. The next while loop waits for the key release, it continuously calls keypad\_getkey( ) until it returns a 0 when key is released. Another debounce for key release is required before the program scan the key again.

**Program 3-5: This program waits for a key to be pressed and then displays the key on the LCD.**

```
...
/* to be used in conjunction with Program 3-4 */

void keypad_init(void);
unsigned char keypad_getchar(void);

int main(void)
{
    unsigned char key;

    keypad_init();
    LCD_init();
    LCD_data('>');

    while(1)
    {
        /* wait until a key is pressed */
        do{
            key = keypad_getkey();
            delayMs(20); /* debounce the key contact */
            while(keypad_getkey() != key);

            LCD_data(key); /* display the key label */

            /* wait until the previous key is released */
            do{
                while(keypad_getkey() != 0);
                delayMs(20); /* wait to debounce */
            }while(keypad_getkey() != 0);
        }
    }
}
```

The debounce code can be put in a new function, as shown in Program 3-6.

**Program 3-6: This program waits for a key to be pressed and then displays the key on the LCD.**

```

...
/* to be used in conjunction with Program 3-4 */

void keypad_init(void);
unsigned char keypad_getchar(void);

int main(void)
{
    unsigned char key;

    keypad_init();
    LCD_init();
    LCD_data('>');

    while(1)
    {
        key = keypad_getchar();
        LCD_data(key);      /* display the key label */
    }
}

/*-----*/
/* This is a blocking function to read the keypad. */
/* When a key is pressed, it returns the key label.*/
/*-----*/
unsigned char keypad_getchar(void)
{
    unsigned char key;
    /* wait until the previous key is released */
    do{
        while(keypad_getkey() != 0);
        delayMs(20);          /* wait to debounce */
    }while(keypad_getkey() != 0);

    do{
        key = keypad_getkey();
        delayMs(20);          /* wait to debounce */
        while(keypad_getkey() != key);

    return key;
}

```

There are IC chips such as National Semiconductor's MM74C923 that incorporate keyboard scanning and decoding all in one chip. Such chips use combinations of counters and logic gates (no microprocessor) to implement the underlying concepts presented in Programs 3-3 through 3-6.

## Review Questions

1. True or false. To see if any key is pressed, all rows are grounded.
2. If D3–D0 = 0111 is the data read from the columns, which column does the key pressed belong to?
3. True or false. Key press detection and key identification require two different processes.
4. In Figure 3-7, if the row has D3–D0 = 1110 and the columns are D3–D0 = 1110, which key

is pressed?

5. True or false. To identify the key pressed, one row at a time is grounded.

## Answers to Review Questions

### Section 3-1

1. Input
2. Input
3. H-to-L
4. High
5. 0x80 and 0xC0

### Section 3-2

1. True
2. Column 3
3. True
4. 0
5. True



## Chapter 4: UART Serial Port Programming

Computers transfer data in two ways: parallel and serial. In parallel data transfers, often eight or more lines (wire conductors) are used to transfer data to another device. In serial communication, the data is sent one bit at a time. Years ago, parallel data transfer was preferred for short distance because it may transfer multiple bits at the same time and provides higher throughput. As technology advances, the data rate of serial communication may exceed parallel communication while parallel communication still retains the disadvantages of the size and cost of cable and connector, the crosstalk between the data lines and the difficulty of synchronizing the arrival time of data lines at longer distance.

Serial communication and the study of associated chips are the topics of this chapter.

## Section 4.1: Basics of Serial Communication

When a microprocessor communicates with the outside world it usually provides the data in byte-sized chunks. For parallel transfer, 8-bit data is transferred at the same time. For serial transfer, 8-bit data is transferred one bit at a time. Figure 4-1 diagrams serial versus parallel data transfers.

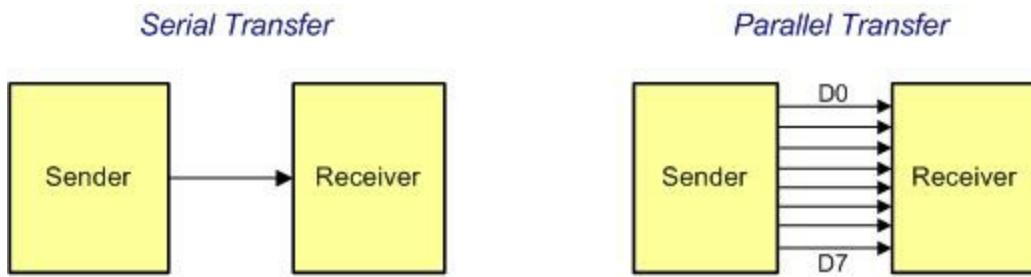


Figure 4-1: Serial vs. Parallel Data Transfer

The fact that in serial communication, a single data line is used instead of the 8-bit data line of parallel communication not only makes it much cheaper but also makes it possible for two computers located in two different cities to communicate.

For serial data communication to work, the byte of data must be grabbed from the 8-bit data bus of the microprocessor and converted to serial bits using a parallel-in-serial-out shift register; then it can be transmitted over a single data line. This also means that at the receiving end there must be a serial-in-parallel-out shift register to receive the serial data, pack it into a byte, and present it to the system at the receiving end. See Figures 4-2 and 4-3.

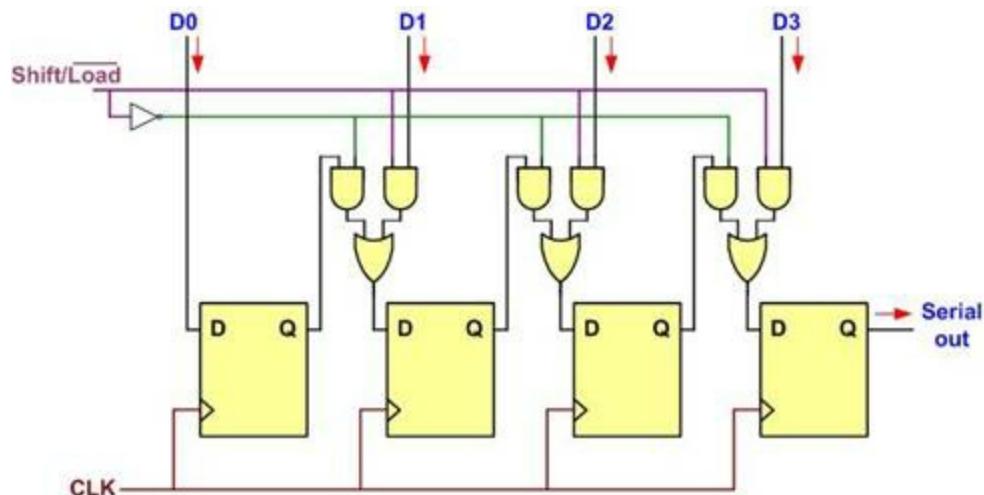


Figure 4-2: Parallel In Serial Out

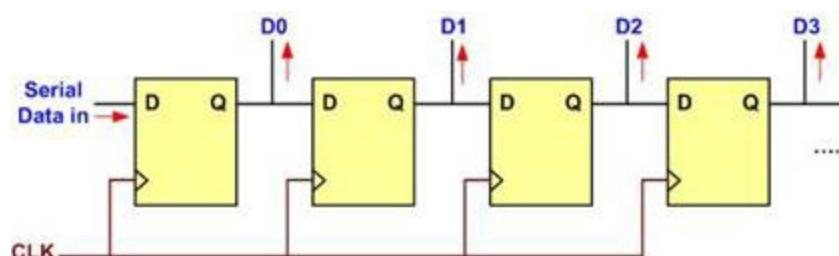


Figure 4-3: Serial In Parallel Out

When the distance is short, the digital signal can be transferred as it is on a simple wire and requires no modulation. This is how PC keyboards transfer data between the keyboard and the motherboard. However, for long-distance data transfers using communication lines such as a telephone, serial data communication requires a modem to modulate (convert from 0s and 1s to audio tones) the data before putting it on the transmission media and demodulate (convert from audio tones to 0s and 1s) at the receiving end.

Serial data communication uses two methods, asynchronous and synchronous. The synchronous method transfers a block of data (characters) at a time while the asynchronous transfers a single byte at a time.

It is possible to write software to use either of these methods, but the programs can be tedious and long. For this reason, special IC chips are made by many manufacturers for serial data communications. These chips are commonly referred to as UART (universal asynchronous receiver-transmitter) and USART (universal synchronous-asynchronous receiver-transmitter). The COM port in the PC uses the UART. When this function is incorporated in a microcontroller, it is often referred as SCI (Serial Communication Interface).

## Half- and full-duplex transmission

In data transmission, a duplex transmission is one in which the data can be transmitted and received. This is in contrast to a simplex transmissions such as printers, in which the computer only sends data. Duplex transmissions can be half or full duplex. If data is transmitted one way at a time, it is referred to as *half duplex*. If the data can go both ways at the same time, it is *full duplex*. Of course, full duplex requires two wire conductors for the data lines (in addition to ground), one for transmission and one for reception, in order to transfer and receive data simultaneously. See Figure 4-4.

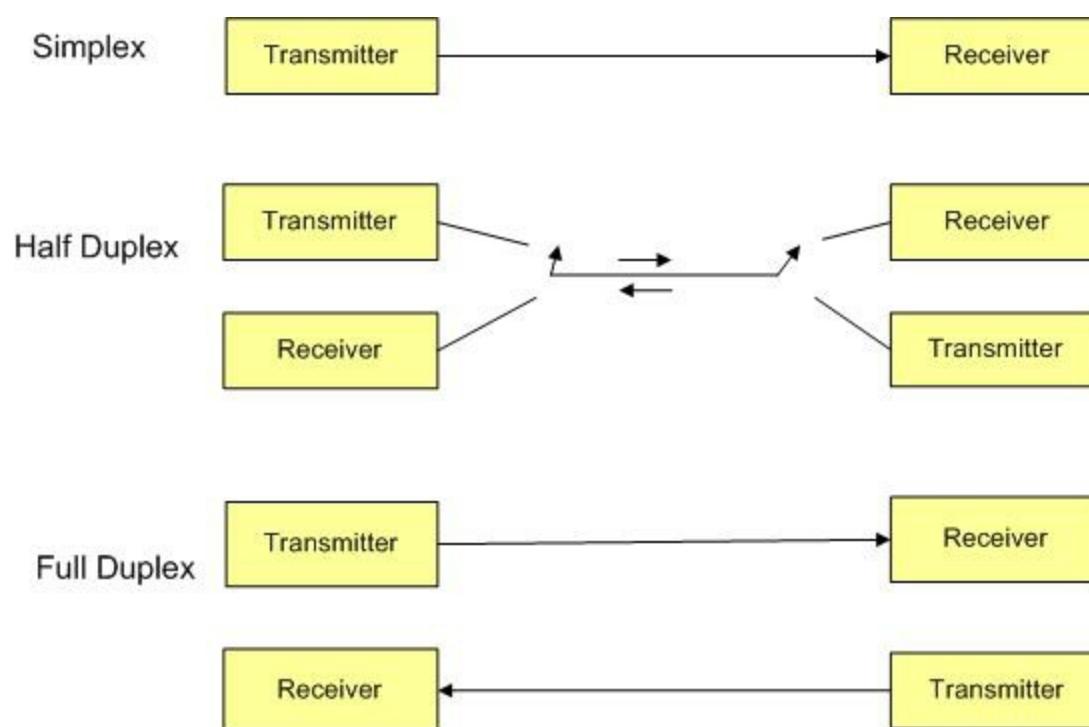


Figure 4-4: Simplex, Half-, and Full-Duplex Transfers

## Asynchronous serial communication and data framing

The data coming in at the receiving end of the data line in a serial data transfer is all 0s and 1s; it is difficult to make sense of the data unless the sender and receiver agree on a set of rules, a *protocol*, on how the data is packed, how many bits constitute a character, and when the data begins and ends.

## Start and stop bits

Asynchronous serial data communication is widely used for character-oriented transmissions. In the asynchronous method, each character, such as ASCII characters, is packed between start and stop bits. This is called *framing*. The start bit is always one bit but the stop bit can be one or two bits. The start bit is always a 0 (low) and the stop bit(s) is 1 (high). For example, look at Figure 4-5 where the ASCII character "A", binary 0100 0001, is framed between the start bit and 2 stop bits. Notice that the LSB is sent out first.

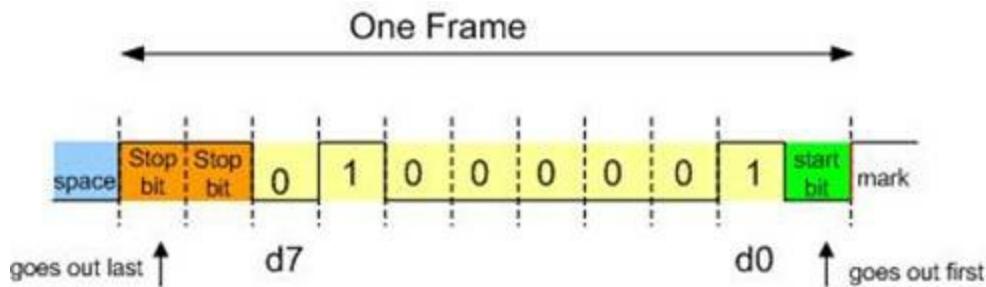


Figure 4-5: Framing ASCII "A" (0x41)

In Figure 4-5, when there is no data transfer, the signal stays 1 (high), which is referred to as *mark*. The 0 (low) is referred to as *space*. Notice that the transmission begins with a start bit followed by D0, the LSB, then the rest of the bits until the MSB (D7), and finally, the 2 stop bits indicating the end of the character "A".

In asynchronous serial communications, peripheral chips can be programmed for data that is 5, 6, 7, or 8 bits wide. While in older systems ASCII characters were 7-bit, the modern systems usually send non-ASCII 8-bit data. The old Baudot code uses 5- or 6-bit characters but they are rarely seen these days even though most of the hardware still supporting them. In some older systems, due to the slowness of the receiving mechanical device, 2 stop bits were used to give the device sufficient time to organize itself before transmission of the next byte. However, in modern PCs the use of 1 stop bit is common. Assuming that we are transferring a text file of ASCII characters using 1 stop bit, we have a total of 10 bits for each character since 8 bits are for the ASCII code, and 1 and 2 bits are for start and stop bits, respectively. Therefore, for each 8-bit character there are an extra 2 bits, or 25% overhead. ( $2/8 \times 100 = 25\%$ )

## Parity bit

In some systems in order to maintain data integrity, the parity bit of the character byte is included in the data frame. This means that for each character (7- or 8-bit, depending on the system) we have a single parity bit in addition to start and stop bits. The parity bit may be odd or even. In the case of an odd-parity the number of data bits, including the parity bit, has an odd number of 1s. Similarly, in an even-parity the total number of bits, including the parity bit, is even. For example, the ASCII character "A", binary 0100 0001, has 0 for the even-parity bit. UART chips allow programming of the parity bit for odd-, even-, and no-parity options, as we

will see in the next section. If a system requires the parity, the parity bit is transmitted after the MSB, and is followed by the stop bit.

## Data transfer rate

The rate of data transfer in serial data communication is stated in *bps* (bits per second). Another widely used terminology for bps is *Baud rate*. However, the baud and bps rates are not necessarily equal. This is due to the fact that baud rate is defined as number of signal changes per second. In modems, it is possible for each signal to transfer multiple bits of data. As far as the conductor wire is concerned, the baud rate and bps are the same, and for this reason in this book we use the terms bps and baud interchangeably.

### Example 4-1

Calculate the total number of bits used in transferring 50 pages of text, each with  $80 \times 25$  characters. Assume 8 bits per character and 1 stop bit.

#### Solution:

For each character a total of 10 bits is used, 8 bits for the character, 1 stop bit, and 1 start bit. Therefore, the total number of bits is  $80 \times 25 \times 10 = 20,000$  bits per page. For 50 pages, 1,000,000 bits will be transferred.

---

### Example 4-2

Calculate the time it takes to transfer the entire 50 pages of data in Example 4-1 using a baud rate of:

- (a) 9600      (b) 57,600

#### Solution:

$$(a) 1,000,000 / 9600 = 104 \text{ seconds}$$

$$(b) 1,000,000 / 57,600 = 17 \text{ seconds}$$

---

### Example 4-3

Calculate the time it takes to download a movie of 2 gigabytes using a telephone line. Assume 8 bits, 1 stop bit, no parity, and 57,600 baud rate.

#### Solution:

$$2 \times 2^{30} \times 10 / 57,600 = 347,222 \text{ seconds} = 4 \text{ days}$$

---

## RS232 and other serial I/O standards

To allow compatibility among data communication equipment made by various manufacturers, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. It has several revisions through the years with an alphabet at the end to denote the revision number such as RS232C. RS stands for recommended standard. It was finally adopted as an EIA standard and renamed EIA232, later on TIA232. In this book we refer to it simply as RS232. Today, RS232 is the most widely used serial I/O interfacing standard. However, since the standard was set long before the advent of the TTL logic family, the input and output voltage levels are not TTL compatible. In the RS232 at the receiver, a 1 is represented by  $-3$  to  $-25$  V, while the 0 bit is  $+3$  to  $+25$  V, making  $-3$  to  $+3$  undefined. For this reason, to connect any RS232 to a TTL-level chip (microprocessor or UART) we must use voltage converters such as MAX232 or MAX233 to convert the TTL logic levels to the RS232 voltage level, and vice versa. MAX232 and MAX233 IC chips are commonly referred to as line drivers. This is shown in Figures 4-4 and 4-5. The MAX232 has two sets of line drivers for transferring and receiving data, as shown in Figure 4-4. The line drivers used for TxD are called T1 and T2, while the line drivers for RxD are designated as R1 and R2. In many applications only one of each is used. Notice in MAX232 that the T1 line driver has a designation of T1in and T1out on pin numbers 11 and 14, respectively. The T1in pin is the TTL side and is connected to TxD of the USART, while T1out is the RS232 side that is connected to the RxD pin of the RS232 DB connector. The R1 line driver has a designation of R1in and R1out on pin numbers 13 and 12, respectively. The R1in (pin 13) is the RS232 side that is connected to the TxD pin of the RS232 DB connector, and R1out (pin 12) is the TTL side that is connected to the RxD pin of the USART. See Figure 4-6. Notice the null modem connection where RxD for one is TxD for the other.

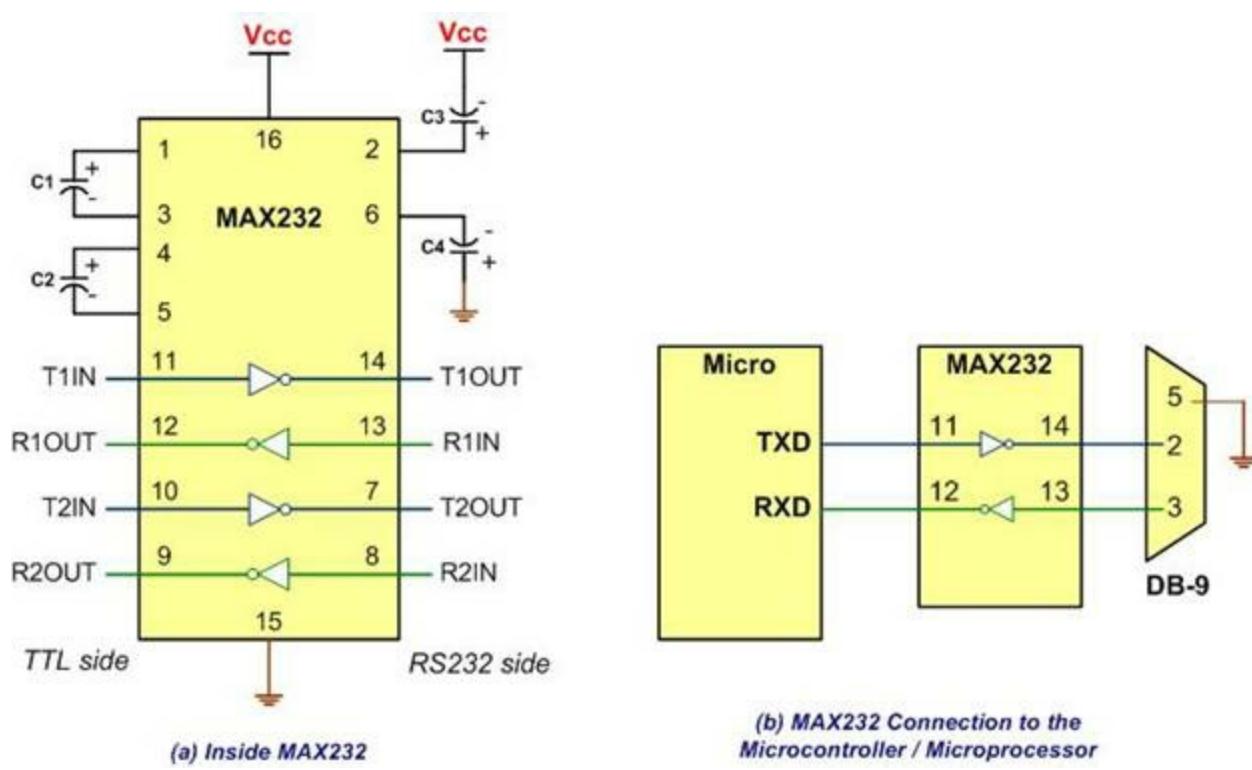


Figure 4-6: MAX232

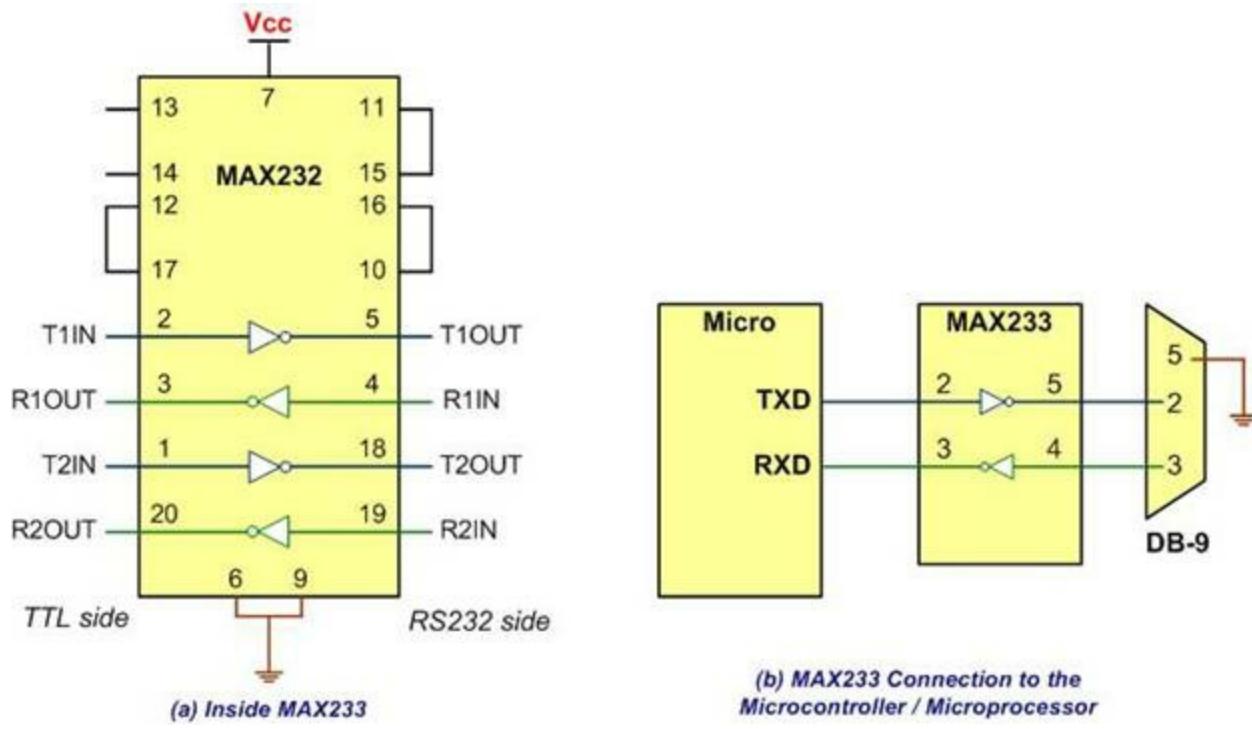


Figure 4-7: MAX233

MAX232 requires four capacitors of  $1\ \mu F$ . To save board space, some designers use the MAX233 chip from Maxim. The MAX233 performs the same job as the MAX232 but eliminates the need for capacitors. However, the MAX233 chip is much more expensive than the MAX232. See Figure 4-7 for MAX233 with no capacitor used.

## RS232 pins

Table 4-1 provides the pins and their labels for the RS232 cable, commonly referred to as the DB-9 connector. The x86 PC 9-pin serial port is shown in Figure 4-8.

Pin	Description
1	Data carrier detect (DCD)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready (DSR)
7	Request to send (RTS)
8	Clear to send (CTS)
9	Ring indicator (RI)

Table 4-1: RS232 Pins

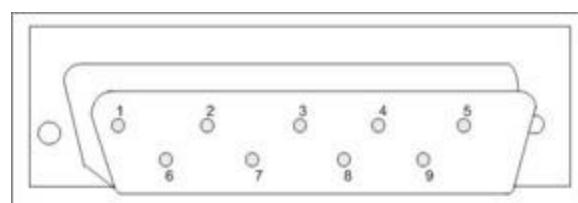


Figure 4-8: 9-Pin Male Connector

## Data communication classification

Current terminology classifies data communication equipment as DTE (data terminal equipment) or DCE (data communication equipment). DTE refers to terminals and computers that send and receive data, while DCE refers to communication equipment, such as modems, that is responsible for transferring the data. Notice that all the RS232 pin function definitions of Table 4-1 are from the DTE point of view.

The simplest connection between two PCs (DTE and DTE) requires a minimum of three pins, TxD, RxD, and ground, as shown in Figure 4-9. Notice that the connection between two DTE devices, such as two PCs, requires pins 2 and 3 to be interchanged as shown in Figure 4-9. In looking at Figure 4-9, keep in mind that the RS232 signal definitions are from the point of view of DTE.

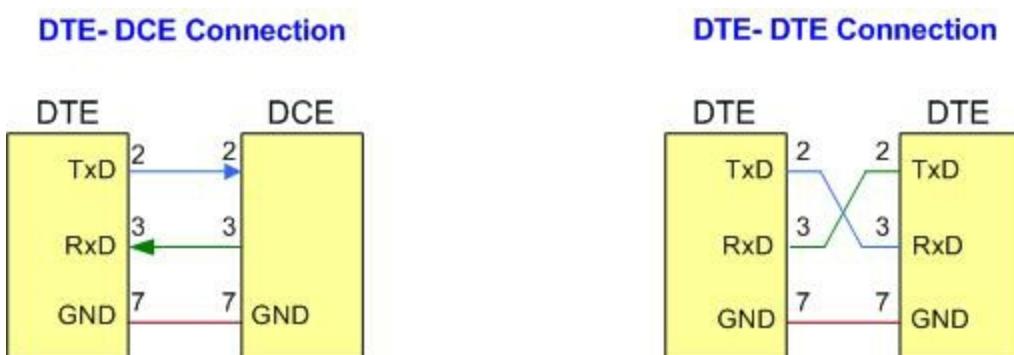


Figure 4-9: DTE-DCE and DTE-DTE Connections

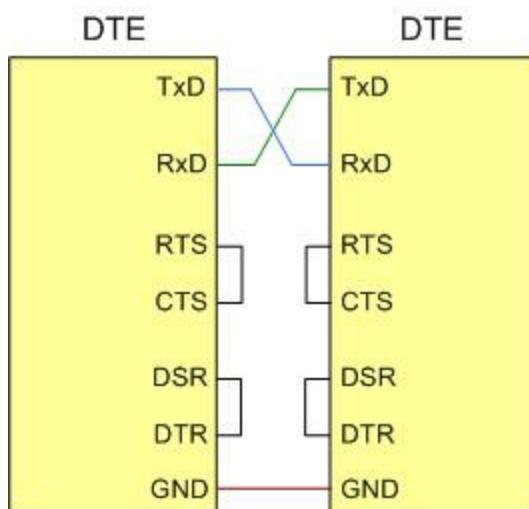
## Examining the RS232 handshaking signals

To ensure fast and reliable data transmission between two devices, the data transfer must be coordinated. Some of the pins of the RS-232 are used for handshaking signals. They are described below. Due to the fact that in serial data communication the receiving device may

have no room for the data there must be a way to inform the sender to stop sending data. So some of these handshaking lines may be used for flow control tool

1. **DTR (data terminal ready):** When the terminal (or a PC COM port) is turned on, after going through a self-test, it sends out signal DTR to indicate that it is ready for communication. If there is something wrong with the COM port, this signal will not be activated. This is an active-low signal and can be used to inform the modem that the computer is alive and kicking. This is an output pin from DTE (PC COM port) and an input to the modem.
2. **DSR (data set ready):** When a DCE (modem) is turned on and has gone through the self-test, it asserts DSR to indicate that it is ready to communicate. Therefore, it is an output from the modem (DCE) and an input to the PC (DTE). This is an active-low signal. If for any reason the modem cannot make a connection to the telephone, this signal remains inactive, indicating to the PC (or terminal) that it cannot accept or send data.
3. **RTS (request to send):** When the DTE device (such as a PC) has a byte to transmit, it asserts RTS to signal the modem that it has a byte of data to transmit. RTS is an active-low output from the DTE and an input to the modem.
4. **CTS (clear to send):** In response to RTS, when the modem has room for storing the data it is to receive, it sends out signal CTS to the DTE (PC) to indicate that it can receive the data now. This input signal to the DTE is used by the DTE to start transmission.
5. **CD (carrier detect, or DCD, data carrier detect):** The modem asserts signal DCD to inform the DTE (PC) that a valid carrier has been detected and that contact between it and the other modem is established. Therefore, DCD is an output from the modem and an input to the PC (DTE).
6. **RI (ring indicator):** An output from the modem (DCE) and an input to a PC (DTE) indicates that the telephone is ringing. It goes on and off in synchronization with the ringing sound. Of the six handshake signals, this is the least often used, due to the fact that modems take care of answering the phone. However, if in a given system the PC is in charge of answering the phone, this signal can be used.

From the above description, PC and modem communication can be summarized as follows: While signals DTR and DSR are used by the PC and modem, respectively, to indicate that they are alive and well, it is RTS and CTS that actually control the flow of data. When the PC wants to send data it asserts RTS, and in response, if the modem is ready (has room) to accept the data, it sends back CTS. If, for lack of room, the modem does not activate CTS, the PC will deassert DTR and try again. RTS and CTS are also referred to as *hardware control flow signals*. See Figure 4-10.



**Figure 4-10: Null Modem Connection with Flow Control Signals**

This concludes the description of the most important pins of the RS232 handshake signals plus TxD, RxD, and ground. Ground is also referred to as SG (signal ground). In the next section we will see serial communication programming for the microcontroller.

## Review Questions

1. The transfer of data using parallel lines is \_\_\_\_\_ (faster, slower) but \_\_\_\_\_ (more expensive, less expensive).
2. In communications between two PCs in New York and Dallas, we use \_\_\_\_\_ (serial, parallel) data communication.
3. In serial data communication, which method fits block-oriented data?
4. True or false. Sending data to a printer is duplex.
5. True or false. In duplex we must have two data lines.
6. The start and stop bits are used in the \_\_\_\_\_ (synchronous, asynchronous) method.
7. Assuming that we are transmitting letter "D", binary 100 0100, with odd-parity bit and 2 stop bits, show the sequence of bits transferred.
8. In Question 7, find the overhead due to framing.
9. Calculate the time it takes to transfer 400 characters as in Question 7 if we use 1200 bps. What percentage of time is wasted due to overhead?
10. True or false. RS232 is not TTL-compatible.

## Section 4.2: Programming UART Ports

In this section, we examine the UART serial port registers of TI ARM Tiva TM4C123GH6PM and show how to program them to transmit and receive data serially. Many of the TI ARM chips come with up to eight on-chip UART ports. They are designated as UART0-UART7. In the TI LaunchPad, the UART0 port of the TM4C123GH6PM is connected to the ICDI (In-Circuit Debug Interface), which is connected to a USB connector. The ICDI USB is located on the right of the slide switch and is labeled as *Debug*. It is on the short side of the board. See Figure 4-11. This ICDI USB connection contains three distinct functions:

- a) the programming (downloading) using LM Flash Programming software,
- b) the debugging using JTAG, and
- c) the use as a virtual COM port.

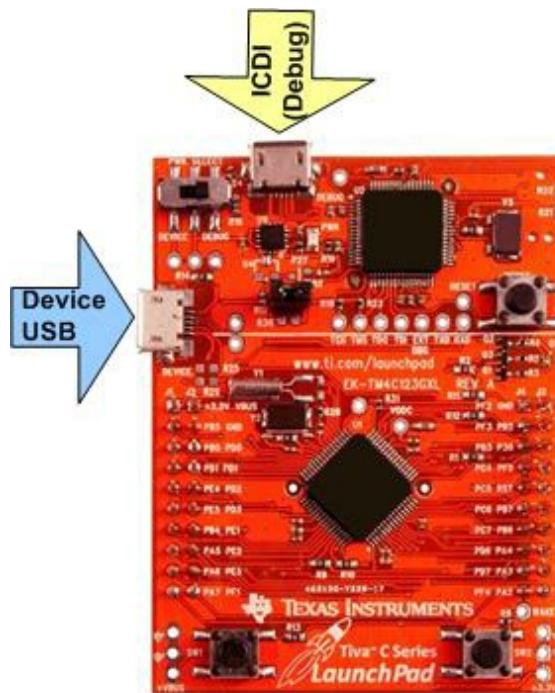


Figure 4-11: Tiva LaunchPad

When the USB cable connects the Tiva LaunchPad, the device driver at the host PC establishes a virtual connection between the PC and the UART0 of the TM4C123GH6PM device. On the LaunchPad, the connection appears as UART0. On the host PC, it appears as a COM port and will work with communication software on the PC such as a terminal emulator. It is called a virtual connection because there is no need for an additional cable to make this connection.

Examining the datasheet of the TM4C123GH6PM LaunchPad, we see the UART0 uses PA0 and PA1 pins as alternate functions for TX0 and RX0, respectively. See Figure 4-12.

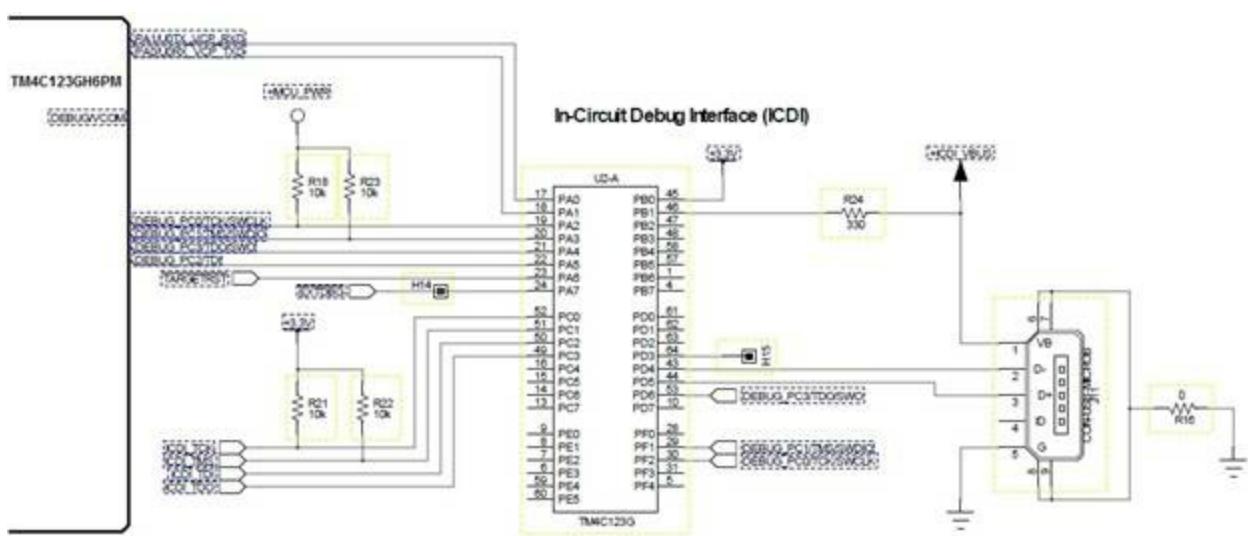


Figure 4-12: ICDI USB Port

Notice that there is a second USB connector on the TI ARM LaunchPad right below the slide switch. It is labeled as **Device** and is on the long side of the board. This USB Device connector is dedicated to USB functionality and uses PD4 and PD5 pins for USB D- and D+ wires, respectively.

As we mentioned earlier, the TI TM4C123GH6PM can have up to 8 UART ports. They are designated as UART0 to UART7. The following shows their Base addresses in the memory map:

- UART0 base: 0x4000.C000
- UART1 base: 0x4000.D000
- UART2 base: 0x4000.E000
- UART3 base: 0x4000.F000
- UART4 base: 0x4001.0000
- UART5 base: 0x4001.1000
- UART6 base: 0x4001.2000
- UART7 base: 0x4001.3000

Figure 4-13 shows the simplified block diagram of the UART units.

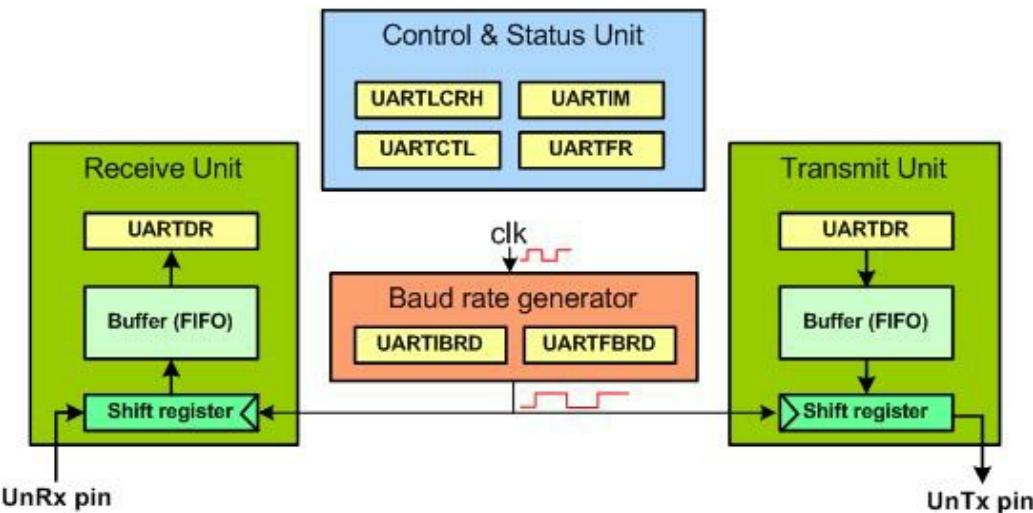


Figure 4-13: a Simplified Block Diagram of UARTn

## UART Registers setup

There are many special function registers associated with each of the above UARTs. In this section, we will be using the UART0 as an example since a virtual connection is available on the TI Tiva LaunchPad. First, we will examine the baud-rate generator registers.

### Baud-Rate Generator

Two registers are used to set the baud rate: They are *UART Integer Baud-Rate Divisor (UARTIBRD)* and *UART Fractional Baud-Rate Divisor (UARTFBRD)*. Their offset addresses are shown below:

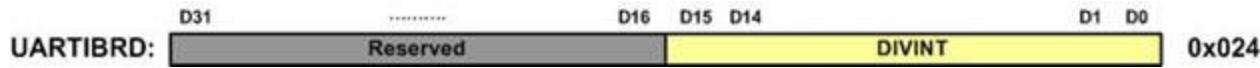


Figure 4-14: UARTIBRD



Figure 4-15: UARTFBRD

For the UART0 used in TI Tiva LaunchPad, their physical addresses are located at 0x4000:C000+0x024 and 0x4000:C000+0x028, respectively. Of the 32-bit of the UARTIBRD, only lower 16 bit are used and of the 32-bit of the UARTFBRD, only the lower 6 bits are used. That gives us total of 22 bits (16 bits integer + 6 bits of fraction). To reduce the rate of error and use the standard baud rate supported by the Terminal programs such as Tera Terminal, we should use both of the above divisor registers when we program the baud rate. Some of the standard baud rates are 4800, 9600, 57600, and 115200. The baud rate can be calculated using the following formula:

$$\text{Desired Baud Rate} = \text{SysClk} / (16 \times \text{ClkDiv})$$

where the SysCLK is the working system clock connected to the UART and ClkDiv is the values we load into divisor registers. The system clock fed to CPU can come from XTAL, PLL, or an internal RC. For TI Tiva LaunchPad, the default system clock is 16 MHz. So we can rearrange the above formula as:

$$\text{Desired Baud Rate} = 16\text{MHz} / (16 \times \text{ClkDiv}) = 1\text{MHz} / \text{ClkDiv}$$

The ClkDiv value gives us both the integer and fractional values needed for the UARTIBRD and UARTFBRD registers. While the integer portion of the divisor is easy to calculate using a simple calculator, the calculation of the fraction part requires some mathematical manipulation. One way would be, to multiply the fraction by 64 and round it by adding 0.5 to it. See example 4-4.

### Example 4-4

Assume SysCLK Frequency=16MHz. Find the values for the divisor registers of UARTIBRD and

UARTFBRD for the following standard baud rates:

- (a) 4800              (b) 9600              (c) 57,600              (d) 115,200

By default, 16 MHz System Clock is divided by 16 before it is fed to the UART. Therefore, we have  $16\text{MHz}/16 = 1\text{MHz}$  and  $\text{ClkDiv} = 1\text{MHz}/\text{baud rate}$ .

- (a)  $1\text{MHz}/4800 = 208.3333$ ,  $\text{UARTIBRD} = 208$  and  $\text{UARTFBRD} = (0.3333 \times 64) + 0.5 = 21.8312 = 21$   
(b)  $1\text{MHz}/9600 = 104.166666$ ,  $\text{UARTIBRD} = 104$  and  $\text{UARTFBRD} = (0.166666 \times 64) + 0.5 = 11$   
(c)  $1\text{MHz}/57600 = 17.361$ ,  $\text{UARTIBRD} = 17$  and  $\text{UARTFBRD} = (0.361 \times 64) + 0.5 = 23$   
(d)  $1\text{MHz}/115200 = 8.680$ ,  $\text{UARTIBRD} = 8$  and  $\text{UARTFBRD} = (0.680 \times 64) + 0.5 = 44$

---

In Example 4-4, it must be noted that default clock is divide-by-16. We can change it to divide-by-8 by setting to 1 the HSE bit of the UART Control (UARTCTL) register. The bits of UARTCTL are covered next.

### UART Control (UARTCTL) register

The next important register in UART is the control register. It is a 32-bit registers and many of the bits are unused. For us the most important bits are RXE, TXE, HSE, and UARLEN.

UARTCTL:	D31	.....	D16	D15	D14	D13	D12	D11	D10	D9	D8						
			Reserved	CTSEN	RTSEN	Reserved		RTS	Res.	RXE	TXE						
	D7	D6	D5	D4	D3	D2	D1	D0	LBE	Res.	HSE	EOT	SMART	SIRLP	SIREN	UARLEN	0x030

Figure 4-16: UART Control (UARTCTL)

**UARLEN (D0) UART enable:** This allows one to enable or disable the UART. During the initialization we must disable it while modifying some of the UART registers,. Also during the execution of some critical tasks, you may want disable the UART to prevent it from interrupting the tasks.

**HSE (D5) High Speed enable:** For the baud rate divisor, by default the system clock is divided by 16 before it is fed to the UART. We can change it to divide-by-8 by setting the HSE=1 to run higher Baud rate with a low system clock frequency.

**RXE (D8) Receive enable:** We must enable this bit to receive data.

**TXE (D9) Transmit Enable:** We must enable this bit to transmit data.

The other bits of this register are used for MODEM signals such as CTS (clear to send), RTS (request to send), parity bit, and so on. We will not use them in this section.

### UART Line Control register

This is the register we use to set the number of bits per character (data length) in a frame and number of stop bits among other things.

UARTLCRH:	D31	.....	D8	D7	D6	D5	D4	D3	D2	D1	D0
			Reserved	SPS	WLEN	FEN	STP2	EPS	PEN	BRK	0x02C

Figure 4-17: UART Line Control (UARTLCRH)

**STP2 (D3) stop bit2.** As we mentioned in the last section, the stop can be 1 or 2. The default is 1 stop bit at the end of each frame. If the receiving device is slow, we can use 2 stop bits by making the STP2=1.

**FEN (D4) FIFO enable.** A major limitation of the one-byte transfer/receive is that it keeps interrupting the CPU for every single byte of data it transmits or receives. In many applications, such a limitation can be a source of severe bottleneck and software overhead, especially in multitasking systems. Therefore, the TI Tiva UART has an internal 16-byte FIFO (first in first out) buffer to store data for transmission. There is also another 16-byte FIFO buffer to save all the received bytes. By enabling FEN bit, we can write up to 16 bytes of data block into its transmission FIFO buffer and let it transfer one byte at a time. The programmer may set up a threshold for the UART to notify the CPU when the level of the FIFO passes the threshold. There is also a 16 byte FIFO for the receiver to buffer the incoming data. Upon Reset, the default for FIFO buffer size is 1 byte (character mode) and that is what we will use in this section.

**WLEN (D6-D5) Word Length:** The number of bits per character data in each frame can be 5, 6, 7, or 8. Generally, we use 8 bits for each character data frame. Notice that default is 5 bits and we must change it to 8 bits as shown below:

D6	D5	
0	0	5 bits
0	1	6 bits
1	0	7 bits
1	1	8 bits

Table 4-2: Data Length

## UART Data Register

To transmit a byte of data we must place it in UART Data register. Although it is a 32-bit register, only the lower 8 bits (D7-D0) are used. It must be noted that "A write to this register initiates a transmission from the UART." In the same way, the received byte is placed in this register and must be retrieved by reading it before it is lost. Notice for transmit, only the lower 8 bits are used. For the receive, the lower 8 bits holds the received byte and other extra 4 bits of D11-D8 are used for error detection such as framing error, parity error, and so on. In this section we only use the lower 8 bits. There is another register called UARTRSR/UARTRCR (UART Receive Status Error/Error Clear) that can be used to check the source of error.

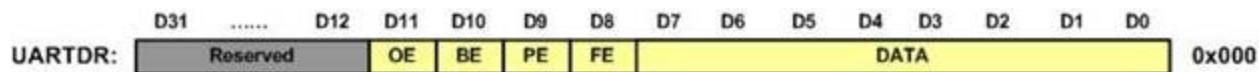


Figure 4-18: UART Data (UARTDR)

## UART Flag Register (Status)

The following discussion assumes that the FIFO buffer is disabled. When FIFO buffer is disabled, the UART Data register holds one byte of data.

UARTFR:	D31	.....	D8	D7	D6	D5	D4	D3	D2	D1	D0	
Type	RO		RO	0x018								
Reset	0		0	1	0	0	1	0	0	0	0	

Figure 4-19: UART Flag Register (UARTFR)

**TXFE (TX FIFO empty, D7):** When we write a byte to Data register to be transmitted, the byte is placed in transmission FIFO buffer. When the transmitter is not busy, it loads one byte from the FIFO buffer and the FIFO becomes empty and the TXFE is raised. The transmitter then frames the byte and sends it out via TxD pin bit by bit serially.

**RXFF (RX FIFO full, D6):** In the same way, when a byte of data is received, this byte is placed in Data register and RXFF (RX FIFO full) flag bit is raised. In other words, when RXFF is HIGH, it means a byte has been received.

**TXFF (TX FIFOI full D5):** When we write a byte to Data register to be transmitted, the byte is placed in transmission FIFO buffer. When the transmitter is not busy, it loads one byte from the FIFO buffer and the FIFO is not full anymore and the TXFF is lowered. The transmitter then frames the byte and sends it out via TxD pin bit by bit serially. This is essentially the opposite of TXFE flag bit when FIFO buffer is disabled. We can monitor TXFF flag and upon going LOW we can write another byte to the Data register while the last byte is being transmitted.

**RXFE (RX FIFO empty, D4):** The received byte of data comes in serially into a Receive buffer (first bit coming in is the start bit, 8-bit data, and stop bit). After the last bit (stop bit) has been received by the receiver, the UART circuitry discards the start and stop bits and delivers a byte of data to the Data register and lowers the RXFE bit (RX FIFO empty). This is essentially the opposite of RXFF flag bit. We can monitor RXFE flag and upon going LOW (buffer not empty) we should read (retrieve) the data from Data register.

**Busy(D3):** When a byte is written to the Data register the Busy flag goes High. While UART busy transmitting data this bit remains high until the stop bit is gone. Upon transmission of the last bit (stop bit), it goes low indicating the transmission is completed. This is only used when it is essential to complete the transmission before starting the next task.

## Enabling clock to UART

As we mentioned in Chapter 2, to conserve power the on-chip peripherals have no clock coming out of reset. The RCGCUART register is used to enable the clock to the UART. In this register, there is a bit for each of the UART0 to UART7 modules. If a given UART is not used, we should disable the clock to it to save power. To use UART0, we set to high the D0 of this register.

RCGCUART:	D31	.....	D8	D7	D6	D5	D4	D3	D2	D1	D0	
Type	RO		RO	R/W	0x618							

Figure 4-20: RCGCUART (UART Run Mode Clock Gating Control)

## The GPIO pins used for UART TxD and RxD

In addition to the UART registers setup, we must also configure the I/O pins used for

UART for their alternate functions. In the case of UART, we need to set up GPIO pins for the alternate functions of TxD and RxD signals. In the last two chapters we used GPIO as simple I/O. We showed the minimum configuration for each port as Digital I/O and providing the Clock to it. When GPIO pins are used for their alternate peripheral functions such as UART, Timer, and ADC, we need to configure five more registers. They are PORTx Run Mode Clock Gating Control, PORTx Digital Enable, PORTx ADC Mode Selection, PORTx Alternate Selection and PORTx Port Control registers. The first two registers are covered in Chapter 2. We examine the next three registers here.

GPIOAMSEL:										0x528
Type	RO	RO	RO	RO	RO	R/W	R/W	R/W	R/W	
Reset	0	0	0	0	0	0	0	0	0	
D31	.....	D8	D7	D6	D5	D4	D3	D2	D1	D0
Reserved										GPIOAMSEL

Figure 4-21: GPIOAMSEL (GPIO Analog Mode Select)

The TI ARM comes with on-chip ADC (analog to digital converter). Assume a given pins can be used for both ADC and UART and we want to use it for UART. In this case, we must isolate the ADC function since it is not used for analog operation. This is done with GPIOAMSEL (GPIO Analog Mode Selection). Although each port has its own PORTxAMSEL register, not all the pins of each port has ADC capability. The good news is upon Reset, the PORTxAMSEL register has all 0s which means ADC function for the pin, if there is one, is disabled.

In addition to Digital I/O and Clock enable, we must also configure the alternate function select register.

GPIOAFSEL:										0x420
Type	RO	RO	R/W							
Reset	0	0	0	0	0	0	0	0	0	
D31	.....	D8	D7	D6	D5	D4	D3	D2	D1	D0
Reserved										AFSEL

Figure 4-22: GPIO Alternate Function Select (GPIOAFSEL)

Upon Reset, the GPIO pins are configured for simple I/O. To use the alternate function (e.g. UART) of given pin of a given Port, there are two steps to be taken. First we must set the corresponding bits in the alternate function select HIGH. In the case of UART0, PA0 and PA1 pins are used for RxD and TxD signals, respectively. Therefore, we must set both of them high in order to be used for the alternate functions of RxD and TxD. A one in the alternate function select register tells the port pin to perform a function other than simple I/O. In addition to that, we need to tell the port pin which alternate function it should do since for most of the pins, there are multiple choices of alternate functions it can serve.

The alternate function is selected by the Port Control Register. Port Control Register is a 32 register. Four bits are used for the selection of the function of each pin. That results nicely in one hexadecimal digit for one pin. For most of the pins, UART function select number is 1. The only exceptions are PC4 and PC5. PC4 and PC5 may be used for either UART4 or UART1. When they are used for UART4, the select number is 1. When they are used for UART1, the select number is 2. See Figure 4-23 and Table 4-3.

D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16
GPIO PCTL:				PMC7	PMC6			PMC5			PMC4				
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
PMC3				PMC2				PMC1				PMC0			

Figure 4-23: GPIO PCTL Register

Pin	Digital Function (GPIO PCTL PMCx)															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
17	PA0	U0Rx								CAN1Rx						
18	PA1	U0Tx								CAN1Tx						
45	PB0	U1Rx						T2CCP0								
46	PB1	U1Tx						T2CCP1								
16	PC4	U4Rx	U1Rx		M0PWM6		IDX1	WT0CCP0	U1RTS							
15	PC5	U4Tx	U1Tx		M0PWM7		PhA1	WT0CCP1	U1CTS							
14	PC6	U3Rx					PhB1	WT1CCP0	USBOEPEN							
13	PC7	U3Tx						WT1CCP1	USBOPFLT							
43	PD4	U6Rx						WT4CCP0								
44	PD5	U6Tx						WT4CCP1								
53	PD6	U2Rx		M0FAULT0		phA0	WT5CCP0									
10	PD7	U2Tx				phB0	WT5CCP1	NMI								
9	PE0	U7Rx							USBOEPEN							
8	PE1	U7Tx							USBOPFLT							
59	PE4	U5Rx		I2C25CL	M0PWM4	M1PWM2			CAN0Rx							
60	PE5	U5Tx		I2C25DA	M0PWM5	M1PWM3			CAN0Tx							
28	PF0	U1RTS	SSI1Rx	CAN0Rx		M1PWM4	PhA0	TO CCP0	NMI							
29	PF1	U1CTS	SSI1Tx			M1PWM5	PhB0	TO CCP1								

Table 4-3: UART Pins and GPIO PCTL Registers

## Steps for transmitting data

Here are the steps to configure the UART0 and transmit a byte of data for TI Tiva TM4C123GH6PM on the LaunchPad:

- 1) Provide clock to UART0 by writing a 1 to RCGCUART register.
- 2) Provide clock to PORTA by writing a 1 to RCGCGPIO register.
- 3) Disable the UART0 by writing 0 to UARTCTL register of UART0.
- 4) Write the integer portion of the Baud rate to the UARTIBRD register of UART0.
- 5) Write the fractional portion of the Baud rate to the UARTRFBRD register of UART0.
- 6) Select the system clock as UART clock source by writing a 0 to UARTCC register of UART0.
- 7) Configure the line control value for 1 stop bit, no FIFO, no interrupt, no parity, and 8-bit data size. That gives us 0x60 for the UARTLCRH register of UART0.
- 8) Set TxE and RxE bits in UARTCTL register to enable the transmitter and receiver of UART0.
- 9) Set UARTEN bit in UARTCTL register to enable the UART0.
- 10) Make PA0 and PA1 pins to be used as Digital I/O.
- 11) Select the alternate functions of PA0 (RxD) and PA1 (TxD) pins using the GPIOAFSEL.
- 12) Configure PA0 and PA1 pins for UART function.
- 13) Wait for TxD output to establish idle high.
- 14) Monitor the TXFF flag bit in UART Flag register and when it goes LOW (buffer not full), write a byte to Data register to be transmitted.

Program 4-1 sends the characters "YES" to the terminal emulator program at a PC. You need to install TeraTerminal (or other terminal program such as HyperTerminal or Putty) on your PC to receive the output. For TeraTerminal download and tutorial see:

[http://microdigitaled.com/tutorials/Tera\\_Terminal.pdf](http://microdigitaled.com/tutorials/Tera_Terminal.pdf)

## Program 4-1: UART0 Transmit

```

/* p4_1.c: Sending "YES" to UART0 on TI ARM Launchpad (TM4C123GH6PM) */
/* UART0 is on USB/Debug */
/* Use TeraTerm to see the message "YES" on a PC */

#include <stdint.h>
#include "tm4c123gh6pm.h"

void UART0Tx(char c);
void delayMs(int n);

int main(void)
{
    SYSCTL->RCGCUART |= 1; /* provide clock to UART0 */
    SYSCTL->RCGCGPIO |= 1; /* enable clock to PORTA */

    /* UART0 initialization */
    UART0->CTL = 0;          /* disable UART0 */
    UART0->IBRD = 104;        /* 16MHz/16=1MHz, 1MHz/104=9600 baud rate */
    UART0->FBRD = 11;         /* fraction part, see Example 4-4 */
    UART0->CC = 0;            /* use system clock */
    UART0->LCRH = 0x60;       /* 8-bit, no parity, 1-stop bit, no FIFO */
    UART0->CTL = 0x301;        /* enable UART0, TXE, RXE */

    /* UART0 TX0 and RX0 use PA0 and PA1. Set them up. */
    GPIOA->DEN = 0x03;        /* Make PA0 and PA1 as digital */
    GPIOA->AFSEL = 0x03;       /* Use PA0,PA1 alternate function */
    GPIOA->PCTL = 0x11;        /* configure PA0 and PA1 for UART */

    delayMs(1);                /* wait for output line to stabilize */

    for(;;)
    {
        UART0Tx('Y');
        UART0Tx('E');
        UART0Tx('S');
        UART0Tx(' ');
    }
}

/* UART0 Transmit */
/* This function waits until the transmit buffer is available then */
/* writes the character in the transmit buffer. It does not wait */
/* for transmission to complete. */
void UART0Tx(char c)
{
    while((UART0->FR & 0x20) != 0); /* wait until Tx buffer not full */
    UART0->DR = c;                  /* before giving it another byte */
}

```

## Steps for receiving data

Here are the steps to receive a byte of data for UART0 in TI ARM Tiva 4MC123G6PM on LaunchPad:

- 1) Provide clock to UART0 by writing a 1 to RCGCUART register.
- 2) Provide clock to PORTA by writing a 1 to RCGCGPIO register.
- 3) Disable the UART0 by writing 0 to UARTCTL register of UART0.
- 4) Write the integer portion of the Baud rate to the UARTIBRD register of UART0.
- 5) Write the fractional portion of the Baud rate to the UARTFBRD register of UART0.
- 6) Select the system clock as UART clock source by writing a 0 to UARTCC register of UART0.
- 7) Configure the line control value for 1 stop bit, no FIFO, no interrupt, no parity, and 8-bit data size. That gives us 0x60 for the UARTLCRH register of UART0.
- 8) Set TxE and RxE bits in UARTCTL register to enable the transmitter and receiver of UART0.
- 9) Set UARTEN bit in UARTCTL register to enable the UART0.
- 10) Make PA0 and PA1 pins to be used as Digital I/O.
- 11) Select the alternate functions of PA0 (RxD) and PA1 (TxD) pins using the GPIOAFSEL.
- 12) Configure PA0 and PA1 pins for UART function.
- 13) Monitor the RXFE flag bit in UART Flag register and when it goes LOW (buffer not empty), read the received byte from Data register and save it.

### Note

The configuration steps (steps 1 to 12) are identical for receiving and sending data.

Program 4-2 receives the bytes of data serially via UART0 and displays it on PORTF LEDs.

### Program 4-2: UART0 Receive

```
/* p4_2.c: Read data from UART0 and display it at the tri-color LEDs. */
/* The LEDs are connected to Port F 3-1. */
/* Press any A-z, a-z, 0-9 key at the terminal emulator */
/* and see ASCII value in binary is displayed on LEDs of PORTF. */

#include <stdint.h>
#include "tm4c123gh6pm.h"

char UART0Rx(void);
void delayMs(int n);

int main(void)
{
    char c;

    SYSCTL->RCGCUART |= 1; /* provide clock to UART0 */
    SYSCTL->RCGCGPIO |= 1; /* enable clock to PORTA */
    SYSCTL->RCGCGPIO |= 0x20; /* enable clock to PORTF */
```

```

/* UART0 initialization */
UART0->CTL = 0;           /* disable UART0 */
UART0->IBRD = 104;         /* 16MHz/16=1MHz, 1MHz/104=9600 baud rate */
UART0->FBRD = 11;          /* fraction part, see Example 4-4 */
UART0->CC = 0;             /* use system clock */
UART0->LCRH = 0x60;        /* 8-bit, no parity, 1-stop bit, no FIFO */
UART0->CTL = 0x301;         /* enable UART0, TXE, RXE */

/* UART0 TX0 and RX0 use PA0 and PA1. Set them up. */
GPIOA->DEN = 0x03;          /* Make PA0 and PA1 as digital */
GPIOA->AFSEL = 0x03;         /* Use PA0,PA1 alternate function */
GPIOA->PCTL = 0x11;          /* configure PA0 and PA1 for UART */

GPIOF->DIR = 0x0E;          /* configure Port F to control the LEDs */
GPIOF->DEN = 0x0E;
GPIOF->DATA = 0;

for(;;)
{
    c = UART0Rx();           /* get a character from UART */
    GPIOF->DATA = c << 1;    /* shift left and write it to LEDs */
}

/* UART0 Receive */
/* This function waits until a character is received then returns it. */
char UART0Rx(void)
{
    char c;
    while((UART0->FR & 0x10) != 0); /* wait until the buffer is not empty */
    c = UART0->DR;                  /* read the received data */
    return c;                        /* and return it */
}

/* Append delay functions and SystemInit() here */

```

## Using UARTx port

The last two program showed how to use the UART0 on TI ARM LaunchPad. Now, you can buy a USB-to-Serial module and connect it to any of the UART1 to UART7 ports. One side of the USB-to-Serial module is TTL signals for TxD, RxD, Gnd and is connected to the UARTx pins on TI ARM LaunchPad. The other side is USB port connected to the PC USB port. See this link.

<http://www.adafruit.com/products/954>

If your computer has a physical COM port, you may use a TTL-RS232 converter such as:

<http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,401,463&Prod=PMOD-RS232>

Program 4-3 is modified from Program 4-1 to use UART5. Compare them you will find that the initialization of the associated port is changed. Program 4-3 also demonstrates how to initialize an array of characters and send the message string to UART.

### Program 4-3: Sending "Hello" to TeraTerminal via UART5

```
/* p4_3.c: Sending "Hello" to UART5 on TI ARM LaunchPad (TM4C123GH6PM) */
/* UART5 Tx is on PE5 */
/* Use TeraTerm to see the message "Hello" on a PC */

#include <stdint.h>
#include "tm4c123gh6pm.h"

void UART5Tx(char c);
void delayMs(int n);

int main(void)
{
    char message[] = "Hello";
    int i;

    SYSCTL->RCGCUART |= 0x20; /* provide clock to UART5 */
    SYSCTL->RCGCGPIO |= 0x10; /* Enable clock to PORTE */

    /* UART5 initialization */
    UART5->CTL = 0;           /* disable UART5 */
    UART5->IBRD = 104;        /* 16MHz/16=1MHz, 1MHz/104=9600 baud rate */
    UART5->FBRD = 11;         /* fraction part, see Example 4-4 */
    UART5->CC = 0;            /* use system clock */
    UART5->LCRH = 0x60;       /* 8-bit, no parity, 1-stop bit */
    UART5->CTL = 0x301;       /* enable UART5, TXE, RXE */

    /* UART5 TX5 and RX5 use PE5 and PE4. Set them up. */
    GPIOE->DEN = 0x30;        /* make PE5, PE4 as digital */
    GPIOE->AMSEL = 0;          /* turn off analog function */
    GPIOE->AFSEL = 0x30;       /* use PE5, PE4 alternate function */
    GPIOE->PCTL = 0x00110000;  /* configure PE5, PE4 for UART5 */

    delayMs(1);               /* wait for output line to stabilize */

    for(;;)
    {
        for (i = 0; i < 5; i++)
            UART5Tx(message[i]);
    }
}

/* UART5 Transmit */
void UART5Tx(char c)
{
    while((UART5->FR & 0x20) != 0); /* wait until Tx buffer not full */
    UART5->DR = c;                  /* before giving it another byte */
}

/* Append delay functions and SystemInit() here */
```

Program 4-4 combines the UART transmit with UART receive. When a key on the PC is pressed with a terminal emulator program, the key character is sent to Tiva LaunchPad UART. The received character is sent back out through UART to the terminal emulator.

#### Program 4-4: Echoing the received data

```
/* p4_4.c: This program sets up UART5 on TI ARM LaunchPad (TM4C123GH6PM) to do terminal echo. When a key is pressed at the terminal emulator of the PC, the character is received by UART5 and it is sent out of UART5 back to the terminal. */
/* UART5 Tx is on PE5, Rx is on PE4. */
/* Use TeraTerm to see that the keys are echoed. */

#include <stdint.h>
#include "tm4c123gh6pm.h"

char UART5Rx(void);
void UART5Tx(char c);
void delayMs(int n);

int main(void)
{
    char c;

    SYSCTL->RCGCUART |= 0x20; /* provide clock to UART5 */
    SYSCTL->RCGCGPIO |= 0x10; /* Enable clock to PORTE */

    /* UART5 initialization */
    UART5->CTL = 0;           /* disable UART5 */
    UART5->IBRD = 104;        /* 16MHz/16=1MHz, 1MHz/104=9600 baud rate */
    UART5->FBRD = 11;         /* fraction part, see Example 4-4 */
    UART5->CC = 0;            /* use system clock */
    UART5->LCRH = 0x60;       /* 8-bit, no parity, 1-stop bit */
    UART5->CTL = 0x301;        /* enable UART5, TXE, RXE */

    /* UART5 TX5 and RX5 use PE5 and PE4. Set them up. */
    GPIOE->DEN = 0x30;        /* make PE5, PE4 as digital */
    GPIOE->AMSEL = 0;          /* turn off analog function */
    GPIOE->AFSEL = 0x30;        /* use PE5, PE4 alternate function */
    GPIOE->PCTL = 0x00110000;   /* configure PE5, PE4 for UART5 */

    delayMs(1);                /* wait for output line to stabilize */
    UART5Tx('>>');             /* send the prompt character */

    for(;;)
    {
        c = UART5Rx();          /* get a character from UART */
        UART5Tx(c);              /* echo it back to the terminal */
    }
}

/* UART0 Receive */
char UART5Rx(void)
{
    char c;
    while((UART5->FR & 0x10) != 0); /* wait until the buffer is not empty */
    c = UART5->DR;                  /* read the received data */
    return c;                        /* and return it */
}

/* UART5 Transmit */
void UART5Tx(char c)
```

```

{
    while((UART5->FR & 0x20) != 0); /* wait until Tx buffer not full */
    UART5->DR = c;                  /* before giving it another byte */
}

/* Append delay functions and SystemInit() here */

```

# Review Questions

1. The TI Tiva LaunchPad comes with maximum of \_\_\_\_ on-chip UART.
  2. In TI Tiva TM4C123G pins \_\_ and \_\_ are used for TxD and RxD of UART0.
  3. Which register is used to set the data size and number of stop bits?
  4. How do we know if the transmit buffer is not full before we load in another byte?
  5. How do we know if a new byte has been received?

# Answer to Review Questions

## Section 4-1

1. False, more expensive
  2. Serial
  3. Synchronous
  4. False; it is simplex.
  5. True
  6. Asynchronous
  7. With 100 0100 binary we have 1 as the odd-parity bit. The bits as transmitted in the sequence are:  

(a) 0 (start bit)	(b) 0	(c) 0	(d) 1
(e) 0	(f) 0	(g) 0	(h) 1
(i) 1 (parity)	(j) 1 (first stop bit)	(k) 1 (second stop bit)	
  8. 4 bits
  9.  $400 \times 11 = 4400$  bits total bits transmitted.  $4400/1200 = 3.667$  seconds,  $4/7 = 58\%$ .
  10. True

## Section 4-2

1. 8
  2. PA0 and PA1
  3. UARTLCRH
  4. the TXFF flag from the UARTFR register goes low.
  5. The RXFE flag from the UARTFR register goes low.



## Chapter 5: TI ARM Timer Programming

In Section 5-0, the counter and timer concepts are reviewed. Section 5-1 covers the System Tick Timer which is available in all ARM Cortex microcontrollers. In Section 5-2, delays are made using 16/32-bit TI timers. Section 5-3 surveys the Real-Time mode. In Section 5-4, input edge-time mode is discussed and the pulse width and frequency measuring is covered. The event counter feature is studied in Section 5-5. The 32/64-bit timers are covered in Section 5-6.

## Section 5.0: Introduction to counters and timers

In the digital design course you connected many flip flops (FFs) together to create up counter/down counter. For example, connecting 3 FFs together we can count up to 7 (000-111 in binary). This is called *3-bit counter*. The same way, to create a 4-bit counter (counting up to 15, or 0000-1111 in binary) we need 4 FFs. For 16-bit counter, we need 16 FFs and it counts up to  $2^{16} - 1$ . Figure 5-1 shows the T flip flop connection and pulse outputs for all three flop flops.

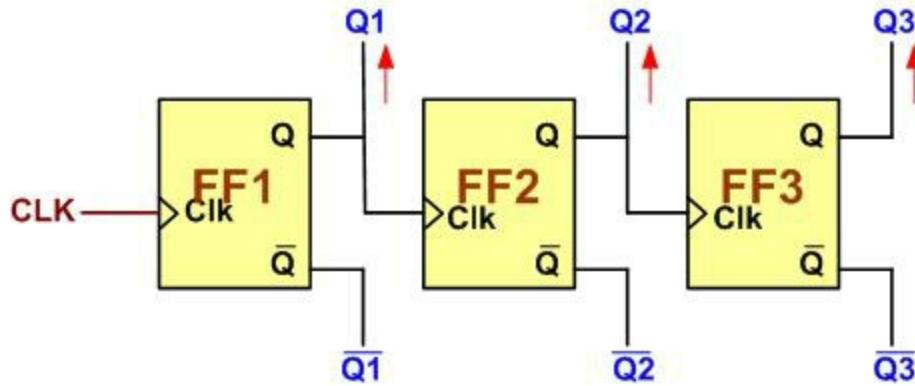


Figure 5-1: A 3-bit Counter

Regarding Figure 5-1, notice the following points:

- 1) The Q outputs gives the down counter.
- 2) The  $\bar{Q}$  (Q not) gives us up counter.
- 3) The frequency on Q3 is  $\frac{1}{3}$  of the Clock fed to FF1.
- 4) We can use the circuit in Figure 5-1 to divide clock frequency.
- 5) We can use the circuit in Figure 5-1 to count the number of pulses fed to CLK pin of FF1.

An up counter begins counting from 0 and its value increases on each clock until it reaches its maximum value. Then, it overflows and rolls over to zero in the next clock. The following figure shows the stages which an 8-bit counter goes through.

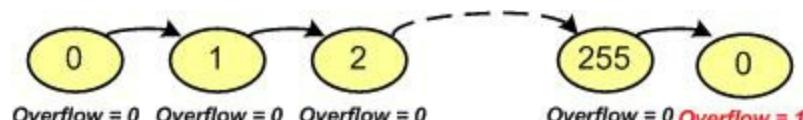


Figure 5-2: an 8-bit Up-Counter Stages

A down counter begins counting from its maximum value and decreases on each clock until it reaches to 0. Then, it underflows and rolls over to its maximum value in the next clock. The following figure shows the stages which an 8-bit down counter goes through.

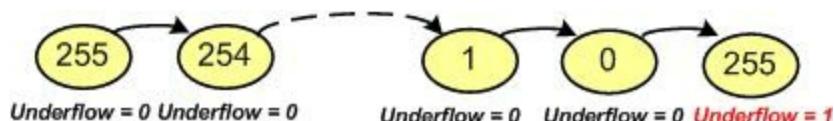


Figure 5-3: an 8-bit Down-Counter Stages

## Counter Usages

Counters have different usages. Some of them are:

1. Counting events
2. Making delays (Using Counter as a Timer)
3. Measuring the time between 2 events

## 1. Counting events

You might need to count the number of cars going through a street or the number of spaghetti packs which produced in a factory. To do so, you can connect the output of a sensor to a counter, as shown in the following figure.

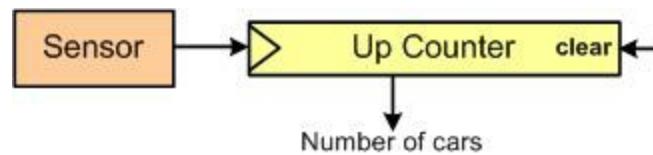


Figure 5-4: Counting Events Using a Counter

## 2. Making delays (Using Counter as a timer)

While controlling devices, it is a common practice to start or terminate a task when a desired amount of time elapsed. For example, a washing machine or an oven do each task for a determined amount of time. To do timing, we can connect a clock generator to a counter, and wait until a desired amount of time elapses. For example, in the following picture, the clock generator makes a 1 Hz signal and the counter increasing every second. The counter reaches to 60 after 60 seconds.

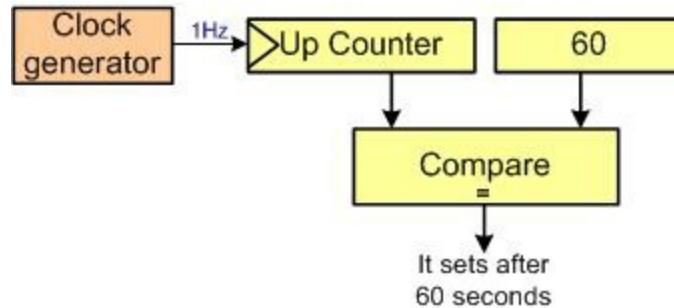


Figure 5-5: Using Counter as a Timer

## 3. Measuring the time between 2 events

You might need to measure the time between 2 events. For example, the amount of time it takes a marathon runner to go from the start to the finish point. In such cases we can use a circuit similar to the following:

The counter is cleared at the start. Then, it increases on each clock pulse. The value of the counter is loaded into another register when the runner passes the finish line.

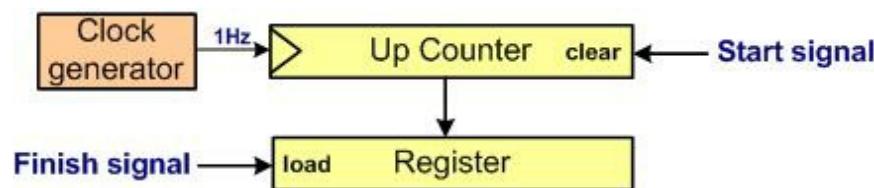


Figure 5-6: Capturing

Nowadays, all the microcontrollers come with on-chip Timer/Counter. If the clock to the Timer comes from internal source such as PLL, XTAL, and RC, then it is called a *Timer*. If the clock source comes from external source, such as pulses fed to the CPU pin, then it is called a *Counter*. By Counter it is meant event-counter since it counts the event happening outside the CPU. In many microcontrollers, the Timers can be used as Timer or Counter.

## Review Questions

1. With 5 FFs we can get maximum of \_\_\_\_\_ count.
2. With 5 FFs we can divide the frequency by maximum of \_\_\_\_\_.
3. When pulses are fed to a timer from the outside it is called \_\_\_\_\_.
4. When clocks pulses are fed to a timer from inside it is called \_\_\_\_\_ .
5. If we need to divide a frequency by 500, we need \_\_\_\_\_ flip flops.

## Section 5.1: System Tick Timer

System tick timer allows the system to initiate an action on a periodic basis. This action is performed internally at a fixed rate without external signal. For example, in a given application we can use SysTick to read a sensor every 200 msec. SysTick is used widely by operating systems so that the system software may interrupt the application software periodically (often 10 ms interval) to monitor and control the system operations. The SysTick is a 24-bit down counter driven by either the system clock or the internal oscillator. It counts down from an initial value to 0. When it reaches 0, in the next clock, it underflows and it raises a flag called COUNT and reloads the initial value and starts all over. We can set the initial value to a value between 0x000000 to 0xFFFFFFF. See the following figure.

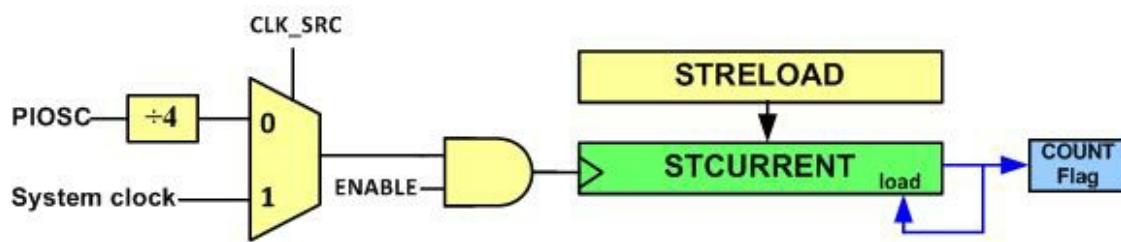


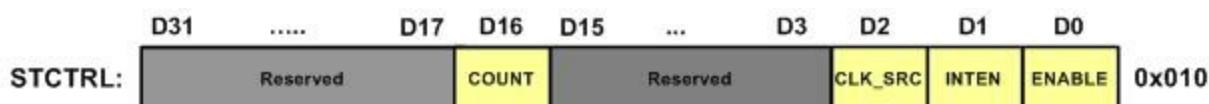
Figure 5-7: System Tick Timer Internal Structure

The down counter is named as *STCURRENT* (SysTick->VAL) in TI ARM products. The counter can receive clock from 2 different sources: the *System clock* (the clock which the CPU and all the peripherals work with it) or the external clock provided to the *PIOOSC* pin. The clock source is chosen using the *CLK\_SRC* bit of *STCTRL* (SysTick->CTRL) register. The clock is ANDed with the *ENABLE* bit of *STCTRL* register. So, it counts down when the *ENABLE* bit is set. The *STCTRL* register is shown in Figure 5-8.

## SysTick Registers

Next, we will describe the SysTick registers. There are three registers in the SysTick module: SysTick Control and Status register, SysTick Reload Value register, and SysTick Current Value register.

The *STCTRL* (SysTick Control and Status) register is located at 0xE000E010. We use it to start the SysTick counter among other things.



bit	Name	Description
0	ENABLE	Enable (0: the counter is disabled, 1: enables SysTick to begin counting down)
1	INTEN	Interrupt Enable 0: Interrupt generation is disabled, 1: when SysTick counts to 0 an interrupt is generated
	CLK_SRC	Clock Source 0: Precision internal oscillator (PIOOSC) divided by 4 1: System clock
	COUNT	Count Flag 0: the SysTick has not counted down to zero since the last time this bit was read 1: the SysTick has counted down to zero Note: this flag is cleared by reading the STCTRL or writing to CTCURRENT register.

Figure 5-8: STCTRL (System Tick Control)

ENABLE (D0): enables or disables the counter. When the *ENABLE* bit is set the counter initializes the STCURRENT with the value of the STRELOAD register and it counts down until it reaches to zero. Then, in the next clock, it underflows which sets the COUNT Flag to high and the counter reloads the STCURRENT with the value of the STRELOAD register and then the process is repeated. See the following figure.

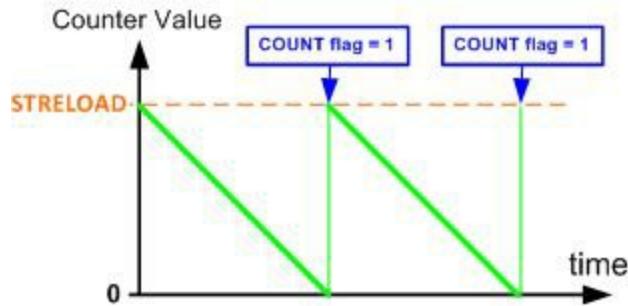


Figure 5-9: System Tick Counting

INTEN (Interrupt Enable, D1). If INTEN=1, an interrupt occurs when the COUNT flag is set. See Chapter 6.

CLK\_SRC (Clock Source D2) We have the choice of clock coming from System clock or Precision Internal Oscillator (PIOSC). If CLK\_SRC=0 then the clock comes from PIOSC/4. If CLK\_SRC=1, then the system clock provides the clock source to SysTick down counter.

COUNT (D16). Counter counts down from the initial value and when it reaches 0, in the next clock it underflows and the COUNT flag is set high. See Figure 5-9. The flag remains high until it is cleared by software. **The flag can be cleared by reading the STCTRL register or writing to the CTCURRENT register.**

### SysTick Reload Value Register (STRELOAD), offset 0x014

The STRELOAD (SysTick Reload Value) register is located at 0xE000E014. This is used to program the start value of SysTick down counter, the STCURRENT register. The STRELOAD should contain the value  $N - 1$  for the COUNT to fire every  $N$  clock cycles because the counter counts down to 0. For example, if we need 1000 clocks of interval, then we make STRELOAD = 999. Although this is a 32-bit register, only the lower 24 bits are used. That means the highest value that can be loaded into this register is 0xFFFFF. See Figures 5-7 and 5-10.



Figure 5-10: STRELOAD vs. STCURRENT

Program 5-1 loads the initial value to the maximum and dumps the current value of the SysTick on LEDs of PORTF as it counts down. On TI ARM LaunchPad, we cannot see the lower 16

bits since it is counting down every 62.5 nsec (1/16MHz=62.5). It takes the lower 16-bits  $65536 \times 62.5\text{ns} = 0.4096$  ms to overflow to the upper 8-bits. That is the reason we shift it right 20 times.

### Program 5-1: Monitoring the value of STCURRENT on LEDs

```
/* p5_1.c */
/* This program lets the SysTick counter run freely and dumps the counter values to the tri-color LEDs continuously.
   The counter value is shifted 20 places to the right so that the changes of LEDs will be slow enough to be visible. */

#include "TM4C123GH6PM.h"

int main (void)
{
    int x;
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2 1) as output */
    GPIOF->DIR = 0x0E;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN = 0x0E;

    /* Configure SysTick */
    SysTick->LOAD = 0xFFFFFFF;      /* reload reg. with max value */
    SysTick->CTRL = 5;            /* enable it, no interrupt, use system clock */

    while (1)
    {
        x = SysTick->VAL;        /* read current value of down counter */
        x = x >> 20;           /* shift right to slow down the rate */
        GPIOF->DATA = x;        /* dump it to the LEDs */
    }
}

/* This function is called by the startup assembly code to perform system specific initialization tasks. */
void SystemInit(void)
{
    __disable_irq();          /* disable all IRQs */

    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}
```

See the following examples.

### Example 5-1

In an ARM microcontroller system clock = 16 MHz. Calculate the delay which is made by the following function.

```
void delay()
{
```

```

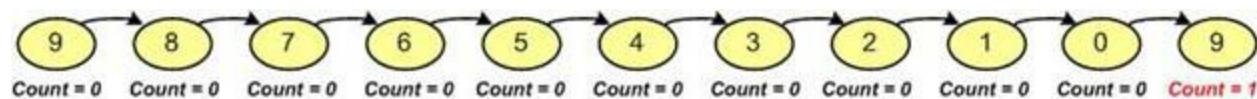
SysTick->LOAD = 9;
SysTick->CTRL = 5; /*Enable the timer and choose system clock as the clock source */

while((SysTick->CTRL &0x10000) == 0) /*wait until the Count flag is set */
{
}
SysTick->CTRL = 0; /*Stop the timer (Enable = 0) */
}

```

### Solution:

The timer is initialized with 9. So, it goes through the following 10 stages:



Since the system clock is chosen as the clock source, each clock lasts  $\frac{1}{\text{sysclk}} = \frac{1}{16\text{MHz}} = 0.0625\mu\text{s}$ .

So, the program makes a delay of  $1.0 \times 0.0625\mu\text{s} = 0.625\mu\text{s} = 625\text{ns}$ .

**Note:** the function call and the instructions execution take a few clock cycles as well. If you want to calculate the exact amount of delay, you should include this overhead, as well. But, in this book we do not consider it since most of the time it is negligible.

---

### Example 5-2

In an ARM microcontroller the system tick is chosen as the clock source for the System tick timer. Calculate the delay which is made by the timer if the STRELOAD register is loaded with N.

### Solution:

The timer is initialized with N. So, it goes through N+1 stages.

Since the system tick is chosen as the clock source, each clock lasts  $1/\text{clk}$

So, the program makes a delay of  $(N + 1) \times (1/\text{clk}) = (N + 1)/\text{clk}$ .

---

### Example 5-3

Using the System Tick timer, write a function that makes a delay of 1ms. Assume sysclk = 16MHz.

### Solution:

$$(N+1) = \text{delay} \times \text{sysclk} = 1\text{ms} \times 16 \text{ MHz} = 16,000 \Rightarrow N = 16,000 - 1 = 15,999$$

```
void delay(void)
{
    SysTick->LOAD = 15999;
    SysTick->CTRL = 0x5; /*Enable the timer and choose cclk as the clock source */

    while((SysTick->CTRL & 0x10000) == 0) /*wait until the Count flag is set */
    {}
    SysTick->CTRL = 0; /*Stop the timer (Enable = 0)*/
}
```

The Program 5-2 uses the SysTick to toggle the PF1 every 1 second. We need the RELOAD value of 16,000,000 since  $1\text{sec} / 62.5\text{nsec} = 16,000,000$ . We assume the CPU clock is 16MHz ( $1/16\text{MHz} = 62.5\text{ns}$ ). Notice, every 16,000,000 clocks the down counter reaches 0, and COUNT flag is raised. Then the RELOAD register is loaded with 16,000,000 automatically. Upon reloading, it clears the COUNT flag and the process starts all over.

### Program 5-2: Toggle PF1 LED every 1 second using the SysTick Counter

```
/* p5_2.c: Toggle PF1 LED every 1second using the SysTick Counter */

/* This program sets up the SysTick to set the COUNT flag at 1 Hz. The system clock is
running at 16 MHz.
1sec / 62.5ns = 16,000,000 for RELOAD register.
The program then polls the flag and toggles the red LED PORTF1 every time COUNT flag
is set.
The COUNT flag is cleared when the STCTRL register is read. */

#include "TM4C123GH6PM.h"

int main (void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2 1) as output */
    GPIOF->DIR = 0x0E;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN = 0x0E;

    /* Configure SysTick */
    SysTick->LOAD = 16000000 - 1; /* reload with number of clocks per second */
    SysTick->CTRL = 5; /* enable it, no interrupt, use system clock */

    while (1)
    {
        if (SysTick->CTRL & 0x10000) /* if COUNT (D16 of CTRL reg.) flag is set */
            GPIOF->DATA ^= 2; /* toggle PORTF1 red LED */
    }
}
```

```

}
/* This function is called by the startup assembly code to perform system specific
initialization tasks.*/
void SystemInit(void)
{
    __disable_irq();      /* disable all IRQs */

    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}

```

### Program 5-3: Making delays using SysTick

```

/* p5_3.c: Count up the LEDs using the SysTick Counter */

/* This program sets up the SysTick to set the COUNT flag at 2 Hz. The system clock is
running at 16 MHz.
0.5sec / 62.5ns = 8,000,000 for RELOAD register since 1 / 0.5sec = 2Hz
The program then polls the flag and counts up a software counter every time COUNT flag
is set. The counter value is written to the tri-color LEDs.
The COUNT flag (D16 of CTRL reg.) is cleared when the STCTRL register is read. */

#include "TM4C123GH6PM.h"

int main (void)
{
    int myCount = 0;

    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, PF2, and PF1) as output */
    GPIOF->DIR = 0x0E;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN = 0x0E;

    /* Configure SysTick */
    SysTick->LOAD = 8000000-1;    /* reload with number of clocks per half second */
    SysTick->CTRL = 5;          /* enable it, no interrupt, use system clock */

    while (1)
    {
        if (SysTick->CTRL & 0x10000)    /* if COUNT flag (D16 of CTRL reg.) is set */
        {
            myCount++;
            GPIOF->DATA = myCount;    /* write the count to LEDs */
        }
    }

    /* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)

```

```
{  
    __disable_irq();      /* disable all IRQs */  
  
    /* Grant coprocessor access */  
    /* This is required since TM4C123G has a floating point coprocessor */  
    SCB->CPACR |= 0x00F00000;  
}
```

The System Tick Timer has a very simple structure and is the same across all the ARM Cortex chips regardless of who makes them. In contrast, TI has some more advanced and complex timers which are covered in the next section.

## Review Questions

1. True or false. The highest number we can place in RELOAD register is \_\_\_\_\_.
2. Assume CPU frequency of 16MHz. Find the value for RELOAD register if we want 5 msec elapsed time.
3. The SysTick is \_\_\_\_-bit wide.
4. Which bit of STCTRL is used to enable the SysTick.
5. The SysTick is (down or up) counter.

## Section 5.2: Generating delays using TI timers

In this section we examine the registers for TI ARM Tiva device timers with some examples showing how to create time delay using one-shots and periodic modes.

### Timer Registers

In TI Tiva microcontrollers, the timers are called *General-Purpose Timer Module (GPTM)*. There are 12 Timer Blocks in the TI Tiva TM4C123G, 6 of them are 16/32-bit timers and the other 6 are 32/64-bit.

The 16/32-bit timer blocks are designated as Timer 0, Timer 1, ..., and Timer 5. The following shows the base addresses for the 16/32-bit Timer blocks:

- 16/32-bit Timer 0 base: 0x4003.0000
- 16/32-bit Timer 1 base: 0x4003.1000
- 16/32-bit Timer 2 base: 0x4003.2000
- 16/32-bit Timer 3 base: 0x4003.3000
- 16/32-bit Timer 4 base: 0x4003.4000
- 16/32-bit Timer 5 base: 0x4003.5000

### TimerA and TimerB

Each of the Timer blocks contains two timers. They are called *TimerA* and *TimerB*. These two timers A and B can work independent of each other as two 16-bit timers or together as one single 32-bit timer. TimerA and TimerB each contains a counter. When the clock is fed to them, they keep counting up (or counting down). We can read their contents as they count and we can load a new value in them. Next, we examine Timer A and give some examples. The discussion about TimerA also applies equally to TimerB. First, we must remember that we need to enable the clock to the Timers before they can be used. This is done with the RCGC Timer register, as shown below:

D31	.....	D6	D5	D4	D3	D2	D1	D0	
RCGCTIMER:	Reserved	R5	R4	R3	R2	R1	R0		0x604
Type	RO	RO	R/W	R/W	R/W	R/W	R/W	R/W	

bit	Name	Description
0	R0	Timer 0 clock control (0: clock disabled, 1: clock enabled)
1	R1	Timer 1 clock control (0: clock disabled, 1: clock enabled)
2	R2	Timer 2 clock control (0: clock disabled, 1: clock enabled)
3	R3	Timer 3 clock enable (0: clock disabled, 1: clock enabled)
4	R4	Timer 4 clock enable (0: clock disabled, 1: clock enabled)
5	R5	Timer 5 clock enable (0: clock disabled, 1: clock enabled)

Figure 5-11: RCGCTimer (Timer Run Mode Clock Gating Control)

The RCGCTIMER is part of the System Control registers. Just like GPIO and UART, we must enable the clock to Timer0 - Timer5 before we can use them. Notice, in RCGCTIMER registers, bit D0 is for Timer0 block, bit D1 is for Timer1 block, and so on.

### GPTMTnV (GPTM Timer Value)

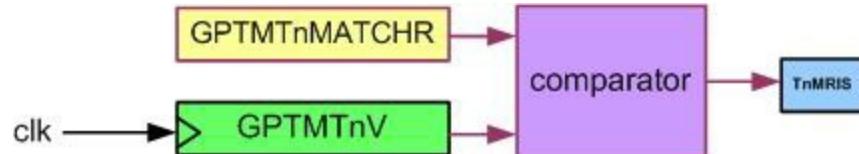
GPTMTAV and GPTMTBV are two 16-bit up/down counter registers. When a timer is in 16-

bit mode, they work as 2 separate counters. When the timer is in 32-bit mode, they are cascaded to form a 32-bit counter.

## GPTM Timer A Match Register (GPTMTAMATCHR)

When TimerA keeps counting it is compared with the contents of this register. Whenever the contents of free-running TimerA counter and Timer A Match register are equal, the TAMRIS Flag goes up indicating there is a match. The D4 of GPTMRIS register belongs to this flag.

Although the Timer A Match is a 32-bit register, only the lower 16 bits are used in the 16-bit configuration selection. See Figure 5-12 and Figure 5-13.



**Note:** In the above figure, n is A for TimerA and B for TimerB.  
For example GPTMTnV is GPTMTAV in TimerA.

Figure 5-12: The Relation Between TnR, TnMATCHR, and TnMRIS

D31	.....	D17	D16
GPTMRIS:	Reserved		WUERIS
Type	RO	RO	R/W
D15	D14	D13	D12
Reserved	TBMRIS	CBERIS	CBMRIS
RO	RO	RO	RO
D11	D10	D9	D8
		TBTORIS	
RO	R/W	R/W	R/W
D7	D6	D5	D4
Reserved	TAMRIS	RTCRIS	CAERIS
RO	R/W	R/W	R/W
D3	D2	D1	D0
		CAMRIS	TATORIS
		R/W	R/W
16	WUERIS	32/64-Bit Wide GPTM Write Update Error Raw Interrupt Status	0x01C

bit	Name	Description
0	TATORIS	Timer A Time-out Raw Interrupt (0: not occurred, 1: occurred)
1	CAMRIS	Timer A Capture Mode Match Raw Interrupt (0: not occurred, 1: occurred)
2	CAERIS	Timer A Capture Mode Event Raw Interrupt (0: not occurred, 1: occurred)
3	RTCRIS	RTC Raw Interrupt (0: not occurred, 1: occurred)
4	TAMRIS	Timer A Match Raw Interrupt
8	TBTORIS	Timer B Time-out Raw Interrupt (0: not occurred, 1: occurred)
9	CBMRIS	Timer B Capture Mode Match Raw Interrupt (0: not occurred, 1: occurred)
10	CBERIS	Timer B Capture Mode Event Raw Interrupt (0: not occurred, 1: occurred)
11	TBMRIS	Timer B Match Raw Interrupt
16	WUERIS	32/64-Bit Wide GPTM Write Update Error Raw Interrupt Status

Figure 5-13: GPTMRIS

### Note

In TI ARM Tiva microcontroller, all the timer registers begin with GPTM. So, for simplicity just consider the letters which come after GPTM. For example, consider GPTMTAV as TAV.

## GPTM Control register

During the initialization of the Timers we must disable them. Modifying the configurations of a running timer may cause unpredictable results. We use bit D0 of GPTMCTL (GPTM Control) register to disable or enable the TimerA. See Figure 5-14. In the same way, D8 of GPTMCTL is used to enable or disable TimerB. See Figure 5-15.

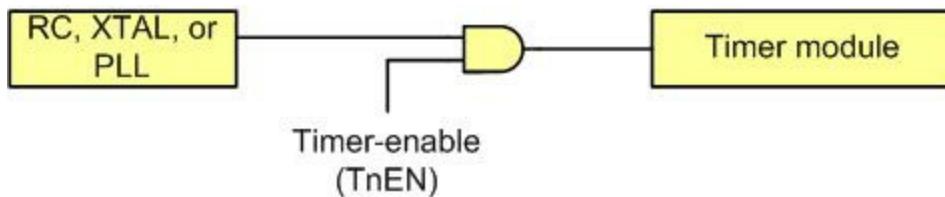


Figure 5-14: Timer Enable (TnEN)

	D31	.....	D15	D14	D13	D12	D11	D10	D9	D8
GPTMCTL:				TBPWML	TBOTE			TBEVENT	TBSTALL	TBEN
Type	RO		RO	R/W	R/W	RO	R/W	R/W	RO	R/W
	D7	D6	D5	D4	D3	D2	D1	D0		
	Reserved	TAPWML	TAOTE	RTCEN	TAEVENT	TASTALL	TAEN			0x00C
	RO	R/W	R/W	R/W	R/W	R/W	R/W			

bit	Name	Description
0	TAEN	Timer A Enable (0: disabled, 1: enabled)
1	TASTALL	Timer A Stall (useful while debugging) 0: Timer A continues counting if the CPU is halted by the debugger, 1: Timer A stalls (stops counting) while the CPU is halted by the debugger. While tracing the code, when you pause on a line the timer stops counting if the bit is set. This facilitates us to see exactly what really happens.
2 & 3	TAEVENT	Timer A Event mode (0: positive edge, 1: negative edge, 2: reserved, 3: both edges)
4	RTCEN	RTC Stall Enable (useful while debugging) 0: RTC stalls (stops counting) while the CPU is halted by the debugger, 1: RTC continues counting if the CPU is halted by the debugger.
5	TAOTE	Timer A Output Trigger Enable (0: ADC trigger disabled, 1: ADC trigger enabled)
6	TAPWML	Timer A Wait-on-Trigger 0: It begins counting when the timer is enabled, 1: It waits for the input to be triggered.

Note: Timer B is configured using bits 8 to 14 in the same way.

Figure 5-15: GPTM Control Register

## GPTM Configuration Register

To configure Timer A as 16- or 32-bit, we must use GPTMCFG (GPTM Configuration) register. Bits D2, D1, and D0 are used to select either 16- or 32-bit option. To use the 16-bit option we need to have D2:D1:D0=0x4. As mentioned above, in 16-bit mode, TimerA and TimerB make two separate timers which work independently.

	D31	.....	D3	D2	D1	D0	
GPTMCFG:		Reserved			GPTMCFG		0x000
Type	RO		RO	R/W	R/W	R/W	
	D2:D1:D0	Mode					
	000	32-bit mode					
	001	RTC counter					
	100	16-bit mode					

Figure 5-16: GPTM Configuration Register

## TimerA Mode selection register

The mode selection such as periodic count up/down selection for Timer A is done with GPTM Timer A Mode (GPTMTAMR) register. See Figure 5-17.

GPTMTAMR:	D31	...	D12	D11	D10	D9	D8
	Reserved			TAPLO	TAMRSU	TAPWMIE	TAILD
	RO	RO	RO	R/W	R/W	R/W	R/W
D7	D6	D5	D4	D3	D2	D1	D0
TASNAPS	TAWOT	TAMIE	TACDIR	TAAMS	TACMR	TAMR	
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

0x004

bit	Name	Description
0 & 1	TAMR	Timer A Mode (See Table 5-1)
2	TACMR	Timer A Capture Mode (0: Edge Count, 1: Edge Time)
3	TAAMS	Timer A Alternate Mode Select (0: Capture or compare mode, 1: PWM mode)
4	TACDIR	Timer A Count Direction (0: Count down, 1: Count up)
5	TAMIE	Timer A Match Interrupt Enable (0: the match interrupt is disabled, 1: enabled)
6	TAWOT	Timer A Wait-on-Trigger (0: It begins counting when enabled, 1: waits for trigger)
7	TASNAPS	Timer A Snap-Shot Mode (0: Snap-shot is disabled)
8	TAILD	Timer A Interval Load Write
9	TAPWMIE	Timer A PWM Interrupt Enable (0: Capture event interrupt is disabled, 1: Enabled)
10	TAMRSU	Timer A Match Register Update
11	TAPLO	Timer A PWM Legacy Operation

Figure 5-17: GPTMTAMR (GPTM Timer A Mode)

The mode selection is done with D1:D0 bits of GPTMTAMR, as shown below:

Mode	D1	D0	Mode Name
0	0	0	Reserved
1	0	1	one-shot Mode
2	1	0	Periodic Mode
3	1	1	Capture Mode

Table 5-1: TAMR Bits of GPTMTAMR

The direction Count is done with D4 (TACDIR). Upon Reset, the default is down counter. By making D4=1, TimerA counts up.

## GPTM Timer n Interval Load (GPTMTnILR)

When the timer is counting down (the timer counts down if the TACDIR bit of the GPTMTAMR register is 0), the timer counter begins counting from GPTMTnILR and goes down until it reaches zero. Then, the timer counter is reloaded with the value from GPTMTnILR and the TnTORIS flag of the GPTMRIS register is set. See Figure 5-18.

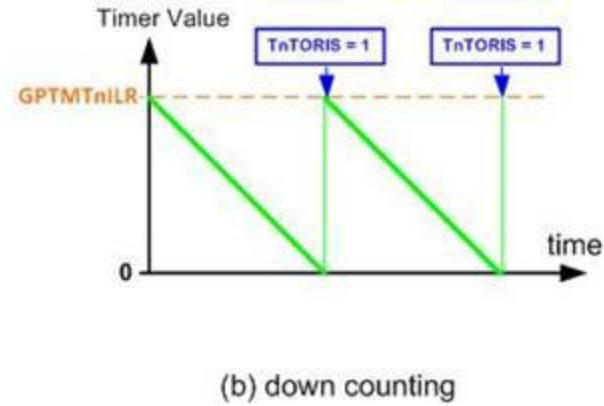
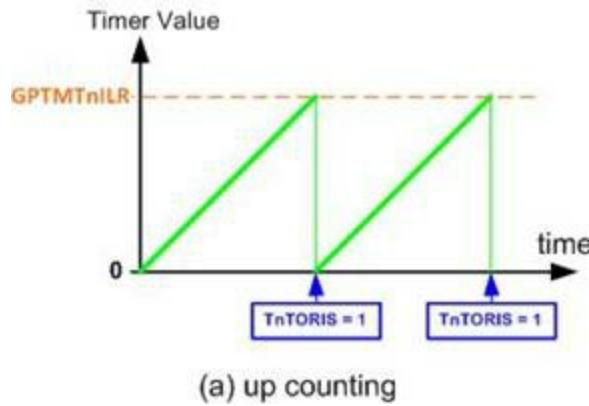


Figure 5-18: The role of GPTMTnR

When the timer is counting up, the timer counter begins counting from 0 and goes up until it reaches the GPTMTnILR value. Then, the timer counter is cleared to zero and the TnTORIS flag of the GPTMRIS register is set.

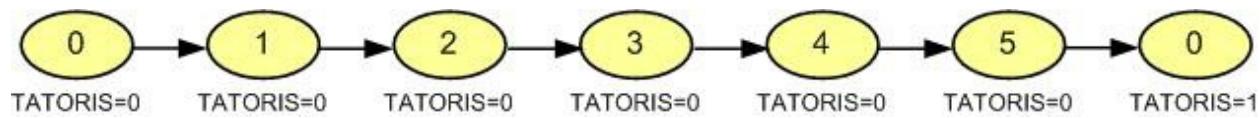
Upon reset, all bits of the GPTMTnILR register are initialized to 1s which makes the biggest value and has no effect on the timer counting. But the software can change the value of GPTMTnILR. The smaller values for the register leads the timer timeout faster and the TnTORIS flag sets sooner. In other words, changes of delay can be made by setting the GPTMTnILR register and monitoring the TnTORIS flag. See Example 5-4.

#### Example 5-4

Assume GPTMTAILR = 5 and Timer A is counting up. Explain when the TATORIS flag is raised.

#### Solution:

The timer counts up with the passing of each clock provided by the crystal oscillator. As the timer counts up, it goes through the states of 0, 1, 2, 3, 4, and 5. One more clock rolls it to 0, raising the TATORIS.



#### Periodic mode vs. one shot mode

As discussed in Table 5-1, the timer can be in 4 different modes including the periodic and one shot modes. In periodic mode the timer continues counting after each timeout as discussed above. But in one shot mode, the timer stops counting after timeout is reached. For example, when it is in up counting and one shot modes, it counts from 0 to GPTMTnILR and then goes to zero just once and then the TnEN bit of GPTMCTL is cleared causing the timer to stop. See Figure 5-19.

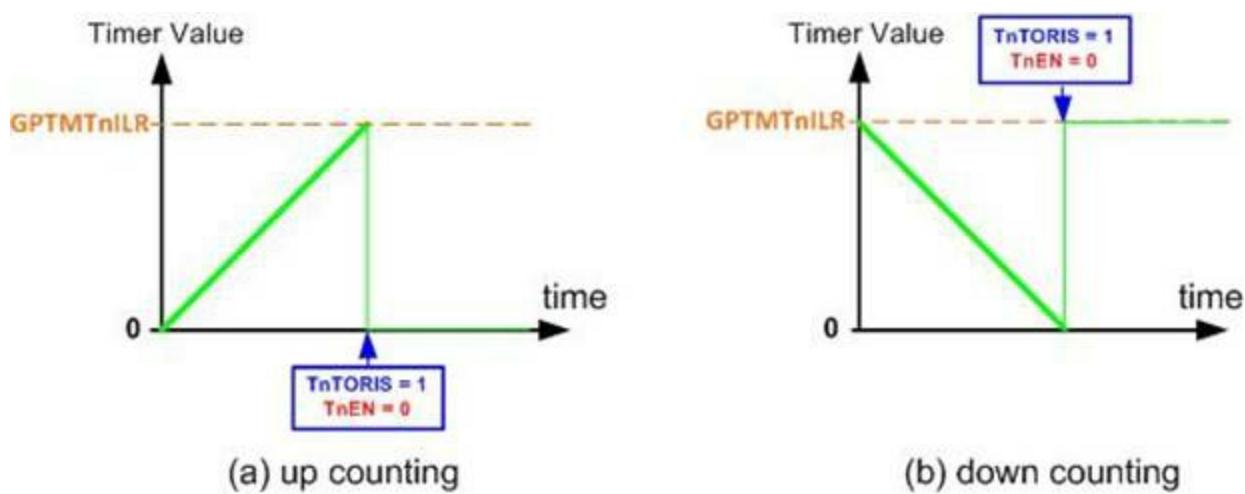


Figure 5-19: Counting in One Shot Mode

The periodic mode can be used to generate periodic interrupts, as we will see in the next chapter. For now, we use the one-shot mode to create a time delay. In Chapter 2 we used simple loop to create time delay. In this section, we use TimerA to create the delay. Program 5-4 toggles the LED of PF1 pin with a delay. The LED will be on for 2 ms and off for 3 ms. You will need to use an oscilloscope to verify the output. This is the repeat of Example 2-3 except it uses 16-bit TimerA of Timer 0 to create the delay. The CPU frequency of 16 MHz means clock period of  $1/16\text{MHz} = 62.5\text{ns}$  (nanosecond) being fed to TimerA. To get 1msec delay we need  $1\text{msec}/62.5\text{ns} = 16,000$  clocks.

The steps to program the timer for one-shot mode are:

- 1) enable the clock to Timer0 block
- 2) disable timer while the configuration is being modified
- 3) select 16-bit mode
- 4) select one-shot mode and down-counter mode
- 5) set interval load register value
- 6) clear timeout flag
- 7) enable timer
- 8) wait for timeout flag to set

#### Program 5-4: Toggling PF1 LED using One-shot Mode of TimerA(of Timer0) Delay

```
/* p5_4.c: This program demonstrates the use of TimerA of Timer0 block to do a delay of
the multiple of milliseconds. Because 16-bit mode is used, it will only work up to 4
ms. */

#include "TM4C123GH6PM.h"

void timer0A_delayMs(int ttime);
void delayMs(int n);

int main (void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
```

```

/* enable the GPIO pins for the LED (PF3, 2 1) as output */
GPIOF->DIR = 0x0E;
/* enable the GPIO pins for digital function */
GPIOF->DEN = 0x0E;

while(1)
{
    GPIOF->DATA = 2;          /* turn on red LED */
    timer0A_delayMs(2);      /* Timer A msec delay */
    GPIOF->DATA = 0;          /* turn off red LED */
    delayMs(3);              /* use old delay function */
}

/* millisecond delay using one-shot mode */
void timer0A_delayMs(int ttime)
{
    SYSCTL->RCGCTIMER |= 1;    /* enable clock to Timer Block 0 */
    TIMER0->CTL = 0;           /* disable Timer before initialization */
    TIMER0->CFG = 0x04;        /* 16-bit option */
    TIMER0->TAMR = 0x01;       /* one-shot mode and down-counter */
    TIMER0->TAILR = 16000 * ttime - 1; /* Timer A interval load value register*/
    TIMER0->ICR = 0x1;         /* clear the TimerA timeout flag*/
    TIMER0->CTL |= 0x01;       /* enable Timer A after initialization*/
    while ((TIMER0->RIS & 0x1) == 0) ; /* wait for TimerA timeout flag to set*/
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}

```

Because 16-bit mode was used in Program 5-4, the interval load register is limited to 4 ms. For longer delay, we will need to incorporate prescaler or use 32-bit mode. These will be discussed later in the chapter. Another approach for longer delay is to program the interval load register for timeout of a short interval and wait for timeout multiple times. In this approach, the overhead time of setting up the timer for one-shot mode will accumulate and causes inaccuracy. It would be better to use periodic mode to avoid the inaccuracy. In program 5-5, we will set up the timer for 1 ms timeout in periodic mode and do 500 timeouts to achieve a one half second delay. The LED should blink at 1 Hz.

```

/* p5_5.c: This program demonstrates the use of TimerA of Timer0 block to do a delay of
the multiple of milliseconds using periodic mode. */

#include "TM4C123GH6PM.h"

void timer0A_delayMs(int ttime);

int main (void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2 1) as output */
    GPIOF->DIR = 0x0E;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN = 0x0E;

    while(1)
    {
        GPIOF->DATA = 2;          /* turn on red LED */
        timer0A_delayMs(500);     /* TimerA 500 msec delay */
        GPIOF->DATA = 0;          /* turn off red LED */
        timer0A_delayMs(500);     /* TimerA 500 msec delay */
    }
}

/* multiple of millisecond delay using periodic mode */
void timer0A_delayMs(int ttime)
{
    int i;
    SYSCTL->RCGCTIMER |= 1;      /* enable clock to Timer Block 0 */

    TIMER0->CTL = 0;             /* disable Timer before initialization */
    TIMER0->CFG = 0x04;           /* 16-bit option */
    TIMER0->TAMR = 0x02;          /* periodic mode and down-counter */
    TIMER0->TAILR = 16000 - 1;    /* Timer A interval load value register */
    TIMER0->ICR = 0x1;            /* clear the TimerA timeout flag */
    TIMER0->CTL |= 0x01;          /* enable Timer A after initialization */

    for(i = 0; i < ttime; i++)
    {
        while ((TIMER0->RIS & 0x1) == 0) ;      /* wait for TimerA timeout flag */
        TIMER0->ICR = 0x1;                      /* clear the TimerA timeout flag */
    }
}

```

Program 5-6 shows the TimerB version of Program 5-5.

#### Program 5-6: Delay function using TimerB (of Timer0)

```

/* p5_6.c: multiple of millisecond delay using periodic mode */
/* Using Timer0B */

void timer0B_delayMs(int ttime)
{

```

```

int i;
SYSCTL->RCGCTIMER |= 1;      /* enable clock to Timer Block 0 */

TIMER0->CTL = 0;            /* disable Timer before initialization */
TIMER0->CFG = 0x04;          /* 16-bit option */
TIMER0->TBMR = 0x02;          /* periodic mode and down-counter */
TIMER0->TBILR = 16000 - 1;    /* TimerB interval load value reg */
TIMER0->ICR = 0x100;          /* clear the TimerB timeout flag */
TIMER0->CTL |= 0x0100;        /* enable TimerB after initialization */

for(i = 0; i < ttime; i++)
{
    while ((TIMER0->RIS & 0x100) == 0); /* wait for TimerB timeout flag */
    TIMER0->ICR = 0x100; /* clear the TimerB timeout flag */
}
}

```

TimerA and TimerB runs independently. We may have both delay functions in the same program and use them to flash different LEDs. Even though the timers are independent, the software is not. When the program calls the delay using TimerA, it is very difficult to monitor TimerB at the same time. One solution to that is to use timer interrupt, which we will discuss in the next chapter.

## Prescaler register for Timer A

In Program 5-5, the largest time delay that we can create is  $65535 \times (1 / 16\text{MHz}) = 65535 \times 62.5 \text{nsec} = 4.096 \text{ msec}$ . One way to create a longer time delay is to use the prescaler register. TimerA in 16-bit mode has an 8-bit prescaler register whose value can go from 0x00 to 0xFF. The 8-bit prescaler extends the 16-bit timer to 24-bit. The prescaler register allows system frequency to be divided by a value between 1 and 256 before it is fed to the TimerA. **Note the prescaler can yield proper division only when the timer is configured as a down counter.** As shown in Figure 5-20, the clock is divided by  $\text{GPTMTnPR} + 1$ . Example 5-5 shows the calculation.

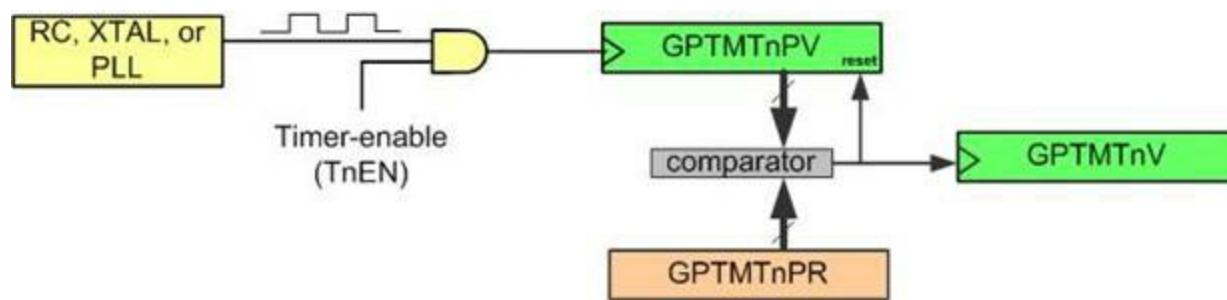


Figure 5-20: Prescaler

## Example 5-5

For CPU Freq=16MHz calculate the largest delay size using a) 16-bit TimerA without prescaler and b) 16-bit TimerA with prescaler.

**Solution:**

$1/16\text{MHz} = 62.5 \text{nsec}$  is period of clock pulses fed to CPU.

- (a)  $65,536 \times 62.5 \text{ nsec} = 4.096 \text{ msec}$  for TimerA 16-bit option .  
(b)  $65,536 \times 256 \times 62.5\text{ns}=1.0485 \text{ second}$  for TimerA 16-bit option with the Prescaler.
- 

### Example 5-6

Use prescaler register for TimerA to create an 1-second delay.

#### Solution:

Using prescaler of 250 gives us  $16\text{MHz}/250=64000\text{Hz}$ . That means the clock fed to timer A is  $1/64000\text{Hz}=15.625 \text{ msec}$ . To get one second delay, we need  $1\text{sec}/15.625\text{ms}=64000$  for the match register.

---

Program 5-7 is a program for TimerA of Timer 1 to create a 1-sec time delay on PF2 LED. It uses the prescaler.

#### Program 5-7:Toggling PF2 LED every Second

```
/* p5_7.c: timer using prescaler */
/* This program is modified from p5_5.c to incorporate the use of prescaler to create a
longer delay. Also the use of timer has been changed to TimerA of timer Block 1. */

#include "TM4C123GH6PM.h"

void timer1A_delaySec(int ttime);

int main (void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2, and 1) as output */
    GPIOF->DIR = 0x0E;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN = 0x0E;

    while(1)
    {
        GPIOF->DATA = 4;           /* turn on blue LED */
        timer1A_delaySec(1);      /* TimerA 500 msec delay */
        GPIOF->DATA = 0;          /* turn off blue LED */
        timer1A_delaySec(1);      /* TimerA 500 msec delay */
    }
}

/* multiple of second delays using periodic mode and prescaler*/
void timer1A_delaySec(int ttime)
{
```

```

int i;
SYSCTL->RCGCTIMER |= 2;      /* enable clock to Timer Block 1 */

TIMER1->CTL = 0;            /* disable Timer before initialization */
TIMER1->CFG = 0x04;          /* 16-bit option */
TIMER1->TAMR = 0x02;         /* periodic mode and down-counter */
TIMER1->TAILR = 64000 - 1;    /* TimerA interval load value reg */
TIMER1->TAPR = 250 - 1;      /* TimerA Prescaler 16MHz/250=64000Hz */
TIMER1->ICR = 0x1;           /* clear the TimerA timeout flag */
TIMER1->CTL |= 0x01;         /* enable Timer A after initialization */

for(i = 0; i < ttime; i++)
{
    while ((TIMER1->RIS & 0x1) == 0) ;      /* wait for TimerA timeout flag */
    TIMER1->ICR = 0x1;           /* clear the TimerA timeout flag */
}
}

```

## Review Questions

- True or false. A 16/32 bit timer can be used as 16-bit timer or 32-bit timer, but not at the same time.
- We have \_\_\_\_\_ 16/32-bit Timer blocks in TM4C123G and they are designated as \_\_\_\_\_.
- True or false. Each of Timer0, Timer1, and Timer2 has TimerA and TimerB.
- Which register is used to enable the clock to the Timer0?
- In TI ARM Tiva, the timers are referred to as \_\_\_\_\_.

## Section 5.3: Pin Selection for Timers

In the last section, we showed how to use timers to generate time delay. In this and following sections, we will examine the use of timers with the I/O pins. There are five modes for each timer block. They are (a) One-shot/Periodic mode, (b) Real-time clock mode, (c) input edge-time mode, (d) input edge-count mode and (e) pulse width modulation (PWM). The TI refers to them as CCP (Capture Compare PWM). The one-shot/periodic mode was discussed in the previous section.

In real-time clock mode, TimerA and TimerB are concatenated as a counter that counts up once every second. A 32.768 KHz clock is required to be connected to CCP0 pin of that timer. The timer circuit will divide the 32.768 KHz clock to a 1 Hz clock and use it to increment the TimerA counter and TimerB counter. The timer counter is preloaded with a value of 1 when it is put in real-time clock mode. So the value of the counter is the number of seconds since the timer was initialized. It is possible for the program to reload the timer counters with new values. We will leave the further exploration of this topic to the readers.

We will discuss the input edge modes in the next two sections. The TI Tiva TM4C123G has dedicated PWM modules that will be covered in Chapter 11 so we skip the PWM mode of the timer.

The rest of the timer modes all involve some of the I/O pins. Next, we will discuss the use of I/O pins with the timer.

### TimerA and TimerB CCP pins

There are TimerA and TimerB for each of the Timer block 0 to 5. There are one or two designated pins for each of the TimerA and TimerB. For example, TimerA of Timer block 4 is connected internally to pin PC0 pin of PORTC and TimerB of Timer block 4 is connected to pin PC1 of PORTC. Notice, TimerA pins are also called CCP0 and TimerB pins are called CCP1. Some of the timers have option to be connected to one of the two pins. For example, TimerA of Timer1 (T1CCP0) can use PB0 or PF2 pins. Table 5-2 shows the pin designations for 16/32-bit Timer block 0 to 5.

	TimerA	Pins	TimerB	Pins
Timer0	T0CCP0	PB6 or PF0	T0CCP1	PB7 or PF1
Timer1	T1CCP0	PB4 or PF2	T1CCP1	PB5 or PF3
Timer2	T2CCP0	PB0 or PF4	T2CCP1	PB1
Timer3	T3CCP0	PB2	T3CCP1	PB3
Timer4	T4CCP0	PC0	T4CCP1	PC1
Timer5	T5CCP0	PC2	T5CCP1	PC3

Table 5-2: the pin designation for 16/32-bit Timer block 0 to 5

### Selecting alternate function for Timers pin

Upon Reset, the GPIOAFSEL register has all 0s meaning the I/O pins are used as simple I/O. To use an alternate function, we first must set the bit in the AFSEL register to 1 for that pin. For example, for the PB6, we need to write 0x40 (01000000 in binary) to GPIO\_PORTB\_AFSL (GPIOB->AFSEL) register. See Tables 5-3 and 5-4. After that, the GPIOCTL register must be

configured for the desired function. To do that, we need to use the information in Table 23-5 of TI Tiva TM4C123GH6PM data sheet. Table 5-3 provides the summary for the Timers pins.

Timer Pin	I/O pin	How to use peripheral function on the pin
T0CCP0	PB6	GPIOB->AFSEL =0x40 (0100 0000 binary)
	PF0	GPIOF->AFSEL =0x01 (0000 0001 binary)
T0CCP1	PB7	GPIOB->AFSEL =0x80 (1000 0000 binary)
	PF1	GPIOF->AFSEL =0x02 (0000 0010 binary)
T1CCP0	PB4	GPIOB->AFSEL =0x10 (0001 0000 binary)
	PF2	GPIOF->AFSEL =0x04 (0000 0100 binary)
T1CCP1	PB5	GPIOB->AFSEL =0x20 (0010 0000 binary)
	PF3	GPIOF->AFSEL =0x08 (0000 1000 binary)
T2CCP0	PB0	GPIOB->AFSEL =0x40 (0000 0001 binary)
	PF4	GPIOF->AFSEL =0x10 (0001 0000 binary)
T2CCP1	PB1	GPIOB->AFSEL =0x02 (0000 0010 binary)
T3CCP0	PB2	GPIOB->AFSEL =0x04 (0000 0100 binary)
T3CCP1	PB3	GPIOB->AFSEL =0x08 (0000 1000 binary)
T4CCP0	PC0	GPIOC->AFSEL =0x01 (0000 0001 binary)
T4CCP1	PC1	GPIOC->AFSEL =0x02 (0000 0010 binary)
T5CCP0	PC2	GPIOC->AFSEL =0x04 (0000 0100 binary)
T5CCP1	PC3	GPIOC->AFSEL =0x08 (0000 1000 binary)
WT0CCP0	PC4	GPIOC->AFSEL =0x10 (0001 0000 binary)
WT0CCP1	PC5	GPIOC->AFSEL =0x20 (0010 0000 binary)
WT1CCP0	PC6	GPIOC->AFSEL =0x40 (0100 0000 binary)
WT1CCP1	PC7	GPIOC->AFSEL =0x80 (1000 0000 binary)
WT2CCP0	PD0	GPIOD->AFSEL =0x01 (0000 0001 binary)
WT2CCP1	PD1	GPIOD->AFSEL =0x02 (0000 0010 binary)
WT3CCP0	PD2	GPIOD->AFSEL =0x04 (0000 0100 binary)
WT3CCP1	PD3	GPIOD->AFSEL =0x08 (0000 1000 binary)
WT4CCP0	PD4	GPIOD->AFSEL =0x10 (0001 0000 binary)
WT4CCP1	PD5	GPIOD->AFSEL =0x20 (0010 0000 binary)
WT5CCP0	PD6	GPIOD->AFSEL =0x40 (0100 0000 binary)
WT5CCP1	PD7	GPIOD->AFSEL =0x80 (1000 0000 binary)

Table 5-3: Timers alternate pin assignment

Timer Pin	I/O Pin	How to select the timer function on the pin
T0CCP0	PB6	GPIOB->PCTL=0x0700 0000
	PF0	GPIOF->PCTL=0x0000 0007
T0CCP1	PB7	GPIOB->PCTL=0x7000 0000
	PF1	GPIOF->PCTL=0x0000 0070
T1CCP0	PB4	GPIOB->PCTL=0x0007 0000
	PF2	GPIOF->PCTL=0x0000 0700
T1CCP1	PB5	GPIOB->PCTL=0x0070 0000
	PF3	GPIOF->PCTL=0x0000 7000
T2CCP0	PB0	GPIOB->PCTL=0x0000 0007
	PF4	GPIOF->PCTL=0x0007 0000
T2CCP1	PB1	GPIOB->PCTL=0x0000 0070
T3CCP0	PB2	GPIOB->PCTL=0x0000 0700

T3CCP1	PB3	GPIOB->PCTL=0x0000 7000
T4CCP0	PC0	GPIOC->PCTL=0x0000 0007
T4CCP1	PC1	GPIOC->PCTL=0x0000 0070
T5CCP0	PC2	GPIOC->PCTL=0x0000 0700
T5CCP1	PC3	GPIOC->PCTL=0x0000 7000
WT0CCP0	PC4	GPIOC->PCTL=0x0007 0000
WT0CCP1	PC5	GPIOC->PCTL=0x0070 0000
WT1CCP0	PC6	GPIOC->PCTL=0x0700 0000
WT1CCP1	PC7	GPIOC->PCTL=0x7000 0000
WT2CCP0	PD0	GPIOD->PCTL=0x0000 0007
WT2CCP1	PD1	GPIOD->PCTL=0x0000 0070
WT3CCP0	PD2	GPIOD->PCTL=0x0000 0700
WT3CCP1	PD3	GPIOD->PCTL=0x0000 7000
WT4CCP0	PD4	GPIOD->PCTL=0x0007 0000
WT4CCP1	PD5	GPIOD->PCTL=0x0070 0000
WT5CCP0	PD6	GPIOD->PCTL=0x0700 0000
WT5CCP1	PD7	GPIOD->PCTL=0x7000 0000

Table 5-4: Timers pin assignment using GPIO\_PCTL (Extracted from Table 23-5 of Tiva data sheet)

See Example 5-7.

### Example 5-7

Write the code to provide the T0CCP1 function on PF1.

**Solution:**

The I/O pin is used as a peripheral function pin by setting the AFSEL register of PORTF and the desired peripheral function is selected by setting the PCTL register of PORTF:

```
GPIOF->AFSEL |= 0x02;
GPIOF->PCTL = 0x00000070;
```

### Review Questions

- True or false. Each of the TimerA and TimerB has its own designated pin.
- True or false. Upon Reset, all the pins are designated as simple I/O.
- True or false. We have a single register for selection the alternate function for all the I/O ports.
- True or false. Each port has its own PCTL register.
- What does pin multiplexing means?

## Section 5.4: Using Timer for input edge-time capturing

### Input edge-time mode

In input edge-time mode, an I/O pin is used to capture the signal transition events. When an event occurs, the content of the timer counter is captured in another register while the counter keeps counting. The program can then read the counter value when the event occurs at a slightly later time.

To configure the timer to input edge time mode the TAMR and TnCMR bits of GPTMTAMR should be set to capture and Time edge mode (TnMR = 3 and TnCMR = 1). See Figure 5-17. In this mode, the timer counter value is stored in the GPTMTnR register whenever the input pin is triggered by an external event. See Figure 5-21.

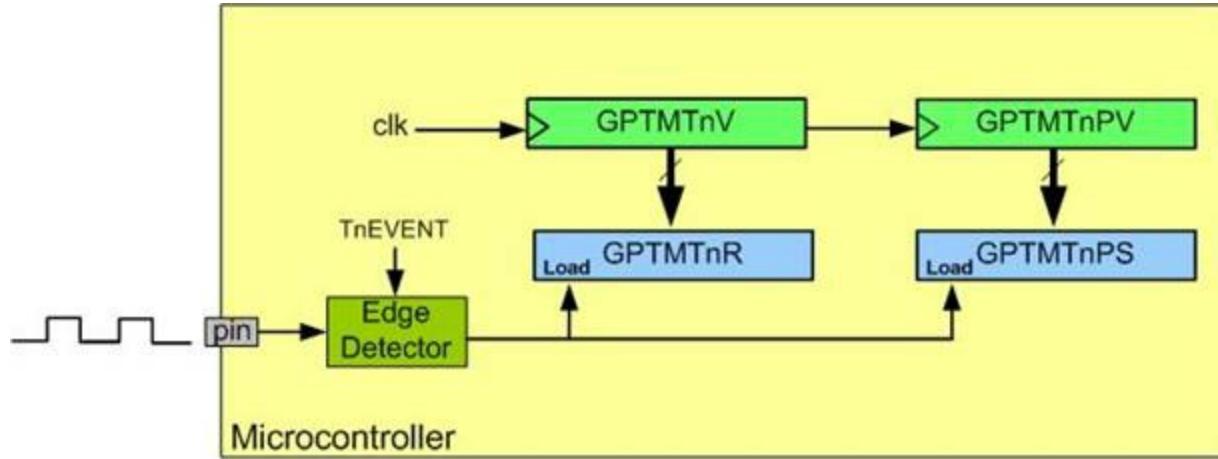


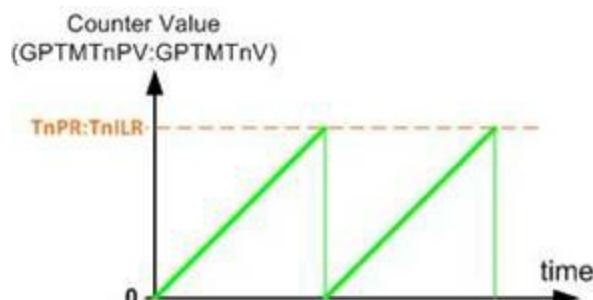
Figure 5-21: Input Edge Time Capturing

The timer can be configured to capture on the falling edge, rising edge, or both. To determine the type of edge that is captured, the TnEVENT bits of the GPTMCTL register should be initialized. See Figure 5-15.

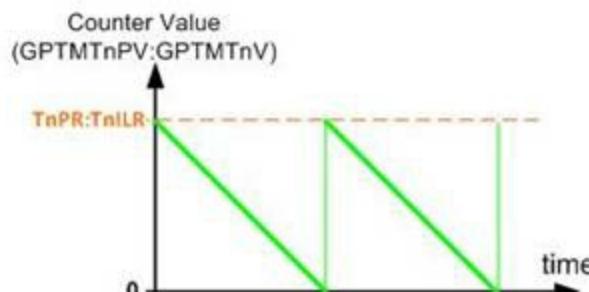
### Timer counting in input edge time mode

In edge time mode GPTMTnV and the optional prescaler are combined to make a 24-bit or 48-bit timer. The timer is 24-bit when 16-bit timers are used. The timer is 48-bit when 32-bit timers are used.

In up counting mode, the GPTMTnV and GPTMTnPv registers are initialized with 0s and they count up until they reach to GPTMTnILR and GPTMTnPR, respectively. Then, they are reset to 0s again.



(a) up counting



(b) down counting

Figure 5-22: Counting in Input Edge-Time Mode

In down counting mode, the GPTMTnV and GPTMTnPv registers are initialized with GPTMTnILR and GPTMTnPv, respectively and they count down until they reach 0. Then, they are reloaded with GPTMTnILR and GPTMTnPv, again.

Notice that capturing has no effect on counting and the timer continues counting when the capturing event takes place.

## Input edge-time mode usages

The input edge time capturing can be used for many applications; e.g. measuring the frequency and pulse width of a signal, recording the arrival time of an event.

As shown in Figure 5-23, to measure the period of a signal we must measure the time between two falling or rising edges. Program 5-8 measures the period of the square wave.

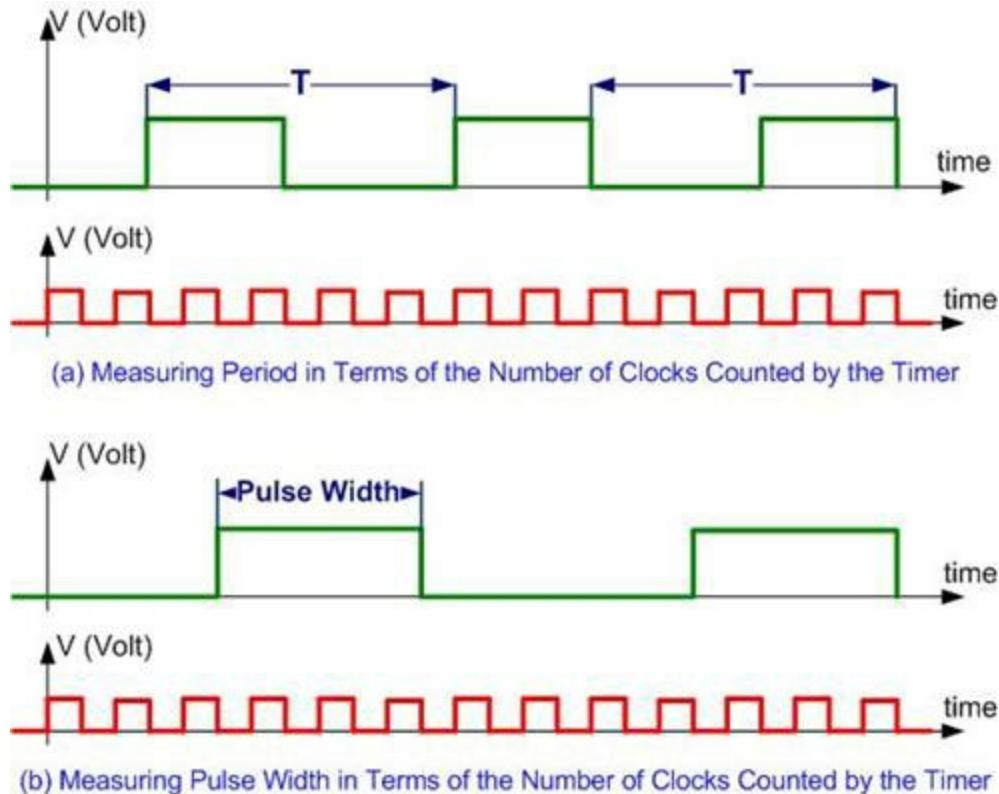


Figure 5-23: Measuring Period and Pulse Width

### Program 5-8 Functions to initialize Timer0A for edge-time capture mode to measure the period of a square input signal

```
/* p5_8.c */
/* square wave signal should be fed to PB6 pin. Make sure it is 3.3 to 5V peak-to-peak.
Initialize Timer0A in edge-time mode to capture rising edges. The input pin of Timer0A
is PB6. See Table 5-2.*/

void Timer0Capture_init(void)
{
    SYSCTL->RCGCTIMER |= 1;      /* enable clock to Timer Block 0 */
    SYSCTL->RCGCGPIO |= 2;       /* enable clock to PORTB */
```

```

GPIOB->DIR &= ~0x40;          /* make PB6 an input pin */
GPIOB->DEN |= 0x40;           /* make PB6 as digital pin */
GPIOB->AFSEL |= 0x40;          /* use PB6 alternate function */
GPIOB->PCTL &= ~0x0F000000;    /* configure PB6 for T0CCP0 */
GPIOB->PCTL |= 0x07000000;

TIMER0->CTL &= ~1;           /* disable timer0A during setup */
TIMER0->CFG = 4;              /* 16-bit timer mode */
TIMER0->TAMR = 0x17;           /* up-count, edge-time, capture mode */
TIMER0->CTL &= ~0x0C;          /* capture the rising edge */
TIMER0->CTL |= 1;             /* enable timer0A */

}

/* This function captures two consecutive rising edges of a periodic signal from Timer
Block 0 Timer A and returns the time difference (the period of the signal). */
int Timer0A_periodCapture(void)
{
    int lastEdge, thisEdge;

    /* capture the first rising edge */
    TIMER0->ICR = 4;             /* clear timer0A capture flag */
    while((TIMER0->RIS & 4) == 0) ; /* wait till captured */
    lastEdge = TIMER0->TAR;       /* save the timestamp */

    /* capture the second rising edge */
    TIMER0->ICR = 4;             /* clear timer0A capture flag */
    while((TIMER0->RIS & 4) == 0) ; /* wait till captured */
    thisEdge = TIMER0->TAR;       /* save the timestamp */

    return (thisEdge - lastEdge) & 0x00FFFFFF; /* return the time difference */
}

```

## Review Questions

- True or false. To capture the input edge time, the timer must be configured to one-shot mode.
- True or false. To measure the frequency of a signal, the time interval between a falling edge and a rising edge are needed.
- True or false. If the time interval between two consecutive falling edges are measured, the frequency of the periodic signal can be calculated.
- True or False. The TI ARM Tiva supports both rising and falling edge detection.
- A Timer must be disabled (before, after) it is initialized.

## Section 5.5: Using Timer As a Counter

As shown in Figure 5-17, a timer works as a counter when the TAMR bits of the GPTMTnMR are configured to capture mode and the TACMR bit is cleared. In this situation, the timer counts whenever the input pin is triggered. See Figure 5-24. The pin can be configured to count on the falling edge, rising edge, or both. To determine the type of edge that is counted, the TnEVENT bits of the GPTMCTL register should be initialized. See Figure 5-16 and 5-17.

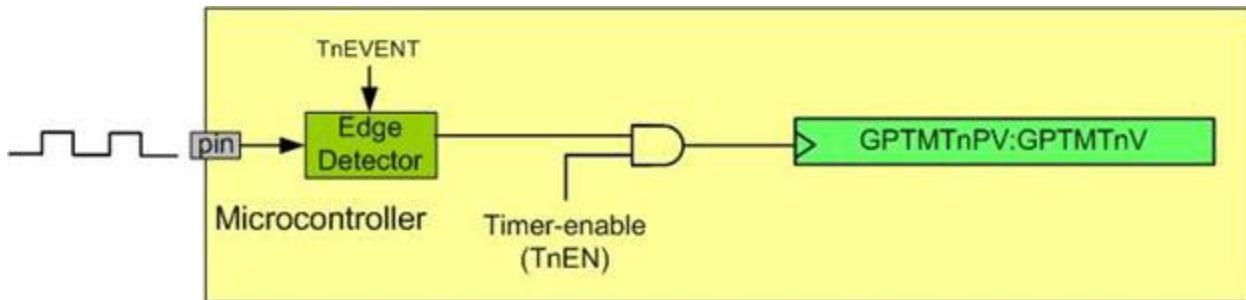


Figure 5-24: Counter Diagram

### Timer counting in input edge count mode

In this mode GPTMTnV and the prescaler are combined to make a 24-bit or 48-bit timer. The timer is 24-bit when TimerA and TimerB are used separately. Otherwise, they make a 48-bit timer.

In up counting mode, the GPTMTnV and GPTMTnPV registers are initialized with 0s and they count up until they reach to GPTMTnMATCHR and GPTMTnPMPR, respectively. Then, they are reloaded with 0s again. See Figure 5-25.

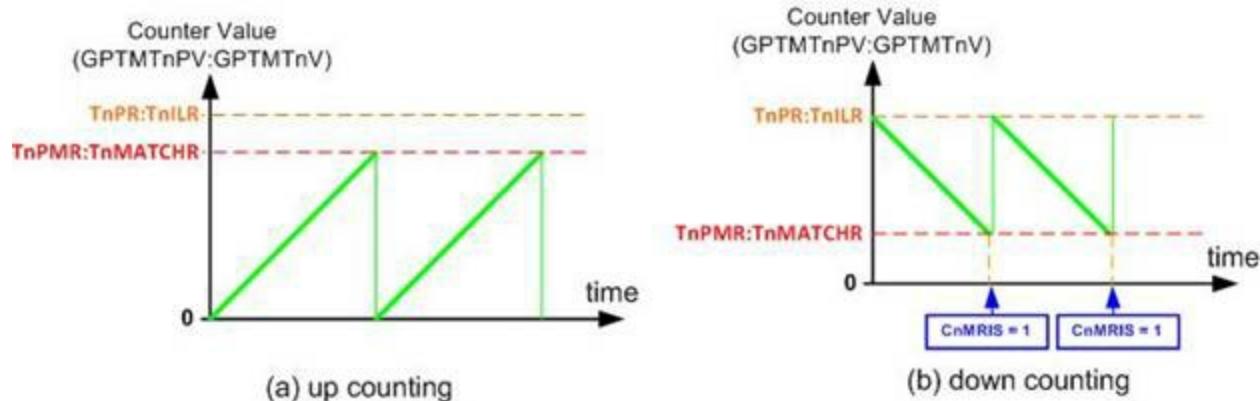


Figure 5-25: Counting in the Input Edge-Count Mode

#### Note

the value of GPTMTnPR and GPTMTnILR must be greater than the value of GPTMTnPMPR and GPTMTnMATCHR.

In down counting, the GPTMTnV and GPTMTnPV registers are initialized with GPTMTnILR and GPTMTnPR, respectively and they count down until they reach to TnMATCHR and TnPMR, respectively. Then, they are reloaded with GPTMTnILR and GPTMTnPR and the CnMRIS flag of the GPTMRIS register is set. As a result, in cases that a task must be done after a specific

number of events, the registers should be initialized so that ( $TnPR:TnILR - TnPMR:TnMATCHR = \text{number of events to be counted}$ ); and then the CnMRIS flag is monitored to be set.

Program 5-9 functions initialize Timer03A for edge-count capture mode and read the current counter value.

#### Program 5-9: Initialize Timer3A to capture rising edges

```
/* p5_9.c: Initialize Timer3A to capture rising edge in edge-count mode. */
/* The input pin of Timer3A is PB2. See Table 5-2. */
void Timer3A_countCapture_init(void)
{
    SYSCTL->RCGCTIMER |= 8;           /* enable clock to Timer Block 3 */
    SYSCTL->RCGCGPIO |= 2;           /* enable clock to PORTB */

    GPIOB->DIR &= ~0x04;             /* make PB2 an input pin */
    GPIOB->DEN |= 0x04;              /* make PB2 a digital pin */
    GPIOB->AFSEL |= 0x04;            /* enable alternate function on PB2 */
    GPIOB->PCTL &= ~0x00000F00;      /* configure PB2 as T3CCP0 pin */
    GPIOB->PCTL |= 0x00000700;

    TIMER3->CTL &= ~1;              /* disable TIMER3A during setup */
    TIMER3->CFG = 4;                /* configure as 16-bit timer mode */
    TIMER3->TAMR = 0x13;             /* up-count, edge-count, capture mode */
    TIMER3->TAMATCHR = 0xFFFF;       /* set the count limit */
    TIMER3->TAPMR = 0xFF;            /* to 0xFFFFFFFF with prescaler */
    TIMER3->CTL &= ~0xC;             /* capture the rising edge */
    TIMER3->CTL |= 1;               /* enable timer3A */
}

int Timer3A_countCapture(void)
{
    return TIMER3->TAR;
}
```

## Review Questions

1. True or false. The Timer can also be used as event counter.
2. True or false. The TI timer can only count the falling edges.
3. True or false. In edge count mode if the up count mode is used the GPTMTnMATCHR must be greater than GPTMTnILR.
4. True or false. With prescaler We can make a 24-bit counter.
5. True or false. To use the timer as even-counter, we must configure it in capture mode.

## Section 5.6: 32/64-bit Timer Programming (Case Study)

We just examined the use of 16/32-bit Timerx as 16-bit timer. To use the 32-bit option, we can combine (concatenate) the 16-bit TimerA and TimerB together to create a 32-bit timer. Exploring the use of 16/32 as 32-bit timer is left to reader. Next, we examine the 32-bit programming of 32/64 Timerx. First, notice the address block assigned to 32/64 timers are different from the 16/32 timers. It must be noted that the 16/32-bit Timers are physically different timers than 32/64-bit timers. For this reason, the 32/64-bit timers are referred to as Wide Timers to distinguish them from the 16/32 bit Timers. In other words, we have 6 Timers (Timer0-Timer5) of 16/32-bit and another 6 timers (WTimer0-WTimer5) of 32/64-bit. The Base addresses for the Wide Timers are shown below:

- 32/64-bit Wide Timer 0 base: 0x4003.6000
- 32/64-bit Wide Timer 1 base: 0x4003.7000
- 32/64-bit Wide Timer 2 base: 0x4004.C000
- 32/64-bit Wide Timer 3 base: 0x4004.D000
- 32/64-bit Wide Timer 4 base: 0x4004.E000
- 32/64-bit Wide Timer 5 base: 0x4004.F000

To configure TimerA as 32-bit timer we use D2:D1:D0=0x4 for the GPTMCFG (GPTM Configuration Register).

### Example 5-8

For CPU Freq=16MHz calculate the largest delay size using a) 32-bit TimerA without prescaler, and b) 16-bit TimerA with prescaler.

#### Solution:

$1/16\text{MHz} = 62.5\text{nsec}$  is period of clock pulses fed to CPU.

(a)  $2^{32} \times 62.5\text{nsec} = 268.4\text{ seconds}$  for TimerA 32-bit option .

(b)  $65,536 \times 256 \times 62.5\text{ ns} = 1.0456\text{ second}$  for TimerA 16-bit option with the Prescaler.

---

Program 5-10 shows how to create 2-sec delay using the 32-bit option of 32/64 bit Timer1. It uses TimerA of WTimer1 block.

#### Program 5-10: Toggling PF3 LED using with 2sec delay by a wide timer

```
/* p5_10.c: This program is modified from p5_4 to use wide timer 1 to create a longer
delay. */

#include "TM4C123GH6PM.h"

void wtimer1A_delaySec(int ttime);

int main (void)
{
    /* enable clock to GPIOF at clock gating control register */
```

```

SYSCTL->RCGCGPIO |= 0x20;
/* enable the GPIO pins for the LED (PF3, 2, and 1) as output */
GPIOF->DIR = 0x0E;
/* enable the GPIO pins for digital function */
GPIOF->DEN = 0x0E;

while(1)
{
    GPIOF->DATA = 8;           /* turn on blue LED */
    wtimer1A_delaySec(2);     /* wtimer1A 2 sec delay */
    GPIOF->DATA = 0;           /* turn off blue LED */
    wtimer1A_delaySec(2);     /* wtimer1A 2 sec delay */
}

/* multiple of second delays using periodic mode */
void wtimer1A_delaySec(int ttime)
{
    SYSCTL->RCGCWTIMER |= 2;      /* enable clock to WTimer Module 1 */

    WTIMER1->CTL = 0;           /* disable WTimer before initialization */
    WTIMER1->CFG = 0x04;         /* 32-bit option */
    WTIMER1->TAMR = 0x01;        /* one-shot mode and down-counter */
    WTIMER1->TAILR = 16000000 * ttime - 1; /* WTimer A interval load value reg */
    WTIMER1->ICR = 1;
    WTIMER1->CTL |= 0x01;        /* enable WTimer A after initialization */

    while((WTIMER1->RIS & 1) == 0); /* wait till timeout */
}

```

Going beyond 32-bit Timer. Using the 8-bit prescaler can extend the 32-bit to a 40-bit timer. A better way will be to use 64-bit option of 32/64 bit wide timers.

### Example 5-9

Assume CPU Freq=80MHz. Calculate the largest delay size using (a) 32-bit and (b) 64-bit timer can create.

#### Solution:

$1 / 80\text{MHz} = 12.5 \text{ nsec}$  is the period of clock pulses fed to timer.

(a)  $2^{32} \times 12.5 \text{ nsec} = 53.687 \text{ seconds}$  for 32-bit timer.

(b)  $2^{64} \times 12.5 \text{ nsec} = 230,584,300,921 \text{ seconds}$  for 64-bit timer.

#### Review Questions

1. True or false. The 16/32-bit and 32/64-bit wide timers use the same address space in the memory map.
2. In TI ARM Tiva, why the 32/64-bit timers are called Wide timer.

3. With CPU Freq=16MHz, calculate the largest time delay size for 64-bit timer.
4. Give the addresses used for 16/32-bit and 32/64-bit Timer0 in TI ARM Tiva.
5. True or false. The 32-bit part of 16/32-bit Timer0 is the same as the 32 part of 32/64-bit of Timer0.

## Answers to Review Questions

### Section 5-0

1. 31
2. 32
3. event counter
4. Timer
5. 9

### Section 5.1

1. 0xFFFFFFF
2.  $1/16\text{MHz} = 62.5 \text{nsec}$ . Now,  $5 \text{ msec}/62.5\text{nsec} = 80,000$ . Therefore, RELOAD=80,000 – 1 = 79,999
3. 24
4. The D0 of STCTRL (the Enable)
5. Down counter

### Section 5.2

1. True
2. 6, Timer0 to Timer5
3. True
4. RCGCTimer which is part of SYTCTL registers.
5. General Purpose Timer Module (GPTM)

### Section 5.3

1. True
2. True
3. False
4. True
5. To use the same pin for different function (of course not at the same time).

### Section 5.4

1. False
2. False
3. True
4. True
5. before

### Section 5.5

1. True
2. False

3. True
4. True
5. True

## Section 5.6

1. False
2. To distinguish them from the 16/32-bit Timers
3.  $1/16\text{MHz} = 62.5\text{ns}$  and  $2^{64} \times 62.5\text{ns} = 1,152,921,504,606.846976 \text{ sec} = 320,255,973 \text{ hours} = 13,343,998 \text{ days}$
4. 0x4000.0000 for 16/32-bit Timer0 and 0x4003.6000 for 32/64-bit Timer0.
5. False.



## Chapter 6: Interrupt and Exception Programming

This chapter examines the interrupts in ARM. We also discuss sources of hardware interrupts in the ARM. In Section 6.1 we discuss the concept of interrupts in the ARM CPU, then we look at the interrupt assignment of the ARM Cortex-M. Section 6.2 examines the NVIC interrupt controller and discusses the Thread and Handler mode in ARM Cortex-M. The interrupt for I/O ports are discussed in Section 6.3. Section 6.4 examines the interrupt for UART. Timers interrupts are explored in Section 6.5. The SysTick interrupt is covered in Section 6.6. The interrupt priority is discussed in Section 6.7.

## Section 6.1: Interrupts and Exceptions in ARM Cortex-M

In this section, first we examine the difference between polling and interrupt and then describe the various interrupts of the ARM Cortex.

### Interrupts vs. polling

A single microprocessor can serve several devices. There are two ways to do that: interrupts or polling. In the *interrupt* method, whenever any device needs service, the device notifies the CPU by sending it an interrupt signal. Upon receiving an interrupt signal, the CPU interrupts whatever it is doing and serves the device. The program associated with the interrupt is called the *interrupt service routine (ISR)* or *interrupt handler*. In *polling*, the CPU continuously monitors the status of a given device; when the status condition is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced. See Figure 6-1.

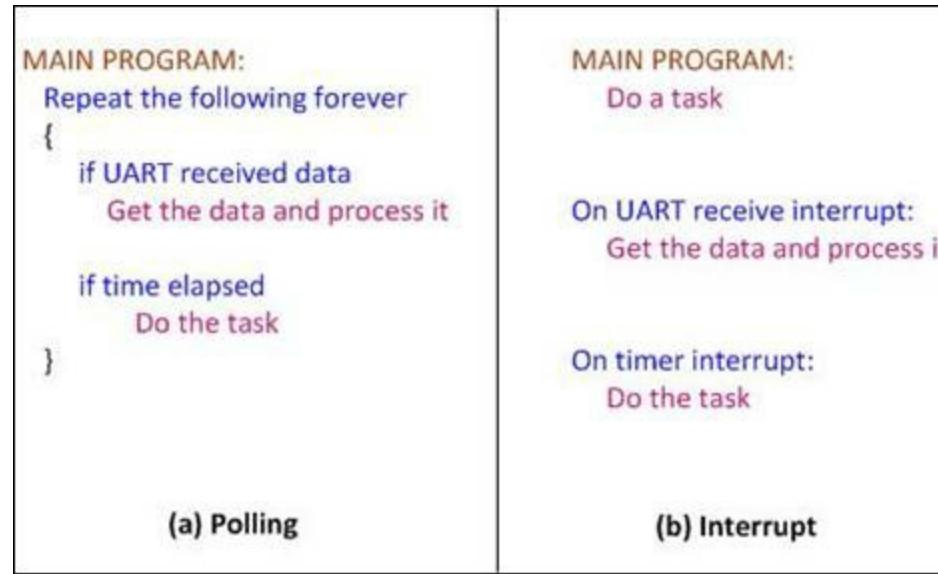


Figure 6-1: Polling vs. Interrupts

Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the CPU time. The polling method wastes much of the CPU's time by polling devices that do not need service. So in order to avoid tying down the CPU, interrupts are used. For example, in Timer we might wait until a determined amount of time elapses, and while we were waiting we cannot do anything else. That is a waste of the CPU's time that could have been used to perform some useful tasks. In the case of the Timer, if we use the interrupt method, the CPU can go about doing other tasks, and when the COUNT flag is raised the Timer will interrupt the CPU to let it know that the time is elapsed. See Figure 6-1.

### Interrupt service routine (ISR)

For every interrupt there must be a program associated with it. When an interrupt occurs this program is executed to perform certain service for the interrupt. This program is commonly referred to as an *interrupt service routine (ISR)*. The interrupt service routine is also called the *interrupt handler*. When an interrupt occurs, the CPU runs the interrupt service routine. Now the question is, how the ISR gets executed?

As shown in Figure 6-2, in the ARM CPU there are pins that are associated with hardware interrupts. They are input signals into the CPU. When the signals are triggered, CPU pushes the PC register onto the stack and loads the PC register with the address of the interrupt service routine. This causes the ISR to get executed.

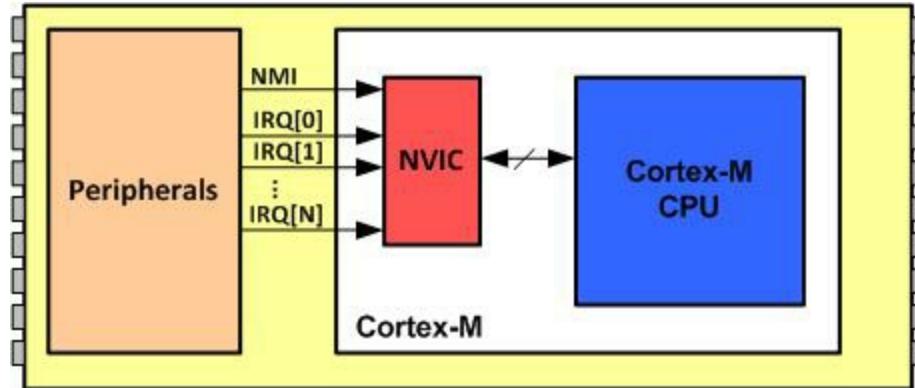


Figure 6-2: NVIC in ARM Cortex-M

As can be seen from Table 6-1, for every interrupt there are four bytes of memory allocated in the interrupt vector table. These four memory locations provide the addresses of the interrupt service routine for which the interrupt was invoked.

## Interrupt Vector Table

Since there is a program (ISR) associated with every interrupt and this program resides in memory (RAM or ROM), there must be a look-up table to hold the addresses of these ISRs. This look-up table is called *interrupt vector table*. In the ARM, the lowest 1024 bytes ( $256 \times 4 = 1024$ ) of memory space are set aside for the interrupt vector table and must not be used for any other function. Table 6-1 provides a list of interrupts and their designated functions as defined by ARM Cortex-M products. Of the 256 interrupts, some are used for software interrupts and some are for hardware IRQ interrupts.

## NVIC (nested vector interrupt controller) In ARM Cortex-M

In the ARM Cortex series we have Cortex-A, Cortex-R and Cortex-M. Currently only the Cortex-M has an on-chip interrupt controller called NVIC (Nested Vector Interrupt Controller). See Figure 6-2. This allows some degree of standardization among the ARM Cortex-Mx (M0, M1, M3, and M4) family members. The classical ARM chips and Cortex-A and Cortex-R series do not have this NVIC interrupt controller, therefore ARM manufacturers' implementation of the interrupt handling varies. This chapter focuses on the interrupts for ARM Cortex-M series. It must be noted that there are substantial differences between the ARM Cortex-M series and classical ARM versions as far as interrupt handling are concerned. The study of classical ARM and ARM Cortex A and R series interrupts are left to the reader since they are used for high performance systems using complex OS and real-time system.

## Interrupt and Exception assignments in ARM Cortex-M

The NVIC of the ARM Cortex-M has room for the total of 255 interrupts and exceptions. The interrupt numbers are also referred to INT type (or INT #) in which the type can be from 1 to 255 or 0x01 to 0xFF. That is INT 01 to INT 255 (or INT 0x01 to INT 0xFF.) The NVIC in ARM Cortex-M assigns the first 15 interrupts for internal use. The memory locations 0-3 are used to

store the value to be loaded into the stack pointer when the device is coming out of reset. See Table 6-1.

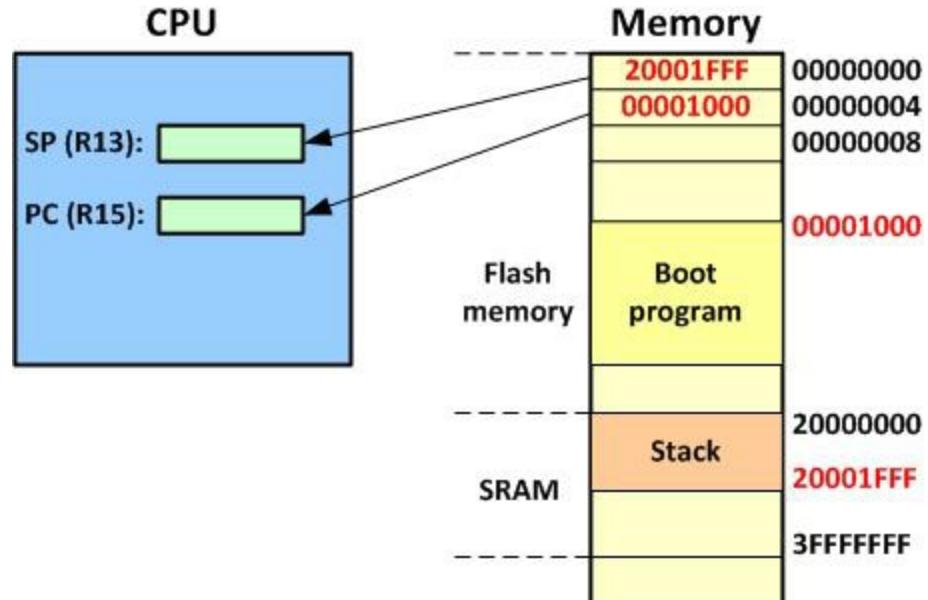
Table 6-1: Interrupt Vector Table for ARM Cortex-M

Interrupt #	Interrupt	Memory Location (Hex)
	<i>Stack Pointer initial value</i>	0x00000000
1	Reset	0x00000004
2	NMI	0x00000008
3	Hard Fault	0x0000000C
4	Memory Management Fault	0x00000010
5	Bus Fault	0x00000014
6	Usage Fault (undefined instructions, divide by zero, unaligned memory access,...)	0x00000018
7	Reserved	0x0000001C
8	Reserved	0x00000020
9	Reserved	0x00000024
10	Reserved	0x00000028
11	SVCall	0x0000002C
12	Debug Monitor	0x00000030
13	Reserved	0x00000034
14	PendSV	0x00000038
15	SysTick	0x0000003C
16	IRQ for peripherals	0x00000040
17	IRQ for peripherals	0x00000044
...	...	...
255	IRQ for peripherals	0x000003FC

## The predefined Interrupts (INT 0 to INT 15)

The followings are the first 15 interrupts in ARM Cortex-M:

### Reset



The ARM devices have a reset pin. It is usually tied to a circuit that keeps the pin low for a while when the power is coming on. This is the power-up reset or power-on reset (POR). On the ARM trainer board, there is often a push-button switch to lower the signal too. The reset signal is normally high after the power is on and when reset is activated during power-on or when the reset button is pressed, it goes low and the CPU goes to a known state with all the registers loaded with the predefined values. When the device is coming out of reset, the ARM Cortex-M loads the program counter from memory location 0x00000004. In ARM Cortex-M system we must place the starting address of the program at the 0x00000004 to get the program running. Notice in Table 6-1, the addresses 0x00000000 to 0x00000003 are set aside for the initial stack pointer value. This ensure that the ARM has access to stack immediately coming out of the reset.

### **Non-maskable interrupt**

As shown in Figure 6-2, there are pins in the ARM chip that are associated with hardware interrupts. They are IRQs (interrupt request) and NMI (nonmaskable interrupt). IRQ is an input signal into the CPU, which can be masked (ignored) and unmasked through the use of software. However, NMI, which is also an input signal into the CPU, cannot be masked by software, and for this reason it is called a *nonmaskable interrupt*. ARM Cortex-M NVIC has embedded "INT 02" into the ARM CPU to be used only for NMI. Whenever the NMI pin is activated, the CPU will go to memory location 0x00000008 to get the address of the interrupt service routine (ISR) associated with NMI. Memory locations 0x00000008, 0x00000009, 0x0000000A, and 0x0000000B contain the 4 bytes of address associated with the ISR belonging to NMI.

### **Exceptions (Faults)**

There is a group of interrupts belongs to the category referred to as *fault* or exception *interrupts*. Internally, they are invoked by the microprocessor whenever there are conditions (exceptions) that the CPU is unable to handle. One such situation is an attempt to execute an instruction that is not implemented in this CPU. Since the result is undefined, and the CPU has no way of handling it, it automatically invokes the invalid instruction exception interrupt. This is commonly referred to as *exception or fault* in the ARM literature. Whenever an invalid instruction is executed, the CPU will go to memory location 0x00000018 to get the address of the ISR to handle the situation. The undefined instruction fault is part of the *Usage Fault* exceptions. Another exception is an attempt to divide a number by zero. Since the result of dividing a number by zero is undefined, and the CPU has no way of handling such a result, it automatically invokes the divide error exception interrupt. As we discussed in Chapter 6 of Volume 1, the unaligned data memory access for word or half-word can cause an exception too. There are many exceptions in the ARM Cortex. See Table 6-1. They are:

### **Hard Fault**

The hard fault is an exception that occurs when the CPU having difficulties executing the ISR for any of the exceptions. One common cause of hard fault is trying to write to the registers of a peripheral before the clock is enabled for that peripheral.

## Memory Management Fault

The memory manager unit fault is used for protection of memory from unwanted access. An example of memory management exception fault is when the access permission in MPU is violated by attempting to write into a region of memory designated as read-only. In an ARM chip with an on-chip MMU, the page fault can also be mapped into the memory management fault. See Chapter 15.

## Bus Fault

The bus fault is an exception that occurs when there is an error in accessing the buses. This can be due to memory access problem during the fetch stage of an instruction or reading and writing to data section of memory. For example, if you try to access memory address location that has not been mapped to a memory chip or peripheral device the Bus Fault exception will occur.

## Usage Fault

The ARM Cortex-M chip has implemented the divide-by-zero, unaligned memory access, undefined instruction, and so on as part of the Usage Fault exception. See your ARM Cortex-M data sheet.

## *SVCall*

An ISR can be called upon as a result of the execution of SVC (supervisor call) instruction. This is referred to as a *software interrupt* since it was invoked from software, not from a fault exception, external hardware, or any peripheral IRQ interrupt. Whenever the SVC instruction is executed, the CPU will go to memory location 0x0000002C to get the address of the ISR associated with SVC. The SVC is widely used by the operating system to call the OS kernel functions and services that can be provided only by the privileged access mode of the OS. In many systems, the API and function calls needed by various User applications are handled by the SVCall to make sure the OS is protected. In the classical ARM literature, SVC was called SWI (software interrupt), but the ARM Cortex-M has renamed it as SVC. Again it must be noted that the SVC is an ARM Cortex-M instruction and can be used like any other ARM instruction.

## *PendSV (pendable service call)*

The PendSV (pendable service call) can be used to do the same thing as the SVC to get the OS services. However, the SVC is an instruction and is executed right away just like all ARM instructions. The PendSV is an interrupt and can wait until NVIC has time to service it when other urgent higher priority interrupts are being taken care. Examine the concept of nested interrupt and pending interrupts at end of this section to see how NVIC handles multiple pending interrupts.

## *Debug Monitor*

In executing a sequence of instructions, there is a need to examine the contents of the CPU's registers and system memory. This is often done by executing the program one instruction at a time and then inspecting registers and memory. This is commonly referred to as *single-stepping*, or performing a trace. ARM has designated INT 12, debug monitor, specifically for implementation of single-stepping.

## SysTick

In the multitasking OS we need a real time interrupt clock to notify the CPU that it needs to service the task. The clock tick happens at a regular interval and is used mainly by the OS system. The SysTick in ARM Cortex is designed for this purpose.

## IRQ Peripheral interrupts

An ISR can be launched as a result of an event at the peripheral devices such as timer timeout or analog-to-digital converter (ADC) conversion complete. The largest number of the interrupts in the ARM Cortex-M belong to this category. Notice from Table 6-1 that ARM Cortex-M NVIC has set aside the first 15 interrupts (INT 1 to INT 15) for internal use and exceptions and is not available to chip designer. The Reset, NMI, undefined instructions, and so on are part of this group of exceptions. The rest of the interrupts can be used for peripherals. Many of the INT 16 to INT 255 are used by the chip manufacturer to be assigned to various peripherals such as timers, ADC, Serial COM, external hardware interrupts, and so on. There is no standard in assigning the INT 16 to INT 255 to the peripherals. Different manufacturers assign different interrupts to different peripherals and you need to examine the data sheet for your ARM Cortex-M chip. Each peripheral device has a group of special function registers that must be used to access the device for configuration. For a given peripheral interrupt to take effect, the interrupt for that peripheral must be enabled. The special function registers for that device provide the way to enable the interrupts.

## *Fast context saving in task switching*

Most of the interrupts are asynchronous, that means they may happen any time in the middle of program execution. When the interrupt is acknowledged and the interrupt service routine is launched, the interrupt service routine will need some CPU resource, mainly the CPU registers, to execute the code. In order not to corrupt the register content of the program that was running before interrupt occurs, these CPU registers need to be preserved. This saving of the CPU contents before switching to interrupt handler is called context switching (or context saving). The use of the stack as a place to save the CPU's contents is tedious and time consuming. It takes time to save all the registers. In executing an interrupt service routine, each task generally needs some key registers such as PC (R15), LR (R14), and CPSR (flag register), in addition to some working registers. For that reason the ARM Cortex-M automatically saves the registers of CPSR, PC, LR, R12, R3, R2, R1, and R0 on stack when an interrupt is acknowledged. See Figure 6-4. If the interrupt service routine needs to use more registers than those preserved, the program has to save the content before using the other registers. The choice of the registers automatically saved adheres to the ARM Architecture Procedure Call Standard (AAPCS) so that an interrupt handler may be written as a plain C function without the need of any special provision.

When floating-point coprocessor is present, the FPU registers need to be saved too. If the interrupt handler does not use floating point coprocessor, saving FPU registers is a waste of time. The Tiva ARM chip allows enabling lazy stacking in FPCC register. When lazy stacking is enabled, the stack pointer will be moved as if the FPU registers are saved for compatibility reason but the content of the FPU registers are not actually saved. This is very useful if no

floating point instructions are used in the interrupt handler. Even with lazy stacking enabled, the interrupt handler still has the option to save the FPU registers before performing floating point instructions.

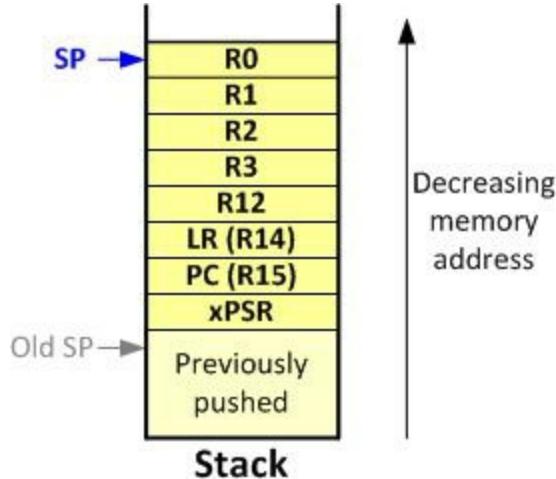


Figure 6-4: ARM Cortex-M Stack Frame upon Interrupt

### Processing interrupts in ARM Cortex-M

When the ARM Cortex-M processes any interrupt (from either Fault Exceptions or peripheral IRQs), it goes through the following steps:

1. The Current processor status register (CPSR) is pushed onto the stack and SP is decremented by 4, since CPSR is a 4-byte register.
2. The current PC (R15) is pushed onto the stack and SP is decremented by 4.
3. The current LR (R14) is pushed onto the stack and SP is decremented by 4.
4. The current R12 is pushed onto the stack and SP is decremented by 4.
5. The current R3 is pushed onto the stack and SP is decremented by 4.
6. The current R2 is pushed onto the stack and SP is decremented by 4.
7. The current R1 is pushed onto the stack and SP is decremented by 4.
8. The current R0 is pushed onto the stack and SP is decremented by 4.
9. Save Floating point coprocessor registers or move SP if lazy stacking is enabled.
10. The CPU goes into the Handler Mode (details will be described later). LR is loaded with a number with bit 31-5 all 1s.
11. The INT number (type) is multiplied by 4 to get the address of the location within the vector table to fetch the program counter of the interrupt service routine (interrupt handler).
12. From the memory locations pointed to by this new PC, the CPU starts to fetch and execute instructions belonging to the ISR program.
13. When one of the return instructions is executed in the interrupt service routine, the CPU recognizes that it is in the Handler Mode from the value of the LR. It then restores the registers saved when entering ISR including the program counter from the stack and makes the CPU run the code where it left off when interrupt occurred. See Figure 6-5.

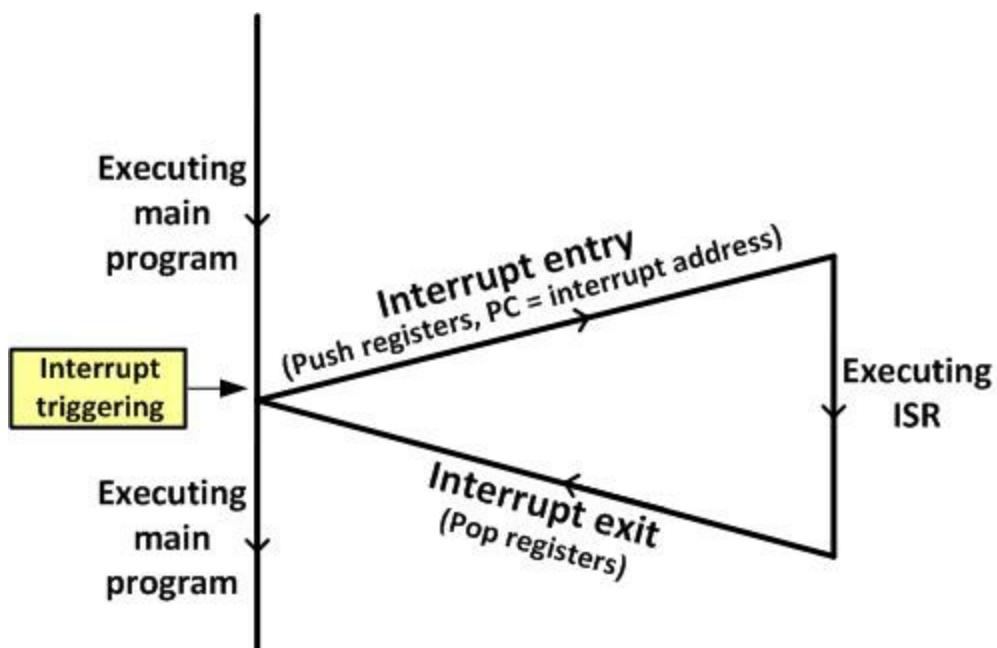


Figure 6-5: Main Program gets Interrupted

## Difference between interrupt and a subroutine call

If the execution of an interrupt saves the program counter of the following instruction and jumps indirectly to the subroutine associated with the interrupt, what is the difference between that and a BL instruction, which also saves the program counter and jumps to the desired subroutine (procedure)? The differences can be summarized as follows:

1. A "BL" instruction can take an argument and jump to any location within the 4-gigabyte address range of the ARM CPU, but "INT" goes to a fixed memory location in the interrupt vector table to get the address of the interrupt service routine.
2. A "BL" instruction is used by the programmer in the sequence of instructions in the program but an externally activated hardware interrupt can come in at any time, requesting the attention of the CPU.
3. A "BL" instruction cannot be masked (disabled), but "INT#" belonging to externally activated hardware interrupts can be masked.
4. A "BL" instruction automatically saves only PC of the next instruction on the stack, while "INT#" saves SP, R12, R3–R0, CPSR (flag register) in addition to PC of the next instruction.
5. An interrupt puts the CPU in the Handler Mode while the "BL" instruction does not change the CPU execution mode.
6. When returning from the end of the subroutine that has been called by the "BL" instruction, the PC is restored to the address of the next instruction after the "BL" instruction. When returning from the interrupt handler, the CPU will restore the registers saved when the CPU entered into ISR (the CPSR, R15, R14, R12, R3–R0 registers) from the top of stack.

## Interrupt priority

The next topic in this section is the concept of priority for exceptions and IRQs. What happens if two interrupts want the attention of the CPU at the same time? Which has priority? In the ARM Cortex-M the Reset, NMI and Hard Fault exceptions have fixed priority levels and are set by the ARM itself and not subject to change. Among the Reset, NMI and Hard Fault, the

Reset has the highest priority. As we can see from Table 6-2, the NMI and Hard Fault have lower priority than Reset, meaning if all three of them are activated at the same time, the Reset will be executed first. If both NMI and an IRQ are activated at the same time, NMI is responded to first since NMI has a higher priority than IRQ. The rest of the exceptions and IRQs have lower priority and are configurable, meaning their priority levels can be set by the programmer. Programmable priority levels are values between 0 and 7 with 7 has the lowest priority.

**Table 6-2: Interrupt Priority for ARM Cortex-M**

Interrupt #	Interrupt	Priority Level
<b>0</b>	<i>Stack Pointer initial value</i>	
<b>1</b>	Reset	-3 Highest
<b>2</b>	NMI	-2
<b>3</b>	Hard Fault	-1
<b>4</b>	Memory Management Fault	Programmable
<b>5</b>	Bus Fault	Programmable
<b>6</b>	Usage Fault (undefined instructions, divide by zero, unaligned memory access,...)	Programmable
<b>7</b>	Reserved	Programmable
<b>8</b>	Reserved	Programmable
<b>9</b>	Reserved	Programmable
<b>10</b>	Reserved	Programmable
<b>11</b>	SVCall	Programmable
<b>12</b>	Debug Monitor	Programmable
<b>13</b>	Reserved	Programmable
<b>14</b>	PendSV	Programmable
<b>15</b>	SysTick	Programmable
<b>16</b>	IRQ for peripherals	Programmable
<b>17</b>	IRQ for peripherals	Programmable
...	...	Programmable
<b>255</b>	IRQ for peripherals	Programmable

Table 6-2 shows standard interrupt assignment for ARM Cortex. Not all Cortex-M chips have all the first 15 interrupts. In some ARM Cortex-M chips if there is no memory management unit then its interrupt is reserved. Make sure you examine your ARM Cortex-M chip manual before you start using it. Again it must be emphasized that for the hardware IRQs coming through NVIC, the NVIC resolves priority depending on the way the NVIC is programmed.

## Interrupt latency

The time from the moment the event that triggers an interrupt signal to the moment the CPU starts to execute the ISR code is called the interrupt latency. This latency depends on whether the source of the interrupt is an internal (e.g., exceptions) or external hardware (e.g., peripheral hardware IRQ) interrupt. The duration of an interrupt latency can also be affected by the type of the instruction which the CPU was executing when the interrupt occurs. It takes longer in cases where the instruction being executed lasts for many instruction cycles compared

to the instructions that last for only one instruction cycle time. In the ARM Cortex-M, we also have extra clocks added to the latency due to the fact that it saves the content of registers CPSR, PC, LR, R12, and R0-R3 on stack. See your ARM Cortex-M manual for the timing data sheet.

Another source of the interrupt latency is the interrupt priority. As mentioned earlier, when several interrupts occur at the same time, the interrupt with the highest priority is acknowledged first. All other interrupts have to wait.

### Interrupt inside an interrupt handler (nested interrupt)

What happens if the ARM is executing an ISR belonging to an interrupt and another interrupt is activated? In such cases, a higher priority interrupt can preempt a lower priority interrupt. The higher priority interrupt will stop the lower priority interrupt handler and launch the higher priority interrupt handler. In the ARM Cortex-M systems, it is up to the software engineer to configure the priority level for each exception and IRQ device and set the policy of how to support nested interrupt. In many older CPUs when an interrupt service routine is launched, all other interrupts are masked. All interrupts happened at this time have to wait. If the interrupt service routine runs too long, there is a risk some interrupts may be lost. The interrupt service routine may unmask the interrupts. But in doing so, it will allow all the interrupts to preempt itself.

The ARM Cortex-M allows only the higher priority interrupts to preempt the lower priority interrupt service routine. The programmer is responsible to assign the proper priority to each IRQ to determine whether an interrupt may preempt the other's interrupt handler. The NVIC in ARM Cortex-M has the ability to capture the pending interrupts and keeps track of each one until all are serviced.

### Review Questions

1. True or false. When any interrupt is activated, the CPU jumps to a fixed and unique address.
2. There are \_\_\_\_\_ bytes of memory in the interrupt vector table for each interrupt.
3. How many K bytes of memory are used by the interrupt vector table, and what are the beginning and ending addresses of the table for the first 256 interrupts?
4. The program associated with an interrupt is also referred to as \_\_\_\_\_.
5. What is the function of the interrupt vector table?
6. What memory locations in the interrupt vector table hold the address for INT 16 ISR?
7. The ARM Cortex-M has assigned INT 2 to NMI. Can that be changed?
8. Which interrupt is assigned to divide error exception handling?

## Section 6.2: ARM Cortex-M Processor Modes

In this section we examine various operation modes in ARM Cortex-M.

### ARM Cortex Thread (application) and Handler (exception) modes

In comparing the traditional ARM7 with ARM Cortex-M series we see some major changes in the ARM Cortex-M series. Among the changes are the CPU modes, stack, interrupt processing and many new instructions. These changes are meant to make the ARM Cortex-M systems to run programs faster and more efficiently. We have examined some of these changes in this chapter since the vast majority of them are related to the interrupt execution. The ARM Cortex-M can run in one of the two modes at any given time. They are: (1) Thread (Application) mode and (2) Handler (Exception) mode. The differences can be stated as follows:

1. When the ARM Cortex-M is powered on and coming out of reset, it automatically goes to the Thread mode. The Thread mode is the mode that vast majority of the applications programs are executed in. The CPU spends most of its time in Thread mode and gets interrupted only to execute ISR for exception faults or peripheral IRQs.
2. The ARM Cortex-M switches to Handler mode only when an exception fault (of course other than the Reset) or an IRQ interrupt from a peripherals is activated to get the attention of the CPU to execute an ISR (interrupt handler). Upon returning from ISR, the CPU automatically changes from Handler mode back to Thread mode. It must be noted that of all the exceptions and IRQs in the Table 6-1, only the Reset forces the CPU into Thread mode and the rest are executed in Handler mode.

A big advantage of having Handler mode is that when returning from Handler mode, the CPU will pop the stack and restore the registers saved during entry to Handler mode. With this an interrupt handlers are written just like any other functions as we will see in the examples soon.

### There are two Stacks in ARM Cortex

The classical ARM has a single stack pointer (R13) to be used to point to RAM area for the purpose of stack. With a multi-threaded operating system, every thread should have their own stack so does the operating system itself. It is much more efficient to have separate stack pointers for the system and the thread. The ARM Cortex-M has two stack pointer registers. They are called PSP (processor stack pointer) and MSP (main stack pointer). Threads running in Thread mode should use the process stack and the kernel and exception handlers should use the main stack.

The bit 1, ASP (active stack pointer), of the special function register called CONTROL register gives the option of choosing MSP or PSP for stack pointer. Upon Reset the ASP=0, meaning that R13 is the Main Stack pointer (MSP) and its value come from the first 4 bytes of the interrupt vector table starting at 0x00000000 address location. By making the ASP=1, the R13 is the same as PSP (processor stack pointer). Next, we examine the privilege levels in ARM Cortex-M.



**nPRIV (Privilege):** Defines the Thread mode privilege level

- 0: Privileged
- 1: Unprivileged

**Active Stack Pointer (ASP):** Defines the currently active stack pointer (ASP = SPSEL)

- 0: MSP is the current stack pointer.
- 1: PSP is the current stack pointer.

**Floating Point Context Active (FPCA)**

- 0: No floating point context active.
- 1: Floating point context active.

Figure 6-6: CONTROL Register in ARM Cortex-M4

Table 6-3: Privileged level Execution and Processor Modes in ARM Cortex-M

Processor Mode	Software	Privilege level
<b>Thread</b>	Applications	Privileged and Unprivileged
<b>Handler</b>	ISR for Exceptions and IRQs	Always Privileged

*In Thread mode, use bit 0 of the CONTROL register to select Privileged or Unprivileged*

## Privileged and Unprivileged levels in ARM Cortex-M

The ARM Cortex-M series has a new feature that did not exist in the previous ARM products. This new feature is called privileged level. There are two privilege levels in ARM Cortex-M. They are called Privileged and Unprivileged. The Privileged level in ARM Cortex-M can be used to limit the CPU access to special registers and protected memory area to prevent the system from getting corrupted due to error in coding or malicious user. Here are summary of the Privileged level software:

1. Privileged level software have access to all registers including the special function registers for interrupts.
2. Privileged level software have access to every region of memory.
3. Privileged level software have access to system timer, NVIC, and system resources.
4. The Privileged level software can execute all the ARM Cortex-M instructions including the MRS, MSR, and CPS.
5. The Handlers for fault exceptions and IRQs can be executed only in Privileged level.
6. Only the Privileged software can access the CONTROL register to see whether execution is in Privileged or Unprivileged mode. In Unprivileged mode one can switch from Unprivileged level to Privileged level by using SVC instruction.

Table 6-4: Processor Modes and Stack Usage in ARM Cortex-M

Processor Mode	Software	Stack Usage
<b>Thread</b>	Applications	MSP or PSP
<b>Handler</b>	ISR for Exceptions and IRQs	MSP

*In Thread mode, use bit 1 of the Control register to select MSP or PSP for stack pointer.*

Here are summary of the Unprivileged level software:

1. Unprivileged level software have no access to some registers such the special function registers for interrupts.
2. Unprivileged level software have limited access to some regions of memory.
3. Unprivileged level software are blocked from accessing system timer, NVIC, and system control block and resources.
4. The Unprivileged level software cannot execute some of the ARM instructions such as CPS. It has limited access to the MRS and MSR instructions.
5. While Handler mode is always executed in the Privileged level, the Thread mode software can be executed in Privileged or Unprivileged level. The bit 0 of the special a function register called CONTROL register gives the option of running the software in Privileged or Unprivileged mode.
6. In Unprivileged mode, one can use SVC instruction to make a supervisor call to switch from Unprivileged level to Privileged level.

Table 6-5: Processor Mode, Privilege, and Stack in ARM Cortex

Mode	Privilege	Stack Pointer	Typical Example usage
Handler	Privileged	Main	Exception Handling
Handler	Unprivileged	Any	Reserved since Handler is always Privileged
Thread	Privileged	Main	Operating system kernel
Thread	Privileged	Process	
Thread	Unprivileged	Main	
Thread	Unprivileged	Process	Application threads

### Special Function register in ARM Cortex

Beside the traditional general purpose registers of R0–R15, the ARM Cortex has many new special function registers. These registers are widely used in programs written for the Cortex-M based embedded systems. See Figure 6-7.

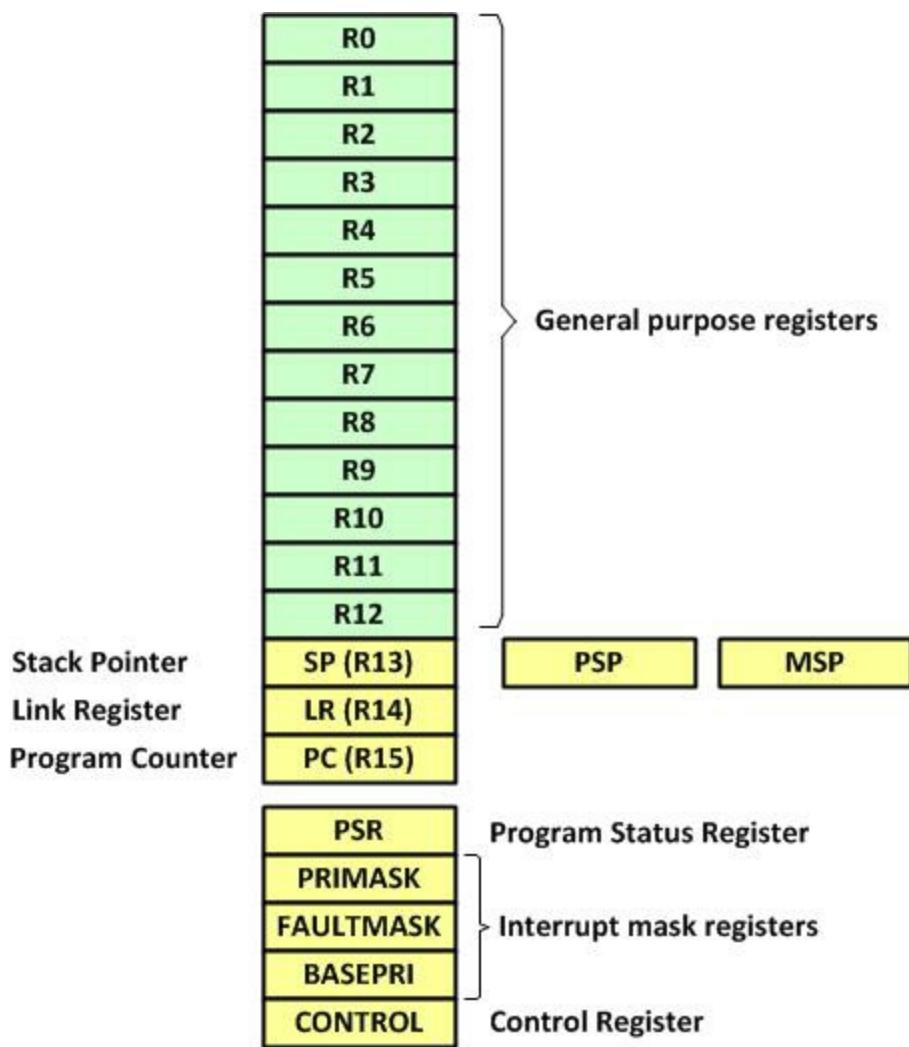


Figure 6-7: ARM Cortex-M Registers

While the general purpose registers of R0–R15 can be accessed using the MOV, LDR, and STR instructions, these new special function registers can be accessed only with the two new instructions MSR and MRS. To manipulate (clear or set) the bits of special function registers, first we must use the MSR to move them to a general purpose register and after changing their values they are moved back by using MRS instruction. Table 6-6 shows special function registers.

Table 6-6: Special function registers of ARM Cortex-M

Register name	Privilege Usage
<b>MSP (main stack pointer)</b>	Privileged
<b>PSP (processor stack pointer)</b>	Privileged or Unprivileged
<b>PSR (Processor status register)</b>	Privileged
<b>APSR (application processor status register)</b>	Privileged or Unprivileged
<b>ISPR (interrupt processor status register)</b>	Privileged
<b>EPSR (execution processor status register)</b>	Privileged
<b>PRIMASK (Priority Mask register)</b>	Privileged
<b>FAULTMASK (fault mask register)</b>	Privileged
<b>BASEPRI (base priority register)</b>	Privileged
<b>CONTROL (control register)</b>	Privileged

*Note: We must use MSR and MRS instructions to access the above registers*

## Review Questions

1. True or false. When a Reset pin is activated, the ARM CPU wakes up in Thread mode.
2. There are only \_\_\_\_\_ processor modes in the ARM Cortex. Give their names
3. How many bytes of data are fetched into CPU from interrupt vector table when ARM Cortex-M is Reset, and what are they?
4. Another name for ISR is \_\_\_\_\_.
5. True or false. When an interrupt comes in from exception fault or IRQ, the ARM CPU switches to Handler mode automatically.

## Section 6.3: I/O Port Interrupt Programming

In Chapter 2, we showed how to use GPIO ports for simple I/O. We also showed a simple program getting (polling) an input switch and placing it on LED. In this section, we show how to program the interrupt capability of the GPIO ports.

### PORTF Interrupt Register

As we mentioned in Chapter 2, the SW1 and SW2 on TI LaunchPad are switches connected to the PFO and PF4 pins, respectively. We will use these two switches to show examples of interrupt programming. However, before we do that, we need to examine the interrupt vector table for the TI Tiva TM4C123G Microcontroller. Table 6-7 shows interrupt assignment in Tiva TM4C123G.

Table 6-7: IRQ assignment in Tiva ARM TM123GH6PM

INT#	IRQ#	Vector location	Device
1-15	none	0000 0000 to 0000 003C	CPU Exception (set by ARM)
16	0	0000 0040	GPIO PORT A
17	1	0000 0044	GPIO PORT B
18	2	0000 0048	GPIO PORT C
19	3	0000 004C	GPIO PORT D
20	4	0000 0050	GPIO PORT E
21	5	0000 0054	UART0
22	6	0000 0058	UART1
23	7	0000 005C	SSIO
24	8	0000 0060	I2C0
25	9	0000 0064	PWM0 Fault
26	10	0000 0068	PWM0 Generator 0
27	11	0000 006C	PWM0 Generator 1
28	12	0000 0070	PWM0 Generator 2
29	13	0000 0074	QEIO
30	14	0000 0078	ADC0 Sequence 0
31	15	0000 007C	ADC0 Sequence 1
32	16	0000 0080	ADC0 Sequence 2
33	17	0000 0084	ADC0 Sequence 3
34	18	0000 0088	Watchdog Timers 0 and 1
35	19	0000 008C	16/32-Bit Timer 0A
36	20	0000 0090	16/32-Bit Timer 0B
37	21	0000 0094	16/32-Bit Timer 1A
38	22	0000 0098	16/32-Bit Timer 1B
39	23	0000 009C	16/32-Bit Timer 2A
40	24	0000 00A0	16/32-Bit Timer 2B
41	25	0000 00A4	Analog Comparator 0
42	26	0000 00A8	Analog Comparator 1
43	27	-	Reserved
44	28	0000 00B0	System Control
45	29	0000 00B4	Flash Memory Control and EEPROM Control
46	30	0000 00B8	GPIO Port F

<b>47-48</b>	31-32	-	Reserved
<b>49</b>	33	0000 00C4	UART2
<b>50</b>	34	0000 00C8	SSI1
<b>51</b>	35	0000 00CC	16/32-Bit Timer 3A
<b>52</b>	36	0000 00D0	16-32-Bit Timer 3B
<b>53</b>	37	0000 00D4	I2C1
<b>54</b>	38	0000 00D8	QEI1
<b>55</b>	39	0000 00DC	CAN0
<b>56</b>	40	0000 00E0	CAN1
<b>57-58</b>	41-42	-	Reserved
<b>59</b>	43	0000 00EC	Hibernation Module
<b>60</b>	44	0000 00F0	USB
<b>61</b>	45	0000 00F4	PWM Generator 3
<b>62</b>	46	0000 00F8	$\mu$ DMA Software
<b>63</b>	47	0000 00FC	$\mu$ DMA Error
<b>64</b>	48	0000 0100	ADC1 Sequence 0
<b>65</b>	49	0000 0104	ADC1 Sequence 1
<b>66</b>	50	0000 0108	ADC1 Sequence 2
<b>67</b>	51	0000 010C	ADC1 Sequence 3
<b>68-72</b>	52-56	-	Reserved
<b>73</b>	57	0000 0124	SSI2
<b>74</b>	58	0000 0128	SSI3
<b>75</b>	59	0000 012C	UART3
<b>76</b>	60	0000 0130	UART4
<b>77</b>	61	0000 0134	UART5
<b>78</b>	62	0000 0138	UART6
<b>79</b>	63	0000 013C	UART7
<b>80-83</b>	64-67	-	Reserved
<b>84</b>	68	0000 0150	I2C2
<b>85</b>	69	0000 0154	I2C3
<b>86</b>	70	0000 0158	16/32-Bit Timer 4A
<b>87</b>	71	0000 015C	16/32-Bit Timer 4B
<b>88-107</b>	72-91	-	Reserved
<b>108</b>	92	0000 01B0	16/32-Bit Timer 5A
<b>109</b>	93	0000 01B4	16/32-Bit Timer 5B
<b>110</b>	94	0000 01B8	32/64-Bit Timer 0A
<b>111</b>	95	0000 01BC	32/64-Bit Timer 0B
<b>112</b>	96	0000 01C0	32/64-Bit Timer 1A
<b>113</b>	97	0000 01C4	32/64-Bit Timer 1B
<b>114</b>	98	0000 01C8	32/64-Bit Timer 2A
<b>115</b>	99	0000 01CC	32/64-Bit Timer 2B
<b>116</b>	100	0000 01D0	32/64-Bit Timer 3A
<b>117</b>	101	0000 01D4	32/64-Bit Timer 3B
<b>118</b>	102	0000 01D8	32/64-Bit Timer 4A
<b>119</b>	103	0000 01DC	32/64-Bit Timer 4B
<b>120</b>	104	0000 01E0	32/64-Bit Timer 5A
<b>121</b>	105	0000 01E4	32/64-Bit Timer 5B

<b>122</b>	<b>106</b>	<b>0000 01E8</b>	System Exception (imprecise)
<b>123-149</b>	<b>107-133</b>	<b>-</b>	Reserved
<b>150</b>	<b>134</b>	<b>0000 0258</b>	PWM Generator 0
<b>151</b>	<b>135</b>	<b>0000 025C</b>	PWM Generator 1
<b>152</b>	<b>136</b>	<b>0000 0260</b>	PWM Generator 2
<b>153</b>	<b>137</b>	<b>0000 0264</b>	PWM Generator 3
<b>154</b>	<b>138</b>	<b>0000 0268</b>	PWM1 Fault

This can be found in the start-up header file of your C compiler. Notice, interrupt numbers 16 to 255 are assigned to the peripherals. The INT 46 is assigned to The GPIO port of PORTF. Although PORTF has 8 pins, we have only one interrupt assigned to the entire PORTF. This is common in many microcontrollers. In other words, when any of the PORTF pins trigger an interrupt, they all go to the same address location in the interrupt vector table. It is job of our interrupt Service Routine (ISR or interrupt Handler) to find out which pin caused the interrupt. Next, we examine the registers associated with the PORTF interrupt.

Upon Reset, all the interrupts are disabled. To enable any interrupt we need three steps:

- 1) Enable the interrupt for a specific peripheral module.
- 2) Enable the interrupts at the NVIC module.
- 3) Enable the interrupt globally .

Next, we look at the details of each one.

- 1) We need to enable the interrupt capability of a given peripheral at the module level. This should be done after other configurations of that peripheral are done. In the case of I/O ports, each pin can be used as a source of external hardware interrupt. This is done with the GPIO Interrupt Mask (GPIOIM) register.

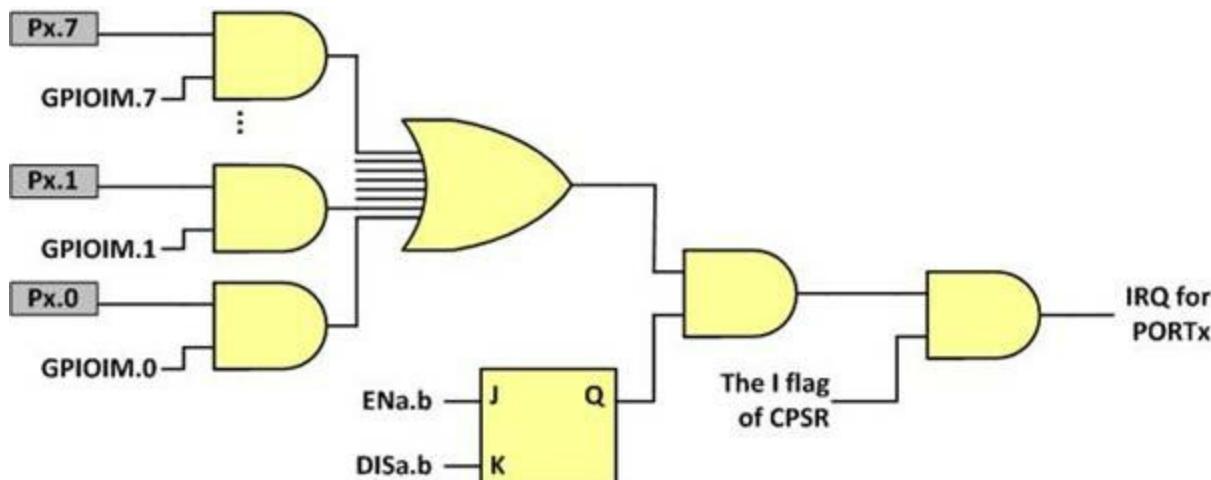


Figure 6-8: Interrupt enabling with all 3 levels

D31	.....	D8	D7	D6	D5	D4	D3	D2	D1	D0	
GPIOIM:	Reserved	IME Px.7	IME Px.6	IME Px.5	IME Px.4	IME Px.3	IME Px.2	IME Px.1	IME Px.0		0x410

**Note:** D0 to D7 are used to enable/disable the interrupt for pins 0 to 7 of the port.

1: Enable interrupt

0: Disable interrupt (mask the interrupt)

Figure 6-9: GPIO Interrupt Mask (GPIOIM)

Notice that, the lower 8 bits of this register is used to enable the interrupt capability of each pin of the I/O port. To enable the interrupts for PF0 and PF4 pins, we will need the following:

GPIOF->IM |= 0x11; /\* unmask interrupt \*/

- 2) In TI Tiva TM4C123G chip, there is an interrupt enable for each entry in the interrupt vector table. These enable bits are in the registers in NVIC. Each register covers 32 IRQ interrupts. For example, register EN0 controls the enable the interrupts for IRQ0 to IRQ31, EN1 for IRQ32 to IRQ63, and so on. See Figures 6-10 to 6-13.

EN0 (ISER[0]):	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0x100

Figure 6-10: Interrupts 0–31 Set Enable (EN0)

EN1 (ISER[1]):	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32

0x104

Figure 6-11: Interrupts 32–63 Set Enable (EN1)

EN2 (ISER[2]):	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64

0x108

Figure 6-12: Interrupts 64–95 Set Enable (EN2)

EN3 (ISER[3]):	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113	112
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96

0x10C

Figure 6-13: Interrupts 94–127 Set Enable (EN3)

Notice these registers are called Interrupt Set Enable and we have an array for all of

them. The array is referred to as ISER[0],ISER[1], and so on.

As we can see in the interrupt vector table in Table 6-7, the PORTF interrupt is assigned to IRQ30. Therefore, to enable the interrupt associated with PORTF in Vector table, we need the following:

```
NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */
```

The interrupts can be enabled using the following function, as well:

```
void NVIC_EnableIRQ(IRQn_Type IRQn);
```

The function is defined in the *Core\_cm4.h* file which is included in the *TM4C123GH6PM.h* header file. To enable an interrupt using the function, the IRQ number of the interrupt should be passed as the argument to the function. For example, the following statement enables the GPIOA\_IRQn interrupt:

```
NVIC_EnableIRQ(19);
```

Since the IRQ number of the interrupts are defined in the *TM4C123GH6PM.h* we can use their names instead of their numbers. For example, to enable GPIOA\_IRQn the following can be used as well:

```
NVIC_EnableIRQ (GPIOA_IRQn);
```

For more information, open the *TM4C123GH6PM.h* file and find "enum IRQn" in the file.

To disable interrupts there are another registers: ICERO to ICER3. See the Figures 6-14 to 6-17.



Figure 6-14: Interrupts 0–31 Clear Enable (DIS0)

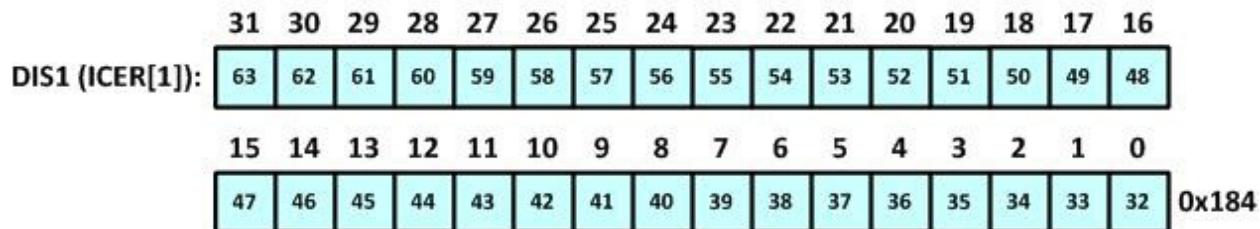


Figure 6-15: Interrupts 32–63 Clear Enable (DIS1)



Figure 6-16: : Interrupts 64–95 Clear Enable (DIS2)

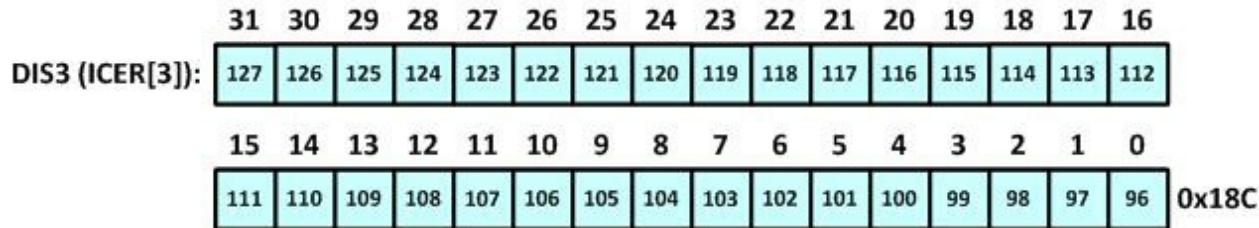


Figure 6-17: : Interrupts 96–127 Clear Enable (DIS3)

Each interrupt can be disabled by writing 1 to the ICER registers. Writing 0 to the ICER registers has no effect on their values. For example, the following instruction disables UART0 interrupt, keeping the other interrupts unchanged:

```
NVIC->ICER[0] = (1<<5); /*disable UART0 Interrupt */
```

The interrupts can be disabled using the following function, as well:

```
void NVIC_DisableIRQ(IRQn_Type IRQn);
```

For example, the following instruction disables the UART0 interrupt:

```
NVIC_DisableIRQ(UART0_IRQn);
```

In fact, each bit of the ISER register together with its peer in the ICER register are connected to a J-K Flip-Flop, as shown below:

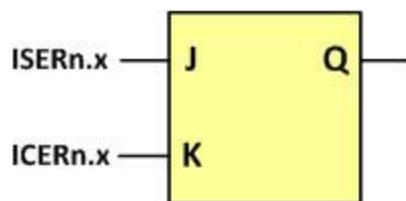


Figure 6-18: Enabling and Disabling an Interrupt

- 3) Global interrupt enable/disable allows us with a single instruction to mask all interrupts during the execution of some critical task such as manipulating a common pointer shared by multiple threads. In ARM Cortex M, we do the global enable/disable of interrupts with assembly language instructions of CPSID I (Change processor state-disable interrupts) and CPSIE I (Change processor state-enable interrupts). In C language we use pseudo-functions of CMSIS:

```
_enable_irq(); /* Enable interrupt Globally */
```

and

```
_disable_irq(); /* Disable interrupt Globally */
```

Now, using the following lines of code, we enable the interrupts for PF0 and PF4 pins at all three levels:

```
GPIOF->IM |= 0x11; /* unmask interrupt */  
NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */  
_enable_irq(); /* global enable IRQs */
```

## IRQ Priority

As described before, since ARM Cortex-M supports higher priority interrupt to preempt the lower interrupt handler, it is important that each interrupt be assigned a proper priority before they are enabled. The IRQ interrupt priorities are controlled by the NVIC registers. For each IRQ number, there is one byte corresponding to that IRQ to assign its priority. The allowed priority levels are ranging from 0 to 7 and they are defined by three bits left justified in that byte. So the C statement to set the priority is:

```
NVIC->IP[INTERRUPT_NUMBER] |= PRIORITY << 5; /* set interrupt priority */
```

To set the GPIOF interrupt (number 30) to priority of 3, we will use the following statement:

```
NVIC->IP[30] = 3 << 5; /* set interrupt priority to level 3 */
```

Section 6.7 covers the interrupt priority in more details.

## Interrupt trigger point

When an input pin is connected to an external device to be used for interrupt, we have 5 choices for trigger point. They are:

- 1) low-level trigger (active Low level),
- 2) high-level trigger (active High level),
- 3) rising-edge trigger (positive-edge going from Low to High),
- 4) falling-Edge trigger (negative-edge going from High to Low),
- 5) Both edge (rising and falling) trigger.

First, we must use GPIO Interrupt Sense (GPIOIS) register to decide the level or edge.

D31	D8	D7	D6	D5	D4	D3	D2	D1	D0	
GPIOIS:	Reserved	IS Px.7	IS Px.6	IS Px.5	IS Px.4	IS Px.3	IS Px.2	IS Px.1	IS Px.0	0x404

**Note:** D0 to D7 are used to choose between edge and level sense for pins 0 to 7 of the port.

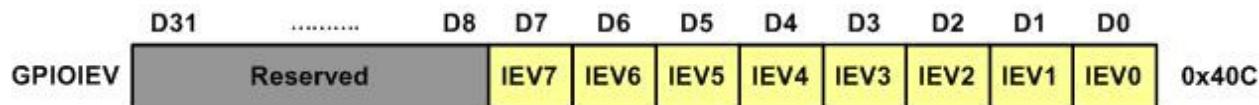
0: Edge-sensitive

1: Level-sensitive

Figure 6-19: GPIO Interrupt Sense (GPIOIS)

Only after using the GPIOIS register do we need to indicate which level or edge. To do that, we use the GPIO Interrupt Event (GPIOIEV) to decide low-level, high-level, falling, or rising-edge. The GPIO Interrupt Both Edges (GPIOIBE) register bits overwrite the decision in GPIOIEV. Unless

both edge interrupt is desired, the bit in GPIOIBE needs to be cleared.



**Note:** D0 to D7 are used to choose between edge and level sense for pins 0 to 7 of the port.  
 0: A falling edge or a Low level on the corresponding pin triggers an interrupt  
 1: A rising edge or a High level on the corresponding pin triggers an interrupt

Figure 6-20: GPIOIEV

Table 6-8: Using GPIOIM and GPIOIEV Registers

IS.n (interrupt sense)	IEV.n (Interrupt Event)	
0	0	Falling edge
0	1	Rising edge
1	0	Low level
1	1	High level

Now, the following lines of code make the PF0 and PF3, active-low level trigger.

```
GPIOF->IS &= ~0x11; /* make bit 4, 0 edge sensitive */
GPIOF->IBE &= ~0x11; /* trigger is controlled by IEV */
GPIOF->IEV &= ~0x11; /* falling edge trigger */
```

We can also do both negative and positive edge trigger too, as we will see soon.

In program 6-1, the main program toggles the red LED of PF1 continuously. When an interrupt comes from SW1 or SW2, it toggles the green LED of PF3 for a short period of time then it returns back to the main. The delay function is called several times in the interrupt handler to make the LED blink. This is done only for demonstration purpose. It is generally a poor practice to do delay or to wait for some event to happen in the interrupt service routine because when interrupt service routine is active, the main program is halted as you can see that when the green LED is blinking, the red LED ceases to blink.

With Keil µVision IDE, a new project will get an assembly startup code **startup\_TM4C123.s** created by the project wizard. For each interrupt, there is a dummy interrupt handler written that does not perform any thing and will never return from the handler. The addresses of these interrupt handlers are listed in the interrupt vector table named **\_Vectors** in the file. To write an interrupt handler, one has to find out the name of the dummy interrupt handler in the interrupt vector table and reuse the name of the dummy interrupt handler. The linker will overwrite the interrupt vector table with the new interrupt handler. In the case of PORTF interrupt, the interrupt handler name is **GPIOF\_Handler**. The interrupt handler is written with a format of a function in C language.

It is critical that the interrupt handler clears the interrupt flag before returning from interrupt handler. Otherwise the interrupt appears as if it is still pending and the interrupt handler will be executed again and again forever and the program hangs. The PORTF interrupt

flag is cleared by writing a 1 to the bit that corresponds to the pin that triggered the interrupt in the GPIO Interrupt Clear Register (GPIOICR). In the case of SW1 (PF4) and SW2 (PF0), the interrupt flags are cleared by:

```
GPIOF->ICR |= 0x11;
```

The ARM Cortex writes are buffered. That means a write does not take effect immediately. It may take many clock cycles before the write to ICR to clear the interrupt flags. When return from interrupt handler and the interrupt flag is still pending, the interrupt handler will be executed again. One way to ensure that interrupt flags are cleared before returning from interrupt handler is to perform a read to force the write to take effect.

### Program 6-1 GPIOF interrupt from SW1 and SW2

```
/* p6_1: Toggle red LED on PF1 continuously. Upon pressing either SW1 or SW2, the green  
LED of PF3 should toggle for three times.  
main program toggles red LED while waiting for interrupt from SW1 or SW2 notice in  
Table 6-7, IRQ30 is assigned to PORTF */  
  
#include "TM4C123GH6PM.h"  
  
void delayMs(int n);  
  
int main(void)  
{  
    SYSCTL->RCGCGPIO |= 0x20; /* enable clock to PORTF */  
  
    /* PORTF0 has special function, need to unlock to modify */  
    GPIOF->LOCK = 0x4C4F434B; /* unlock commit register */  
    GPIOF->CR = 0x01; /* make PORTF0 configurable */  
    GPIOF->LOCK = 0; /* lock commit register */  
  
    /* configure PORTF for switch input and LED output */  
    GPIOF->DIR &= ~0x11; /* make PORTF4 input for switch */  
    GPIOF->DIR |= 0x0E; /* make PORTF3, 2, 1 output for LEDs */  
    GPIOF->DEN |= 0x1F; /* make PORTF4-0 digital pins */  
    GPIOF->PUR |= 0x11; /* enable pull up for PORTF4, 0 */  
  
    /* configure PORTF4, 0 for falling edge trigger interrupt */  
    GPIOF->IS &= ~0x11; /* make bit 4, 0 edge sensitive */  
    GPIOF->IBE &= ~0x11; /* trigger is controlled by IEV */  
    GPIOF->IEV &= ~0x11; /* falling edge trigger */  
    GPIOF->ICR |= 0x11; /* clear any prior interrupt */  
    GPIOF->IM |= 0x11; /* unmask interrupt */  
  
    /* enable interrupt in NVIC and set priority to 3 */  
    NVIC->IP[30] = 3 << 5; /* set interrupt priority to 3 */  
    NVIC->ISER[0] |= 0x40000000; /* enable IRQ30 (D30 of ISER[0]) */  
  
    __enable_irq(); /* global enable IRQs */  
  
    /* toggle the red LED (PF1) continuously */  
    while(1)  
    {  
        GPIOF->DATA |= 0x02;
```

```

        delayMs(500);
        GPIOF->DATA &= ~0x02;
        delayMs(500);
    }

}

/* SW1 is connected to PF4 pin, SW2 is connected to PF0. */
/* Both of them trigger PORTF interrupt */
void GPIOF_Handler(void)
{
    int i;
    volatile int readback;
    /* toggle green LED (PF3) three times */
    for (i = 0; i < 3; i++)
    {
        GPIOF->DATA |= 0x08;
        delayMs(500);
        GPIOF->DATA &= ~0x08;
        delayMs(500);
    }
    GPIOF->ICR |= 0x11; /* clear the interrupt flag before return */
    readback = GPIOF->ICR; /* a read to force clearing of interrupt flag */
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    __disable_irq(); /* disable all IRQs */

    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}

```

Notice in Program 6-1, it does not make distinction between SW1 or SW2. The reason is that only one interrupt (IRQ30) is associated with the entire PORTF. In other words, whichever pin of PORTF interrupt is activated it goes to the same interrupt handler belonging to PORTF. Now, we can modify the Program 6-1 to distinguish between various pins of PORTF. That means, by pressing SW1 (PF4) we can toggle the green LED (PF3) and when SW2 (PF0) is pressed blue LED (PF2) is toggled. Each of the PORTF pin interrupt sets a pin in the GPIO Masked Interrupt Status register (GPIO MIS). The interrupt handler can poll the GPIO MIS register to find out which pin(s) is/are requesting interrupt. See Program 6-2. This is like using an 8-bit port for security system in which each pin is assigned to a window or a door. A different message can be

produced depending on which door or window one is opened. Notice when both switches are pressed and released, both interrupts will be served sequentially.

### Program 6-2: Rewrite of the interrupt handler in Program 6-1 to distinguish the interrupt pin

```
/* p6_2: Toggle red LED on PF1 continuously. Upon pressing either SW1 or SW2, the green or blue LED will toggle three times. main program toggles red LED while waiting for interrupt from SW1 or SW2. notice in Table 6-7, IRQ30 is assigned to PORTF */

#include "TM4C123GH6PM.h"

void delayMs(int n);

int main(void)
{
    SYSCTL->RCGCGPIO |= 0x20; /* enable clock to PORTF */

    /* PORTF0 has special function, need to unlock to modify */
    GPIOF->LOCK = 0x4C4F434B; /* unlock commit register */
    GPIOF->CR = 0x01; /* make PORTF0 configurable */
    GPIOF->LOCK = 0; /* lock commit register */

    /* configure PORTF for switch input and LED output */
    GPIOF->DIR &= ~0x11; /* make PORTF4 input for switch */
    GPIOF->DIR |= 0x0E; /* make PORTF3, 2, 1 output for LEDs */
    GPIOF->DEN |= 0x1F; /* make PORTF4-0 digital pins */
    GPIOF->PUR |= 0x11; /* enable pull up for PORTF4, 0 */

    /* configure PORTF4, 0 for falling edge trigger interrupt */
    GPIOF->IS &= ~0x11; /* make bit 4, 0 edge sensitive */
    GPIOF->IBE &= ~0x11; /* trigger is controlled by IEV */
    GPIOF->IEV &= ~0x11; /* falling edge trigger */
    GPIOF->ICR |= 0x11; /* clear any prior interrupt */
    GPIOF->IM |= 0x11; /* unmask interrupt */

    /* enable interrupt in NVIC and set priority to 3 */
    NVIC->IP[30] = 3 << 5; /* set interrupt priority to 3 */
    NVIC->ISER[0] |= 0x40000000; /* enable IRQ30 (D30 of ISER[0]) */

    __enable_irq(); /* global enable IRQs */

    /* toggle the red LED (PF1) continuously */
    while(1)
    {
        GPIOF->DATA |= 0x02;
        delayMs(500);
        GPIOF->DATA &= ~0x02;
        delayMs(500);
    }
}

/* SW1 is connected to PF4 pin, SW2 is connected to PF0. Both of them trigger PORTF interrupt */
void GPIOF_Handler(void)
{
    int i;
    volatile int readback;
```

```

while (GPIOF->MIS != 0)
{
    if (GPIOF->MIS & 0x10) /* is it SW1(PF4)? */
    {
        /* GPIOF4 pin interrupt */
        /* toggle green LED (PF3) three times */
        for (i = 0; i < 3; i++)
        {
            GPIOF->DATA |= 0x08;
            delayMs(500);
            GPIOF->DATA &= ~0x08;
            delayMs(500);
        }
        GPIOF->ICR |= 0x10; /* clear the interrupt flag */
        readback = GPIOF->ICR; /* a read to force clearing of interrupt flag */
    }
    else if (GPIOF->MIS & 0x01) /* then it must be SW2(PF0) */
    {
        /* GPIOF0 pin interrupt */
        /* toggle blue LED (PF2) three times */
        for (i = 0; i < 3; i++)
        {
            GPIOF->DATA |= 0x04;
            delayMs(500);
            GPIOF->DATA &= ~0x04;
            delayMs(500);
        }
        GPIOF->ICR |= 0x01; /* clear the interrupt flag */
        readback = GPIOF->ICR; /* a read to force clearing of interrupt flag */
    }
    else
    {
        /* We should never get here. */
        /* But if we do, clear all pending interrupts. */
        GPIOF->ICR = GPIOF->MIS;
        readback = GPIOF->ICR; /* a read to force clearing of interrupt flag */
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */

void SystemInit(void)
{
    __disable_irq(); /* disable all IRQs */

    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
}

```

```
SCB->CPACR |= 0x00F00000;
```

```
}
```

## Interrupting on both edge

Another application for an interrupt can be to connect a square wave of 5 V to a pin of PORTx and let an LED toggle at the same rate as the input frequency. We can make it to toggle on positive or negative edge or both. See Program 6-3. Note in this program, the interrupt flag is cleared at the beginning of the interrupt handler so that the subsequent register read/write will force the interrupt flag clear to take effect. Also notice in Table 6-7, IRQ3 is assigned to PORTD.

### Program 6-3 External falling edge interrupt from PORTD6

```
/* p6_3 external interrupt by falling edge PORTD6 is configured to trigger interrupt by
falling edge.
In the interrupt handler, the green LED (PF3) is toggled. The green LED should have
half the frequency of the input signal at PORTD6. Notice in Table 6-7, IRQ3 is assigned
to PORTD */

#include "TM4C123GH6PM.h"

int main(void)
{
    SYSCTL->RCGCGPIO |= 0x20; /* enable clock to PORTF */
    SYSCTL->RCGCGPIO |= 0x08; /* enable clock to PORTD */

    /* configure PORTF for LED output */
    GPIOF->DIR |= 0x0E; /* make PORTF3, 2, 1 output for LEDs */
    GPIOF->DEN |= 0x0E; /* make PORTF4-0 digital pins */

    /* configure PORTD6 for falling edge trigger interrupt */
    GPIOD->DIR &= ~0x40; /* make PORTD6 input pin */
    GPIOD->DEN |= 0x40; /* make PORTD6 digital pin */
    GPIOD->IS &= ~0x40; /* make bit 4, 0 edge sensitive */
    GPIOD->IBE &= ~0x40; /* trigger is controlled by IEV */
    GPIOD->IEV &= ~0x40; /* falling edge trigger */
    GPIOD->ICR |= 0x40; /* clear any prior interrupt */
    GPIOD->IM |= 0x40; /* unmask interrupt */

    /* enable interrupt in NVIC and set priority to 6 */
    NVIC->IP[3] = 6 << 5; /* set interrupt priority to 6 */
    NVIC->ISER[0] |= 0x00000008; /* enable IRQ3 for PORTD (D3 of ISER[0]) */

    __enable_irq(); /* global enable IRQs */

    while(1)
    { /* wait for interrupts */
    }

void GPIOD_Handler(void)
{
    volatile int readback;
```

```

GPIOF->DATA ^= 8;           /* toggle green LED */
GPIOD->ICR |= 0x40;         /* clear the interrupt flag */
readback = GPIOD->ICR;     /* a read to force clearing of interrupt flag */
}

/* This function is called by the startup assembly */
/* code to perform system specific initialization tasks. */
void SystemInit(void)
{
    __disable_irq();        /* disable all IRQs */

    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}

```

### Program 6-4: External interrupt on both edges

```

/* p6_4: external interrupt on both edges */
/* PORTD6 is configured to trigger interrupt by both edges. In the interrupt handler,
the green LED (PF3) is toggled. The green LED should have the same frequency of the
input signal at PORTD6. */
/* Notice in Table 6-7, IRQ3 is assigned to PORTD */

#include "TM4C123GH6PM.h"

int main(void)
{
    SYSCTL->RCGCGPIO |= 0x20;    /* enable clock to PORTF */
    SYSCTL->RCGCGPIO |= 0x08;    /* enable clock to PORTD */

    /* configure PORTF for LED output */
    GPIOF->DIR |= 0x0E;          /* make PORTF3, 2, 1 output for LEDs */
    GPIOF->DEN |= 0x0E;          /* make PORTF4-0 digital pins */

    /* configure PORTD6 for falling edge trigger interrupt */
    GPIOD->DIR &= ~0x40;        /* make PORTD6 input pin */
    GPIOD->DEN |= 0x40;          /* make PORTD6 digital pin */
    GPIOD->IS  &= ~0x40;          /* make bit 4, 0 edge sensitive */
    GPIOD->IBE |= 0x40;          /* trigger on both edges */
    GPIOD->ICR |= 0x40;          /* clear any prior interrupt */
    GPIOD->IM  |= 0x40;          /* unmask interrupt */

    /* enable interrupt in NVIC and set priority to 6 */
    NVIC->IP[3] = 6 << 5;       /* set interrupt priority to 6 */
    NVIC->ISER[0] |= 0x00000008; /* enable IRQ3 for PORTD */

    __enable_irq(); /* global enable IRQs */

    while(1)
    {   /* wait for interrupts */
    }
}

```

```

void GPIOD_Handler(void)
{
    volatile int readback;

    GPIOF->DATA ^= 8;           /* toggle green LED */
    GPIOD->ICR |= 0x40;         /* clear the interrupt flag */
    readback = GPIOD->ICR;      /* a read to force clearing of interrupt flag */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    __disable_irq();          /* disable all IRQs */

    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}

```

## Review Questions

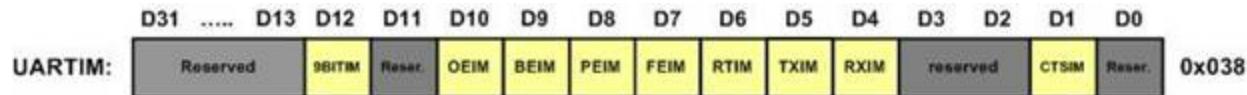
1. IRQ0 is assigned to INT number \_\_\_\_.
2. True or false. There is an interrupt assigned to each pin of every GPIO.
3. True or false. The I/O ports in TI Tiva supports both level and edge trigger interrupts.
4. We use \_\_\_\_\_ in C to enable the interrupts globally.
5. Show 3 levels of interrupt enabling we must go through before we start using it.

## Section 6.4: UART Serial Port Interrupt Programming

In Chapter 4, we showed the programming of UART0 in TI ARM using polling. This chapter shows how to do the same thing using interrupt.

### UART0 Interrupt Programming to receive data

Program 4-2 showed how UART0 receives data by polling the RXFE status flag. The disadvantage with that program is that it ties down the CPU polling the status flag. We can modify it to make it an interrupt driven program. Examining the UARTIM (UART Interrupt Mask) register, we see bit 4 allows us to enable the receive interrupt. If the receive interrupt for UART is enabled, when a byte is received, the receive flag is directed to NVIC and that causes the interrupt handler associated with the UART0 to be executed. In the UART handler we must read the received byte and clear the interrupt flag.



bit	Name	Description
1	CTSIM	1: an interrupt is made when the CTSRIS bit of UARTRIS is set, 0: masked (disabled)
4	RXIM	1: an interrupt is made when the RXRIS bit of UARTRIS is set, 0: masked (disabled)
5	TXIM	1: an interrupt is made when the TXRIS bit of UARTRIS is set, 0: masked (disabled)
6	RTIM	1: an interrupt is made when the RTSRIS bit of UARTRIS is set, 0: masked (disabled)
7	FEIM	1: an interrupt is made when the FERIS bit of UARTRIS is set, 0: masked (disabled)
8	PEIM	1: an interrupt is made when the PERIS bit of UARTRIS is set, 0: masked (disabled)
9	BEIM	1: an interrupt is made when the BERIS bit of UARTRIS is set, 0: masked (disabled)
10	OEIM	1: an interrupt is made when the OERIS bit of UARTRIS is set, 0: masked (disabled)
12	9BITIM	1: an interrupt is made when the 9BITRIS bit of UARTRIS is set, 0: masked (disabled)

Figure 6-21: UART Interrupt Mask (UARTIM)

From Table 6-7 we see IRQ5 is assigned to UART0. We enable the receive interrupt in UART0 as follow:

```
UART0->IM |= 0x0010; /* enable RX interrupt */  
NVIC->ISER[0] |= 0x00000020; /* enable IRQ5 (D5 of ISER[0]) */  
__enable_irq(); /* global enable IRQs */
```

In Program 6-5, pressing a key at the terminal emulator causes the PC to send the ASCII code of the key to the Tiva LaunchPad. When the character is received by UART0, the interrupt handler reads the character and write it on LEDs of PORTF.

#### Program 6-5: Using the UART0 interrupt

```
/* p6_5.c: Modified p4_2.c to use interrupt driven. */  
  
/* Read data from UART0 and display it at the tri-color LEDs. The LEDs are connected to  
Port F 3-1. Press any A-z, a-z, 0-9 key at the terminal emulator and see ASCII value in  
binary is displayed on LEDs of PORTF. */  
  
/* notice in Table 6-7, IRQ5 is assigned to UART0 */
```

```

#include "tm4c123gh6pm.h"

int main(void)
{
    SYSCTL->RCGCUART |= 1; /* provide clock to UART0 */
    SYSCTL->RCGCGPIO |= 1; /* enable clock to PORTA */
    SYSCTL->RCGCGPIO |= 0x20; /* enable clock to PORTF */

    /* UART0 initialization */
    UART0->CTL = 0;           /* disable UART0 */
    UART0->IBRD = 104;        /* 16MHz/16=1MHz, 1MHz/104=9600 baud rate */
    UART0->FBRD = 11;         /* fraction part, see Example 4-4 */
    UART0->CC = 0;            /* use system clock */
    UART0->LCRH = 0x60;       /* 8-bit, no parity, 1-stop bit, no FIFO */
    UART0->IM |= 0x0010;      /* enable RX interrupt */
    UART0->CTL = 0x301;       /* enable UART0, TXE, RXE */

    /* UART0 TX0 and RX0 use PA0 and PA1. Set them up. */
    GPIOA->DEN = 0x03;        /* Make PA0 and PA1 as digital */
    GPIOA->AFSEL = 0x03;      /* Use PA0,PA1 alternate function */
    GPIOA->PCTL = 0x11;       /* configure PA0 and PA1 for UART */

    GPIOF->DIR = 0x0E;        /* configure Port F to control the LEDs */
    GPIOF->DEN = 0x0E;
    GPIOF->DATA = 0;

    /* enable interrupt in NVIC and set priority to 6 */
    NVIC->IP[5] = 3 << 5;    /* set interrupt priority to 3 */
    NVIC->ISER[0] |= 0x00000020; /* enable IRQ5 for UART0 */

    __enable_irq(); /* global enable IRQs */

    for(;;)
    {
    }

}

void UART0_Handler(void)
{
    volatile int readback;

    char c;
    if (UART0->MIS & 0x0010)
    {
        c = UART0->DR;           /* read the received data */
        GPIOF->DATA = c << 1;   /* shift left and write it to LEDs */
        UART0->ICR = 0x0010;     /* clear Rx interrupt flag */
        readback = UART0->ICR;   /* a read to force clearing of interrupt flag */
    }
    else
    {
        /* should not get here. But if it does, */
        UART0->ICR = UART0->MIS; /* clear all interrupt flags */
        readback = UART0->ICR;   /* a read to force clearing of interrupt flag */
    }
}

/* This function is called by the startup assembly code to perform system specific

```

```
initialization tasks. */
void SystemInit(void)
{
    __disable_irq();      /* disable all IRQs */

    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}
```

Also notice that, in program 6-5 we have only a single interrupt for both receive and transmit. If we want to implement both transmit and receive interrupt, then we have to test the TXMIS and RXMIS bits in register UARTRIS to see which one caused the interrupt.

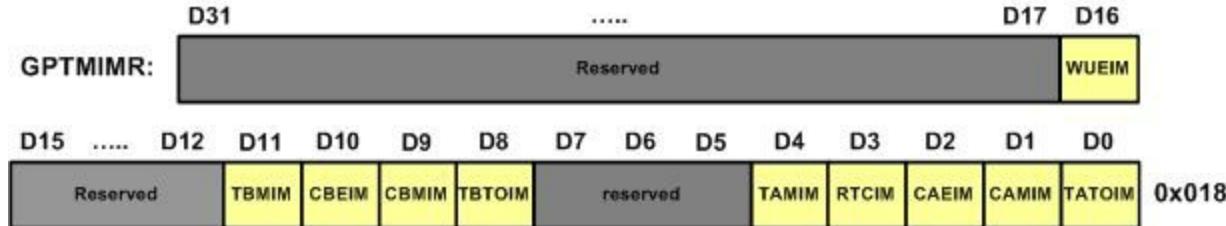
## Review Questions

1. In Tiva TM4CG123, Which IRQ is assigned to UART0?
2. True or false. There is only one interrupt for both Receive and Transmit.
3. True or false. Which pins of PORTA are assigned to TxDO and RxDO.
4. We use register \_\_\_\_\_ enable the interrupt associated with UART0.
5. True or false. Upon Reset, UART0 is enabled and ready to go.

## Section 6.5: Timer Interrupt Programming

In Chapter 5, we showed how to program the timers. In those programming examples, we used polling to see if a timeout event occurred. In this section, we give interrupt-based version of those programs.

Examine the programs in Section 5.2 of Chapter 5. Notice, we could run those programs only one at a time since we have to monitor the timer flag continuously. By using interrupt, we can run several of timer programs all at the same. To do that, we need to enable the timer interrupts using the GPTMIMR (GPTM Interrupt Mask) register.



bit	Name	Description
0	TATOIM	Timer A Time-out Interrupt Mask (0: interrupt is disabled, 1: interrupt is enabled)
1	CAMIM	Timer A Capture mode Match Interrupt Mask (0: interrupt is disabled, 1: enabled)
2	CAEIM	Timer A Capture Mode Event Interrupt Mask (0: interrupt is disabled, 1: enabled)
3	RTCIM	RTC Interrupt Mask (0: interrupt is disabled, 1: interrupt is enabled)
4	TAMIM	Timer A Match Interrupt Mask (0: interrupt is disabled, 1: interrupt is enabled)
8	TBTOIM	Timer B Time-out Interrupt Mask (0: interrupt is disabled, 1: interrupt is enabled)
9	CBMMIM	Timer B Capture mode Match Interrupt Mask (0: interrupt is disabled, 1: enabled)
10	CBEIM	Timer B Capture Mode Event Interrupt Mask (0: interrupt is disabled, 1: enabled)
11	TBMMIM	Timer B Match Interrupt Mask (0: interrupt is disabled, 1: interrupt is enabled)
16	WUEIM	32/64-Bit Wide GPTM Write Update Error Interrupt Mask (0: disabled, 1: enabled)

Figure 6-22: GPTM Interrupt Mask (GPTMIMR)

From Table 6-7, notice that IRQ21 is assigned to Timer1A and IRQ23 to Timer2A. The following will enable these timers in NVIC:

```
NVIC->ISER[0] |= 0x00200000; /* enable IRQ21 (D21 of ISER[0]) */
NVIC->ISER[0] |= 0x00800000; /* enable IRQ23 (D23 of ISER[0]) */
```

In Program 6-6, the main program toggles the blue LED of PF2 continuously. Using interrupts, Timer1A Toggles red LED (PF1) and Timer2A toggles green LED (PF3), every so often.

### Program 6-6: Toggling the green LED using the Timer1A interrupt

```
/* p6_6.c: This program is modified from p5_4 to use Timer1A and Timer2A timeout
events to trigger interrupts. */
/* Timer1A is configure to timeout once every second. In the interrupt handler, the red
LED is toggled. Timer2A is configure to timeout at 10 Hz. */
/* In the interrupt handler, the green LED is toggled. The infinite loop in the main
program is blinking the blue LED while the interrupts are going on. */
/* notice in Table 6-5, IRQ21 is assigned to Timer1A. */
```

```

/* notice in Table 6-5, IRQ22 is assigned to Timer2A. */

#include "TM4C123GH6PM.h"

void timer1A_init(void);
void timer2A_init(void);
void delayMs(int n);

int main (void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2 1) as output */
    GPIOF->DIR = 0x0e;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN = 0x0e;

    timer1A_init(); /* setup timer1A interrupt */
    timer2A_init(); /* setup timer2A interrupt */

    __enable_irq(); /* global enable IRQs */

    while(1)
    {
        GPIOF->DATA ^= 4;           /* toggle blue LED */
        delayMs(500);              /* wait for half second */
    }
}

/* Setup Timer1A to create 1 Hz interrupt */
void timer1A_init(void)
{
    SYSCTL->RCGCTIMER |= 2;      /* enable clock to Timer Block 1 */
    TIMER1->CTL = 0;             /* disable Timer1 before initialization */
    TIMER1->CFG = 0x04;           /* 16-bit option */
    TIMER1->TAMR = 0x02;          /* periodic mode and down-counter */
    TIMER1->TAPR = 250;           /* 16000000 Hz / 250 = 64000 Hz */
    TIMER1->TAILR = 64000;         /* 64000 Hz / 64000 = 1 Hz */
    TIMER1->ICR = 0x1;            /* clear the Timer1A timeout flag */
    TIMER1->IMR |= 0x01;          /* enable Timer1A timeout interrupt */
    TIMER1->CTL |= 0x01;           /* enable Timer1A after initialization */
    NVIC->ISER[0] |= 0x00200000; /* enable IRQ21 (D21 of ISER[0]) */
}

/* Setup Timer2A to create 10 Hz interrupt */
void timer2A_init(void)
{
    SYSCTL->RCGCTIMER |= 4;      /* enable clock to Timer Block 2 */
    TIMER2->CTL = 0;              /* disable Timer2 before initialization */
    TIMER2->CFG = 0x04;           /* 16-bit option */
    TIMER2->TAMR = 0x02;          /* periodic mode and down-counter */
    TIMER2->TAPR = 250;           /* 16000000 Hz / 250 = 64000 Hz */
    TIMER2->TAILR = 6400;          /* 64000 Hz / 6400 = 10 Hz */
    TIMER2->ICR = 0x1;            /* clear the Timer2A timeout flag */
    TIMER2->IMR |= 0x01;          /* enable Timer2A timeout interrupt */
    TIMER2->CTL |= 0x01;           /* enable Timer2A after initialization */
    NVIC->ISER[0] |= 0x00800000; /* enable IRQ23 (D23 of ISER[0]) */
}

```

```

void TIMER1A_Handler(void)
{
    volatile int readback;
    if (TIMER1->MIS & 0x1)      /* Timer1A timeout flag */
    {
        GPIOF->DATA ^= 2;      /* toggle red LED */
        TIMER1->ICR = 0x1;     /* clear the Timer1A timeout flag */
        readback = TIMER1->ICR; /* a read to force clearing of interrupt flag */
    }
    else
    {   /* should not get here, but if we do */
        TIMER1->ICR = TIMER1->MIS; /* clear all flags */
        readback = TIMER1->ICR;     /* a read to force clearing of interrupt flag */
    }
}
void TIMER2A_Handler(void)
{
    volatile int readback;
    if (TIMER2->MIS & 0x1)      /* Timer2A timeout flag */
    {
        GPIOF->DATA ^= 8;      /* toggle green LED */
        TIMER2->ICR = 0x1;     /* clear the Timer2A timeout flag */
        readback = TIMER2->ICR; /* a read to force clearing of interrupt flag */
    }
    else
    {   /* should not get here, but if we do */
        TIMER2->ICR = TIMER2->MIS; /* clear all flags */
        readback = TIMER2->ICR;     /* a read to force clearing of interrupt flag */
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    __disable_irq(); /* disable all IRQs */

    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}

```

## Review Questions

1. Which IRQ is assigned to Timer1A?
2. True or false. There is only one interrupt for all the Timer0 through Timer5.

3. We use register \_\_\_\_\_ to enable the interrupt associated with Timer1A.
4. Show the contents of EN0 register for enabling Timer2A.
5. True or false. Upon Reset, Timer1B is enabled and ready to go.

## Section 6.6: SysTick Programming and Interrupt

Another useful interrupt in ARM is the SysTick. The SysTick timer was discussed in Chapter 5. Next, you learn how to use the SysTick interrupt. See Figure 6-23.

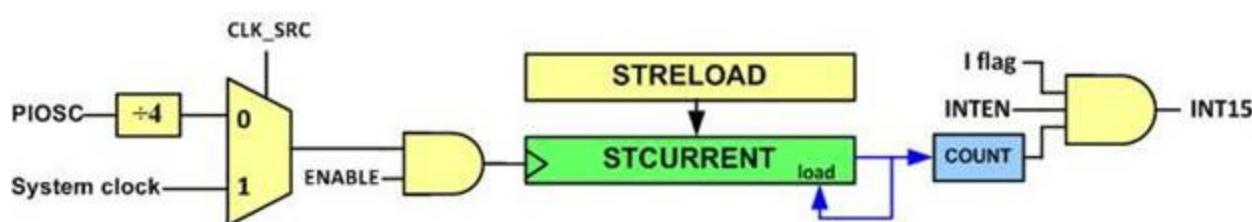
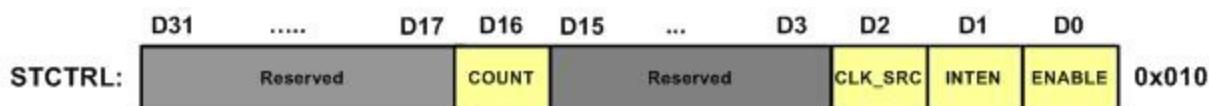


Figure 6-23: SysTick Internal Structure

If INTEN=1, when the COUNT flag is set, it causes an interrupt via the NVIC. INTEN is D1 of the STCTRL register, as shown in Figure 6-24.



bit	Name	Description
0	ENABLE	Enable (0: the counter is disabled, 1: enables SysTick to begin counting down)
	INTEN	Interrupt Enable 0: Interrupt generation is disabled, 1: when SysTick counts to 0 an interrupt is generated
	CLK_SRC	Clock Source 0: Precision internal oscillator (PIOSC) divided by 4 1: System clock
	COUNT	Count Flag 0: the SysTick has not counted down to zero since the last time this bit was read 1: the SysTick has counted down to zero <i>Note: this flag is cleared by reading the STCTRL or writing to CTCURRENT register.</i>

Figure 6-24: SysTick Control and Status Register (STCTRL)

The SysTick interrupt can be used to initiate an action on a periodic basis. This action is performed internally at a fixed rate without external signal. For example, in a given application we can use SysTick to read a sensor every 200 msec. SysTick is used widely for an operating system so that the system software may interrupt the application software periodically (often 10 ms interval) to monitor and control the system operations. See Figure 6-25.

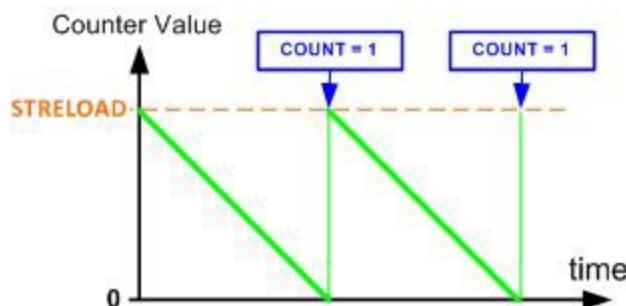


Figure 6-25: SysTick Counting

### Using SysTick with Interrupt

Examining interrupt vector table for ARM Cortex, we see the SysTick is the interrupt #15 assigned to the system itself.

The Program 6-7 uses the SysTick to toggle the red LED of PF1 every second. We need the RELOAD value of 16,000,000-1 since  $1\text{sec} / 62.5\text{nsec} = 16,000,000$ . We assume the CPU clock is 16MHz ( $1/16\text{MHz}=62.5\text{ns}$ ). The COUNT flag is raised every 16,000,000 clocks and an interrupt occurs. Then the RELOAD register is loaded with 16,000,000-1 automatically.

Notice the enabling of interrupt in SysTick Control register. Since SysTick is an internal interrupt, the COUNT flag is cleared automatically when the interrupt occurs. Also notice the SysTick is part of INT1-INT15 and not part of IRQ. For this reason we cannot use Table 6-7 to enable it. Instead, we must use the CTLR register which is part of SysTick in NVIC and standardized by ARM Corp. for ARM Cortex series.

### Program 6-7: SysTick interrupt

```
/* p6_7.c: Toggle the red LED using the SysTick interrupt */
/* This program sets up the SysTick to interrupt at 1 Hz.
The system clock is running at 16 MHz.
1sec/62.5ns=16,000,000 for RELOAD register.
In the interrupt handler, the red LED is toggled. */

#include "TM4C123GH6PM.h"

int main (void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2, 1) as output */
    GPIOF->DIR = 0x0e;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN = 0x0e;

    /* Configure SysTick */
    SysTick->LOAD = 16000000-1;    /* reload with number of clocks per second */
    SysTick->CTRL = 7;           /* enable SysTick interrupt, use system clock */

    __enable_irq();              /* global enable interrupt */

    while (1)
    {
    }
}

void SysTick_Handler(void)
{
    GPIOF->DATA ^= 2;        /* toggle the red LED */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    __disable_irq();          /* disable all IRQs */

    /* Grant coprocessor access */
```

```
/* This is required since TM4C123G has a floating point coprocessor */  
SCB->CPACR |= 0x00F00000;
```

```
}
```

## Review Questions

1. Which interrupt is assigned to SysTick in Tiva TM4C123G?
2. The highest number we can place in RELOAD register is \_\_\_\_\_.
3. We use register \_\_\_\_\_ to enable the interrupt associated with SysTick.
4. True or false. We use ENx register to enable SysTick interrupt.
5. Assume CPU frequency of 16MHz. Find the value for RELOAD register if we want 5 msec elapsed time.

## Section 6.7: Interrupt Priority Programming in TI ARM

The implementation of interrupt varies from vendor to vendor. While ARM Holdings Co. has control over the standardization of the first 3 interrupts (INT0, INT1, and INT2), the ARM licensees are free to implement the interrupts of INT3-INT255. The first three interrupts are Reset, NMI, and Hard Fault. For these three interrupts, Reset has the highest priority (with -3 priority number), then NMI (with -2), and Hard Fault (with -1), in that order. In ARM, the lower value has higher priority. All other interrupts have the priority number 0 meaning they have lower priority than Reset, NMI, and Hard Fault. In the case of TI ARM Tiva TM4C123G, it groups several interrupts together with specific interrupt priority. There are several special function registers dealing with the interrupts belonging to system exceptions of 4 to 15. You can explore them by reading the TI ARM Tiva TM4C123G data sheet. In this section, we deal with the priority of peripheral interrupts of INT16 (IRQ0) to INTx (IRQx).

### IRQ0 to IRQ138 in TI ARM Tiva

The INT16 is assigned to IRQ0 since the first 16 interrupts (INT0-INT15) are used by the CPU itself. Not all the IRQs are implemented in all ARM chips. For example, The TI ARM Tiva chip implements up to IRQ138 (or INT154). In other words, TI ARM Tiva has IRQ0 to IRQ138. Notice that if we add 16 to IRQ# we get its INT#. We learned in Section 6.1 that, by multiplying the INT# by 4 we get its address in the interrupt vector table. Now, as far as peripherals are concerned, we must pay special attention to the IRQ# since this is used in the priority scheme used by the NVIC. According to TI ARM Tiva data sheet, an interrupt (exception) can be in one of the following four states:

- **Inactive.** The exception is not active and not pending.
- **Pending.** The exception is waiting to be serviced by the processor. An interrupt request from a

peripheral or from software can change the state of the corresponding interrupt to pending.

- **Active.** An exception that is being serviced by the processor but has not completed.

**Note:** An exception handler can interrupt the execution of another exception handler. In this

case, both exceptions are in the active state.

- **Active and Pending.** The exception is being serviced by the processor, and there is a pending exception from the same source."

From Table 6-7, we see IRQ0 through IRQ4 are assigned to GPIO PORTA to GPIO PORTE, IRQ5 is assigned UART0, and so on. If we do not use priority, the lower IRQ number has higher priority. In other words, if IRQ2 (GPIOC) and IRQ5 (UART0) are activated at the same time, the IRQ2 is serviced first and then the IRQ5. Also if IRQ2 comes in at the middle of servicing the IRQ5, IRQ5 is put on hold (preempted) and IRQ2 is serviced since IRQ2 has higher priority. IRQ5 service will resume when IRQ2 service is completed and no other higher interrupts are pending. See Example 6-2.

Assume NMI, IRQ4, IRQ9, IRQ1 are activated at the same time. Give the order in which they are serviced.

### Solution:

They are executed in order of NMI (first), IRQ1, IRQ4, and IRQ9 (last).

## The PRI registers and Priority Grouping in TI ARM Tiva

In TI ARM Tiva, there are group of registers called *PRIx* (*PRI*ority $x$ ) in which  $x$  can be 0 to 34. The PRI $x$  registers allow several IRQ interrupts to group together and assign them a priority. For example, PRI0 (PRI zero) has been assigned the task of prioritization of IRQ0, IRQ1, IRQ2 and IRQ3. In the same way, IRQ4, IRQ5, IRQ6 and IRQ7 are assigned to PRI1, the IRQ8, 9, 10, 11 are assigned to PRI2 register, and so on. See Figure 6-26.

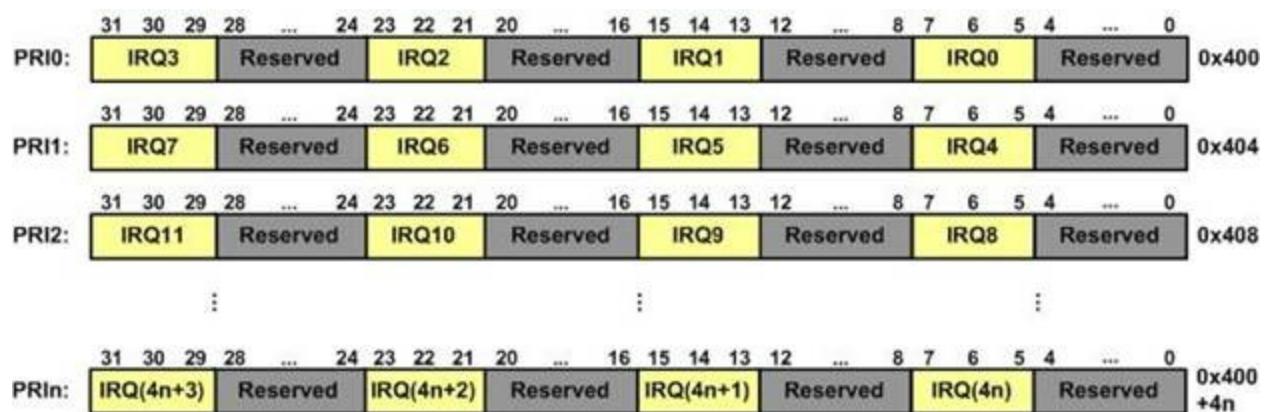


Figure 6-26: PRIn Registers

Notice, there is a pattern in the PRI# and IRQ# assignments. It follows the following formula:

$$\text{PRI}_n = \text{IRQ}(4n), \text{IRQ}(4n+1), \text{IRQ}(4n+2), \text{ and IRQ}(4n+3)$$

In other words, we multiply the PRI# number by 4 (#x4) to get the first IRQ and from there we add 1, 2, and 3 to get all the three IRQs it supports. To ease the calculation of finding the correct bits of the correct register to set the priority, the CMSIS header file provides a struct definition of NVIC->IP[240] with each entry for an IRQ. Each entry has the size of 8 bits and the priority is left justified so the priority number needs to be left shifted 5 positions. To assign IRQ $n$  a priority  $p$ , you only need to use the statement:

`NVIC->IP[n] = p << 5;`

For example, if you want to assign priority 4 to IRQ 20, use the statement:

`NVIC->IP[20] = 4 << 5;`

To set the priority, the following function is also available:

`void NVIC_SetPriority (IRQn_Type IRQn, uint32_t priority);`

For example the following statement sets the priority of TIMER2A to 4:

```
NVIC_SetPriority(TIMER2A_IRQn, 4); /* set EINT3 priority to 4 */
```

Program 6-8 illustrates two interrupts with different priority. In this example, delay function is called in the interrupt handler to demonstrate the preemption by higher priority interrupt. (In real work, it is a bad practice to call delay function in interrupt handler.) Timer1A is programmed to interrupt at 1 second interval. In the interrupt handler, the red LED is turned on for 500 ms. Timer2A is programmed to interrupt at 100 ms interval and in its interrupt handler, the green LED is turned on for 20 ms. Since Timer1A has higher priority, you will observe that the green LED is not blinking when Timer1A interrupt is running (when the red LED is on). Now change the priority of the Timer2A to be higher than Timer1A by changing the line in Timer2\_init( ) from

```
NVIC->IP[23] = 5 << 5; /* set timer2A interrupt priority to 5 */
```

to

```
NVIC->IP[23] = 3 << 5; /* set timer2A interrupt priority to 3 */
```

You will see that the green LED is blinking all the time so is the red LED because the Timer2A (green LED) preempts Timer1A interrupt handler (red LED).

#### Program 6-8: Interrupt priority demonstration

```
/* P6_8.c: Testing nested interrupts */

/* Timer1A is setup to interrupt at 1 Hz. In timer interrupt handler, the red LED is
turned on and a delay 500 ms function is called. The LED is turned off at the end of
the delay. */

/* Timer2A is setup to interrupt at 10 Hz. In timer interrupt handler, the green LED is
turned on and a delay 20 ms function is called. The LED is turned off at the end of the
delay. */

/* When Timer1A has higher priority, the Timer2A interrupts are blocked by Timer1A
interrupt handler. You can see that when the red LED is on, the green LED is not
blinking. When Timer2A has higher priority, the Timer1A interrupt handler is preempted
by Timer2A interrupts and the green LED is blinking all the time. */

#include "TM4C123GH6PM.h"

void Timer1_init(void);
void Timer2_init(void);
void delayMs(int n);

int main (void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2 1) as output */
    GPIOF->DIR = 0x0e;
    /* enable the GPIO pins for digital function */
```

```

GPIOF->DEN = 0x0e;

Timer1_init();
Timer2_init();
__enable_irq();

while(1)
{
}

void TIMER1A_Handler(void)
{
    volatile int readback;

    GPIOF->DATA |= 2; /* turn on red LED */
    delayMs(500);
    GPIOF->DATA &= ~2; /* turn on red LED */
    TIMER1->ICR = 0x1;
    readback = TIMER1->ICR; /* a read to force clearing of interrupt flag */
}

void TIMER2A_Handler(void)
{
    volatile int readback;

    TIMER2->ICR = 0x1;
    GPIOF->DATA |= 8; /* turn on green LED */
    delayMs(20);
    GPIOF->DATA &= ~8; /* turn on green LED */
    readback = TIMER2->ICR; /* a read to force clearing of interrupt flag */
}

void Timer1_init(void)
{
    SYSCTL->RCGCTIMER |= 2; /* enable clock to Timer Module 1 */
    TIMER1->CTL = 0; /* disable Timer1 before initialization */
    TIMER1->CFG = 0x04; /* 16-bit option */
    TIMER1->TAMR = 0x02; /* periodic mode and up-counter */
    TIMER1->TAPR = 250; /* 16000000 Hz / 250 = 64000 Hz */
    TIMER1->TAILR = 64000; /* 64000 Hz / 64000 = 1 Hz */
    TIMER1->ICR = 0x1; /* clear the Timer1A timeout flag */
    TIMER1->IMR |= 0x01; /* enable Timer1A timeout interrupt */
    TIMER1->CTL |= 0x01; /* enable Timer1A after initialization */
    NVIC->IP[21] = 4 << 5; /* set interrupt priority to 4 */
    NVIC->ISER[0] |= 0x00200000; /* enable IRQ21 (D21 of ISER[0]) */
}

void Timer2_init(void)
{
    SYSCTL->RCGCTIMER |= 4; /* enable clock to Timer Module 2 */
    TIMER2->CTL = 0; /* disable Timer2 before initialization */
    TIMER2->CFG = 0x04; /* 16-bit option */
    TIMER2->TAMR = 0x02; /* periodic mode and up-counter */
    TIMER2->TAPR = 250; /* 16000000 Hz / 250 = 64000 Hz */
    TIMER2->TAILR = 6400; /* 64000 Hz / 6400 = 10 Hz */
    TIMER2->ICR = 0x1; /* clear the Timer2A timeout flag */
    TIMER2->IMR |= 0x01; /* enable Timer2A timeout interrupt */
}

```

```

    TIMER2->CTL |= 0x01; /* enable Timer2A after initialization */
    NVIC->IP[23] = 5 << 5; /* set timer2A interrupt priority to 5 */
    NVIC->ISER[0] |= 0x00800000; /* enable IRQ23 (D23 of ISER[0]) */
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int32_t i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    __disable_irq();
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00F00000;
}

```

## Review Questions

1. In ARM which interrupt has the highest priority?
2. True or false. Upon Reset all the IRQs have the same priority.
3. We use register \_\_\_\_\_ to modify the interrupt priority of IRQ8.
4. To assign priority to IRQ20, we need to program the PRI\_\_ register.

## Answer to Review Questions

### Section 6.1

1. True
2. 4
3. 1K byte beginning at 00000000 and ending at 000003FFH
4. Interrupt service routine (ISR) or interrupt handler
5. To hold the starting address of each ISR
6. 0x00000040, 41, 42, and 43
7. No; it is internally embedded into the NVIC.
8. INT 6

### Section 6.2

1. True
2. Thread and Handler
3. 8 bytes. 4 bytes for the address of the stack and 4 bytes for the address of boot ROM
4. Interrupt handler
5. True

## Section 6.3

1. INT#16
2. False
3. True
4. `_enable_irq(); /* Enable interrupt Globally */`
5. a) on the peripheral device level. b) on the system level with ENx register. c) on the global level with the `_enable_irq();` statement.

## Section 6.4

1. IRQ5
2. True
3. PA0 and PA1
4. EN0
5. False

## Section 6-5

1. IRQ21
2. False
3. EN0
4. IRQ23 is assigned to Timer2A. Therefore, EN0=0x0080000
5. False

## Section 6-6

1. INT15
2. 0xFFFFFFF
3. STCTRL
4. False. ENx registers are used for IRQs (external interrupts) and SysTick is not part of them
5.  $1/16\text{MHz}=62.5 \text{nsec}$ . Now,  $5 \text{ msec}/62.5\text{nsec}=80,000$ . Therefore, RELOAD=80,000 -1  
=79,999

## Section 6-7

1. Reset
2. False.
3. PRI2
4. PRI5



## Chapter 7: ADC, DAC, and Sensor Interfacing

This chapter explores more real-world devices such as ADCs (analog-to-digital converters) and sensors. We will also explain how to interface the TI Tiva TM4C123G to these devices. In Section 7.1, we describe analog-to-digital converter (ADC) chips. We will program the ADC module of the Tiva TM4C123G chip in Section 7.2. In Section 7.3, we show the interfacing of sensors and discuss the issue of signal conditioning. The characteristics of DAC chips are discussed in Section 7.4.

## Section 7.1: ADC Characteristics

This section will explore ADC generally. First, we describe some general aspects of the ADC itself, then focus on the functionality of some important pins in ADC.

### ADC devices

Analog-to-digital converters are among the most widely used devices for data acquisition. Digital computers use binary (discrete) values, but in the physical world everything is analog (continuous). Temperature, pressure (wind or liquid), humidity, and velocity are a few examples of physical quantities that we deal with every day. A physical quantity is converted to electrical (voltage, current) signals using a device called a *transducer*. Transducers used to generate electrical outputs are also referred to as *sensors*. Sensors for temperature, velocity, pressure, light, and many other natural physical quantities produce an output that is voltage (or current). Therefore, we need an analog-to-digital converter to translate the analog signals to digital numbers so that the microcontroller can read and process the numbers. See Figures 7-1 and 7-2.



Figure 7-1: Microcontroller Connection to Sensor via ADC

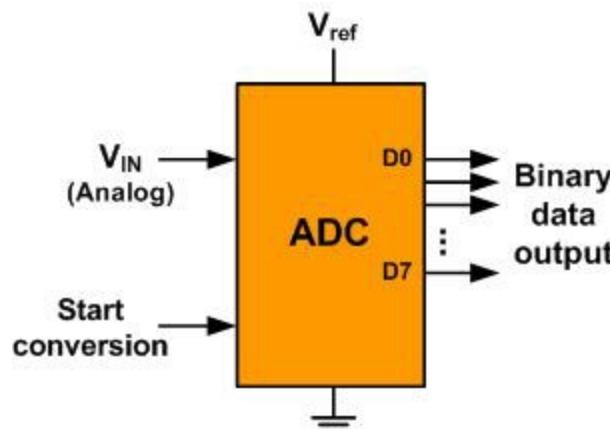


Figure 7-2: An 8-bit ADC Block Diagram

### Some of the major characteristics of the ADC

#### Resolution

The ADC has  $n$ -bit resolution, where  $n$  can be 8, 10, 12, 16, or even 24 bits. Higher-resolution ADCs provide a smaller step size, where *step size* is the smallest change that can be discerned by an ADC. Some widely used resolutions for ADCs are shown in Table 7-1. Although the resolution of an ADC chip is decided at the time of its design and cannot be changed, we can control the step size with the help of what is called  $V_{ref}$ . This is discussed below.

n-bit	Number of steps	Step size
8	256	$5V / 256 = 19.53 \text{ mV}$
10	1024	$5V / 1024 = 4.88 \text{ mV}$

12	4096	$5V / 4096 = 1.2 \text{ mV}$
16	65,536	$5V / 65,536 = 0.076 \text{ mV}$
<b>Note:</b> $V_{ref} = 5V$		

Table 7-1: Resolution versus Step Size for ADC (Vref = 5V)

## Vref

Vref is an input voltage used for the reference voltage. The voltage connected to this pin, along with the resolution of the ADC chip, determine the step size. For an 8-bit ADC, the step size is Vref / 256 because it is an 8-bit ADC, and 2 to the power of 8 gives us 256 steps. See Table 7-1. For example, if the analog input range needs to be 0 to 4 volts, Vref is connected to 4 volts. That gives  $4V / 256 = 15.62 \text{ mV}$  for the step size of an 8-bit ADC. In another case, if we need a step size of 10 mV for an 8-bit ADC, then  $V_{ref} = 2.56 \text{ V}$ , because  $2.56 \text{ V} / 256 = 10 \text{ mV}$ . For the 10-bit ADC, if the  $V_{ref} = 5V$ , then the step size is 4.88 mV as shown in Table 7-1. Tables 7-2 and 7-3 show the relationship between the Vref and step size for the 8- and 10-bit ADCs, respectively. In some applications, we need the differential reference voltage where  $V_{ref} = V_{ref(+)} - V_{ref(-)}$ . Often the  $V_{ref(-)}$  pin is connected to ground and the  $V_{ref(+)}$  pin is used as the  $V_{ref}$ .

V <sub>ref</sub> (V)	V <sub>in</sub> Range (V)	Step Size (mV)
5.00	0 to 5	$5 / 256 = 19.53$
4.00	0 to 4	$4 / 256 = 15.62$
3.00	0 to 3	$3 / 256 = 11.71$
2.56	0 to 2.56	$2.56 / 256 = 10$
2.00	0 to 2	$2 / 256 = 7.81$
1.28	0 to 1.28	$1.28 / 256 = 5$
1.00	0 to 1	$1 / 256 = 3.90$
<b>Note:</b> In an 8-bit ADC, step size is $V_{ref}/256$		

Table 7-2: Vref Relation to Vin Range for an 8-bit ADC

V <sub>ref</sub> (V)	V <sub>in</sub> Range (V)	Step Size (mV)
5.00	0 to 5	$5 / 1024 = 4.88$
4.96	0 to 4.096	$4.096 / 1024 = 4$
3.00	0 to 3	$3 / 1024 = 2.93$
2.56	0 to 2.56	$2.56 / 1024 = 2.5$
2.00	0 to 2	$2 / 1024 = 2$
1.28	0 to 1.28	$1.28 / 1024 = 1.25$
1.024	0 to 1.024	$1.024 / 1024 = 1$
<b>Note:</b> In a 10-bit ADC, step size is $V_{ref}/1024$		

Table 7-3: Vref Relation to Vin Range for an 10-bit ADC

## Conversion time

In addition to resolution, conversion time is another major factor in selecting an ADC. *Conversion time* is defined as the time it takes the ADC to convert the analog input to a digital number. The conversion time is dictated by the clock source connected to the ADC in addition

to the method used for data conversion and technology used in the fabrication of the ADC.

## Digital data output

In an 8-bit ADC we have an 8-bit digital data output of D0–D7, while in the 10-bit ADC the data output is D0–D9. To calculate the output voltage, we use the following formula:

$$D_{\text{OUT}} = V_{\text{IN}} / \text{StepSize}$$

where  $D_{\text{out}}$  = digital data output (in decimal),  $V_{\text{in}}$  = analog input voltage, and step size (resolution) is the smallest change, which is  $V_{\text{ref}}/256$  for an 8-bit ADC.

Figure 7-3 shows a simple 2-bit ADC. In the circuit, the voltage between  $V_{\text{ref}(+)}$  and  $V_{\text{ref}(-)}$  is divided into 4 since resistors have the same values. As a result, the step size is  $(V_{\text{ref}(+)} - V_{\text{ref}(-)}) / 4$ .

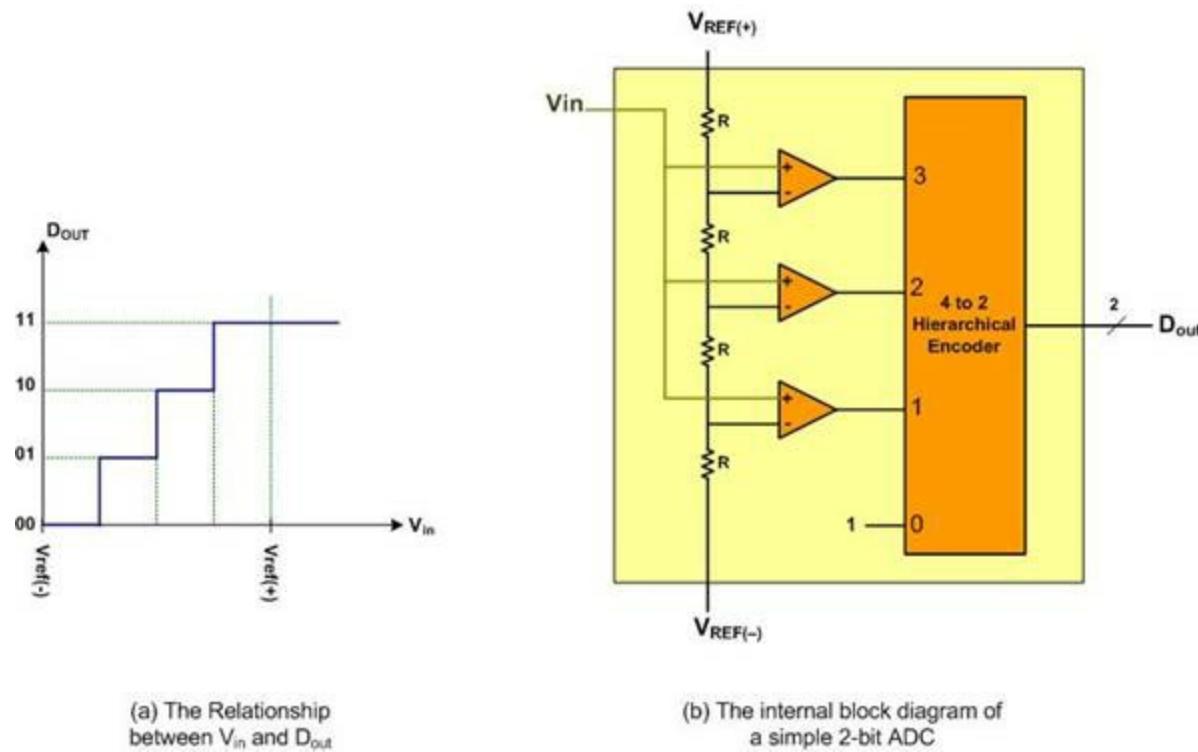


Figure 7-3: A Simultaneous 2-bit ADC

If  $V_{\text{in}}$  is below step size all the comparators send out zeros. When  $V_{\text{in}}$  is between step size and step size  $\times$  2, the lowest comparator sends out 1 and the encoder gives 01.

If  $V_{\text{in}}$  is between step size  $\times$  2 and step size  $\times$  3, the second comparator and the first comparator sends out 1. Since the encoder is hierarchical priority, it sends out the highest value in cases that more than 1 input is high. As a result, 2 (10 in binary) will be sent out.

When  $V_{\text{in}}$  is bigger than step size  $\times$  3, the third comparator becomes high and 3 will be sent out.

See Example 7-1. This data is brought out of the ADC chip either one bit at a time (serially), or in one chunk, using a parallel line of outputs. This is discussed next.

For a given 8-bit ADC (e.g. ADC0848), we have  $V_{ref} = 2.56$  V. Calculate the D0–D7 output if the analog input is: (a) 1.7 V, and (b) 2.1 V.

### Solution:

Since the step size is  $2.56/256 = 10$  mV, we have the following.

(a)  $D_{OUT} = 1.7V/10\text{ mV} = 170$  in decimal, which gives us 10101011 in binary for D7–D0.

(b)  $D_{OUT} = 2.1V/10\text{ mV} = 210$  in decimal, which gives us 11010010 in binary for D7–D0.

---

### Parallel versus serial ADC

The ADC chips are either parallel or serial. In parallel ADC, we have 8 or more pins dedicated to bringing out the binary data, but in serial ADC we have only one pin for data out. The D0–D7 data pins of the 8-bit ADC provide an 8-bit parallel data path between the ADC chip and the CPU. In the case of the 16-bit parallel ADC chip, we need 16 pins for the data path. In order to save pins, many 12- and 16-bit ADCs use pins D0–D7 to send out the upper and lower bytes of the binary data. In recent years, for many applications where space is a critical issue, using such a large number of pins for data is not feasible. For this reason, serial devices such as the serial ADC are becoming widely used. While the serial ADCs use fewer pins and their smaller packages take much less space on the printed circuit board, more CPU time is needed to get the converted data from the ADC because the CPU must get data one bit at a time, instead of in one single read operation as with the parallel ADC. ADC0848 is an example of a parallel ADC with 8 pins for the data output, while the MAX1112 is an example of a serial ADC with a single pin for  $D_{out}$ . Figures 7-4 and 7-5 show the block diagram for ADC0848 and MAX1112.

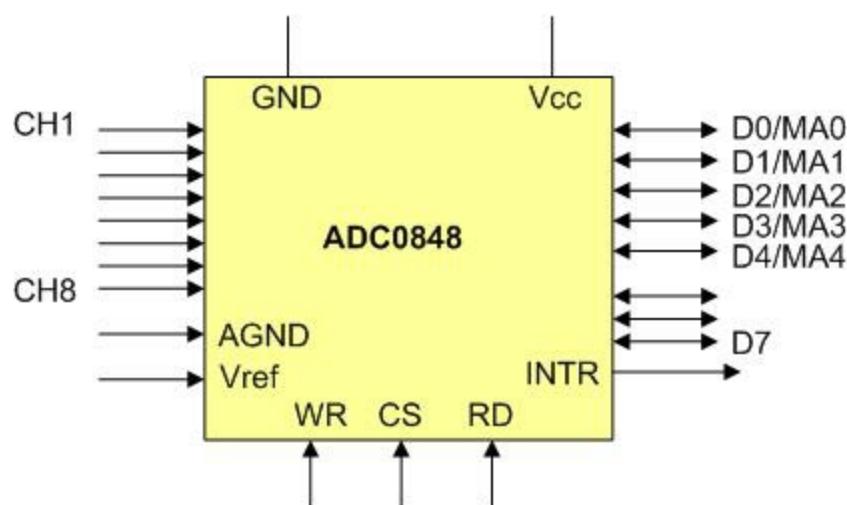


Figure 7-4: ADC0848 Parallel ADC Block Diagram

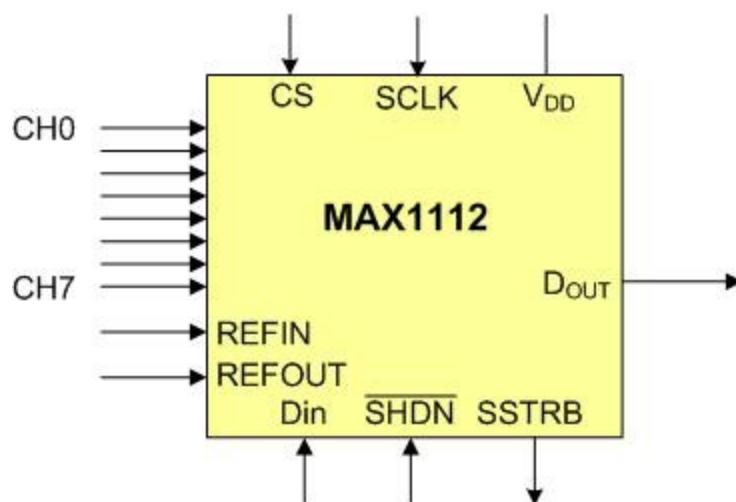


Figure 7-5: MAX1112 Serial ADC Block Diagram

## Analog input channels

Many data acquisition applications need more than one analog input for ADC. For this reason, we see ADC chips with 2, 4, 8, or even 16 channels on a single chip. Multiplexing of analog inputs is widely used as shown in the ADC848 and MAX1112. In these chips, we have 8 channels of analog inputs, allowing us to monitor multiple quantities such as temperature, pressure, flow, and so on. Nowadays, some ARM microcontroller chips come with 16-channel on-chip ADC.

## Start conversion and end-of-conversion signals

For the conversion to be controlled by the CPU, there are needs for start conversion (SC) and end-of-conversion (EOC) signals. When SC is activated, the ADC starts converting the analog input value of  $V_{in}$  to a digital number. The amount of time it takes to convert varies depending on the conversion method. When the data conversion is complete, the end-of-conversion signal notifies the CPU that the converted data is ready to be picked up.

## Successive Approximation ADC

Successive Approximation is a widely used method of converting an analog input to digital output. It has three main components: (a) successive approximation register (SAR), (b) comparator, and (c) control unit. See the figure below.

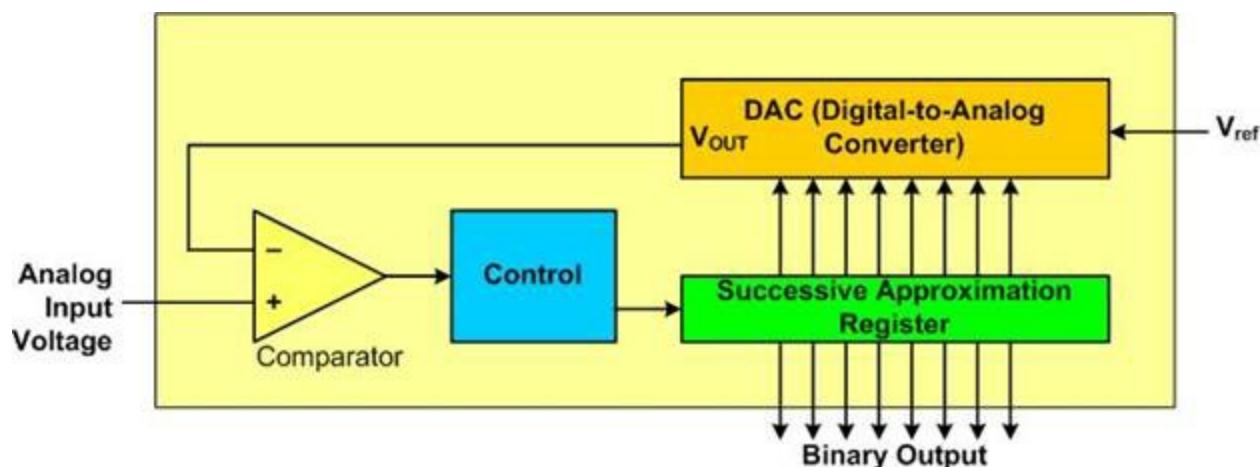


Figure 7-6: Successive Approximation ADC

The successive approximation register is loaded with only the most significant bit set at the start. An internal digital-to-analog converter converts the value of SAR to an analog voltage which is used to compare to the input voltage. If the input voltage is higher, the bit is kept. If the voltage is lower, the bit is cleared. The next bit is tried and the DAC and compare are exercised. This process is repeated for all bits of the SAR. Assuming a step size of 10 mV, the 8-bit successive approximation ADC will go through the following steps to convert an input of 1 Volt:

- (1) It starts with binary number 10000000. Since  $128 \times 10 \text{ mV} = 1.28 \text{ V}$  is greater than the 1 V input, bit 7 is cleared (dropped).
- (2) 01000000 gives us  $64 \times 10 \text{ mV} = 640 \text{ mV}$  and bit 6 is kept since it is smaller than the 1 V input.
- (3) 01100000 gives us  $96 \times 10 \text{ mV} = 960 \text{ mV}$  and bit 5 is kept since it is smaller than the 1 V input,
- (4) 01110000 gives us  $112 \times 10 \text{ mV} = 1120 \text{ mV}$  and bit 4 is dropped since it is greater than the 1 V input.
- (5) 01101000 gives us  $108 \times 10 \text{ mV} = 1080 \text{ mV}$  and bit 3 is dropped since it is greater than the 1 V input.
- (6) 01100100 gives us  $100 \times 10 \text{ mV} = 1000 \text{ mV} = 1 \text{ V}$  and bit 2 is kept since it is equal to input. Even though the answer is found it does not stop.
- (7) 011000110 gives us  $102 \times 10 \text{ mV} = 1020 \text{ mV}$  and bit 1 is dropped since it is greater than the 1 V input.
- (8) 01100101 gives us  $101 \times 10 \text{ mV} = 1010 \text{ mV}$  and bit 0 is dropped since it is greater than the 1 V input.

Notice that the Successive Approximation method goes through all the steps even if the answer is found in one of the earlier steps. The advantage of the Successive Approximation method is that the conversion time is fixed since it has to go through all the steps.

## Review Questions

1. Give two factors that affect the step size calculation.
2. The ADC0848 is a(n) \_\_\_\_\_ -bit converter.
3. True or false. While the ADC0848 has 8 pins for Dout, the MAX1112 has only one Dout pin.
4. Find the step size for an 8-bit ADC, if  $V_{ref} = 1.28 \text{ V}$ .
5. For question 4, calculate the output if the analog input is: (a) 0.7 V, and (b) 1 V.

## Section 7.2: ADC Programming with the Tiva TM4C123G

Because the ADC is widely used in data acquisition, in recent years an increasing number of microcontrollers have on-chip ADC modules. In this section we discuss the ADC feature of the TI Tiva TM4C123G and show how it is programmed.

The TI ARM Tiva TM4C123GH6PM comes with two on-chip ADC modules. These ADC modules have 12-bit resolution. To program them, we need to understand some of the major registers. Figure 7-7 shows a simplified block diagram of a Tiva ADC module.

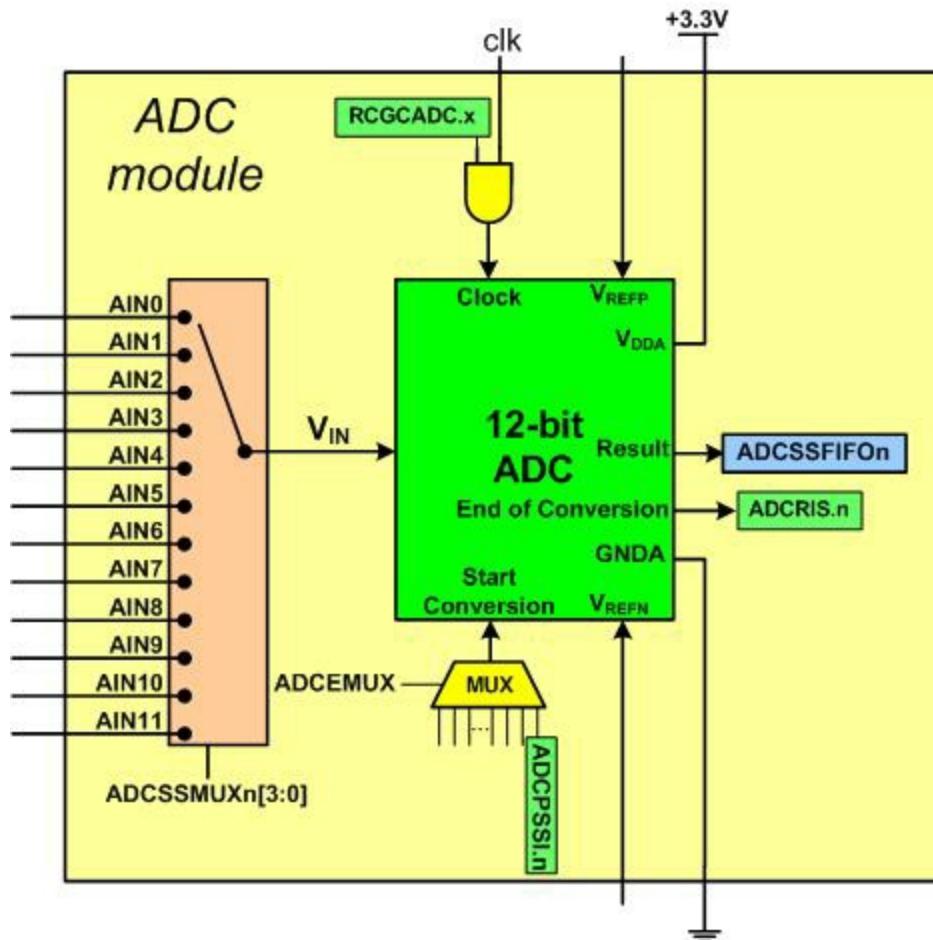


Figure 7-7: Simplified Block Diagram of a TI ADC Module

In this section, we examine some of these registers and show how to program the ADC.

### Enabling Clock to ADC

First thing we need to do is to enable the clock to the ADC0 or ADC1. Bit 0 and bit 1 of RCGCADC register are used to enable the clock to ADC0 and ADC1 modules, respectively. The RCGCADC is part of the System Control register and is located at base address of 0x400FE000 with offset 0x638. That means, the RCGCADC is located at physical address of 0x400FE638 ( $0x400FE000 + 0x638 = 0x400FE638$ ) in memory map. See Figure 7-8.

RCGCADC:	D31	.....	D2	D1	D0	0x638
Type	RO	Reserved		R1	R0	
			RO	R/W	R/W	
<b>bit</b>	<b>Name</b>	<b>Description</b>				
0	R0	0: ADC module 0 is disabled, 1: Enable and provide a clock to ADC module 0				
1	R1	0: ADC module 1 is disabled, 1: Enable and provide a clock to ADC module 1				

Figure 7-8: ADC Run Mode Clock Gating Control (RCGCADC)

## The Sample Sequencer

The Sample Sequencer is a part of the ADC module that moves the conversion result of the ADC to one of the FIFOs. There are 4 Sample Sequencers. They are called SS3, SS2, SS1, and SS0. Each one of them is associated with a FIFO. The FIFOs have different sizes so the sample sequencers have different lengths of sequences. The longest sequence has 8 samples and the shortest has only one. Table 7-4 shows the relation between the sample sequencers and FIFO sizes.

Sequencer	Number of Samples	Depth of FIFO
SS0	8	8
SS1	4	4
SS2	4	4
SS3	1	1

Table 7-4: Samples and FIFO Depth of Sequencers

In this section, we will use the SS3 with a single sample. We use ADCACTSS (ADC Active Sample Sequencer) to enable the SS3. When bit 3 (ASEN3) is set to 1 the SS3 is enabled. We must disable the SS3 before configuring the sample sequencer so that no erroneous events occur during initialization. After the initialization is done, we must enable it to use it.

ADCACTSS:	D31	.....	D17	D16	D15	.....	D4	D3	D2	D1	D0	0x000
Type	RO		RO	RO	RO		RO	R/W	R/W	R/W	R/W	

Figure 7-9: ADC Active Sample Sequencer (ADCACTSS)

## Start Conversion trigger options

There are many start-conversion (trigger) options. Among them are using timer, PWM, analog comparator, external signal from GPIO, and software. The selection of trigger for SS3 is done via the bits 15-12 of ADCEMUX register. The default is software and that is what we use in this section.

D31	.....	D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
ADCEMUX:	Reserved	EM3 EM2 EM1 EM0
Type	RO	R/W

0x014

EMx bits select the trigger source for Sample Sequencer x. By default the field is 0x0 which means the ADC conversion begins when the SSn bit of the ADCPSSI register is set by software. The following table shows the available choices for trigger. For more information see the datasheet.

EMx value	Trigger source
0x0	Processor (default)
0x1	Analog Comparator 0
0x2	Analog Comparator 1
0x3	reserved
0x4	External (GPIO Pins)
0x5	Timer
0x6	PWM generator 0
0x7	PWM generator 1
0x8	PWM generator 2
0x9	PWM generator 3
0xF	Always (continuously sample)

Figure 7-10: ADC Event Multiplexer Select (ADCEMUX)

After we select the software option bit (which is the default) bit in the ADCEMUX, we must use bit D3 of ADCPSSI register to start a conversion every time we want a new reading from the ADC input channel.

D31	.....	D28	D27	D26	.....	D4	D3	D2	D1	D0
ADCPSSI:	GSYNC	Reserved	SYNCWAIT	Reserved	SS3	SS2	SS1	SS0	0x028	
Type	R/W	RO	R/W	RO	RO	WO	WO	WO	WO	WO

SSx value	Trigger source
0	No effect
1	Begin sampling on sample sequence x

Figure 7-11: ADC Processor Sample Sequence Initiate (ADCPSSI)

Notice that we can trigger the SS3 option only if we have enabled the SS3 in the ADCACTSS register.

## Choosing V<sub>in</sub> input channel

The channel selection is done through the ADCSSMUX<sub>n</sub> (n=0,1,2,3) registers. For the SS3, the ADCSSMUX3 is used. Since SS3 only handles single conversion, bits 3 – 0 are used to specify the analog channel to be converted. The number of available channels in the TI Tiva TM4C123G varies among the family members. In the case of TI Tiva TM4C123GH6PM, there are 12 channels. They are designated as AIN0 (analog input 0) to AIN11 (analog input 11). Their designated pins are shown in Table 7-5.

Pin Name	Description	Pin	Pin Number
AIN0	ADC input 0	PE3	6

AIN1	ADC input 1	PE2	7
AIN2	ADC input 2	PE1	8
AIN3	ADC input 3	PE0	9
AIN4	ADC input 4	PD3	64
AIN5	ADC input 5	PD2	63
AIN6	ADC input 6	PD1	62
AIN7	ADC input 7	PD0	61
AIN8	ADC input 8	PE5	60
AIN9	ADC input 9	PE4	59
AIN10	ADC input 10	PB4	58
AIN11	ADC input 11	PB5	57

Table 7-5: Analog input pin assignment in TI Tiva TMC123GH6PM

ADCSSMUX3:	D31	.....	D4	D3	D2	D1	D0	0x0A0	
	.....	Reserved		MUX0					
<b>Type RO</b> RO      R/W      R/W      R/W      R/W									
bit	Name	Description							
0-3	MUX0	Sample Input Select							

Figure 7-12: ADC Sample Sequence Input Multiplexer Select 3 (ADCSSMUX3)

## Polling or interrupt

The end-of-conversion is indicated by a flag bit in the ADCRIS (ADC Raw Interrupt) register. Upon the completion of conversion for the SS3, the D3 bit (INT3) flag goes high. By polling this flag, we know if the conversion is complete and we can read the value in ADCSSFIFO3 register. We can also use an interrupt to inform us that the conversion is complete but that will require us to set up the interrupt mask ADCIM. By default, the interrupts are not enabled.

ADCRIS:	D31	.....	D28	D16	D15	.....	D4	D3	D2	D1	D0	0x004
	.....	Reserved		INRDC	Reserved			INR3	INR2	INR1	INR0	
<b>Type RO</b> RO      RO												

bit	Name	Description
0	INR0	SS0 Raw Interrupt Status 0: An interrupt has not occurred 1: A sample has completed conversion and the respective ADCSSCTL0 len bit is set, enabling a raw interrupt. Note: This bit is cleared by writing a 1 to the IN0 bit in the ADCISC register.
1	INR1	SS1 Raw Interrupt Status
2	INR2	SS2 Raw Interrupt Status
3	INR3	SS3 Raw Interrupt Status
16	INRDC	Digital Comparator Raw Interrupt Status 0: All bits in the ADCDCISC register are clear 1: At least one bit in the ADCDCISC register is set, meaning that a digital comparator interrupt has occurred.

Figure 7-13: ADC Raw Interrupt Status (ADCRIS)

## ADC Data result

Upon the completion of conversion, the binary result is placed in the ADCSSFIFO register. Since we are using SS3, we need to read the result from ADCSSFIFO3 register. This is 32-bit register but only the lower 12 bits are used.

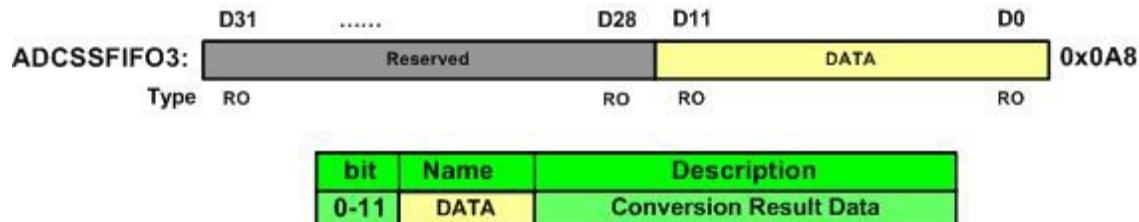


Figure 7-14: ADC Sample Sequence Result FIFO 3 (ADCSSFIFO3)

## Clearing end-of-conversion flag

After reading the data from the ADCSSFIFOx register, we must clear the INT3 flag bit in ADCRIS register so that we may detect another conversion complete. The raw interrupt flag in ADCRIS is cleared by writing to ADCISC (ADC Interrupt Status and Clear) register. By writing a 1 to bit 3 (IN3 bit) of ADCISC, the interrupt flag is cleared and we can do another conversion again.

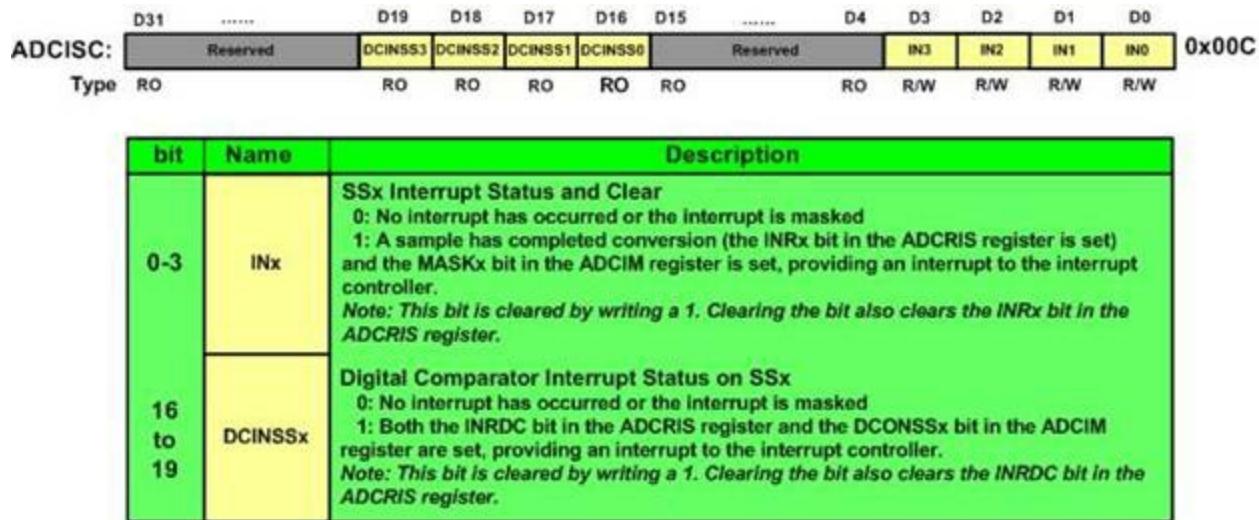


Figure 7-15: ADC Interrupt Status and Clear (ADCISC)

## Differential versus Single-Ended

In some applications, our interest is in the differences between two analog signal voltages (the differential voltages). Rather than converting two channels and calculate the differences between them, the Tiva TM4C123G has the option of converting the differential voltages of two analog channels. The bit 0 of ADCSSCTL3 (ADC Sample sequence Control 3) register allows us to enable the differential option. Upon Reset, the default is the single-ended input and we will leave it at that. The pairing of the analog inputs for differential is hardwired. Table 7-6 shows

the pairing of the ADC input channels for differential option.

Differential Pair	Analog Inputs
0	0 and 1
1	2 and 3
2	4 and 5
3	6 and 7
4	8 and 9
5	10 and 11

Table 7-6: ADC channel pairing for differential

The other bits in ADCSSCTL3 register are bit 1 (END0), bit 2 (IE0), and bit 3 (TS) bits. On some of the ADC inputs, we have an internal temperature sensor embedded into the chip. Making bit 3 = 1, reads the temperature sensor value inside the chip itself. Since we are using one sample in the SSE3, we must enable the bit 1 (END0) to let ADC know that first sample is the only sample and there is no sample coming after that. Bit 2 (IE0) is the Interrupt enable. It causes the raw interrupt flag to set when this sample conversion is completed. However, in order to redirect the end-of-conversion to the NVIC interrupt controller, we must also enable the bit in ADCIM register, as we will see soon. It must be noted, even if we do not want to use interrupt, we must still set IE0 = 1, in order to post the raw interrupt flag for polling the conversion complete.



bit	Name	Description
0	D0	<b>Sample Differential Input Select</b> 0: The analog inputs are not differentially sampled 1: The analog input is differentially sampled. The corresponding ADCSSMUXn nibble must be set to the pair number “i”, where the paired inputs are $2i$ and $2i+1$ . <i>Note: Because the temperature sensor does not have a differential option this bit must not be set when the TS0 bit is set.</i>
1	END0	<b>End of Sequence:</b> This bit must be set before initiating a single sample sequence. 0: Sampling and conversion continues 1: This is the end of sequence.
2	IE0	<b>Sample Interrupt Enable</b> 0: The raw interrupt is not asserted to the interrupt controller 1: The raw interrupt signal (INR0 bit) is asserted at the end of this sample's conversion. If the MASK0 bit in the ADCIM register is set, the interrupt is promoted to the interrupt controller. It is legal to have multiple samples within a sequence generate interrupts.
3	TS0	<b>1st Sample Temperature Sensor Select</b> 0: The input pin specified by the ADCSSMUXn register is read during the first sample of the sample sequence. 1: The temperature sensor is read during the first sample of the sample sequence.

Figure 7-16: ADC Sample Sequence Control 3 (ADCSSCTL3)

## Masking interrupt for SS3

Since we are using polling for the end-of-conversion, we must mask the interrupt option for the SS3 to prevent it from interrupting us via us NVIC. This is done with bit 3 of ADCIM (ADC

Interrupt Mask) register. Upon Reset, the default for the bit 3 (MASK3) is 0. With the MASK3 = 0, it disables the interrupt and we will leave it like that. However, if we like to handle end-of-conversion by interrupt, we need to set this bit to 1 and write an interrupt handler to read the conversion result.

bit	Name	Description
0-3	MASKx	<b>SSx Interrupt Mask</b> 0: The status of Sample Sequencer x does not affect the SSx interrupt status. 1: The raw interrupt signal from Sample Sequencer x (ADCRIS register INRx bit) is sent to the interrupt controller.
16 to 19	DCONSSx	<b>Digital Comparator Interrupt on SSx</b> 0: The status of the digital comparators does not affect the SSx interrupt status. 1: The raw interrupt signal from the digital comparators (INRDC bit in the ADCRIS register) is sent to the interrupt controller on the SSx interrupt line.

Figure 7-17: ADC Interrupt Mask (ADCIM)

## V<sub>ref</sub> in Tiva LunachPad

In the TI ARM Tiva chip series, the pin for V<sub>ref</sub> (+) is called VDDA (VDD analog) and V<sub>ref</sub> (-) pin is called GNDA (Ground Analog). In the TI Tiva LaunchPad, the VDDA pin is connected to 3.3V, the same supply voltage as the digital part of the chip. Even if we connect the VDDA to a separate power source other than the VDD of the chip, it cannot go beyond the VDD voltage. With V<sub>ref</sub>=3.3V, we have the step size of 3.3V / 4096= 0.8057 mV since the ADC resolution is 12 bits. See Example 7-2.

### Example 7-2

Give the digital converted output if the analog input voltage is 1.2V for the TI Tiva LaunchPad.

#### Solution:

Since the step size is 3.3V / 4096 = 0.8057 mV, we have 1.2V / 0.8057 mV = 1489 = 0x5D1 as ADC output.

## Configuring GPIO for ADC input

In using ADC, we must also configure the GPIO pins to allow the connection of an analog signal through the input pin. In this regard, it is the same as all other peripherals. The steps are as follow:

1. Enable the clock to GPIO pin by using RCGCGPIO.
2. Set the GPIOAFSEL (GPIO alternate function) bit for ADC input pin to 1.

3. Configure AINx signal to be used as analog input by clearing the bit in the GPIODEN (GPIO Digital enable) register.
4. Disable the Analog isolation circuit for ADC input pins by writing a 1 to the GPIOAMSEL register.

ADC Channel	Pin
AIN0	PE3
AIN1	PE2
AIN2	PE1
AIN3	PE0
AIN4	PD3
AIN5	PD2
AIN6	PD1
AIN7	PDO
AIN8	PE5
AIN9	PE4
AIN10	PB4
AIN11	PB5

Table 7-7: ADC Channel pin designation

## Configuring ADC and reading ADC channel

After the GPIO configuration, we need to take the following steps to configure the ADC for Sample Sequencer 3 (SS3):

1. Enable the clock to ADC0 or ADC1 modules using RCGCADC register of System Registers. The  $\text{SYSCTL}\rightarrow\text{RCGCADC} \mid= 3$  will enable the clock to both ADC0 and ADC1 modules. For ADC0 module use  $\text{SYSCTL}\rightarrow\text{RCGCADC} \mid= 1$
2. Disable the Sample Sequencer using the ADCACTSS register before changing the configuration of the sequencer.  $\text{ADC0}\rightarrow\text{ACTSS} \&= \sim 8$  disables the SS3.
3. Choose the software trigger using the ADCEMUX register. Use  $\text{ADC0}\rightarrow\text{EMUX} \&= \sim 0xF000$  for software trigger.
4. Select the ADC input channel using the ADCSSMUXn register. In the case of SS3 we use the ADCSSMUX3. For example, The  $\text{ADC0}\rightarrow\text{SSMUX3} = 0$  selects AIN0 channel on pin PE3.
5. Select the single-ended option, one-conversion per sample, and raw interrupt bit for end-of-conversion using the ADCSSCTL3 register. Use  $\text{ADC0}\rightarrow\text{SSCTL3} \mid= 6$  for single-ended, one-conversion, and raw interrupt.
6. Enable the Sample Sequencer SS3 using ADCACTSS register.  $\text{ADC0}\rightarrow\text{ACTSS} \mid= 8$  enables the SS3.
7. Use ADCPSSI register to start a new conversion. Use  $\text{ADC0}\rightarrow\text{PSSI} \mid= 8$  to start a conversion by Sample Sequencer 3.
8. Keep monitoring the end-of-conversion INT3 flag in ADCRIS register.
9. When the INT3 goes HIGH, read the ADC result from the ADCSSFIFO3 and save it.
10. After reading the ADC result in step 9, clear the INT3 flag in ADCRIS register to allow for the next conversion. To clear INT3 flag, write to ADCICS register with  $\text{ADC0}\rightarrow\text{ISC} = 8$ .

## 11. Repeat steps 8 through 10 for the next conversion.

Program 7-1 illustrates the steps for ADC conversion shown above. Figure 7-18 shows the hardware connection of Program 7-1.

### Program 7-1: Using ADC0 to convert input from AIN0

```
/* p7_1.c: A to D conversion */

/* This program converts the analog input from AIN0 (J3.9 of LaunchPad) using sample
sequencer 3 and software trigger continuously.
Notice from Table 7-7, AIN0 channel is on PE3 pin. */

#include "TM4C123GH6PM.h"

int main(void)
{
    volatile int result;

    /* enable clocks */
    SYSCTL->RCGCGPIO |= 0x10;      /* enable clock to GPIOE (AIN0 is on PE3) */
    SYSCTL->RCGCADC |= 1;          /* enable clock to ADC0 */

    /* initialize PE3 for AIN0 input */
    GPIOE->AFSEL |= 8;            /* enable alternate function */
    GPIOE->DEN &= ~8;             /* disable digital function */
    GPIOE->AMSEL |= 8;            /* enable analog function */

    /* initialize ADC0 */
    ADC0->ACTSS &= ~8;           /* disable SS3 during configuration */
    ADC0->EMUX &= ~0xF000;        /* software trigger conversion */
    ADC0->SSMUX3 = 0;             /* get input from channel 0 */
    ADC0->SSCTL3 |= 6;            /* take one sample at a time, set flag at 1st sample */
    ADC0->ACTSS |= 8;             /* enable ADC0 sequencer 3 */

    while(1)
    {
        ADC0->PSSI |= 8;          /* start a conversion sequence 3 */
        while((ADC0->RIS & 8) == 0); /* wait for conversion complete */
        result = ADC0->SSFIFO3; /* read conversion result */
        ADC0->ISC = 8;             /* clear completion flag */
    }
}

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}
```

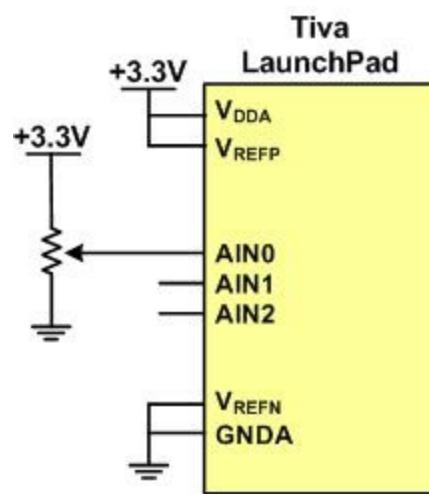


Figure 7-18: ADC Connection for Program 7-1

## Timer trigger conversion

For digital signal processing, not only the precision of the ADC conversion result is important, the precision of the time that the analog input is sampled is also important. Using software trigger conversion does not yield precision timing especially when there are other software tasks running at the same time. One way of getting precision sampling timing interval is to use the timer to trigger the conversion. To do so, three steps must be taken: (1) configure the ADC to use timer trigger, (2) configure a timer to generate periodic timeout, and (3) connect timer to ADC trigger.

For step 1, the four bits of ADCEMUX register for the sample sequencer used should be loaded with the value of 5 for timer trigger. The periodic timer function was covered in chapter 5 already. To connect the timer to ADC trigger, the TAOTE or TBOTE bit of the GPTMCTL should be set to 1. Program 7-2 shows an example of using Wide Timer0A to trigger the ADC.

## Temperature sensor

There is an internal temperature sensor embedded into the TI ARM Tiva chip. This gives us the temperature inside the ARM chip itself. We can use the following formula to calculate the Temperature in Celsius:

$$\text{Temp} = 147.5 - ((75 \times \text{Vref}(+)) - \text{Vref}(-)) \times \text{ADC\_output}) / 4096$$

Since the  $\text{Vref} = 3.3 \text{ V}$  in TI Tiva Launchpad, we have:

$$\text{Temp} = 147.5 - ((75 \times 3.3\text{V}) \times \text{ADC\_output}) / 4096$$

$$\text{Temp} = 147.5 - (247.5 \times \text{ADC\_output}) / 4096$$

See Example 7-3.

### Example 7-3

Find the temperature inside the ARM chip for the TI Tiva Launchpad if the reading of ADC output is = 0x7D0.

**Solution:**

The 0x7D0 is 2000 in decimal. Now, we have

$$\text{Temp} = 147.5 - (247.5 \times 2000) / 4096 = 147.5 - 120.85 = 26.6 \text{ Celsius}$$

**Program 7-2:  
Converting the on-chip temperature sensor with timer trigger**

```

/* p7_2.c: Convert on-chip temperature */

/* This program converts the on-chip temperature sensor output using sample sequencer 3
and timer trigger at 1 Hz. */

#include "TM4C123GH6PM.h"

int main(void)
{
    volatile int temperature;

    /* enable clocks */
    SYSCTL->RCGCADC |= 1;           /* enable clock to ADC0 */
    SYSCTL->RCGCWTIMER |= 1;         /* enable clock to WTimer Block 0 */

    /* initialize ADC0 */
    ADC0->ACTSS &= ~8;              /* disable SS3 during configuration */
    ADC0->EMUX &= ~0xF000;
    ADC0->EMUX |= 0x5000;
    ADC0->SSMUX3 = 0;
    ADC0->SSCTL3 |= 0x0E;
    ADC0->ACTSS |= 8;               /* take chip temperature, set flag at 1st sample */
                                    /* enable ADC0 sequencer 3 */

    /* initialize wtimer 0 to trigger ADC at 1 sample/sec */
    WTIMER0->CTL = 0;               /* disable WTimer before initialization */
    WTIMER0->CFG = 0x04;             /* 32-bit option */
    WTIMER0->TAMR = 0x02;            /* periodic mode and down-counter */
    WTIMER0->TAILR = 16000000;        /* WTimer A interval load value reg (1 s) */
    WTIMER0->CTL |= 0x20;             /* timer triggers ADC */
    WTIMER0->CTL |= 0x01;             /* enable WTimer A after initialization */

    while(1)
    {
        while((ADC0->RIS & 8) == 0); /* wait for conversion complete */
        temperature = 147 - (247 * ADC0->SSFIFO3) / 4096;
        ADC0->ISC = 8;                /* clear completion flag */
    }
}

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## Digital Comparator unit

Many microcontrollers come with digital comparator for monitoring an analog input voltage. Using the digital comparator, we can monitor an analog input against two threshold values to determine whether the analog input value is above high threshold, between two threshold values or below low threshold. When the analog input falls in the selected region, the comparator may generate an interrupt or trigger an A-to-D conversion and let the conversion completion generate an interrupt. Using digital comparator, the software does not have to continuously monitor the analog input.

For example, we may program the comparator for the on-chip temperature sensor. When the temperature exceeds a preset threshold, the cooling fan is turned on. When the temperature drops below the threshold, the cooling fan is turned off. The software only has to set the threshold and interrupt. It does not have to use the ADC continuously to monitor the temperature.

For the details of programming the digital comparator unit, we will leave it to the reader.

## Review Questions

1. The ADC in TI ARM Tiva is \_\_\_\_\_ bit.
2. The highest number we can form for the ADC output is \_\_\_\_\_ in hex.
3. Assume  $V_{ref} = 3.3V$ . Find the ADC output in decimal and hex if  $V_{in}$  of analog input is 1.9V.
4. IN TI ARM Tiva, which register provides the ADC output converted data if we use SS3?

## Section 7.3: Sensor Interfacing and Signal Conditioning

This section will show how to interface sensors to the microcontroller. We examine some popular temperature sensors and then discuss the issue of signal conditioning. Although we concentrate on temperature sensors, the principles discussed in this section are the same for other types of sensors such as light and pressure sensors.

### Temperature sensors

*Transducers* convert physical data such as temperature, light intensity, flow, and speed to electrical signals. Depending on the transducer, the output produced is in the form of voltage, current, resistance, or capacitance. For example, temperature is converted to electrical signals using a transducer called a *thermistor*. A thermistor responds to temperature change by changing resistance, but its response is not linear, as seen in Table 7-8 and Figure 7-19.

Temperature ('C)	T <sub>f</sub> (K ohms)
0	29.490
25	10.000
50	3.893
75	1.700
100	0.817

Table 7-8: Thermistor Resistance vs. Temperature

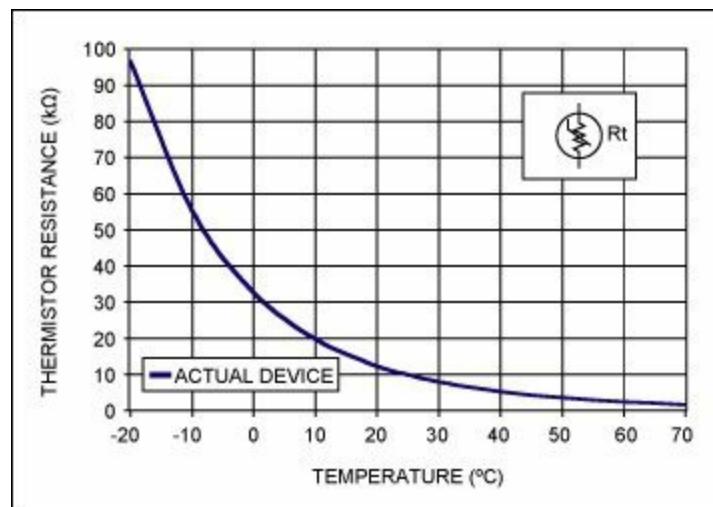


Figure 7-19: Thermistor (Copied from <http://www.maximintegrated.com>)

The resistance of a thermistor is typically modeled by Steinhart-Hart equation and requires a logarithmic amplifier to produce a linear output. The complexity associated with the circuit for such nonlinear devices has led many manufacturers to market a linear temperature sensor. Simple and widely used linear temperature sensors include the LM34 and LM35 series from National Semiconductor (now part of TI Corp.) They are discussed next.

### LM34 and LM35 temperature sensors

The sensors of the LM34 series are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Fahrenheit temperature. See Figure 7-20. The LM34 requires no external calibration because it is internally calibrated. It outputs 10 mV

for each degree of Fahrenheit temperature.

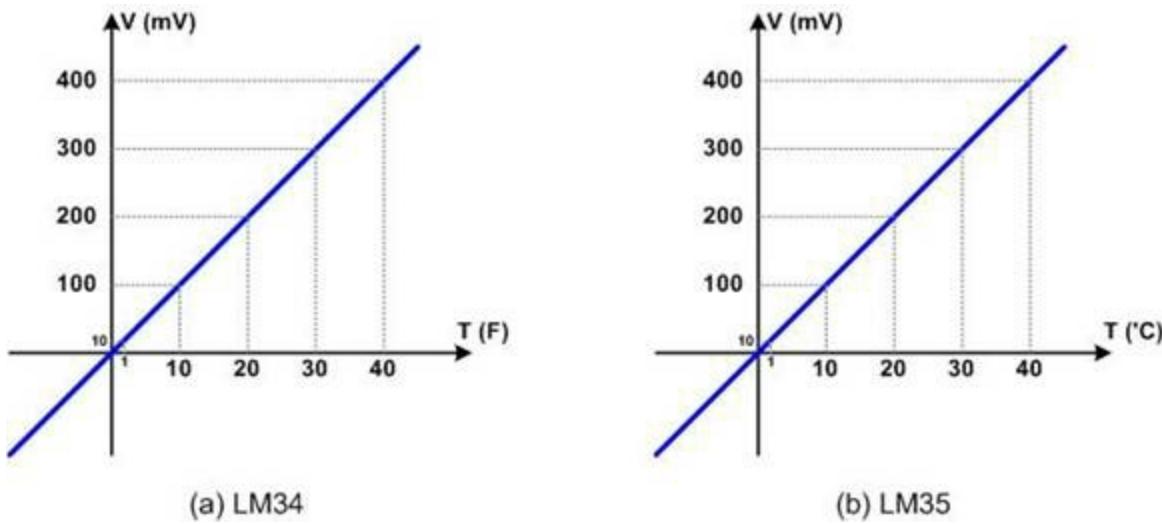


Figure 7-20: LM34 and LM35

The LM35 series sensors are similar to LM34 series sensors except that the output voltage is linearly proportional to the Celsius (centigrade) temperature. It outputs 10 mV for each degree of centigrade temperature. See Figure 7-20.

## Signal conditioning

The common transducers produce an output in the form of voltage, current, charge, capacitance, or resistance. In order to perform A-to-D conversion on these signals, they need to be converted to voltage unless the transducer output is already voltage. In addition to the conversion to voltage, the signal may also need gain and offset adjustment to achieve optimal dynamic range. A low-pass analog filter is often incorporated in the signal conditioning circuit to eliminate the high frequency to avoid aliasing. Figure 7-21 shows a block diagram of the input of a data acquisition system.

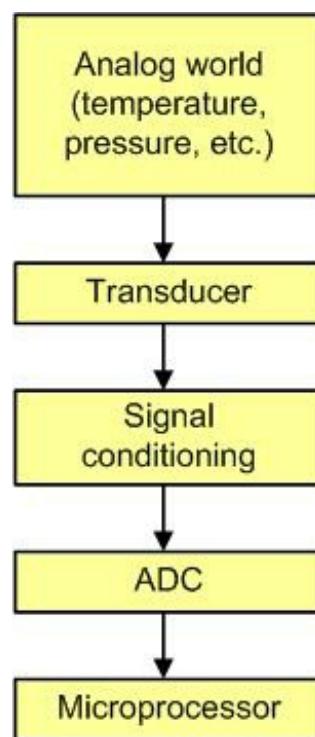


Figure 7-21: Getting Data to the CPU

## Interfacing the LM34 to the TI Tiva TM4C123G Microcontroller

The A/D of TI Tiva TM4C123G Microcontroller has 12-bit resolution with a maximum of 4096 steps, and the LM34 produces 10 mV for every degree of temperature change. The maximum operating temperature of the LM34 is 300 degrees F, so the highest output will be 3000 mV (3.00 V), which is below 3.3V of  $V_{ref}$ . The LM34/35 can be connected to the microcontroller as shown in Figure 7-22.

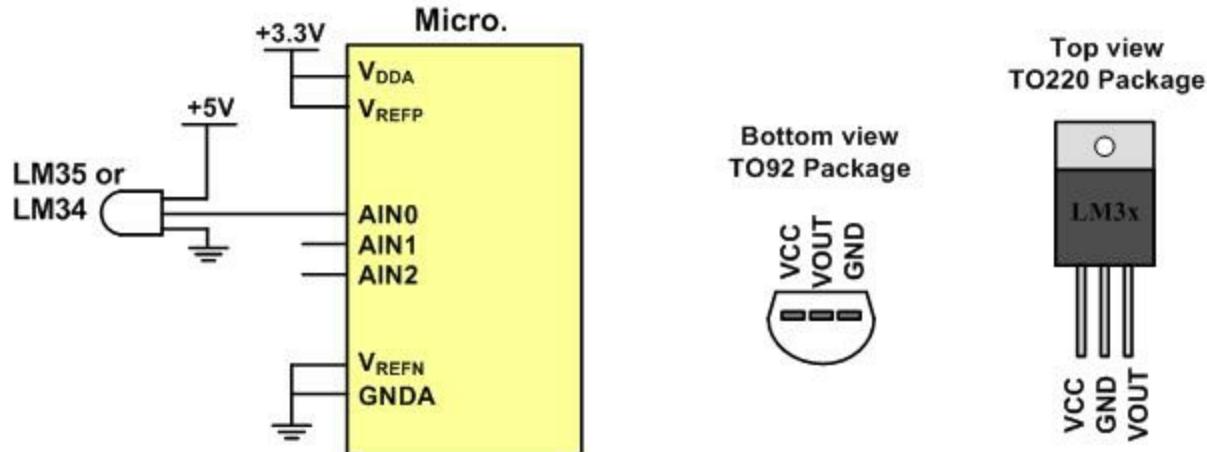


Figure 7-22: LM34/35 Connection to ARM and Its Pin Configuration

To convert the ADC result to temperature in degree, use the equation:

$$\text{temperature} = \text{ADC0} \rightarrow \text{SSFIFO3} * 330 / 4096;$$

## Reading and displaying temperature

Programs 7-3 shows code for reading and displaying temperature in C. Notice that in Figure 7-22, the LM34 (or LM35) is connected to channel 0 (ADC0 pin).

### Program 7-3: Reading Temperature Sensor in F

```
/* p7_3: Interfacing LM34 temperature sensor */
/* LM34 is connected to Chan0. See Figure 7-22 */

/* This program converts the analog output from an LM34 and convert it to temperature
in Fahrenheit.
Notice from Table 7-5, AIN0 channel is on PE3 pin Temp is sent to UART to be viewed on
TeraTerminal */

#include "TM4C123GH6PM.h"
#include <stdio.h>

void UART0Tx(char c);
void UART0_init(void);
void UART0_puts(char* s);
void delayMs(int n);

int main(void)
{
```

```

int temperature;
char buffer[16];

/* initialize UART0 for output */
UART0_init();

/* enable clocks */
SYSCTL->RCGCGPIO |= 0x10; /* enable clock to GPIOE */
SYSCTL->RCGCADC |= 1; /* enable clock to ADC0 */

/* initialize PE3 for AIN0 input */
GPIOE->AFSEL |= 8; /* enable alternate function */
GPIOE->DEN &= ~8; /* disable digital function */
GPIOE->AMSEL |= 8; /* enable analog function */

/* initialize ADC0 */
ADC0->ACTSS &= ~8; /* disable SS3 during configuration */
ADC0->EMUX &= ~0xF000; /* software trigger conversion */
ADC0->SSMUX3 = 0; /* get input from channel 0 */
ADC0->SSCTL3 |= 6; /* take one sample at a time, set flag at 1st sample */
ADC0->ACTSS |= 8; /* enable ADC0 sequencer 3 */

while(1)
{
    ADC0->PSSI |= 8; /* start a conversion sequence 3 */
    while((ADC0->RIS & 8) == 0); /* wait for conversion complete */
    temperature = ADC0->SSFIFO3 * 330 / 4096;
    ADC0->ISC = 8; /* clear completion flag */
    sprintf(buffer, "\r\nTemp = %dF", temperature);
    UART0_puts(buffer);
    delayMs(1000);
}
}

void UART0_init(void)
{
    SYSCTL->RCGCUART |= 1; /* provide clock to UART0 */
    SYSCTL->RCGCGPIO |= 0x01; /* enable clock to GPIOA */

    /* UART0 initialization */
    UART0->CTL = 0; /* disable UART0 */
    UART0->IBRD = 104; /* 16MHz/16=1MHz, 1MHz/104=9600 baud rate */
    UART0->FBRD = 11; /* fraction part, see Example 4-4 */
    UART0->CC = 0; /* use system clock */
    UART0->LCRH = 0x60; /* 8-bit, no parity, 1-stop bit, no FIFO */
    UART0->CTL = 0x301; /* enable UART0, TXE, RXE */

    /* UART0 TX0 and RX0 use PA0 and PA1. Set them up. */
    GPIOA->DEN = 0x03; /* Make PA0 and PA1 as digital */
    GPIOA->AFSEL = 0x03; /* Use PA0,PA1 alternate function */
    GPIOA->PCTL = 0x11; /* configure PA0 and PA1 for UART */
}

void UART0Tx(char c)
{
    while((UART0->FR & 0x20) != 0); /* wait until Tx buffer not full */
    UART0->DR = c; /* before giving it another byte */
}

```

```

void UART0_puts(char* s)
{
    while (*s != 0)          /* if not end of string */
        UART0Tx(*s++);      /* send the character through UART0 */
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int32_t i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

void SystemInit(void)
{
    /* Grant coprocessor access
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## Review Questions

1. True or false. The transducer must be connected to signal conditioning circuitry before its signal is sent to the ADC.
2. The LM35 provides \_\_\_\_\_ mV for each degree of \_\_\_\_\_ (Fahrenheit, Celsius) temperature.
3. The LM34 provides \_\_\_\_\_ mV for each degree of \_\_\_\_\_ (Fahrenheit, Celsius) temperature.
4. What is the temperature if the ADC output is 0000 0011 1110?

## Section 7.4: Interfacing to a DAC

This section will discuss the fundamentals of a DAC (digital-to-analog converter). Then we demonstrate how to generate a saw tooth wave on the scope using the DAC.

### Digital-to-analog (DAC) converter

The digital-to-analog converter (DAC) is a device widely used to convert digital signals to analog signals. In this section we discuss the basics of a DAC.

Recall from your digital electronics book the two methods of making a DAC: binary weighted and R/2R ladder. The vast majority of integrated circuit DACs, including the DAC0808 used in this section, use the R/2R method since it can achieve a much higher degree of precision. The first criterion for selecting a DAC is its resolution, which is a function of the number of bits of the digital input. The common ones are 8, 10, and 12 bits. The number of digital input bits decides the resolution of the DAC since the number of analog output levels is equal to  $2^n$ , where n is the number of digital input bits. Therefore, the 8-bit DAC such as the DAC0808 provides 256 discrete voltage (or current) levels of output. See Figure 7-23.

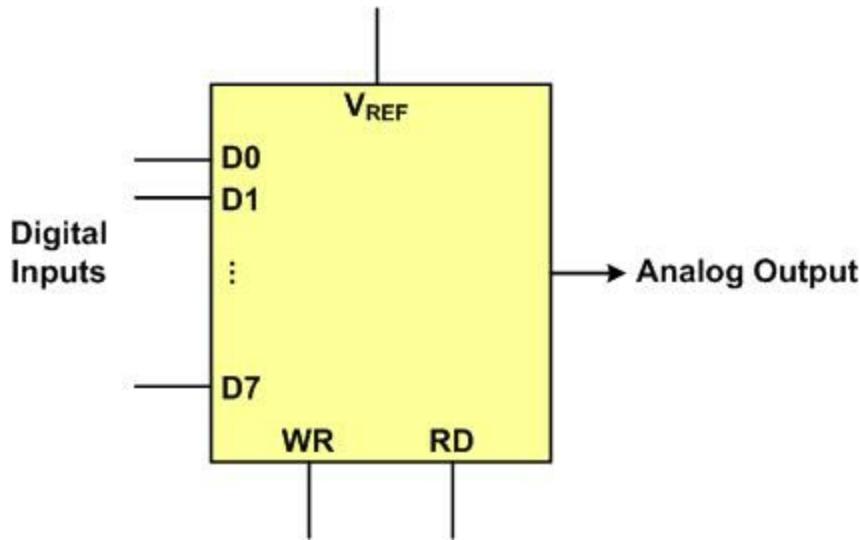


Figure 7-23: DAC Block Diagram

Similarly, the 12-bit DAC provides 4096 discrete voltage levels. Although there are 16-bit DACs, they are much more expensive.

### DAC0808

In the DAC0808, the digital inputs are converted to current ( $I_{OUT}$ ). By connecting a resistor to the  $I_{OUT}$  pin, we convert the conversion result current to voltage. The total current provided by the  $I_{OUT}$  is a function of the binary numbers at the D0–D7 inputs of the DAC0808 and the reference current ( $I_{ref}$ ), and is as follows:

$$I_{OUT} = I_{ref} \times (D7 / 2 + D6 / 4 + D5 / 8 + D4 / 16 + D3 / 32 + D2 / 64 + D1 / 128 + D0 / 256) = I_{ref} \times \text{Data} / 256$$

where D0 is the LSB, D7 is the MSB for the inputs, and  $I_{ref}$  is the reference input current

that must be applied to pin 14. The  $I_{ref}$  current is generally set to 2.0 mA. Figure 7-24 shows the generation of current reference (setting  $I_{ref} = 2$  mA) by using the standard 5-V power supply and 5K ohm resistors.

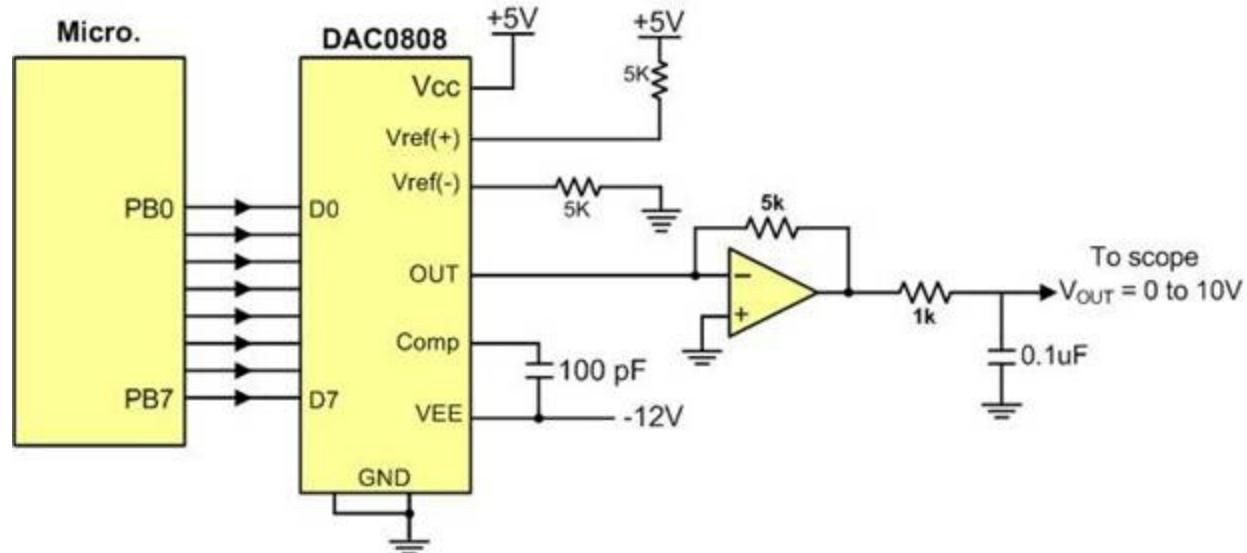


Figure 7-24: Microcontroller Connection to DAC0808

Some also use the Zener diode reference voltage device (LM336), which overcomes fluctuations associated with the power supply voltage. Now assuming that  $I_{ref} = 2$  mA, if all the input bits to the DAC are high, the maximum output current is 1.99 mA (verify this for yourself).

### Converting $I_{out}$ to voltage in DAC0808

We connect the output pin  $I_{OUT}$  to a resistor, convert this current to voltage, and monitor the output on the scope. However, in real life this can cause inaccuracy since the input resistance of the load where it is connected will also affect the output voltage. For this reason, the  $I_{ref}$  current output is buffered by connecting it to an op amp such as the 741 with  $R_f = 5K$  ohms for the feedback resistor. Assuming that  $R = 5K$  ohms, by changing the binary input, the output voltage changes as shown in Example 7-4.

### Example 7-4

Assuming that  $R = 5K$  and  $I_{ref} = 2$  mA, calculate  $V_{out}$  for the following binary inputs:

- (a) 10011001 binary (0x99)      (b) 11001000 (0xC8)

### Solution:

- (a)  $I_{out} = 2$  mA ( $153/255$ ) = 1.195 mA and  $V_{out} = 1.195$  mA  $\times$  5K = 5.975 V  
 (b)  $I_{out} = 2$  mA ( $200/256$ ) = 1.562 mA and  $V_{out} = 1.562$  mA  $\times$  5K = 7.8125 V

### Generating a stair-step ramp

In order to generate a stair-step ramp, you can set up the circuit in Figure 7-24 and load

Program 7-4 on the microcontroller. To see the result wave, connect the output to an oscilloscope. Figure 7-25 shows the output.

#### Program 7-4: Generating Saw Tooth Wave

```
/* p7_4.c: DAC programming to generate saw tooth wave*/  
  
#include "TM4C123GH6PM.h"  
  
void delayMs(int n);  
  
int main(void)  
{  
    uint8_t i = 0;  
    SYSCTL->RCGCGPIO |= 0x02; /* enable clock to GPIOB */  
  
    /* PORTB for DAC */  
    GPIOB->DIR = 0xFF;           /* PORTB as output */  
    GPIOB->DEN = 0xFF;           /* PORTB as digital pins */  
  
    for (;;) {  
        GPIOB->DATA = i;          /* send out i */  
        i++;  
        delayMs(1);  
    }  
}  
  
/* delay n milliseconds (16 MHz CPU clock) */  
void delayMs(int n)  
{  
    int i, j;  
    for(i = 0 ; i < n; i++)  
        for(j = 0; j < 3180; j++)  
            {} /* do nothing for 1 ms */  
}  
  
/* This function is called by the startup assembly code to perform system specific  
initialization tasks. */  
void SystemInit(void)  
{  
    /* Grant coprocessor access */  
    /* This is required since TM4C123G has a floating point coprocessor */  
    SCB->CPACR |= 0x00f00000;  
}
```

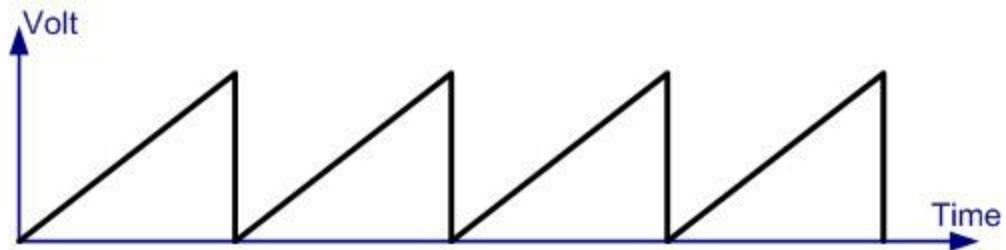


Figure 7-25: Saw Tooth WaveForm

## Generating a sine wave

To generate a sine wave, we first need a table whose values represent the magnitude of the sine of angles between 0 and 360 degrees. The values for the sine function vary from -1.0 to +1.0 for 0 to 360 degree angles. Therefore, the table values are integer numbers representing the voltage magnitude for the sine of the angle. This method ensures that only integer numbers are output to the DAC by the processor. Table 7-9 shows the angles, the sine values, the voltage magnitudes, and the integer values representing the voltage magnitude for each angle with 30-degree increments.

Angle $\Theta$ (degrees)	Sin $\Theta$	$V_{OUT}$ (Voltage Magnitude) $5V + (5V \times \sin \Theta)$	Values Sent to DAC (decimal) (Voltage Mag. $\times 25.6$ )
0	0	5	128
30	0.5	7.5	192
60	0.866	9.33	238
90	1.0	10	255
120	0.866	9.33	238
150	0.5	7.5	192
180	0	5	128
210	-0.5	2.5	64
240	-0.866	0.669	17
270	-1.0	0	0
300	-0.866	0.669	17
330	-0.5	2.5	64
360	0	5	128

Table 7-9: Angle vs. Voltage Magnitude for Sine Wave

To generate Table 7-9, we assumed the full-scale voltage of 10V for the DAC output. Full-scale output of the DAC is achieved when all the data input bits of the DAC are high. Therefore, to achieve the full-scale 10V output, we use the following equation:

$$V_{OUT} = 5V + (5V \times \sin \Theta)$$

To find the value sent to the DAC for various angles, we simply multiply the  $V_{OUT}$  voltage by 25.6 because there are 256 steps and full-scale  $V_{OUT}$  is 10 volts. Therefore, 256 steps / 10 V = 25.6 steps per volt. To further clarify this, look at Example 7-5.

### Example 7-5

Verify the values of Table 7-9 for the following angles: (a) 30 (b) 60.

**Solution:**

$$(a) V_{OUT} = 5 V + (5 V \times \sin \Theta) = 5 V + 5 \times \sin 30 = 5 V + 5 \times 0.5 = 7.5 V$$

$$\text{DAC input values} = 7.5 V \times 25.6 = 192 \text{ (decimal)}$$

$$(b) V_{OUT} = 5 \text{ V} + (5 \text{ V} \times \sin \Theta) = 5 \text{ V} + 5 \times \sin 60 = 5 \text{ V} + 5 \times 0.866 = 9.33 \text{ V}$$

DAC input values =  $9.33 \text{ V} \times 25.6 = 238$  (decimal)

The following program sends the values of Table 7-9 to the DAC. See Figure 7-26.

### Program 7-5: Generating Sine Wave

```
/* p7_5.c: DAC programming to generate sine wave */
#include "TM4C123GH6PM.h"

void delayMs(int n);

uint8_t sineWaveLookup[]={128,192,238,255,238,192,128,64,17,0,17,64};

int main(void)
{
    uint8_t i = 0;
    SYSCTL->RCGCGPIO |= 0x02; /* enable clock to GPIOB */

    /* PORTB for DAC */
    GPIOB->DIR = 0xFF;          /* PORTB as output */
    GPIOB->DEN = 0xFF;          /* PORTB as digital pins */

    for (;;)
    {
        for (i = 0; i < 12; i++)
        {
            GPIOB->DATA = sineWaveLookup[i];
            i++;
            delayMs(5);
        }
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}
```

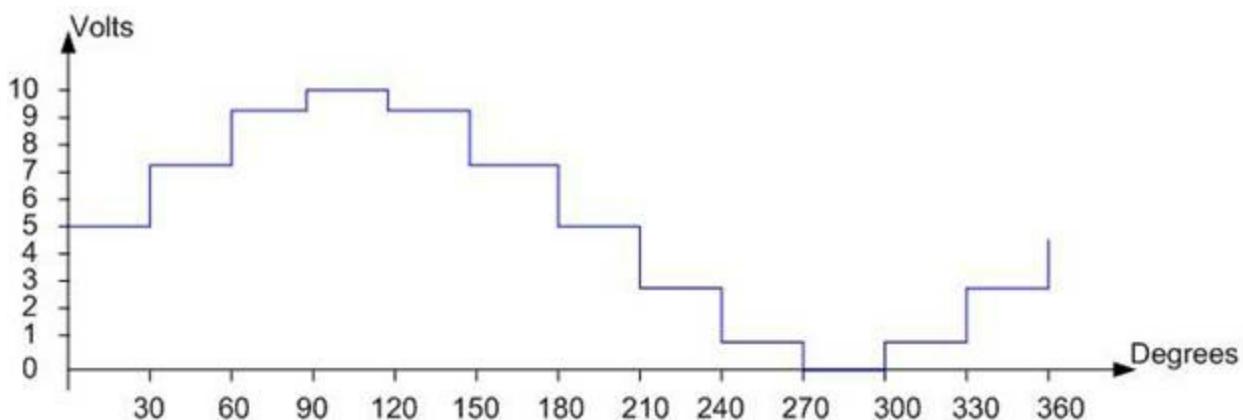


Figure 7-26: Angle vs. Voltage Magnitude for Sine Wave

Program 7-6 uses the C math library functions to generate sine wave.

#### Program 7-6: Generating Sine Wave Using Math Functions

```
/* p7_6.c: DAC programming to generate sine wave using math functions */

#include <math.h>

#define WAVEFORM_LENGTH 1024
int sinewave[WAVEFORM_LENGTH];

int main(void)
{
    int i;
    float fRadians;
#ifndef M_PI
    const float M_PI = 4 * atan(1.0);
#endif

    /* construct data table for a sine wave */
    fRadians = ((2 * M_PI) / WAVEFORM_LENGTH);
    for (i = 0; i < WAVEFORM_LENGTH; i++)
    {
        sinewave[i] = 500 * (sinf(fRadians * i) + 1);
    }
}
```

See Chapter 8 for the ARM connection and programming of LTC1661 DAC using SPI bus.

#### Review Questions

1. In a DAC, input is \_\_\_\_\_ (digital, analog) and output is \_\_\_\_\_ (digital, analog).
2. DAC0808 is a(n) \_\_\_\_\_-bit D-to-A converter.
3. The output of DAC0808 is in \_\_\_\_\_ (current, voltage).

#### Answers to Review Questions

## Section 7.1

1. Number of steps and  $V_{ref}$  voltage
2. 8
3. True
4.  $1.28 \text{ V} / 256 = 5 \text{ mV}$
5.
  - (a)  $0.7 \text{ V} / 5 \text{ mV} = 140$  in decimal and  $D7-D0 = 10001100$  in bin
  - (b)  $1 \text{ V} / 5 \text{ mV} = 200$  in decimal and  $D7-D0 = 11001000$  in binary.

## Section 7.2

1. 12
2. 0xFFFF
3. Step size is  $3.3\text{V} / 4096 = 0.8057 \text{ mv}$  and  $1.9\text{V} / 0.809\text{mv} = 2358$  in decimal or 0x936.
4. ADCSSFIFO3

## Section 7.3

1. True
2. 10, Celsius
3. 10, Fahrenheit
4.  $00111110$  (binary) = 62  $\rightarrow$  Temperature =  $62 \times 330 / 4096 = 5$

## Section 7.4

1. Digital, analog
2. 8
3. current



## Chapter 8: SPI Protocol and DAC Interfacing

The SPI (serial peripheral interface) is a bus interface incorporated in many devices such as ADC, DAC, and EEPROM. In Section 8.1 we will examine the signals of the SPI bus and show how the read and write operations in the SPI work. Section 8.2 examines the TI ARM Tiva SPI registers. In Section 8.3 we show LTC1661 DAC interfacing to ARM using SPI bus.

## Section 8.1: SPI Bus Protocol

The SPI bus was originally started by Motorola (now Freescale), but in recent years has become a widely used by many semiconductor chip companies. SPI devices use only 2 pins for data transfer, called SDI (Din) and SDO (Dout), instead of the 8 or more pins used in traditional buses. This reduction of data pins reduces the package size and power consumption drastically, making them ideal for many applications in which space is a major concern. The SPI bus has the SCLK (serial clock) pin to synchronize the data transfer between two chips. The last pin of the SPI bus is CE (chip enable), which is used to initiate and terminate the data transfer. These four pins, SDI, SDO, SCLK, and CE, make the SPI a 4-wire interface. See Figure 8-1.

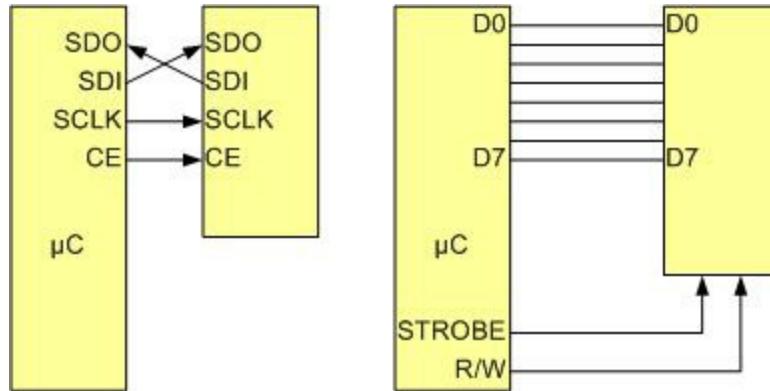


Figure 8-1: SPI Bus vs. Traditional Parallel Bus Connection to Microcontroller

In many chips, the SDI, SDO, SCLK, and CE signals are alternatively named as MOSI, MISO, SCK, and SS as shown in Figure 8-2 (compare with Figure 8-1). There is also a widely used standard called a 3-wire interface bus. In a 3-wire interface bus, we have SCLK and CE, and only a single pin for data transfer. The SPI 4-wire bus can become a 3-wire interface when the SDI and SDO data pins are tied together. However, there are some major differences between the SPI and 3-wire devices in the data transfer protocol. For that reason, a device must support the 3-wire protocol internally in order to be used as a 3-wire device. Many devices support both SPI and 3-wire protocols.

### How SPI works

SPI consists of two shift registers, one in master and the other in the slave side. Also there is a clock generator in the master side that generates the clock for the shift registers.

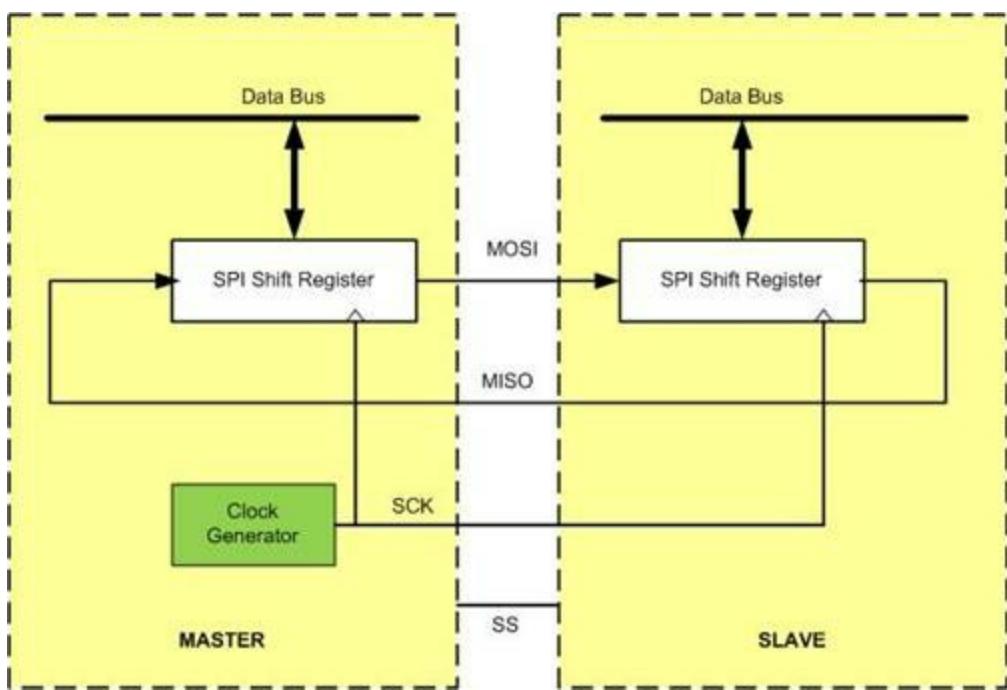


Figure 8-2: SPI Architecture

As you can see in Figure 8-2, serial-out pin of the master shift register is connected to the serial-in pin of the slave shift register by MOSI (Master Out Slave In) and the serial-in pin of the master shift register is connected to the serial-out pin of the slave shift register by MISO (Master In Slave Out). The master clock generator provides clock to shift register in both master and slave shift registers. The clock input of the shift registers can be falling- or rising-edge triggered. This will be discussed shortly.

In SPI, the shift registers are 8 bits long. It means that after 8 clock pulses, the contents of the two shift registers are interchanged. When the master wants to send a byte of data, it places the byte in its shift register and generates 8 clock pulses. After 8 clock pulses, the byte is transmitted to the slave shift register. When the master wants to receive a byte of data, the slave side should place the byte in its shift register and after 8 clock pulses the data will be received by the master shift register. It must be noted that SPI is full duplex meaning that it sends and receives data at the same time.

## Clock polarity and phase in SPI device

As we mentioned before in UART communication, transmitter and receiver must agree on a clock frequency (baud rate). In SPI communication, both master and slave uses the same clock but the master and slave(s) must agree on the clock polarity and phase with respect to the data. Freescale names these two options as CPOL (clock polarity) and CPHA (clock phase) respectively, and most companies have adopted that convention. At CPOL= 0 the idle value of the clock is zero while at CPOL=1 the idle value of the clock is one. CPHA=0 means sample data on the leading (first) clock edge, while CPHA=1 means sample data on the trailing (second) clock edge. Notice that if the idle value of the clock is zero the leading (first) clock edge is a rising edge but if the idle value of the clock is one, the leading (first) clock edge is a falling edge. See Table 8-1 and Figure 8-3.

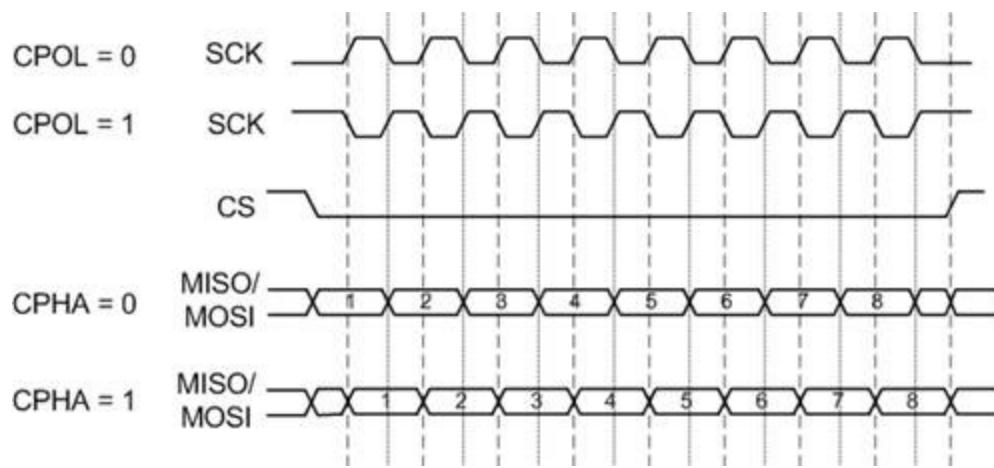


Figure 8-3: SPI Clock Polarity and phase

CPOL	CPHA	Data Read and change time	SPI Mode
0	0	read on rising edge, changed on a falling edge	0
0	1	read on falling edge, changed on a rising edge	1
1	0	read on falling edge, changed on a rising edge	2
1	1	read on rising edge, changed on a falling edge	3

Table 8-1: SPI Clock Polarity and phase

## Review Questions

- True or false. SPI is an Asynchronous protocol.
- True or false. In the SPI protocol, the clock is always generated by the master device.

## Section 8.2: SPI programming in TI ARM Tiva

The TI ARM Tiva chip comes with four on-chip SPI modules. The TI literature uses the SSI (Synchronous Serial Interface) name since it supports all three protocols of SPI, MICROWIRE, and TI Synchronous Serial Interface. In this section, we examine the SPI features of SSI module since it is the most widely used protocol. The SSI modules are located at the following base addresses:

SSI Module	Base Address
SSI0	0x40008000
SSI1	0x40009000
SSI2	0x4000A000
SSI3	0x4000B000

Table 8-2: SSI Module Base Address

### Enabling Clock to SSI

To enable and use any of the peripheral modules in the Tiva chip, we must enable the clock to it. We use RCGCSSI register to enable the clock to SSI modules. Notice again RCGCSSI is part of the SYSTCL registers. Writing a one to each of the R3 to R0, enables the corresponding SSI module. We need RCGCSSI = 0x0F to enables the clock to all four SSI modules.

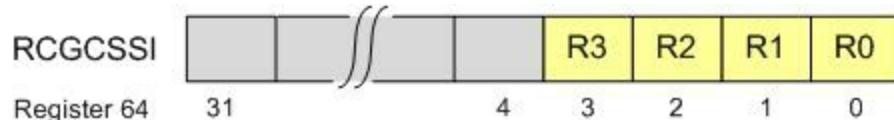


Figure 8-4: RCGSSI, Offset 0x61C

### Configuring the SSI Module

The SSICR0 (SSI Control register 0) sets SSI configuration. Figure 8-5 shows the bits of SSICR0 and table 8-3 describe the function of each bit. Although the conventional SPI uses only 8-bit data, the SSI modules allow transfer of data between 4 bits to 16 bits.

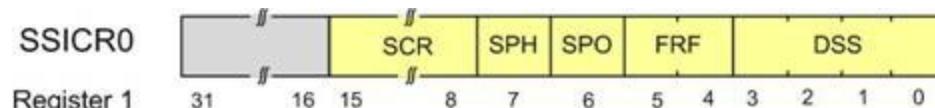


Figure 8-5: SSICR0, Offset 0

Bits	Name	Function	Description
0-3	DSS	SSI Data Size Select	0x03 to 0x0F for 4-bit to 16-bit data size 0x07 means 8-bit data size
4-5	FRF	SSI Frame Format Select	0 for SPI, 1 for TI, and 2 for MICROWIRE frame format
6	SPO	SSI Serial Clock Polarity	Clock polarity

7	SPH	SSI Serial Clock Phase	Clock phase
8-15	SCR	SSI Serial Clock Rate	BR=SysClk/(CPSDVSR * (1 + SCR))

Table 8-3: SSICR0

## Setting Bit Rate

SSI module clock source can be either of System Clock or PIOSC (Precision Internal Oscillator). The selected frequency is fed to prescaler before it is used by the Bit Rate circuitry, as shown in Figure 8-6

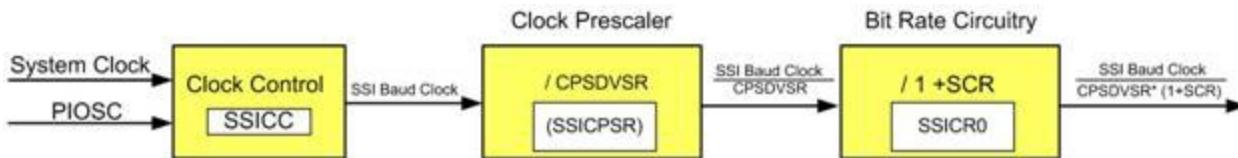


Figure 8-6: Bit Rate selection

The CPSDVSR (CPS Divisor) value comes from the prescaler divisor register shown in Fig 8-7.

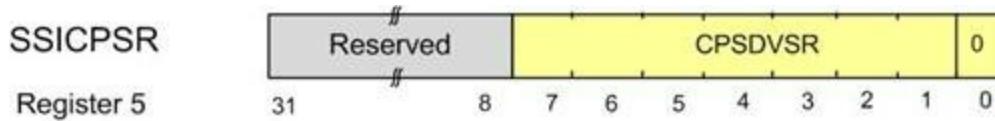


Figure 8-7: Register 5, SSI Clock Prescale (SSICPSPR), offset 0x010

The lower 8 bits of SSICPSPR (SSI Clock Prescale) register are used to divide the CPU clock before it is fed to the Bit Rate circuitry. Only even values can be used for the prescaler since the D0 must be 0. For the prescaler register, the lowest value is 2 and the highest is 254.

The SSICR0 (SSI Control register 0) allows the Bit Rate selection among other things. See Figures 8-5 and 8-6 and table 8-3 again. The output of clock prescaler circuitry is divided by 1 + SCR and then used as the SSI baud rate clock. The value of SCR can be from 0 to 255.

We can use the following formula to calculate the Bit Rate (BR):

$$BR = SysClk / (CPSDVSR \times (1 + SCR))$$

See Examples 8-1 and 8-2.

### Example 8-1

Assume the prescaler register has SSICPSPR=0x20 and system clock frequency is 16MHz. Find the values for the SCR part of the SSOCR0 register for the bit rate of (a) 100K, (b) 250K, and (c) 500K.

### Solution

Since the prescaler is 0x20 (32 in decimal), we have  $16\text{MHz} / 32 = 500\text{KHz}$ . Now:

- (a)  $16\text{MHz} / (32 \times (1 + 4)) = 100\text{KHz}$ .
- (b)  $16\text{MHz} / (32 \times (1 + 1)) = 250\text{KHz}$ .
- (c)  $16\text{MHz} / (32 \times (1 + 0)) = 500\text{KHz}$ .

## Example 8-2

In a given program, we have Bit Rate=50KHz and SCR=03 in SSICR0 register. Find the prescaler register value if system clock frequency is 16MHz.

### Solution

$$BR = \text{SysClk} / (\text{CPSDVSR} \times (1 + SCR))$$

50KHz = 16MHz / (X × (1 + 3)). Now, the prescaler value is 80 in decimal or 0x50 in Hex. Therefore the SSICPSR = 0x50.

## Other SPI configurations

The D3-D0 bits of SSICR0 are used to select the data size. See Figure 8-9. In most cases, we use 8-bit data size. The bits 5-4 are used to select various protocols. Upon Reset, the default protocol is Freescale SPI.

## Example 8-3

In example 8-1, find the values for the control register SSICR0 register if we want the Freescale SPI with 8-bit data size and SPH=1, SPO=1.

### Solution

- (a) SSICR0=00000100 00110111 in binary or = 0x437. Notice, we have set SPH and SPO bits to 1. (b)0x137, and (c)0x037.

## Master or Slave?

The SSI Module inside the Tiva chip can be Master or Slave. We use MS bit in SSI control register 1 (SSICR1) to designate the Tiva chip as master or slave. Another important bit in the SSICR1 register is enable/ disable bit (SSE). We must disable SSIX module during the initialization process and enable it afterward. See Figure 8-8.

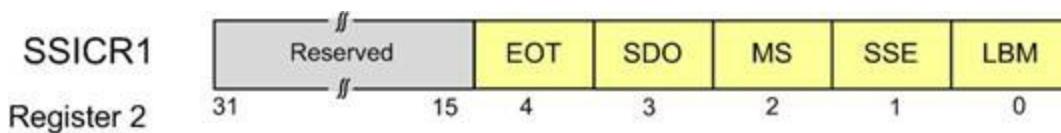


Figure 8-8: Register 2, SSI Control 1 (SSICR1), offset 0x004

## Data Register

The data is placed in SSIDR (SSI Data register) for transmission. The SSIDR is also used for the received data buffer. In 8-bit data size selection, we must place the data into the lower 8-bits of the register and the rest of the register are unused. In the receive mode, data is right-

justified meaning the lower 8-bit holds the received data.

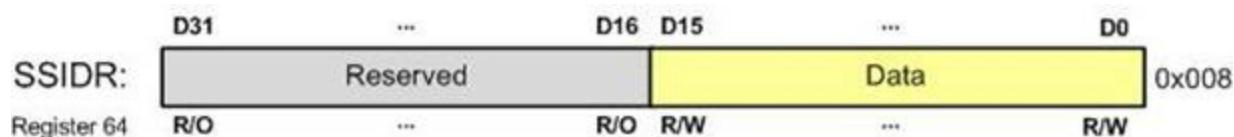


Figure 8-9: SSIDR register

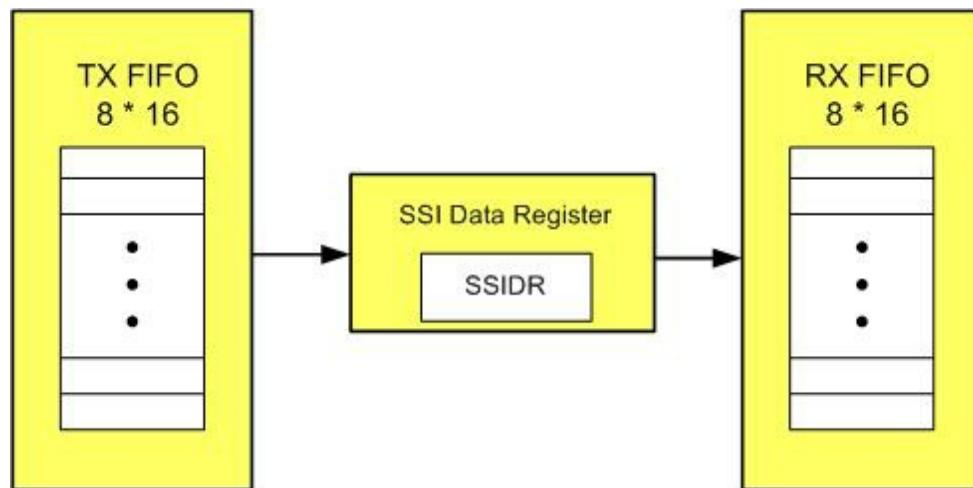


Figure 8-10: Register 3, SSI Data (SSIDR), offset 0x008

## Status Flag Register

We use the SSI Status flag register (SSISR) to monitor to see whether a byte has been received or if the transmission buffer is empty and ready for the next byte to be transmitted. See Figure 8-11 and Table 8-4.

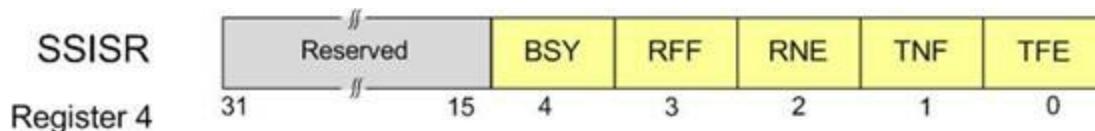


Figure 8-11: Register 4, SSI Status (SSISR), offset 0x00C

Bits	Name	Function	Description
0	TFE	SSI Transmit FIFO Empty	The bit is 1 when the transmit FIFO is empty
1	TNF	SSI Transmit FIFO Not Full	The bit is 1 when the SSI transmit FIFO not full
2	RNE	SSI Receive FIFO Not Empty	The bit is 1 when the receive FIFO is not empty
3	RFF	SSI Receive FIFO Full	The bit is 1 when the receive FIFO is full
4	BSY	SSI Busy Bit	The bit is 1 when the SSI is currently transmitting or receiving

Table 8-4: SSI Status (SSISR)

## Configuring GPIO for SSI

In using SSI, we must also configure the GPIO pins to allow the connection of the CPU pins to SPI device pins. See Table 8-5. In this regard, it is the same as all other peripherals. The steps are as follow:

1. Enable the clock to GPIO pin by using RCGCGPIO.
2. Set the GPIO AFSEL (GPIO alternate function) for SSI pins.

3. Enable digital pins in the GPIODEN (GPIO Digital enable) register.
4. Assign the SPI signals to specific pins using GPIOCTL register.

SSI Module Pin	GPIO Pin	SSI Module Pin	GPIO Pin
SS10Clk	PA2	SS12Clk	PB4
SS10Fss	PA3	SS12Fss	PB5
SS10Rx	PA4	SS12Rx	PB6
SS10Tx	PA5	SS12TX	PB7
SS11Clk	PD0 or PF2	SS13Clk	PD0
SS11Fss	PD1 or PF3	SS13Fss	PD1
SS11Rx	PD2 or PF0	SS13Rx	PD2
SS11Tx	PD3 or PF1	SS13TX	PD3

Table 8-5: GPIO Pin Assignment for all 4 SSI Modules

## Configuring SSI for SPI data transmission

After the GPIO configuration, we need to take the following steps to configure the SSI for the SPI protocol:

1. Enable the clock to SSI module using RCGCSSI.
2. Disable the SSI via bit 1 of SSICR1 register before initialization.
3. Set the Bit Rate with the SSICPSR prescaler and SSICR0 control registers.
4. Also select the SPI mode, phase, polarity, and data size in SSICR0 control register.
5. Select the master mod in SSISCR1 register.
6. Enable SSI using SSICR1 register.
7. Assert slave select signal.
8. Wait until the TNF flag (transmit FIFO not full) in SSISR (status register) goes high, then load a byte of data into SSIDR (data register) to be transmitted.
9. Repeat step 8 above until all the data are in the FIFO.
10. Wait until transmit is complete (transmit FIFO empty and SSI not busy).
11. Deassert the slave select signal.

### Program 8-1: sending A to Z characters via SPI1

```
/* p8_1.c: Using SSI1 to send A to Z characters via SPI1 */

#include "TM4C123GH6PM.h"

void init_SSI1(void);
void SSI1Write(unsigned char data);

int main(void)
{
    unsigned char i;

    init_SSI1();

    for(;;)
    {
        for (i = 'A'; i <= 'Z'; i++)
        {

```

```

        SSI1Write(i); /* write a character */
    }
}

void SSI1Write(unsigned char data)
{
    GPIOF->DATA &= ~0x04;           /* assert SS low */
    while((SSI1->SR & 2) == 0); /* wait until FIFO not full */
    SSI1->DR = data;             /* transmit high byte */
    while(SSI1->SR & 0x10);    /* wait until transmit complete */
    GPIOF->DATA |= 0x04;          /* keep SS idle high */
}

void init_SSI1(void)
{
    SYSCTL->RCGCGSSI |= 2;      /* enable clock to SSI1 */
    SYSCTL->RCGCGPIO |= 8;       /* enable clock to GPIOD for SSI1 */
    SYSCTL->RCGCGPIO |= 0x20;    /* enable clock to GPIOF for slave select */

    /* configure PORTD 3, 1 for SSI1 clock and Tx */
    GPIOD->AMSEL &= ~0x09;        /* disable analog for these pins */
    GPIOD->DEN |= 0x09;          /* and make them digital */
    GPIOD->AFSEL |= 0x09;         /* enable alternate function */
    GPIOD->PCTL &= ~0x0000F00F; /* assign pins to SSI1 */
    GPIOD->PCTL |= 0x00002002;  /* assign pins to SSI1 */

    /* configure PORTF 2 for slave select */
    GPIOF->DEN |= 0x04;          /* make the pin digital */
    GPIOF->DIR |= 0x04;          /* make the pin output */
    GPIOF->DATA |= 0x04;          /* keep SS idle high */

    /* SPI Master, POL = 0, PHA = 0, clock = 4 MHz, 16 bit data */
    SSI1->CR1 = 0;               /* disable SSI and make it master */
    SSI1->CC = 0;                /* use system clock */
    SSI1->CPSR = 2;              /* prescaler divided by 2 */
    SSI1->CR0 = 0x0007;          /* 8 MHz SSI clock, SPI mode, 8 bit data */
    SSI1->CR1 |= 2;              /* enable SSI1 */
}

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## Interrupt using NVIC

We can use an interrupt handler to do the job of transmit and receive. By enabling the interrupt bits in SSIIM (SSI Interrupt Mask) register, we direct the interrupt to the NVIC interrupt controller and write an interrupt handler. Upon Reset, the interrupts are masked and raising of the flag will not direct it to NVIC. See Figure 8-12 and Table 8-6.

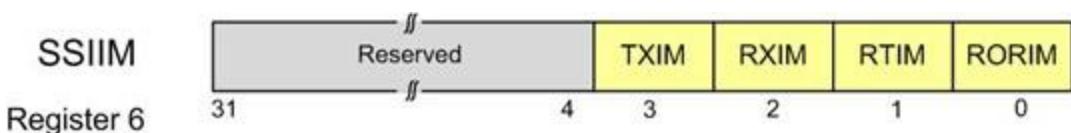


Figure 8-12: Register 6, SSIIM

Bits	Name	Function	Description
0	RORIM	SSI Receive Overrun Interrupt Mask	The receive FIFO overrun interrupt is masked when RORIM is zero and not masked when it is one
1	RTIM	SSI Receive Time-Out Interrupt Mask	The receive FIFO time-out interrupt is masked when RTIM is zero and not masked when it is one
2	RXIM	SSI Receive FIFO Interrupt Mask	The receive FIFO interrupt is masked when RXIM is zero and not masked when it is one
3	TXIM	SSI Transmit FIFO Interrupt Mask	The transmit FIFO interrupt is masked when TXIM is zero and not masked when it is one

Table 8-6: Register 6, SSIIM

## Review Questions

3. True or false. The SSI module in TI ARM Tiva does not support SPI protocol.
4. True or false. The Prescaler register of SSISPSR can have any odd or even number between 1 and 255.
5. In TI ARM Tiva the CS of the SPI module is called \_\_\_\_\_.
6. In TI ARM Tiva, which register is used to enable the clock to SSI module?
7. In TI ARM Tiva, which register is used to set the data size?

## Section 8.3: LTC1661 SPI DAC

In Chapter 7 we examined DAC concepts. In this section we show an SPI-based DAC and its interfacing to ARM. The LTC1661 is a 10-bit SPI serial DAC from Linear Technology. It has two separate output channels, named A and B, as shown in Figure 8-13.

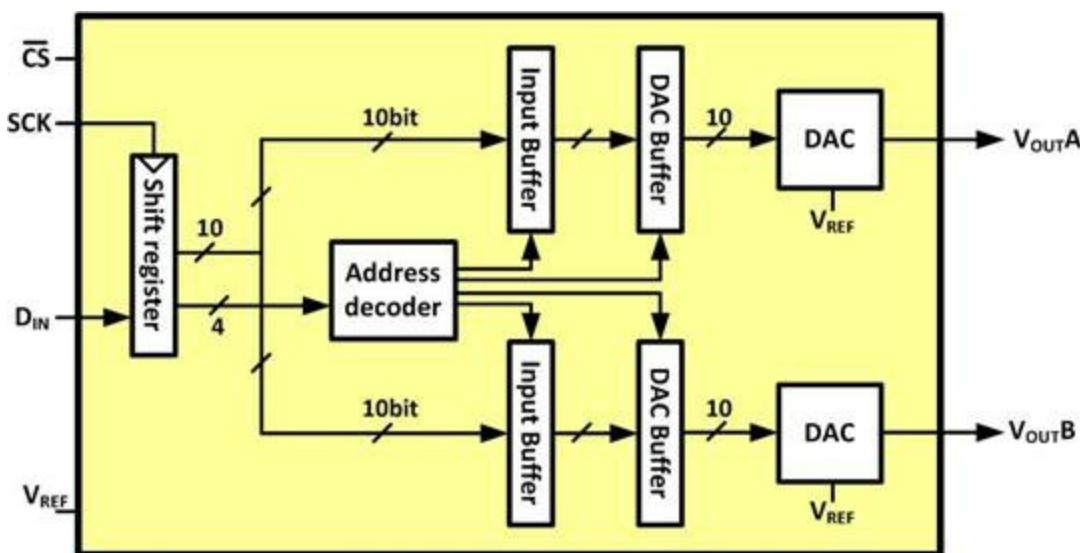


Figure 8-13: LTC1661 Internal Block Diagram

The relation between the input number and the output voltage is as follows:

$$V_{OUT} = (V_{REF} \times D_{IN}) / 1024$$

We can control the LTC1661 by sending 2 bytes of data. As shown in Figure 8-14, the 16-bit is made of 3 parts: control code (4 bits), data (10 bits), and not used (2 bits). The control code are used to control the internal parts of the LTC1661.

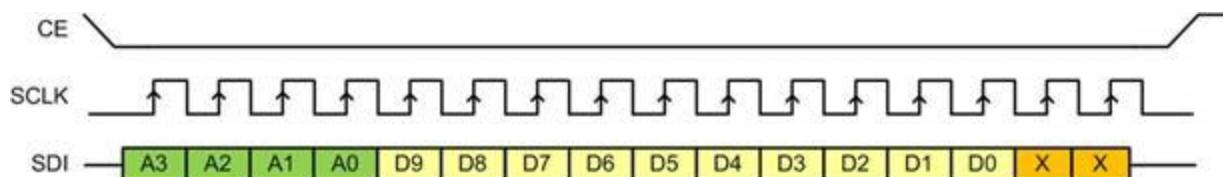


Figure 8-14: Sending a Packet of Data to LTC166x

As shown in Figure 8-13, each DAC is double buffered to provide simultaneous update. To do so, we load input buffers with proper values and then load the DAC buffers simultaneously. Table 8-7 shows the list of available control codes. To decrease power consumption, the DAC has a sleep mode, as well. We can switch between sleep and awake modes using proper control code.

A3 A2 A1 A0	Interrupt Register	DAC Register	Power Down Status	Comments
0 0 0 0	No Change	No Update	No Change	No operation. power-down status unchanged
0 0 0 1	Load DAC A	No Update	No Change	Load input register A with data. DAC outputs unchanged. power-down Status unchanged
0 0 1 0	Load DAC B	No Update	No Change	Load input register B with data. DAC outputs unchanged. power-down status unchanged

0011	-	-	-	Reserved
0100	-	-	-	Reserved
0101	-	-	-	Reserved
0110	-	-	-	Reserved
0111	-	-	-	Reserved
1000	No Change	Update Outputs	Wake	Load both DAC Regs with existing contents of input Regs. Outputs update. Part wakes up.
1001	Load DAC A	Update Outputs	Wake	Load input Reg A. Load DAC Regs with new contents of input Reg A and existing contents of Reg B. Outputs update.
1010	Load DAC B	Update Outputs	Wake	Load input Reg B. Load DAC Regs with existing contents of input Reg A and new contents of Reg B. Outputs update
1011	-	-	-	Reserved
1100	-	-	-	Reserved
1101	No Change	No Update	Wake	Part wakes up. Input and DAC Regs unchanged. DAC outputs reflect existing contents of DAC Regs
1110	No Change	No Update	Sleep	Part goes to sleep. Input and DAC Regs unchanged. DAC outputs set to high impedance state
1111	Load ADCs A, B with same 10-bit code	Update Outputs	Wake	Load both input Regs. Load both DAC Regs with new contents of input Regs. Outputs update. Part wakes up

Table 8-7: LTC1661 DAC Control Functions

See Examples 8-4 and 8-5.

#### Example 8-4

Assuming that  $V_{REF} = 5$  V, find the result of sending the following packets to LTC1661:

- a) 0001 0001 0000 0000 binary (0x1100)
- b) 1010 1000 1111 1100 binary (0xA8FC)

**Solution:**

- a) In 0001 0001 0000 0000, control code is 0001. As a result it loads data = 0001000000 (decimal 64) to the input buffer register for channel A. Note the output is not updated with this control code. The output will be updated after a control code of 1000 is sent. Therefore  $V_{OUTA} = V_{REF} * D_{IN} / 1024 = 5 * 64 / 1024 = 0.31V$ .
- b) In 1010 1000 1111 1100, the control code is 1010. As a result it loads data 1000111111 (decimal 575) to the input buffer register of channel B and also updates the output. Therefore,  $V_{OUTB} = V_{REF} * D_{IN} / 1024 = 5 * 575 / 1024 = 2.81V$ .

## Example 8-5

Assuming that  $V_{REF} = 5$  V, find the packets that should be sent to LTC1661 to:

- a) send out 0.5V from channel A
- b) send out 1.0V from channel B
- c) send out 0.5V and 1.0V from channels A and B, simultaneously

**Solution:**

- a)  $0.5V = V_{OUTA} = V_{REF} \times D_{IN} / 1024 = 5 \times D_{IN} / 1024$ .  $D_{IN} = 102$  (binary 0001100110). We send 1001 0001 1001 1000 binary (0x9198) since control code 9 loads input buffer register of channel A and updates the channel output.
- b)  $1.0V = V_{OUTB} = V_{REF} \times D_{IN} / 1024 = 5 \times D_{IN} / 1024$ .  $D_{IN} = 204 = 0011001100$  we send 1010 0011 0011 0000 binary (0xA330) since control code 9 loads input buffer register of channel B and updates the channel output.
- c) In order to change the outputs simultaneously, we need to load the input buffer register for channel A without updating the output first then load the input buffer register for channel B and update the outputs at the same time.  
First, we load channel A input buffer register using control code 0001 0001 1001 1000 binary (0x1198). Note it is the same code as (Part a) without the most significant bit set. Then, we load channel B input buffer register, and update the outputs using control code 1010 0011 0011 0000 binary (0xA330)

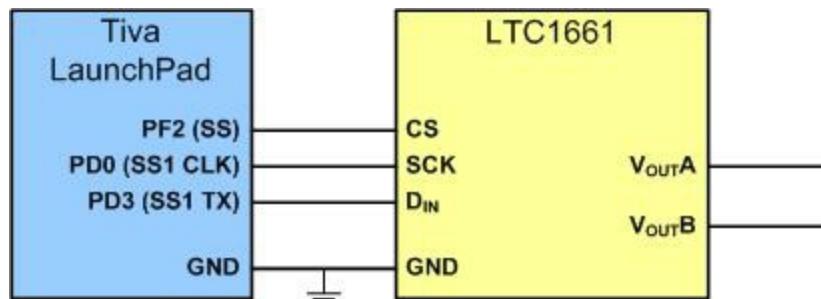


Figure 8-15: Connecting LTC1661 to the Microcontroller

### Program 8-2: Generating a saw tooth waveform with LTC1661 SPI DAC

```
/* p8_2.c: Using SSI1 to connect to LTC1661 DAC */  
/* PORTD 0 - SSI1 clock
```

```

PORTD 3 - SSI1 Tx
PORTF 2 - slave select
Generate sawtooth on output A */

#include "TM4C123GH6PM.h"

void init_SSI1(void);
void LTC1661Write(int chan, short d);

int main(void)
{
    int i;

    init_SSI1();

    for(;;)
    {
        for (i = 0; i < 1023; i++)
        {
            LTC1661Write(0, i); /* write a sawtooth to channel A */
        }
    }
}

void LTC1661Write(int chan, short data)
{
    GPIOF->DATA &= ~0x04;           /* assert SS low */
    data = (data & 0x03FF) << 2; /* bit 1-0 unused */
    if (chan == 0)                 /* add control code with channel number */
        data |= 0x9000;
    else
        data |= 0xA000;
    while((SSI1->SR & 2) == 0); /* wait until FIFO not full */
    SSI1->DR = data >> 8;       /* transmit high byte */
    while((SSI1->SR & 2) == 0); /* wait until FIFO not full */
    SSI1->DR = data & 0xFF;      /* transmit low byte */
    while(SSI1->SR & 0x10);     /* wait until transmit complete */
    GPIOF->DATA |= 0x04;          /* keep SS idle high */
}

void init_SSI1(void)
{
    SYSTIMER->RCGCGSSI |= 2;      /* enable clock to SSI1 */
    SYSTIMER->RCGCGPIO |= 8;       /* enable clock to GPIOD for SSI1 */
    SYSTIMER->RCGCGPIO |= 0x20;    /* enable clock to GPIOF for slave select */

    /* configure PORTD 3, 1 for SSI1 clock and Tx */
    GPIOD->AMSEL &= ~0x09;         /* disable analog for these pins */
    GPIOD->DEN |= 0x09;            /* and make them digital */
    GPIOD->AFSEL |= 0x09;          /* enable alternate function */
    GPIOD->PCTL &= ~0x0000F00F;   /* assign pins to SSI1 */
    GPIOD->PCTL |= 0x00002002;    /* assign pins to SSI1 */

    /* configure PORTF 2 for slave select */
    GPIOF->DEN |= 0x04;            /* make the pin digital */
    GPIOF->DIR |= 0x04;             /* make the pin output */
    GPIOF->DATA |= 0x04;            /* keep SS idle high */
}

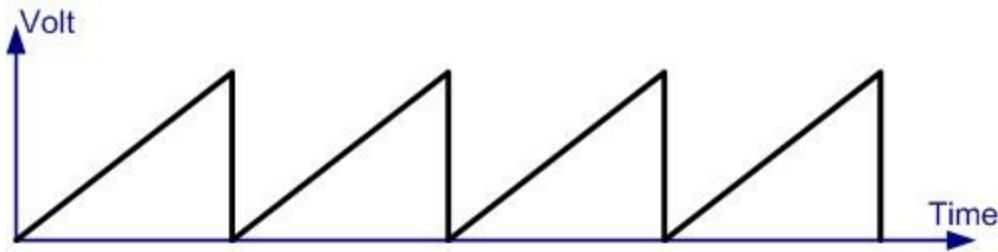
```

```

/* SPI Master, POL = 0, PHA = 0, clock = 4 MHz, 16 bit data */
SSI1->CR1 = 0;           /* disable SSI and make it master */
SSI1->CC = 0;            /* use system clock */
SSI1->CPSR = 2;          /* prescaler divided by 2 */
SSI1->CR0 = 0x0007;       /* 8 MHz SSI clock, SPI mode, 8 bit data */
SSI1->CR1 |= 2;          /* enable SSI1 */
}

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```



**Figure 8-16: The Generated sawTooth waveform**

### Program 8-3: Generating a sine waveform at LTC1661 DAC through SPI

```

/* p8_3.c: Using SSI1 to connect to LTC1661 DAC */
/* Generate sine wave on output B */
/* floating point processor is enabled to generate waveform data array */

#include "TM4C123GH6PM.h"
#include <math.h>

#define WAVEFORM_LENGTH 1024
int sinewave[WAVEFORM_LENGTH];

void init_SSI1(void);
void LTC1661Write(int chan, short d);

int main(void)
{
    int i;
    float fRadians;
#ifndef M_PI
    float M_PI = 4 * atan(1.0);
#endif

    init_SSI1();

    /* construct data table for a sine wave */
    fRadians = ((2 * M_PI) / WAVEFORM_LENGTH);
    for (i = 0; i < WAVEFORM_LENGTH; i++)
    {
        sinewave[i] = (short)(511 * (sinf(fRadians * i) + 1));
    }

    for(;;)

```

```

{
    for (i = 0; i < WAVEFORM_LENGTH; i++)
    {
        /* write a sine wave to channel B */
        LTC1661Write(1, sinewave[i]);
    }
}

void LTC1661Write(int chan, short data)
{
    GPIOF->DATA &= ~0x04;           /* assert SS low */
    data = (data & 0x03FF) << 2; /* bit 1-0 unused */
    if (chan == 0)                 /* add control code with channel number */
        data |= 0x9000;
    else
        data |= 0xA000;
    while((SSI1->SR & 2) == 0); /* wait until FIFO not full */
    SSI1->DR = data >> 8;       /* transmit high byte */
    while((SSI1->SR & 2) == 0); /* wait until FIFO not full */
    SSI1->DR = data & 0xFF;      /* transmit low byte */
    while(SSI1->SR & 0x10);    /* wait until transmit complete */
    GPIOF->DATA |= 0x04;         /* keep SS idle high */
}

void init_SSI1(void)
{
    SYSCTL->RCGCGSSI |= 2;       /* enable clock to SSI1 */
    SYSCTL->RCGCGPIO |= 8;        /* enable clock to GPIOD for SSI1 */
    SYSCTL->RCGCGPIO |= 0x20;     /* enable clock to GPIOF for slave select */

    /* configure PORTD 3, 1 for SSI1 clock and Tx */
    GPIOD->AMSEL &= ~0x09;        /* disable analog for these pins */
    GPIOD->DEN |= 0x09;          /* and make them digital */
    GPIOD->AFSEL |= 0x09;         /* enable alternate function */
    GPIOD->PCTL &= ~0x0000F00F;   /* assign pins to SSI1 */
    GPIOD->PCTL |= 0x00002002;   /* assign pins to SSI1 */

    /* configure PORTF 2 for slave select */
    GPIOF->DEN |= 0x04;          /* make the pin digital */
    GPIOF->DIR |= 0x04;          /* make the pin output */
    GPIOF->DATA |= 0x04;          /* keep SS idle high */

    /* SPI Master, POL = 0, PHA = 0, clock = 4 MHz, 16 bit data */
    SSI1->CR1 = 0;               /* disable SSI and make it master */
    SSI1->CC = 0;                /* use system clock */
    SSI1->CPSR = 2;              /* prescaler divided by 2 */
    SSI1->CR0 = 0x0007;          /* 8 MHz SSI clock, SPI mode, 8 bit data */
    SSI1->CR1 |= 2;              /* enable SSI1 */
}

void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## Review Questions

1. True or false. LTC1661 is an ADC.
2. True or false. LTC1661 is a 10-bit DAC.
3. True or false. There are 4 output channels in the LTC1661.

## Answers to Review Questions

### Section 8.1

1. False
2. True

### Section 8.2

3. False
4. False
5. SSF
6. RCGCSSI
7. SSISCRO

### Section 8.3

1. False
2. True
3. False



# Chapter 9: I2C Protocol and RTC Interfacing

## Section 9.1: I2C Bus Protocol

The IIC (Inter-Integrated Circuit) is a bus interface connection incorporated into many devices such as sensors, RTC, and EEPROM. The IIC is also referred to as I2C or I square C in many technical literatures. In this section we examine the signals of the I2C bus and focus on I2C terminology and protocols.

### I2C Bus

The I2C bus was originally started by Philips, but in recent years has become a widely used standard adopted by many semiconductor companies. I2C is ideal to attach low-speed peripherals to a motherboard or embedded system or anywhere that a reliable communication over a short distance is required. As we will see in this chapter, I2C provides a connection oriented communication with acknowledgement. I2C devices use only 2 pins for data transfer, instead of the 8 or more pins used in traditional parallel buses. These two signals are called SCL (Serial Clock) which synchronize the data transfer between two chips, and SDA (Serial Data). This reduction of communication pins reduces the package size and power consumption drastically, making them ideal for many applications in which space is a major concern. These two pins, SDA, and SCK, make the I2C a 2-wire interface. In many application notes, I2C is referred to as Two-Wire Serial Interface (TWI). In this chapter we use I2C and TWI interchangeably

### I2C line electrical characteristics

I2C devices use only 2 bidirectional open-drain pin for data communication. To implement I2C, only a 4.7k ohm pull-up resistor for each of bus lines is needed (see Figure 9-1). This implements a wired-AND which is needed to implement I2C protocols. It means that if one or more devices pull the line to low (zero) level, the line state is zero. The level of line will be 1 only if none of devices pull the line to low level.

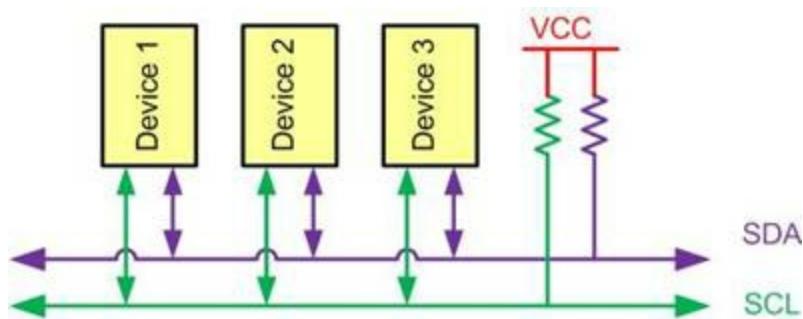


Figure 9-1: I2C Bus Characteristics

### I2C Nodes

In I2C protocol, more than 100 devices can share an I2C bus. Each of these devices is called a *node*. In I2C terminology, each node can operate as either master or slave. Master is a device that generates the Clock for the system, it also initiates and terminates a transmission. Slave is a node that receives the clock and is addressed by the master. In I2C, both master and slave can receive or transmit data. So there are 4 modes of operation. They are: master transmitter, master receiver, slave transmitter and slave receiver. Notice that each node can have more than one mode of operation at different times but it has only one mode of

operation at the same time. See Example 9-1

### Example 9-1

Give an example to show how a device (node) can take more than one mode of operation.

**Solution:**

If you connect a microcontroller to an EEPROM with I2C, the microcontroller does master transmit operation to write to EEPROM and master receive operation to read from EEPROM

In next sections, you will see that a node can do the operations of master and slave at different time.

### Bit Format

I2C is a synchronous serial protocol; each data bit transferred on the SDA line is synchronized by a high to low pulse of clock on SCL line. According to I2C protocols the data line cannot change when the clock line is high, it can change only when the clock line is low. See Figure 9-2. STOP and START condition are the only exceptions to this rule.

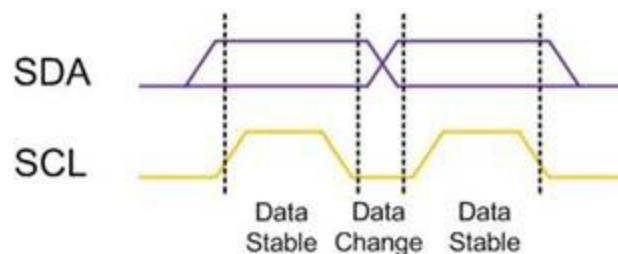


Figure 9-2: I2C Bit Format

### START and STOP conditions

As we mentioned before, I2C is a connection oriented communication protocol, it means that each transmission is initiated by a START condition and is terminated by STOP condition. Remember that the START and STOP conditions are generated by the master.

STOP and START conditions must be distinguished from bits of address or data and that is why they do not obey the bit format rule that we mentioned before.

START and STOP conditions are generated by keeping the level of the SCL line to high and then changing the level of the SDA line. START condition is generated by a high-to-low change in SDA line when SCL is high. STOP condition is generated by a low-to-high change in SDA line when SCL is low. See Figure 9-3.

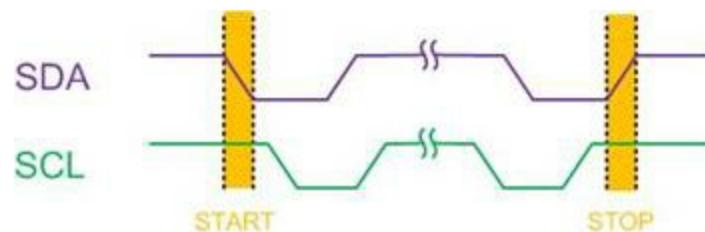


Figure 9-3: START and STOP Conditions

The bus is considered busy between each pair of START and STOP conditions and no other master tries to take control of the bus when it is busy. If a master, which has the control of the bus, wishes to initiate a new transfer and does not want to release the bus before starting the new transfer, it issues a new START condition between a pair of START and STOP condition. It is called REPEATED START condition or simply RESTART condition. See Figure 9-4.

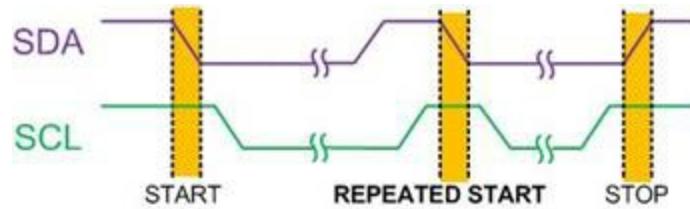


Figure 9-4: REPEATED START Condition

Example 9-2 shows why REPEATED START condition is necessary.

### Example 9-2

Give an example to show when a master must use REPEATED START condition. What will happen if the master does not use it?

#### Solution:

If you connect 2 microcontrollers (uA and uB) and an EEPROM with I<sup>2</sup>C, and the uA wants to display the addition of contents of address 0x34 and 0x35 of EEPROM, it has to use REPEATED START condition. Let's see what may happen if the uA does not use REPEATED START condition. uA transmit a START condition, reads the content of address 0x34 of EEPROM and transmit a STOP condition to release the bus. Before uA reads the contents of address 0x35, the uB seize the bus and change the contents of address 0x34 and 0x35 of EEPROM. Then uA reads the content of address 0x35, adds it to last content of address 0x34 and display the result to LCD. The result on the LCD is neither the sum of old values of address 0x34 and 0x35 nor the sum of the new values of address 0x34 and 0x35 of EEPROM!

#### *Message format in I<sup>2</sup>C*

In I<sup>2</sup>C, each address or data to be transmitted must be framed in 9 bit long. The first 8 bits are put on SDA line by the transmitter and the 9th bit is the acknowledge by the receiver or it may be NACK (negative acknowledge). Notice that the clock is always generated by the master, regardless of it being transmitter or receiver. To allow acknowledge, the transmitter release the SDA line during the 9th clock so the receiver can pull the SDA line low to indicate an ACK. If the receiver doesn't pull the SDA line low, it is considered as NACK. See Figure 9-5.

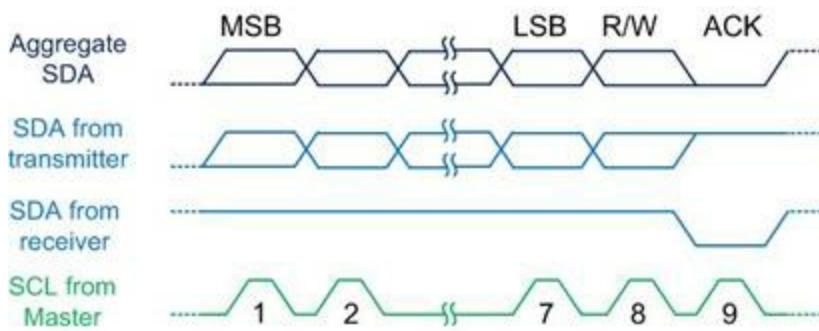


Figure 9-5: Byte Format in I2C

In I2C, each byte may contain either address or data. Also notice that: **START condition + slave address byte + one or more data byte + STOP condition** together form a complete data transfer. Next we will study slave address and data byte formats and how to combine them to make a complete transmission.

### *Address Byte Format*

Like any other bytes, all address bytes transmitted on the I2C bus are nine bits long. It consists of seven address bits, one READ/WRITE control bit and an acknowledge bit. (See Figure 9-6)

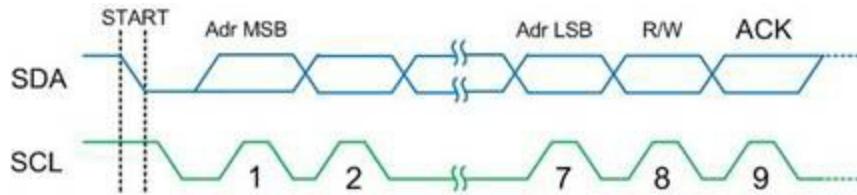


Figure 9-6: Address Byte Format in I2C

Slave address bits are used to address a specific slave device on the bus. 7 bit address let the master to address maximum of 128 slaves on the bus. Although address 0000 000 is reserved for general call and all address of the format 1111 xxx are reserved in many devices. That means  $119 = (128-1-8)$  device can share an I2C bus. In I2C bus the MSB of the address is transmitted first.

The 8th bit in the byte is READ/WRITE control bit. If this bit is set, the master will read the next byte from the slave, otherwise, the master will write the next byte on the bus to the slave. When a slave detects its address on the bus, it knows that it is being addressed and it should acknowledge in the ninth clock cycle by pulling SDA to low. If the addressed slave is not ready or for any reason does not want to respond to the master, it should leave the SDA line high in the 9th clock cycle. It is considered as NACK. In case of NACK, the master can transmit a STOP condition to terminate the transmission, or a REPEATED START condition to initiate a new transmission.

Example 9-3 shows how a master says that it wants to write to a slave.

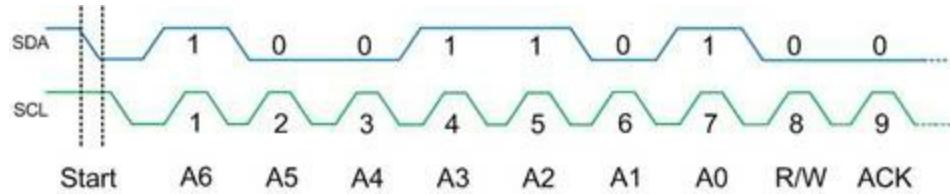
### Example 9-3

Show how a master initiate a write to a slave with address 1001101?

**Solution:**

The following actions are performed by the master:

- 1) The master put a high to low pulse on SDA while SCL is high to generate a start condition to start the transmission
- 2) The master transmit 1001101 0 into the bus. the first seven bits (1001101) indicates the slave address and the 8th bit (0) indicates Write operation and the master will write the next byte (data) into the slave.



---

An address byte consisting of a slave address and a READ is called SLA+R while an address byte consisting of a slave address and a WRITE is called SLA+W.

As we mentioned before, address 0000 000 is reserved for general call. It means that when a master transmit address 0000 000 all slaves respond by changing the SDA line to zero for one clock cycle for an ACK and wait to receive the data byte. It is useful when a master want to transmit the same data byte to all slaves in the system. Notice that the general call address cannot be used to read data from slaves because no more than one slave is able to write to the bus at a given time. Also not all the devices responds to a general call.

### Data Byte Format

Like other bytes, data bytes are 9 bits long too. The first 8 bits are a byte of data to be transmitted and the 9th bit, is ACK. If the receiver has received the last byte of data and does not wish to receive more data, it may signal a NACK by leaving the SDA line high. The master should terminate the transmission with a STOP after a NACK appears. In data bytes, like address bytes, MSB is transmitted first.

### Combining Address and Data Bytes into a Transmission

In I2C, normally, a transmission is started by a START condition, followed by an address byte (SLA+R/W), one or more data bytes and finished by a STOP condition. Figure 9-7 shows a typical data transmission. Try to understand each element in the figure. (See Example 9-4)

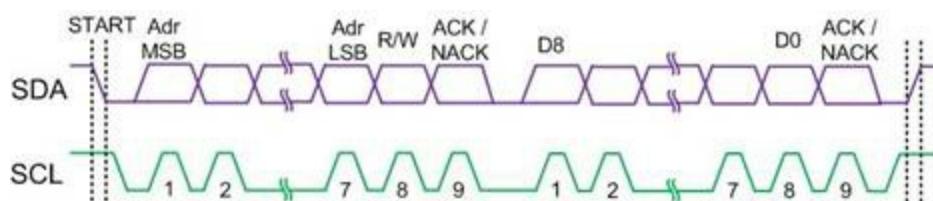


Figure 9-7: Typical Data Transmission

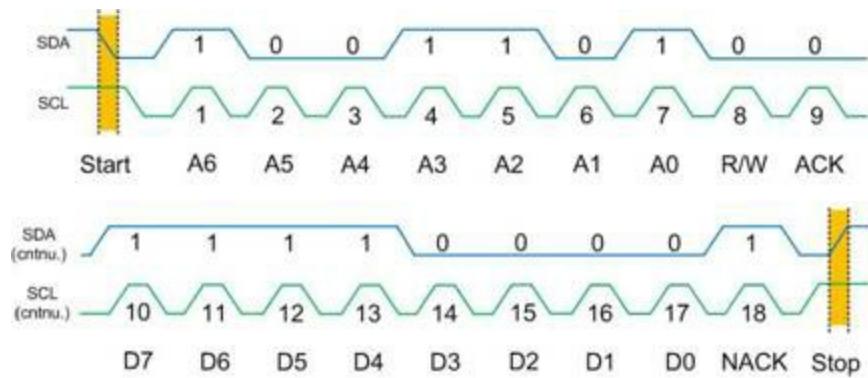
### Example 9-4

Show how a master writes data value 11110000 to a slave with an address 1001101?

## Solution:

The following actions are performed by the master:

- 1) The master put a high to low transition on SDA while SCL is high to generate a START condition to start the transmission
- 2) The master transmit 1001101 0 on the bus. the first seven bits (1001101) indicates the slave address and the 8th bit (0) indicates Write operation and say that the master will write the next byte (data) into the slave.
- 3) The slave pulls the SDA line low at the 9th clock pulse to signal an ACK to say that it is ready to receive data byte
- 4) After receiving the ACK, the master will transmit the data byte (11110000) on the SDA line. (MSB first)
- 5) When the slave device receives the data it leaves the SDA line high to signal NACK and inform the master that the slave received the last data byte and does not need any more data
- 6) After receiving the NACK, the master will know that no more data should be transmitted. The master changes the SDA line when the SCL line is high to transmit a STOP condition and then releases the bus.



## Clock stretching

One of the features of the I<sup>2</sup>C protocol is clock stretching. It is a kind of flow control. If an addressed slave device is not ready to process more data it will stretch the clock by holding the clock line (SCL) low after receiving (or sending) a bit of data so the master will not be able to raise the clock line (because devices are wire-ANDed) and will wait until the slave releases the SCL line to show it is ready for the next bit. See Figure 9-8

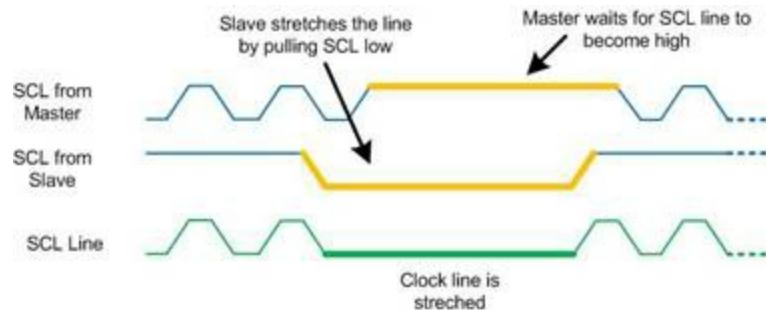


Figure 9-8: Clock Stretching

## Arbitration

I2C protocol supports multi-master bus system. It doesn't mean that more than one master can use the bus at the same time. Each master waits for the current transmission to finish and then start to use the bus. But it is possible that two or more masters initiate a transmission at about the same time. In this case the arbitration happens.

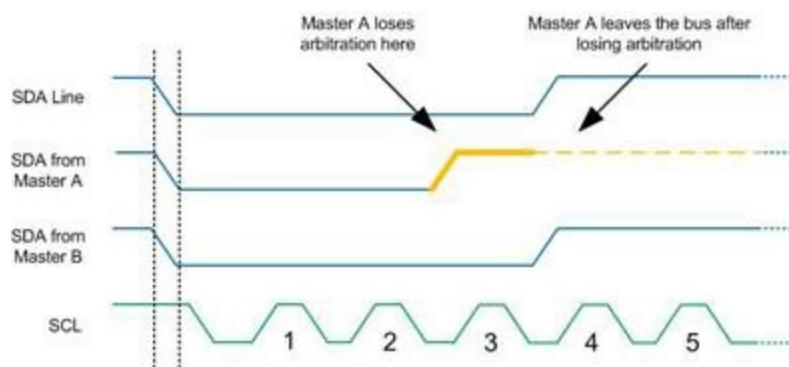
Each master has to check the level of the bus and compare it with the levels it is driving; if it doesn't match, that master has lost the arbitration, and will switch to slave mode. In the case of arbitration, the winning master will continue its job. Notice that neither the bus is corrupted nor the data is lost. See Example 9-5

### Example 9-5

If two master A and B start at about the same time. What happens if master A wants to write to slave 0010 000 and master B wants to write to slave 0001 111 ?

#### Solution:

Master A will lose the arbitration in the third clock because the SDA line is different from output of master A at the third clock. Master A switches to slave mode and stops driving the bus after losing the arbitration.



## Multi-byte burst write

Burst mode writing is an effective means of loading data into consecutive memory locations. It is supported in I2C like SPI and many other serial protocols. In burst mode, we provide the address of the first memory location, followed by the data for that location. From then on, consecutive bytes are written to consecutive memory locations. In this mode, the I2C device internally increments the address location as long as STOP condition is not detected. The following steps are used to send (write) multiple bytes of data in burst mode for I2C devices.

1. The master generates a START condition.
2. The master transmits the slave address followed by a zero bit (for write).
3. The master transmits the address of the first location.
4. The master transmits the data for the first location and from then on, the master simply provides consecutive bytes of data to be placed in consecutive memory locations in the slave.
5. The master generates a STOP condition.

Figure 9-9 shows how to write 0x05, 0x16 and 0x0B to 3 consecutive locations starting from location 00001111 of slave 1111000.

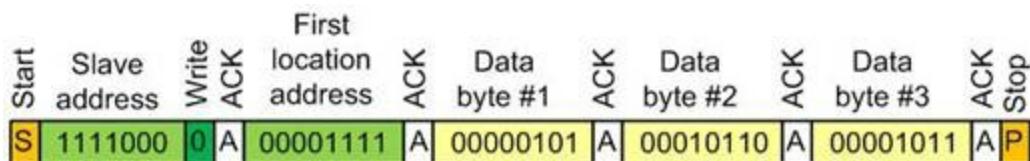


Figure 9-9: Multi-byte Burst Write

### Multi-byte burst read

Burst mode reading is an effective means of bringing out the contents of consecutive memory locations. In burst mode, we provide the address of the first memory location only. From then on, contents are brought out from consecutive memory locations. In this mode, the I2C device internally increments the address location as long as STOP condition is not detected. The following steps are used to get (read) multiple bytes of data using burst mode for I2C devices.

1. The master generates a START condition.
2. The master transmits the slave address followed by a zero bit (for writing the address).
3. The master transmits the address of the first memory location.
4. The master generates a RESTART condition to switch the bus direction from write to read.
5. The master transmits the slave address followed by a one bit (for read).
6. The master clocks the bus 8 times and the slave device provides the data for the first location.
7. The master provides an ACK.
8. The master reads the consecutive locations and provides an ACK for each byte.
9. The master gives a NACK for the last byte received to signal the slave that the read is complete.
10. The master generate a STOP condition.

Figure 9-10 shows how to read three consecutive locations starting from location 00001111 of slave number 1111000.

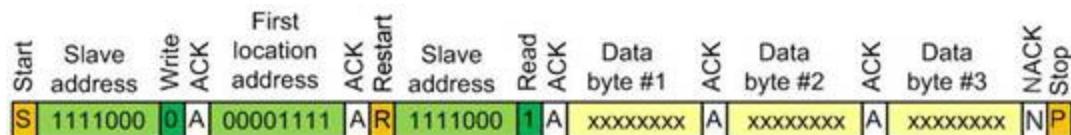


Figure 9-10: Multi-byte Burst Read

### Review Questions

1. True or false. I2C protocol is ideal for short distance.
2. How many bits are there in a frame? Which bit is for acknowledge?
3. True or false. START and STOP conditions are generated when the SDA is high.
4. What is the name of flow control method in the I2C protocol?
5. True or false. After arbitration of two masters, both of them must start transmission from beginning.

## Section 9.2: I2C Programming in TI ARM Tiva

The TI ARM Tiva chip comes with four on-chip I2C modules. In this section, we examine the registers of and features of I2C module. The I2C modules are located at the following base addresses:

SSI Module	Base Address
I2C 0	0x4002.0000
I2C 1	0x4002.1000
I2C 2	0x4002.2000
I2C 3	0x4002.3000

Table 9-1: I2C Module Base Address for TI Tiva

### Enabling Clock to I2C Module

To enable and use any of the peripherals, we must enable the clock to it. We use RCGCI2C register to enable the clock to I2C modules. Notice again RCGCI2C is part of the SYSCTL registers. We need  $\text{SYSCTL} \rightarrow \text{RCGCI2C} |= 0x0F$  to enables the clock to all four I2C modules.

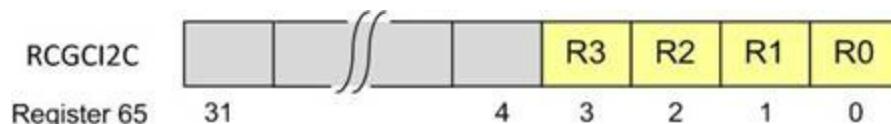


Figure 9-11: RCGCI2C, Offset 0x620

Bits	Name	Function	Description
0	R0	I2C 0 Run Mode Clock Gating Control	1 to enable and 0 to disable
1	R1	I2C 1 Run Mode Clock Gating Control	1 to enable and 0 to disable
2	R2	I2C 2 Run Mode Clock Gating Control	1 to enable and 0 to disable
3	R3	I2C 3 Run Mode Clock Gating Control	1 to enable and 0 to disable

Table 9-2: RCGCI2C Description

### I2C Clock speed

The I2CMTPR (I2C Master Timer Period) register allows us to set the clock rate for the SCL. The SCL clock comes from the system clock and goes through clock divider circuit controlled by I2CMTPR register. The I2C standard defines four clock speeds for the SCL. They are Standard mode of 100Kbps (bits per second), Fast mode of 400Kbps, and Fast Plus mode of 1Mbps. There is also a High-Speed (HS) mode which can be as high as 3.3Mbps. The lower 7 bits (D6-D0, TPR) of I2CMTPR register are used to set the I2C clock speed. See Figure 9-12.

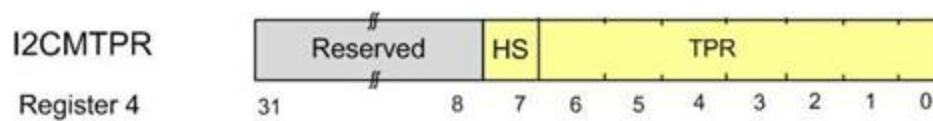


Figure 9-12: I2CMTPR, Offset 0x0C

We use the following formula to set the I2C clock speed:

$$\text{SCL\_PERIOD} = 2 \times (1 + \text{TPR}) \times (\text{SCL\_LP} + \text{SCL\_HP}) \times \text{CLK\_PRD}$$

In the above formula, SCL\_PERIOD is the I2C clock period, CLK\_PRD is the system clock period. The SCL\_LP is the SCL Low Period and is fixed at 6. The SCL\_HP is the SCL High Period

and is fixed at 4. Now, if rearrange the above formula, we have:

$$SCL\_PERIOD = 2 \times (1 + TPR) \times (4 + 6) \times CLK\_PRD$$

$$SCL\_PERIOD = 2 \times (1 + TPR) \times 10 \times CLK\_PRD$$

$$SCL\_PERIOD = (20 \times (1 + TPR)) / System\_Clock\_Freq$$

Now, solving for TPR(the value we need for the I2CMTPR register) we have:

$$TPR = ((System\_Clock\_Freq \times SCL\_PERIOD) / 20) - 1$$

$$TPR = (System\_Clock\_Freq) / (20 \times I2C\_Clock) - 1$$

See Examples 9-6 and 9-7.

### Example 9-6

Assume the system clock frequency is 16MHz. Find the values for the I2CMTPR register if we want I2C clock of (a) 100Kbps, (b) 400Kbps, and (c) 1Mbps.

**Solution:**

Using 16MHz for the CPU Frequency, we have:

$$TPR = (System\_Clock\_Freq) / (20 \times I2C\_Clock) - 1$$

$$TPR = (16MHz) / (20 \times I2C\_Clock) - 1$$

$$(a) TPR = (16MHz) / (20 \times 100K) - 1 = 8 - 1 = 7$$

$$(b) TPR = (16MHz) / (20 \times 400K) - 1 = 2 - 1 = 1$$

(c)  $TPR = (16MHz) / (20 \times 1M) - 1$ . This is not allowed due to the system clock frequency being too low.

### Example 9-7

In example 9-6,find the values for the I2CMTPR for (a) 100Kbs and (b) 400Kbps.

**Solution:**

(a) I2CMTPR = 00000111 in binary or = 0x07. Notice, the HS (D7) is Low. The HS is used only for the High Speed of 3.3MBps.

(b) I2CMTPR = 00000001 in binary or = 0x01.

## Master or Slave?

The I2C Module inside the ARM chip can be Master or Slave. We use I2CMCR (I2C Master Configuration register) to designate the ARM chip as master or slave. Setting bit D4 to 1 makes the I2C of ARM chip as Master. Therefore, we need I2CMCR = 00010000 = 0x10 for Master.

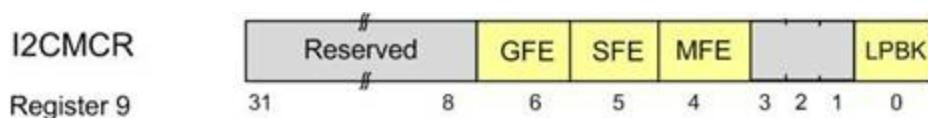


Figure 9-13: I2CMCR, Offset 0x020

Bits	Name	Function	Description
0	LPBK	I2C Loopback	0: Normal operation, 1: Loopback
4	MFE	I2C Master Function Enable	1: Enable Master Function, 0: Disable
5	SFE	I2C Slave Function Enable	1: Enable Slave Function, 0: Disable
6	GFE	I2C Glitch Filter Enable	1: Enable Glitch Filter, 0: Disable

Table 9-3: I2CMCR Description

## Slave Address

To transmit a byte of data to an I2C device, we must specify its I2C address. We use I2CMSC (I2C Master Slave Address) register to hold the address of the slave device. Notice, the addresses in I2C are only 7 bits (maximum of 127 devices). In the I2CMSC register, the D7-D1 bits are used for the slave address and the LSB of D0 is used to indicate if we are transmitting to slave device or receiving from a slave device.

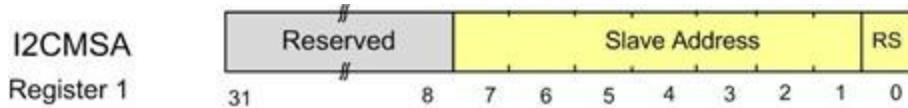


Figure 9-14: I2CMSC, Offset 0x000

## Data Register

In Master transmit mode, we place a byte of data in I2CMDR (I2C Master Data register) for transmission. Only the lower 8 bits of this register is used.

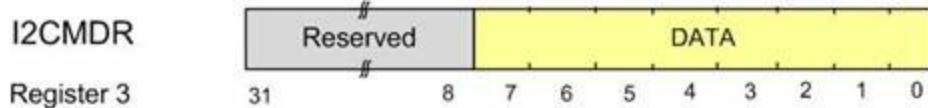


Figure 9-15: I2CMDR, Offset 0x008

## Control and Status Flag Register

We use the I2CMCS (I2C Master Control/Status) register for both control and status. When we write to I2CMCS register, it configures the I2C controller operation. Upon reading it, we get the status to see if a byte has been transmitted and transmission buffer is empty and ready for the next byte. The other way to look at this is that we have two registers with the same name. One we write to control the I2C module, the other one we read from to get the status of the I2C module.

After placing a byte of data in I2C Data register and the slave address in I2C Master Slave address register, we may write a value of 0x07 to I2CMCS register for the I2C to start a single byte data transmission from Master (microcontroller) to slave device. Notice, writing 0x07 to this register has all the three of STOP = 1, RUN = 1, and START = 1 in it. Because the START and STOP are set, the I2C module will generate a START condition, send the slave address with R/W bit, check the acknowledge bit, send the data byte, check the acknowledge bit and generate the STOP condition to terminate the transmission. For multiple byte burst write or read, the

protocols are different. We will discuss that later.

For a single byte write, after the START the bus will go busy until the data byte is written and the STOP is sent. To monitor the progress of the transmission, we need to check the BUSBSY bit (bit 6) of the I2CMCS register. When the bit goes low, the transmission is complete. Because ARM Cortex architecture incorporates buffered writes, the write of 0x07 to I2CMCS does not take effect immediately. Checking the BUSBSY bit immediately after setting the I2CMCS register to 0x07 may return a 0 to indicate the transmission has not started yet. To avoid this, the program should read the I2CMCS back to flush the write first before reading it again to check the BUSBSY bit.

When the transmission is complete, the program should check the ERROR bit (bit 1) to make sure there was no error in the transmission. Transmission error may be caused by either slave address was not acknowledged or the data write was not acknowledged. In either case, the ADRACK (bit 2) or the DATAACK (bit 3) will be set.

In a system with multiple I2C masters, the program should check the BUSBSY (bit 6) of I2CMCS register to be sure that the bus is idle before starting a transmission. After the transmission is started, the program should check the ARBLST (bit 4) to see whether it lost the arbitration or not. If the arbitration is lost, the program should attempt the transmission after the bus is not busy anymore.

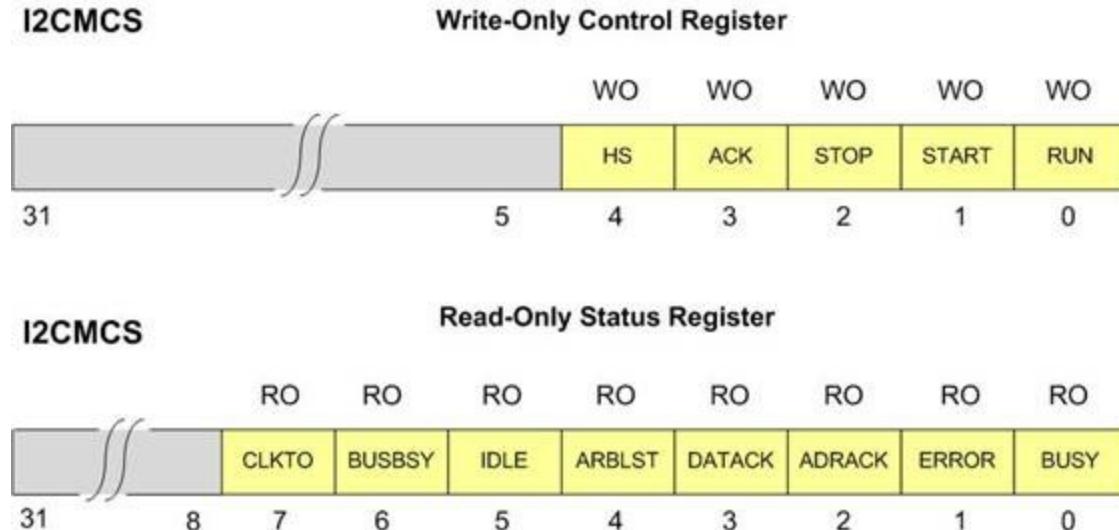


Figure 9-16: I2CMCS, Offset 0x004

Bits	Name	Function	Description
0	RUN	I2C Master Enable	1: Master is enable to transfer data
1	START	Generate START	It should be 1 to generate Start condition
2	STOP	Generate STOP	It should be 1 to generate Stop condition
3	ACK	Data Acknowledge Enable	It should be 1 to generate auto ACK
4	HS	High-Speed Enable	It should be 1 to run in high speed mode

Table 9-4: I2CMS Write-Only Register

Bits	Name	Function	Description
0	BUSY	I2C Busy	0: The controller is idle
1	ERROR	Error	0: No error was detected

2	ADRACK	Acknowledge Address	0: The transmitted address was acknowledged
3	DATACK	Acknowledge Data	0: The transmitted data was acknowledged
4	ARBLST	Arbitration Lost	0: The I2C controller won arbitration
5	IDLE	I2C Idle	0: The I2C controller is not idle
6	BUSBSY	Bus Busy	0: The I2C bus is idle
7	CLKTO	Clock Timeout Error	0: No clock timeout error

Table 9-5: I2CMS Read-Only Register

## Enabling Open Drain

Most of the digital output pins are configured as totem-pole output (because the transistors of the output driver are stacked up like as totem-pole). The other name for this configuration is push-pull because it is pushing the current out when high and pulling the current in when low. This configuration allows for faster transition when the output is switching from high to low or from low to high. The problem with a totem-pole output is when more than one output is connected together and one output is high the other is low, the high outputs push the current out and the low outputs pull in the current. A large amount of current could flow between the outputs and damages the circuit.

One common solution to allow multiple outputs connected together is to use the open-drain output. (open-drain for CMOS devices or open-collector for TTL devices) In this output configuration, the output pin is connected to the drain of the output transistor while the source of that transistor is grounded. When the transistor is on, the output pin is grounded and when the transistor is off, the output pin (the drain) is open. The open-drain outputs may be connected together. A pull-up resistor is added so that the signal is high when none of the outputs is active. When any one of the outputs is active, the signal is low. It forms a "wired-AND" logic and is exactly what is required for the I2C bus.

Figure 9-1 in the last section showed the physical connection of the I2C buses. For the I2C Module inside the TI ARM Tiva, we must enable the open-drain option for the I/O pins used by the I2C buses. We use the GPIOODR (GPIO Open Drain) register to enable the open drain. Depending on which pins we use for the I2C connection, we have to enable the GPIOODR register for that port.

## Configuring GPIO for I2C

In using I2C, we must configure the GPIO pins to allow the connection of the CPU pins to I2C device pins. In this regard, it is same as all other peripherals. The steps are as follow:

1. Enable the clock to GPIO pin by using RCGCGPIO.
2. Set the GPIO AFSEL (GPIO alternate function) for I2C pins.
3. Enable digital pins in the GPIODEN (GPIO Digital enable) register.
4. Assign the I2C signals to specific pins using GPIOCTL register.
5. Enable the open-drain option for the I/O pins used by the I2C signals. *With TI Tiva devices, the I2C pins are configured as open-drain by only setting the I2CSDA pin to open-drain. The I2CSCL pin will become open-drain with I2CSDA. Setting I2CSCL pin as open-drain*

*will not work.*

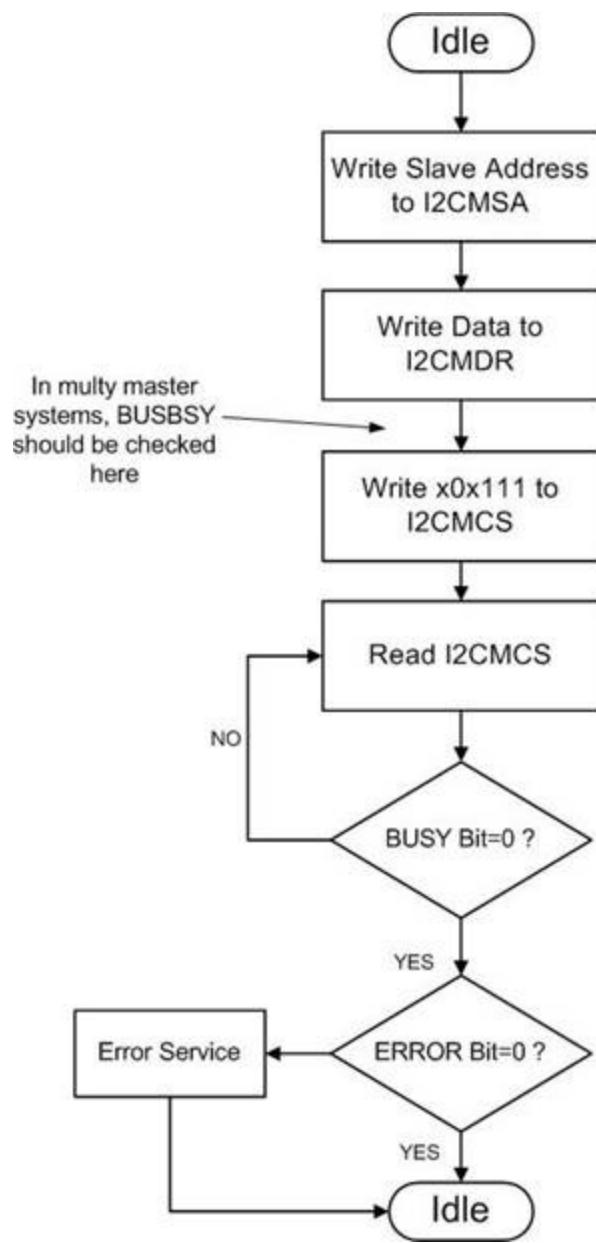
SSI Module Pin	GPIO Pin	SSI Module Pin	GPIO Pin
I2C0SCL	PB2	I2C2SCL	PB4
I2C0SDA	PB3	I2C2SDA	PB5
I2C1SCL	PA6	I2C3SCL	PDO
I2C1SDA	PA7	I2C3SDA	PD1

Table 9-6: I2C Pin Assignment

## Configuring I2C for data transmission

After the GPIO configuration, we need to take the following steps to configure the I2C and send a byte of data to an I2C slave device.

1. Enable the clock to I2C module using RCGCI2C.
2. Initialize the I2C as Master using I2CMCR register.
3. Set the I2C clock speed using I2CMTPR register.
4. Place the slave address with the R/W bit cleared for write in the I2CMSA register.
5. Place the byte of data to be transmitted into the I2CMDR register.
6. Initiate the data transmission by writing value 0x07 to the I2CMCS register. With I2CMCS = 0x07, will make the STOP = 1, RUN = 1, and START = 1.
7. Read back I2CMCS register to force the write out of the write buffer.
8. Keep reading the I2CMCS register and check the BUSY flag. Wait until it goes low.
9. Now, read the I2CMCS register again and check the ERROR flag to make sure there was no error.



**Figure 9-17: Master Single Transmit**

To send n bytes of data, you should send the first bit like single bit transmission but leave STOP bit of I2CMCS 0 because we do not want to generate Stop Condition after sending the first byte (Instead of writing value 0x07 to the I2CMCS register, write value 0x03). Now repeat from step 5 to send more data by writing value 0x01 to the I2CMCS register. Note that you should leave the START and STOP bits of I2CMCS 0 because no Stop or Start Condition should be generated. Before sending the last byte you should make STOP bit of I2CMCS 1 to generate STOP Condition after sending the last byte of data.

## Review Questions

- True or false. The I2C module in TI ARM Tiva does not support speed beyond 3.3Mbps.
- True or false. The I2CMCS(I2C Master Control/Status) bits have two different roles in WO(Write-Only) and RO(Read-Only) operations.
- True or false. There is no CS (chip select) pin in I2C.
- In TI ARM Tiva, which register is used to enable the clock to I2C module?
- In TI ARM Tiva, which register is used to set the SCL rate for I2C?

## Section 9.3: DS1307 RTC Interfacing and Programming

The real-time clock (RTC) is a widely used device that provides accurate time and date information for many applications. Many systems such as the PC come with such a chip on the motherboard. The RTC chip in the PC provides the time components of hour, minute, and second, in addition to year, month, and day. Many RTC chips use an external battery, which keeps the time and date even when the power of the PC is off. Although some microcontrollers come with the RTC already embedded into the chip, we have to interface the vast majority of them to an external RTC chip. The DS1307 is a serial RTC with an I<sup>2</sup>C bus. In this section, we interface and program the DS1307 RTC. According to the DS1307 data sheet from Maxim, "The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. The end of the month date is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The clock operates in either the 24-hour or 12-hour format with AM/PM indicator. The DS1307 has a built-in power-sense circuit that detects power failures and automatically switches to the battery supply." The DS1307 does not support the Daylight Savings Time option. Next, we describe the pins of the DS1307. See Figure 9-18.

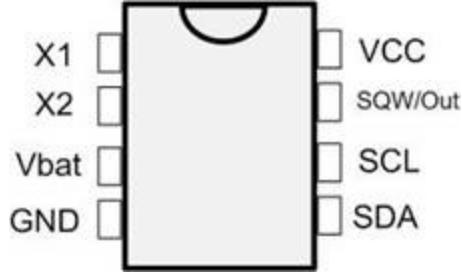


Figure 9-18: DS1307 Pins

### X1-X2

These are input pins that allow the DS1307 connection to an external crystal oscillator to provide the clock source to the chip. We must use a 32.768 kHz quartz crystal. The accuracy of the clock depends on the quality of this crystal oscillator.

### Vbat

Pin 3 can be connected to an external +3 V lithium battery, thereby providing the power source to the chip when the external supply voltage is not available. We must connect this pin to ground if it is not used. A 48mAh lithium battery can provide the power needed for more than 10 years to power the chip.

### GND

Pin 4 is the ground.

### SDA (Serial Data)

Pin 5 is the SDA pin and must be connected to the SDA line of the I<sup>2</sup>C bus.

### SCL (Serial Clock)

Pin 6 is the SCL pin and must be connected to the SCL line of the I<sup>2</sup>C bus.

### SQW/OUT

Pin 7 is an output pin providing 1 kHz, 4 kHz, 8 kHz, or 32 kHz frequency if enabled. This pin needs an external pull-up resistor to generate the frequency because it is open drain. If you do not want to use this pin you can omit the external pull-up resistor. We will see shortly how to control this pin.

## VCC

Pin 8 is used as the primary voltage supply to the chip. This primary voltage source is generally set to +5 V. When Vcc falls below the Vbat level, the DS1307 switches to Vbat and the external lithium battery provides power to the RTC. When the system is powered on, the DS1307 automatically switches to the power supply from battery power.

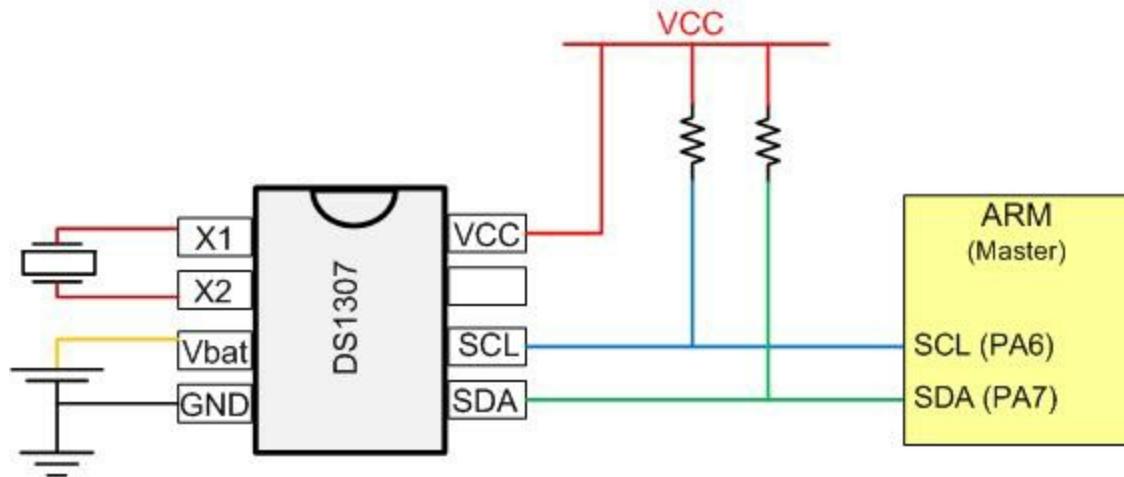


Figure 9-19: DS1307 Connections

## Address map of the DS1307

The DS1307 has a total of 64 bytes of RAM space with addresses 00–3FH. The first seven locations, 00–06, are set aside for RTC values of time and date. The next byte is used for the control register. It is located at address 07 in hex. That leaves 56 bytes, from addresses 07H to 3FH, available for general-purpose data storage. That means the entire 64 bytes of RAM are accessible directly for read or write. Table 9-7 shows the address map of the DS1307. Next, we study the control register, and time and date access in DS1307.

Address	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Function	Range
00H	CH		10 Seconds			Seconds			Seconds	
01H	0		10 Minutes			Minutes			Minutes	
02H	0	12 24	10hour PM/AM	10hour		Hours			Hours	1-12 0-23
03H	0	0	0	0	0	Day			Day	0-7
04H	0	0	10 Date			Date			Date	01-31
05H	0	0	0	10Mnt		Month			Month	1-12
06H		10 Year				Year			Year	00-99
07H	out	0	0	SQWE	0	0	RS1	RS0	Control	-
08H-3FH									RAM 56x8	0-FFH

Table 9-7: DS1307 Address Map

## The DS1307 control register

As shown in Table 9-7, the control register has an address of 07H. In the DS1307 control register, the bits control the function of the SQW/OUT pin. If the square wave output is disabled, setting the OUT bit to 1 will make the SQW/OUT pin low, and clearing the OUT bit to zero will make the SQW/OUT pin high. If SQWE (square wave enable) bit is set HIGH, the

oscillator output is enabled, otherwise, it is disabled. RS1-RS0 (rate select) bits select the output frequency of the oscillator output according to Table 9-8.

RS1	RS0	Output Frequency
0	0	1 Hz
0	1	4.096 kHz
1	0	8.192 kHz
1	1	32.768 kHz

Table 9-8: RS bits

### CH bit in address 00

One of the most important bits in the Seconds (address location 00) in the DS1307 is the CH (Clock Halt) bit. It is the seventh bit of address location 00. Setting the CH bit to one disables the oscillator, while setting CH to zero enables the oscillator. The CH bit is undefined upon reset. In order to enable the oscillator, we must clear the CH during initial configuration.

### Time and date address locations and modes

The byte addresses 0–6 are set aside for the time and date, as shown in Table 9-7. The DS1307 provides data in BCD format only. Notice the data range for the hour mode. We can select 12-hour or 24-hour mode with bit 6 of Hours register at location 02. When bit 6 is 1, the 12-hour mode is selected, and bit 6 = 0 provides us the 24-hour mode. In the 12-hour mode, bit 5 indicates whether it is AM or PM. If bit 5 = 0, it is AM and if bit 5 = 1 it is PM. See Example 9-8.

### Example 9-8

What value should be placed at location 02 to set the hour to: (a) 21, (b) 11AM, (c) 12 PM

#### Solution:

- (a) For 24-hour mode, we have D6 = 0. Therefore, we place 0010 0001 (or 0x21) at location 02, which is 21 in BCD.
- (b) For 12-hour mode, we have D6 = 1. Also, we have D5 = 0 for AM. Therefore, we place 0101 0001 at location 02, which is 51 in BCD.
- (c) For 12-hour mode, we have D6 = 1. Also, we have D5 = 1 for PM. Therefore, we place 0111 0010 at location 02, which is 72 in BCD.

### Register pointer

In DS1307, there is a register pointer that specifies the byte that will be accessed in the next read or write command. The first read or write operation sets the value of the pointer. After each read or write operation, the content of the register pointer is automatically incremented to point to the next location. It is useful in multi-byte read or write.

### Writing to DS1307

To set the value of the register pointer and write one or more bytes of data to DS1307, you can use the following steps:

1. To access the DS1307 for a write operation, after sending a START condition, you should

- transmit the address of DS1307 (1101 000) followed by 0 to indicate a write operation.
2. The first byte of data in the write operation will set the register pointer. For example, if you want to write to the control register you should send 0x07.
  3. Check the acknowledge bit to be sure that DS1307 responded.
  4. If you want to write one or more bytes of data, you should transmit them one byte at a time and check the acknowledge bit at the end of each byte sent. Remember that the register pointer is automatically incremented and you can simply transmit bytes of data to consecutive locations in a multi-byte burst write.
  5. Transmit a STOP bit condition.

## ***Reading from DS1307***

Notice that before reading a byte you should load the address of the byte to the register pointer by doing a write operation as mentioned before.

To read one or more bytes of data from the DS1307 you should do the following steps:

1. To access the DS1307 for a read operation, you need to set the register pointer first. After sending a START condition, you should transmit the address of DS1307 (1101 000) followed by 0 to indicate a write operation (writing the register pointer).
2. Check the acknowledge bit to be sure that DS1307 responded.
3. The byte of data in the write operation will set the register pointer. For example, if you want to read from the control register you should send 0x07. Check the acknowledge bit to be sure that DS1307 responded.
4. Now you need to change the bus direction from a transmit to receive. Send a START condition (a REPEATED START), then transmit the address of DS1307 (1101 000) followed by 1 to indicate a read operation. Check the acknowledge bit to be sure that DS1307 responded.
5. You can read one or more bytes of data. Remember that the register pointer indicates which location will be read. The ACK bit in the I2CMCS register should be set for the master to acknowledge the data received. Also notice that the register pointer is automatically incremented and you can simply receive consecutive bytes of data in a multi-byte burst read.
6. Before reading the last byte, clear the ACK bit in the I2CMCS register. The last byte read will have a NACK to signal the DS1307 that the burst read is complete.
7. Transmit a STOP bit condition.

## ***Setting the Time of DS1307 in TI ARM Tiva***

Program 9-1 initializes the clock at 16:58:55 using the 24-hour clock mode. It uses the single-byte operation for writing seconds, minutes, and hours. Notice that in this program we assume that there is only one master on the bus and we do not deal with checking the BUSBSY bit or arbitration.

### **Program 9-1: Setting the time of DS1307 using single byte write**

```
/* p9_1: I2C to DS1307 single byte writes */
```

```

/* This program communicate with the DS1307 Real-time Clock via I2C. */
/* The seconds, minutes, and hours are written one byte at a time. */
/* Set the time to 16:58:55 */

/* DS1307 parameters:
   fmax = 100 kHz */
/*
I2C1SCL PA6
I2C1SDA PA7 */

#include "TM4C123GH6PM.h"

#define SLAVE_ADDR 0x68      /* 1100 1000 */

void I2C1_init(void);
char I2C1_byteWrite(int slaveAddr, char memAddr, char data);

int main(void)
{
    char timeDateToSet[7] = {0x55, 0x58, 0x16, 0x01, 0x19, 0x10, 0x09};

    I2C1_init();

    /* write hour, minute, second with single byte writes */
    I2C1_byteWrite(SLAVE_ADDR, 0, timeDateToSet[0]); /* second */
    I2C1_byteWrite(SLAVE_ADDR, 1, timeDateToSet[1]); /* minute */
    I2C1_byteWrite(SLAVE_ADDR, 2, timeDateToSet[2]); /* hour */

    for (;;)
    {
    }
}

/* initialize I2C1 as master and the port pins */
void I2C1_init(void)
{
    SYSCTL->RCGCI2C |= 0x02;      /* enable clock to I2C1 */
    SYSCTL->RCGCGPIO |= 0x01;     /* enable clock to GPIOA */

    /* PORTA 7, 6 for I2C1 */
    GPIOA->AFSEL |= 0xC0;        /* PORTA 7, 6 for I2C1 */
    GPIOA->PCTL &= ~0xFF000000; /* PORTA 7, 6 for I2C1 */
    GPIOA->PCTL |= 0x33000000;
    GPIOA->DEN |= 0xC0;          /* PORTA 7, 6 as digital pins */
    GPIOA->ODR |= 0x80;          /* PORTA 7 as open drain */

    I2C1->MCR = 0x10;           /* master mode */
    I2C1->MTPR = 7;              /* 100 kHz @ 16 MHz */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

```

/* Wait until I2C master is not busy and return error code */
/* If there is no error, return 0 */
static int I2C_wait_till_done(void)
{
    while(I2C1->MCS & 1); /* wait until I2C master is not busy */
    return I2C1->MCS & 0xE; /* return I2C error code */
}

/* Write one byte only */
/* byte write: S-(saddr+w)-ACK-maddr-ACK-data-ACK-P */
char I2C1_byteWrite(int slaveAddr, char memAddr, char data)
{
    char error;

    /* send slave address and starting address */
    I2C1->MSA = slaveAddr << 1;
    I2C1->MDR = memAddr;
    I2C1->MCS = 3;                                /* S-(saddr+w)-ACK-maddr-ACK */

    error = I2C_wait_till_done();                  /* wait until write is complete */
    if (error) return error;

    /* send data */
    I2C1->MDR = data;
    I2C1->MCS = 5;                                /* -data-ACK-P */
    error = I2C_wait_till_done();                  /* wait until write is complete */
    while(I2C1->MCS & 0x40);                    /* wait until bus is not busy */
    error = I2C1->MCS & 0xE;
    if (error) return error;

    return 0;           /* no error */
}

```

## Setting the date of DS1307 in TI ARM Tiva

Program 9-2 shows how to set the date to Monday October 19th, 2009. It uses multi-byte burst mode for writing day (of the week), date, month, and year. As you can see in the program, to access the locations of the day, you should write 0x03 into the register pointer and then you can use multi-byte burst write to write values of date, month and year in the consecutive locations. Also, notice that in this code we assume that there is only one master on the bus and we do not deal with checking the BUSBSY bit and arbitration.

### Program 9-2: Setting the date of DS1307 using burst write

```

/* p9_2: I2C to DS1307 multi-byte burst write */

/* This program communicate with the DS1307 Real-time Clock via I2C. The day (of the
week), date, month and year are written using burst write. */
/* It sets the date to Monday Oct. 19th, 2009 */
/*

```

```

DS1307 parameters:
fmax = 100 kHz */

/* I2C1SCL PA6
I2C1SDA PA7 */

#include "TM4C123GH6PM.h"

#define SLAVE_ADDR 0x68      /* 0110 1000 */

void I2C1_init(void);
char I2C1_burstWrite(int slaveAddr, char memAddr, int byteCount, char* data);

int main(void)
{
    char timeDateToSet[7] = {0x55, 0x58, 0x16, 0x01, 0x19, 0x10, 0x09};

    I2C1_init();

    /* use burst write to write day, date, month, and year */
    I2C1_burstWrite(SLAVE_ADDR, 3, 4, &timeDateToSet[3]);

    for (;;)
    {
    }
}

/* initialize I2C1 as master and the port pins */
void I2C1_init(void)
{
    SYSCTL->RCGCI2C |= 0x02;      /* enable clock to I2C1 */
    SYSCTL->RCGCGPIO |= 0x01;     /* enable clock to GPIOA */

    /* PORTA 7, 6 for I2C1 */
    GPIOA->AFSEL |= 0xC0;        /* PORTA 7, 6 for I2C1 */
    GPIOA->PCTL &= ~0xFF000000; /* PORTA 7, 6 for I2C1 */
    GPIOA->PCTL |= 0x33000000;
    GPIOA->DEN |= 0xC0;         /* PORTA 7, 6 as digital pins */
    GPIOA->ODR |= 0x80;          /* PORTA 7 as open drain */

    I2C1->MCR = 0x10;           /* master mode */
    I2C1->MTPR = 7;              /* 100 kHz @ 16 MHz */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

/* Wait until I2C master is not busy and return error code */
/* If there is no error, return 0 */
static int I2C_wait_till_done(void)
{
    while(I2C1->MCS & 1);    /* wait until I2C master is not busy */
}

```

```

        return I2C1->MCS & 0xE; /* return I2C error code */
}

/* Use burst write to write multiple bytes to consecutive locations */
/* burst write: S-(saddr+w)-ACK-maddr-ACK-data-ACK-data-ACK-...-data-ACK-P */
char I2C1_burstWrite(int slaveAddr, char memAddr, int byteCount, char* data)
{
    char error;

    if (byteCount <= 0)
        return -1; /* no write was performed */

    /* send slave address and starting address */
    I2C1->MSA = slaveAddr << 1;
    I2C1->MDR = memAddr;
    I2C1->MCS = 3; /* S-(saddr+w)-ACK-maddr-ACK */

    error = I2C_wait_till_done(); /* wait until write is complete */
    if (error) return error;

    /* send data one byte at a time */
    while (byteCount > 1)
    {
        I2C1->MDR = *data++; /* write the next byte */
        I2C1->MCS = 1; /* -data-ACK- */
        error = I2C_wait_till_done();
        if (error) return error;
        byteCount--;
    }

    /* send last byte and a STOP */
    I2C1->MDR = *data++; /* write the last byte */
    I2C1->MCS = 5; /* -data-ACK-P */
    error = I2C_wait_till_done();
    while(I2C1->MCS & 0x40); /* wait until bus is not busy */
    if (error) return error;

    return 0; /* no error */
}

```

## Reading the date and time of DS1307 in TI ARM Tiva

Program 9-3 shows how to read the date and time from DS1307 using multi-byte burst mode for reading. As you can see in the program, the register pointer is set to 0 and then you can use multi-byte burst read to read the values of second, minute, hour, day, date, month and year in the consecutive locations. Also, notice that in this code we assume that there is only one master on the bus and we do not deal with checking the BUSSY bit and arbitration.

### Program 9-3: Reading date time of DS1307 using burst read

```

/* p9_3: I2C to DS1307 Multi-byte burst read */

/* This program communicate with the DS1307 Real-time Clock via I2C. */
/* It reads all seven time/date registers using burst read. */

```

```

/* DS1307 parameters:
fmax = 100 kHz

I2C1SCL PA6
I2C1SDA PA7 */

#include "TM4C123GH6PM.h"

#define SLAVE_ADDR 0x68      /* 0110 1000 */

void I2C1_init(void);
char I2C1_read(int slaveAddr, char memAddr, int byteCount, char* data);

int main(void)
{
    char timeDateReadBack[7];

    I2C1_init();

    for (;;)
    {
        /* use burst read to read time and date */
        I2C1_read(SLAVE_ADDR, 0, 7, timeDateReadBack);
        /* display date time here */
    }
}

/* initialize I2C1 as master and the port pins */
void I2C1_init(void)
{
    SYSCTL->RCGCI2C |= 0x02;      /* enable clock to I2C1 */
    SYSCTL->RCGCGPIO |= 0x01;     /* enable clock to GPIOA */

    /* PORTA 7, 6 for I2C1 */
    GPIOA->AFSEL |= 0xC0;         /* PORTA 7, 6 for I2C1 */
    GPIOA->PCTL &= ~0xFF000000;   /* PORTA 7, 6 for I2C1 */
    GPIOA->PCTL |= 0x33000000;
    GPIOA->DEN |= 0xC0;           /* PORTA 7, 6 as digital pins */
    GPIOA->ODR |= 0x80;           /* PORTA 7 as open drain */

    I2C1->MCR = 0x10;            /* master mode */
    I2C1->MTPR = 7;              /* 100 kHz @ 16 MHz */
}

/* This function is called by the startup assembly */
/* code to perform system specific initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

/* Wait until I2C master is not busy and return error code */
/* If there is no error, return 0 */
static int I2C_wait_till_done(void)
{

```

```

while(I2C1->MCS & 1); /* wait until I2C master is not busy */
return I2C1->MCS & 0xE; /* return I2C error code */
}

/* Read memory */
/* read: S-(saddr+w)-ACK-maddr-ACK-R-(saddr+r)-ACK-data-ACK-data-ACK-...-data-NACK-P */
char I2C1_read(int slaveAddr, char memAddr, int byteCount, char* data)
{
    char error;

    if (byteCount <= 0)
        return -1; /* no read was performed */

    /* send slave address and starting address */
    I2C1->MSA = slaveAddr << 1;
    I2C1->MDR = memAddr;
    I2C1->MCS = 3; /* S-(saddr+w)-ACK-maddr-ACK */
    error = I2C_wait_till_done();
    if (error)
        return error;

    /* to change bus from write to read, send restart with slave addr */
    I2C1->MSA = (slaveAddr << 1) + 1; /* restart: -R-(saddr+r)-ACK */

    if (byteCount == 1) /* if last byte, don't ack */
        I2C1->MCS = 7; /* -data-NACK-P */
    else /* else ack */
        I2C1->MCS = 0xB;
    error = I2C_wait_till_done();
    if (error) return error;

    *data++ = I2C1->MDR; /* store the data received */

    if (--byteCount == 0) /* if single byte read, done */
    {
        while(I2C1->MCS & 0x40); /* wait until bus is not busy */
        return 0; /* no error */
    }

    /* read the rest of the bytes */
    while (byteCount > 1)
    {
        I2C1->MCS = 9; /* -data-ACK- */
        error = I2C_wait_till_done();
        if (error) return error;
        byteCount--;
        *data++ = I2C1->MDR; /* store data received */
    }

    I2C1->MCS = 5; /* -data-NACK-P */
    error = I2C_wait_till_done();
    *data = I2C1->MDR; /* store data received */
    while(I2C1->MCS & 0x40); /* wait until bus is not busy */

    return 0; /* no error */
}

```

## Review Questions

1. True or false. All of the RAM contents of the DS1307 are nonvolatile.
2. How many bytes of RAM in the DS1307 are set aside for the clock and date?
  - (a) 7 bytes
  - (b) 8 bytes
  - (c) 56 bytes
  - (d) 64 bytes
3. How many bytes of RAM in the DS1307 are set aside for general-purpose applications?
  - (a) 7 bytes
  - (b) 8 bytes
  - (c) 56 bytes
  - (d) 64 bytes
4. True or false. The DS1307 has a single pin for data.
5. Which pin of the DS1307 is used for clock in I<sub>2</sub>C connection?
6. What is the common voltage for V<sub>bat</sub> in the DS1307?
7. True or false. The value of the CH bit is zero at power-up time.
8. What is the address location for the control register?
  - (a) 07H
  - (b) 08H
  - (c) 56H
  - (d) 64H

## Answers to Review Questions

### Section 9-1

1. True
2. 9, the 9th bit is for acknowledge
3. False, START and STOP conditions are generated when the SCL is high.
4. Arbitration
5. False, the master who won the arbitration will continue.

### Section 9-2

1. True
2. True
3. True
4. RCGCI2C
5. I2CMTPR

### Section 9-3

1. True
2. a
3. c ( $64 - 8 = 56$  bytes)
4. True
5. SCL
6. 3V

7. False

8. a



# **Chapter 10: Relay, Optoisolator, and Stepper Motor Interfacing**

Microcontrollers are widely used in motor control. We also use relays and optoisolators in motor control. This chapter discusses motor control and shows ARM interfacing with relays, optoisolators, and stepper motors.

## Section 10.1: Relays and Optoisolators

This section begins with an overview of the basic operations of electromechanical relays, solid-state relays, reed switches, and optoisolators. Then we describe how to interface them to the ARM. We use the C language programs to demonstrate their control.

### Electromechanical relays

A *relay* is an electrically controllable switch widely used in industrial controls, automobiles, and appliances. It allows the isolation of two separate sections of a system with two different voltage sources. For example, a +5 V system can be isolated from a 120 V system by placing a relay between them. One such relay is called an *electromechanical* (or *electromagnetic*) *relay* (EMR) as shown in Figure 10-1. The EMRs have three components: the coil, spring, and contacts.

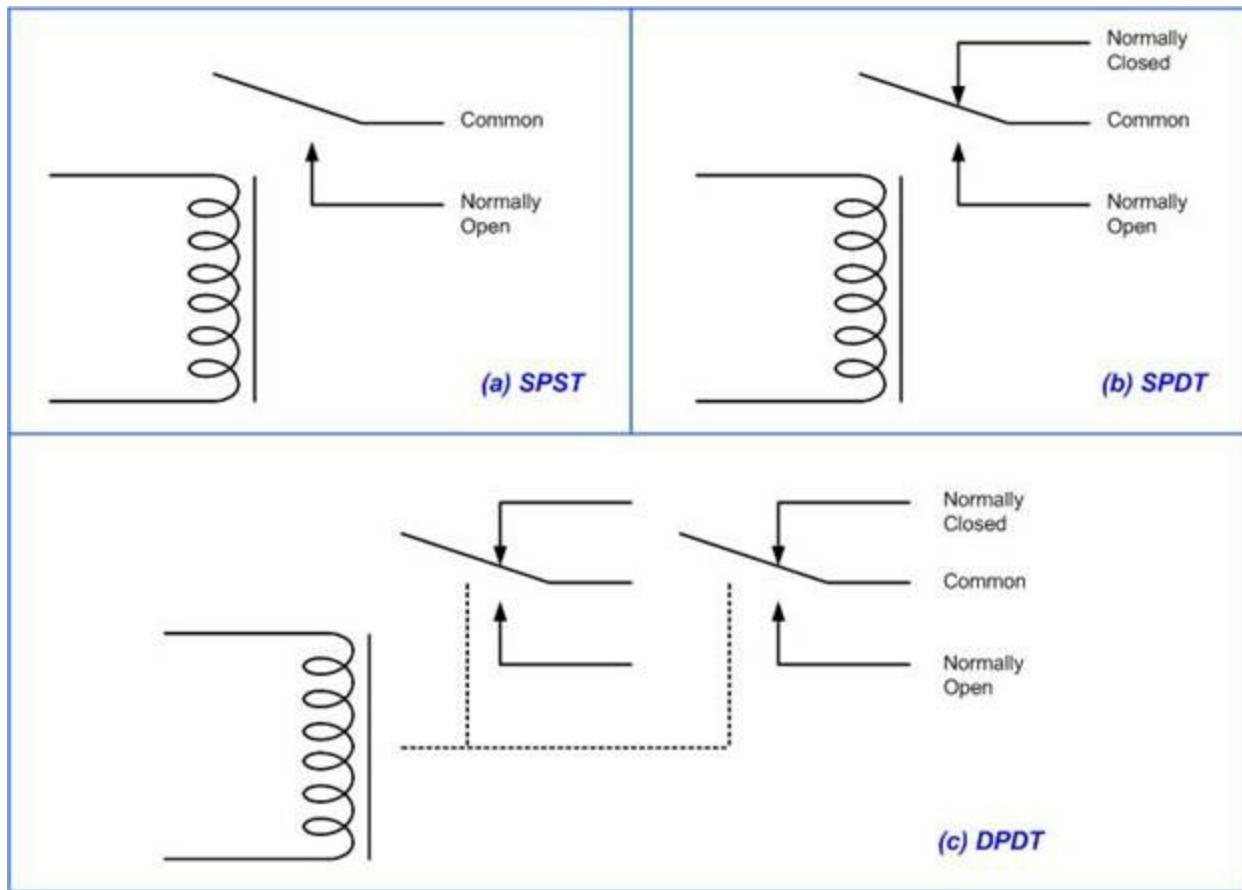


Figure 10-1: Relay Diagrams

In Figure 10-1, a digital +5 V on the left side can control a 12 V motor on the right side without any physical contact between them. When current flows through the coil, a magnetic field is created around the coil (the coil is energized), which causes the armature to be attracted to the coil. The armature's contact acts like a switch and closes or opens the circuit. When the coil is not energized, a spring pulls the armature to its normal state of open or closed. In the block diagram for electromechanical relays (EMR) we do not show the spring, but it does exist internally. There are all types of relays for all kinds of applications. In choosing a relay the following characteristics need to be considered:

1. The contacts can be normally open (NO) or normally closed (NC). In the NC type, the

contacts are closed when the coil is not energized. In the NO type, the contacts are open when the coil is unenergized.

2. There can be one or more contacts. For example, we can have SPST (single pole, single throw), SPDT (single pole, double throw), and DPDT (double pole, double throw) relays.
3. The voltage and current needed to energize the coil. The voltage can vary from a few volts to 50 volts, while the current can be from a few mA to 20 mA. The relay has a minimum voltage, below which the coil will not be energized. This minimum voltage is called the “pull-in” voltage. In the datasheets for relays we might not see current, but rather coil resistance. The V/R will give you the pull-in current. For example, if the coil voltage is 5 V, and the coil resistance is 500 ohms, we need a minimum of 10 mA ( $5\text{ V}/500\text{ ohms} = 10\text{ mA}$ ) pull-in current.
4. The maximum DC/AC voltage and current that can be handled by the contacts. This is in the range of a few volts to hundreds of volts, while the current can be from a few amps to 40 A or more, depending on the relay. Notice the difference between this voltage/current specification and the voltage/current needed for energizing the coil. The fact that one can use such a small amount of voltage/current on one side to handle a large amount of voltage/current on the other side is what makes relays so widely used in industrial controls. Examine Table 10-1 for some relay characteristics.

Part No.	Contact Form	Coil Volts	Coil Ohms	Contact Volts	Current
106462CP	SPST-NO	5 VDC	500	100 VDC	0.5 A
138430CP	SPST-NO	5 VDC	500	100 VDC	0.5 A
106471CP	SPST-NO	12 VDC	1000	100 VDC	0.5 A
138448CP	SPST-NO	12 VDC	1000	100 VDC	0.5 A
129875CP	DPDT	5 VDC	62.5	30 VDC	1 A

Table 10-1: Selected DIP Relay Characteristics ([www.Jameco.com](http://www.Jameco.com))

## Driving a relay

Digital systems and microcontroller pins lack sufficient current to drive the relay. While the relay's coil needs around 10 mA to be energized, the microcontroller's pin can provide a maximum of 8 mA current. For this reason, we place a driver, such as the ULN2803, or a transistor between the microcontroller and the relay as shown in Figure 10-2. In the circuit we can turn the lamp on and off by setting and clearing the PB0.

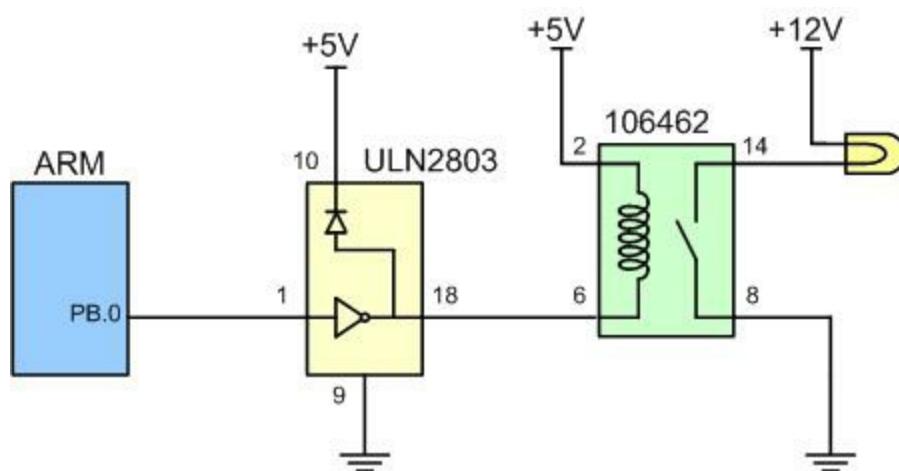


Figure 10-2: ARM Connection to Relay

Program 10-1 turns the lamp shown in Figure 10-2 on and off by energizing and de-energizing the relay every second.

### Program 10-1

```
/* p10_1: Relay control */
/* This program turns the relay connect to PB0 on and off every second. */

#include "TM4C123GH6PM.h"

void delayMs(int n);

int main(void)
{
    SYSCTL->RCGCGPIO |= 0x02; /* enable clock to GPIOB */

    /* PORTB 0 for relay control */
    GPIOB->DIR |= 0x01;          /* PORTB 0 as output */
    GPIOB->DEN |= 0x01;          /* PORTB 0 as digital pins */

    for (;;)
    {
        GPIOB->DATA ^= 1;        /* toggle PB0 at 1 Hz */
        delayMs(500);
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly */
/* code to perform system specific initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}
```

## Solid-state relay

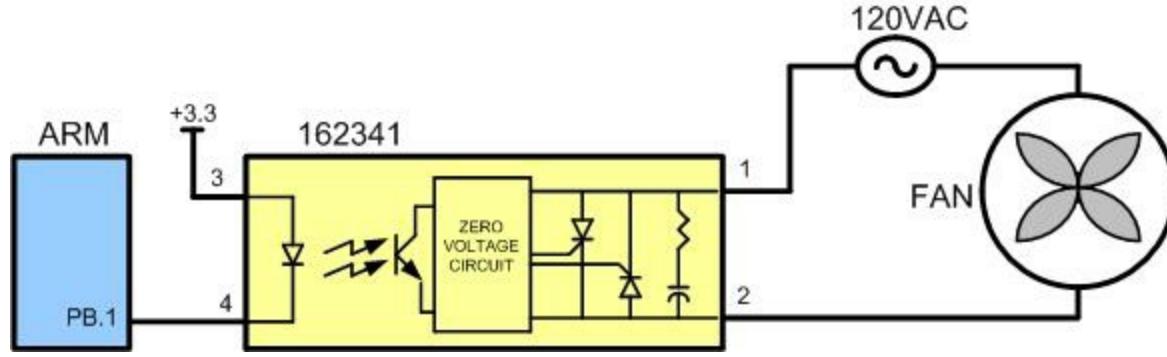
Another widely used relay is the solid-state relay. See Table 10-2.

Part No.	Contact Style	Control Volts	Contact Volts	Contact Current
<b>143058CP</b>	SPST	4-32 VDC	240 VAC	3 A
<b>139053CP</b>	SPST	3-32 VDC	240 VAC	25 A
<b>162341CP</b>	SPST	3-32 VDC	240 VAC	10 A
<b>172591CP</b>	SPST	3-32 VDC	60 VAC	2 A

<b>175222CP</b>	SPST	3-32 VDC	60 VAC	4 A
<b>176647CP</b>	SPST	3-32 VDC	120 VAC	5 A

**Table 10-2: Selected Solid-State Relay Characteristics ([www.Jameco.com](http://www.Jameco.com))**

In this relay, there is no coil, spring, or mechanical contact switch. The entire relay is made out of semiconductor materials. Because no mechanical parts are involved in solid-state relays, their switching response time is much faster than that of electromechanical relays. Another advantage of the solid-state relay is its greater life expectancy. The life cycle for the electromechanical relay can vary from a few hundred thousand to a few million operations. Wear and tear on the contact points can cause the relay to malfunction after a while. Solid-state relays, however, have no such limitations. Extremely low input current and small packaging make solid-state relays ideal for microcontroller and logic control switching. They are widely used in controlling pumps, solenoids, alarms, and other power applications. Some solid-state relays have a phase control option, which is ideal for motor-speed control and light-dimming applications. Figure 10-3 shows control of a fan using a solid-state relay (SSR).



**Figure 10-3: ARM Connection to a Solid-State Relay**

## Reed switch

Another popular switch is the reed switch. When the reed switch is placed in a magnetic field, the contact is closed. When the magnetic field is removed, the contact is forced open by its spring. See Figure 10-4. The reed switch is ideal for moist and marine environments where it can be submerged in fuel or water. Reed switches are also widely used in dirty and dusty atmospheres because they are tightly sealed.

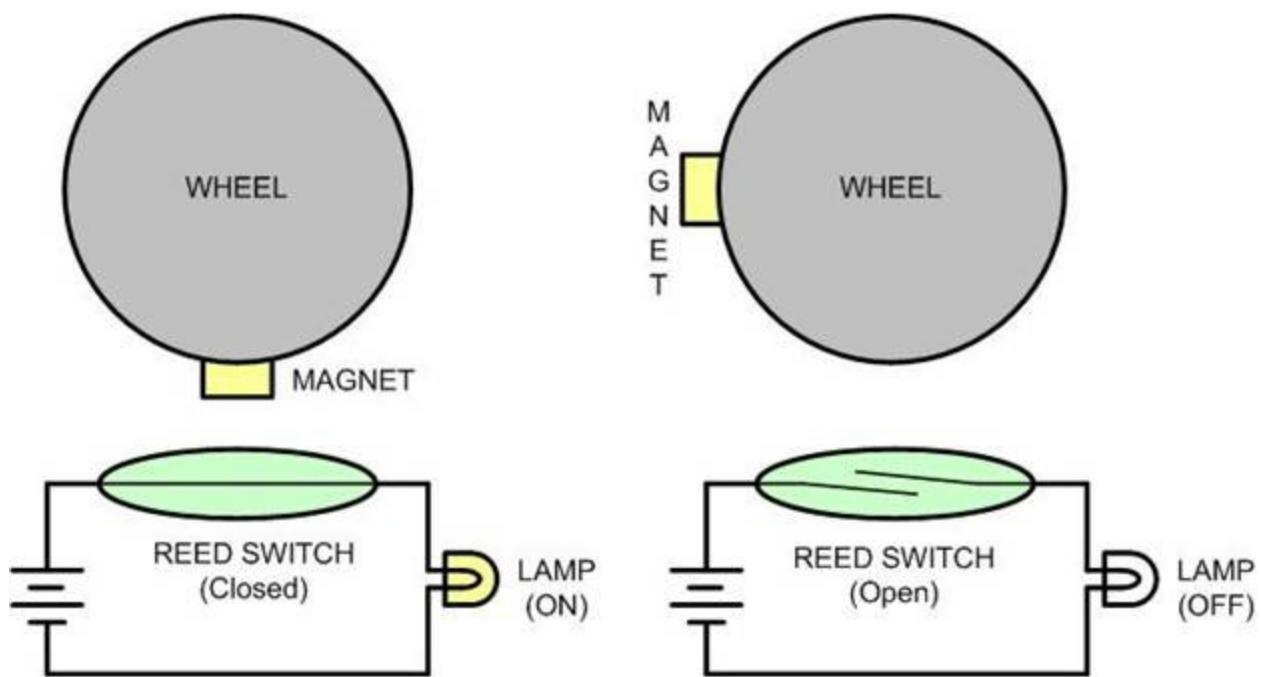


Figure 10-4: Reed Switch and Magnet Combination

## Optoisolator

In some applications we use an optoisolator (also called optocoupler) to isolate two parts of a system. An example is driving a motor. Motors can produce what is called *back EMF*, a high-voltage spike produced by a sudden change of current as indicated in the formula  $V = Ldi/dt$ . In situations such as printed circuit board design, we can reduce the effect of this unwanted voltage spike (called *ground bounce*) by using decoupling capacitors (see Appendix A). In systems that have inductors (coil winding), such as motors, a decoupling capacitor or a diode will not do the job. In such cases we use optoisolators. An optoisolator has an LED (light-emitting diode) transmitter and a photosensor receiver, separated from each other by a gap. When current flows through the diode, it transmits a signal light across the gap and the receiver produces the same signal with the same phase but a different current and amplitude. See Figure 10-5. Optoisolators are also widely used in communication equipment such as modems. This device allows a computer to be connected to a telephone line without risk of damage from high voltage of telephone line. The gap between the transmitter and receiver of optoisolators prevents the electrical voltage surge from reaching the system.

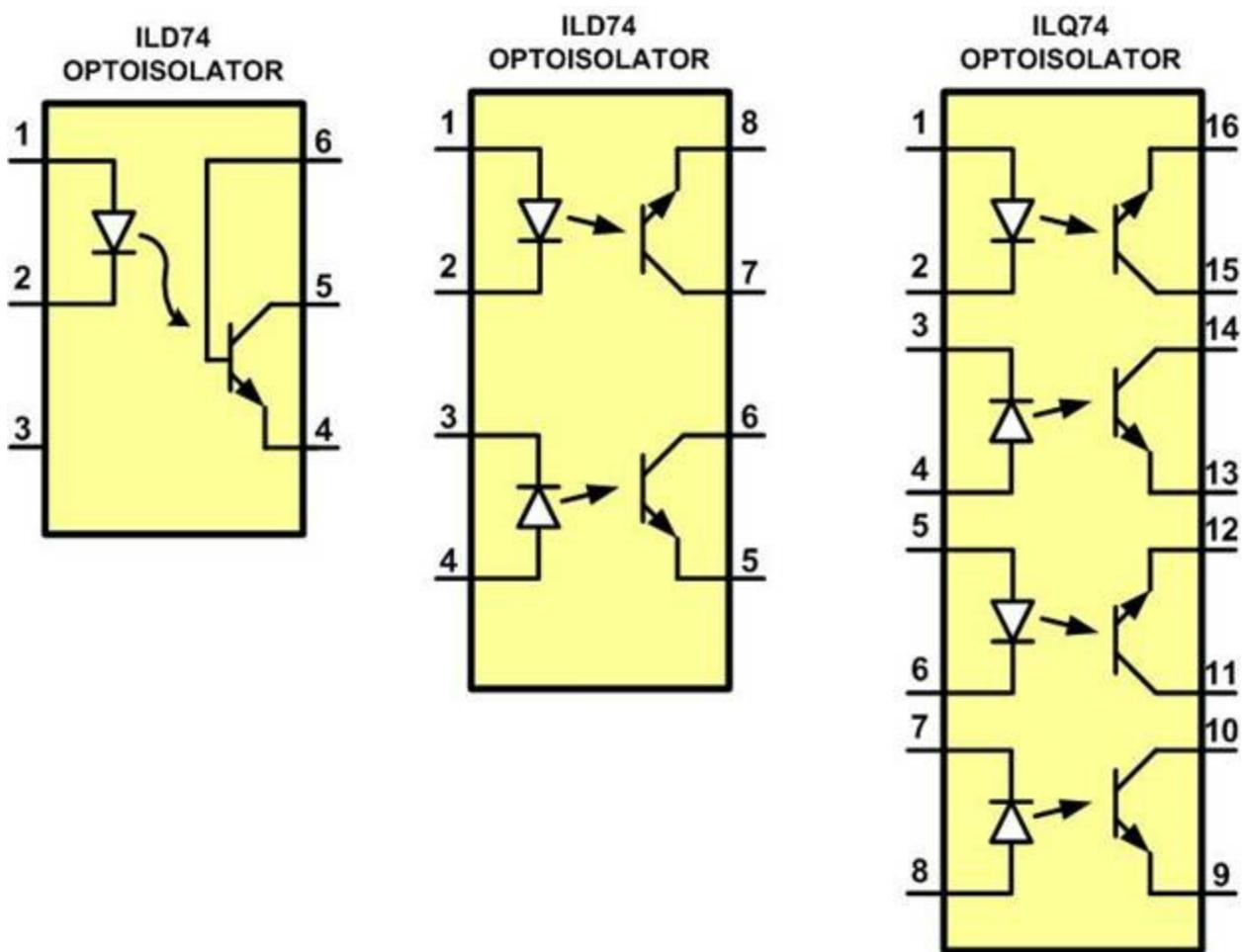


Figure 10-5: Optoisulator Package Examples

### Interfacing an optoisolator

The optoisolator comes in a small IC package with four or more pins. There are also packages that contain more than one optoisolator. When placing an optoisolator between two circuits, we must use two separate voltage sources, one for each side, as shown in Figure 10-6. Unlike relays, no drivers need to be placed between the microcontroller/digital output and the optoisolators.

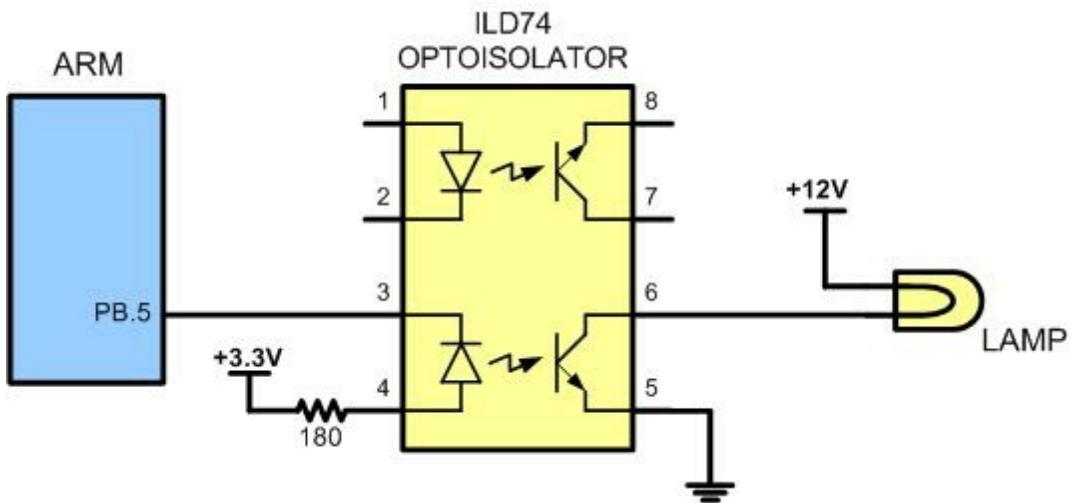


Figure 10-6: Controlling a Lamp via an Optoisolator

### Review Questions

1. Give one application where you would use a relay.

2. Why do we place a driver between the microcontroller and the relay?
3. What is an NC relay?
4. Why are relays that use coils called electromechanical relays?
5. What is the advantage of a solid-state relay over EMR?
6. What is the advantage of an optoisolator over an EMR?

## Section 10.2: Stepper Motor Interfacing

This section begins with an overview of the basic operation of stepper motors. Then we describe how to interface a stepper motor to the ARM. Finally, we use C language programs to demonstrate control of the rotation of stepper motor.

### Stepper motors

A *stepper motor* is a widely used device that translates electrical pulses into mechanical movement. In applications such as dot matrix printers and robotics, the stepper motor is used for position control. Stepper motors commonly have a permanent magnet *rotor* (also called the *shaft*) surrounded by a stator (see Figure 10-7).

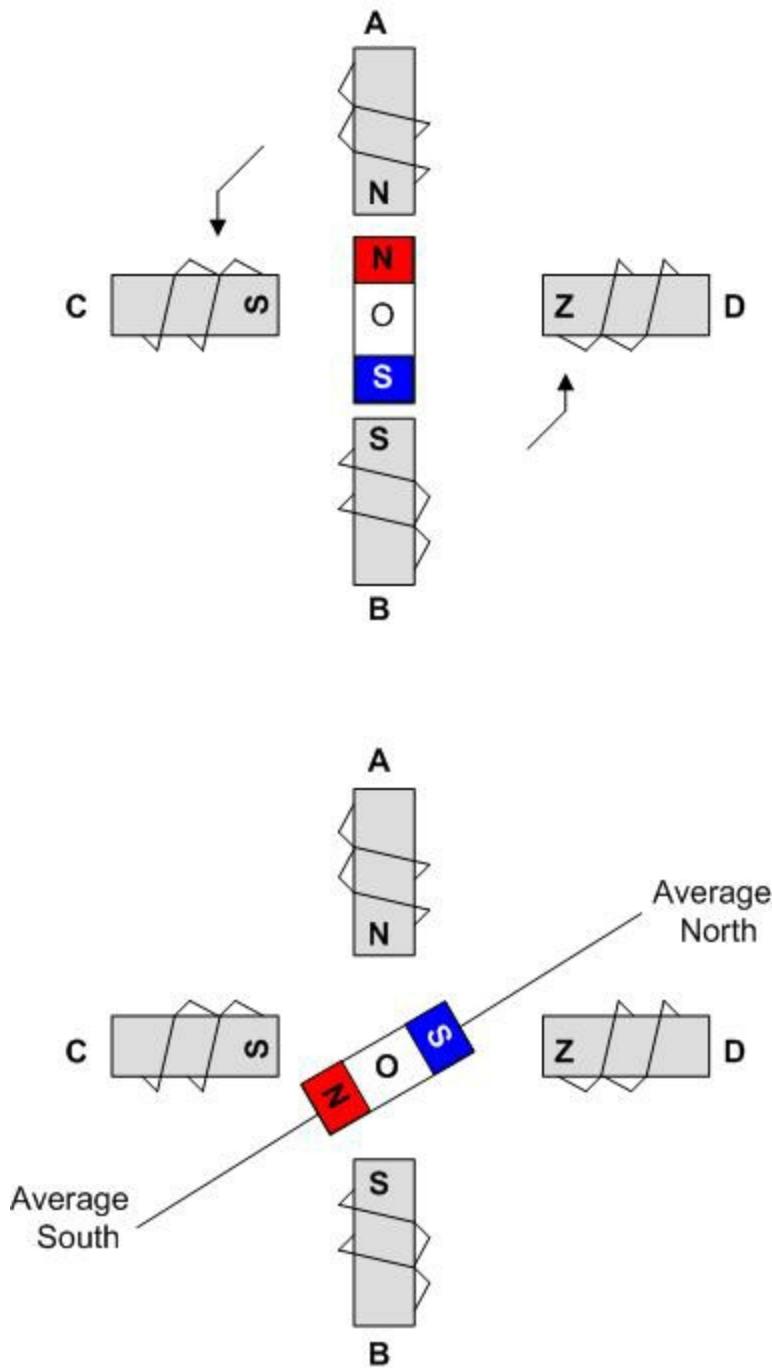


Figure 10-7: Rotor Alignment

There are also steppers called *variable reluctance stepper motors* that do not have a permanent magnet rotor. The most common stepper motors have four stator windings that are

paired with a center-tapped common as shown in Figure 10-8.

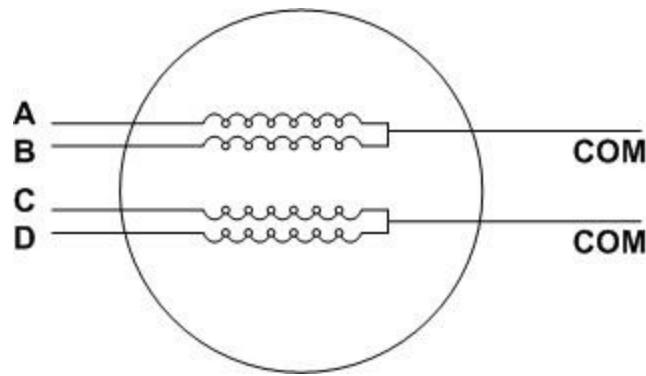


Figure 10-8: Stator Winding Configuration

This type of stepper motor is commonly referred to as a four-phase or unipolar stepper motor. The center tap allows a change of current direction in each of two coils when a winding is grounded, thereby resulting in a polarity change of the stator. Notice that while a conventional motor shaft runs freely, the stepper motor shaft moves in a fixed repeatable increment, which allows it to move to a precise position. This repeatable fixed movement is possible as a result of basic magnetic theory where poles of the same polarity repel and opposite poles attract. The direction of the rotation is dictated by the stator poles. The stator poles are determined by the current sent through the wire coils. As the direction of the current is changed, the polarity is also changed causing the reverse motion of the rotor. The stepper motor discussed here has a total of six leads: four leads representing the four stator windings and two commons for the center-tapped leads. As the sequence of power is applied to each stator winding, the rotor will rotate. There are several widely used sequences, each of which has a different degree of precision. Table 10-3 shows a two-phase, four-step stepping sequence.

Clockwise	Step #	Winding A	Winding B	Winding C	Winding D	Counter Clockwise
	1	1	0	0	1	
	2	1	1	0	0	
	3	0	1	1	0	
	4	0	0	1	1	

Table 10-3: Normal Four-Step Sequence

Note that although we can start with any of the sequences in Table 10-3, once we start we must continue in the proper order. For example, if we start with step 3 (0110), we must continue in the sequence of steps 4, 1, 2, and so on.

### Step angle

How much movement is associated with a single step? This depends on the internal construction of the motor, in particular the number of teeth on the stator and the rotor. The step angle is the minimum degree of rotation associated with a single step. Various motors have different step angles. Table 10-4 shows some step angles for various motors. In Table 10-4, notice the term steps per revolution. This is the total number of steps needed to rotate one complete rotation or 360 degrees (e.g., 180 steps  $\times$  2 degrees = 360).

Step Angle	Step per Revolution

0.72	500
1.8	200
2.0	180
2.5	144
5.0	72
7.5	48
15	24

**Table 10-4: Stepper Motor Step Angles**

It must be noted that perhaps contrary to one's initial impression, a stepper motor does not need more terminal leads for the stator to achieve smaller steps. All the stepper motors discussed in this section have four leads for the stator winding and two COM wires for the center tap. Although some manufacturers set aside only one lead for the common signal instead of two, they always have four leads for the stators. See Example 10-1. Next we discuss some associated terminology in order to understand the stepper motor further.

### Example 10-1

Describe the ARM connection to the stepper motor of Figure 10-9 and code a program to rotate it continuously.

#### Solution:

The following steps show the ARM connection to the stepper motor and its programming:

1. Use an ohmmeter to measure the resistance of the leads. This should identify which COM leads are connected to which winding leads.
2. The common wire(s) are connected to the positive side of the motor's power supply. In many motors, +5 V is sufficient.
3. The four leads of the stator winding are controlled by four bits of the ARM port (PB0–PB3). Because the microcontroller lacks sufficient current to drive the stepper motor windings, we must use a driver such as the ULN2003 (or ULN2803) to energize the stator. Instead of the ULN2003, we could have used transistors as drivers, as shown in Figure 10-11. However, notice that if transistors are used as drivers, we must also use diodes to take care of inductive current generated when the coil is turned off. One reason that using the ULN2003 is preferable to the use of transistors as drivers is that the ULN2003 has an internal diode to take care of back EMF.

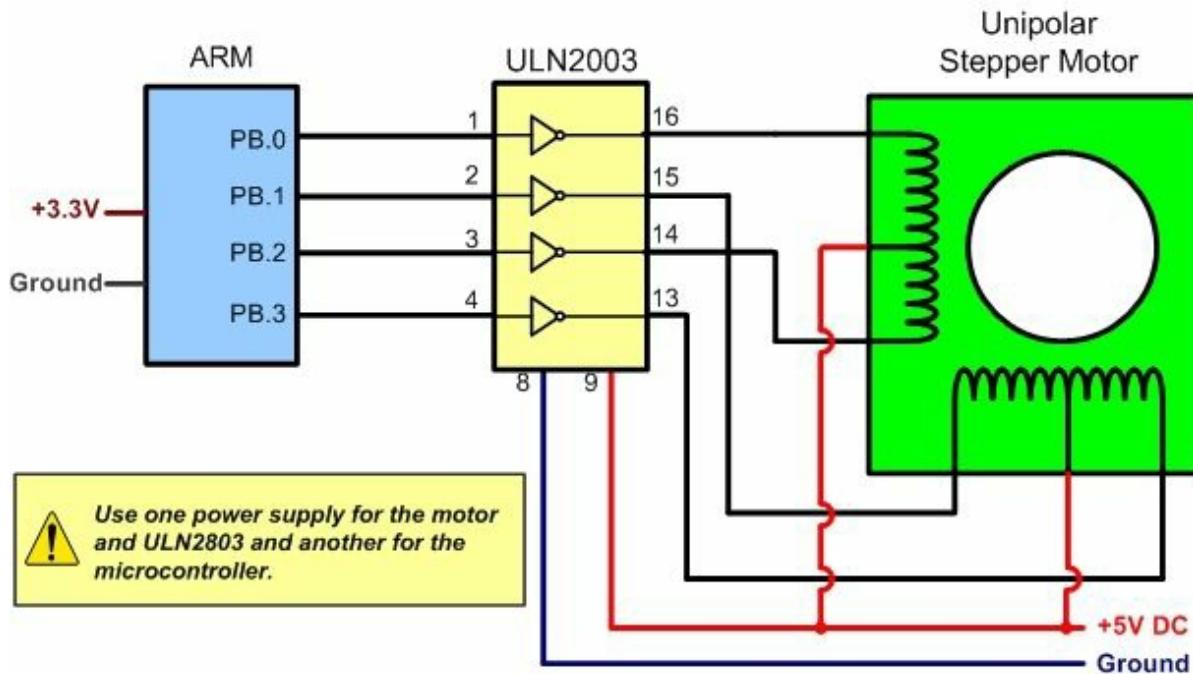


Figure 10-9: ARM Connection to Stepper Motor

### Steps per second and RPM relation

The relation between RPM (revolutions per minute), steps per revolution, and steps per second is as follows.

$$\text{Step per second} = \frac{\text{RPM} \times \text{Steps per revolution}}{60}$$

### The 4-step sequence and number of teeth on rotor

The switching sequence shown earlier in Table 10-3 is called the 4-step switching sequence because after four steps the same two windings will be "ON". How much movement is associated with these four steps? Therefore, in a stepper motor with 200 steps per revolution, the rotor has 50 teeth because  $4 \times 50 = 200$  steps are needed to complete one revolution. This leads to the conclusion that the minimum step angle is always a function of the number of teeth on the rotor. In other words, the smaller the step angle, the more teeth the rotor has. See Example 10-2.

### Example 10-2

Give the number of times the four-step sequence in Table 10-3 must be applied to a stepper motor to make an 80-degree move if the motor has a 2-degree step angle.

#### Solution:

A motor with a 2-degree step angle has the following characteristics:

Step angle: 2 degrees

Steps per revolution: 180

Number of rotor teeth: 45

Movement per 4-step sequence: 8 degrees

To move the rotor 80 degrees, we need to send 10 consecutive 4-step sequences, because  $10 \times 4 \text{ steps} \times 2 \text{ degrees} = 80 \text{ degrees}$ .

Looking at Example 10-2, one might wonder what happens if we want to move 45 degrees, because the steps are 2 degrees each. To provide finer resolutions, all stepper motors allow what is called an 8-step switching sequence. The 8-step sequence is also called half-stepping, because in the 8-step sequence each step is half of the normal step angle. For example, a motor with a 2-degree step angle can be used as a 1-degree step angle if the sequence of Table 10-5 is applied.

Clockwise	Step #	Winding A	Winding B	Winding C	Winding D	Counter Clockwise
	1	1	0	0	1	
	2	1	0	0	0	
	3	1	1	0	0	
	4	0	1	0	0	
	5	0	1	1	0	
	6	0	0	1	0	
	7	0	0	1	1	
	8	0	0	0	1	

Table 10-5: Half-Step 8-Step Sequence

## Motor speed

The motor speed, measured in steps per second (steps/s), is a function of the switching rate. Notice in Example 10-1 that by changing the length of the time delay loop, we can achieve various rotation speeds.

## Holding torque

The following is a definition of holding torque: "With the motor shaft at standstill or zero rpm condition, the amount of torque, from an external source, required to break away the shaft from its holding position. This is measured with rated voltage and current applied to the motor." The unit of torque is ounce-inch (or kg-cm).

## Wave drive 4-step sequence

In addition to the 8-step and the 4-step sequences discussed earlier, there is another sequence called the *wave drive 4-step sequence*. It is shown in Table 10-6.

Clockwise	Step #	Winding A	Winding B	Winding C	Winding D	Counter Clockwise
	1	1	0	0	0	
	2	0	1	0	0	
	3	0	0	1	0	
	4	0	0	0	1	

Table 10-6: Wave Drive 4-Step Sequence

Notice that the 8-step sequence of Table 10-5 is simply the combination of the wave drive 4-step and normal 4-step normal sequences shown in Tables 10-6 and 10-3, respectively. Experimenting with the wave drive 4-step sequence is left to the reader.

## Unipolar versus bipolar stepper motor interface

There are three common types of stepper motor interfacing: universal, unipolar, and

bipolar. They can be identified by the number of connections to the motor. A universal stepper motor has eight, while the unipolar has six and the bipolar has four. The universal stepper motor can be configured for all three modes, while the unipolar can be either unipolar or bipolar. Obviously the bipolar cannot be configured for universal nor unipolar mode. Table 10-7 shows selected stepper motor characteristics.

Part No.	Step Angle	Drive System	Volts	Phase Resistance	Current
151861CP	7.5	unipolar	5 V	9 ohms	550 mA
171601CP	3.6	unipolar	7 V	20 ohms	350 mA
164056CP	7.5	bipolar	5 V	6 ohms	800 mA

Table 10-7: Selected Stepper Motor Characteristics ([www.Jameco.com](http://www.Jameco.com))

Figure 10-10 shows the basic internal connections of all three type of configurations.

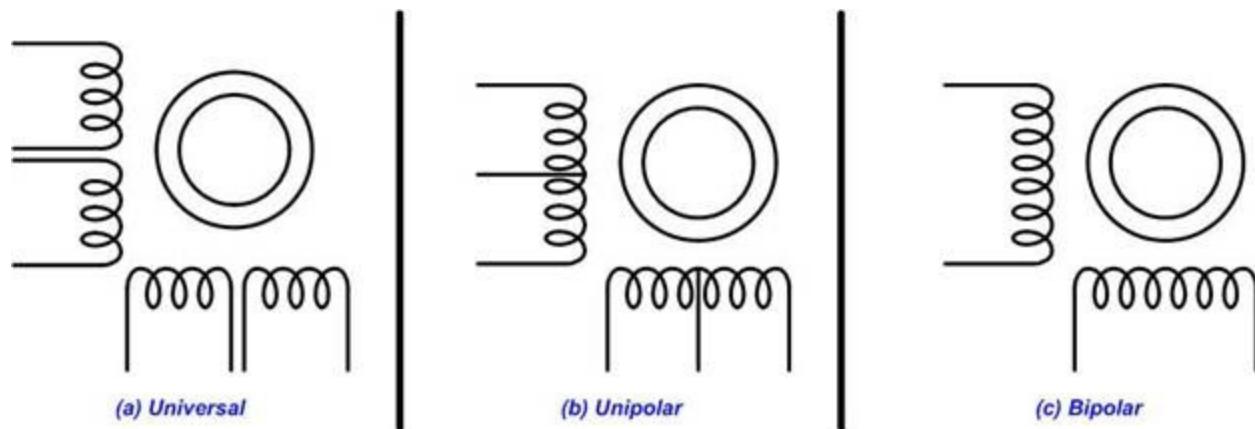


Figure 10-10: Common Stepper Motor Types

Unipolar stepper motors can be controlled using the basic interfacing shown in Figure 10-11, whereas the bipolar stepper requires H-Bridge circuitry. Bipolar stepper motors require a higher operational current than the unipolar; the advantage of this is a higher holding torque.

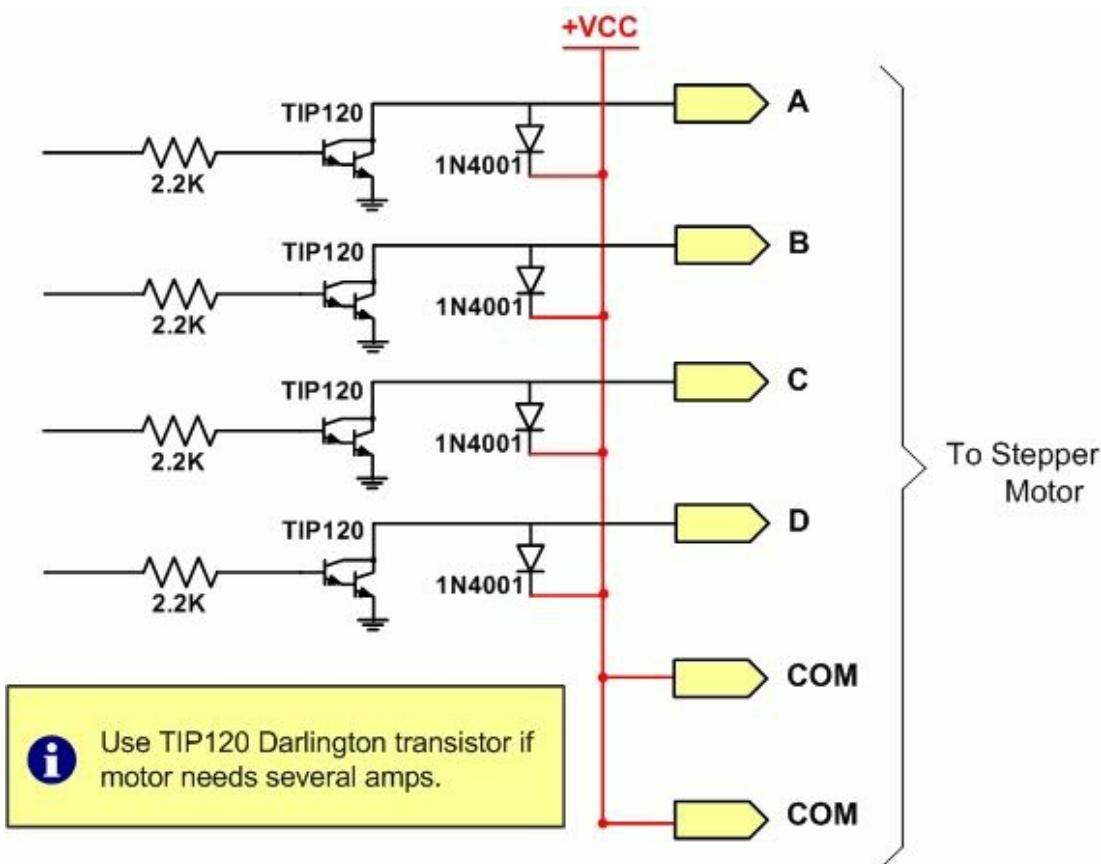


Figure 10-11: Using Transistors for Stepper Motor Driver

### Using transistors as drivers

Figure 10-11 shows an interface to a unipolar stepper motor using transistors. Diodes are used to reduce the back EMF spike created when the coils are energized and de-energized, similar to the electromechanical relays discussed earlier. TIP transistors can be used to supply higher current to the motor. Table 10-8 lists the common industrial Darlington transistors. These transistors can accommodate higher voltages and currents.

NPN	PNP	V <sub>CEO</sub> (volts)	I <sub>C</sub> (amps)	hfe (common)
TIP110	TIP115	60	2	1000
TIP111	TIP116	80	2	1000
TIP112	TIP117	100	2	1000
<b>TIP120</b>	TIP125	60	5	1000
TIP121	TIP126	80	5	1000
TIP122	TIP127	100	5	1000
TIP140	TIP145	60	10	1000
TIP141	TIP146	80	10	1000
TIP142	TIP147	100	10	1000

Table 10-8: Darlington Transistor Listing

### Controlling stepper motor via optoisolator

In the first section of this chapter we examined the optoisolator and its use. Optoisolators are widely used to isolate the stepper motor's EMF voltage and keep it from damaging the digital/microcontroller system. This is shown in Figure 10-12.

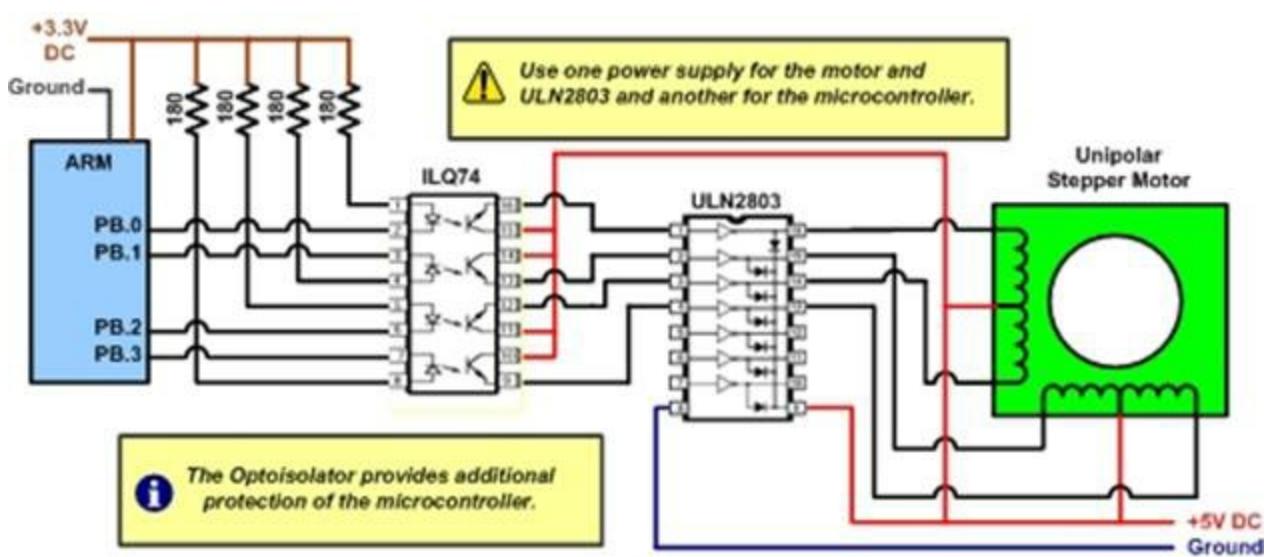


Figure 10-12: Controlling Stepper Motor via Optoisolator

See Program 10-2.

### Program 10-2

```
/*
 * p10_2.c: Stepper motor control */
/* This program controls a unipolar stepper motor using PB 3, 2, 1, 0. */

#include "TM4C123GH6PM.h"

void delayMs(int n);

int delay = 10;
int direction = 0;

int main(void)
{
    const char steps[ ] = {0x9, 0x3, 0x6, 0xC};
    int i = 0;

    SYSCTL->RCGCGPIO |= 0x02; /* enable clock to GPIOB */

    /* PORTB 3, 2, 1, 0 for motor control */
    GPIOB->DIR |= 0x0F;          /* PORTB 3, 2, 1, 0 as output */
    GPIOB->DEN |= 0x0F;          /* PORTB 3, 2, 1, 0 as digital pins */

    for (;;)
    {
        if (direction)
            GPIOB->DATA = (steps[i++ & 3]);
        else
            GPIOB->DATA = (steps[i-- & 3]);
        delayMs(delay);
    }

    /* delay n milliseconds (16 MHz CPU clock) */
    void delayMs(int n)
    {
        int i, j;
```

```

        for(i = 0 ; i < n; i++)
            for(j = 0; j < 3180; j++)
                {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## Review Questions

1. Give the 4-step sequence of a stepper motor if we start with 0110.
2. A stepper motor with a step angle of 5 degrees has \_\_\_\_ steps per revolution.
3. Why do we put a driver between the microcontroller and the stepper motor?

## Answers to Review Questions

### Section 10.1

1. With a relay we can use a 5 V digital system to control 12 V–120 V devices such as horns and appliances.
2. Because microcontroller/digital outputs lack sufficient current to energize the relay, we need a driver.
3. When the coil is not energized, the contact is closed.
4. When current flows through the coil, a magnetic field is created around the coil, which causes the armature to be attracted to the coil.
5. It is faster and needs less current to get energized.
6. It is smaller and can be connected to the microcontroller directly without a driver.

### Section 10.2

1. 0110, 0011, 1001, 1100 for clockwise; and 0110, 1100, 1001, 0011 for counterclockwise
2. 72
3. The microcontroller pins do not provide sufficient current to drive the stepper motor.



## Chapter 11: PWM and DC Motor Control

This chapter discusses the topic of PWM (pulse width modulation) and shows ARM interfacing with DC motors. The characteristics of DC motors are discussed along with their interfacing to the ARM. We use C programming examples to create PWM pulses.

## Section 11.1: DC Motor Interfacing and PWM

This section begins with an overview of the basic operation of the DC motors. Then we describe how to interface a DC motor to the ARM. Finally, we use C language programs to demonstrate the concept of pulse width modulation (PWM) and show how to control the speed and direction of a DC motor.

### DC motors

A direct current (DC) motor is a widely used device that translates electrical current into mechanical movement. In the DC motor we have only + and – leads. Connecting them to a DC voltage source moves the motor in one direction. By reversing the polarity, the DC motor will rotate in the opposite direction. One can easily experiment with the DC motor. For example, some small fans used in many motherboards to cool the CPU are run by DC motors. While a stepper motor moves in discrete steps of 1 to 15 degrees, the DC motor moves continuously. In a stepper motor, if we know the starting position we can easily count the number of steps the motor has moved and calculate the final position of the motor. This is not possible in a DC motor. The maximum speed of a DC motor is indicated in RPM and is given in the data sheet. The DC motor has two types of RPM: no-load and loaded. The manufacturer's data sheet gives the no-load RPM. The no-load RPM can be from a few thousand to tens of thousands. The RPM is reduced when moving a load and it decreases as the load is increased. For example, a drill turning a screw has a much lower RPM speed than when it is in the no-load situation. DC motors also have voltage and current ratings. The nominal voltage is the voltage for that motor under normal conditions, and can be from 1 to 150 V, depending on the motor. As we increase the voltage, the RPM goes up. The current rating is the current consumption when the nominal voltage is applied with no load, and can be from 25 mA to a few amps. As the load increases, the RPM is decreased, unless the current or voltage provided to the motor is increased, which in turn increases the torque. With a fixed voltage, as the load increases, the current (power) consumption of a DC motor is increased. If we overload the motor it will stall, and that can damage the motor due to the heat generated by high current consumption.

### Unidirectional control

Figure 11-1 shows the DC motor clockwise (CW) and counterclockwise (CCW) rotations.

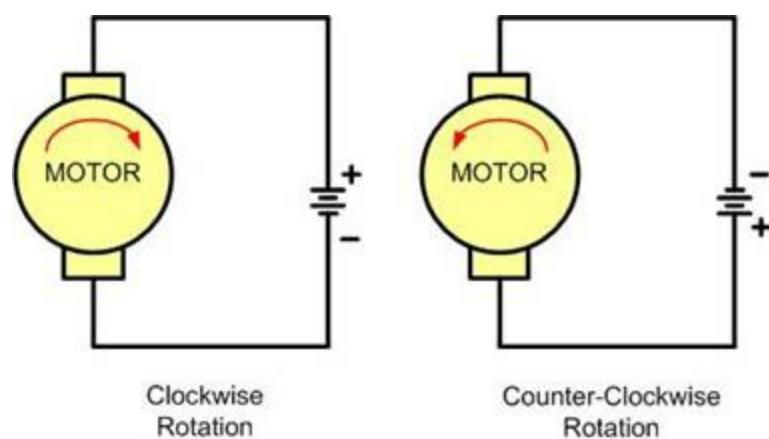


Figure 11-1: DC Motor Rotation (Permanent Magnet Field)

See Table 11-1 for selected DC motors.

Part No.	Nominal Volts	Volt Range	Current	RPM	Torque
154915CP	3 V	1.5–3 V	0.070 A	5,200	4.0 g-cm
154923CP	3 V	1.5–3 V	0.240 A	16,000	8.3 g-cm
177498CP	4.5 V	3–14 V	0.150 A	10,300	33.3 g-cm
181411CP	5 V	3–14 V	0.470 A	10,000	18.8 g-cm

Table 11-1: Selected DC Motor Characteristics (<http://www.Jameco.com>)

## Bidirectional control

With the help of relays, transistor circuit or some specially designed chips we can change the direction of the DC motor rotation. Figures 11-2 through 11-4 show the basic concepts of the H-Bridge control of DC motors.

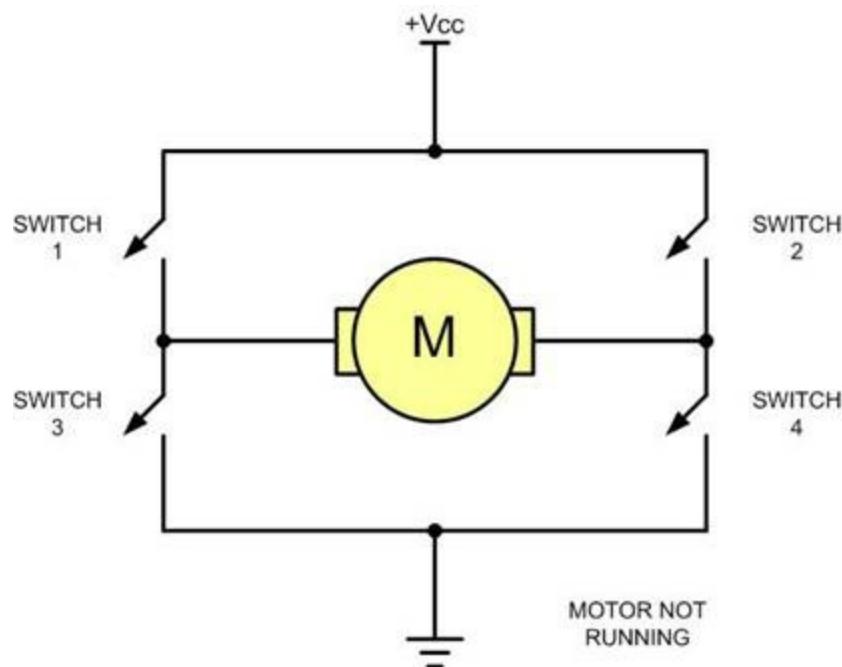


Figure 11-2: H-Bridge Motor Configuration

Figure 11-2 shows the connection of an H-Bridge using simple switches. All the switches are open, which does not allow the motor to turn.

Figure 11-3 shows the switch configuration for turning the motor in one direction. When switches 1 and 4 are closed, current is allowed to pass through the motor.

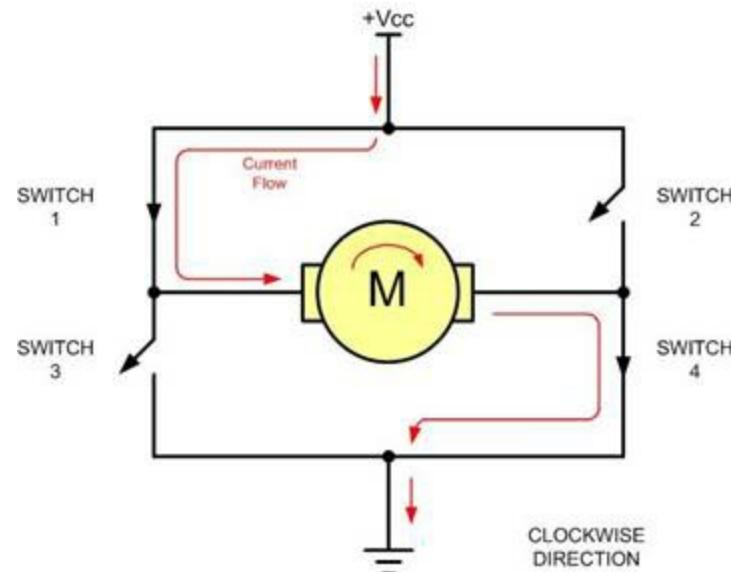


Figure 11-3: H-Bridge Motor Clockwise Configuration

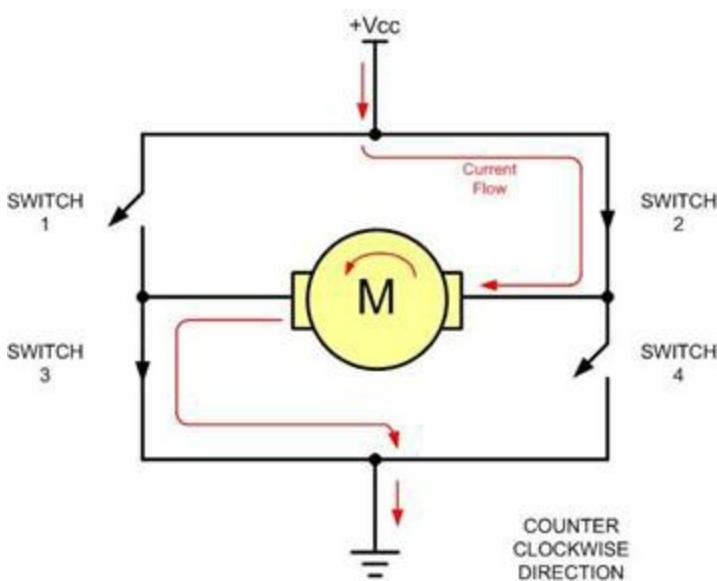


Figure 11-4: H-Bridge Motor Counterclockwise Configuration

Figure 11-4 shows the switch configuration for turning the motor in the opposite direction from the configuration of Figure 11-3. When switches 2 and 3 are closed, current is allowed to pass through the motor.

Figure 11-5 shows an invalid configuration. Current flows directly to ground, creating a short circuit. The same effect occurs when switches 1 and 3 are closed or switches 2 and 4 are closed.

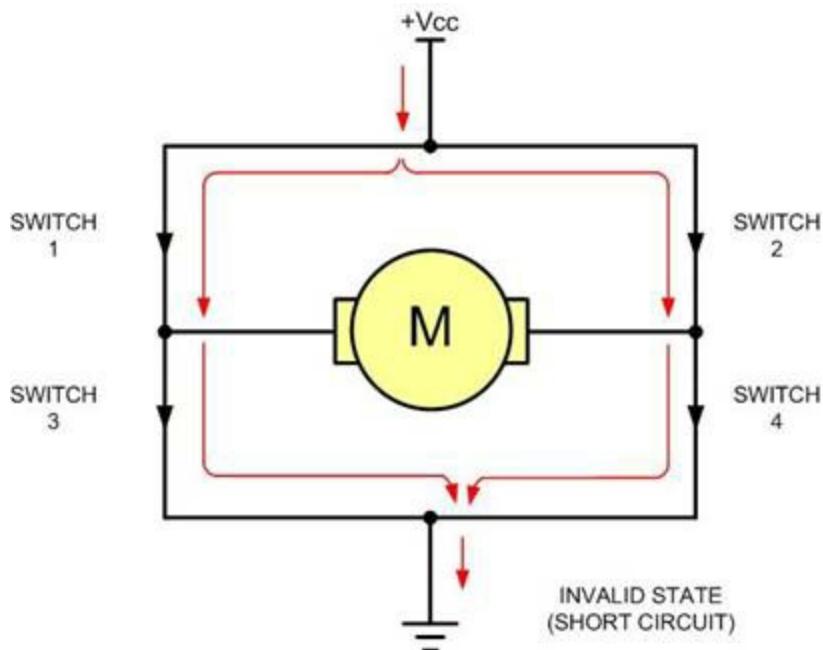


Figure 11-5: H-Bridge in an Invalid Configuration

Table 11-2 shows some of the logic configurations for the H-Bridge design.

Motor Operation	SW1	SW2	SW3	SW4
Off	Open	Open	Open	Open
Clockwise	Closed	Open	Open	Closed
Counterclockwise	Open	Closed	Closed	Open

Invalid

Closed

Closed

Closed

Closed

Table 11-2: Some H-Bridge Logic Configurations for Figure 11-2

H-Bridge control can be created using relays, transistors, or a single IC solution such as the L298. When using relays and transistors, you must ensure that invalid configurations do not occur.

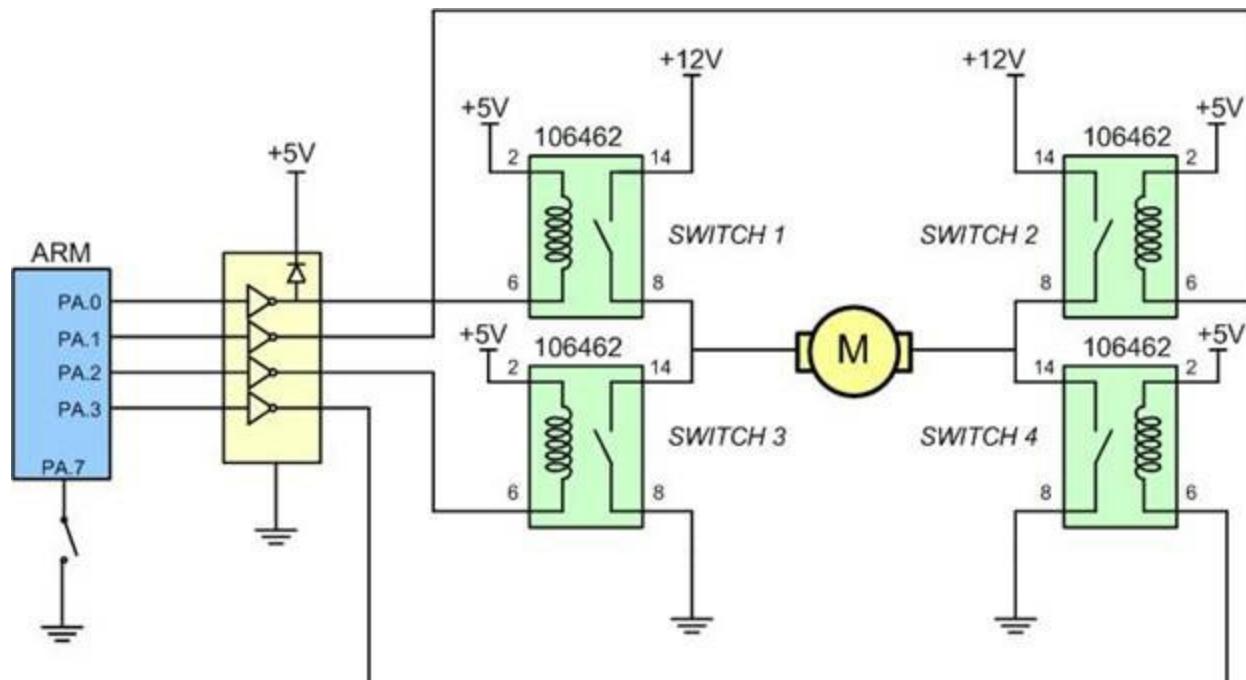
Although we do not show the relay control of an H-Bridge, Example 11-1 shows a simple program to operate a basic H-Bridge.

### Example 11-1

A switch is connected to pin PA7 (PORTA.7). Using relays, make the H-Bridge in Table 11-2 and write the proper program. We must perform the following:

- (a) If PA7 = 0, the DC motor moves clockwise.
- (b) If PA7 = 1, the DC motor moves counterclockwise.

#### Solution 1 (Using SPST Relays):



```

int main ()
{
    GPIOA->DIR = 0x0F;
    GPIOA->DEN = 0xFF;

    while(1)
    {
        if((GPIOA->DATA&(1<<7)) != 0) /* PA7 == 1 */
        {
            GPIOA->DATA &= 0xF0; /* make all switches open */
        }
    }
}

```

```

        delay(); /*wait 0.1 second */
        GPIOA->DATA |= 0x06; /* close SW2 & SW3 */

        while((GPIOA->DATA&(1<<7)) != 0); /* PA7 == 1 */

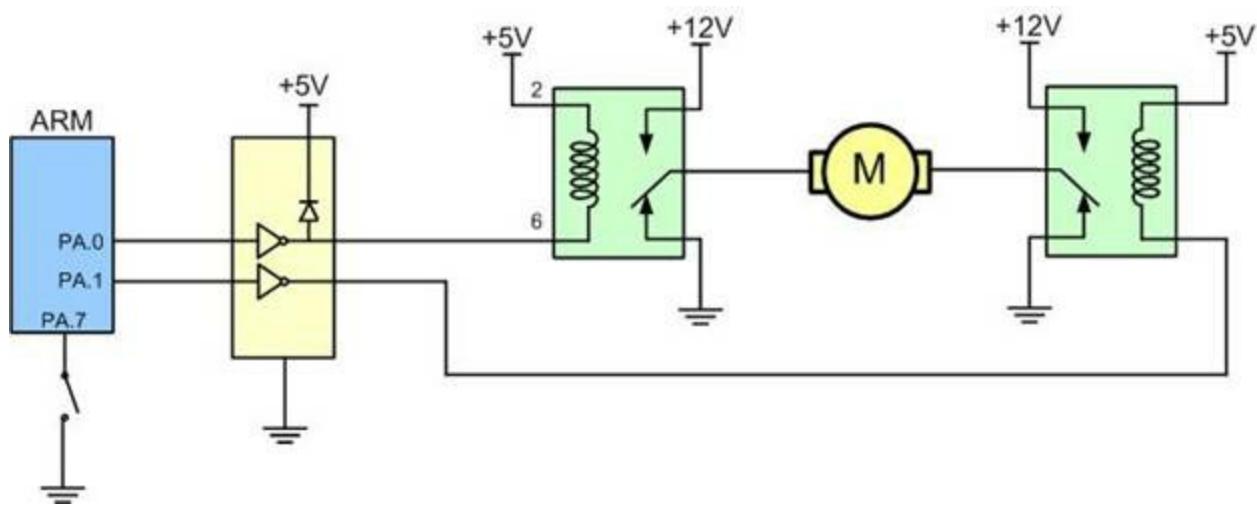
    }
else
{
    GPIOA->DATA &= 0xF0; /* make all switches open */
    delay(); /* wait 0.1 second */
    GPIOA->DATA |= 0x09; /* close SW1 & SW4 */

    while((GPIOA->DATA&(1<<7)) == 0); /*PA7 == 0 */
}
}
}

```

## Solution 2 (Using SPDT Relays):

The H-bridge can also be made using two SPDT relays as shown in the following figure.



```

int main ()
{
    GPIOA->DIR = 0x03;
    GPIOA->DEN = 0xFF;

    while(1)
    {
        if((GPIOA->DATA&(1<<7)) != 0) /* PA7 == 1 */
        {
            GPIOA->DATA &= 0x1; /* Relay 1 = Off */
            GPIOA->DATA |= 0x2; /* Relay 2 = On */
        }
    }
}

```

```

    else
    {
        GPIOA->DATA &= 0x2; /* Relay 2 = Off */
        GPIOA->DATA |= 0x1; /* Relay 1 = On */
    }
}

```

Figure 11-6 shows the connection of the L298N to a Tiva LaunchPad. Be aware that the L298N will generate heat during operation. For sustained operation of the motor, use a heat sink.

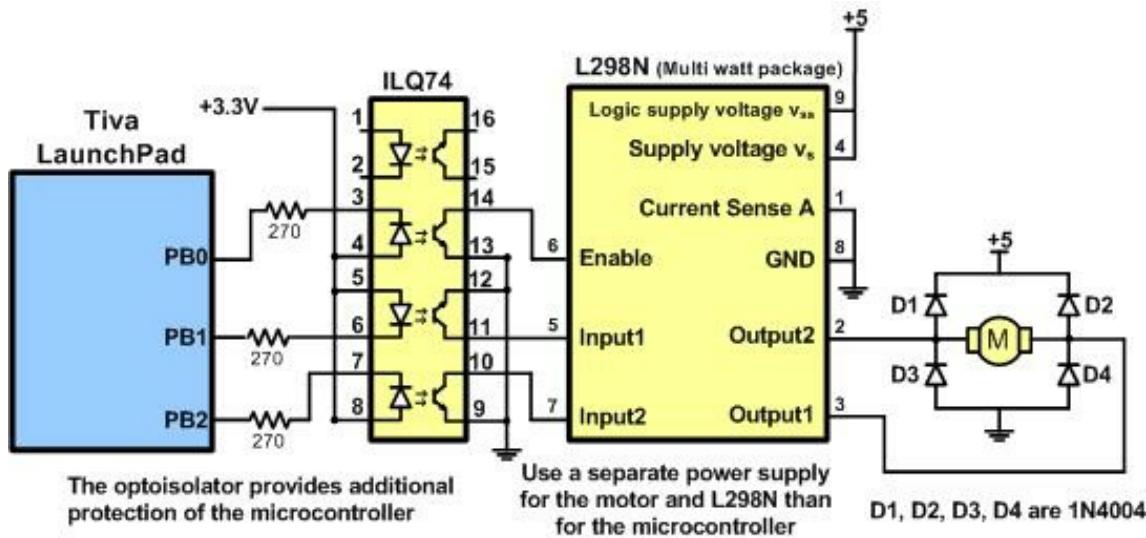


Figure 11-6: Bidirectional Motor Control Using an L298N Chip

## Pulse width modulation (PWM)

The speed of the motor depends on three factors: (a) load, (b) voltage, and (c) current. For a given fixed load we can maintain a steady speed by using a method called pulse width modulation (PWM). By changing (modulating) the width of the pulse applied to the DC motor we can increase or decrease the amount of power provided to the motor, thereby increasing or decreasing the motor speed. Notice that, although the voltage has a fixed amplitude, it has a variable duty cycle. That means the wider the pulse, the higher the speed. PWM is so widely used in DC motor control that many microcontrollers come with an on-chip PWM circuitry. In such microcontrollers all we have to do is load the proper registers with the values of the high and low portions of the desired pulse, and the rest is taken care of by the microcontroller. This allows the microcontroller to do other things. For microcontrollers without on-chip PWM circuitry, we must create the various duty cycle pulses using software, which prevents the microcontroller from doing other things. The ability to control the speed of the DC motor using PWM is one reason that DC motors are preferable over AC motors. AC motor speed is dictated by the AC frequency of the voltage applied to the motor and the frequency is generally fixed.

As a result, we cannot control the speed of the AC motor when the load is increased. As will be shown later, we can also change the DC motor's direction and torque. See Figure 11-7 for PWM comparisons.

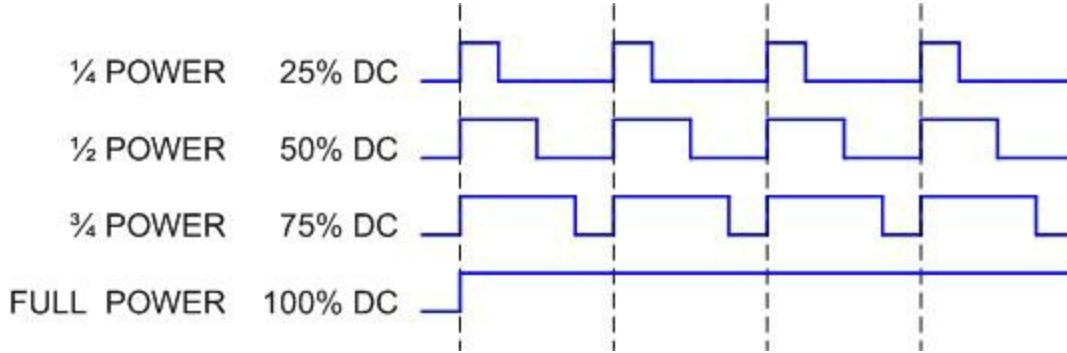


Figure 11-7: Pulse Width Modulation Comparison

## DC motor control with optoisolator

The optoisolator is indispensable in many motor control applications. Figures 11-8 and 11-9 show the connections to a simple DC motor using a bipolar and a MOSFET transistor. Notice that the microcontroller is protected from EMI created by motor brushes by using an optoisolator and a separate power supply.

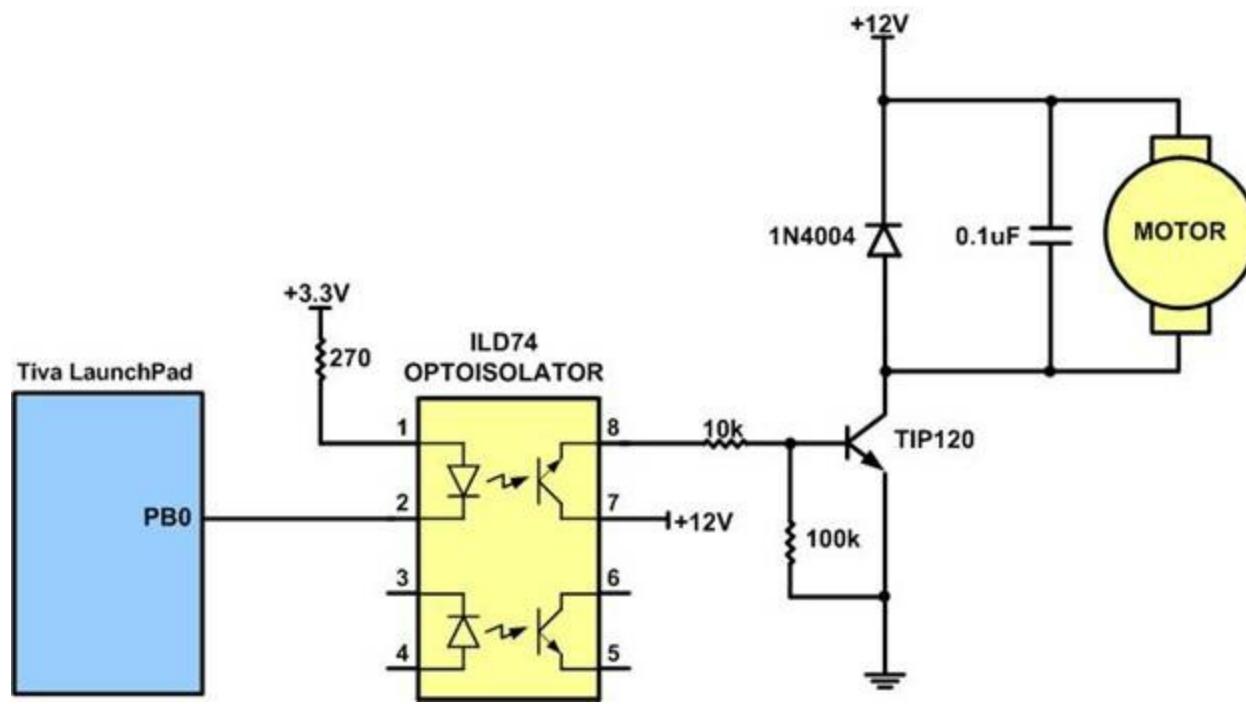


Figure 11-8: DC Motor Connection Using a Darlington Transistor

Figures 11-8 and 11-9 show optoisolators for single directional motor control, and the same principle should be used for most motor applications. Separating the power supplies of the motor and logic will reduce the possibility of damage to the control circuit. Figure 11-8 shows the connection of a bipolar transistor to a motor. Protection of the control circuit is provided by the optoisolator. The motor and the microcontroller use separate power supplies. The separation of power supplies also allows the use of high-voltage motors. Notice that we use a decoupling capacitor across the motor; this helps reduce the EMI created by the motor. The motor is switched on by clearing bit PB0.

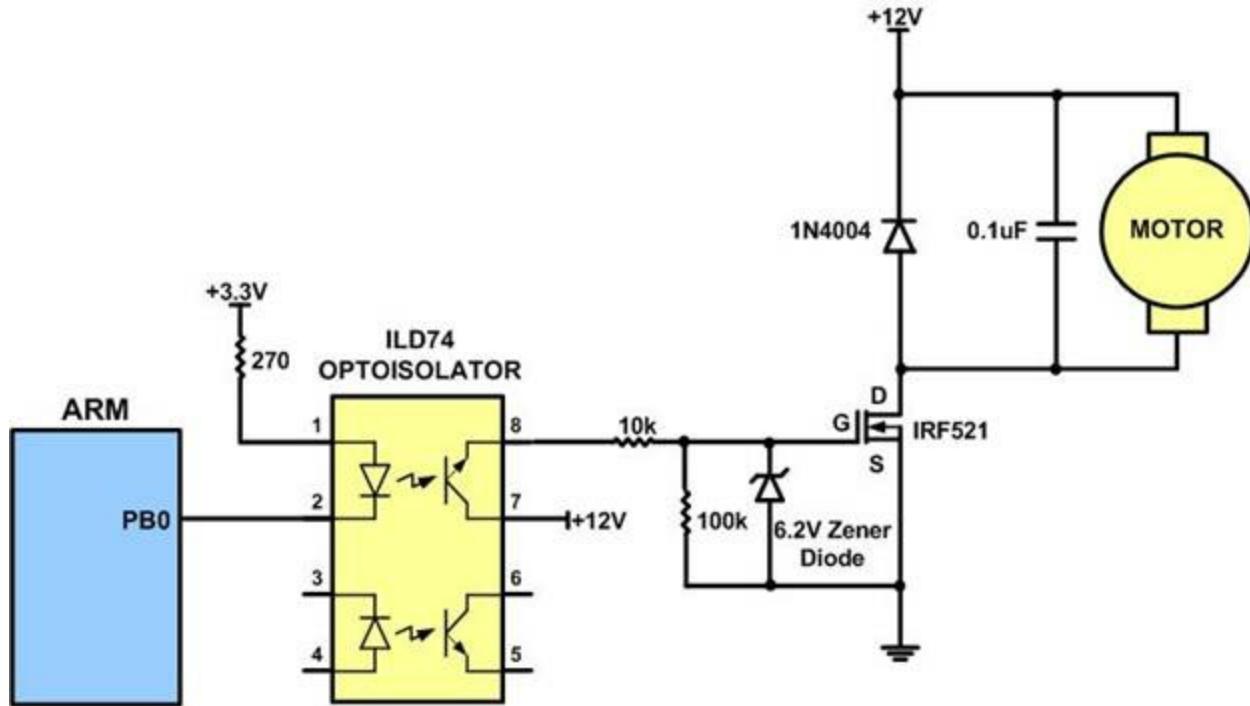


Figure 11-9: DC Motor Connection Using a MOSFET Transistor

Figure 11-9 shows the connection of a MOSFET transistor. The optoisolator protects the microcontroller from EMI. The Zener diode is required for the transistor to reduce gate voltage below the rated maximum value.

## Review Questions

- True or false. The permanent magnet field DC motor has only two leads for + and – voltages.
- True or false. As with a stepper motor, one can control the exact angle of a DC motor's move.
- Why do we put a driver between the microcontroller and the DC motor?
- How do we change a DC motor's rotation direction?
- What is stall in a DC motor?
- The RPM rating given for the DC motor is for \_\_\_\_\_ (no-load, loaded).

## Section 11.2: Programming PWM in TI Tiva LaunchPad

The TI Tiva LaunchPad comes with two on-chip PWM (Pulse Width Modulation) modules. The PWM0 and PWM1 modules are located at Base addresses of 0x40028000 and 0x40029000, respectively. Following are the basic facts about the PWM features of TI Tiva LaunchPad:

1. There are two PWM modules, PWM0 and PWM1.
2. Each PWM module has four Generators.
3. Each Generator has a Counter (Timer).
4. Each Generator has two Compare registers CMPA and CMPB.
5. Each Generator has two output pins, which means there are a total of 8 PWM pins per module.
6. The Counter of the Generator can be programmed to count-up or count-up/down.
7. As the Counter counts , it may change the output pin when the counter value matches the Compare registers, reaches zero, or is reloaded. The options for output pin change are toggle, driven LOW, or driven High. See Figure 11-10.

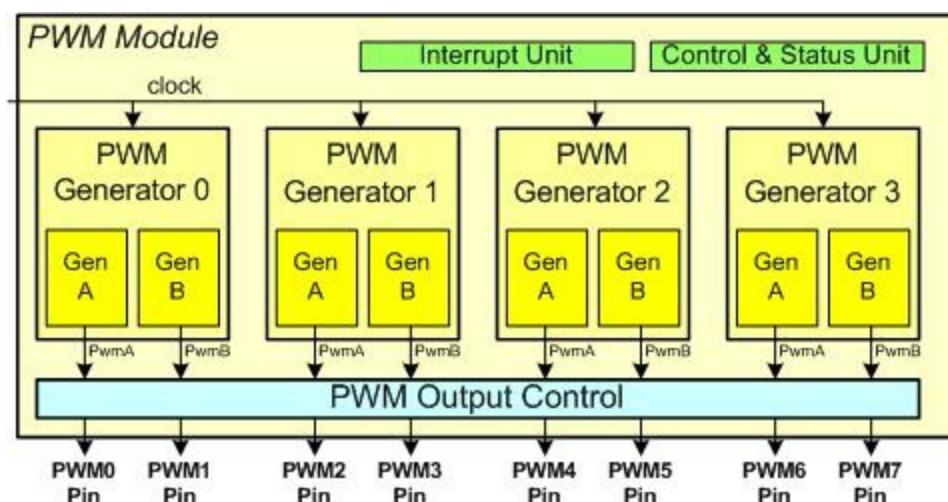


Figure 11-10: a PWM Module

In this section, we examine the PWM features and show how to program them.

### PWM Clock source

The Clock source to the PWM module in TI Tiva LaunchPad is enabled via D1 or D0 bits of RCGCPWM register. The RCGCPWM register is part of the System Control registers and located at the physical address of x400FE000 with offset 0x640. See Figure 11-11.

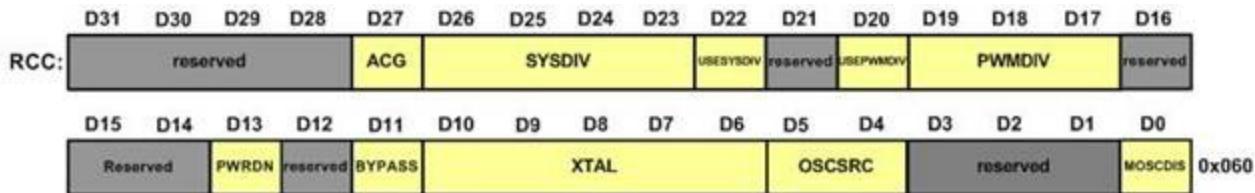


Figure 11-11: RCGCPWM Register

We enable the clock source to PWM0 module with `SYSCTL->RCGCPWM |= 1;` and to PWM1 module with `SYSCTL->RCGCPWM |= 2;`

The next register we may consider is the RCC (Run-Mode Clock Configuration) register. This register is used to pre-divide the system clock before feeding it to the PWM modules. The

clock source to the PWM module may come directly from the system clock or after going through a divider. See Figures 11-12 and 11-13.



bit	Name	Description								
19-17	PWMDIV	PWM Unit Clock Divisor: The system clock is divided by $2^{PWMDIV+1}$ when the USEPWMDIV bit is set to one.								
		PWMDIV value	0	1	2	3	4	5	6	7
20	USEPWMDIV	Division	Clk/2	Clk/4	Clk/8	Clk/16	Clk/32	Clk/64	Clk/64	Clk/64
		Enable PWM Clock Divisor (Use PWM clock Divisor) 0: The PWM clock divider is by passed, 1: The PWM clock divider is used as the PWM clock source.								

Figure 11-12: Run-Mode Clock Configuration (RCC), offset 0x060

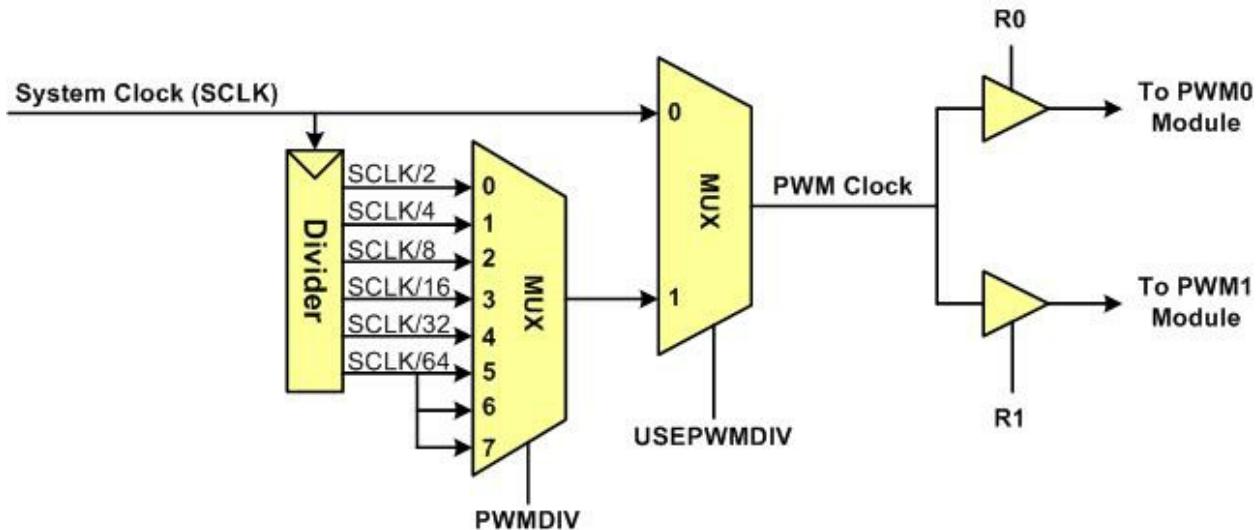


Figure 11-13: Clock Sources to PWM

The D20 bit (USEPWMDIV) of RCC register allows us to make this selection. With Bit D20 = 1, we have the choices of dividing the system clock by 2, 4, 8, 16, 32, and 64 (default) before it is fed to the PWM modules. The D19-D17 bits of RCC register determines the value of the divisor. Notice that the default is divide by 64 if the pre-divider is enabled by D20 bit. See Example 11-2.

### Example 11-2

Find the value of bit 20-17 of the RCC register for PWM Module clock frequencies of (a) 8MHz, (b) 2MHz, (b) 1MHz, and (c) 250KHz. Assume the system clock frequency is 16MHz.

**Solution:**

With D20 = 1, we have the following:

- (a) D19-D17 = 000 gives us  $16\text{MHz}/2=8\text{MHz}$  for PWM Module Frequency.
- (b) D19-D17 = 010 gives us  $16\text{MHz}/8=2\text{MHz}$  for PWM Module Frequency.
- (c) D19-D17 = 011 gives us  $16\text{MHz}/16=1\text{MHz}$  for PWM Module Frequency.
- (d) D19-D17 = 111 gives us  $16\text{MHz}/64=250\text{KHz}$  for PWM Module Frequency (default).

## Enabling PWMx Generator (Counter)

In TI Tiva LaunchPad, we have two PWM modules PWM0 and PWM1. Each PWM module has four Generators (Counters) as shown in Figure 11-10. Figure 11-14 shows the simplified structure of a Generator.

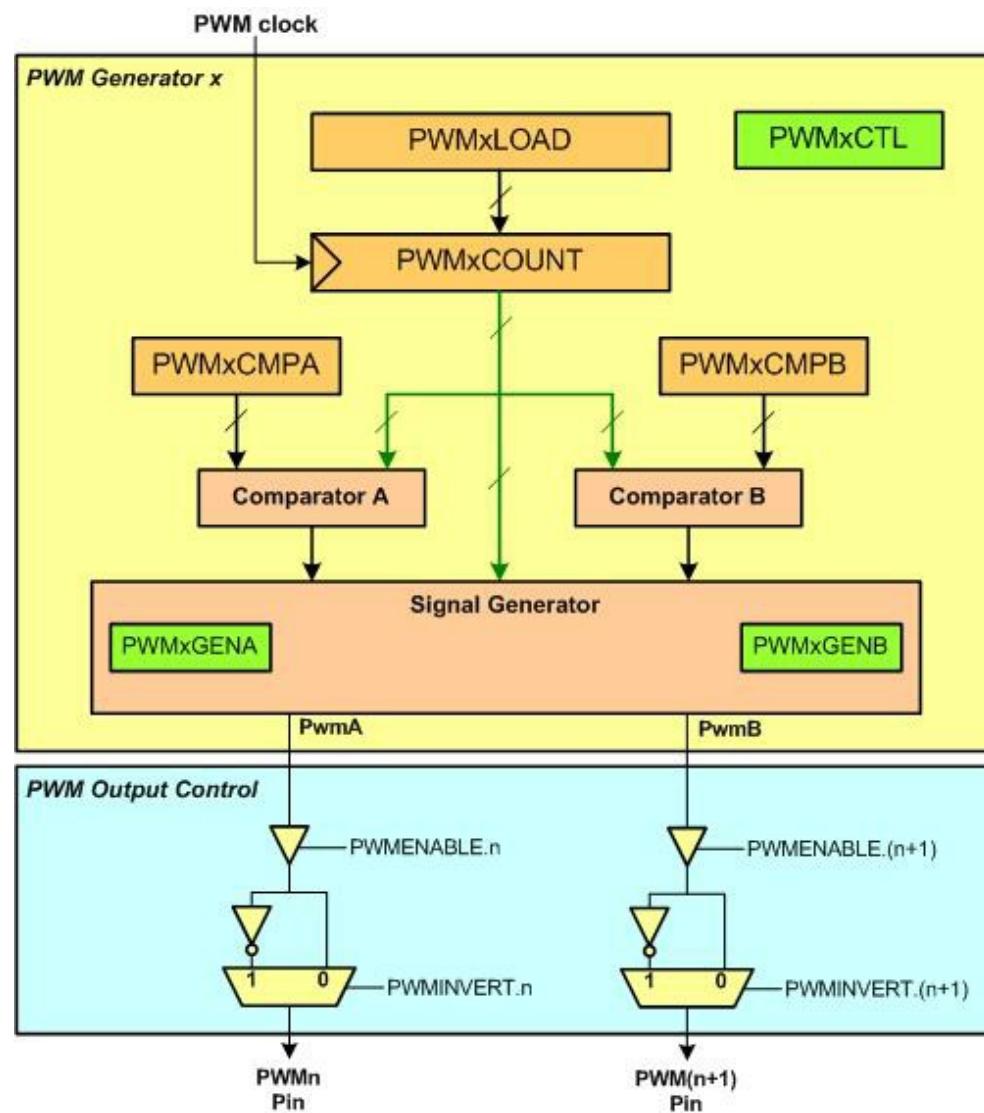


Figure 11-14: A PWM Generator (Counter) and PWMA and PWMB Output

We use PWMxCTL (PWMx Control) register to enable the Generator (Counter). Notice that we use the terms Counter and Generator interchangeably since there is one Counter per Generator and all the programming of the Generator surrounds the Counter. Also notice that we have a PWMxCTL for each Generator. See Figure 11-15.

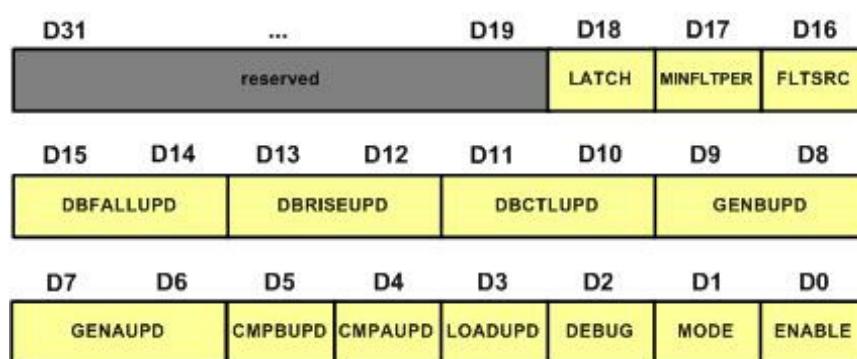
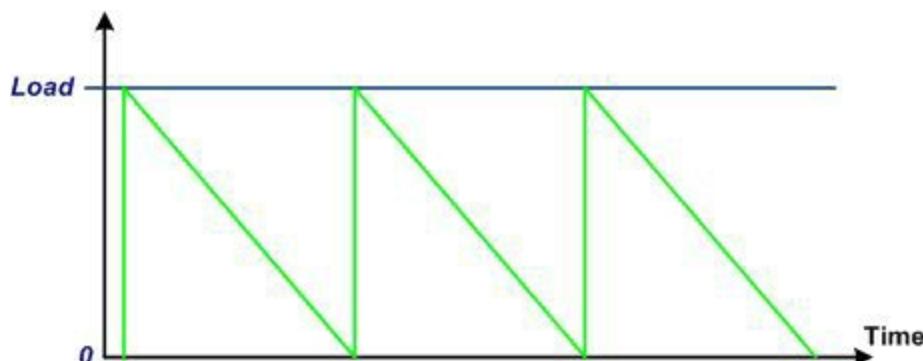


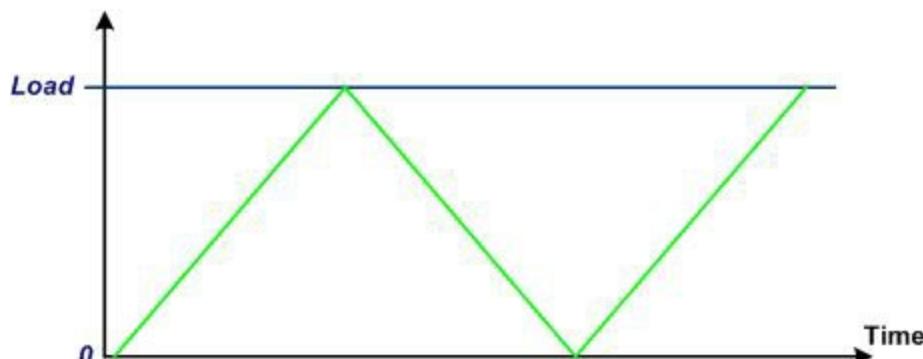
Figure 11-15: PWMxCTL Register

The most important bit of the PWMxCTL is D0 (Enable). This bit is used to start or stop the Counter. The next important bit is D1 (MODE). The counter has two modes:

- 1) **Count Down:** counts down from the load value until it reaches 0. Upon reaching zero, the load value is placed in the counter and the countdown starts again. This is the default option.
- 2) **Count Up-Down:** counts up from 0 until it reaches the load value. After reaching the load value, it turns around and counts down to 0 and upon reaching 0 it repeats the process. See Figure 11-16.



(a) Down counting



(b) Up-Down counting

Figure 11-16: Up/down and down-counter

In both modes, the load value is in a register called LOAD (PWMxLOAD) register. Notice that we must disable a given Generator (Counter) before the initialization. After the initialization is done, we must enable it to start counting.

Table 11-3 shows the offset address for Control register of PWM0 Module. We also have four Control registers for PWM1 Module. The offset addresses are the same except the Base address for the PWM1 module is different, as we mentioned earlier.

Register	Offset
PWM0_0_CTL (Control for Counter0)	0x040
PWM0_1_CTL (Control for Counter1)	0x080
PWM0_2_CTL (Control for Counter2)	0x0C0
PWM0_3_CTL (Control for Counter3)	0x100

Table 11-3: The Offset Addresses of PWMCTL Registers in TI TIVA

## The LOAD register

To set the maximum value for the Counter, we have to load register called PWMxLOAD. Since each of PWM0 and PWM1 has four Generators (Counters), we also have four PWMxLOAD registers. The PWMxLOAD register is 16-bit wide and it determines the maximum count and hence the PWM output frequency.

Figure below shows the LOAD registers of PWM0 Module. We also have four LOAD registers for PWM1 Module. The offset addresses are the same except the base address for the PWM1 module is different, as we mentioned earlier.

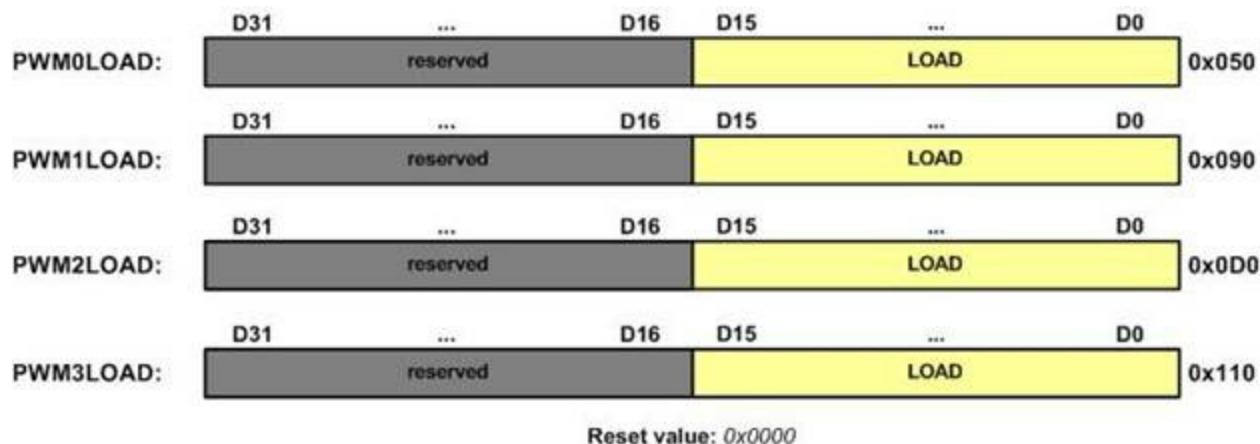


Figure 11-17: PWMxLOAD Register

## Reading the Current Counter Value

The current value of the Counter (Generator) can be read from the PWMxCOUNT register. As the Counter (Generator) counts up or counts down, we can read the value of this register at any time. Notice, this is a Read-Only (RO) register. This is similar to free-running counters we see in other microcontrollers. Since the Counter is 16-bit, the PWMxCOUNT gives us the current value of the counter which can be between 0 to 0xFFFF. We can set a maximum value for the Counter using the LOAD register. Figure 11-18 shows the Current Count registers of PWM0

## Module.

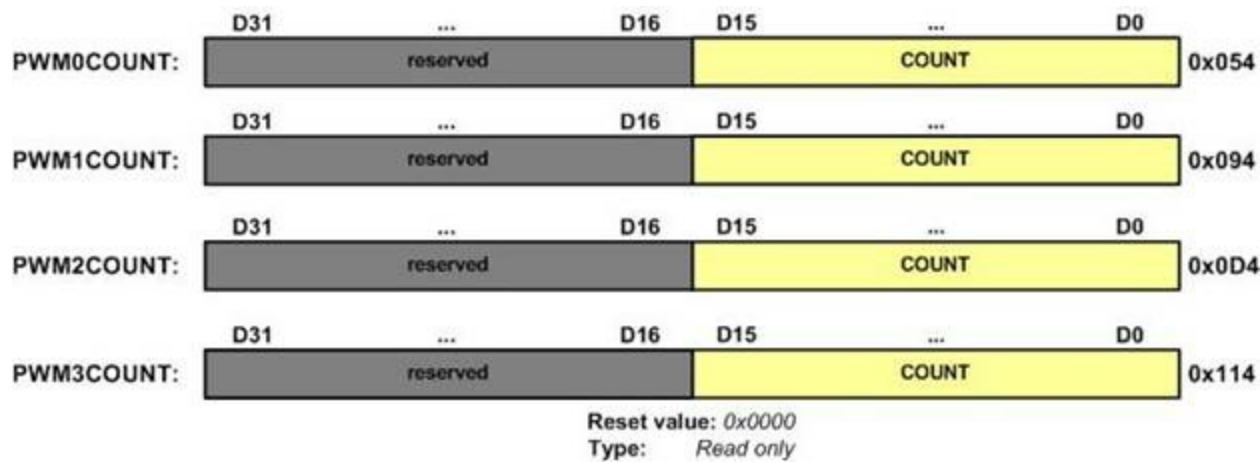


Figure 11-18: PWMxCOUNT (Current Counter)

We also have four Current Count registers for PWM1 Module. The offset addresses are the same except the Base address for the PWM1 module is different, as we mentioned earlier. Program 11-1 shows the current count on LEDs of PORTF.

**Program 11-1. Write a program to set the PWM Module to the lowest frequency assuming System-Clock=16MHz. Then enable the Counter0 (Generator0) of PWM Module 0 and read the Current Counter value and dump it on LEDs of PORTF. Place the upper 3 bits on LEDs of PORTF.**

```
/* p11_1.c: Reading the PWM Counter */

/* This program sets up the counter in PWM1 Generator 3 for down counter and reload
with maximum value. The counter value is continuously read and the three most
significant bits are written to the LEDs. */

#include "TM4C123GH6PM.h"

int main(void)
{
    int x;

    /* Enable Peripheral Clocks */
    SYSCTL->RCGCPWM |= 2;      /* enable clock to PWM1 */
    SYSCTL->RCGCGPIO |= 0x20;   /* enable clock to PORTF */
    SYSCTL->RCC |= 0x00100000;  /* use pre-divide for PWM clock */
    SYSCTL->RCC |= 0x000E0000;  /* use 64 for pre-divide for PWM */

    /* Enable PORTF3-1 for tri-color LEDs */
    GPIOF->PCTL &= ~0x0000FFF0; /* make PORTF2 PWM output pin */
    GPIOF->DIR |= 0x0E;          /* pins output */
    GPIOF->DEN |= 0x0E;          /* pins digital */

    /* set up counter in PWM1 Generator 3 */
    PWM1->_3_CTL = 0;           /* stop counter */
    PWM1->_3_LOAD = 0X0000FFFF; /* set max load value */
    PWM1->_3_CTL = 1;           /* start counter */

    for(;;)
```

```

    {
        x = PWM1->_3_COUNT;      /* read counter value */
        x = x >> 12;           /* shift it right 12 bits */
        GPIOF->DATA = x;        /* write it to the LEDs */
    }

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## Compare A and Compare B

Figure 11-19 shows the offset address for CompareA registers of PWM0 Module. We also have four CompareA registers for PWM1 Module. The offset addresses are the same except the Base address for the PWM1 module is different, as we mentioned earlier.

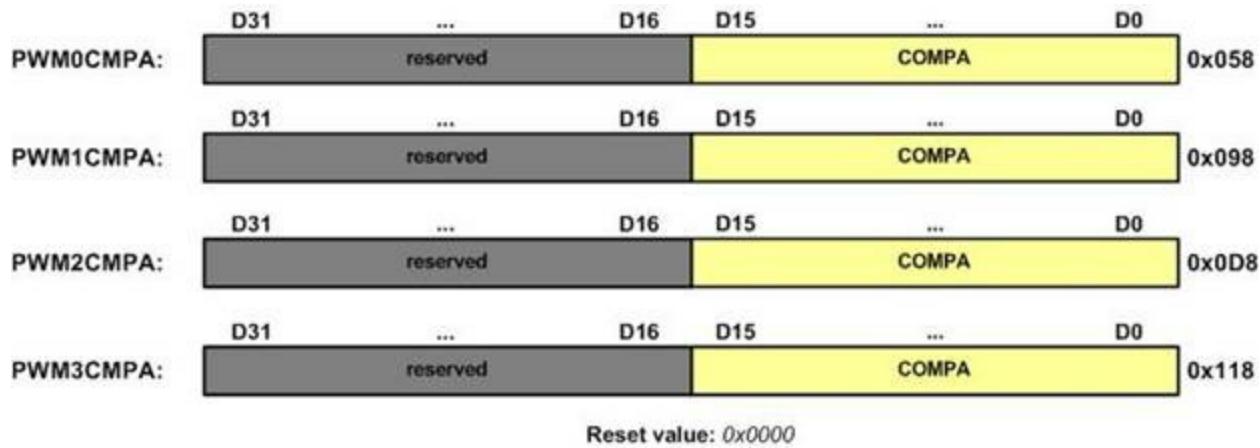


Figure 11-19: PWMyCMPA

Figure 11-20 shows the offset address for CompareB registers of PWM0 Module. We also have four CompareB registers for PWM1 Module. The offset addresses are the same except the Base address for the PWM1 module is different, as we mentioned earlier.

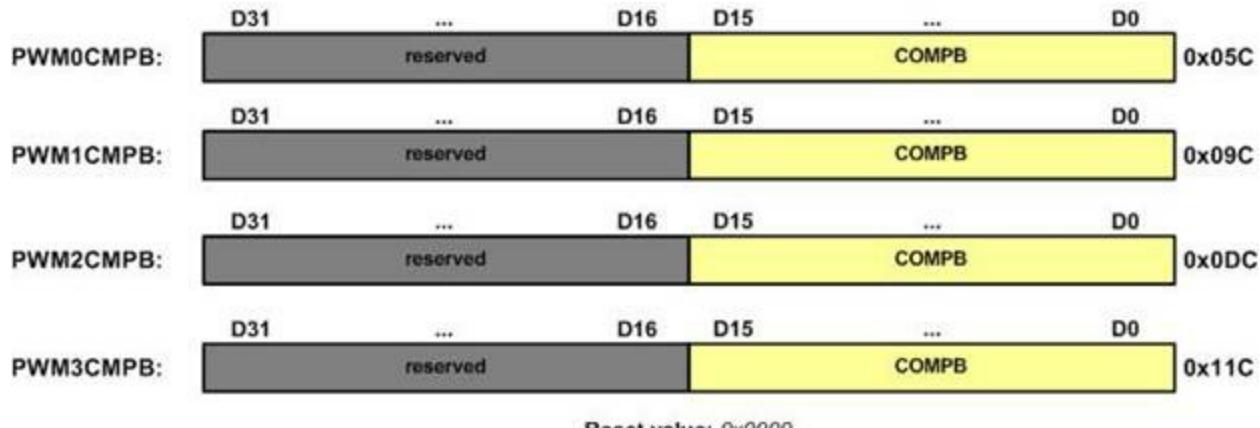


Figure 11-20: PWMyCMPB

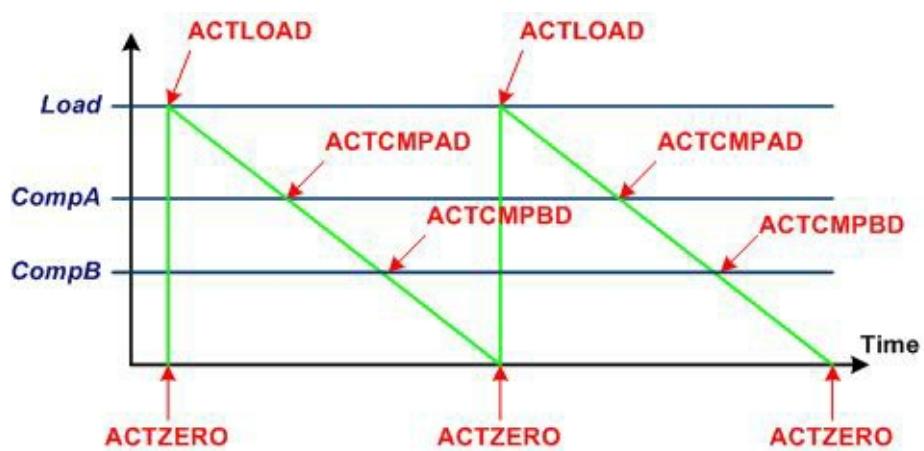
## Generating Square waves using PWM generators

In TI Tiva LaunchPad, each of the Generator has two Compare registers. They are called PWMxCMPA (PWMx Compare A) and PWMxCMPB (PWMx Compare B). As the Counter counts down (or up), its value is compared with the PWMxCMPx register and upon a match, a PWM output pin will do one of the following:

- 1) do nothing,
- 2) toggle,
- 3) driven HIGH,
- 4) driven LOW

These output actions are not limited to the compare register, you may choose one of these actions when the counter reaches zero and when the counter reloads. The selections of these actions are made in the PWM generator (PWMxGENx) register. Each register is associated with an output pin and has six actions you may specify:

- 1) action when the counter matches comparator B while counting down.
- 2) action when the counter matches comparator B while counting up.
- 3) action when the counter matches comparator A while counting down.
- 4) action when the counter matches comparator A while counting up.
- 5) action when the counter is reloaded.
- 6) action when the counter reaches zero.



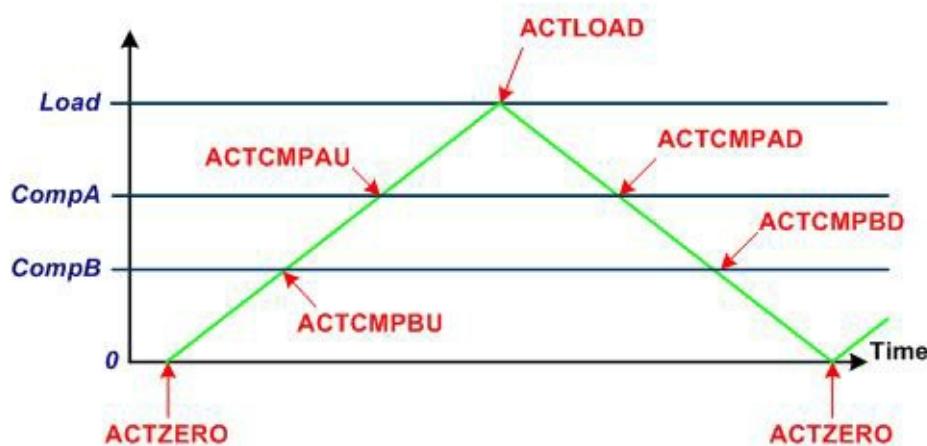
ACTLOAD: Action when the counter is reloaded

ACTZERO: Action when the counter reaches zero

ACTCMPAD: Action when the counter matches comparator A while counting down

ACTCMPBD: Action when the counter matches comparator B while counting down

#### (a) Down counting



ACTCMBD: Action when the counter matches comparator B while counting down

ACTCMBU: Action when the counter matches comparator B while counting up

ACTCMPAU: Action when the counter matches comparator A while counting up

ACTCMBU: Action when the counter matches comparator B while counting up

ACTLOAD: Action when the counter is reloaded

ACTZERO: Action when the counter reaches zero

#### (b) Up-Down counting

Figure 11-21: PWM Events in TI TIVA

These options allow us to generate some elaborate output waveforms. Each wave generator has 2 outputs: PwmA and PwmB; Using PWMxGENA and PWMxGENB registers the action of the outputs are chosen as was shown in Figure 11-14. The details of PWMxGENx register is shown in Figure 11-22. Each event may cause an action specified by two bits.

	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
PWMxGENA:		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU	ACTLOAD		ACTZERO					
	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
PWMxGENB:		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU	ACTLOAD		ACTZERO					

**ACTCMPBD:** Action when the counter matches comparator B while counting down

**ACTCMPBU:** Action when the counter matches comparator B while counting up

**ACTCMPAD:** Action when the counter matches comparator A while counting down

**ACTCMPAU:** Action when the counter matches comparator A while counting up

**ACTLOAD:** Action when the counter is reloaded

**ACTZERO:** Action when the counter reaches zero

For each of the above events the following actions can be chosen:

Value	Description
0x00	Do nothing
0x01	Invert pwmA
0x02	Drive pwmA Low
0x03	Drive pwmA High

Figure 11-22: PWMxGENA and PWMxGENB registers

Figure 11-23 shows the offset address for GeneratorA Control registers of PWM0 Module. We also have four GeneratorA Control registers for PWM1 Module. The offset addresses are the same except the Base address for the PWM1 module is different, as we mentioned earlier.

	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
PWM0GENA:		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU	ACTLOAD		ACTZERO		0x060			
	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
PWM1GENA:		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU	ACTLOAD		ACTZERO		0x0A0			
	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
PWM2GENA:		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU	ACTLOAD		ACTZERO		0xE0			
	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
PWM3GENA:		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU	ACTLOAD		ACTZERO		0x120			

Figure 11-23: PWMxGENA

Figure 11-24 shows the offset address for GeneratorB Control registers of PWM0 Module. We also have four GeneratorB Control registers for PWM1 Module. The offset addresses are the same except the Base address for the PWM1 module is different, as we mentioned earlier.

PWM0GENB:	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU		ACTLOAD		ACTZERO					0x064
PWM1GENB:	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU		ACTLOAD		ACTZERO					0x0A4
PWM2GENB:	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU		ACTLOAD		ACTZERO					0x0E4
PWM3GENB:	D31	.....	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
		Reserved		ACTCMPBD	ACTCMPBU	ACTCMPAD	ACTCMPAU		ACTLOAD		ACTZERO					0x124

Figure 11-24: PWMxGENB

## Generating periodic square wave using down counting mode

For now, we will start with a simple periodic square wave. To do that, a PWMxGENx register is configured so that the output is driven high when the counter is reloaded and the output is driven low when the counter matches a comparator register while counting down. See Figure 11-25.

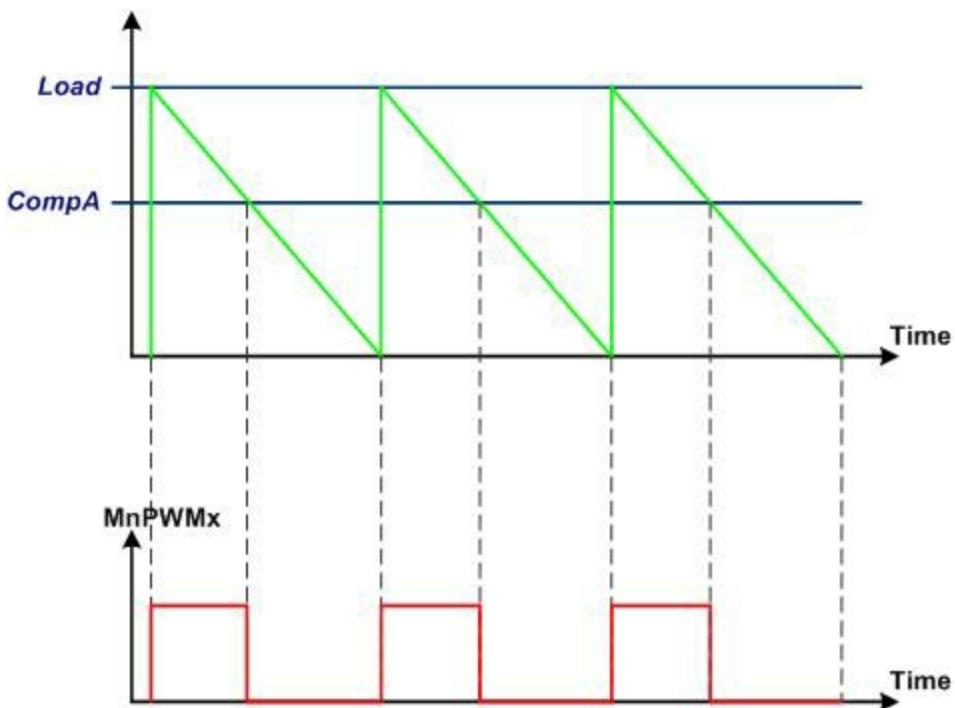


Figure 11-25: PWM generation using count-down mode and ComparatorA

## The PWM Output Frequency in Count Down mode

To calculate the value for the PWMxLOAD register for a desired PWM output frequency, we divide the period of the desired PWM output by the period of the PWM module clock. Examples 11-3 and 11-4 show how to calculate the LOAD register value.

Assume the PWM1 Module clock frequency is 8MHz. Find the value of the PWM3LOAD register if we want the PWM3 output Frequency of (a) 5KHz, (b) 10KHz, and (c) 25KHz.

### Solution:

The clock period for PWM3 Module is  $1/8\text{MHz} = 0.125\mu\text{s}$ (micro second).

(a) The PWM3 output period is  $1/5\text{KHz} = 200\mu\text{s}$ . Now,  $\text{PWM3LOAD} = 200\mu\text{s}/0.125\mu\text{s} = 1600$  or  $0x640$ .

(b) The PWM3 output period is  $1/10\text{KHz} = 100\mu\text{s}$ . Now,  $\text{PWM3LOAD} = 100\mu\text{s}/0.125\mu\text{s} = 800$  or  $0x320$ .

(c) The PWM3 output period is  $1/25\text{KHz} = 40\mu\text{s}$ .  $\text{PWM3LOAD} = 40\mu\text{s}/0.125\mu\text{s} = 320$  or  $0x140$ .

---

### Example 11-4

In a given PWM application, we need the PWM output frequency of 60Hz. Using the PWM Module frequencies of Example 11-3, find out the value of the PWM $x$ LOAD register.

### Solution:

The period for the PWM output is  $1/60\text{Hz} = 16.6\text{ms}$

In Example 11-3, we have the following cases:

(a) The PWM Module clock period is  $1/8\text{MHz} = 0.125\mu\text{s}$ (micro second).

$\text{PWM}x\text{LOAD} = 16.6\text{ms}/0.125\mu\text{s} = 132,800$ . This is not acceptable since it is larger than 65535, the maximum value the  $\text{PWM}x\text{LOAD}$  register can hold.

(b) The PWM Module clock period is  $1/2\text{MHz} = 0.5\mu\text{s}$ .  $\text{PWM}x\text{LOAD} = 16.6\text{ms}/0.5\mu\text{s} = 33,333$ .

(c) The PWM Module clock period is  $1/1\text{MHz} = 1\mu\text{s}$ .  $\text{PWM}x\text{LOAD} = 16.6\text{ms}/1\mu\text{s} = 16,600$ .

(d) The PWM Module clock period is  $1/250\text{KHz} = 4\mu\text{s}$ .  $\text{PWM}x\text{LOAD} = 16.6\text{ms}/4\mu\text{s} = 4,150$ .

---

### The PWM output duty cycle in Count Down mode

In this configuration, the duty cycle (the percentage of the time the output is high) is determined by the ratio between  $\text{PWM}x\text{CMP}x$  and  $\text{PWM}x\text{LOAD}$  registers and is shown below:

$$\text{PWM}x\text{CMP}x = (100\% - \text{Duty Cycle}\%) \times \text{PWM}x\text{LOAD}$$

See Example 11-5.

### Example 11-5

Assume the PWM0 Module System clock frequency is 16MHz. Find the value of the  $\text{PWM}x\text{LOAD}$  and  $\text{PWM}x\text{CMP}A$  registers for the following PWM output frequencies and duty cycles:

(a) 1KHz with 25%, (b) 5KHz with 60%, (c) 20KHz with 80%, and (d) 2KHz of 50%.

### Solution:

The System Clock period for PWM0 Module is  $1/16\text{MHz} = 62.5\text{ns}$  (nano sec).

- (a) The PWM output period is  $1 / 1\text{KHz} = 1\text{msec}$ . Now,  $\text{PWM0LOAD} = 1\text{ms} / 62.5\text{ns} = 16000$   
 $\text{PWMxCMPA} = (100\% - \text{Duty Cycle}) \times \text{PWMxLOAD} = (100\% - 25\%) \times 16000 = 75\% \times 16000 = 12000$
- (b) The PWM output period is  $1/5\text{KHz} = 0.2\text{msec}$ . Now,  $\text{PWM0LOAD} = 2\text{ms} / 62.5\text{ns} = 3200$   
 $\text{PWMxCMPA} = (100\% - \text{Duty Cycle}) \times \text{PWMxLOAD} = (100\% - 60\%) \times 3200 = 40\% \times 3200 = 1280$
- (c) The PWM output period is  $1/20\text{KHz} = 0.05\text{msec}$ . Now,  $\text{PWM0LOAD} = 0.05\text{ms} / 62.5\text{ns} = 800$   
 $\text{PWMxCMPA} = (100\% - \text{Duty Cycle}) \times \text{PWMxLOAD} = (100\% - 80\%) \times 800 = 20\% \times 800 = 160$
- (d) The PWM output period is  $1/2\text{KHz} = 0.5\text{msec}$ . Now,  $\text{PWM0LOAD} = 0.5\text{ms} / 62.5\text{ns} = 8000$   
 $\text{PWMxCMPA} = (100\% - \text{Duty Cycle}) \times \text{PWMxLOAD} = (100\% - 50\%) \times 8000 = 50\% \times 8000 = 4000$
- 

## Enable the PWM output to the output pin

To provide an easy way to turn off the PWM output, each PWM generator has an enable register PWMENABLE. Since many of the PWM registers are shared, this allows us to disable a given PWM signal from going to the output pin without disturbing the rest of the PWMs. The lower 8 bits of the PWMENABLE register are used for enabling or disabling the PWM1 to PWM7. When the bit is enabled, the signal generated by the PWM generator is connected to the output pin. When the bit is disabled, there will be no output but the PWM generator still runs without any change.

There are total of 8 PWM pins supported by each PWM module. The PWM pins for each of PWM module are designated as PWM0, PWM1, PWM2, ..., PWM7 as shown in Figure 11-26.

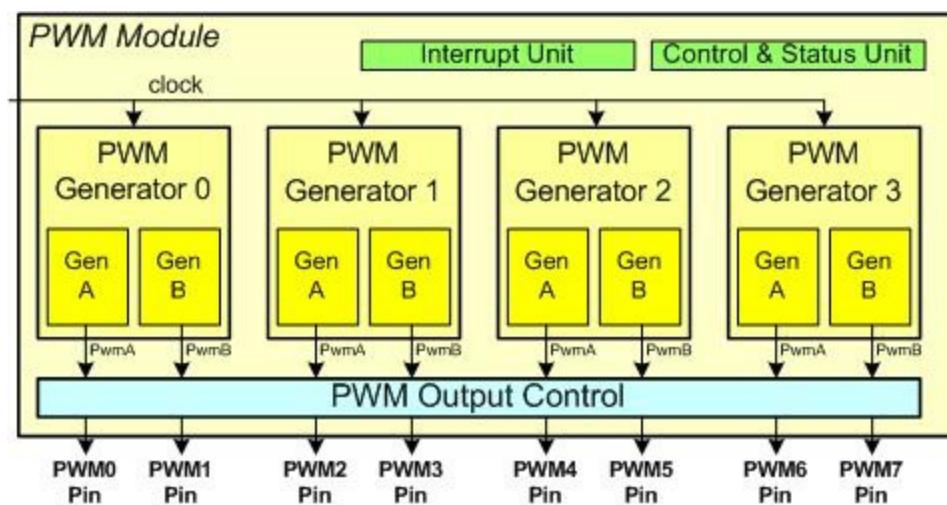


Figure 11-26: a PWM Module (repeated)

We use PWM0->ENABLE and PWM1->ENABLE to control the outputs of PWM0 and PWM1, respectively. The offset addresses are the same except the base address is different, as we mentioned earlier. See Figure 11-27.

PWMENABLE:	D31	...	D19	D7	D6	D5	D4	D3	D2	D1	D0	
	reserved			PWM7EN	PWM6EN	PWM5EN	PWM4EN	PWM3EN	PWM2EN	PWM1EN	PWM0EN	0x008

bit	Name	Description
0	PWM0EN	MnPWM0 Output Enable 0: The MnPWM0 signal has a zero value, 1: The generated pwm3B signal is passed to the MnPWM0 pin.
	PWM1EN	MnPWM1 Output Enable 0: The MnPWMx signal has a zero value, 1: The generated pwm3B signal is passed to the MnPWM1 pin.
	PWMxEN	MnPWMx Output Enable 0: The MnPWMx signal has a zero value, 1: The generated pwm3B signal is passed to the MnPWMx pin.

Figure 11-27: PWMENABLE

The role of the PWMENABLE register is shown in the *PWM Output Control* part of Figure 11-14.

### Selecting alternate function for PWMx pin

Upon reset, the GPIOAFSEL register has all 0s meaning the I/O pins are used as simple I/O. To use the alternate function, we first must set to 1 the bit in the AFSEL register for that pin. For example, for the PB6, we need to write 0x40 (0100 0000 in binary) to GPIO\_PORTB\_AFSL register. See Tables 11-4 and 11-6. After that, the GPIOPCTL register must be configured for the desired function, as shown in Tables 11-5 and 11-7. The alternate functions for each I/O pin are listed in Table 23-5 of TI Tiva LaunchPad manual. Tables 11-4 through 11-7 provide the summary for the MOPWMx and M1PWMx pins. See Example 11-6.

MOPWMx	PIN	GPIOAFSL register
<b>MOPWM0</b>	PB6	GPIOB->AFSEL=0x40 (0100 0000 binary)
<b>MOPWM1</b>	PB7	GPIOB->AFSEL =0x80 (1000 0000 binary)
<b>MOPWM2</b>	PB4	GPIOB->AFSEL =0x10 (0001 0000 binary)
<b>MOPWM3</b>	PB5	GPIOB->AFSEL =0x20 (0010 0000 binary)
<b>MOPWM4</b>	PE4	GPIOE->AFSEL =0x10 (0001 0000 binary)
<b>MOPWM5</b>	PE5	GPIOE->AFSEL =0x20 (0010 0000 binary)
<b>MOPWM6</b>	PC4	GPIOC->AFSEL = 0x10 (0001 0000 binary)
	PD0	GPIOD->AFSEL = 0x01 (0000 0001 binary)
<b>MOPWM7</b>	PC5	GPIOC->AFSEL =0x020 (0010 0000 binary)
	PD1	GPIOD->AFSEL = 0x02 (0000 0010 binary)

Table 11-4: Module 0 (M0) PWM alternate pin assignment

MOPWMx	PIN	Value for GPIOPCTL register
<b>MOPWM0</b>	PB6	GPIO_PORTB_PCTL=0x0400000
<b>MOPWM1</b>	PB7	GPIO_PORTB_PCTL=0x4000000
<b>MOPWM2</b>	PB4	GPIO_PORTB_PCTL=0x0004000
<b>MOPWM3</b>	PB5	GPIO_PORTB_PCTL=0x00400000
<b>MOPWM4</b>	PE4	GPIO PORTE_PCTL=0x00400000
<b>MOPWM5</b>	PE5	GPIO PORTE_PCTL=0x00400000
<b>MOPWM6</b>	PC4	GPIO PORTC_PCTL=00040000
	PD0	GPIO PORTD_PCTL=0x00000004

<b>M0PWM7</b>	PC5	GPIO_PORTC_PCTL=0x00400000
	PD1	GPIO_PORTD_PCTL=0x00000040

Table 11-5: Module 0 (M0) PWM pin assignment using GPIO\_PCTL (Extracted from Table 23-5 of Tiva Manual)

M1PWMx	PIN	GPIO_AFSL register
<b>M1PWM0</b>	PDO	GPIOD->AFSEL =0x01 (0000 0001 binary)
<b>M1PWM1</b>	PD1	GPIOD->AFSEL =0x02 (0000 0010 binary)
<b>M1PWM2</b>	PA6 or	GPIOA->AFSEL =0x40 (0100 0000 binary)
	PE4	GPIOE->AFSEL =0x10 (0001 0000 binary)
<b>M1PWM3</b>	PA7	GPIOA->AFSEL =0x80 (1000 0000 binary)
	PE5	GPIOE->AFSEL =0x20 (0010 0000 binary)
<b>M1PWM4</b>	PF0	GPIOF->AFSEL =0x01 (0000 0001 binary)
<b>M1PWM5</b>	PF1	GPIOF->AFSEL =0x02 (0000 0010 binary)
<b>M1PWM6</b>	PF2	GPIOF->AFSEL =0x04 (0000 0100 binary)
<b>M1PWM7</b>	PF3	GPIOF->AFSEL =0x08 (0000 1000 binary)

Table 11-6: Module 1 (M1) PWM Alternate pin assignment

M1PWMx	PIN	Value for GPIOPCTL
<b>M1PWM0</b>	PDO	GPIOD->PCTL= 0x00000005
<b>M1PWM1</b>	PD1	GPIOD->PCTL=0x00000050
<b>M1PWM2</b>	PA6	GPIOA->PCTL=0x50000000
	PE4	GPIOE->PCTL=0x00050000
<b>M1PWM3</b>	PA7	GPIOA->PCTL=0x50000000
	PE5	GPIOE->PCTL=0x00500000
<b>M1PWM4</b>	PF0	GPIOF->PCTL=0x00000005
<b>M1PWM5</b>	PF1	GPIOF->PCTL=0x00000050
<b>M1PWM6</b>	PF2	GPIOF->PCTL=0x00000500
<b>M1PWM7</b>	PF3	GPIOF->PCTL=0x00005000

Table 11-7: Module 1 (M1) PWM pin assignment using GPIO\_PCTL register (Extracted from Table 23-5 of Tiva Manual)

## Example 11-6

Show how to select the alternative function for (a) M0PWM0, (b) M0PWM1, and (c) M1PWM2 pins.

### Solution:

(a) From Tables 11-4 and 11-5, we see M0PWM0 is the alternate function for pin PB6.

```
GPIOB->AFSEL |= 0x40; /* PB6 alternative function for M0PWM0 */
GPIOB->PCTL &= ~0xF0000000; /* clear alternate function for PB6 */
GPIOB->PCTL |= 0x04000000; /* clear alternate function of PB6 for M0PWM0 */
```

(b) From Table 11-4 and 11-5, we see M0PWM1 is the alternate function for pin PB7.

```
GPIOB->AFSEL |= 0x80; /* PB7 alternative function for M0PWM1 */
GPIOB->PCTL &= ~0xF0000000; /* clear alternate function for PB7 */
GPIOB->PCTL |= 0x04000000; /* clear alternate function of PB7 for M0PWM1 */
```

(c) From Table 11-6 and 11-7, we see M1PWM2 is the alternate function for pin PE4.

```
GPIE->AFSEL |= 0x10; /* PE4 alternative function for M1PWM2 */  
GPIOE->PCTL &= ~0x000F0000; /* clear alternate function for PE4 */  
GPIOE->PCTL |= 0x00040000; /* clear alternate function of PE4 for M1PWM2 */
```

## Configuring GPIO pin for PWM

In using PWM, we must configure the GPIO pins for PWM output. In this regard, it is same as all other peripherals. The steps are as follow:

1. Enable the clock to GPIO pin by using RCGCGPIO.
2. Set the GPIOAFSEL (GPIO alternate function) for PWM output pins.
3. Enable digital pins in the GPIODEN (GPIO Digital enable) register.
4. Assign the PWM signals to specific pins using GPIOCTL register. See Tables 11-4 through 11-7.

## Configuring PWM generator to create pulses

After the GPIO configuration, we need to take the following steps to configure the PWM:

1. Disable the generator using PWM $\times$ CTL register.
2. Configure PWM $\times$ GENA (or PWM $\times$ GENB).
3. Load the value into PWM $\times$ LOAD register to set the desired output frequency.
4. Load the value into PWM $\times$ CMPA (or PWM $\times$ CMPB) register to set the desired duty cycle.
5. Start the PWM generator using PWM $\times$ CTL.
6. Configure PWM $\times$ ENABLE register to direct the PWM $\times$  to output pin.

See the next few programming examples. Program 11-2 uses PWM1 Channel 7, which is wired to the green LED on the Tiva LaunchPad. The Load register is set to 16000 with the PWM counter running of 16 MHz system clock, the PWM output frequency will be 1 kHz. The PWMCMPA has the value of 8000 that gives 50% duty cycle. PWM3GENB is connected to channel 7. The generator is programmed to set the output at reload and clear the output when the counter value matches the PWMCMPA. When the program is running, the green LED will light up. You do need an oscilloscope on PF3 pin of the LaunchPad to observe the waveform.

### Program 11-2. Using M1PWM7 to create 1kHz frequency with 50% duty cycle on PF3 pin (green LED)

```
/* p11_2.c: Generate 1kHz 50% PWM output */  
  
/* Using M1PWM7, write a program to create 1kHz frequency with 50% duty cycle on PF3  
pin (green LED). Use System Clock of 16MHz without division. Set the options of PWMGENB  
register to set the output when reload and clear the output when PWMCMPA match. */  
  
#include "TM4C123GH6PM.h"  
  
int main(void)  
{  
    /* Enable Peripheral Clocks */  
    SYSCTL->RCGCPWM |= 2;           /* enable clock to PWM1 */
```

```

SYSCTL->RCGCGPIO |= 0x20; /* enable clock to PORTF */
SYSCTL->RCC &= ~0x00100000; /* no pre-divide for PWM clock */

/* Enable port PF3 for PWM1 M1PWM7 */
GPIOF->AFSEL = 8; /* PF3 uses alternate function */
GPIOF->PCTL &= ~0x0000F000; /* make PF3 PWM output pin */
GPIOF->PCTL |= 0x00005000;
GPIOF->DEN |= 8; /* pin digital */

PWM1->_3_CTL = 0; /* stop counter */
PWM1->_3_GENB = 0x0000008C; /* M1PWM7 output set when reload, */
/* clear when match PWMCMPA */
PWM1->_3_LOAD = 16000; /* set load value for 1kHz (16MHz/16000) */
PWM1->_3_CMPA = 8000; /* set duty cycle to 50% */
PWM1->_3_CTL = 1; /* start timer */
PWM1->ENABLE = 0x80; /* start PWM1 ch7 */

for(;;) { }

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

Program 11-3 is based on Program10-2 but in the infinite loop, the value of CMPA is decremented by 100 every 20 ms. The decreasing CMPA value lengthens the duty cycle and increase the LED intensity.

### Program 11-3. Use PWM to control LED intensity

```

/* p11_3.c: Use PWM to control LED intensity */

/* This program is based on p10_2. In the infinite loop, the value of CMPA register is
decremented by 100 every 20 ms. The decreasing CMPA value causes the duty cycle to
lengthen. */

#include "TM4C123GH6PM.h"

int main(void)
{
    void delayMs(int n);
    int x = 15999;

    /* Enable Peripheral Clocks */
    SYSCTL->RCGCPWM |= 2; /* enable clock to PWM1 */
    SYSCTL->RCGCGPIO |= 0x20; /* enable clock to PORTF */
    SYSCTL->RCC &= ~0x00100000; /* no pre-divide for PWM clock */

    /* Enable port PF3 for PWM1 M1PWM7 */
    GPIOF->AFSEL = 8; /* PF3 uses alternate function */
    GPIOF->PCTL &= ~0x0000F000; /* make PF3 PWM output pin */

```

```

GPIOF->PCTL |= 0x00005000;
GPIOF->DEN |= 8; /* pin digital */

PWM1->_3_CTL = 0; /* stop counter */
PWM1->_3_GENB = 0x0000008C; /* M1PWM7 output set when reload, */
/* clear when match PWMCMPA */
PWM1->_3_LOAD = 16000; /* set load value for 1kHz (16MHz/16000) */
PWM1->_3_CMPA = 15999; /* set duty cycle to min */
PWM1->_3_CTL = 1; /* start timer */
PWM1->ENABLE = 0x80; /* start PWM1 ch7 */

for(;;)
{
    x = x - 100;
    if (x <= 0) x = 16000;
    PWM1->_3_CMPA = x;
    delayMs(20);
}
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

Program 11-4 uses the PWM clock pre-divide of SYSCTRLCC register to divide the system clock by 64 before using it for PWM generators. The LOAD register is set to the maximal value of 0xFFFF, which gives about 3.7 Hz. This is slow enough that the duty cycle changes can be observed with naked eyes.

**Program 11-4. Based on Program 11-3 but slow down the PWM frequency so that the duty cycle change can be observed with naked eyes**

```

/* p11_4: Use PWM to control LED duty cycle */

/* This program is based on p11_3. Some parameters are changed to slow down the PWM
frequency so that it runs slow enough to observe the duty cycle change from the LED
with naked eyes. */

#include "TM4C123GH6PM.h"

```

```

int main(void)
{
    void delayMs(int n);
    int x = 0xFFFF;

    /* Enable Peripheral Clocks */
    SYSCTL->RCGCPWM |= 2;           /* enable clock to PWM1 */
    SYSCTL->RCGCGPIO |= 0x20;       /* enable clock to PORTF */
    SYSCTL->RCC |= 0x00100000;      /* use pre-divide for PWM clock */
    SYSCTL->RCC |= 0x000E0000;      /* set 64 for pre-divide for PWM */

    /* Enable port PF3 for PWM1 M1PWM7 */

    GPIOF->AFSEL = 8;              /* PF3 uses alternate function */
    GPIOF->PCTL &= ~0x0000F000;    /* make PF3 PWM output pin */
    GPIOF->PCTL |= 0x00005000;
    GPIOF->DEN |= 8;               /* pin digital */

    PWM1->_3_CTL = 0;             /* stop counter */
    PWM1->_3_GENB = 0x0000008C;   /* M1PWM7 output set when reload, */
                                   /* clear when match PWMCMPA */
    PWM1->_3_LOAD = 0xFFFF;        /* set load value with max value */
    PWM1->_3_CMPA = x;            /* set duty cycle to 50% */
    PWM1->_3_CTL = 1;             /* start timer */
    PWM1->ENABLE = 0x80;           /* start PWM1 ch7 */

    for(;;)
    {
        x = x - 1000;
        if (x <= 0) x = 0xFFFF;
        PWM1->_3_CMPA = x;
        delayMs(300);
    }
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
    int i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## Review Questions

1. To enable the clock to PWM modules, we use register \_\_\_\_\_.
2. If System\_Clock=16MHz, what is lowest and highest clock frequency that PWM Module can use?
3. We use \_\_\_\_\_ register to set the PWM output Period/Frequency.
4. We use \_\_\_\_\_ register to set the PWM output pulse width.
5. In TI Tiva LaunchPad, if we have only 4 Generators (Counters) per PWM module , how come we can have up to 8 PWM output pins for each PWM module?

## Answers to Review Questions

### Section 11.1

1. True
2. False
3. Because microcontroller/digital outputs lack sufficient current to drive the DC motor, we need a driver.
4. By reversing the polarity of voltages connected to the leads
5. The DC motor is stalled if the load is beyond what it can handle.
6. No-load

### Section 11.2

1. SYSCTL\_RCGC0\_R = 0x00100000.
2.  $16\text{MHz}/64 = 250\text{KHz}$  and  $16\text{MHz}$ .
3. MxLOADx
4. MxCMPx
5. Each Generator (Counter) has 2 outputs of PWMA and PWMB. That gives us  $4 \times 2 = 8$  outputs of MxPWM0 to MxPWM7.



## Chapter 12: Programming Graphics LCD

Chapter 3 used the character LCD. In this chapter, we examine the graphics LCDs and show some programming examples, although an entire book can be dedicated to graphics LCD and its programming. Section 12.1 covers some basic concepts of graphics LCDs. In Section 12.2, we give some programming examples of graphics LCD.

## Section 12.1: Graphics LCDs

The screen of Graphics LCDs are made of pixels. Using the pixels pictures and texts are made. See Figures 12-1 and 12-2.



Figure 12-1: A Picture on a Mono-color LCD

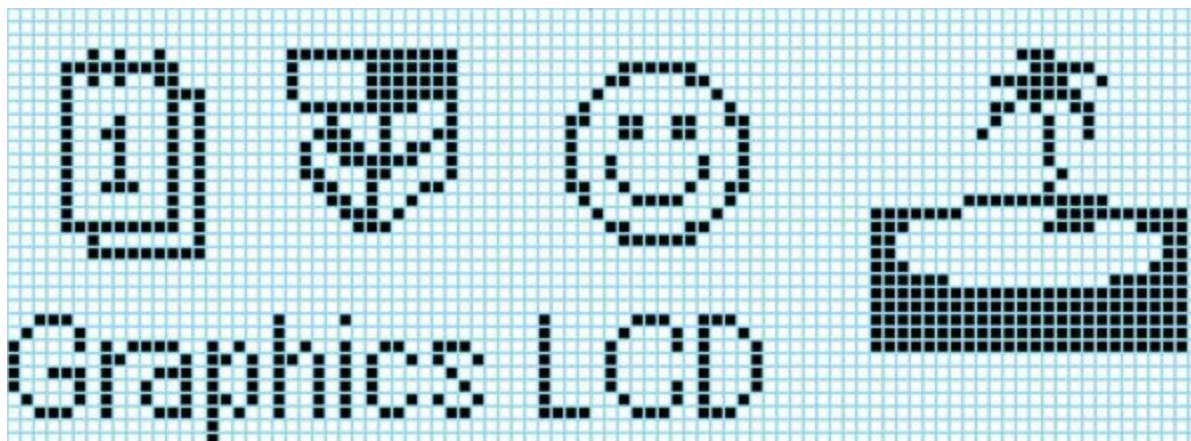


Figure 12-2: A Zoomed Picture on a Mono-color LCD

The graphics LCDs can be mono-colored or colored. In mono-colored LCDs each pixel can be on or off; in contrast in colored LCDs each pixel can have different colors. In fact the colored pixels can display red, green, and blue; using the 3 base lights they make different colors.

## Some LCD Characteristics

### Resolution

The total number of pixels (dots) per screen is a major factor in assessing an LCD and is shown below:

$$\text{Resolution} = \text{Pixels per line} \times \text{number of lines}$$

For example, when the resolution of an LCD is  $720 \times 350$ , there are 720 pixels per line and 350 lines per screen, giving a total of 252,000 pixels. The total number of pixels per screen is determined by the size of the pixel and how far apart pixels are spaced. For this reason, one must look at what is called the *dot pitch* in LCD specifications.

### Dot pitch

Dot pitch is the distance between adjacent pixels (dots) and is given in millimeters. For example, a dot pitch of 0.31 means that the distance between pixels is 0.31 mm. Consequently, the smaller the size of the pixel itself and the smaller the space between them, the higher the total number of pixels and the better the resolution. Dot pitch varies from 0.6 inch in some low-resolution LCDs to 0.2 inch in higher-resolution LCDs. Figure 12-3 shows Dot Pitch and Dot Size parameters.

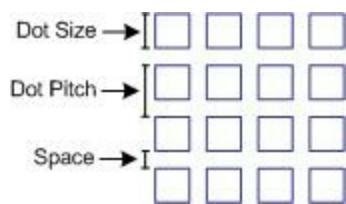


Figure 12-3: Dot Pitch and Dot Size

The specifications of a mono-colored LCD is shown in Figure 12-4.



Figure 12-4: Mechanical Specifications of a GDM12864 128x64 LCD

In some LCD specifications, it is given in terms of the number of dots per square inch, which is the same way it is given for laser printers, for example, 300 DPI (dots per inch).

### **Dot pitch and LCD size**

LCDs, like televisions, are advertised according to their diagonal size. For example, a 14-inch monitor means that its diagonal measurement is 14 inches. There is a relation between the number of horizontal and vertical pixels, the dot pitch, and the diagonal size of the image on the screen. The diagonal size of the image must always be less than the LCD's diagonal size. The following simple equation can be used to relate approximately these three factors to the diagonal measurement. It is derived from the Pythagorean theorem:

$$\begin{aligned}
 (\text{image diagonal size})^2 &= (\text{number of horizontal pixels} \times \text{dot pitch})^2 \\
 &\quad + (\text{number of vertical pixels} \times \text{dot pitch})^2
 \end{aligned}$$

Since the dot pitch is in millimeters, the size given by the equation above would be in mm, so it must be multiplied by 0.039 to get the size of the monitor in inches. See Example 12-1.

#### **Example 12-1**

A manufacturer has advertised a 14-inch monitor of  $1024 \times 768$  resolution with a dot pitch of 0.28.

Calculate the diagonal size of the image on the screen. It must be less than 14 inches.

#### **Solution:**

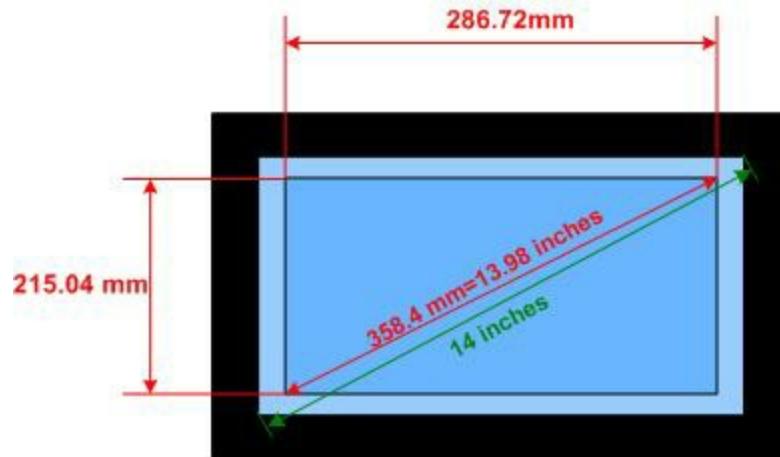
The calculation is as follows:

$(\text{image diagonal size})^2 = (\text{number of horizontal pixels} \times \text{dot pitch})^2 + (\text{number of vertical pixels} \times \text{dot pitch})^2$

$$(\text{diagonal size})^2 = (1024 \times 0.28 \text{ mm})^2 + (768 \times 0.28 \text{ mm})^2 = 358.4 \text{ mm}$$

$$\text{diagonal size (inches)} = 358.4 \text{ mm} \times 0.039 \text{ inch per mm} = 13.98 \text{ inches}$$

In the LCD the diagonal size of the image area is 13.98 inches while the diagonal size of the viewing area is 14 inches.



## Displaying on the graphics LCDs

To display a picture on the screen, a distinct color must be shown on each pixel of the LCD. To do so, there is a display memory (frame buffer) that retrieves the attributes (colors) of the entire pixels of the screen and there is an LCD controller which displays the contents of the frame buffer memory on the LCD. See Figure 12-5.

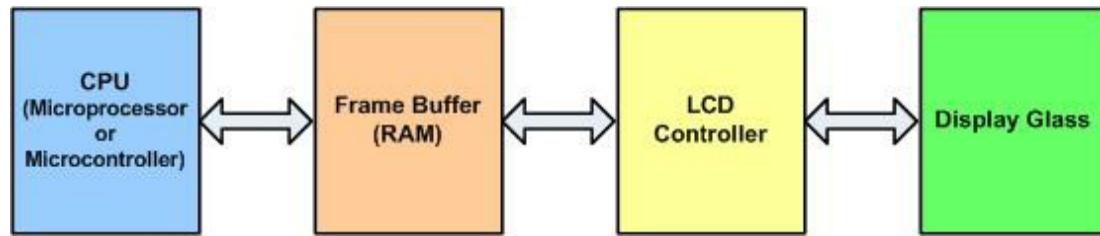


Figure 12-5: The Relationship between CPU and LCD

Graphics LCDs might come with or without frame buffer and the LCD controller. In cases that the LCD does not have frame buffer memory or controller they must be provided externally. Some new microcontrollers have the LCD controllers internally which can directly drive the LCDs. To display a picture on the screen the microcontroller writes it to the frame buffer memory.

Since the attributes (colors) of the entire pixels are stored in the frame buffer memory, the higher the number of pixels and colors options, the larger the amount of memory is needed to store them. In other words, the memory requirement goes up as the resolution and the number of colors supported go up. The number of colors displayed at one time is always  $2^n$  where n is the number of bits set aside for the color. For example, when 4 bits are assigned for

the color of the pixel, this allows 16 combinations of colors to be displayed at one time because  $2^4 = 16$ . The number of bits used for a pixel color is called color depth or bits per pixel (BPP). See Table 12-1.

BPP	Colors
1	on or off (monochrome)
2	4
4	16
8	256
16	65,536
24	16,777,216

Table 12-1: BPP (bit per pixel) vs. color

In Table 12-1, notice that in a mono color LCD a single bit is assigned for the color of the pixel and it is for "on" or "off".

## Mixing RGB (Red, Green, Blue) colors

We can get other colors by mixing the three basic colors of Red, Green, and Blue. The intensity (proportion) of the colors mixed can also effect the color we get. In many high-end graphics systems, an 8 bit value is used to represent the intensity. Its value can be between 0 to 255 (0 to 0xFF) representing high intensity (255) and zero intensity. See Table 12-2. Using three basic colors and intensity, we can make any color we want. See Figure 12-6.

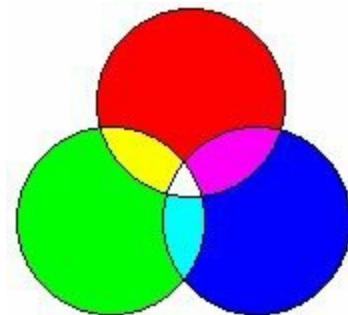


Figure 12-6: Making New Lights by Mixing the 3 Lights

I	R	G	B	Color
0	0	0	0	Black
0	0	0	1	Blue
0	0	1	0	Green
0	0	1	1	Cyan
0	1	0	0	Red
0	1	0	1	Magenta
0	1	1	0	Brown
0	1	1	1	Light Gray
1	0	0	0	Dark Gray
1	0	0	1	Light blue
1	0	1	0	Light green
1	0	1	1	Light cyan

1	1	0	0	Light red
1	1	0	1	Light Magenta
1	1	1	0	Yellow
1	1	1	1	White

Table 12-2: The 16 Possible Colors

### Example 12-2

In a certain graphics LCD, a maximum of 256 colors can be displayed at one time. How many bits are set aside for the color of the pixels?

#### Solution:

To display 256 colors at once, we must have 8 bits set for color since  $2^8 = 256$ .

### LCD Buffer memory size and color

In discussing the graphics, we need to clarify the relationship between pixel resolution, the number of colors supported, and the amount of frame buffer RAM needed to store them. There are two facts associated with every pixel on the screen:

1. The location of the pixel
2. Its attributes: color and intensity

These two facts must be stored in the frame buffer RAM. The higher the number of pixels and colors options, the larger the amount of memory that is needed to store them. In other words, the memory requirement goes up as the resolution and the number of colors supported go up. As we just mentioned, the number of colors displayed at one time is always  $2^n$  where n is the number of bits set aside for the color. For example, when 4 bits are assigned for the color of the pixel, this allows 16 combinations of colors to be displayed at one time because  $2^4 = 16$ . The commonly used graphics resolutions are 176 x 144 (QCIF), 352x288 (CIF), 320x240 (QVGA), 480x272 (WQVGA), 640x480 (VGA) and 800x480 (WVGA). Google to see the definition of the these initials.

We use the following formula to calculate the minimum frame buffer memory requirement for a graphics LCD:

$$\text{Buffer memory size (in bytes)} : \frac{\text{Horizontal Pixels} \times \text{Vertical Pixels} \times \text{color BPP}}{8}$$

Example 12-3 shows how to calculate the memory need for various resolutions and color depth.

### Example 12-3

Find the frame buffer RAM needed for (a) 176x144 with 4 BPP and (b) 640x480 resolution with

256 colors.

### Solution:

(a) For this resolution, there are a total of 25,344 pixels ( $176 \text{ columns} \times 144 \text{ rows} = 25,344$ ). With 4 bits for the color of each pixel, we need total of  $(25,344 \times 4)/8 = 16,672$  bytes of frame buffer RAM. These 4 bits give rise to 16 colors.

(b) For this resolution, there are a total of  $640 \times 480 = 307200$  pixels. With 256 colors, we need 8 bits for color of each pixel. Now, total of  $(640 \times 480 \times 8) / 8 = 307200$  bytes of frame buffer RAM needed.

---

In VGA,  $640 \times 480$  resolution with support for 256 colors displayed at one time requires a minimum of  $640 \times 480 \times 8 = 2,457,600$  bits = $307,200$  bytes of memory, but due to the memory organization used, the amount of memory used is higher.

### Storing pixels in the memory of mono-color LCDs

In mono-colored LCDs each pixel can be on or off. Therefore, 1 bit can preserve the state of 1 pixel and a byte preserves 8 adjacent pixels. In some LCDs, e.g. GDM12864A and PCD8544, pixels are stored vertically in the bytes, as shown in Figure 12-7, while in some other LCDs, e.g. T6963, the pixels are stored horizontally.

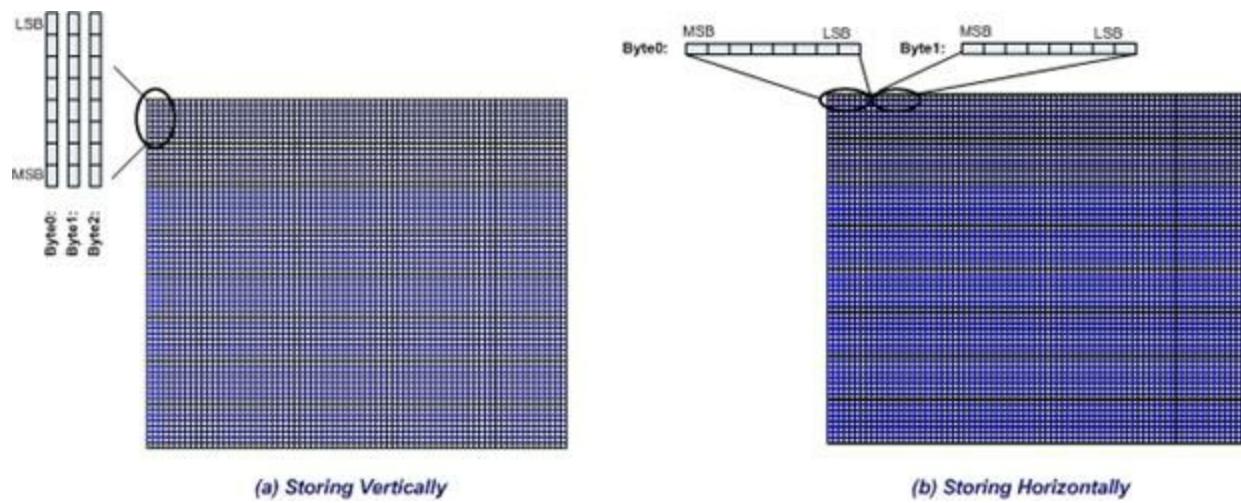


Figure 12-7: Storing Data in the LCD Memory of Mono-colored LCDs

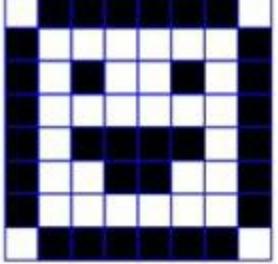
### Review Questions

1. As the number of pixels goes up, the size of display memory \_\_\_\_\_ (increases, decreases).
2. If a total of 24 bits is set aside for color, how many colors are available?
3. Calculate the total video memory needed for  $1024 \times 768$  resolution with 16 colors displayed at the same time.
4. With BPP of 16, we get \_\_\_\_\_ colors.

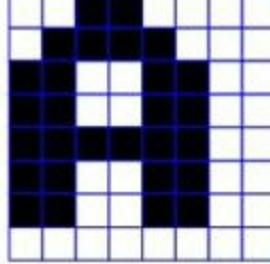
## Section 12.2: Displaying Texts on Graphics LCDs

As shown in Figure 12-8, each character can be made by putting points next to each other.

Hex	Binary
7E	01111110
81	10000001
A5	10100101
81	10000001
BD	10111101
99	10011001
81	10000001
7E	01111110



Hex	Binary
30	00110000
78	01111000
CC	11001100
CC	11001100
FC	11111100
CC	11001100
CC	11001100
00	00000000



```
unsigned char font8x8 [ ][8]={  
    0x7E,0x81,0xA5,0x81,0xBD,0x99,0x81,0x7E, //smile  
    0x30,0x78,0xCC,0xCC,0xFC,0xCC,0xCC,0x00 //A  
};
```

Figure 12-8: Pixel Patterns of Characters Happy Face and A

To display characters on the screen, we must have the pixel patterns of the entire characters. Whenever we want to display a character on the screen we copy its pixel pattern into the display memory. See Figure 12-9.

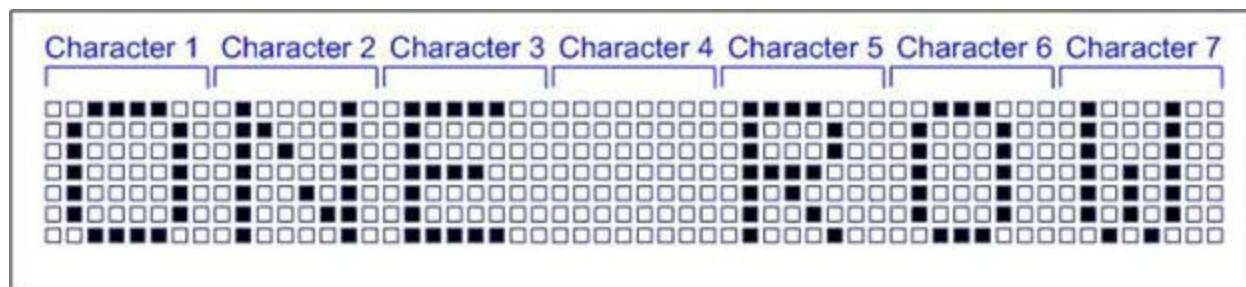
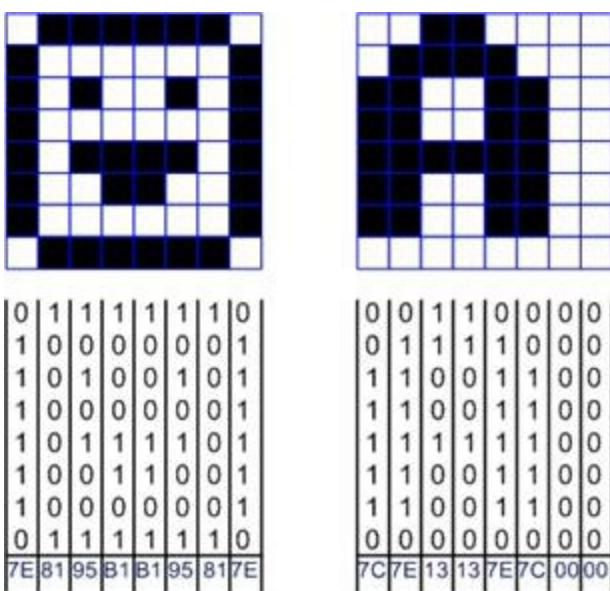


Figure 12-9: A Sample Text

The pixel patterns are stored in an array in the same way that they should be stored in the LCD memory. This means that for horizontal LCDs the bits are stored horizontally and for vertical LCDs the pixels are stored vertically. Figure 12-8 shows the way patterns are stored for horizontal LCDs. In Figure 12-10 the same patterns are stored for vertical LCDs.



```
unsigned char font8x8 [ ][8]={
0x7E,0x81,0x95,0xB1,0xB1,0x95,0x81,0x7E, //smile
0x7C,0x7E,0x13,0x13,0x7E,0x7C,0x00,0x00 //A
};
```

Figure 12-10: Pixel Patterns of Characters Happy Face and A and its Font for Vertical LCD

To get better-looking characters, the font resolution must be increased, which translates to more pixels horizontally and vertically. See Figure 12-11.

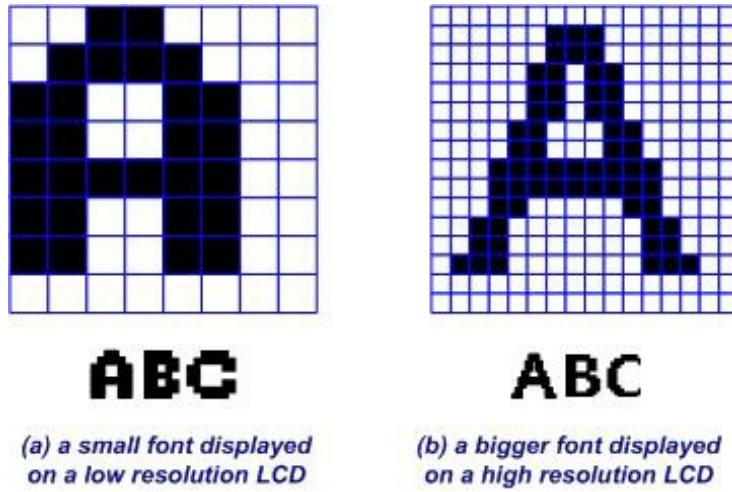


Figure 12-11: A Bigger Font vs. a Smaller Font

See Program 12-1. A lookup table of the pixel patterns of the characters are made using an array. The `Lcd_putchar` function accesses the lookup array to display characters on the LCD. The connection between the PCD8544 LCD and the microcontroller is shown in Figure 12-12. For more information about the PCD8544 see its datasheet on the Web.

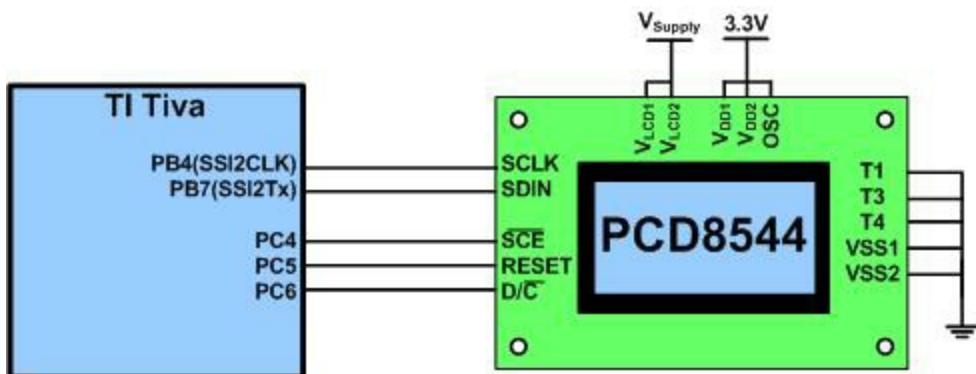


Figure 12-12: The PCD8544 LCD connection to the TI Tiva

### Program 12-1: Displaying a text on the PCD8544 GLCD

```
/*P12_1.c: Programming PCD8544 GLCD via SPI with TI TIVA ARM*/
#include "tm4c123gh6pm.h"

#define RESET 0x20 /* PC5 reset pin */
#define DC     0x40 /* PC6 register select pin */

/* define the pixel size of display */
#define GLCD_WIDTH 84
#define GLCD_HEIGHT 48

void GLCD_setCursor(uint8_t x, uint8_t y);
void GLCD_clear(void);
void GLCD_init(void);
void GLCD_data_write(uint8_t data);
void GLCD_command_write(uint8_t data);
void SSI2_init(void);
void SSI2_write(uint8_t data);
void delayMs(uint32_t n);
void delayUs(uint32_t n);

int main(void)
{
    /* sample font table */
    char font_table[][6] = {
        {0x7e, 0x11, 0x11, 0x11, 0x7e, 0}, /* A */
        {0x7f, 0x49, 0x49, 0x49, 0x36, 0}, /* B */
        {0x3e, 0x41, 0x41, 0x41, 0x22, 0}}; /* C */

    int i;

    GLCD_init();      /* initialize the GLCD controller */
    GLCD_clear();    /* clear display and home the cursor */

    /* display letter A */
    for (i = 0; i < 6; i++)
        GLCD_data_write(font_table[0][i]);

    /* display letter B */
    for (i = 0; i < 6; i++)
        GLCD_data_write(font_table[1][i]);

    /* display letter C */
    for (i = 0; i < 6; i++)
        GLCD_data_write(font_table[2][i]);

    while(1) { }

    void GLCD_setCursor(uint8_t x, uint8_t y)
    {
        GLCD_command_write(0x80 | x); /* column */
        GLCD_command_write(0x40 | y); /* bank (8 rows per bank) */
    }
}
```

```

/* clears the GLCD by writing zeros to the entire screen */
void GLCD_clear(void)
{
    int32_t index;
    for (index = 0 ; index < (GLCD_WIDTH * GLCD_HEIGHT / 8) ; index++)
        GLCD_data_write(0x00);

    GLCD_setCursor(0, 0); /*After we clear the display, return to the home position */
}

void SSI2_init(void)
{
    SYSCTL->RCGCGSSI |= 0x04; /* enable clock to SSI2 */
    SYSCTL->RCGCGPIO |= 0x02; /* enable clock to GPIOB */
    SYSCTL->RCGCGPIO |= 0x04; /* enable clock to GPIOC */

    /* PORTB 7, 5, 4 for SSI2 */
    GPIOB->AFSEL |= 0xB0; /* PORTB 7, 5, 4 for SSI2 */
    GPIOB->PCTL &= ~0xF0FF0000; /* PORTB 7, 5, 4 for SSI2 */
    GPIOB->PCTL |= 0x20220000;
    GPIOB->DEN |= 0xB0; /* PORTB 7, 5, 4 as digital pins */

    /* PORTC 5, 6, 7 for CE, DC, Reset */
    GPIOC->DATA |= 0x60; /* set PORTC 5, 6, 7 idle high */
    GPIOC->DIR |= 0x60; /* set PORTC 5, 6, 7 as output for CS */
    GPIOC->AMSEL &= ~0x60; /* disable analog */
    GPIOC->DEN |= 0x60; /* set PORTC 5, 6, 7 as digital pins */

    SSI2->CR1 = 0; /* disable SSI2 and make it master */
    SSI2->CC = 0; /* use system clock */
    SSI2->CPSR = 16; /* clock prescaler divide by 16 gets 1 MHz clock */
    SSI2->CR0 = 0x0007; /* clock rate div by 1, phase/polarity 0 0, mode freescale,
data size 8 */
    SSI2->CR1 = 2; /* enable SSI2 */
}

/* send the initialization commands to PCD8544 GLCD controller */
void GLCD_init(void)
{
    SSI2_init();

    /* hardware reset of GLCD controller */
    GPIOC->DATA &= ~RESET;
    GPIOC->DATA |= RESET;

    GLCD_command_write(0x21); /* set extended command mode */
    GLCD_command_write(0xB0); /* set LCD Vop for contrast */
    GLCD_command_write(0x04); /* set temp coefficient */
    GLCD_command_write(0x14); /* set LCD bias mode 1:48 */
    GLCD_command_write(0x20); /* set normal command mode */
    GLCD_command_write(0x0C); /* set display normal mode */
}

/* write to GLCD controller data register */
void GLCD_data_write(uint8_t data)
{
    /* select data register */
}

```

```

GPIOC->DATA |= DC;

/* send data via SSI */
SSI2_write(data);
}

/* write to GLCD controller command register */
void GLCD_command_write(uint8_t data)
{
    /* select command register */
    GPIOC->DATA &= ~DC;

    /* send data via SSI */
    SSI2_write(data);
}

void SSI2_write(uint8_t data)
{
    SSI2->DR = data;           /* write data */
    while (SSI2->SR & 0x10) ;  /* wait for transmit done */
    data = SSI2->DR;
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(uint32_t n)
{
    uint32_t i, j;
    for(i = 0; i < n; i++)
        for(j = 0; j < 3180; j++)
            {} /* do nothing for 1 ms */
}

/* delay n microseconds (16 MHz CPU clock) */
void delayUs(uint32_t n)
{
    uint32_t i, j;
    for(i = 0 ; i < n; i++)
        for(j = 0; j < 3; j++)
            {} /* do nothing for 1 ms */
}

/* This function is called by the startup assembly code to perform system specific
initialization tasks. */
void SystemInit(void)
{
    /* Grant coprocessor access */
    /* This is required since TM4C123G has a floating point coprocessor */
    SCB->CPACR |= 0x00f00000;
}

```

## Review Questions

1. True or false. The same font can be used for vertical and horizontal LCDs.
2. True or false. To display a character on the LCD, its pixel pattern should be copied onto the LCD memory.

# Answers to Review Questions

## Section 12-1:

1. Increases
2.  $2^{24} = 16.7$  million
3.  $1024 \times 768 \times 4 = 3,145,728$  bits = 384K bytes, but it uses 512 KB due to bit planes.
4.  $2^{16} = 65,536$

## Section 12-2:

1. False
2. True



# Chapter 13: DRAM Memory Technology and DMA Controller

Many ARM chips come with on-chip DRAM controllers. These ARM chips allow the connection of external DRAM memory to the CPU. As the ARM-based motherboard becomes widely available for the Microsoft Windows, Linux and Android operating systems, the issue of DRAM interfacing becomes as important as the x86-based PCs. In this chapter, we examine DRAM memory. In Section 13.1 we look at memory cycle of the CPU and introduce some concepts such as burst access and banking. In the first part of Section 13.2 we discuss various types of DRAMs, such as fast page mode and static column. Then we examine the newer and faster DRAMs of EDO and SDRAM technologies. Section 13.3 explores the issue of data integrity in DRAM and ROM. You may wish to review DRAM organization and capacity, covered in Chapter 0 ([http://www.microdigitaled.com/ARM/ARM\\_books.htm](http://www.microdigitaled.com/ARM/ARM_books.htm)). In Section 13.4 the direct memory access (DMA) concept is discussed.

## Section 13.1: Concept of Memory Cycle

When interfacing a microprocessor to memory, the first issue is how much time is provided by the CPU for one complete read or write cycle. In other words, what is the memory cycle time of the CPU? In early microprocessors, the memory cycle time consisted of 4 clocks, which leaves plenty of time to access memory. In those CPUs, with a working frequency of 10 MHz, it had a 400-ns memory cycle ( $4 \times 100 \text{ ns} = 400$ ,  $T = 1/10 \text{ MHz} = 100 \text{ ns}$ ). A memory cycle of 400 ns means that the CPU can access memory every 400 ns, and not faster. This is enough time to access even the slow and inexpensive DRAMs. However, for the newer CPUs, memory cycle time consists of only two clocks. This makes memory design a challenging task, especially when the speed of the CPU goes beyond 100 MHz.

### Memory cycle time and inserting wait states

To access an external device such as memory or I/O, the CPU provides a fixed amount of time called a *bus cycle time*. During this bus cycle time, the read and write operation of memory or I/O must be completed. Here, we cover the memory bus cycle time. For the sake of clarity we will concentrate on reading memory, but the concepts apply to write operations as well. The bus cycle time used for accessing memory is often referred to as *MC (memory cycle)* time. The time from when the CPU provides the addresses at its address pins to when the data is expected at its data pins is called *memory read cycle time*. While in older processors the memory cycle time takes 4 clocks, in the newer CPUs the memory cycle time is 2 clocks. If memory is slow and its access time does not match the MC time of the CPU, extra time can be requested from the CPU to extend the read cycle time. This extra time is called a *wait state* (WS). In the 1980s, the clock speed for memory cycle time was the same as the CPU's clock speed. For example, in the 20 MHz processors, the buses were working at the same speed of 20 MHz. This resulted in  $2 \times 50 \text{ ns} = 100 \text{ ns}$  for the memory cycle time ( $1/20 \text{ MHz} = 50 \text{ ns}$ ). When the CPU's speed was under 100 MHz, the bus speed was comparable to the CPU speed. In the 1990s the CPU speed exploded to 1 GHz (gigahertz) while the bus speed maxed out at around 133 MHz. The gap between the CPU speed and the bus speed is one of the biggest problems in the design of high-performance computers. To avoid the use of too many wait states in interfacing memory to CPU, cache memory and high-speed DRAMs were invented.

It must be noted that memory access time is not the only factor in slowing down the CPU, even though it is the largest one. The other factor is the delay associated with signals going through the data and address path. Delay associated with reading data stored in memory has the following three components:

1. The time taken for address signals to go from CPU pins to memory pins, going through memory decoding logic circuitry and address and data bus buffers.
2. The time it takes for the data to travel from memory to CPU going through any logic gates on the pathway. This is referred to as a *path delay*. The path delay is large in the motherboards and very small in the SOC (system-on-chip) since the chip-to-chip delay is eliminated.
3. The memory access time to get the data out of the memory chip. This is the largest of the three components.

The total sum of these three must equal the memory read cycle time provided by the CPU. Memory access time is the largest and takes about 90% of the read cycle time. See Examples 13-1 through 13-3 for further clarification of these points. These concepts are critical in the design of microprocessor-based products. As we have seen, wait states degrade computer performance, as shown in Example 13-3. It does not make sense to buy a high-frequency CPU, then interface it with slow memory.

### Example 13-1

Calculate the memory cycle time of a 100-MHz bus system with

- (a) 0 WS,
- (b) 1 WS, and
- (c) 2 WS.

#### Solution:

$1/100 \text{ MHz} = 10 \text{ ns}$  is the bus clock period. Since the bus cycle time of zero wait states is 2 clocks, we have:

100 MHz bus speed	
Memory cycle time with 0 WS	$2 \times 10 = 20 \text{ ns}$
Memory cycle time with 1 WS	$20 + 10 = 30 \text{ ns}$
Memory cycle time with 2 WS	$20 + 10 + 10 = 40 \text{ ns}$

It is preferred that all bus activities be completed with 0 WS. However, if the read and write operations cannot be completed with 0 WS, we request an extension of the bus cycle time. This extension is in the form of an integer number of WS. That is, we can have 1, 2, 3, and so on WS, but not 1.25 WS.

### Example 13-2

A 100-MHz bus system is using ROM of 50 ns speed. Calculate the number of wait states needed if the path delay is 5 ns.

#### Solution:

If ROM access time is 50 ns and the path delay is 5 ns, every time the CPU accesses ROM it must spend a total of 55 ns to get data into the CPU. A 100-MHz bus with zero WS provides only 20 ns ( $2 \times 10 \text{ ns} = 20 \text{ ns}$ ) for the memory read cycle time. To match the CPU bus speed with this ROM we must insert 4 wait states. This makes the cycle time 60 ns ( $20 + 10 + 10 + 10 + 10 = 60 \text{ ns}$ ). Notice that we cannot ask for 3.5 WS since the number of WS must be an integer. That would be like going to the store and wanting to buy half an apple. You must get one or more complete WS or none at all.

### Example 13-3

Find the effective memory performance of a 50-MHz bus speed with one wait state.

#### Solution:

Since the 0 WS memory cycle is 40 ns ( $1/50 \text{ MHz} = 20 \text{ ns}$  and  $20 \text{ ns} \times 2 = 40 \text{ ns}$ ), for 1 WS we have a memory cycle time of 60 ns. That means that the memory performance is the same as that of a 33.33 MHz bus speed ( $60 \text{ ns}/2 = 30 \text{ ns}$ , then  $1/30 \text{ ns} = 33.33 \text{ MHz}$ ) as far as memory access is concerned. This is 67% performance of the CPU with zero wait states.

---

### Burst Cycle

Some CPUs have the burst cycle. The memory cycle time of the CPU with the normal zero wait states is 2 clocks. In other words, it takes a minimum of 2 clocks to write to external memory. To increase the bus performance of the CPU, designer provides an additional option of implementing what is called a *burst cycle*. The CPUs have two types of memory cycles, non-burst (which is 2 clocks) and burst mode. In the burst cycle, the CPU can perform 4 memory cycles in just 5 clocks. The way the CPU performs the burst cycle read is as follows. The initial read is performed in a normal 2-clock memory cycle time, but the next three reads are performed each with only one clock. Therefore, four reads are performed in only 5 clocks. This is commonly referred to as 2-1-1-1 read, which means 2 clocks for the first read and 1 clock for each of the following three reads. This is in contrast to traditional CPUs, which was 2-2-2-2 for reading 4 words of aligned data. Of course, burst cycle reading is most efficient if the data and codes are in 4 consecutive locations. In other words, the burst cycle can be used to fetch a maximum of 4 words of information into the CPU in only 5 clocks, provided that they are aligned on word boundaries. See Figure 13-1. In many DRAM controllers, one can set the number of cycles to match the cache line refill. In the next section we will see how the static column DRAMs extend the burst cycle concept to read a large number of words.

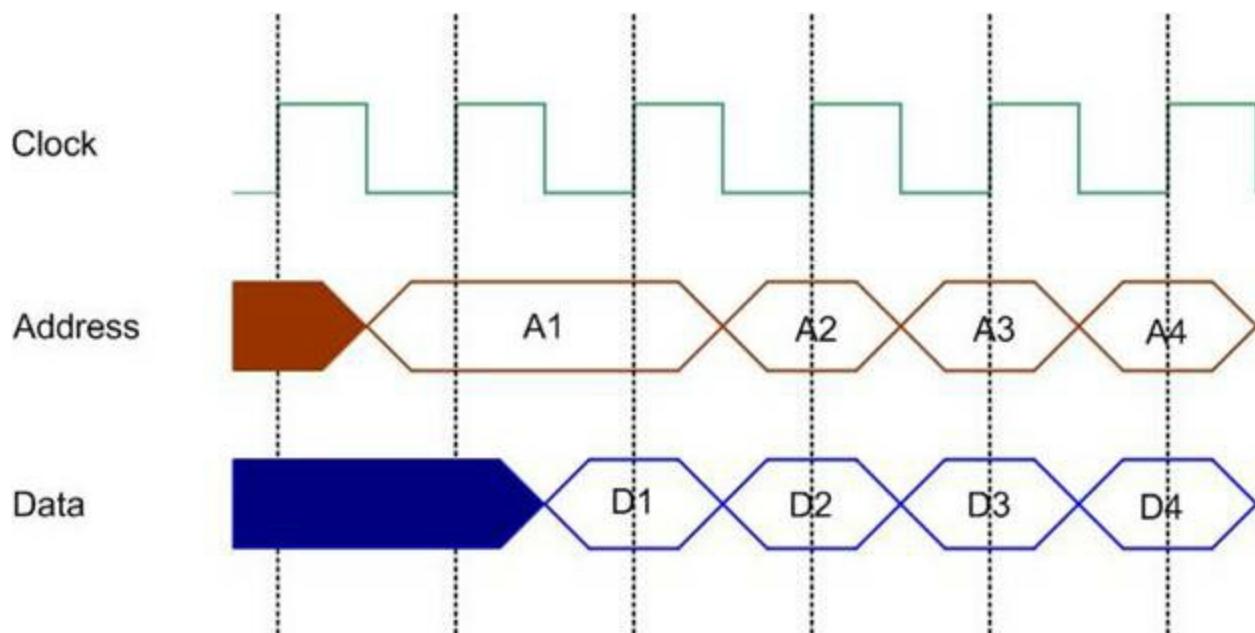


Figure 13-1: Burst Cycle Read in CPU

## DRAM memory banks

In Chapter 0 we examined the DRAM organization and capacity. The arrangement of DRAM chips on the system or memory module boards such as DIMM (dual in-line memory module) is often referred to as a *memory bank*. For example, the 8M bytes of DRAM can be arranged as one bank of 8 chips of  $1M \times 1$  organization, or 4 banks of  $256M \times 8$  organization. Figures 13-2 and 13-3 show the memory banks for 8-bit and 16-bit systems. Notice the use of an extra bit for every 8-bit of data to store the parity bit. With the extra parity bit, every bank requires an extra chip of  $\times 1$  organization for parity check.



Figure 13-2: A Possible Memory Configuration for 640M DRAM



Figure 13-3: DRAM Banks for 16-bit systems

## Memory cycle in ARM

Memory transfer cycle in ARM is one of these categories:

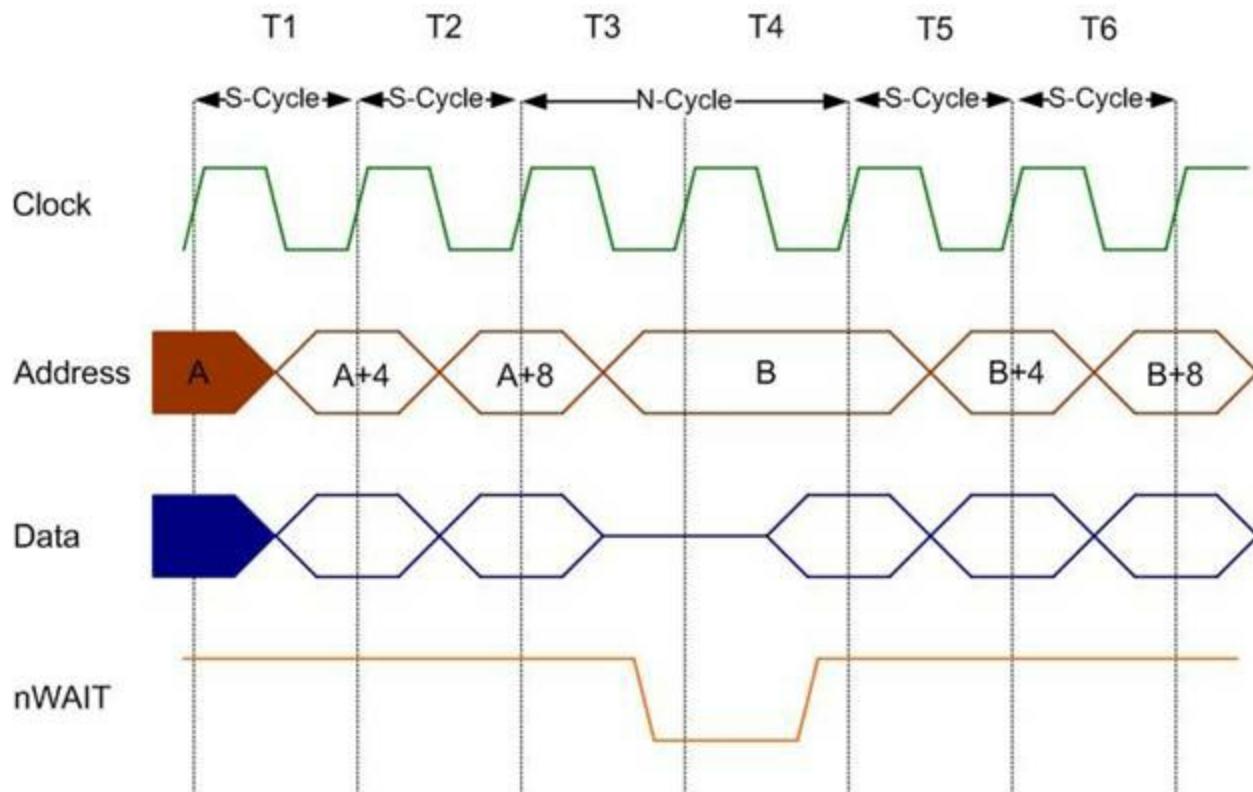
1. Non-sequential cycle: in non-sequential cycle, a location of memory is accessed which is not the same or near the last accessed memory location
2. Sequential cycle: in sequential cycle, a memory location is accessed from either the same location or the memory location after the location of preceding memory access.

In ARM documents Non-sequential cycle is referred to as N-cycle and Sequential cycle is referred to as S-cycle. Notice that N-cycle is longer than S-cycle to allow for the DRAM precharge and row access time.

We stretch the processor clock by lowering the nWAIT signal to generate wait state in ARM. See Example 13-4 to see S and N-cycles and wait state in ARM.

### Example 13-4

Analyze the following waveform of two read cycle in ARM



**Solution:**

T1: Address of memory location A+4 is put on the address bus. It is S-cycle because the new address is one word after the last accessed location (A) and nWAIT is high because no wait state is needed.

T2: Address of memory location A+8 is put on the address bus. It is S-cycle because the new address is a word after the last accessed location and nWAIT is high because no WS is needed.

T3: Address of memory location B is put on the address bus. It is start of N-cycle because the new address is not related to the last accessed location and nWAIT is low because one wait state is needed to access the data.

T4: Address of memory location B is still on the address bus. Now the data is available on the bus and nWAIT is returned high because no more wait state is needed.

T5: Address of memory location B+4 is put on the address bus. It is S-cycle because the new address is a word after the last accessed location and nWAIT is high because no WS is needed.

T6: The same as T5

---

In the next section we will study more about DRAM technology.

## Review Questions

1. Find the read/write cycle time of the following bus systems
  - (a) 40-MHz with 0 WS    (b) 50-MHz with 1 WS
  - (c) 66-MHz with 1 WS
2. A given CPU has a read/write cycle time of 50 ns. What does this mean?
3. Find the effective working frequency for memory access in each of the following.
  - (a) 40-MHz with 1 WS    (b) 50-MHz with 1 WS
4. If a given CPU has a read cycle time of 60 ns and 10 ns is used for the decoder and address/data path delay, how much is for memory access time?
5. If a given system is designed with 1 WS and has a 90-ns memory cycle time, find the CPU's frequency if the read/write cycle time of this CPU is 2 clocks.

## Section 13.2: DRAM Technology

To learn interfacing memory to high-performance computers, the different types of available RAM must first be understood. Although SRAMs are fast, they are expensive and consume a lot of space due to the use of flip-flops in the design of the memory cell. At the opposite end of the spectrum is DRAM, which is cheaper but is slow (compared to CPU speed) and needs to be refreshed periodically. The refreshing overhead together with the long access time of DRAM is a major issue in the design of high-performance computers. The problem of the time taken for refreshing DRAM is minimal since it uses only a small percentage of bus time, but the solution to the slow access time of DRAM is very involved. One common solution is using a combination of a small amount of SRAM, called cache (pronounced "cash"), along with a large amount of DRAM, thereby achieving the goal of near zero wait states. We discuss cache memory in Chapter 14. But we must understand what resources are available to high-performance system designers. To this end, the different types of available DRAM will be discussed. First we clarify some widely used terminology such as memory cycle time vs. memory access time. Then we describe different types of DRAMs. SDRAM which is most common type of DRAM in ARM systems is discussed in the last part of this section.

### Memory timing

Memory access time is defined as the time interval from the moment the addresses are applied to the memory chip address pins to the time the data is available at the memory's data pins. The memory data sheets refer to it as  $t_{AA}$  (address access time). Another commonly used time interval is  $t_{CA}$  (access time from CS), which is measured from the time the chip select pin of memory is activated to the time the data is available. In some cases, notably EEPROM,  $t_{OE}$  is the time interval between the moment OE (READ) is activated to the time the data is available. However, memory access time  $t_{AA}$  is the one most often advertised.

Memory cycle time is the shortest time interval between two consecutive accesses to the same memory chip. For example, a memory chip of 100 ns cycle time can be accessed no faster than 100 ns, which means that two back-to-back reads can be performed no faster than 200 ns, and 3 back-to-back reads will take 300 ns, and so on. It must be noted that while in SRAM the memory cycle time is equal to memory access time, this is not so in DRAM memory, as discussed later.

### Types of DRAM

There are different types of DRAM, which are categorized according to their mode of data access. The most widely used is SDRAM which is discussed at the end of this section. Other classic modes include standard, fast page mode (FPM), and extended data out (EDO) DRAM. Static column mode is a variant of FPM that will be discussed shortly.

### DRAM (standard mode)

Standard mode (also called *random access*) DRAM, which has the longest memory cycle time, requires the row address to be provided first and then the column address for each cell. Each group of address is latched in by the activation of RAS (row address select) and CAS

(column address select) inputs, respectively. See Figure 13-4.

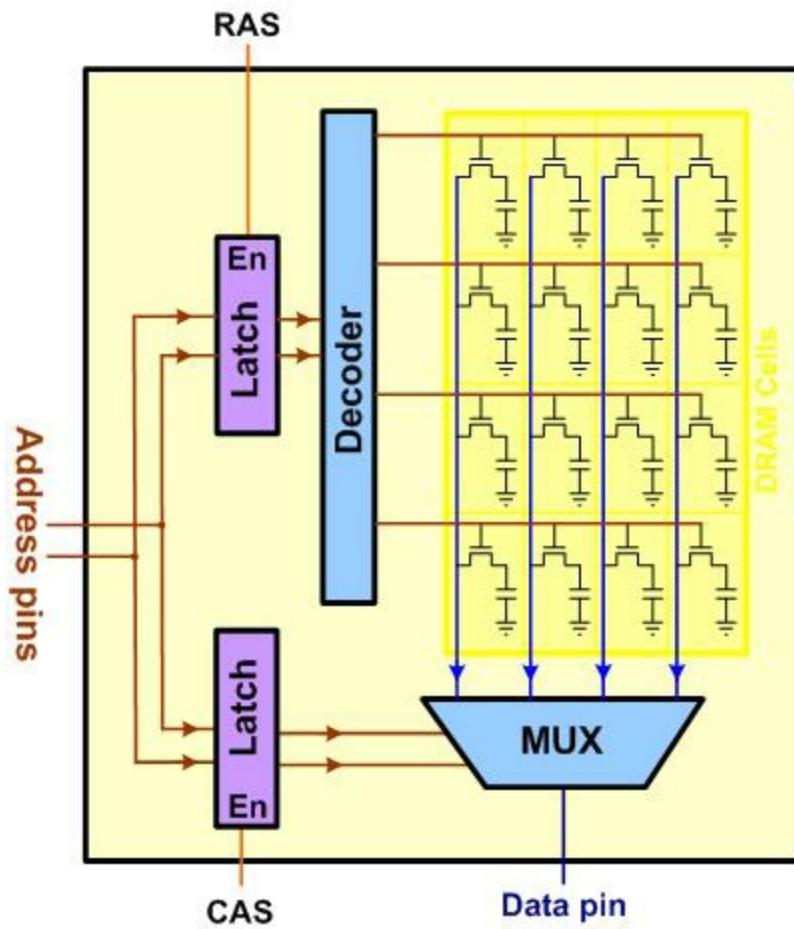


Figure 13-4: The Internal Structure of a 16x1 DRAM

The access time is from the time that the row address is provided to the time that the data is available at the output data pin of the DRAM chip. This is the access time that is commonly advertised and is called  $t_{RAC}$  (RAS access time, the access time from the moment RAS is provided). This is acceptable if we are accessing a random cell within DRAM. However, since most of the time data and code processed by the CPU are in consecutive memory locations and the CPU does not jump around to random locations (unless there is a branch or call instruction), the DRAM will be accessed with back-to-back read operations. Unfortunately, DRAM cannot provide the code (or data) in the amount of time called  $t_{RAC}$  if there is a back-to-back read from the same DRAM chip because DRAM needs a precharge time ( $t_{RP}$ ) after each RAS has been deactivated to get ready for the next access. This leads us back to the concept of memory cycle time for DRAM memory chips. The memory cycle time for memory chips is the minimum time interval between two back-to-back read/write operations. In SRAM and ROM, the access time and memory cycle time are always equal, but that is not the case for DRAMs. In DRAM, after RAS makes the transition to the inactive state (going from low to high), it must stay high for a minimum of  $t_{RP}$  (RAS precharge) to precharge the internal device circuitry for the next active cycle. Therefore, in DRAM we have the following approximate relationship between the memory access time and memory cycle time.

$$t_{RC} = t_{RAC} + t_{RP} \quad (\text{This is for standard mode})$$

read cycle time = RAS access time + RAS precharge time

For example, if DRAM has an access time of 100 ns, the memory cycle time is really about 190 ns (100 ns access time plus 90 ns precharge time). To access a single location in such a DRAM, 100 ns is enough, but to access more than one successively, 190 ns is required for each access due to the precharge time that is needed internally by DRAM to get ready to access the next capacitor cell. Tables 13-1 and 13-2 show DRAM and SRAM memory cycle times, respectively.

DRAM	RAS Access ( $t_{RAC}$ ) (ns)	Read Cycle ( $t_{RC}$ ) (ns)	RAS Precharge ( $t_{RP}$ ) (ns)
MCM44100-60	60	110	45
MCM44100-70	70	130	50
MCM44100-80	80	150	60

Table 13-1: DRAM Access Time vs. Cycle Time (4M × 1)

SRAM (IDT Product)	Address Access ( $t_{AA}$ ) (ns)	Read Cycle ( $t_{RC}$ ) (ns)
IDT71258S25	25	25
IDT71258S35	35	35
IDT71258S45	45	45
IDT71258S70	70	70

Table 13-2: SRAM Access Time vs. Cycle Time

The read cycle time not being equal to the access time is one of the major differences between SRAM and DRAM. Although in SRAM the write cycle time is equal to the access time, in DRAM of standard mode the write cycle time is about twice the access time normally advertised ( $t_{ACC}$ ). This could make a difference in the total time spent by the CPU to access memory. Look at Examples 13-5 and 13-6.

### Example 13-5

Compare the minimum CPU time needed to read 150 random memory locations of a given bank in each of the following.

(a) DRAM with  $T_{ACC} = 100$  ns and  $T_{RC} = 190$  ns

(b) SRAM of  $T_{ACC} = 100$  ns

**Solution:**

(a) DRAM requires 190 ns to access each location. Therefore, a total of  $150 \times 190 = 28,500$  ns would be spent by the CPU to access all those 150 memory locations.

(b) In the case of SRAM, the CPU spends only  $150 \times 100$  ns = 15,000. This would have been needed since  $T_{access} = T_{read\ cycle}$  ( $t_{ACC} = t_{RC}$ ).

### Example 13-6

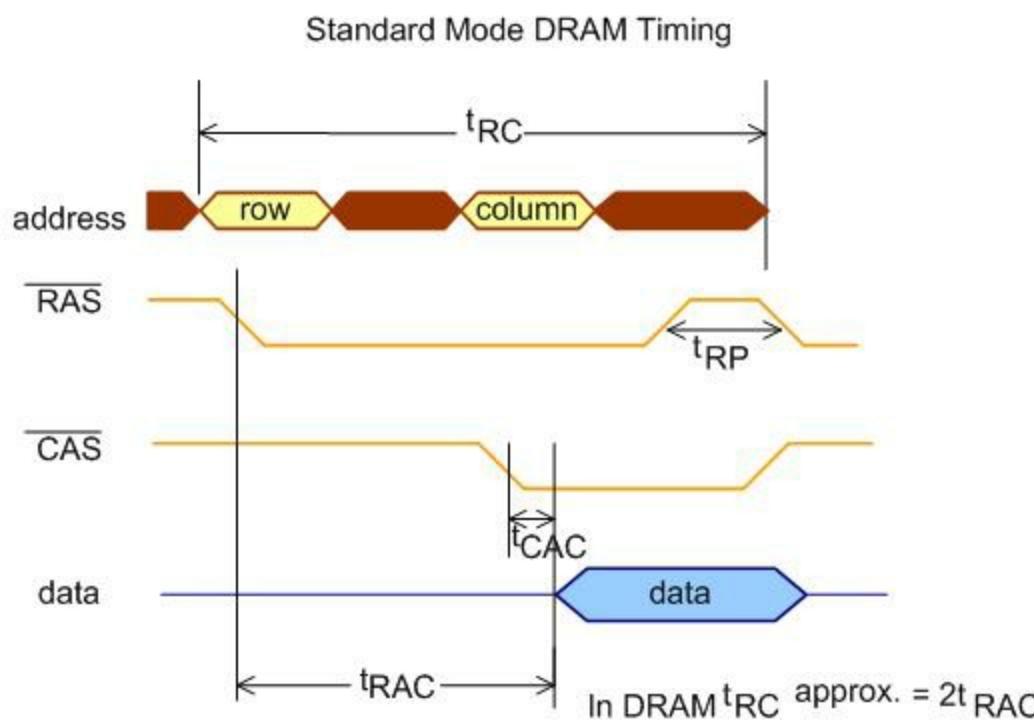
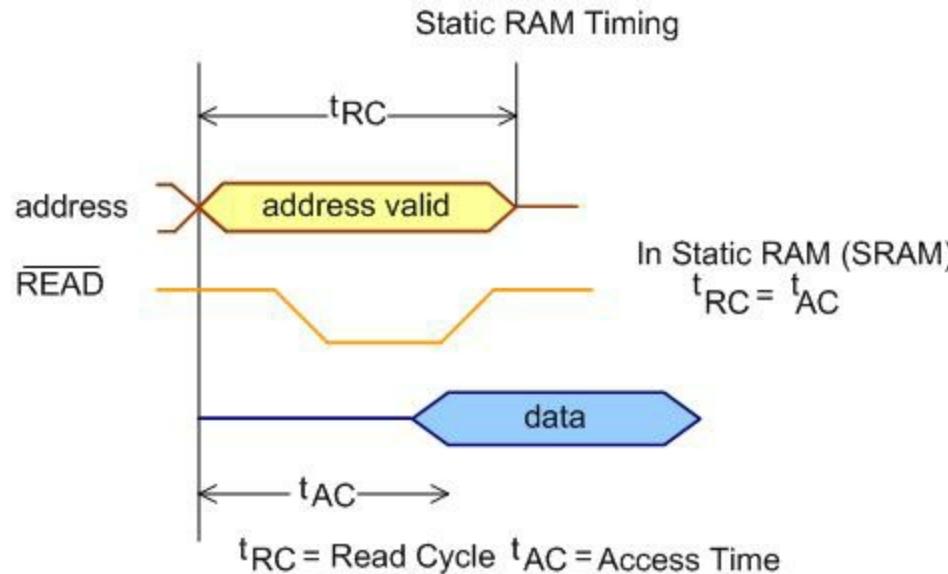
Calculate the time to access 1024 random bits of a 1M × 1 chip if  $t_{RC} = 85$  ns and  $t_{RAC} = 165$  ns.

## Solution:

For standard mode (also called random) we have the following for reading 1024 bits:  
time to read 1024 random bits =  $1024 \times t_{RC} = 1024 \times 165 \text{ ns} = 168,960 \text{ ns}$

---

From the above discussion and Example 13-5 we can conclude that for successive accesses of random locations inside the DRAM the CPU must spend a minimum of  $t_{RC}$  time on each access. See Figure 13-5 for DRAM and SRAM timing.



$t_{RAC}$  = access time from RAS

$t_{RC}$  = read cycle time

$t_{CAC}$  = access time from CAS

$t_{RP}$  = RAS precharge time

Figure 13-5: DRAM vs. SRAM Timing

## Fast Page Mode (FPM) DRAM

The storage cells inside DRAM are organized in a matrix of N rows and N columns. In reading a given cell, the address for the row ( $A_1-A_n$ ) is provided first and RAS is activated; then the address for the column ( $A_1-A_n$ ) is provided and CAS is activated. In DRAM literature the term page refers to a number of column cells in a given row. See Figure 13-4 and Examples 13-7 and 13-8.

### Example 13-7

Show how memory storage cells are organized in each of the following DRAM chips.

- (a)  $256K \times 1$       (b)  $1M \times 1$       (c)  $4M \times 1$

#### Solution:

(a) The  $256K \times 1$  has 9 address pins ( $A_0-A_8$ ); therefore, cells are organized in a matrix of  $2^9 \times 2^9 = 512 \times 512$ , giving 512 rows, each consisting of 512 columns of cells.

(b)  $1024 \times 1024$

(c)  $2048 \times 2048$

### Example 13-8

Assuming that the DRAMs in Example 13-7 are of page mode, show how each chip is organized into pages. Find the number of columns per page for (a), (b), and (c).

#### Solution:

(a) For  $1M \times 1$  we have 512 pages, where each page has 512 columns of cells.

(b) 1024 pages, where each page has 1024 bits (columns).

(c) 2048 pages each of 2048 bits

The idea behind page mode is that since memory locations are accessed consecutively in most situations, there is no need to provide both the row and column address for each location, as was the case in DRAM with standard timing. Instead, in page mode, first the row address is provided, RAS latches in the row address, and then the column addresses are provided and CAS toggles back and forth, latching in the column addresses until the last column of a given page is accessed. Then the address of the next row (page) is provided and the process is repeated. While the access time of the first cell is the standard access time using both row and column ( $t_{RAC}$ ), the access time in accessing the second cell on the last cell of the same page (row) is much shorter. In page mode DRAM when we are in a given page, each

successive cell can be accessed no faster than  $t_{PC}$  (page cycle time). See Figure 13-6.

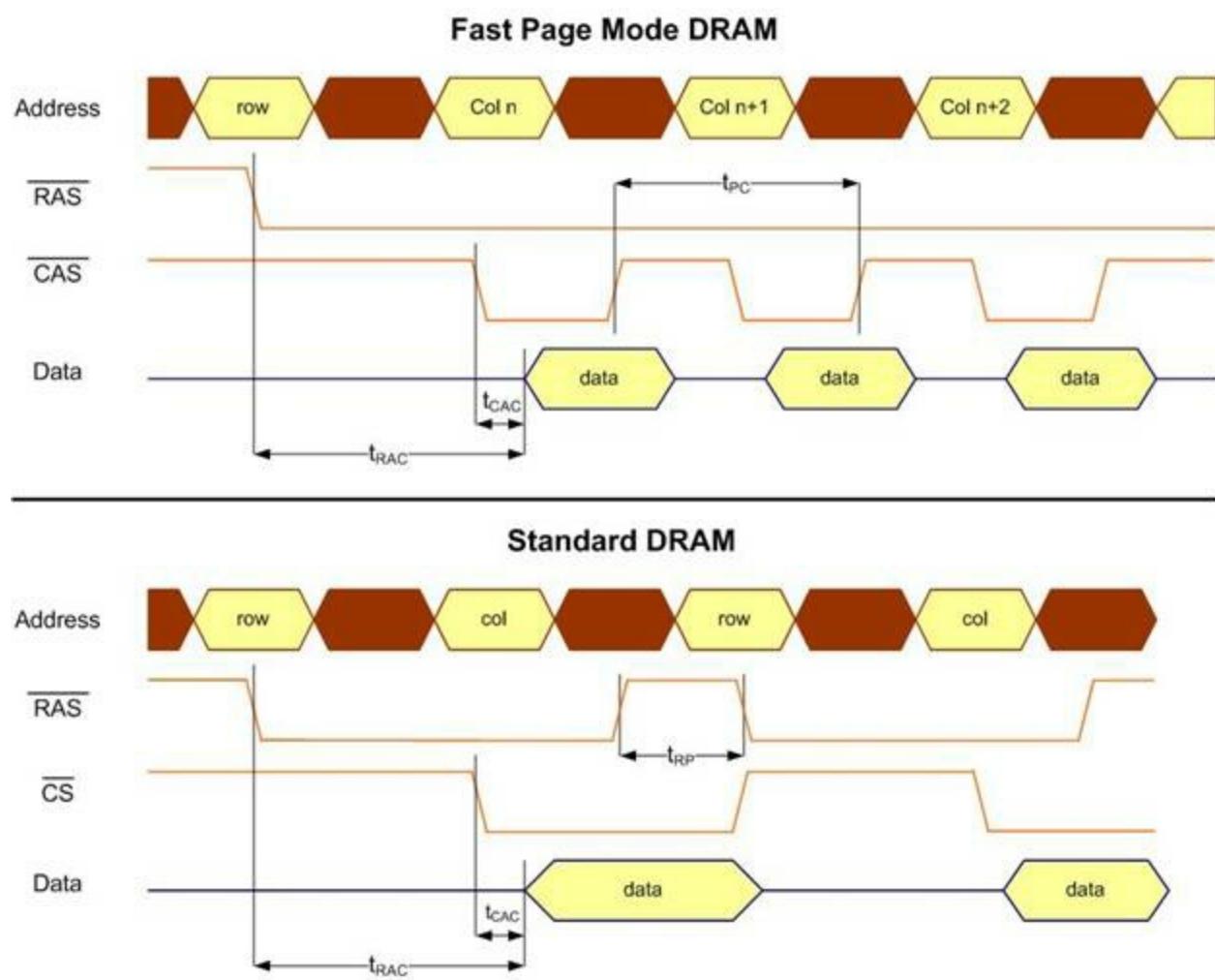


Figure 13-6: DRAM Fast Page Mode and Standard Mode Comparison

Table 13-3 gives page mode timing parameters. In DRAM of page mode both the standard mode and page mode are supported. See Example 13-9.

Page Mode DRAM	Access Time from RAS, $t_{RAC}$ (ns)	Read Cycle Time, $t_{RC}$ (ns)	Access Time from CAS, $t_{CAC}$ (ns)	Page Cycle Time, $t_{PC}$ (ns)
MCM44100-60	60	110	15	40
MCM44100-70	70	130	20	45
MCM44100-80	80	150	20	50

Table 13-3: Page Mode DRAM Timing Parameters (4M x 1)

### Example 13-9

Calculate the total time spent by the CPU to access an entire page of memory if the memory banks are page mode DRAM of  $1M \times 1$  with  $t_{RC} = 165$  ns,  $t_{RAC} = 85$  ns, and  $t_{PC} = 50$  ns.

**Solution:**

For page mode we have the following for reading 1024 bits:

Time to read 1024 bits of the same page =  $t_{RAC} + 1023 \times t_{PC} = 85 \text{ ns} + 1023 \times 50 \text{ ns} = 51,235 \text{ ns}$

## Static column mode

Static column mode is a variant of page mode. It makes accessing all the columns of a given row much simpler by eliminating the need for CAS. In this mode, the first location is accessed with a standard read cycle where the row address is latched by RAS followed by the column address and CS (chip select). As long as RAS and CS remain low, the contents of successive cells appear at the data output pin of DRAM until the last column of a given row is accessed. This means that the initial access time of the first cell is the standard access time ( $t_{RAC}$ ), but each subsequent column in that row is accessed in a time called  $t_{AA}$  (access time from column address).

In static column mode where the initial standard access time is  $t_{RAC}$ , when we are in a given page, any cell can be accessed with the access time of  $t_{AA}$ , but all the successive bits can be accessed no faster than  $t_{SC}$  (static column cycle time). See Figure 13-7. Table 13-4 gives static column mode timing parameters.

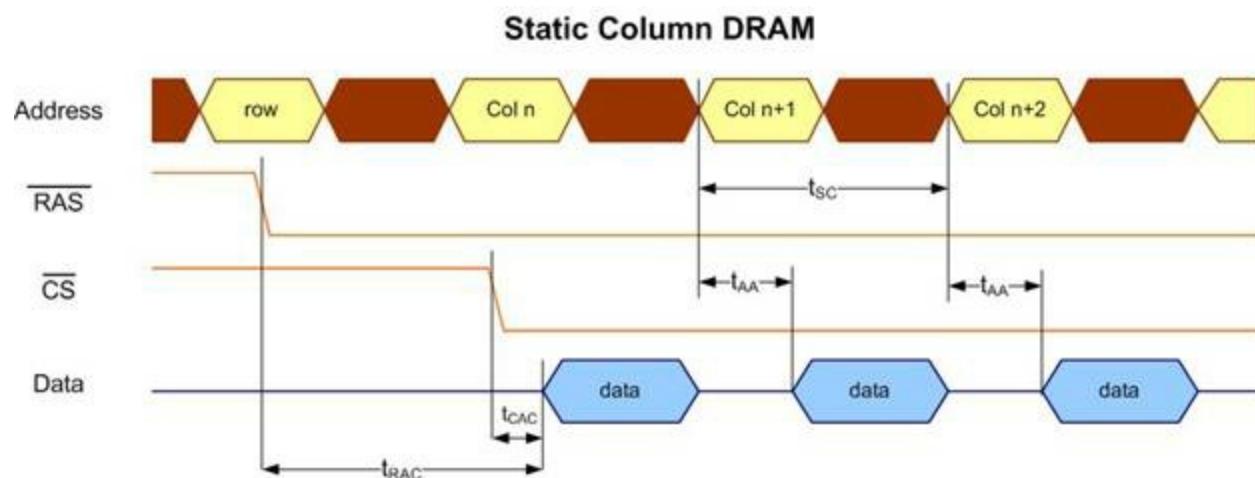


Figure 13-7: DRAM Static Column Mode Timings

Static Column DRAM	T RAS Access Time, $t_{RAC}$ (ns)	T Read Cycle, $t_{RC}$ (ns)	T Column Access Time, $t_{AA}$ (ns)	Cycle Time, $t_{SC}$ (ns)
MCM54102A-60	60	110	30	35
MCM54102A-70	70	130	35	45
35MCM54102A-80	80	150	40	45

Table 13-4: Static Column DRAM Timing Parameters (4M × 1)

Comparing Examples 13-9 and 13-10, if the time spent by the CPU is the same for both

the page mode and static column mode, what is the advantage of static column mode? The answer is that static-column-mode DRAM design is simpler since there is no circuit or timing requirement for the CAS pin. Notice in Figure 13-7 that we need to keep both RAS and CS (chip select) low in order to access successive cells. Table 13-5 compares cycle time of standard, fast page mode, and static column DRAMs.

### Example 13-10

Calculate the total time spent by the CPU to access the entire page of memory if the memory banks are static-column-mode DRAMs of  $1M \times 1$  with  $t_{RC} = 165$  ns,  $t_{RAC} = 85$  ns, and  $t_{SC} = 50$  ns.

#### Solution:

For static column mode we have the following for reading 1024 bits:

$$\begin{aligned} \text{time to read 1024 bits of the same page} &= t_{RAC} + 1023 \times t_{SC} = 85 \text{ ns} + 1023 \times 50 \text{ ns} = \\ &51,235 \text{ ns} \end{aligned}$$

Name	Standard Notation, time	FPM Notation, time	Static Column Notation, time
Access time from row	$t_{RAC} = 85\text{ns}$	$t_{RAC} = 85\text{ns}$	$t_{RAC} = 85\text{ns}$
Access time from column			$t_{AA} = 45\text{ns}$
Cycle time	$t_{RC} = 165\text{ns}$	$t_{PC} = 50\text{ns}$	$t_{SC} = 50\text{ns}$

Table 13-5: Timing comparison of FPM, Static Column and Standard Mode DRAM

### EDO DRAM: origin and operation

We discussed standard and fast page mode DRAM. The following describes the operation and limitations of fast page DRAM and how it led to EDO DRAM.

1. The row address is provided and latched in when RAS falls. This opens the page.
2. The column address is latched in when CAS falls and data shows up after  $t_{CAC}$  has elapsed. However, the next column of the same row (page) cannot be accessed faster than  $t_{PC}$  (page cycle time). This means that accessing consecutive columns of opened pages is limited by the  $t_{PC}$ . The  $t_{PC}$  timing itself is influenced by how long CAS has to stay low before it goes high. Why don't DRAM designers pull up the CAS faster in order to shorten the  $t_{PC}$ ? This seems like a very logical suggestion. However, there is a problem with this approach in fast page mode: When the CAS goes high, the data output is turned off. So if CAS is pulled high too fast (to shorten the  $t_{PC}$ ), the CPU does not have enough time to catch the data. One solution is to change the internal circuitry of fast page DRAM to allow the data to be available longer (even if CAS goes high). This is exactly what happened. As a result of this change, the name EDO (extended data-out) was given to avoid confusion with fast page mode DRAM. This is the reason that EDO is sometimes called *hyper-page* since it is the hyper version of fast page DRAM. Table 13-6 shows a

comparison of FPM and EDO DRAM timing. Notice in both cases that all the parameters are the same except  $t_{PC}$ . For the EDO version of page mode, the  $t_{PC}$  is 10 ns less than fast page mode.

	FPM	EDO
$t_{RAC}(ns)$	60	60
$t_{RC}(ns)$	110	110
$t_{PC}(ns)$	35	25

Table 13-6: 60ns 4M DRAM Timing

In examining  $t_{PC}$  timing in Figure 13-8, notice that  $t_{PC}$  (page cycle time) consists of two portions:  $t_{CP}$  (CAS precharge time) and  $t_{CAS}$  (CAS pulse width). The  $t_{CP}$  is similar across 70 ns, 60 ns, and 50 ns DRAMs of FPM and EDO (about 10 ns). It is  $t_{CAS}$  that varies among these DRAMs. In EDO this portion is made as small as possible. Figure 13-8 compares FPM and EDO timing.

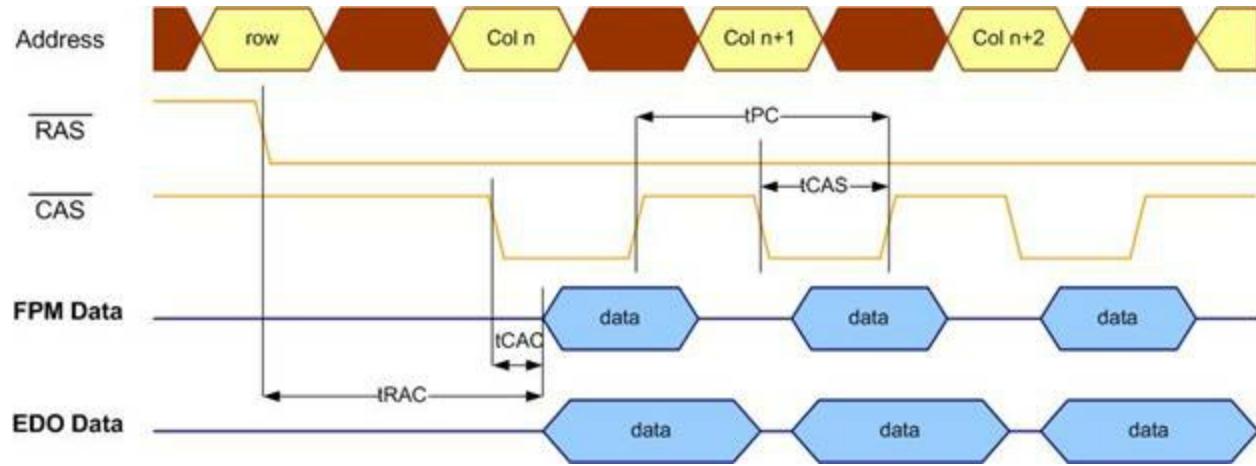


Figure 13-8: FPM and EDO Comparison

## SDRAM (Synchronous DRAM)

When the CPU bus speed goes beyond 75 MHz, even EDO is not fast enough. SDRAM is a memory for such systems. First, let us see why it is called synchronous DRAM. In all the traditional DRAMs (standard, FPM, and EDO), CPU timing is not synchronized with DRAM timing, meaning that there is no common clock between the CPU and DRAM for reference. In those systems it is said that the DRAM is asynchronous with the microprocessor since the CPU presents the address to DRAM and memory provides the data in the master/slave fashion. If data cannot be provided on time, the CPU is notified and the CPU inserts a wait state into its bus timing and waits until the DRAM is ready. In other words, the CPU bus timing is dependent upon the DRAM speed. This is not the case in synchronous DRAM. In systems with SDRAM, there is a common clock (called the system clock, main clock, or master clock) that runs between the CPU and SDRAM. All bus activities (address, data, control) between the CPU and SDRAM are synchronized with this common clock. That is, the common clock is the point of reference for both the CPU and SDRAM and there is no deviation from it and hence no waiting by the CPU. As you examine the timing figures in EDO and page mode, you will not find such a clock.

## Active and precharge command in SDRAM

To select a row (page) we should activate it by making the RAS low at a rising edge of clock signal. Active command is used to activate a row for subsequent access. In SDRAM a page remains active for both of read and write operations until we make it inactive by using a precharge command. Notice that we have to issue a precharge command before activating another row. A precharge command can be issued by making both of RAS and WE low while holding CAS high at a rising edge of clock signal. Figure 13-9 shows a simplified write command followed by precharge. Table 13-7 shows the timings of MT48LC16M16 which is a 256Mb SDRAM from the Micron Corp.

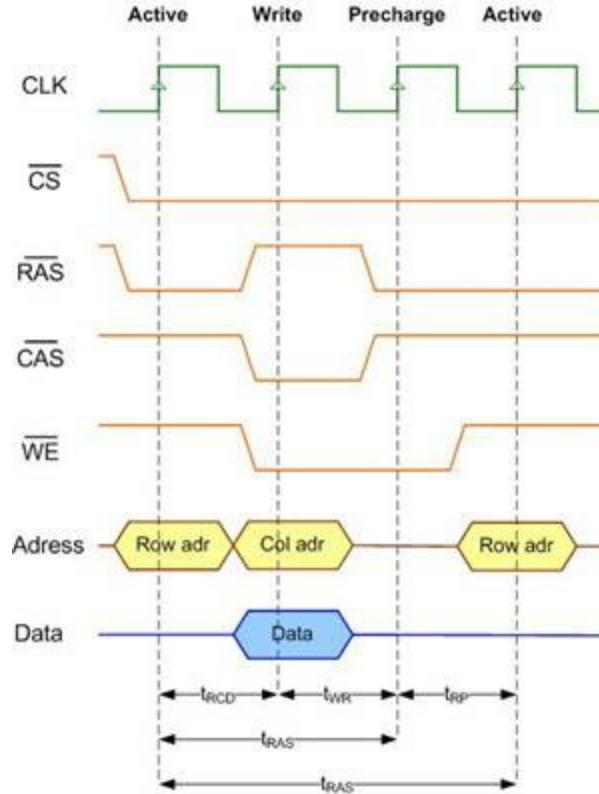


Figure 13-9: SDRAM Simple Write Command

Name	Description	Time
$t_{RCD}$	Active to read or write delay	15ns
$t_{WR}$	Write recovery time	7ns + 1 CLK
$t_{RP}$	Precharge command period	15ns
$t_{RAS}$	Active to precharge command	37ns and not more than 120ms
$t_{RC}$	Active to active command period	60ns

Table 13-7: SDRAM Timings

## NOP and Inhibit in SDRAM

When the CPU is faster than SDRAM, no operation (NOP) or command inhibit can be issued to an SDRAM device to prevent unwanted commands from being registered during wait states. Notice that the operation already in progress are not affected by NOP or command inhibit. According to Table 13-7, the  $t_{WR}$  is 7ns + 1CLK. It means that there must be at least one NOP or command inhibit after any write operation. To calculate the number of NOPs needed for a timing parameter, we should divide it by the clock period and then rounded. Figure 13-10 shows a write command followed by precharge for a 100MHz clock. Notice that nop1 is added

to make tRCD more than 15ns. nop2 and nop3 are added to make tRAS more than 37ns and nop4 and nop5 are added to make tRC more than 60ns.

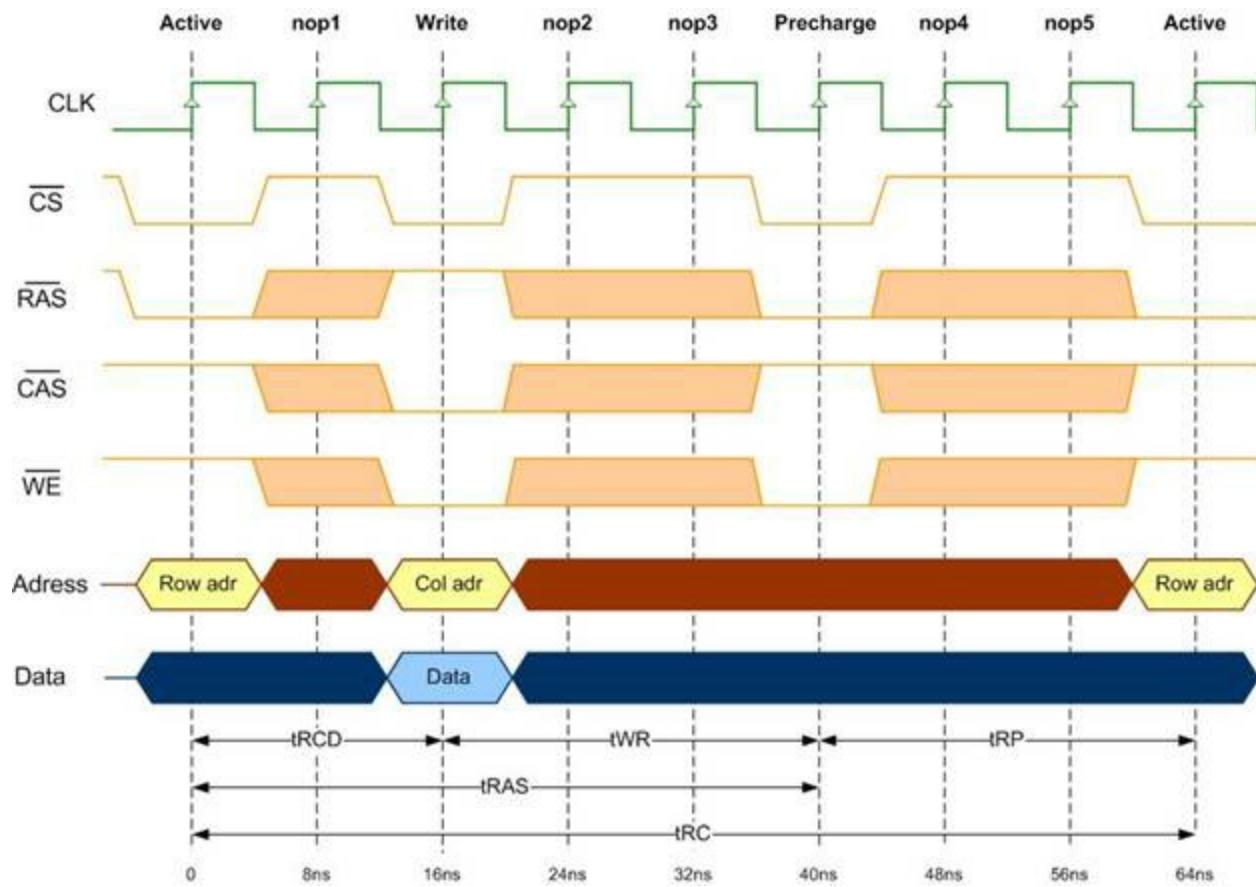


Figure 13-10: SDRAM Write Operation with NOPs

## SDRAM and burst mode

The presence of the common system clock between the CPU and SDRAM lends itself to what is called burst I/O. Although burst I/O will do both read and write, we will discuss the read operation for the sake of simplicity.

In burst read, the address of the first location is provided as normal. RAS is first, followed by CAS. However, most of the times, we need to read several consecutive locations in a page and there is no need to provide the full address of each column and pay the timing penalty for address setup and hold time. Why not simply program the burst SDRAM to let it know how many consecutive locations are needed? That is exactly the idea behind many SDRAMs. They are capable of being programmed to output up to 256 consecutive locations inside one page of DRAM. In other words, the number of burst reads can be 1, 2, 4, 8, 16, or 256, and burst SDRAM can be programmed in advance for any number of these reads. The number of burst reads is referred to as burst length. In many recent SDRAMs, the burst length can be as high as a whole page. Burst read shortens memory access time substantially. For example, if burst length is programmed for 8, for the first location we need the full address of RAS followed by CAS. However, for the second, third, ..., eighth, we can get the data out of the SDRAM with a minimum delay, limited only by the internal circuitry of DRAM. See Figure 13-11.

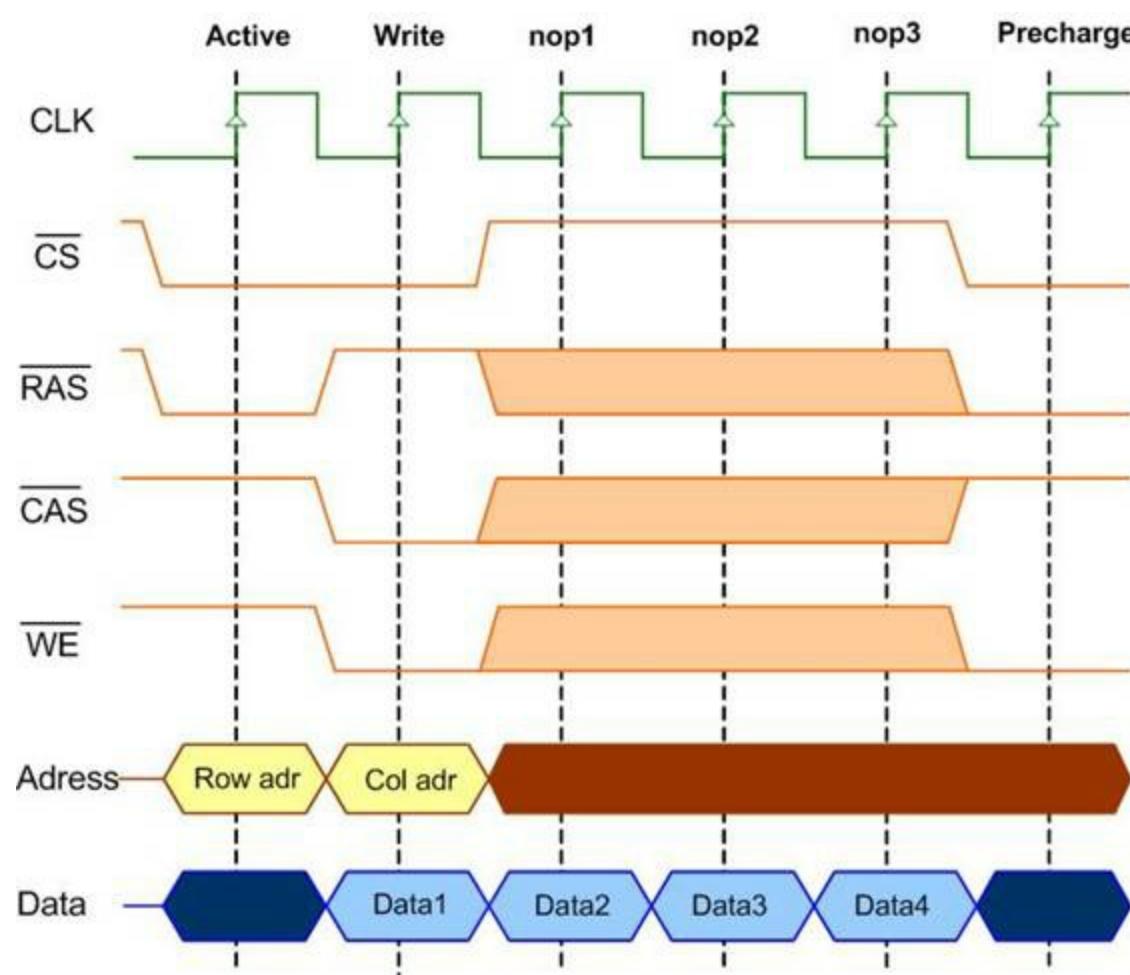


Figure 13-11: SDRAM Burst Write Operation

## SDRAM banks and interleaving

One of the methods used to overcome the problem of precharge time in DRAMs is the interleaving. In this method, each SDRAM chip has two or four sets of banks and the CPU accesses each set of banks alternately. In this way the precharge time of one set of banks is hidden behind the access time of the other one. This means that while the CPU is accessing one set of banks, the other set is being precharged. Assume that a CPU has a memory cycle time of 100 ns. Using DRAM with access time of 70 ns and the precharge of 65 ns gives a DRAM cycle time of 135 ns ( $70 + 65 = 135$ ). This is much longer than the 100 ns provided by the CPU. Using interleaved memory design can solve this problem. In this case when the CPU accesses bank set A, it goes on to access bank set B while set A takes care of its precharge time. Similarly, when the CPU accesses set A, the set B banks will have time to precharge. Notice that SDRAM chip has one or two extra pins called BA0 and BA1 to select a memory bank. Look at Figure 13-12. By incorporating both the burst mode and interleaving concepts into SDRAM, it is used by many ARM-based systems for external memory connection.

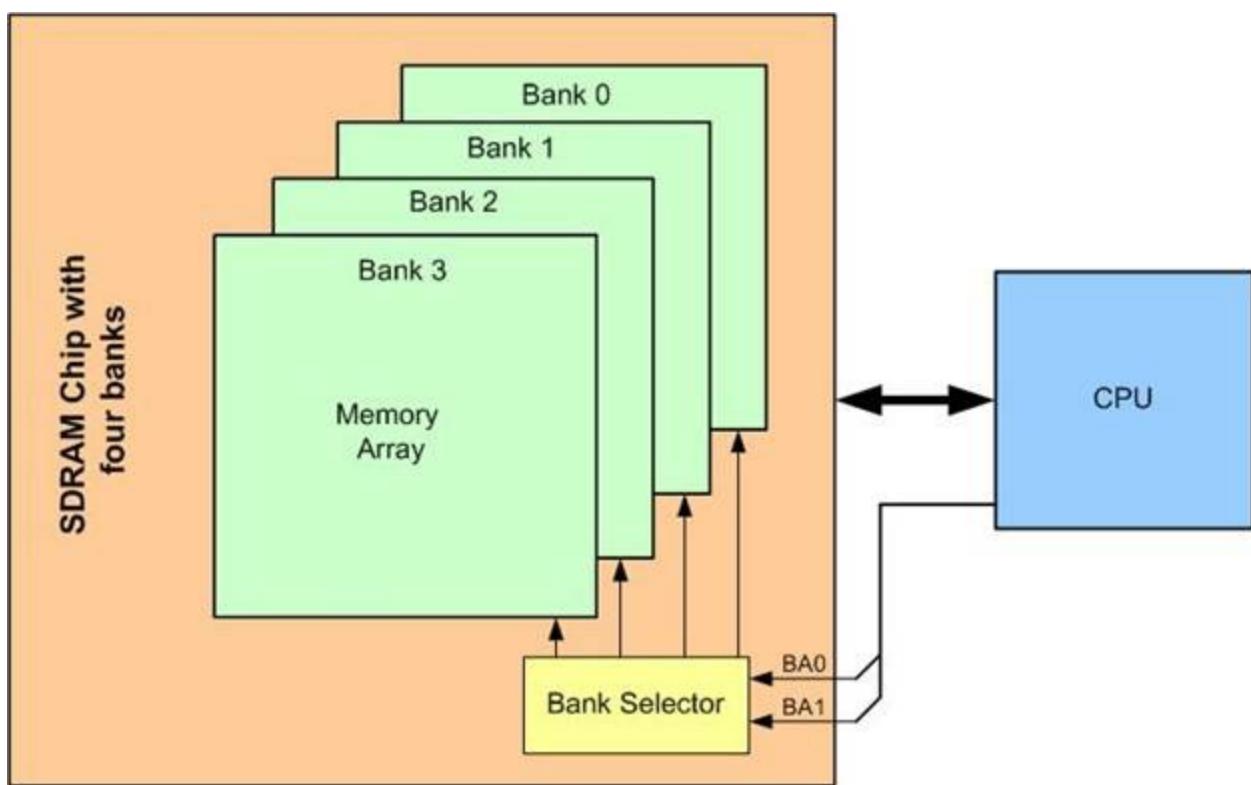


Figure 13-12: SDRAM with Four Banks

## CAS Latency

In read operation it can be programmed that how many clocks after CAS will the data appear at the data pins. It is called CAS latency or read latency and can be 1, 2, or 3 clocks. Figure 13-13 shows data for CAS latency (CL) = 2 and CL = 3 .

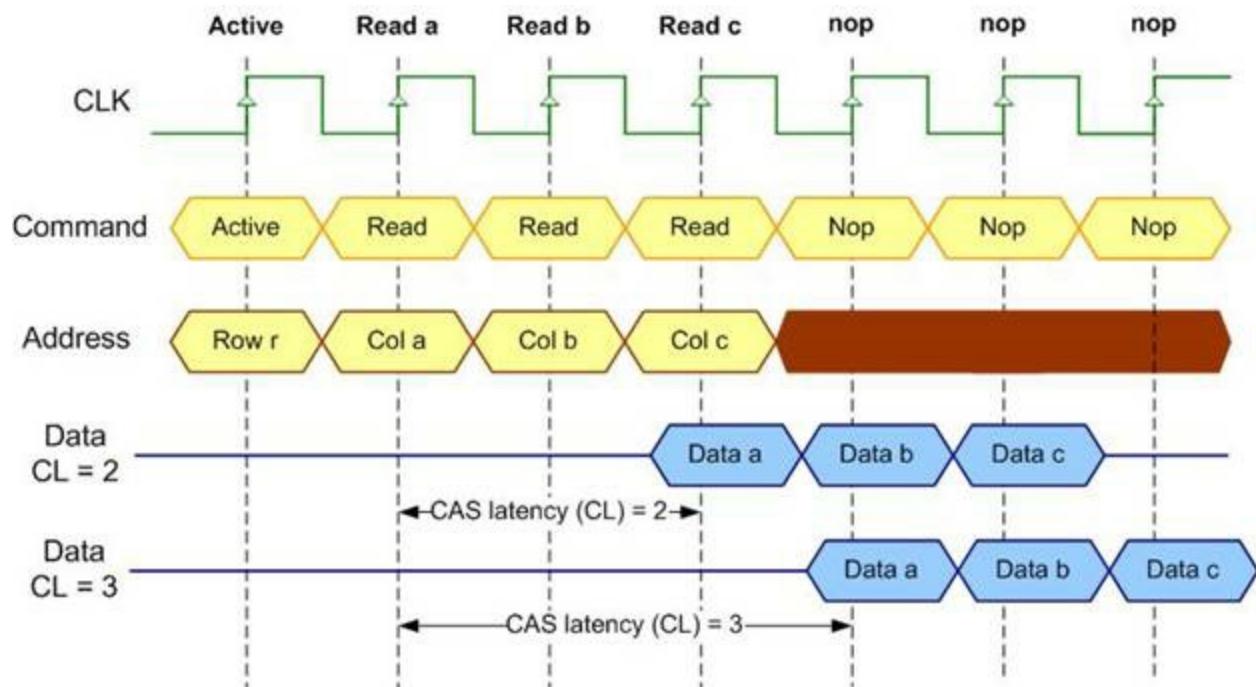


Figure 13-13: SDRAM CAS Latency

## Double data rate (DDR)

DDR (Double Data Rate) DRAMs are the same as SDRAM but use a double rate interface to transfer data on both rising edge and falling edge of the clock. DDR2 and DDR3 increased this factor by  $\times 4$  and  $\times 8$ .

This concludes the discussion of DRAM operation modes. It must be noted that in many systems one of the above modes is implemented in order to eliminate the need for the wait state to access every bit of DRAM. As seen from the above discussion, even the best of any of these modes still cannot eliminate the need for the wait state entirely unless SRAM is used for the entire memory, which is prohibitively expensive. The best solution is to use a combination of SRAM and DRAM and using cache memory. See the next chapter.

## Review Questions

1. In which type(s) of memory is the read cycle time equal to the memory access time?
2. A given DRAM is advertised to have an access time of 50 ns. What is the approximate memory cycle time for this DRAM?
3. A given DRAM has a 120-ns memory read cycle time. What is its access time ( $t_{RAC}$ )?
4. In DRAM, a read cycle consists of \_\_\_\_\_ and \_\_\_\_\_.
5. Assume an ARM system of interleaved memory with 2M bytes initial DRAM for each of the following.
  - (a) Show how the banks are organized.
  - (b) What is the minimum memory addition?
6. True or false. In page mode, the initial read takes  $t_{RAC}$ .
7. For page mode DRAM, while we are in a given page, we can access successive memory locations no faster than \_\_\_\_\_.
8. Calculate the time the CPU must spend to access 100 locations all within the same page if  $t_{RAC} = 60$  ns and  $t_{PC} = 30$  ns.
9. The higher the system frequency, the less noise can be tolerated in the system. Which is preferable in a 20-MHz system, static column or page mode DRAM?
10. A 200-MHz ARM has a bus frequency of \_\_\_\_\_.
11. A 100-MHz ARM has a bus frequency 2/3 of the CPU. What is the read cycle time for this processor?
12. When a page is opened, what limits us in accessing consecutive columns?
13. True or false. In EDO, when CAS goes up the data output is turned off.
14. Which of the following DRAMs has a common synchronous clock with the CPU?
  - (a) FPM
  - (b) EDO
  - (c) SDRAM
  - (d) all of the above
15. True or false. SDRAM incorporates interleaved memory internally.

## Section 13.3: Data Integrity in DRAM and ROM

In this section we examine the methods used in checking the data integrity in ROM and RAM.

### Using checksum byte in ROM

To ensure the integrity of the contents of ROM, every system must perform a checksum calculation. The process of checksum will detect any corruption of the contents of ROM. One of the causes of ROM corruption is current surge, either when the system is turned on or during operation. The checksum method uses a checksum byte. This checksum byte is an extra byte that is tagged to the end of a series of bytes of data. To calculate the checksum byte of a series of bytes of data, the following steps can be taken.

1. Add the bytes together and drop the carries.
2. Take the 2's complement of the total sum, and that is the checksum byte, which becomes the last byte of the stored information.

To perform the checksum operation, add all the bytes, including the checksum byte. The result must be zero. If it is not zero, one or more bytes of data have been changed (corrupted). To clarify these important concepts, see Examples 13-11 and 13-12. These two examples show the check-sum byte since the system ROM is assumed to be byte-wide. In the 16-bit systems the check-sum value is calculated by adding the 16-bit values. In the ARM systems the check-sum is calculated by adding the 32-bit words since the instructions size is 32-bit wide.

#### Example 13-11

Assume that we have 4 bytes of hexadecimal data: 0x25, 0x62, 0x3F, and 0x52.

- (a) Find the checksum byte.
- (b) Perform the checksum operation to ensure data integrity.
- (c) If the second byte 62H had been changed to 22H, show how checksum detects the error.

#### Solution:

- (a) The checksum is calculated by first adding the bytes.

$$\begin{array}{r} 25 \\ + 62 \\ + 3F \\ + 52 \\ \hline 118 \end{array}$$

The sum is 0x118, and dropping the carry, we get 0x18. The checksum byte is the 2's complement of 0x18, which is 0xE8.

- (b) Adding the series of bytes including the checksum byte must result in zero. This

indicates that all the bytes are unchanged and no byte is corrupted.

$$\begin{array}{r} 25 \\ + 62 \\ + 3F \\ + 52 \\ + \underline{E8} \\ 2\ 00 \text{ (dropping the carry)} \end{array}$$

(c) Adding the series of bytes including the checksum byte shows that the result is not zero,

which indicates that one or more bytes have been corrupted.

$$\begin{array}{r} 25 \\ + 22 \\ + 3F \\ + 52 \\ + \underline{E8} \\ 1\ C0 \text{ dropping the carry, we get } 0xC0. \end{array}$$

---

### Example 13-12

Assuming that the last byte of the following data is the checksum byte, show whether the data has been corrupted or not: 0x28, 0xC4, 0xBF, 0x9E, 0x87, 0x65, 0x83, 0x50, 0xA7, and 0x51.

#### Solution:

The sum of the bytes plus the checksum byte must be zero; otherwise, the data is corrupted

$$28 + C4 + BF + 9E + 87 + 65 + 83 + 50 + A7 + 51 = 500$$

By dropping the accumulated carries (the 5), we get 00. The data is not corrupted.

---

### Checksum program

When the system is turned on, one of the first things the ROM boot does is to test the system ROM. In the 16-bit systems such as Thumb the check-sum value is calculated by adding the 16-bit values. In the ARM systems the check-sum is calculated by adding the 32-bit words since the instructions size is 32-bit wide. Program 13-1 shows the program using the checksum method. Notice in the code how all the words are added together without keeping the track of carries. Then, the total sum is tested to see if it is zero. The zero flag is expected to be set to high if there is no corruption of data. If it is not, the ROM is corrupted.

#### Program 13-1

```
; CHECK-SUM PROGRAM TEST
LDR    R0,=ROM_ADDRESS      ; pointer to data ROM
LDR    R2,=ROM_SIZE         ; data size
```

```

MOV     R1,#0           ; R1 is used to hold the sum
; add the words including the check-sum word
OVER   LDR     R3,[R0]      ; load the word
ADD    R1,R3          ; add the word
ADD    R0,R0,#4        ; point to next one
SUBS   R2,R2,#1        ; decrement counter
BNE    OVER          ; until all is done
; test it
TST    R1,R1          ; see if the sum is zero
BEQ    NO_ER         ; if it is, go display the message
; display the message for error
B      .              ; stay here

NO_ER
;display the message for no error

```

## Hard and soft error detection in DRAM

There are two types of errors that can occur in DRAM chips: soft error and hard error. In a hard error, some bits or an entire row of memory cells inside the memory chip get stuck to high or low permanently, thereafter always producing 1 or 0 regardless of what you write into the cell(s). In a soft error, a single bit is changed from 1 to 0 or from 0 to 1 due to current surge or certain kinds of particle radiation in the air. Parity is used to detect such errors. Including a parity bit to ensure data integrity in RAM is the most widely used method since it is the simplest and cheapest. See Figure 13-14. This method can only indicate if there is a difference between the data that was written to memory and the data that was read. It cannot correct the error as is the case with some high-performance computers. In those computers the EDC (error detection and correction) method is used to detect and correct the error bit. There are many parity bit generator and checker chips and ASIC circuits. These chips have 9 inputs and 2 outputs. Depending on whether an even or odd number of ones appears in the input, the even or odd output is activated. If all 9 inputs have an even number of 1 bits, the even output goes high. If the 9 inputs have an odd number of high bits, the odd output goes high. When a byte of information is written to a given memory location in DRAM, the even-parity bit is generated and saved on the ninth DRAM chip as a parity bit. When a byte of data is read from the same location, the parity bit is generated again. If there is a difference between the data written and the data read the parity bit checker (using an Exclusive-OR) is activated indicating that there is a parity bit error, meaning that the data read is not the same as the data written.

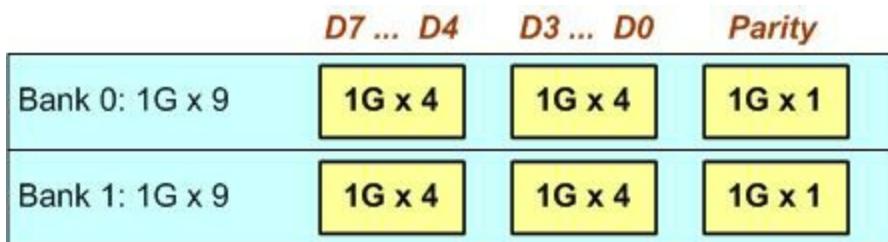


Figure 13-14: A Possible Memory Configuration for 2G DRAM

## Review Questions

- Find the checksum byte for the following bytes: 0x24, 0x76, 0xF5, 0x98, 0x89, 0x7A, 0x61,

0xC2.

2. To detect corruption of information stored in RAM and ROM memories, system designers use the \_\_\_\_\_ method for RAM and the \_\_\_\_\_ method for ROM.
3. Assume that due to slight current surge in the power supply, a byte of RAM has been corrupted while the computer is on. Can the system detect the corruption while the computer is on? Is this also the case for ROM?

## Section 13.4: Concept of DMA

In computers there is often a need to transfer a large amount of data between memory or between memory and peripherals such as disk drives. In such cases, using the CPU to transfer the data is too slow since the data first must be fetched into the CPU and then sent to its destination. In addition, the process of fetching and decoding the instructions themselves adds to the overhead and also stops the CPU from processing other tasks. For these reasons, in most computers and microcontrollers there is a DMAC (direct memory access controller), whose function is to bypass the CPU and provide a direct connection between peripherals and memory, thus transferring the data as fast as possible.

One problem with using DMA is that there is only one set of buses (one set of each bus: data bus, address bus, control bus) in a given computer and no bus can serve two masters at the same time. The buses can be used either by the main CPU or the DMA. Since the CPU has primary control over the buses, it must give permission to DMA to use them. How is this done? The answer is that any time the DMA needs to use the buses to transfer data, it sends a signal called *HLDR* (*hold request*) to the CPU and the CPU will respond by sending back the signal *HLDA* (*hold acknowledge*) to indicate to the DMA that it can go ahead and use the buses. While the DMA is using the buses to transfer data, the CPU is sitting idle, and conversely, when the CPU is using the bus, the DMA is sitting idle. After DMA finishes its job it will make HOLD go low and then the CPU will regain control over the buses. See Figure 13-15. When a cache is added to the system, while the DMA controller has the bus, the CPU may still execute programs out of cache without stopping.

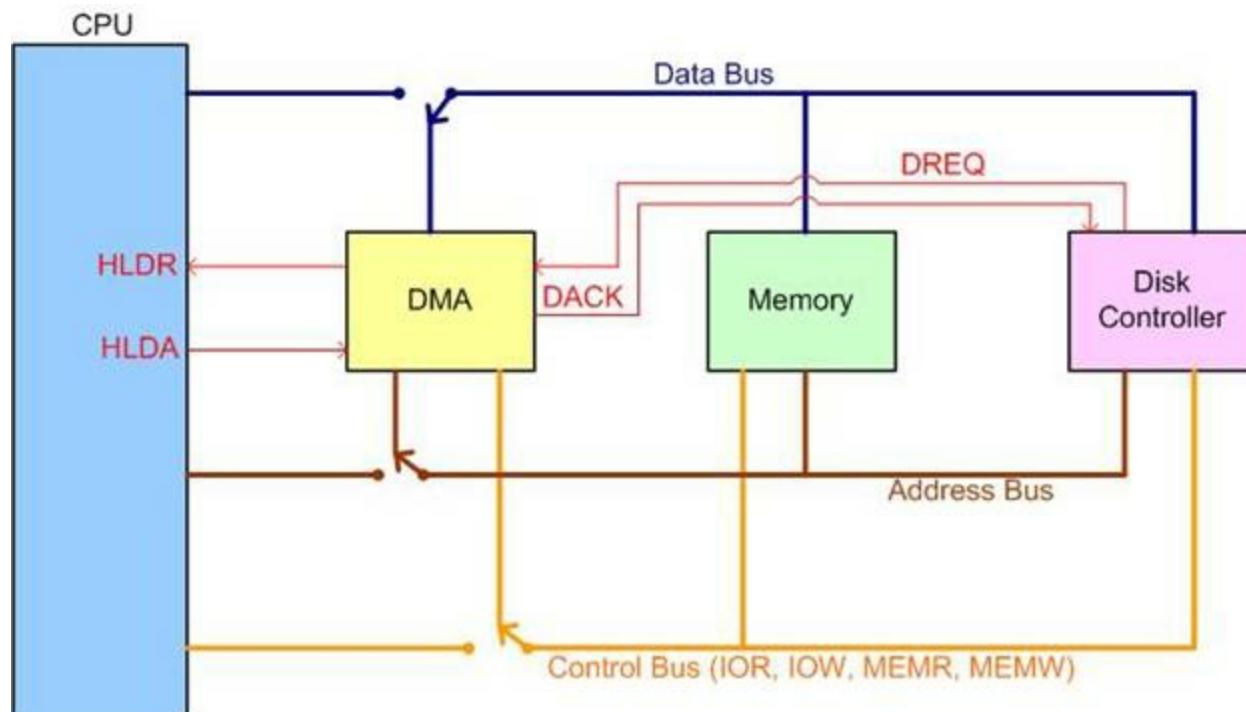


Figure 13-15: DMA Usage of System Bus

For example, if the DMA is to transfer a block of data from memory to an I/O device such as a disk, it must know the address of the beginning of the block (address of the first byte of data) and the number of bytes (count) it needs to transfer. Then it will go through the following

steps:

1. The peripheral device (such as the disk controller) will request the service of DMA by pulling DREQ (DMA request) high.
2. The DMA will put a high on its HLDR (hold request), signaling the CPU through its HOLD pin that it needs to use the buses.
3. The CPU will finish the present bus cycle and respond to the DMA request by putting high on its HLDA (hold acknowledge), thus telling the DMA controller that it can go ahead and use the buses to perform its task. HLDR must remain active high as long as DMA is performing its task.
4. DMA will activate DACK (DMA acknowledge), which tells the peripheral device that it will start to transfer the data.
5. DMA starts to transfer the data from memory to the peripheral by putting the address of the first byte of the block on the address bus and activating MEMR, thereby reading the byte from memory into the data bus; it then activates IOW (I/O Write) to write the data to the peripheral. Then DMA decrements the counter and increments the address pointer and repeats this process until the count reaches zero and the task is finished.
6. After the DMA has finished its job it will deactivate HLDR, signaling the CPU that it can regain control over its buses.

This above discussion indicates that DMA can only transfer information; unlike the CPU, it cannot decode and execute instructions. One could look at the DMA as a kind of CPU without the instruction decoder/executer logic circuitry. For the DMA to be able to transfer data it is equipped with the address bus, data bus, and control bus signals.

## Review Questions

1. True or false. When the DMA is working, the CPU is sitting idle.
2. True or false. When the CPU is working, the DMA is sitting idle.
3. True or false. No bus can serve two masters at the same time.

## Answers to Review Questions

### Section 13.1: CPU Memory Cycle Time

1.
  - (a)  $1/40 \text{ MHz} = 25 \text{ ns}$ ; therefore,  $2 \times 25 = 50 \text{ ns}$ ;
  - (b)  $1/50 \text{ MHz} = 20 \text{ ns}$ ; therefore,  $2 \times 20 \text{ (for 0 WS)} + 20 \text{ (1 WS)} = 60 \text{ ns}$ ;
  - (c)  $2 \times 15 \text{ ns} + 15 = 45 \text{ ns}$
2. It means that the CPU cannot access memory faster than every 50 ns.
3.
  - (a) The read cycle time is 75 ns; therefore, the effective working frequency is the same as 26.6 MHz of 0 WS ( $1/37.5 \text{ ns} = 26.6 \text{ MHz}$ ).
  - (b) The read cycle time is 60 ns; therefore, the effective working frequency is the same

as 33 MHz ( $1/30 \text{ ns} = 33 \text{ MHz}$ ).

4. A total of 50 ns is left for the memory access time.
5. Since  $2 + 1 \text{ WS} = 3$  clocks for each read cycle time,  $30 \text{ ns} (90/3 = 30)$  for the CPU clock duration; therefore, the CPU frequency is 33 MHz ( $1/30 \text{ ns} = 33 \text{ MHz}$ ).

## Section 13.2: DRAM Technology

1. SRAM and ROM
2. 100 ns
3. 60 ns
4.  $t_{RAC}$  (RAS access time),  $t_{RP}$  (RAS precharge time)
5. (a) There are two sets of 1M byte; therefore, each set consists of 4 banks of  $256\text{K} \times 9$  memory where each bank belongs to 1 byte of the D31–D0 data bus.  
(b) 2M
6. True
7.  $t_{PC}$
8. Total time =  $t_{RAC} + 99 \times t_{PC} = 60 + 99 \times 30 = 3030 \text{ ns}$
9. Static column
10. Often less than 100 MHz; many times it is only 66 MHz.
11.  $2/3 \times 100 \text{ MHz} = 66 \text{ MHz}$ . Now  $1/66 \text{ MHz} = 15 \text{ ns}$ .  $2 \times 15 \text{ ns} = 30 \text{ ns}$  read cycle time.
12. The  $t_{PC}$  (page cycle time)
13. False
14. SDRAM
15. True

## Section 13.3: Data Integrity in RAM and ROM

1. Adding the bytes:  $24 + 76 + F5 + 98 + 89 + 7A + 61 + C2 = 44D$ . Dropping the carries, we get 4D, and taking the 2's complement, we have B3 for the checksum byte.
2. Parity bit generation/checker, checksum
3. While the computer is on, any corruption in the contents of RAM is detected by the parity bit error checking circuitry when that data is accessed (read) again. However, the ROM corruption is not detected since the checksum detection is performed only when the system is booted.

## Section 13.4: Concept of DMA

1. True
2. True
3. True



# Chapter 14: Cache Memory

There are different memories in a computer. Among them are hard disk, RAM and Cache. The hard disk has a huge space that accommodates all programs and files. But it is too slow to provide data directly to the CPU. In contrast, cache works as fast as the CPU but has a small amount of memory.

As an analogy, you have different spaces in your house; storeroom, bookcase and cupboards, and your desk. You work on your desk but it has a small space. In contrast, storeroom has a huge space but takes you a while to bring tools from it. If you are using your desk to make a circuit, you put the needed tools on your desk and when the work is finished you free up your desk to use for another purpose. See Figure 14-1.

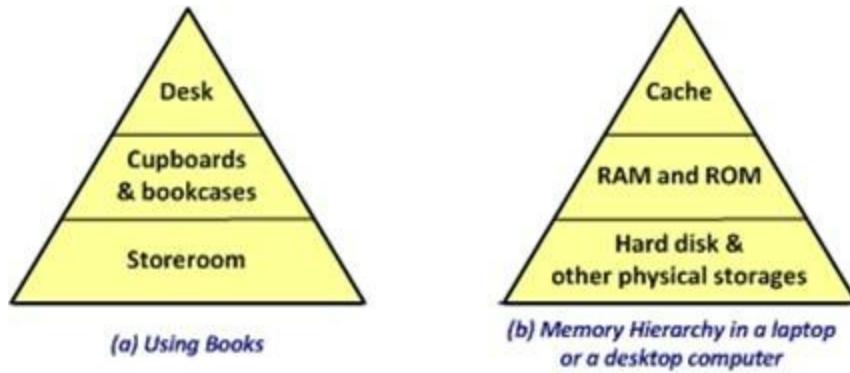


Figure 14-1: Memory Hierarchy

In the same way, there is a huge amount of data in your hard disk, CDs, and DVDs. When you click on a program, the OS (Operating System) brings it into the RAM and a small amount of it is brought into the Cache to be executed by the CPU. When you run a program, e.g. the Keil IDE, you do not use the whole parts of the program all at the same time. So, just a small portion of the program is brought to the Cache to be executed. See Figure 14-2.

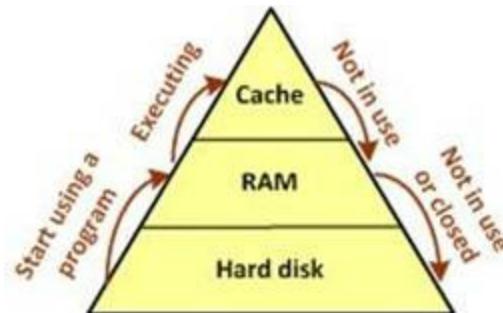


Figure 14-2: a Program Moving in the Memory Hierarchy

The potential power of high-performance microprocessors can be exploited only if memory is fast enough to respond to the microprocessor's need to fetch code and data. There is no use in choosing a fast processor and then interfacing it with slow memory. Many of the ARM chips come with on-chip cache. In this chapter, we deal with issue of cache memory. In Section 14.1, the cache memory organizations are discussed. In Section 14.2, some concepts and terminologies related to cache memory are examined. The cache memory of ARM and its multicore features are examined, as well.

## Section 14.1: Cache Memory Organizations

The most widely used memory design for high-performance CPUs implements DRAMs for main memory along with a small amount (compared to the size of main memory) of SRAM for cache memory. This takes advantage of the speed of SRAM and the high density and cheapness of DRAM. To implement the entire memory of the computer with SRAM is too expensive and to use all DRAM degrades performance. Cache memory is placed between the CPU and main memory. See Figure 14-3.

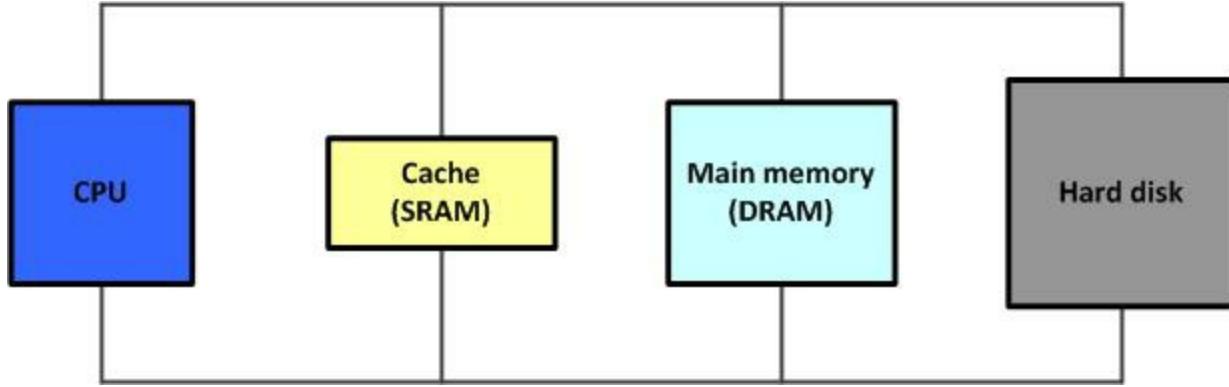


Figure 14-3: CPU and Its Relation to Various Memories

When the CPU initiates a memory access, it first asks cache for the information (data or code). If the requested data is there, it is provided to the CPU with zero wait states (WS), but if the data is not in cache, the memory controller circuitry will transfer the data from main memory to the CPU while giving a copy of it to cache memory. In other words, at any given time the cache controller has knowledge of which information (code or data) is kept in cache; therefore, upon request for a given piece of code or data by the CPU the address issued by the CPU is compared with the addresses of data kept by the cache controller. If they match (hit) they are presented to the CPU with zero WS, but if the needed information is not in cache (miss) the cache controller along with the memory controller will fetch the data and present it to the CPU in addition to keeping a copy of it in cache for future reference. The reason a copy of data (or code) fetched from main memory is kept in the cache is to allow any subsequent request for the same information to result in a hit and provide it to the CPU with zero wait states. If the requested data is available in cache memory, it is called a hit; otherwise, if the data must be brought in from main memory, it is a miss.

It must be noted that when the CPU accesses memory, it is most likely to access the information in the vicinity of the same addresses, at least for a time. This is called the *principle of locality of reference*. In other words, even for a short program of 50 bytes, the CPU is accessing those 50 memory locations from cache with zero wait states. If it were not for this principle of locality and the fact that the CPU accesses memory randomly, the idea of cache would not work. This implies that branch and call instructions are bad for the performance of cache-based systems. The hit rate, the number of hits divided by the total number of tries, depends on the size of the cache, how it is organized (cache organization), and the nature of the program.

### Cache organization

There are three types of cache organization:

1. fully associative
2. direct mapped
3. set associative

The following is a discussion of each organization with its advantages and disadvantages. For the sake of clarity and simplicity, an 8-bit data bus and a 16-bit address bus are assumed.

### Fully associative cache

In fully associative cache, only a limited number of bytes from main memory are held by cache along with their addresses. The SRAMs holding data are called *data cache* and the SRAMs holding addresses of the data are called *tag cache*. This discussion assumes that the microprocessor is sending a 16-bit address to access a memory location that has 8 bits of data and that the cache is holding 128 of the possible 65,536 ( $2^{16}$ ) locations. This means that the width of the tag is 16 bits since it must hold the address, and that the depth is 128. When the CPU sends out the 16-bit address, it is compared with all 128 addresses kept by the tag. If the address of the requested data matches one of the addresses held by the tags, the data is read and is provided to the CPU (a hit). If it is not in the cache (a miss), the requested data must be brought in from main memory to the CPU while a copy of it is given to cache. When the information is brought into cache, the contents of the memory locations and their associated addresses are saved in the cache (tag cache holds the address and data cache holds the data).

In fully associative cache, the more data that is kept, the higher the hit rate. An analogy is that the more books you have on a table, the better the chance of finding the book you want on the table before you look for it on the book shelf. The problem with fully associative is that if the depth is increased to raise the hit rate, the number of comparisons is inefficient. For example, a fully associative cache with a depth of 1024 requires 1024 comparisons, and that is too time consuming or needs a huge circuit to compare them all in parallel. On the other hand, with a depth of 16 the CPU ends up waiting for data too often. This is because the operating system is swapping information in and out of cache, since its size is too small. This replacement policy is discussed later. In the above example of 128 depth, the amount of SRAM for tag is  $128 \times 16$  bits and  $128 \times 8$  for data, that is, 256 bytes for tag and 128 bytes for data cache for a total of 384 bytes. Although the above example used a total of 384 bytes of SRAM, it is said that the system has 128 bytes of cache. In other words, the data cache size is what is advertised. The SRAM inside the cache controller provides the space for storing the tag bits. Tag bits are not included in cache size. In Figure 14-4, DRAM location F992 contains data 0x85. The left portion of the figure shows when the data is moved from DRAM to cache.

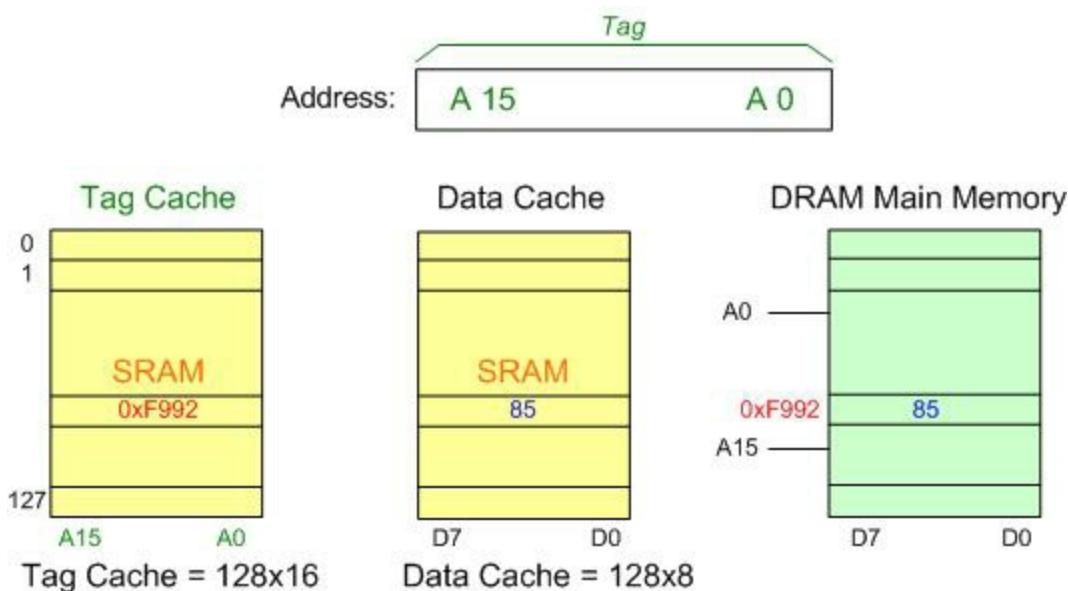
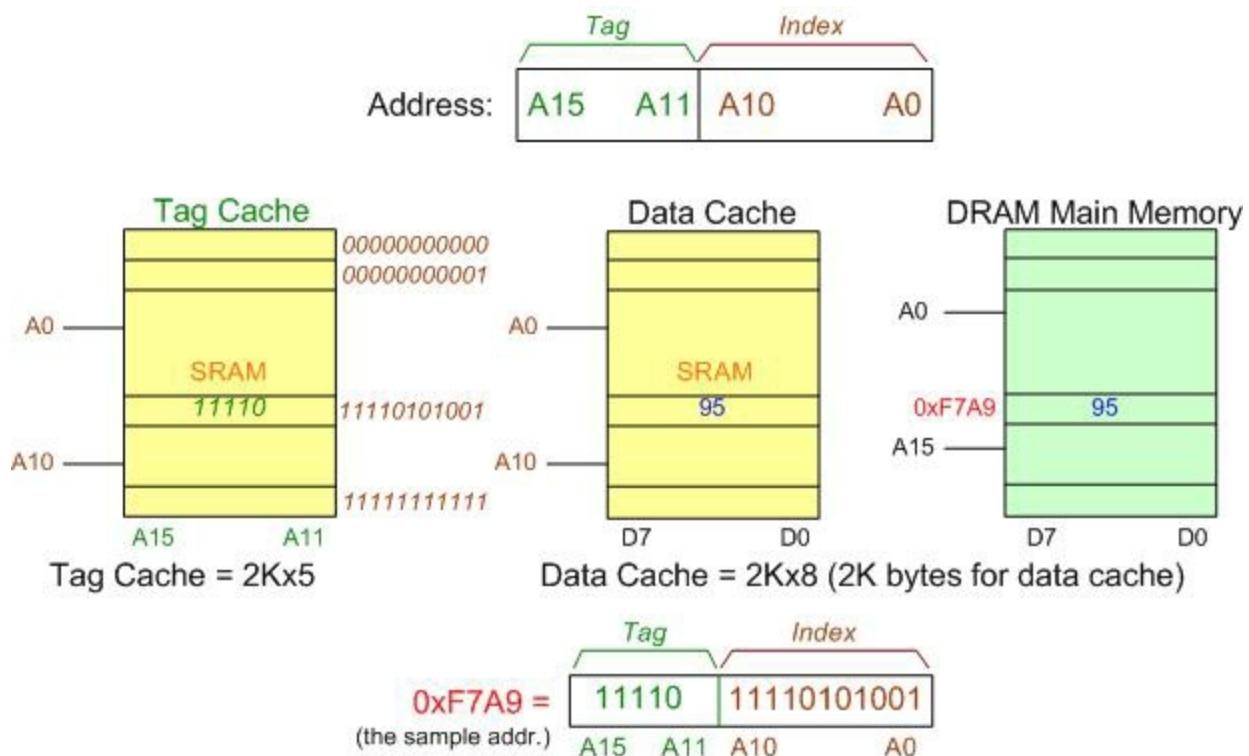


Figure 14-4: a 128-Byte Fully Associative Cache for a 16-bit System

## Direct-mapped cache

Direct-mapped cache is the opposite extreme of fully associative. It requires only one comparison. In this cache organization, the address is divided into two parts: the *index* and the *tag*. The index is the lower part of the address, which is directly mapped into SRAM, while the upper part of the address is held by the tag SRAM.

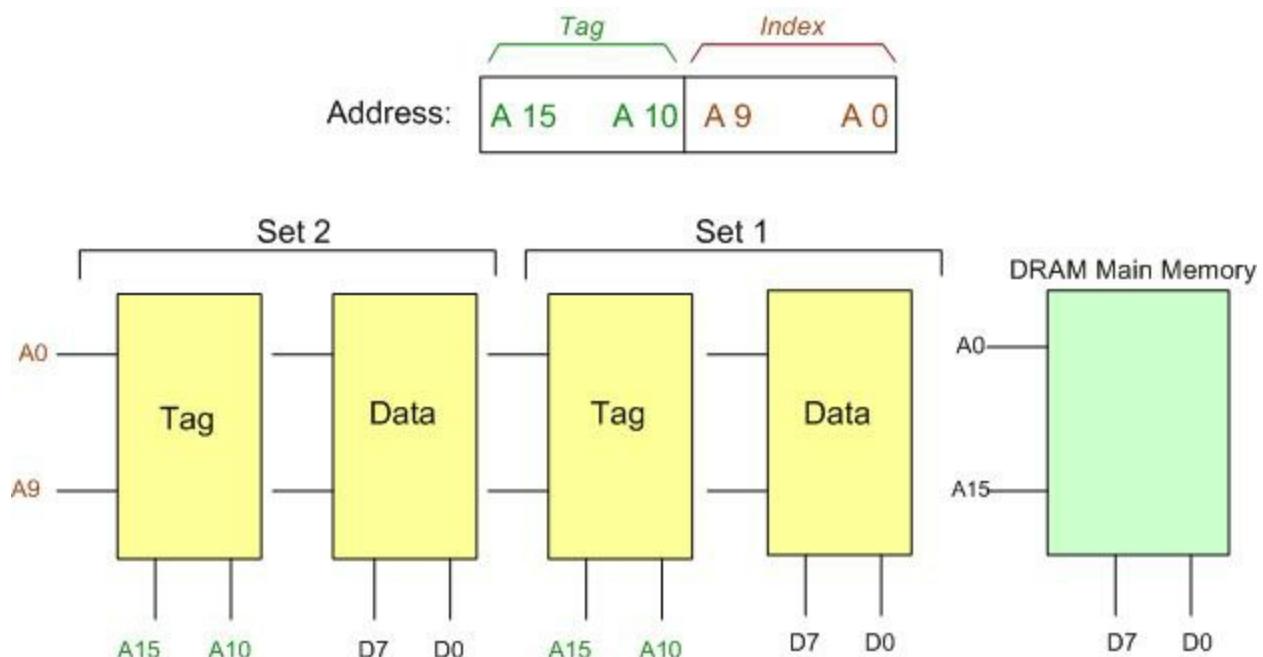
To provide a 1024 byte direct-mapped cache to a 16-bit system, A0 to A10 are the index and A11 to A15 are the tag. Assuming that CPU addresses location 0xF7A9, the 7A9 goes to the index but the data is not read until the contents of tag location 7A9 is compared with 11110B. If it matches (its content is 11110), the data is read to the CPU; otherwise, the microprocessor must wait until the contents of location F7A9 are brought from main memory DRAM into the CPU while a copy of it is issued to cache for future reference. There is only one unique location with index address of 7A9, but 32 possible tags ( $2^5 = 32$ ). Any of these possibilities, such as C7A9, 27A9, or 57A9, could be in tag cache. In such a case, when the tag of a requested address does not match the tag cache, a cache miss occurs. Although the number of comparisons has been reduced to one, the problem of accessing information from locations with the same index but different tag, such as F7A9 and 27A9, is a drawback. The SRAM requirement for this cache is shown below. While the data cache is 2K bytes, the tag requirement is  $2K \times 5 = 10K$  bits or about 1.25K bytes. See Figure 14-5.



**Figure 14-5: a 2KB Direct-Mapped Cache for a 16-bit System**

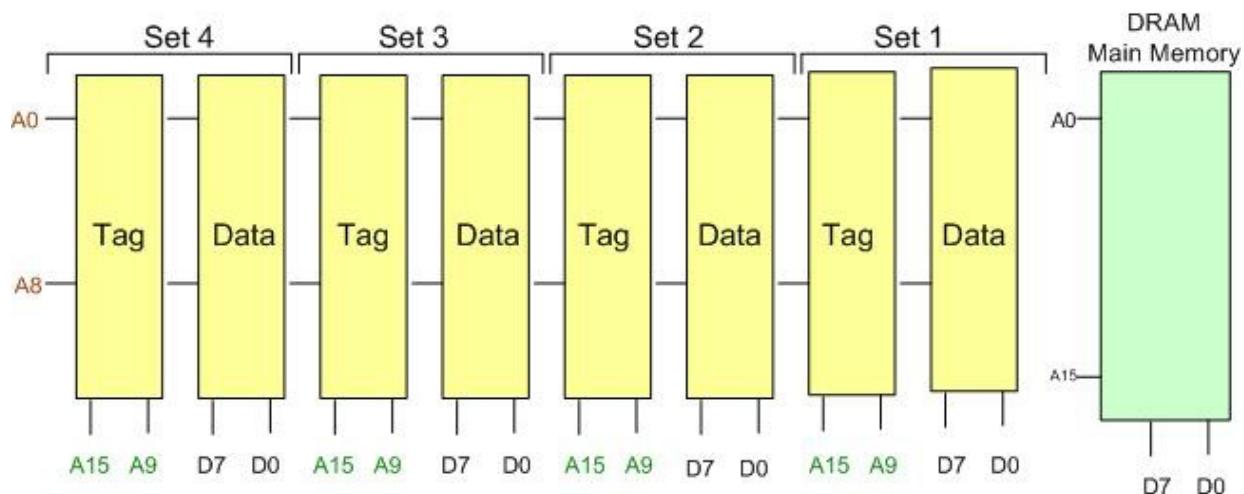
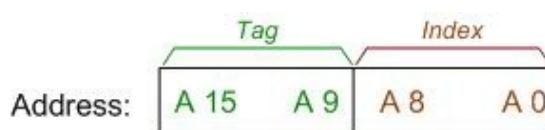
## Set associative

This cache organization is in between the extremes of fully associative and direct mapped. While in direct mapped there is only one tag for each index, in set associative, the number of tags for each index is increased, thereby increasing the hit rate. In 2-way set associative, there are two tags for each index, and in 4-way there are 4 tags for each index. See Figures 14-6 and 14-7.



[Tag = 1k x 6    Data = 1k x 8] for each set (2k bytes for data cache)

**Figure 14-6: a 2KB Two-way Set Associative Cache for a 16-bit System**



[Tag = 512 x 7      Data = 512 x 8] for each set (2k bytes for data cache)

**Figure 14-7: a 2KB Four-way Set Associative Cache for a 16-bit System**

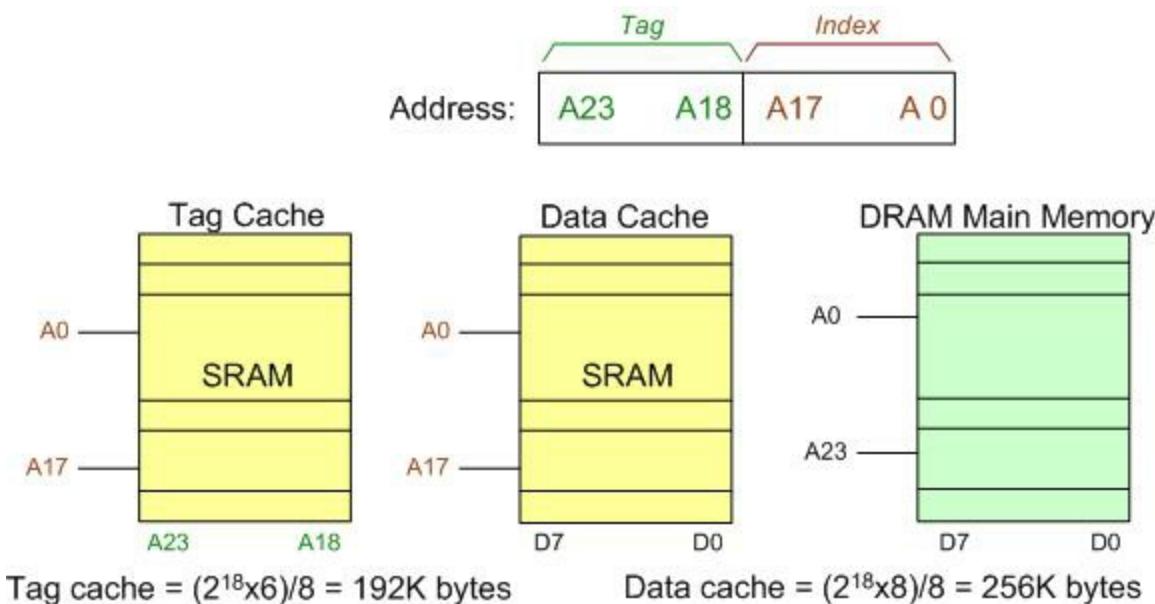
Comparing direct-mapped and 2-way set associative, one can see that with only a small amount of extra SRAM, a better hit rate can be achieved. In this organization, if the microprocessor is requesting the contents of memory location 0x41E6, there are 2 possible tags that could hold it, since cache circuitry will access index 0xE6 and compare the contents of both tags with "0100 00". If any of them matches it, the data of index location E6 is read to the CPU, and if none of the tags matches "0100 00", the miss will force the cache controller to bring the data from DRAM to cache, while a copy of it is provided to the CPU at the same time. In 4-way set associative, the search for the block of data starting at 41E6 is initiated by comparing the 4 tags with "0100 000", which will increase the chance of having the data in the cache by 50%, compared with 2-way set associative. As seen in the above example, the number of comparisons in set associative depends on the degree of associativity. It is 2 for 2-way set associative, 4 for 4-way set associative, 8 for 8-way, n for n-way set associative, and in the thousands for fully set associative. The higher the set, the better the performance, but the amount of SRAM required for tag cache is also increased, making the 8-way and 16-way associatives' increased costs unjustifiable compared to the small increase in hit rate. The increase in the set also increases the number of tag comparisons. Most cache systems that use this organization are implemented in 4-way set associative.

From a comparison of these two cache organizations, the difference between them in organization and SRAM requirements can be seen. In 2-way, the tag of 1K x 6 and data of 1K x 8 for each set gives a total of 14K bits [2 x (1K x 6 + 1K x 8) = 28K bits]. In 4-way, there is 512 x 7 for the tag and 512 x 8 for data, giving a total of 32K bits [(512 x 7 + 512 x 8) x 4 = 32K bits] of SRAM requirement. Only with an extra 4K bits the hit rate improves substantially. As the degree of associativity is increased, the size of the index is reduced and added to the tag and this increases the tag cache SRAM requirement, but the size of data cache remains the same for all cases of direct map, 2-way, and 4-way associative. These concepts are clarified further in

Examples 14-1, 14-2, and 14-3.

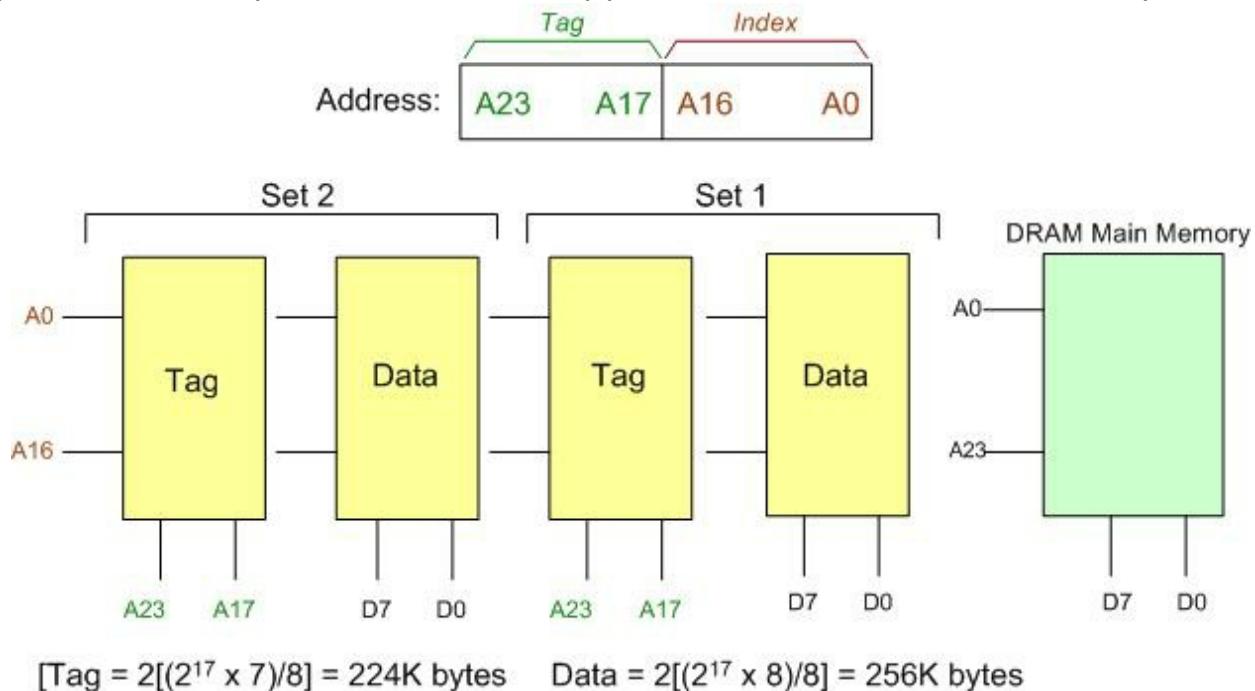
### Example 14-1

This example shows directed-mapped cache for 16M main memory.



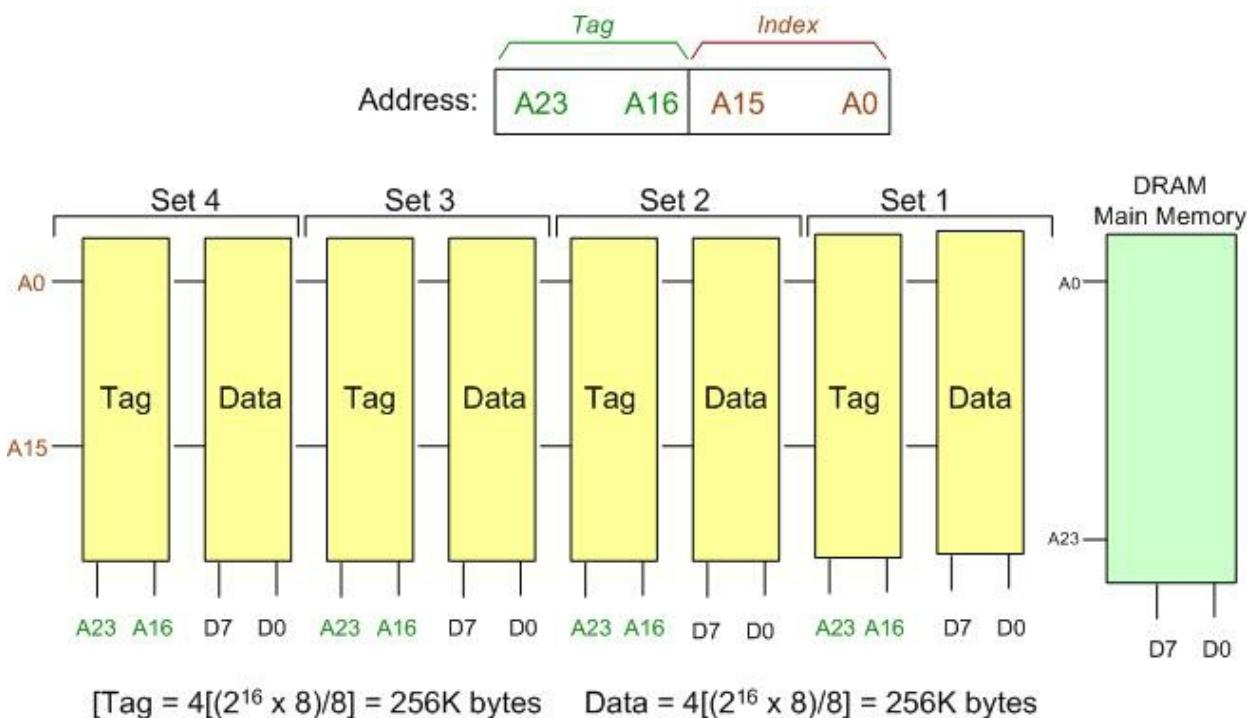
### Example 14-2

This example shows 2-way set associative mapped cache for 16M main memory.



### Example 14-3

This example shows 4-way set associative mapped cache for 16M main memory.



## Review Questions

1. Cache is made of \_\_\_\_\_ (DRAM, SRAM).
  2. From which does the CPU ask for data first, cache or main memory?
  3. Rank the following from fastest to slowest as far as the CPU is concerned.  
(a) main memory      (b) register      (c) cache memory
  4. In fully associative cache of 512 depth, there will be \_\_\_\_\_ comparisons for each data request.
  5. Which cache organization requires the least number of comparisons?
  6. A 4-way set associative organization requires \_\_\_\_\_ comparisons.

## Section 14.2: Cache Memory and Multicore Systems

In systems with cache memory, there must be a way to make sure that no data is lost and that no stale data is used by the CPU, since there could be copies of data in two places associated with the same address, one in main memory and one in cache. A sound policy on how to update main memory will ensure that a copy of any new data written into cache will also be written to main memory before it is lost since the cache memory is nothing but a temporary buffer located between the CPU and main memory. To prevent data inconsistency between cache and main memory, there are two major methods of updating the main memory: (1) write-through and (2) write-back. The difference has to do with main memory traffic.

### Write-through

In write-through, the data will be written to cache and to main memory at the same time. Therefore, at any given time, main memory has a copy of valid data contained in cache. At the cost of increasing bus traffic to main memory, this policy will make sure that main memory always has valid data, and if the cache is overwritten, the copy of the latest valid data can be accessed from main memory. See Figure 14-8.

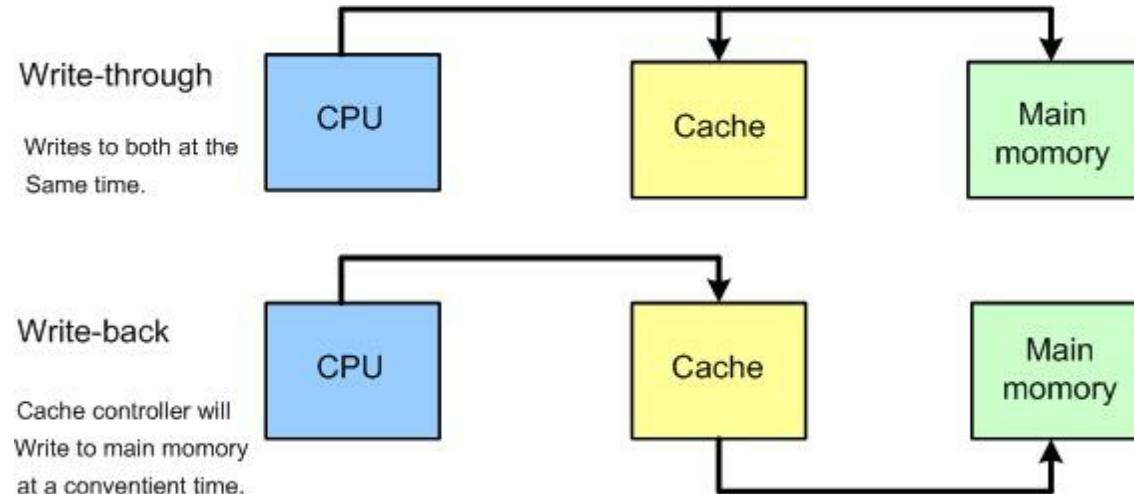


Figure 14-8: Method of Updating Main Memory

### Write-back (copy-back)

In the write-back (sometimes called copy-back) policy, a copy of the data is written to cache by the processor and not to main memory. The data will be written to main memory by the cache controller only if cache's copy is about to be replaced with another data. The cache has an extra bit called the dirty bit (also called the altered bit). If data is written to cache, the dirty bit is set to 1 to indicate that the cache data is new data that exists only in cache and not in main memory. At a later time, the cache data is written to main memory and the dirty bit is cleared. In other words, when the dirty bit is high it means that the data in cache has changed and is different from the corresponding data in main memory; therefore, the cache controller will make sure that before erasing the new data in cache, a copy of it is given to main memory. Getting rid of information in cache is often referred to as cache flushing. This updating of the main memory at a convenient time can reduce the traffic to main memory so that main memory buses are used only if cache has been altered. If the cache data has not been altered

and is the same as main memory, there is no need to write it again and thereby increase the bus traffic as is the case in the write-through policy. See Figure 14-8.

## Cache coherency

In systems in which main memory is accessed by more than one processor (DMA or multiprocessors), it must be ensured that cache always has the most recent data and is not in possession of old (or stale) data. In other words, if the data in main memory has been changed by one processor, the cache of that processor will have the copy of the latest data and the stale data in the cache memory is marked as dirty (stale) before the processor uses it. In this way, when the processor tries to use the stale data, it is informed of the situation. In cases where there is more than one processor and all share a common set of data in main memory, there must be a way to ensure that no processor uses stale data. This is called cache coherency.

## Cache replacement policy

What happens if there is no room for the new data in cache memory and the cache controller needs to make room before it brings data in from main memory? This depends on the cache replacement policy adopted. In the LRU (least recently used) algorithm, the cache controller keeps account of which block of cache has been accessed (used) the least number of times, and when it needs room for the new data, this block will be swapped out to main memory or flushed if a copy of it already exists in main memory. This is similar to the relation between virtual memory and main memory. The other replacement policies are to overwrite the blocks of data in cache sequentially or randomly, or use the FIFO (first in, first out) policy. Depending on the computer's design objective and its intended use, any of these replacement policies can be adopted.

## Cache fill block size

If the information asked for by the CPU is not in cache and the cache controller must bring it in from main memory, how many bytes of data are brought in whenever there is a miss? If the block size is too large (let's say 5000 bytes), it will be too slow since the main memory is accessed normally with 1 or 2 WS. At the other extreme, if the block is too small, there will be too many cache misses. There must be a middle-of-the-road approach. The block size transfer from the main memory to CPU (and simultaneous copy to cache) varies in different computers, anywhere between 32 and 512 bytes. If the block size is 32 bytes, then it is called the 8-line cache refill policy, where each line is 4 bytes of the 32-bit data bus.

## Moore's Law

In the mid 1960s, Intel cofounder Gordon Moore made the following astounding prediction: "The number of transistors that would be incorporated on a silicon die would double every 18 months for the next several years." Examining some of the chips on the market shows how this prediction has come. In recent years the number of gates on a single chip has reached to over a billion gates. Many vendors of ARM are using a large number of gates to incorporate features such as cache and multicore into a single chip. In this part we will examine modern CPUs and their caches.

## Level 1, Level 2, and Level 3 caches

In many new CPUs, the concept of level 2 (L2) cache is being introduced. In such processors, we have few K bytes of cache for code (instruction) and another few K bytes of cache for data, feeding code and data to the fetch unit. This is called level 1 (L1) cache. See Figure 14-9. Many of the new CPUs also have L2 cache. While L1 cache feeds code (instruction) and data into the fetch and execution units and is part of the inner working of the CPU, the Level 2 cache is sitting outside the CPU die but still on the same package as the CPU itself. Since the L1 cache is on the same die as the CPU, it works at the same clock speed as the CPU. For example, if a given ARM has clock speed of 800 MHz, then the L1 cache feeds the CPU information at that speed. L2 cache works at a fraction of the CPU speed. For example, if a given ARM has clock speed of 1 GHz, then the L2 cache feeds the CPU information at 200 MHz. When an ARM with L2 cache brings in code and data from externally located DRAM memory, it places them in L2 cache. Then the memory management unit of the core CPU brings in the information from the L2 cache and separates the code and data, placing each in data or code L1 caches (Harvard architecture). See Figure 14-9.

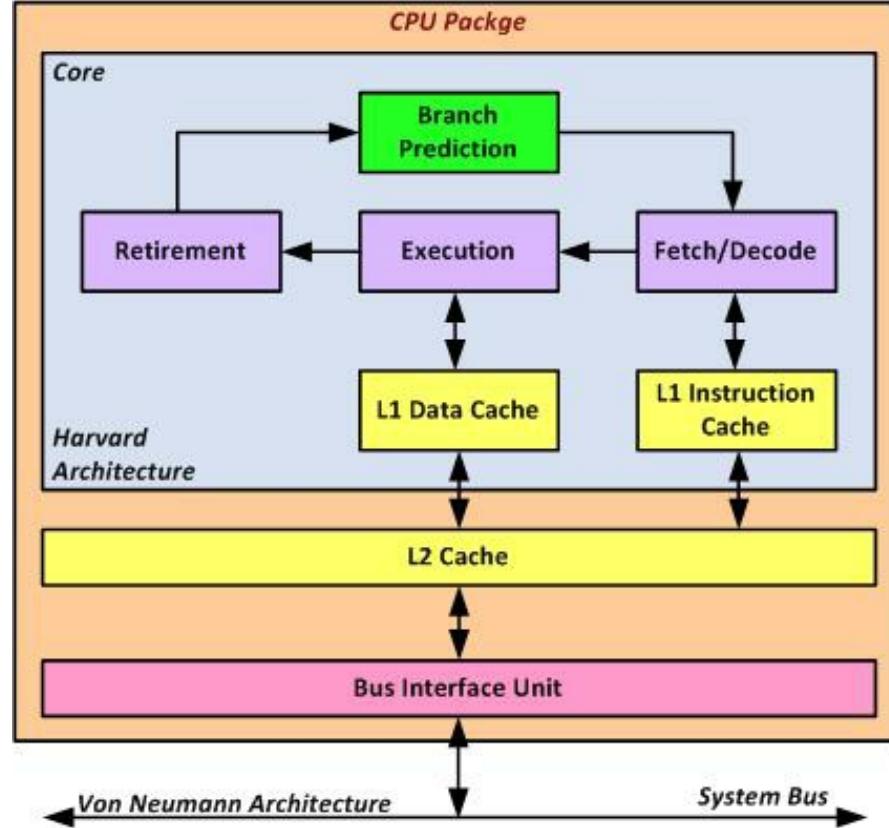


Figure 14-9: L2 Cache Feeding Code and Data to L1 Caches

Notice that code and data caches are separate, which is not the case with the L2 cache. L2 cache is unified cache meaning that the cache is used for both code and data. The amount allocated to data and code varies dynamically, depending on the nature of the program being run. If the program being run is more data intensive, then more of the L2 cache is allocated to data. With CPU speed rising above 1 GHz, the biggest problem is external (that is, external to the CPU chip) memory access time. For that reason, in some high-performance ARM-base systems for Windows and Linux the designers place level 3 (L3) cache outside the CPU on the motherboard to speed up the external memory access. This L3 cache is sitting between the CPU chip and DRAM memory module on the motherboard. See Figures 14-10.

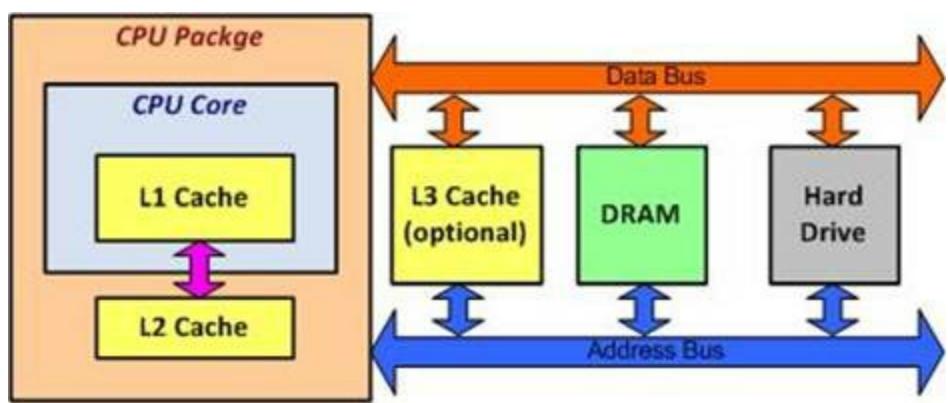


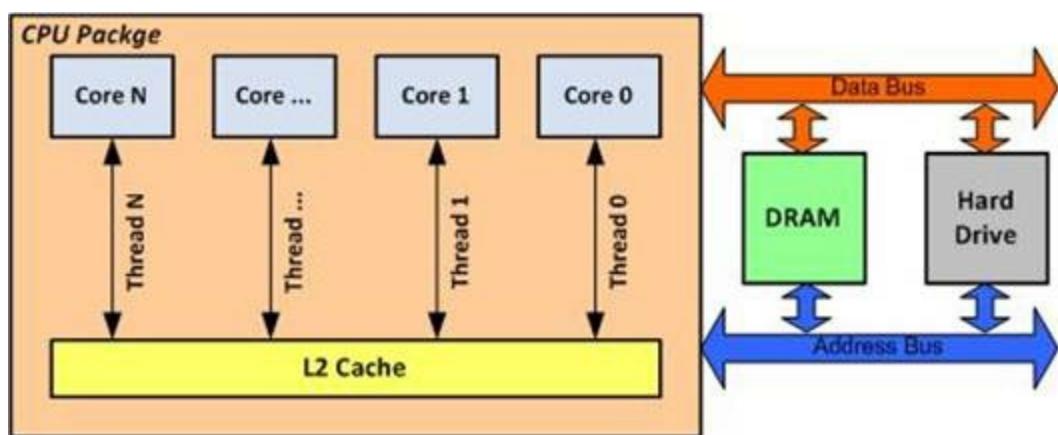
Figure 14-10: L1, L2, and L3 Cache

In some new ARM processors with multiple cores there is L2 cache on the same package as the CPU, but located outside the CPU die. In such a processor one can summarize the role of L1–L3 caches as follows:

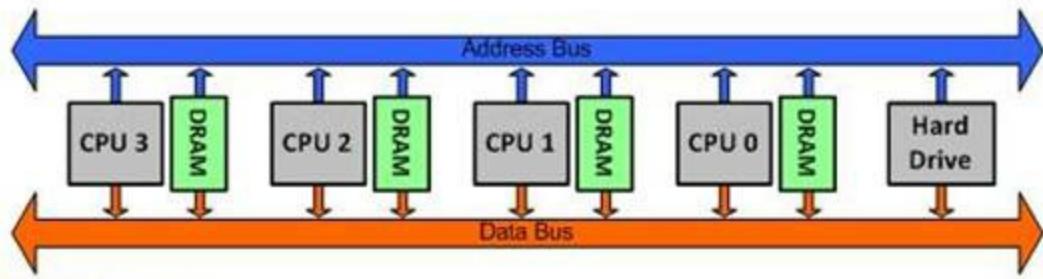
1. The speed of the L1 cache is the same as the CPU speed, since it must feed the CPU as fast as the instructions are executed.
2. L2 cache works at a fraction of the CPU since it is on the same package as the CPU.
3. L3 cache works at a fraction of the speed of the bus since it is located outside the CPU package.

## Hyper-Threading

In the new CPUs, the concept of multithreaded execution is being introduced. First, the definition of thread: It is a series of parallel programs that can run on different CPUs simultaneously. In the multiprocessor environment, each program is given its own CPU and memory. Vendors place multiple CPUs into a single chip and call it hyper-threading. Therefore, hyper-threading in its simplest form is to allow a single CPU to execute two or more threads of code simultaneously. Of course, to do that the CPU must be equipped with internal logic and resources to execute the threads. The early ARM CPUs were not equipped with hyper-threaded technology, since it requires large amounts of transistors to duplicate many of the resources inside the CPU. As far as the operating system is concerned, the CPU with hyper-threaded capability appears to be multiple logical CPUs inside a single physical CPU. Therefore, to take full advantage of hyper-threading technology, both the operating system and the application must be rewritten (or reconfigured) to make them threaded-aware. The ideal situation in the multithreaded environment is to write the application programs so that threads can execute independently of each other. However, that is not the case in the real world. Since both logical processors inside the hyper-threaded CPU use the same bus to access memory, they can get in each other's way and slow down program execution. Figure 12-11 shows the system bus access for the threaded CPU and multiprocessing. Note that in threaded CPUs, internal logical CPUs must share the system bus access. This is in contrast to using multiple processors in which each CPU has its own access to the system bus.



(a) a Hyper-Threaded CPU



(b) Multiple Processors

Figure 14-11: Hyper-threaded CPU vs. Multiple Processors

In computer architecture literature the words threads and tasks are used interchangeably. However, there is a difference between a task and a thread. In multitasking you are running multiple tasks such as playing music, typing into a word processor, and running a virus scan all on a single CPU. In multitasking the CPU switches from one task to another in a round robin (circular) fashion, giving each task a slice of the CPU's time. In contrast, true multithreading attempts to parallelize the execution of a single program in order to speed up the execution of that program. Not all applications lend themselves to parallelization and that is the reason that not all programs benefit equally from multithreaded CPUs. For more discussion of multithreading and multitasking, see the following article:

<http://arstechnica.com/articles/pedia/cpu/hyperthreading.ars>

## Multicore Technology

Many newer-generation of CPUs have what is called multicore technology. Multicore packs two or more independent microprocessors (called cores) into a single chip. At this time, many vendors are introducing the ARM chips with dual-core and quad-core features. Many of them are working on processors with 8 cores. In the dual-core CPU, almost everything is doubled, which is like putting two physical CPUs into a single chip. The difference between multicore and multiprocessor CPUs is that in the multicore CPU there is one pathway to the system memory for the CPU while in the multiprocessor CPUs each processor has its own memory space independent of the others. See Figure 14-12.

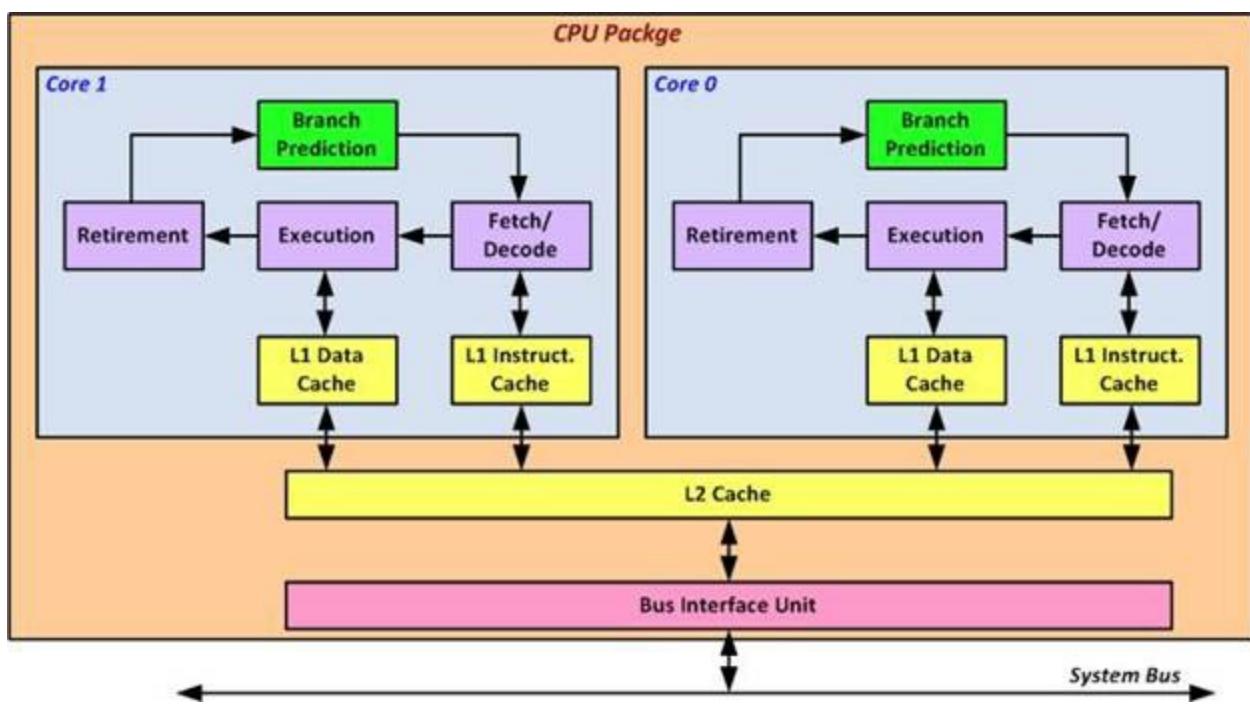


Figure 14-12: An Example of Dual-Core Processor

## Review Questions

1. What does write-through refer to?
2. Which one increases the bus traffic, write-through or write-back?
3. What does LRU stand for, and how is it used?
4. What does cache refill policy of 4 lines refer to?
5. True or false. Some ARM chips come with on-chip L1 cache.
6. True or false. All ARM chips come with on-chip L2 cache.
7. True or false. All ARM chips have the hyper-threading feature.
8. What is the difference between multicore and multiprocessor?

## Answers to Review Questions

### Section 14.1

1. SRAM
2. Cache
3. Register, cache, and main memory
4. 512
5. Direct map
6. 4

### Section 14.2

1. The CPU writes to cache and main memory at the same time when updating main memory.
2. Write-through
3. LRU (least recently used) is a cache replacement policy. When there is a need for room in the cache memory the cache controller flushes the LRU data to make room for new data.
4. When the cache is filled with new data, it is done a minimum of 4 lines ( $4 \times 4 = 16$  bytes)

at a time.

5. True
6. False
7. False
8. In multicore, we have several CPUs inside a single chip accessing the same main memory space, while in multiprocessor, each CPU has its own memory space.



## Chapter 15: MMU, Virtual Memory and MPU in ARM

Many of the ARM chips come with on-chip MMU (memory management unit). The MMU is responsible for the virtual memory. Using the ARM for operating systems such as Linux and Microsoft Windows, one needs the virtual memory. This chapter examines the concepts of virtual memory. Section 15.1 provides an introduction to virtual memory of the ARM and its MMU. In Section 15.2, some of the registers of MMU responsible for operation and access permission are discussed. An overview of MPU (memory protection unit) of ARM is provided in Section 15.3.

## SECTION 15.1: MMU and Virtual Memory in ARM

Some of the high end of ARM microprocessors such as ARM9 come with on-chip MMU allowing to implement virtual memory for operating systems such as Linux and Microsoft Windows. Due to complexity associated with the virtual memory many chapters are dedicated to it in operating system books. In this chapter, we simply provide an overview of the resources available in ARM for the implementation of virtual memory and memory protection.

### What is virtual memory?

A CPU with virtual memory is fooled into thinking that it has access to an unlimited amount of physical memory. DRAM primary memory is also called main memory or physical memory. In this scheme, every time the CPU looks for certain information (code or data), the operating system will first search for it in main DRAM memory and if it is not there, it will bring it into RAM from secondary memory (hard disk or flash memory). What happens if there is no room in RAM? It is the job of the operating system to swap information out of RAM and make room for new data. Which data will be swapped out depends on how the operating system is designed. Some operating systems use the LRU (least recently used) algorithm to swap data in and out of primary memory (DRAM). In the LRU method, the operating system keeps account of which data has been used the least number of times in a certain period, and when there is need for room it will swap out the least recently used data to hard disk to make room for the new data. The total amount of RAM on an ARM-based computer could be maximum of 4G with a hard disk capacity of 500G bytes, but the CPU is fooled into thinking that it has access to all 500G of memory (or just the amount of the swap space allocated on the disk.) Among the operating systems, Microsoft Windows 2000, XP, Vista, Windows 7 and 8, all the variations of Unix and Linux (including Android), Sun Microsystems' Solaris, and Apple Mac OS X use the capability of the CPU's virtual memory.

In systems without virtual memory, only one task can be active at a time and all other tasks are sitting idle (dormant). In multitasking operating systems such as Microsoft Windows and Linux each task is given a slice of the CPU's time, and many tasks can be active concurrently. For example, a word processor can be used while the web browser is receiving and sending data via network card, a spreadsheet program is doing some calculations, and an MP3 player is playing music. Of course, since there is only one microprocessor taking care of all these tasks, it is the job of the multitasking operating system to slice the CPU time and assign each task time on a circular rotational basis. If there are too many tasks and all are active, they all seem to be slow since each task gets less time (attention) from the CPU. Of course, one way to solve this slowness is to use high-performance CPUs with GHz speed. The multitasking operating system can be cooperative or preemptive. In cooperative multitasking, two or more applications cooperate with each other in taking turns to use the CPU alternately. If one application misbehaves, it can cause the whole system to be unstable and crash. In preemptive multitasking, a task can be interrupted preemptively at any point by another program. If a task is interrupted by another task, its present state will be saved by the operating system and it will be serviced after the new task is given a chance to use the CPU. In the multitasking operating system it is the job of the OS to make sure the tasks are available in the DRAM for

the CPU to execute them. Since we have limited DRAM space and lots of tasks in hard disk, it is the OS that makes sure the tasks are swapped in and out of DRAM when it runs out of the space in DRAM. To make the job of OS easier in implementing virtual memory, the CPUs have what is called memory management unit (MMU). This on-chip MMU is available only in high-end CPUs. Many ARM chips used in embedded products do not have the on-chip MMU for the virtual memory implementation. The ARM9 chips have on-chip MMU.

## Segmentation vs. paging

To implement virtual memory, two methods are used: segmentation and paging. In segmentation, the size of the data swapped in and out can vary from few hundred bytes to a few megabytes. In paging, the size is a multiple of one page. Although the page can be 1K, 4K, 64K, or 1M bytes, but unlike segmentation, its size is fixed. We use 4K bytes page for sake of simplicity.

In segmentation the whole segment of a program goes to memory next to each other. When the segmentation is used after some memory allocation and deletion, the available memory becomes fragmented into small sections of varied sizes; and the operating system must continuously move contents of memory around to make room for the new segment, which could be any size. Paging is used widely since it prevents memory fragmentation. Paging makes the job of the operating system much easier since all the processes will be a multiple of 4K bytes. If the size of a process is not a multiple of 4K bytes (which is the case most of the time), the operating system will leave the unused portion empty and the next file will be placed on a 4K boundary. This is similar to the cluster in hard disks. The disk allocates memory to each file in clusters. For example, if 4 sectors are used for each cluster, each cluster can store 2048 ( $4 \times 512$ ) bytes per sector. If a given file is 12,249 bytes, the operating system will assign a total of 7 clusters or 14,168 ( $7 \times 2048 = 14,168$ ) bytes. All bytes between 12,249 and 14,168 are unused. This results in wasting some memory space on the disk but at the same time makes the design of the disk controller and operating system much easier. This concept applies as well to the paging method of virtual memory as far as the allocation of main memory (DRAM) to data and code is concerned. See Figure 15-1. One can briefly define the segmentation and paging virtual memory mechanisms in the following statement. In segmentation virtual memory, the process can be any byte size, located anywhere it can fit into main memory. In paging virtual memory, the process is always a multiple of 4096 bytes and located on a 4K-byte boundary in main memory.

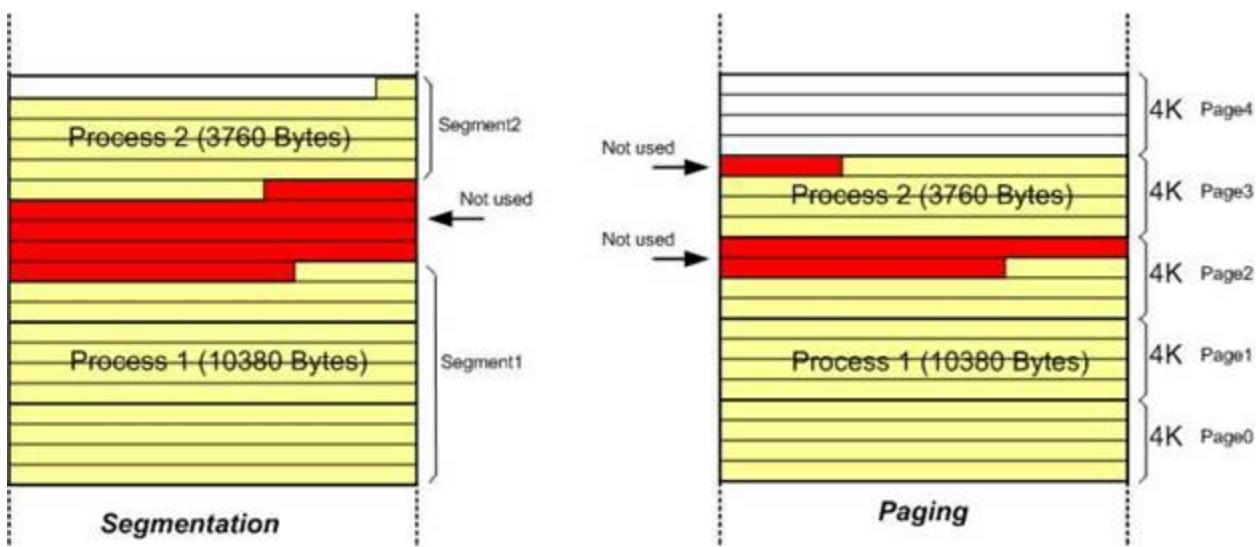


Figure 15-1: Paging vs. Segmentation

All high-performance RISC microprocessors such as ARM have paging virtual memory and very few use the segmentation method any more. The x86 has both segmentation and paging options. The reason that x86 Pentium processors supports segmentation (in addition to paging) is that they had to stay compatible with the 80286 microprocessor.

## Paging sizes in ARM

In paging virtual memory, main memory is divided into fixed 4K-byte chunks. The ARM supports the page sizes 1K (tiny), 4K (small), 64K(large), or 1M(section) bytes. Some recent ARM chips no longer support the 1K (tiny) page size. The 1M byte page size is available for use in graphics intensive applications. In this chapter we only examine the 4K(small) and 1M(section) page sizes. If a given piece of code or data is not present in main memory, the operating system brings it into main memory from the hard disk, 4Kbyte at a time. Next, the terms virtual address and physical address in the ARM are contrasted.

## Physical and virtual address

As discussed in previous chapters, physical addresses for an instruction in the ARM is the value held by the program counter (R15) register. The physical memory is the Flash ROM, SRAM, I/O ports, and DRAM memory accessed by the CPU. As we have seen, the 4GB memory has addresses in the range of 0x00000000 to 0xFFFFFFFF. In virtual memory, however, the physical address of blocks of data or code is held by look-up tables. Among the information held by this look-up table, in addition to the physical address of the code or data, is access permission (AP). This provides the ARM system with a protection mechanism. The lack of protection of the operating system or users' programs is one of the weaknesses of CPUs such as 8088/8086 used in the first IBM PC in 1981. This weakness is due to the inability of these processors to block general instructions from accessing the core (kernel) of the operating system. In these CPUs, any program can access and go from any code section to any code section, so it is easy to crash the system. In contrast, the ARM with virtual memory capability provides resources to the operating system that prevent the user from either accidentally or maliciously taking over the core (kernel) of the operating system and forcing the system to crash. Of course, this idea of protection is nothing new; it is commonly used in mainframes, where it is often referred to as user and supervisor mode. We will discuss the access permission in next section. In the ARM

system with virtual memory implemented, the virtual address is the address shown by a given register of ARM as seen by the compiler/assembler. The physical address is 32-bit address that is placed on the 32 pin of the CPU to locate an actual physical location such as a RAM, I/O, or ROM location. This physical address allows access to any 4G bytes of memory locations of ARM memory space. We must be reminded that to access any memory location in the 4G bytes address space of ARM, we need the entire 32-bit addresses of A31-A0.

## Going from a virtual address to a physical address in 4K page size

In paging, the virtual address is divided into three parts. They are:

- a) The upper 12 bits (A31–A20) are used for an entry into what is called a ***translation table*** or page directory. There is a 32-bit register inside the ARM MMU that holds this physical base address of the translation table. Since the upper 12 bits of the virtual address point to the entry in the translation table, there can be 4,096 entries ( $2^{12} = 4096$ ). Each entry in the translation table is 4 bytes and the pointer to each entry should be word aligned. Note that the two LSB bits of the pointer to an entry in the translation table are zeroes. Figure 15-2 shows how bits A31 to A20 of virtual address and bits 31 to 14 of Translation Table Base Register are concatenated to produce a pointer to an entry in the translation table.

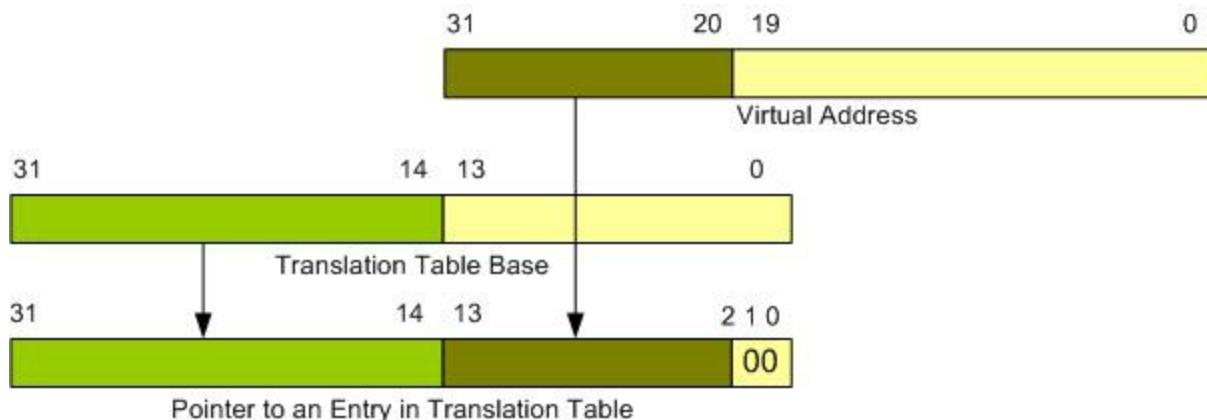


Figure 15-2: Accessing Translation Table

Each entry in the translation table is 4 bytes and called ***descriptor***. Of the 4 bytes, the upper 22 bits are used to point to a second table called ***page table***. This second table holds the physical address of the 4K page frame. The next step shows how the correct entry in the second table (page table) is located.

- b) A19–A12 (8 bits total) of the virtual address are used to point to one of the page table entries. Since the middle 8 bits of the virtual address point to the entry in the second-level page table directory, there can be 256 page directories ( $2^8 = 256$ ). Figure 15-3 shows how A19–A12 (8 bits total) of the virtual address and the upper 22 bits of translation table descriptor are concatenated to produce pointer to an entry in the page table.

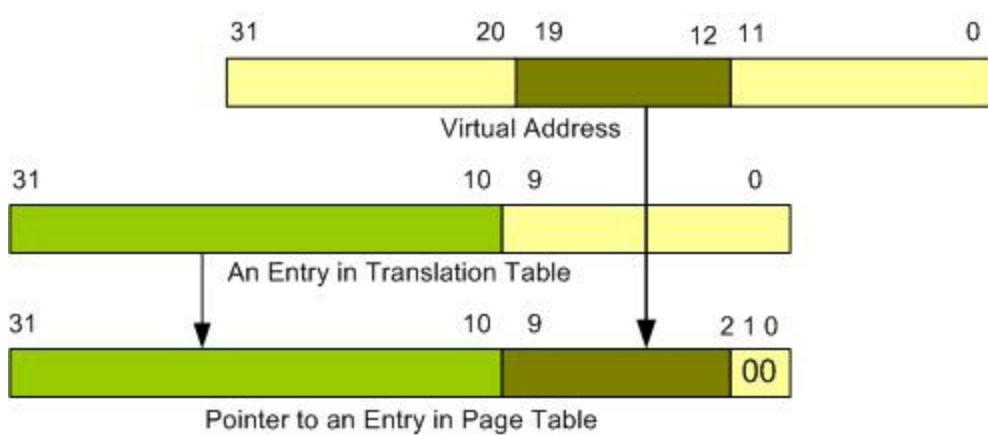


Figure 15-3: Accessing Page Table

Again, each entry in page table (second table) has 4 bytes. The upper 20 bits are for A31–A12 of the physical address of where data is located.

- c) The lower 12 bits of the physical address are the lower 12 bits of the virtual address (A11-A0). See Figure 15-4. In other words, only the lower 12 bits of the virtual address match the lower 12 bits of the physical location in RAM (or ROM) where data is located, and the upper 20 bits of the virtual address must go through two levels of translation tables to get the actual physical address of the beginning page where the data is held.

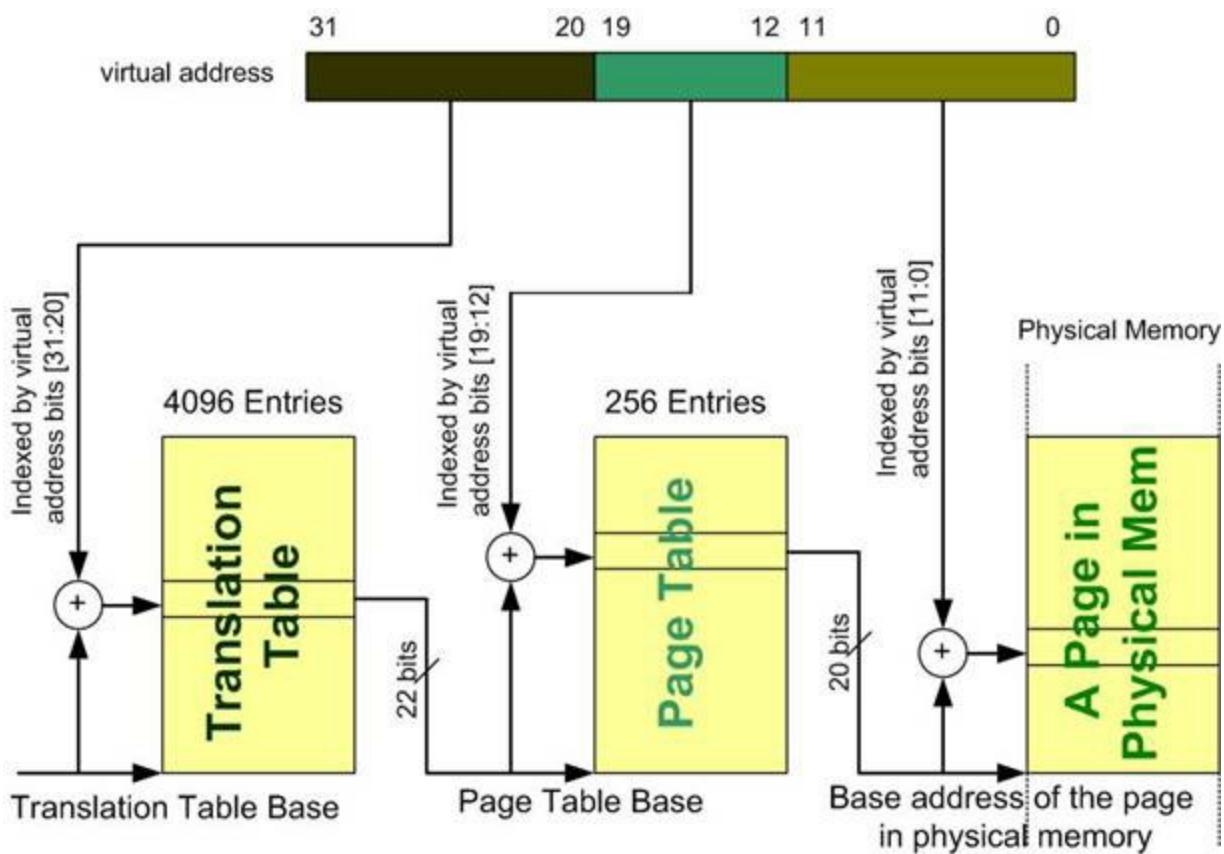


Figure 15-4: 4K Paging Mechanism in ARM

The above scheme seems like a very long and inefficient procedure for each memory access, and it is. This is the reason for the existence of TLB (translation lookaside buffer) inside the CPU itself. Next, we examine how TLB works.

## TLB and paging

The ARM keeps a table for the 32 most recently used pages present in main memory.

This table is called the translation lookaside buffer (TLB) and is kept inside the ARM. The TLB holds (caches) the list of the most recently (commonly) used physical addresses of the page frames. When the CPU wants to access a piece of information (data or code) by providing the virtual address, it first compares the 20-bit upper address with the TLB to see if the table entry for the desired page is already inside the MMU. This results in two possibilities:

- (1) If it matches (TLB hit), it picks the 20-bit physical address (A31-A12) of the page and combines it with the lower 12 bits of the virtual address (A11-A0) to make a 32-bit physical address (A31-A0) to put on the 32 address pins to fetch the data (or code);
- (2) If it does not match (TLB miss), the CPU walks through the table and replaces the TLB entry. This will take several extra memory cycle times since it must go through 2 levels of translation. The ARM literature refers to this translation as table-walking. In the 4K page, there are two levels of table-walking level 1 and level 2. The goal is to have minimum page faults (TLB misses) and avoid table-walking. To do that we can increase the TLB size. Next, we discuss the pros and cons of bigger TLBs. See Figure 15-5.

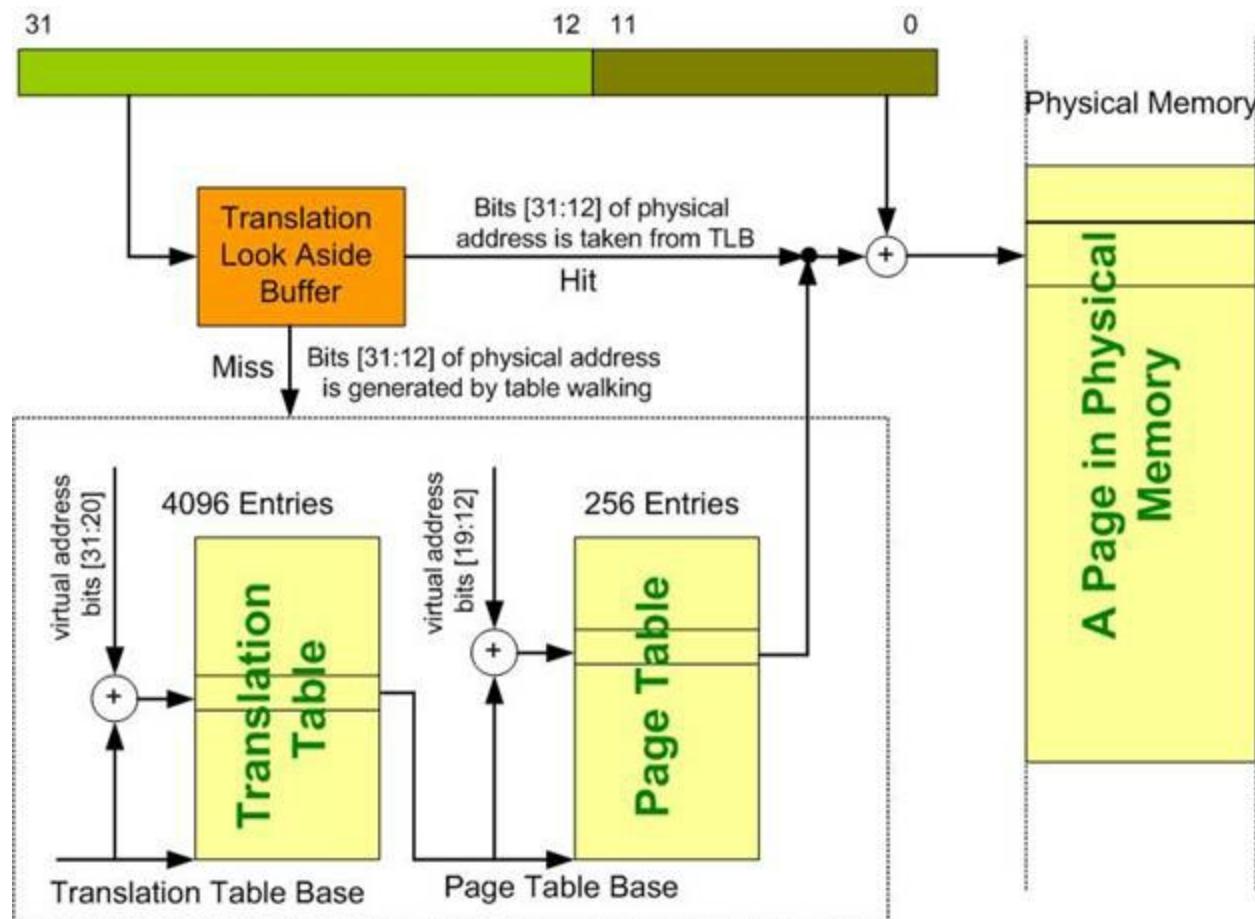


Figure 15-5: Translation Look Aside Buffer in ARM

## The bigger the TLB, the better

Let's assume an ARM chip has the TLB with 64 entries. That means the TLB inside the CPU keeps the list of addresses for the 64 most recently used pages, which allows the CPU to have instant access to 256K bytes ( $64 \times 4K = 256K$ ) of code and data at any time without going through the time-consuming process of converting the virtual address to a physical address.

(two levels of table-walking). See Figure 15-3. The 64 TLB entries use  $64 \times 32$ -bit of SRAM since the addresses are 32-bit in ARM. Therefore, one way to avoid the process of virtual-to-physical address translation (to maximize TLB hit) is to increase the number of pages held by the TLB. Then, the question is why not have 1024 entries for TLB and have instant access to 4M bytes ( $1024 \times 4K = 4,096K = 4M$ ) of code and data? The problem is searching for the 1024 entries takes time even if we have a very fast comparators and that defeats the purpose of TLB. Also using 1024 entries TLB needs  $1024 \times 4$  bytes = 4096 bytes of SRAM inside the ARM. With 64 entries TLB we only need  $64 \times 4$  bytes = 256 bytes of SRAM. Obviously larger TLB is more expensive. So some vendor of ARM CPUs might use 2-way or 4-way set associative with 32 entries to reduce the occurrence of page fault, that is to reduce virtual-to-physical address translation time (table-walking). See the reference manual for vendor of your ARM chip for TLB size and its associativity. Also see cache memory chapter for explanation of n-way set associative concepts. See Example 15-1

### Example 15-1

Assume a given ARM chip has 32 entries for TLB. After caching in the addresses of 32 most recently used pages calculate how much physical memory it has instant access to before using table-walking translation for:

- c) 4K page
- d) 1M section(page)

#### Solution:

- a) with 32 entries TLB, the CPU has instant access to 128K bytes ( $32 \times 4K = 128K$ ) of code and data at any time without going through the time-consuming process of two levels of table-walking.
- b) with 32 entries TLB, the CPU has instant access to 32M bytes ( $32 \times 1M = 32M$ ) of code and data at any time without going through the time-consuming process of one level of table-walking.

### Going from a virtual address to a physical address in 1M page size

Using 4K bytes page size can result in too many page faults in applications such as graphics where the files are large. The ARM has also 1M page size but the ARM literature calls it section instead of page. In section size of 1M bytes, the virtual address is divided into two parts. They are:

- a) The upper 12 bits (A31–A20) are used for an entry into translation table, just like the 4K bytes page. A 32-bit master register inside the ARM holds the physical base address of the translation table, just like the 4K paging (actually they are the same as we will see in next section). Since the upper 12 bits of the virtual address point to the entry in the

translation table, there can be 4096 entries in the translation table ( $2^{12} = 4096$ ). Each entry in the translation table is 4 bytes and is called *section descriptor*. Of the 4 bytes of each section descriptor, the upper 12 bits are used for the upper 12-bit physical address (A31-A20) of the 1M section memory section frame.

- b) The lower 20 bits of the virtual address are used for the lower 20 bits of the physical address (A19-A0). See Figure 15-6. In other words, only the lower 20 bits of the virtual address match the lower 20 bits of the physical block location (A19-A0) in RAM or ROM where data is located, and the upper 12 bits of the virtual address must go through only one level of translation tables to get the actual physical address of the beginning memory section where the data is held.

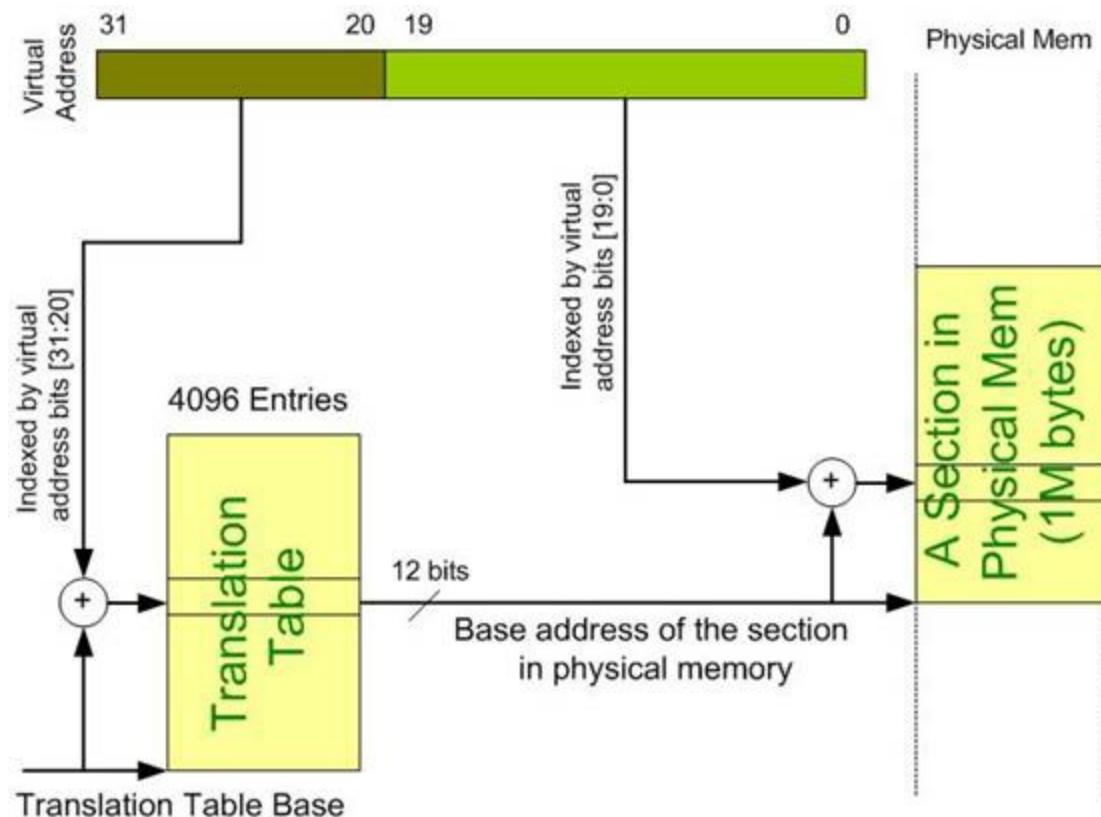


Figure 15-6: One Level Virtual-to-Physical Address Translation for 1M Page in ARM

As we can see from above scheme this is less time consuming going through one level of table-walking and updating TLB instead of 2 levels of table-walking used for 4K paging. The only problem is to bring in 1M bytes of information from hard drive into DRAM can be time consuming and it works efficiently if the files are large.

#### 4 Gigabytes of virtual memory

As seen in Figure 15-6, the 12 bits of virtual address pointing to translation table, can have 4,096 ( $2^{12}$ ) possible combinations. Each possible value can access a 1M bytes of physical memory. Therefore, we have  $2^{12} \times 2^{20} = 4$  gigabytes of virtual memory for the ARM. That is why we do not need two level of table walking when we use 1 MB pages. With a TLB of 64 entries, the TLB inside the CPU can have 64 (out of the 4096) 1 MB pages of the most recently used sections, which allows the CPU to have instant access to 64M bytes ( $64 \times 1M = 64M$ ) of code.

and data at any time without going through the time-consuming process of converting the virtual address to a physical address (1-level table-walking). Again, it must be emphasized that the virtual memory and its protection mechanism is not available in all ARM products such as ARM Cortex-M series.

## Review Questions

1. True or false. If an ARM does not have MMU, then physical address is the same as virtual address.
2. Virtual memory refers to \_\_\_\_\_ (main DRAM, hard disk) memory.
3. How does the operating system decide which code (or data) should be abandoned to make room for new code?
4. True or false. ARM supports both segmentation and paging.
5. What are the page sizes supported by the ARM MMU?
6. True or false. The TLB is held by the DRAM main memory.
7. With 64 entries for TLB, the CPU must make \_\_\_\_\_ comparison to see if it already has the physical address.

## Section 15.2: Page Table Descriptors and Access Permission in ARM

In this section we examine some of the registers used in the MMU. We also explore access permission bits in the page descriptor and see how they are used to protect memory from unwanted access. First, we examine the steps the system goes through upon power-up.

### The operating system role in multitasking system

Upon powering a computer system with operating systems such as Windows and Linux it goes through the following steps:

- 1) Upon applying power, the CPU wakes up at an address held by the program counter. This address is assigned to ROM chip. The ROM chip has a program called boot program. In the x86 PC it is called BIOS (basic input output system).
- 2) The boot program residing in ROM tests everything in the system including the CPU, ROM, DRAM and peripherals. One way to test the CPU itself is by writing a fixed value to a register and then move the contents of that register to another register until the value has gone through all the registers. At the end if the original value is still the same, that means the CPU is working fine. That is what the BIOS of x86 PC does upon turning the system on. To test the ROM boot, one can use the check-sum byte (or check-sum word) to make sure the ROM is not corrupted. Another important job of the boot program is to test the DRAMs and provides the amount of DRAM installed in the system to the operating system. The CMOS chip in x86 PC keeps the account of the total amount of DRAM installed and other system information that OS needs.
- 3) The last thing the boot program does is it loads the operating system from hard disk into DRAM and the control of the system is handed over to the OS. The OS occupies a portion of DRAM as long as the system is on.
- 4) From then on any time the user activates an application (Web browser, word processor, ...) the OS brings the application program residing in disk into DRAM and allocate a portion of DRAM to it. The scheduler in OS runs the applications. As we activate more and more applications, the OS runs out of space in DRAM. This is when the memory management of the OS kernel starts to swap the applications out of the DRAM back into hard disk to create room in DRAM for new application programs. If we have small amount of DRAM, most of the OS time (and for that matter the CPU time) is spent swapping files back and forth between DRAM and disk. That is the reason many OSes require certain minimum amount of DRAM. The DRAM must be large enough that not only it take care of the portion of OS residing in DRAM (this portion called the kernel), but also to take care of memory needs of enough applications being run by the OS. And that is also the reason the more DRAM you have in a system the better the system performance since the CPU is running the applications instead of running the memory management part of OS kernel to swap information back and forth between DRAM and hard disk. Next, we examine the resources in the MMU of ARM to help it to manage virtual memory.

### The Control register in ARM

Many ARM chips used in embedded systems do not have MMU for the virtual memory. Therefore for ARM chips without MMU, there is only the physical address as we have seen in

previous chapters. The ARM9 chip has an on-chip MMU. There are 16 registers in a group of registers called CP15 and they are designated as c0 to c15. According to ARM manual “The system control coprocessor (CP15) is used to configure and control the ARM9 processor. The caches, Tightly-Coupled Memories (TCMs), Memory Management Unit (MMU), and most other system options are controlled using CP15 registers. You can only access CP15 registers with MRC and MCR instructions in a privileged mode.” Next, we examine the role the major registers of c1, c2, and c3 play in MMU operation.

### The c1 Control register in ARM

The most important of the CP15 registers is the c1 register. The c1 register is used to turn on the MMU of ARM among other things. See Figure 15-7.

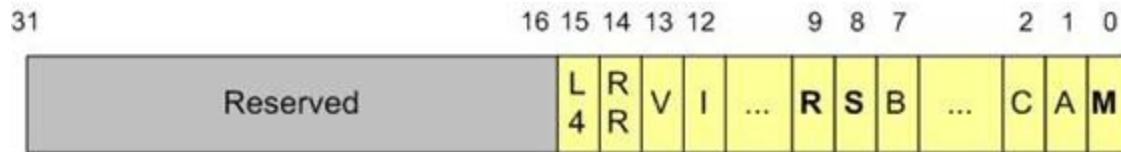


Figure 15-7: c1 Bits

We examine some of the bits of the c1 register. Meanwhile examine the other bits of c1 register to see how it can be used to enable and disable the caches. See Table 15-1.

Bits	Name	Function
0	M	MMU enable/disable: 0 = disabled, 1 = enabled.
1	A	Alignment fault enable/disable: 0 = Data address alignment fault checking disabled, 1 = enabled.
2	C	Cache enable/disable: 0 = Cache disabled 1 = Cache enabled.
7	B	Endianness: 0 = Little-endian, 1 = Big-endian.
8	S	System protection. It modifies the MMU protection system.
9	R	ROM protection. This bit modifies the ROM protection system.
12	I	ICache enable/disable: 0 = ICache disabled 1 = ICache enabled.
13	V	Location of exception vectors:0 = Normal, 1 = High exception vectors.
14	RR	Replacement strategy for ICache and DCache: 0 = Random replacement 1 = Round-robin replacement.
15	L4	Determines if the T bit is set when load instructions change the PC: 0 = loads to PC set the T bit, 1 = loads to PC do not set T bit

Table 15-1: Bits of c1 Control Register in ARM

#### M bit

The bit 0 of c1 register is used to turn on the MMU since upon power-on Reset the MMU is disabled.

#### R (read) bit

This bit allows code or data to be write-protected. For example, the core of the operating system can be write-protected to prevent from writing into it and crashing the system. In the case of old DOS, any program could alter the core of the operating system residing in main memory (DRAM), thereby crashing the PC.

#### S (system) bit

This bit allows a section of memory to be designated as system. For example, the core of the operating system can be designated as both R and S, which prevents the user program from writing to it and crashing the system.

### c2 (Translation Table Base Address) register

Another important CP15 register is c2 register. As we can see from Figure 15-8, the lower 14-bits of 32-bit of the Translation Table Base register is not used. The CPU uses the upper 18 bits of the Translation Table Base register to locate the starting address of the DRAM in which the first-level descriptor tables are located. We will see more about this soon.

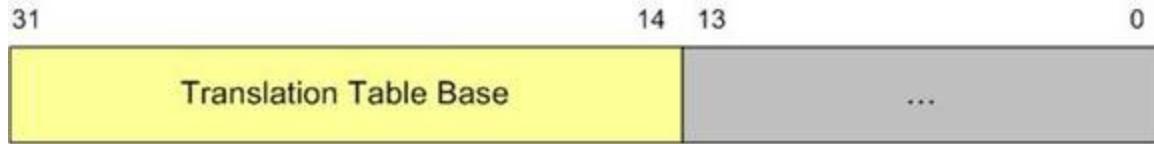


Figure 15-8: c2 Translation Table Base register

### c3 (Domain access control) register

The ARM MMU uses register c3 to divide the memory into 16 domains. Since the c3 register is 32-bit we can get total of 16 domains using 2 bits for each domain. See Figure 15-9. The 2-bits for each Domain are used to assign access permission to a given block of memory. They are as follow:

- 00: No Access
- 01: Client
- 10: Reserved
- 11: Manager

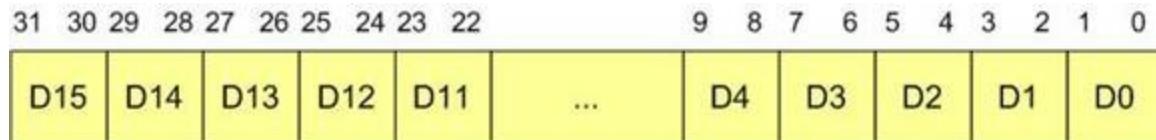


Figure 15-9: c3 Domain register

ARM provides protection mechanism by allowing any memory page belonging to data or code to be assigned Client (user) or Manager (supervisor) access levels. While operating systems are always assigned the Manager level, the user and applications such as word processors are assigned the Client level. Using this mechanism any attempt by the Client to take over the operating system (Manager) is blocked. As we see next, the page table descriptor uses 4 bits (0000 to 1111) to select one of the 16 Domains. This domain selection levels of Manager and Client along with some bits in page table descriptor provides protection to various memory pages belonging to kernel and various tasks. Again, it must be noted that memory page access permission are primarily controlled through the use of domains bits. The MMU first checks the domain access permission before it uses the page table descriptor to see if it is allowed to access the memory.

### The physical Locations of descriptor Table

As we saw in c2 register (Figure 15-8), the lower 14-bits of 32-bit of the Translation Table

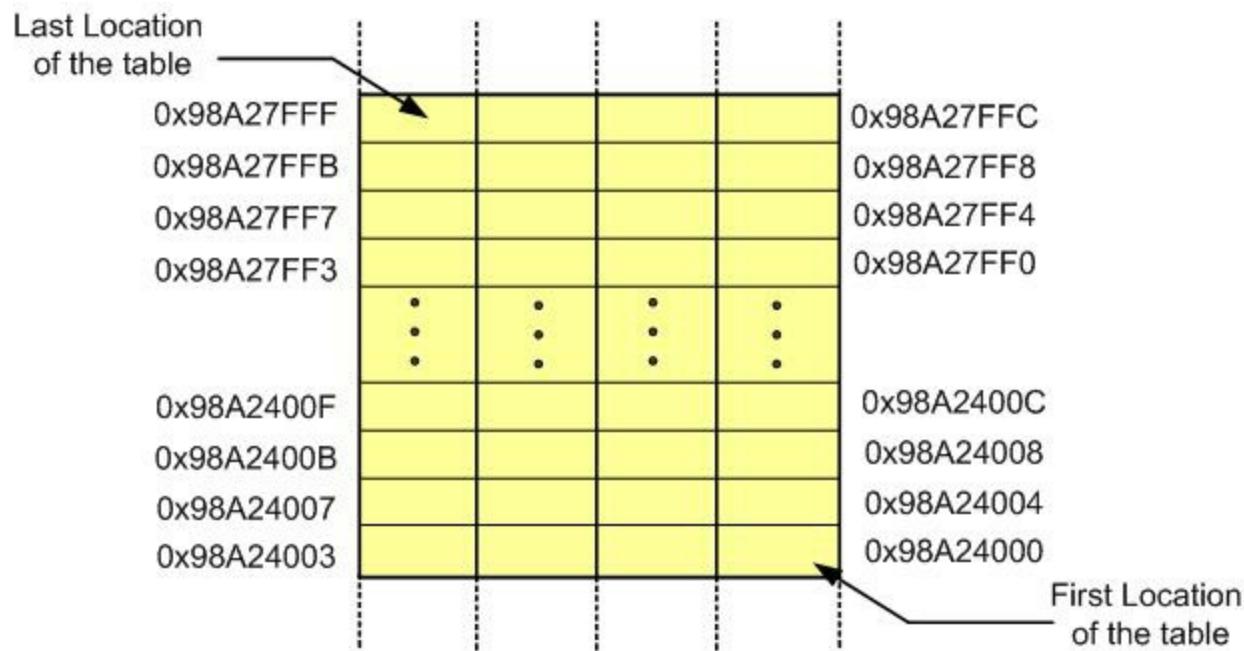
Base Address register is not used. The CPU uses the upper 18 bits of the Translation Table Base Address register to find the starting address of the DRAM in which it should find all the first-level descriptor tables. Since the lower 14-bits of Translation Table Base Address register is not used, it can give us total of  $2^{12} = 4096$  possibilities. That means we have Maximum of 4096 entries. Since each entry is 4 bytes, the table entries can take maximum of 16K bytes ( $4096 \times 4$  bytes) of DRAM if all the table entries are created. See Example 15-2.

### Example 15-2

Assume the Translation Table Base Address register (c2 of CP15) has the value of 0x98A24000. Give the beginning and end memory address of the descriptor table if a given OS builds all possible entries for table.

#### Solution:

Each descriptor table uses 4 bytes of memory. So if a system builds all the possible 4096 descriptor entries it uses  $4096 \times 4$  bytes = 16,384 = 16K bytes. The OS builds table starting at DRAM address 0x98A24000 and goes to 98A27FFF since  $0x98A27FFF - 0x98A24000 = 3FFF = 16,383$ . Now  $16,384 + 1 = 16384$  since the first location starts at 0.



### Examining the descriptor table for 1M section size

The 4 bytes of the page table descriptor gives us information about the type of page, its location in DRAM, and its access permission. See Figure 15-10. The upper 12 bits of the virtual address are used to locate one of 4096 entries of the first-level descriptor table entry. The lowest 2-bits of first-level page table descriptor tells us page size selection. The 01 is for 1M section size and 10 is for 4K page size. In the case of 1M page size, the upper 12 bits of the first-level descriptor are used as the upper 12 bits address of the 1M section. The lower 20 bit address comes from the virtual address, as we mentioned in the last section. That shows why we have only first-level descriptor for 1M section size.

Section Base Address	...	AP	Domain	1	C	B	1	0
----------------------	-----	----	--------	---	---	---	---	---

Figure 15-10: 1M page(section) descriptor

In the first-level section descriptor table there are several bits dedicated to domain selection and access permission. The 4-bits for domain selection comes from one of the 16 possibilities of the c2 register. As mentioned earlier, the domain selection gives us the options of designating a memory page as no access, manager, or client. The C and B in descriptor table are for cacheable and bufferable, respectively. This allows making the section of virtual memory cacheable and bufferable. The 2 bits of the first-level descriptor is designed for the AP (access permission). After examining the descriptor table for 4K page size, we look at the access permission.

### Examining the descriptor for 4K (small) page size

The 4 bytes of each entry in the translation table (first level descriptor) gives us information about the type of page it is pointing to, its location in DRAM, and its access permission. See Figure 15-11.

31	10 9 8	5 4 3 2 1 0
Page Table Base Address	Domain	1 0 1

Figure 15-11: 4K page descriptor

Just like the 1M bytes section, the upper 12 bits of virtual address are used to locate one of 4096 entries of the first-level descriptor table entry. The lowest 2-bits of first-level descriptor table tells us page size selection. The 01 is for 4K page size. The 4-bits for domain selection in the first-level descriptor table comes from one of the 16 possibilities of the c2 register. As mentioned earlier, domain selection gives us the options of designating a memory page as no access, manager, or client. This is just like the 1M section. In the case of 4K page size, the upper 22 bits of the first-level descriptor table are used as a base address into the second-level descriptor table. Now, total of 8 bits (bits 19-12) of virtual address is used to select one of the 256 entries for second-level descriptor. Notice since 8 plus 22 gives us 30 bits, the lower 2 bits of the second-level descriptor table entry are always 00. The upper 20 bits address of the 4K page is located in this second-level descriptor table. Now, combining the lower 12 bits of the virtual address and the upper 20-bits from the second-level descriptor table we have the 32 bit address we need for physical address.

In the second-level page descriptor table entry there are total of 8 bits (bits 4 to 11) for access permission (AP3-AP0). See Figure 15-12.

31	12 11 10 9 8 7 6 5 4 3 2 1 0
Page Table Base Address	AP3 AP2 AP1 AP0 C B 1 0

Figure 15-12: Second Level Descriptor

Each 2 bits are used to assign access permission to 1K bytes of the 4K page. The C and B in second-level descriptor table are for cacheable and bufferable, respectively.

## Access permission

The ARM MPU provides protection for virtual memory by allowing a memory page to be assigned a permission level. The permission levels are: No Access, Read only, and Read/Write. We use the AP bits to set the access permission for 4 K page or 1 M section. Next, we will examine the AP bits.

### No Access

If a memory page is designated as No Access, any attempt by a program to access it is aborted and will result in memory access fault (exception). In a given system, one can designate a section of the operating system as No Access permission level, therefore any attempt by the user program to take over the operating system is blocked.

### Read Only

This allows a code or data region to be write protected (read only). For example, the core of the operating system can be write protected, which prevents the user from writing into it and crashing the system. In earlier processor such as 8086, any program could alter the core of the operating system residing in main memory, thereby crashing the system. The Read Only option block such an attempt. In most cases, we make the code (instructions) region a Read Only to prevent the program overwrite.

### R/W (read/write)

This allows a data region to be readable and writeable. For example, the pages belonging to RAM for scratch pad and stack memory must be assigned the R/W access permission. The same way the memory region belonging to I/O ports also must be designated as R/W.

### Privileged and User (Unprivileged) modes

As we saw in the interrupt chapter, we have two levels of Privileged permission and User (Unprivileged) permission. According to ARM manual “code can execute as privileged or unprivileged. Unprivileged execution limits or excludes access to some resources. Privileged execution has access to all resources.” It must be noted that some of the ARM manual refer to Unprivileged as User. See Table 15-2.

AP	Function	Privileged permission	User permission
00	No Access	No access	No access
01	Privileged Access Only	Read/write	No access
10	write by user will generate fault	Read/write	Read-only
11	Full Access by both	Read/write	Read/write

Table 15-2: AP (access permission) bits options

The interrupt chapter gives more information about the Privileged and Unprivileged modes.

The 4-bits of domain selection along with the 2-bit of AP (access permission) in the first-level descriptor and S and R bits of c1 register can gives us all kinds of memory access

permission for the virtual memory. See Table 15-3.

AP	S	R	Privileged permission	User permission
00	0	0	No access	No access
00	1	0	Read-only	No access
00	0	1	Read-only	Read-only
00	1	1	reserved	reserved
01	x	x	Read/write	No access
10	x	x	Read/write	Read-only
11	x	x	Read/write	Read/write

Table 15-3: Using AP bits with S and R bits

## Cacheable and bufferable memory regions

Many of the ARM chips come with on-chip cache and buffers. The ARM MPU allows the designation of a memory region as cacheable and bufferable.

### C (cacheable)

This allows designating a memory region as cacheable or non-cacheable. In the case of code residing in the Flash ROM, one can designate it as cacheable if the ARM chip has on-chip cache. By doing this, the code is brought into the cache and speed up the program execution. If the ARM chip has separate caches for code (instruction) and data, then we can designate the data region also as cacheable allowing the CPU to bring the data into the data cache. Bit 3 (C) in section descriptor indicate whether the area of memory mapped by this page is treated as cacheable or non-cacheable.

### B (bufferable)

This allows designating a memory region as bufferable or non-bufferable. Bit 2 (B) in section descriptor indicates whether the area of memory mapped by this page is treated as bufferable or non-bufferable. Remember that bit 7 of c1 is endianness. Do not confuse bufferable bit of section descriptor with endianness bit of c1.

Bit C along with bit B in section descriptor indicate whether the area of memory mapped by this page is treated as write-back cacheable, write-through cacheable, noncached buffered, or noncached nonbuffered. See Example 15-3.

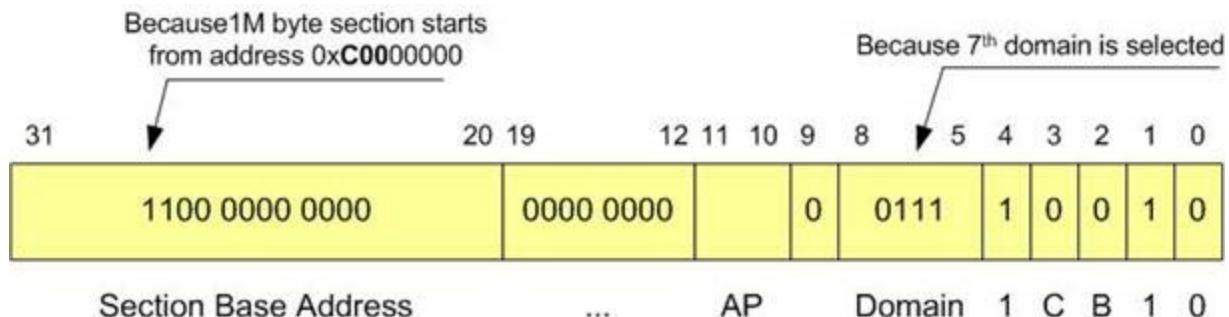
### Example 15-3

Assume we have c2=0x6000000 for Translation Table Base Address, c3=000000EB00 for domain selection. and S=1 and R=1 in the c1 register. Find the following:

- Location of descriptor table if virtual address for the desired section is 0x00900400.
- The contents of descriptor table if we are using the 7th domain and 1M byte section page size occupying address range of 0xC0000000 - 0xC00FFFFF in DRAM. Assume that B and C bits are zero.
- The exact physical address of the information we are accessing.

**Solution:**

- a) Ignoring the lowest 20 bits of the virtual address of 0x00900400 and using the upper 12 bits we get 0x009. That means the 9th entries in the translation table. Since Translation Table Base Address register c2 has 0x6000000 the address of the descriptor is located at  $0x60000000 + (9 \times 4) = 0x60000024$  since each descriptor table entry uses 4 bytes of RAM. Notice  $9 \times 4 = 36 = 0x24$ .



- b) The 9th descriptor table content is 0b1100 0000 0000 0000 0XX0 1111 0010.
  - c) The exact physical address of the information we are accessing is 0xC0000400 in which bits 0 to 19 (0x00400) are the same as 20 rightmost bits of the virtual address (0x00900400) and bits 20 to 31 (0xC00) are from section base address field in the section descriptor.

Using the MMU along with the MPU, one can provide some powerful protection mechanism for the ARM software. We examine some of the features of MPU in the next section.

## Dirty bit and Access bit

Many operating systems implement what is called the A bit for accessed bit by the way of software since some descriptor tables do not have it. If the data or instruction code in the main memory is accessed,  $A = 1$ ; otherwise,  $A = 0$ . This allows the operating system to monitor the A bit periodically to see if the CPU is using this piece of code or data. If a piece of code or data has not been accessed recently, the next time the operating system needs to make room in main memory for new pieces of code (or data), it can move this code (or data) back to the hard disk. The A bit also allows the operating system to decide if a given piece of information (code or data) needs to be saved. For example, if a piece of data has not been accessed, the operating system can trash it and avoid wasting time saving it on the hard disk. On the other hand, if the data was accessed and it was written into, the operating system must save a copy of it on the hard disk before it abandons it to create room in main memory for some other data or code. Another bit that can be implemented by way of software is called dirty bit (D). Assume that there is some memory that can be written into. The accessed (A) bit indicates if the data has been accessed but does not indicate if any new data was written into it. Why should the operating system care whether the memory is altered (written into)? If the data is altered, it is the job of the operating system to save it on the disk to make sure that the hard disk always has the latest data. If the dirty bit is zero ( $D = 0$ ), it means that the data has not been altered and the operating system can abandon it when it needs room for new data (or code) since the

original copy is still on the hard disk. This will save time for the operating system. If the dirty bit is one (D = 1), the operating system must save the data before it is overwritten or abandoned.

## Review Questions

1. In 4K page size, where is the physical address of the desired code or data located?
2. Of the 4 bytes of the first-level descriptor table entry for 1M bytes section size, which bits of the virtual address are the same as the physical address?
3. True or false. In 1M bytes section size, the virtual and physical addresses are the same.
4. To get the physical address in ARM 4K bytes page size, the virtual address must go through \_\_\_ (1, 2) stage(s) of translation.
5. Of the 4 bytes of the second-level descriptor table entry for 4K bytes page size, which bits of the virtual address are the same as the physical address?
6. Which register holds the start of the DRAM address in which the first-level descriptor table is located?
7. What is the maximum number of table entries for the first-level descriptor table?

## Section 15.3: MPU and Memory Protection in ARM

The MPU (memory protection unit) allows the protection of any portion of 4G bytes of physical memory from unwanted access. Not all ARM chips have an on-chip MMU since many microcontroller-based systems do not use virtual memory. However, most of the ARM chips have an on-chip MPU. While MMU is used to implement the virtual memory for systems with mass storage such as disk, the MPU is used for the protection of the physical memory. The MPU is disabled upon power-on Reset. That means in order to use the ARM MPU one must enable it before it is used. If for some reason an ARM chip lacks the on-chip MPU feature or we do not enable it, the 4G bytes of the memory space can be accessed by any program regardless of having Privileged (Supervisor) or Unprivileged (User) permission. According to ARM manual "*Code can execute as privileged or unprivileged. Unprivileged execution limits or excludes access to some resources. Privileged execution has access to all resources.*" It must be noted that some of the ARM manuals refer to Unprivileged as User. The interrupt chapter gives more information about the Privileged and Unprivileged modes. Much of the ARM memory protection concepts discussed in this section applies to all MPUs of the ARM family regardless of the version or the maker.

### Region size for MPU

The MPU of ARM allows us to divide the physical (RAM, ROM, or I/O Peripherals) memory space into regions and assign access permission, size, location, and memory attributes to each region. If a program tries to access a memory region which is not allowed to access, the protection unit will abort the access and causes the MemManage fault exception. It is the job of the system designer (or operating system) to set (or update) the region setting. The size of the region can vary from 4K bytes to 4 Giga bytes as long as they are power of 2. The allowed region sizes start with the 4KB size and it is doubled as we go up in size. They are 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, and so on.

The following compares and contrasts the MPU and MMU in ARM

1. The protection mechanism of MPU can be used only when the MPU is enabled. In this regard it is just like MMU.
2. Upon Reset both MMU and MPU are disabled.
3. While MMU page sizes are limited to choices of 1K, 4K, 64K, and 1M bytes, the MPU regions sizes can vary from 4K to 4G as long as it is power of 2.
4. While, the MMU must use one or two levels of page translation to access the physical memory, there is no need for translation table in MPU.
5. Although both MMU and MPU use four levels of access permission, each has its own set of registers to set the access permission.

The ARM Cortex series have enhanced the MPU greatly. Next, we describe some of the registers and how they are used.

### MPU Type Register

We can assign separate regions to instruction (IREGION) and data (DREGION). The D0 bit

of the ARM MPU\_TYPE register tells us if this option is supported. If D0=0, it means the memory is unified and does not support separate regions for code and data. In that case, the DREGION is used for both code and data and there is no IREGION designation. See Figure 15-13 and Table 15-4. It also means we use the DREGION part of the MPU\_TYPE register to see the number of regions the ARM Cortex support. In many cases this is set to 8 which is the maximum number of regions support by a given ARM chip. We have no control over it since it is read only (RO) register.

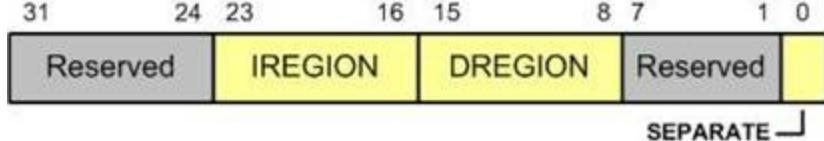


Figure 15-13: MPU\_TYPE register

Bits	Name	Function
0	SEPARATE	Indicates support for unified or separate instruction and date memory maps. (0: unified)
7-1	RESERVED	
15-8	DREGION	Indicates the number of supported MPU data regions: 0x08= Eight MPU regions.
15-8	RESERVED	
23-16	IREGION	Indicates the number of supported MPU instruction regions. Always contains 0x00. The MPU memory map is unified and is described by the DREGION field.

Table 15-4: MPU\_TYPE Register

## How to enable MPU?

The D0 bit of the MPU\_CTRL register allows us to enable the MPU option if there is an on-chip MPU. See Figure 15-14. The other bits of this register allow us to use MPU during the execution of Interrupt Service Routine (ISR). See the manual of your ARM chip.

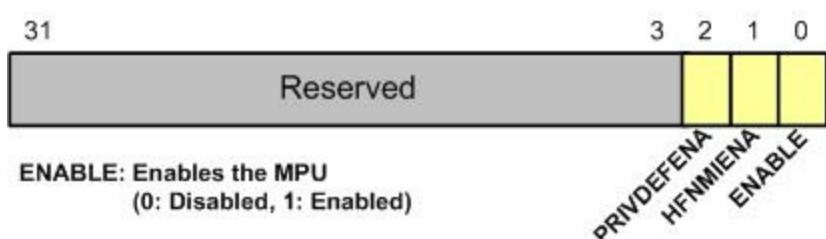


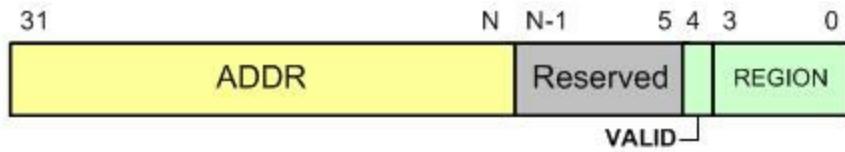
Figure 15-14: MPU\_CTRL Register

## How to select a region?

We use D7-D0 bits of MPU\_RNR register to select one of the 8 regions. Although D7-D0 can take values of 0x00-0xFF, we can use maximum value of 0x07 for it since the ARM Cortex does not support more than 8 regions. This register is very important since it is used by both the MPU address (MPU\_RBAR) register and MPU attributes (MPU\_RASR) register. In other words, we do not have 8 MPU address registers and 8 MPU attribute registers, one for each region. We must always set the value of this register first before we can access the MPU address and MPU attribute registers.

## Choosing region address

To set the region physical address, we use bits D31-D5 of MPU\_RBAR (Region Base Address) register. Of course this is done after we have loaded the region number into the MPU\_RNR register. Notice this is only 27 bits since D4-D0 bits are not available. This also means the smallest region size we can choose is 32 bytes ( $2^5=32$ ) since lower 5 bits are not used for region address.



ADDR: Region base address field. The value of N depends on the region size.

Figure 15-15: MPU\_RBAR (MPU region Base Address) Register

Now, we use this register to set the region address. How does it know which of the 8 regions it is referring to? Whenever this register is accessed it is assumed that the MPU\_RNR register holds the desired region number. This is a clever way of accessing regions without having to set aside hundreds of registers for regions.

## Choosing region attributes and size

We set the MPU region size and attribute register using the MPU\_RASR (Region Attribute and Size) register. Now, let's look at the allowed size. The D5-D1 bits are used to set the memory region size. Although the value can range from 00000 to 11111 (0 to 31), the lowest value it can take is 00100 or 4. The region size in bytes is set as follow:

$$\text{Region size in Byte} = 2^{(\text{size} + 1)}$$

Since the size bits cannot be lower than 00100 means  $2^{(4+1)} = 2^5 = 32$  byte is the smallest memory region size the MPU supports. This also matches the fact that the lower 5 bits of MPU address region, MPU\_RBAR, is not available, as shown in Figure 15-15. The highest region size is 4GB if we set the size bits to 11111 (31 in decimal) since  $2^{(31+1)} = 2^{32} = 4G$ . In this case the region Base address is 0x00000000. See Table 15-5. Also see Example 15-4.

N Bits	$2^{(\text{N}+1)}$	Bytes
00100	$2^{(4+1)}$	32
00101	$2^{(5+1)}$	64
00110	$2^{(6+1)}$	128
00111	$2^{(7+1)}$	256
01000	$2^{(8+1)}$	512
01001	$2^{(9+1)}$	1K
....	....	....
11101	$2^{(29+1)}$	1G
11110	$2^{(30+1)}$	2G
11111	$2^{(31+1)}$	4G

Table 15-5: Allowed region size for ARM Cortex MPU

### Example 15-4

Verify the region size calculation for (a) 1KB, (b) 64KB, (c) 1GB, and (d) 4GB

**Solution:**

(a) With N=01001 we have  $2^{(9+1)} = 1\text{KB}$

(b) With N=01111 we have  $2^{(15+1)} = 64\text{KB}$

(c) With N=11101 we have  $2^{(29+1)} = 1\text{GB}$

(d) With N=11111 we have  $2^{(31+1)} = 4\text{GB}$

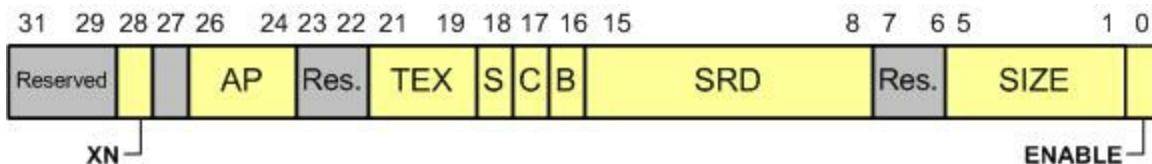


Figure 15-16: MPU\_RASR (MPU Region Attribute and Size Register)

Notice from the Figure 15-16, the D0 bit is used to enable the region. The D5-D1 bits are used to set the region size. We can also have sub region within a region. D8-D15 are set aside for that. We set the region attributes using the upper 16 bits of MPU\_RASR register. Notice the XN bit of the MPU\_RASR. The XN bit is used to designate a region as Executable Only. XN=0 if the region belongs to program code. See Tables 15-6 to 15-9.

Bits	Name	Function
0	Enable	Region enable bit
5-1	SIZE	Specifies the size of the MPU protection region. The minimum permitted value is 3 (0b00010).
7-6	Reserved	
15-8	SRD	Subregion disable bits. For each bit in this field: 0 = corresponding sub-region is enabled 1 = corresponding sub-region is disabled Region sizes of 128 bytes and less do not support subregions. When writing the attributes for such a region, write the SRD field as 0x00.
18	S	Shareable bit, see Table 15-8
16	B	
17	C	Memory access attributes, see Table 15-8
21-19	TEX	
26-24	AP	Access permission field, see Table 15-7
28	XN	Instruction access disable bit: 0 = instruction fetches enabled

1 = instruction fetches disabled.

Table 15-6: MPU\_RASR (MPU Region Attribute and Size Register)

AP	Privileges permissions	Unprivileged permissions	Description
000	No access	No access	All accesses generate a permission fault
001	RW	No access	Access from privileged software only
010	RW	RO	Writes by unprivileged software generate a permission fault
011	RW	RW	Full access
100	Unpredictable	Unpredictable	Reserved
101	RO	No access	Reads by privileged software only
110	RO	RO	Read only, by privileged or unprivileged software
111	RO	RO	Read only, by privileged or unprivileged software

Table 15-7: AP (Access Permission) Encoding

X	C	B	S	Memory type	Shareability	Other attributes
0	0	0	X	Strongly-ordered	Shareable	-
0	0	1	X	Device	Shareable	-
0	1	0	0	Normal	Not shareable	Outer and inner write-through. No write allocated
0	1	0	1	Normal	Shareable	
0	1	1	0	Normal	Not shareable	Outer and inner write-back. No write allocated
0	1	1	1	Normal	Shareable	
1	0	0	0	Normal	Not shareable	
1	0	0	1	Normal	Shareable	Outer and inner non-cacheable
1	0	1	X	Reserved encoding	-	-
1	1	0	X	Implementation defined attributes	-	-
1	1	1	0	Normal	Not shareable	Outer and inner write-back. Write and read allocate
1	1	1	1	Normal	Shareable	
0	0	0	X	Device	Not shareable	Nonshared Device
0	0	1	X	Reserved encoding	-	-
0	1	X	X	Reserved encoding	-	-
<sub>1</sub> B <sub>0</sub>	A <sub>1</sub>	A <sub>0</sub>	0	Normal	Not shareable	Cached memory, B <sub>1</sub> and B <sub>0</sub> define the outer policy,
<sub>1</sub> B <sub>0</sub>	A <sub>1</sub>	A <sub>0</sub>	1	Normal	Shareable	while the inner policy is defined by A <sub>1</sub> and A <sub>0</sub> bits. See Table 15-9.

Table 15-8: TEX, C, B, and S encoding

A <sub>1</sub>	A <sub>0</sub>	Corresponding cache policy
0	0	Non-cacheable
0	1	Write back, write and read allocate
1	0	Write through, no write allocate
1	1	Write back, no write allocate
<b>Caution:</b> B <sub>1</sub> and B <sub>0</sub> define the outer policy, in the same way.		

Table 15-9: Cache policy for memory attribute encoding

### Example 15-5

Show the register values for the following regions:

- region 0 with starting address of 0x00000000 and region size of 64KB.
- region 1 with starting address of 0x0100000 and region size of 32KB.

**Solution:**

(a)

```
MPU_CTRL=0x00000001;      /* Enable on-chip MPU */
MPU_RNR=0x00000000;      /* region number 0 */
MPU_RBAR=0x00000000      /* region 0 address */
MPU_RASR=0x00000001F     /* size for 64KB and enable region */
```

(b)

```
MPU_CTRL=0x00000001;      /* Enable on-chip MPU */
MPU_RNR=0x00000001;      /* region number 1 */
MPU_RBAR=0x0010000        /* region 1 address, 32KB aligned */
MPU_RASR=0x00000001F     /* size for 32KB and enable region */
```

In above examples the upper 16-bit attributes are not shown. We leave it to the reader to explore them. Also notice that we must use D0 of size/attribute register to enable the region.

### Review Questions

- True or false. All the ARM chips come with MPU
- True or false. Upon Power-on Reset, the MPU is enabled and ready to go.
- True or false. If an MPU is not enabled, the 4G bytes of the memory space can be accessed by any program regardless of having Privileged or Unprivileged (User) permission.
- The I/O peripherals such as GPIO (general purpose I/O) region must be assigned the \_\_\_\_\_ access permission
- In ARM, the \_\_\_\_\_ (Privileged, Unprivileged) is assigned to Operating System.

### Answers to Review Questions

#### Section 15.1

- True

2. Hard disk
3. According to rule of the least recently used
4. False
5. 1K, 4K, 64K, and 1M bytes
6. False
7. 64

## Section 15.2

1. In the second-level descriptor table
2. The lower 20 bits
3. False
4. 2 stages
5. The lower 12 bits
6. The c2 of CP15
7. 4096 since 12 bits are used

## Section 15.3

1. False
2. False
3. True
4. 32
5. N=13 (or 1101 in binary) since  $2^{(13+1)} = 2^{14} = 32\text{KB}$ .

# **Appendix A: IC Interfacing, System Design, and Failure Analysis**

The invention of the transistor and the subsequent advent of integrated circuit (IC) technology is believed by many to be the start of the second industrial revolution. In this chapter we provide an overview of IC technology and interfacing. In addition, we look at the computer system as a whole and examine some general considerations in system design. In Section A.1 we provide an overview of IC technology. IC interfacing and system design considerations are examined in Section A.2. In Section A.2 we also discuss failure analysis in systems.

## Section A.1: Overview of IC Technology

In this section we provide an overview of IC technology and discuss some developments in logic families.

The transistor was invented in 1947 by three scientists at Bell Laboratories. In the 1950s, transistors replaced vacuum tubes in many electronics systems, including computers. It was not until in 1959 that the first integrated circuit was successfully fabricated and tested by Jack Kilby of Texas Instruments. Prior to the invention of the IC, the use of transistors, along with other discrete components such as capacitors and resistors, was common in computer design. Early transistors were made of germanium, which was later abandoned in favor of silicon. This was due to the fact that the slightest rise in temperature resulted in massive current flows in germanium-based transistors. In semiconductor terms, it is because the band gap of germanium is much smaller than that of silicon, resulting in a massive flow of electrons from the valence band to the conduction band when the temperature rises even slightly. By the late 1960s and early 1970s, the use of the silicon-based IC was widespread in mainframes and minicomputers. Transistors and ICs were based on P-type materials. Due to the fact that the speed of electrons is much higher (about two and a half times) than the speed of the holes, N-type devices replaced P-type devices. By the mid-1970s, NPN and NMOS transistors had replaced the slower PNP and PMOS transistors in every sector of the electronics industry, including in the design of microprocessors and computers. Since the early 1980s, CMOS (complementary MOS) has become the dominant method of IC design. Next we provide an overview of differences between MOS and bipolar transistors.

### MOS vs. bipolar transistors

There are two type of transistors: bipolar and MOS (metal-oxide semiconductor). Both have three leads. In bipolar transistors, the three leads are referred to as the *emitter*, *base*, and *collector*, while in MOS transistors they are named *source*, *gate*, and *drain*. In bipolar, the carrier flows from the emitter to the collector and the base is used as a flow controller. In MOS, the carrier flows from the source to the drain and the gate is used as a flow controller. See Figure A-1.

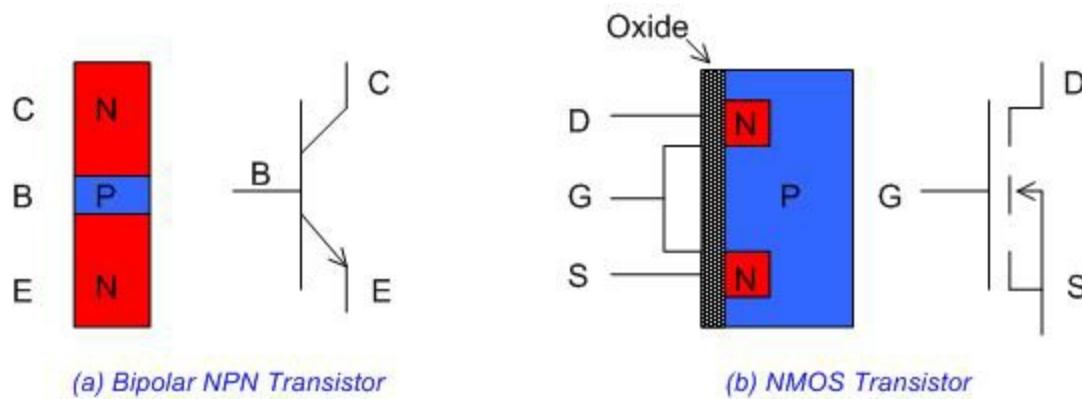


Figure A-1: Bipolar vs. MOS Transistors

In NPN-type bipolar transistors, the electron carrier leaving the emitter must overcome two voltage barriers before it reaches the collector. One is the N-P junction of the emitter-base and the other is the P-N junction of the base-collector. The voltage barrier of the base-collector

is the most difficult one for the electrons to overcome (since it is reversed biased) and it causes the most power dissipation. This led to the design of the unipolar type transistor called *MOS*. In N-channel MOS transistors, the electrons leave the source reaching the drain without going through any voltage barrier. The absence of any voltage barrier in the path of the carrier is one reason why MOS dissipates much less power than bipolar transistors. The low power dissipation of MOS allows putting millions of transistors on a single IC chip. In today's million-transistor microprocessors and DRAM memory chips, the use of MOS technology is indispensable. Without the MOS transistor, the advent of desktop personal computers would not have been possible, at least not so soon. The use of bipolar transistors in both the mainframe and minicomputer of the 1960s and 1970s required expensive cooling systems and large rooms due to their bulkiness. MOS transistors do have one major drawback: They are slower than bipolar transistors. This is due partly to the gate capacitance of the MOS transistor. For MOS to be turned on, the input capacitor of the gate takes time to charge up to the turn-on (threshold) voltage, leading to a longer propagation delay.

## Overview of logic families

Logic families are judged according to (1) speed, (2) power dissipation, (3) noise immunity, (4) input/output interface compatibility, and (5) cost. Desirable qualities are high speed, low power dissipation, and high noise immunity (since it prevents the occurrence of false logic signals during switching transition). In interfacing logic families, the more inputs that can be driven by a single output, the better. This means that high-driving-capability outputs are desired. This plus the fact that the input and output voltage levels of MOS and bipolar transistors are not compatible means that one must be concerned with the ability of one logic family driving the other one. In terms of the cost of a given logic family, it is high during the early years of its introduction and prices decline as production and use rise.

## The case of inverters

As an example of logic gates, we look at a simple inverter. In a one-transistor inverter, while the transistor plays the role of a switch, R is the pull-up resistor. See Figure A-2.

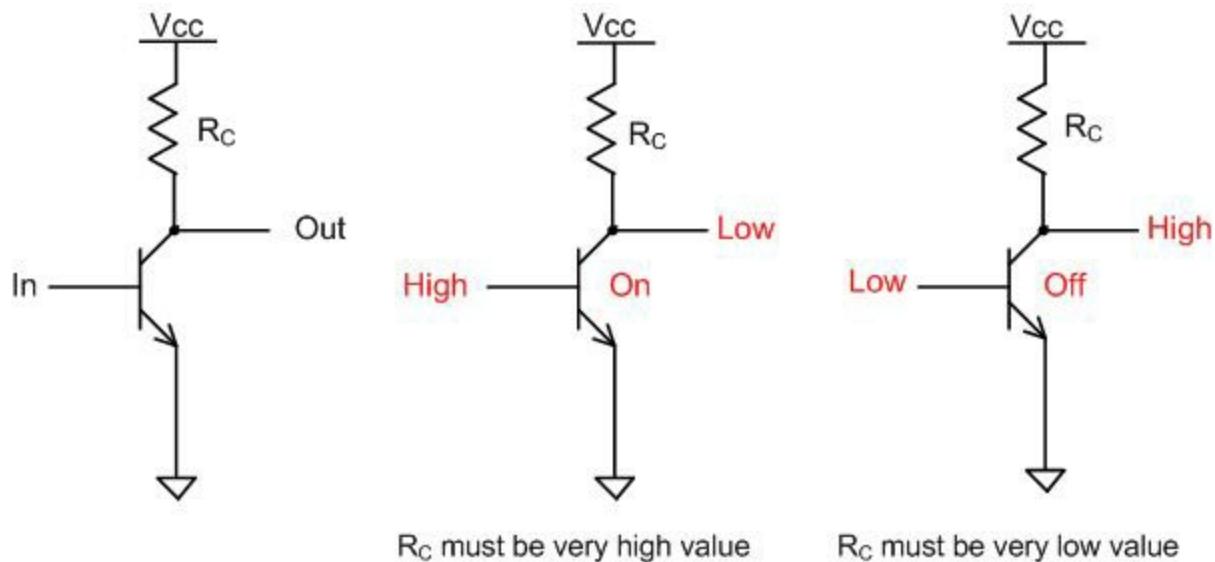


Figure A-2: One-Transistor Inverter with Pull-up Resistor

However, for this inverter to work effectively in digital circuits, the R value must be high

when the transistor is "on" to limit the current flow from VCC to ground in order to have low power dissipation ( $P = VI$ , where  $V = 5\text{ V}$ ). In other words, the lower the  $I$ , the lower the power dissipation. On the other hand, when the transistor is "off",  $R$  must be a small value to limit the voltage drop across  $R$ , thereby making sure that  $V_{OUT}$  is close to  $V_{CC}$ . These are opposing demands on the value of  $R$ . This is one reason that logic gate designers use active components (transistors) instead of passive components (resistors) to implement the pull-up resistor  $R$ .

The case of a TTL inverter with totem pole output is shown in Figure A-3. In Figure A-3, Q3 plays the role of a pull-up resistor.

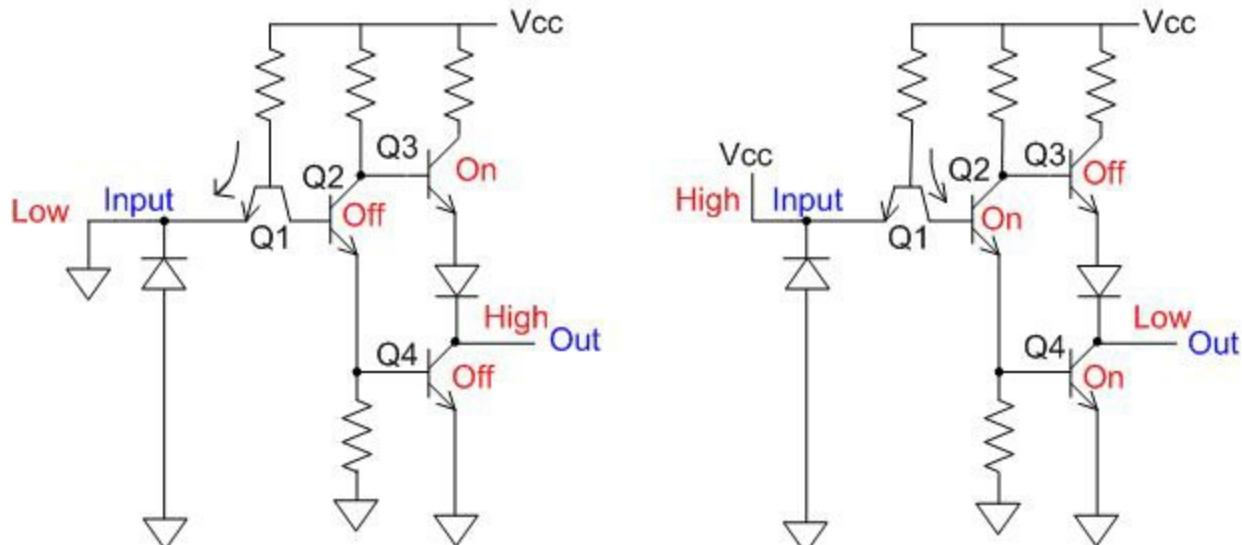


Figure A-3: TTL Inverter with Totem-Pole Output

## CMOS inverter

In the case of CMOS-based logic gates, PMOS and NMOS are used to construct a CMOS (complementary MOS) inverter as shown in Figure A-4.

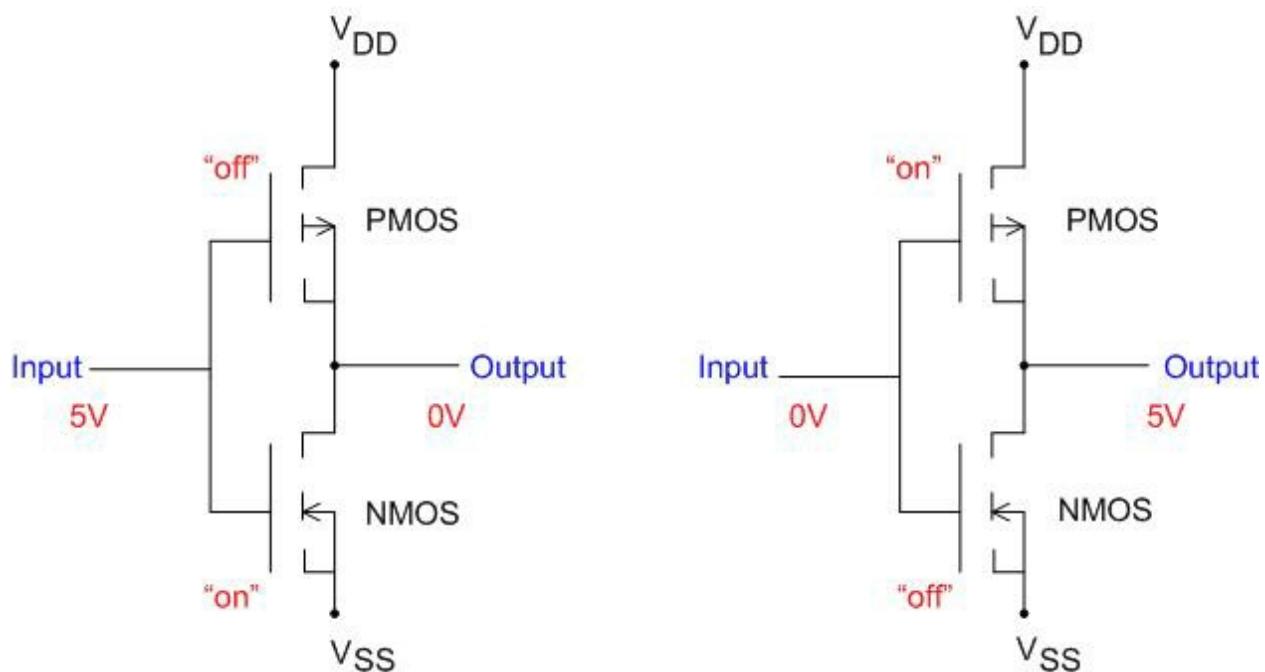


Figure A-4: CMOS Inverter

In CMOS inverters, when the PMOS transistor is off, it provides a very high impedance

path, making leakage current almost zero (about 10 nA); when the PMOS is on, it provides a low resistance on the path of  $V_{DD}$  to load. Since the speed of the hole is slower than that of the electron, the PMOS transistor is wider to compensate for this disparity; therefore, PMOS transistors take more space than NMOS.

## Input, output characteristics of some logic families

In 1968 the first logic family made of bipolar transistors was marketed. It was commonly referred to as the standard TTL (transistor-transistor logic) family. The first MOS-based logic family, the CD4000/74C series, was marketed in 1970. The addition of the Schottky diode to the base-collector of bipolar transistors in the early 1970s gave rise to the S family. The Schottky diode shortens the propagation delay of the TTL family by preventing the collector from going into what is called *deep saturation*. Table A-1 lists major characteristics of some logic families. In Table A-1, note that as the CMOS circuit's operating frequency rises, the power dissipation also increases. This is not the case for bipolar-based TTL.

Characteristic	STD TTL	LSTTL	ALSTTL	HCMOS
$V_{CC}$	5V	5V	5V	5V
$V_{IH}$	2.0V	2.0V	2.0V	3.15V
$V_{IL}$	0.8V	0.8V	0.8V	1.1V
$V_{OH}$	2.4V	2.7V	2.7V	3.7V
$V_{OL}$	0.4V	0.5V	0.4V	0.4V
$I_{IL}$	-1.6 mA	-0.36 mA	-0.2 mA	-1 $\mu$ A
$I_{IH}$	40 $\mu$ A	20 $\mu$ A	20 $\mu$ A	1 $\mu$ A
$I_{OL}$	16 mA	8 mA	4 mA	4 mA
$I_{OH}$	-400 $\mu$ A	-400 $\mu$ A	-400 $\mu$ A	4 mA
Propagation delay	10 ns	9.5 ns	4 ns	9 ns
Static power dissipation ( $f=0$ )	10 mW	2 mW	1 mW	0.0025 nW
Dynamic power dissipation at $f = 100$ kHz	10 mW	2 mW	1 mW	0.17 mW

Table A-1: Characteristics of Some Logic Families

## History of logic families

Early logic families and microprocessors required both positive and negative power voltages. In the mid-1970s, 5V VCC became standard. For example, Intel's 4004, 8008, and 8080 all used negative and positive voltages for the power supply. In the late 1970s, advances in IC technology allowed combining the speed and drive of the S family with the lower power of LS to form a new logic family called FAST (Fairchild Advanced Schottky TTL). In 1985, AC/ACT (Advanced CMOS Technology), a much higher speed version of HCMOS, was introduced. With the introduction of FCT (Fast CMOS Technology) in 1986, at last the speed gap between CMOS and TTL was closed. Since FCT is the CMOS version of FAST, it has the low power consumption of CMOS but the speed is comparable with TTL. Table A-2 provides an overview of logic families up to FCT.

Product	Year Introduced	Speed (ns)	Current (mA)	Drive (mA)
Std TTL	1968	40	30	-2/32
CD4K/74C	1970	70	0.3	-0.48/6.4
LS/S	1971	18	54	-15/24
HC/HCT	1977	25	0.08	-6/-6
FAST	1978	6.5	90	-15/64
AS	1980	6.2	90	-15/64
ALS	1980	10	27	-15/64
AC/ACT	1985	10	0.08	-24/24
FCT	1986	6.5	1.5	-15/64

*Reprinted by permission of Electronic Design Magazine, c. 1991.*

Table A-2: Logic Family Overview

## Recent advances in logic families

As the speed of high-performance microprocessors such as the 386 and 486 reached 25 MHz, it shortened the CPU's cycle time, leaving less time for the path delay. Designers normally allocate no more than 25% of a CPU's cycle time budget to path delay. Following this rule means that there must be a corresponding decline in the propagation delay of logic families used in the address and data path as the system frequency is increased. In recent years, many semiconductor manufacturers have responded to this need by providing logic families that have high speed, low noise, and high drive. Table A-3 provides the characteristics of high-performance logic families introduced in recent years.

Family	Year	Number Suppliers	Tech Base	I/O Level	Speed (ns)	Static Current	$I_{OH}/I_{OL}$
ACQ	1989	2	CMOS	CMOS/CMOS	6.0	80 $\mu$ A	-24/24 mA
ACTQ	1989	2	CMOS	TTL/CMOS	7.5	80 $\mu$ A	-24/24 mA
FCTx	1987	3	CMOS	TTL/CMOS	4.1–4.8	1.5 mA	-15/64 mA
FCTx-T	1990	2	CMOS	TTL/TTL	4.1–4.8	1.5 mA	-15/64 mA
FASTr	1990	1	Bipolar	TTL/TTL	3.9	50 mA	-15/64 mA
BCT	1987	2	BICMOS	TTL/TTL	5.5	10 mA	-15/64 mA

*Reprinted by permission of Electronic Design Magazine, c. 1991.*

Table A-3: Advanced Logic General Characteristics

ACQ/ACTQ are the second-generation advanced CMOS (ACMOS) with much lower noise. While ACQ has the CMOS input level, ACTQ is equipped with TTL-level input. The FCTx and FCTx-T are second-generation FCT with much higher speed. The x in the FCTx and FCTx-T refers to various speed grades, such as A, B, and C, where the A designation means low speed and C means high speed. For designers who are well versed in using the FAST logic family, the use of FASTr is an ideal choice since it is faster than FAST, has higher driving capability ( $I_{OL}, I_{OH}$ ), and produces much lower noise than FAST. At the time of this writing, next to ECL and gallium arsenide logic gates, FASTr is the fastest logic family in the market (with the 5V VCC), but the power consumption is high relative to other logic families, as shown in Table A-3. Since early 2000, a 3.3V VCC with higher speed and lower power consumption has become standard. The combining of high-speed bipolar TTL and the low power consumption of CMOS has given birth to what is called BICMOS. Although BICMOS seems to be the future trend in IC design, at this

time it is expensive due to the extra steps required in BICMOS IC fabrication, but in some cases there is no other choice. For example, Intel's Pentium microprocessor, a BICMOS product, had to use high-speed bipolar transistors to speed up some of the internal functions in order to keep up with RISC processor performance. Table A-3 provides advanced logic characteristics. Table A-4 shows logic families used in systems with different speeds. The x is for the different speeds where A, B, and C are used for designation. A is the slowest one while C is the fastest one. The above data is for the 'LS244 buffer.

System Clock Speed (MHz)	Clock Period (ns)	Predominant Logic for Path
2 – 10	100 – 500	HC, LS
10 – 30	33 – 100	ALS, AS, FAST, FACT
30 – 66	15 – 33	FASTr, BCT, FCTA

Table A-4: Importance of Speed

## Review Questions

1. State the main advantages of MOS and bipolar transistors.
2. True or false. In logic families, the higher the noise margin, the better.
3. True or false. Generally, high-speed logic consumes more power.
4. Power dissipation increases linearly with the increase in frequency in \_\_\_\_\_ (CMOS, TTL).
5. In a CMOS inverter, indicate which transistor is on when the input is high.
6. For system frequencies of 10–30 MHz, which logic families are used for the address and data path?

## Section A.2: IC Interfacing and System Design Issues

There are several issues to be considered in designing a microprocessor-based system. They are IC fan-out, capacitance derating, ground bounce,  $V_{CC}$  bounce, crosstalk, transmission lines, power dissipation, and chip failure analysis. This section provides an overview of these design issues in order to provide a sampling of what is involved in high-performance system design.

### IC fan-out

In IC interfacing, fan-out/fan-in is a major issue. How many inputs can an output signal drive? This question must be addressed for both logic "0" and logic "1" outputs. Fan-out for low and fan-out for high are as follows:

Fan-out (of low)  $\frac{g_{OGC}}{g_{IK}}$

Fan-out (of high)  $\frac{g_{OHN}}{g_{AHF}}$

Of the above two values the lower number is used to ensure the proper noise margin. Figure A-5 shows the sinking and sourcing of current when ICs are connected.

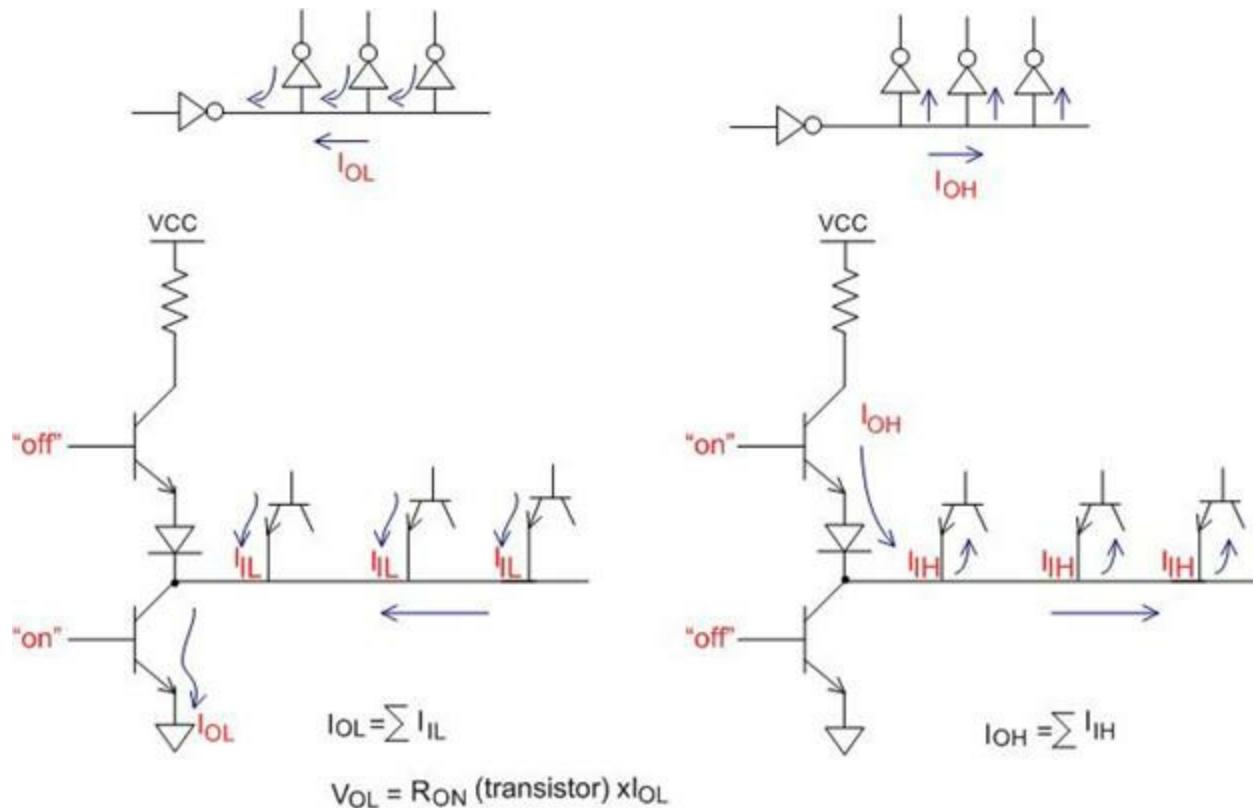


Figure A-5: Current Sinking and Sourcing in TTL

In Figure A-5, as the number of inputs connected to the output increases,  $I_{OL}$  rises, which causes  $V_{OL}$  to rise. If this continues, the rise of  $V_{OL}$  makes the noise margin smaller, and this results in the occurrence of false logic due to the slightest noise.

In designing the system, very often an output is connected to various kinds of inputs. See Examples A-1 and A-2.

## Example A-1

Find how many unit loads (UL) can be driven by the output of the LS logic family.

**Solution:**

The unit load is defined as  $I_{IL} = 1.6 \text{ mA}$  and  $I_{IH} = 40 \mu\text{A}$ . Table A-1 shows  $I_{OL} = 8 \text{ mA}$  and  $I_{OH} = 400 \mu\text{A}$  for the LS family. Therefore, we have

$$\text{fan-out (low)} = I_{OL}/I_{IL} = 8 \text{ mA} / 1.6 \text{ mA} = 5$$

$$\text{fan-out (high)} = I_{OH}/I_{IH} = 400 \mu\text{A} / 40 \mu\text{A} = 10$$

This means that the fan-out is 5. In other words, the LS output must not be connected to more than 5 inputs with unit load characteristics.

---

## Example A-2

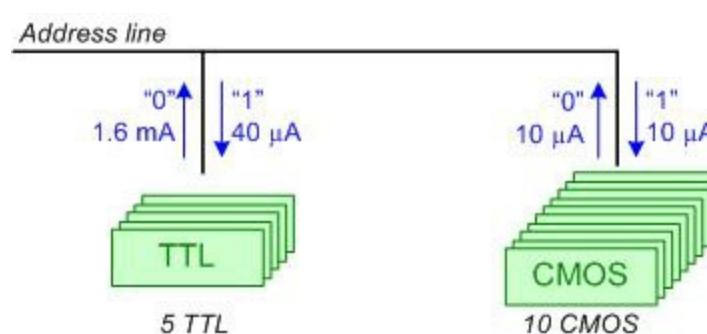
An address pin needs to drive 5 standard TTL loads in addition to 10 CMOS inputs of DRAM chips. Calculate the minimum current to drive these inputs for both logic "0" and "1".

**Solution:**

The standard load for TTL is  $I_{IH} = 40 \mu\text{A}$  and  $I_{IL} = 1.6 \text{ mA}$ , and for CMOS,  $I_{IL} = I_{IH} = 10 \mu\text{A}$ .

minimum current for "0" = total of all  $I_{IL} = 5 \times 1.6 \text{ mA} + 10 \times 10 \mu\text{A} = 8.1 \text{ mA}$

minimum current for "1" = total of all  $I_{IH} = 5 \times 40 \mu\text{A} + 10 \times 10 \mu\text{A} = 300 \mu\text{A}$



---

The total  $I_{IL}$  and  $I_{IH}$  requirement of all the loads on a given output must be less than the driver's maximum  $I_{OL}$  and  $I_{OH}$ . This is shown in Example A-3.

## Example A-3

Assume that the microprocessor address pin in Example A-2 has specifications  $I_{OH} = 400 \mu A$  and  $I_{OL} = 2 mA$ . Do the input and output current needs match?

### Solution:

For a high output state, there is no problem since  $I_{OH} > I_{IH}$ . However, the number of inputs exceeds the limit for  $I_{OL}$  since an  $I_{IL}$  of 8.1 mA is much larger than the maximum  $I_{OL}$  allowed by the microprocessor.

In cases such as Example A-3 where the receiver current requirements exceed the drivers' capability, we must use a buffer (booster), such as the 74xx245 and 74xx244. The 74xx245 is used for bidirectional and the 74xx244 for unidirectional signals. See current 74LS244 and 74LS245 characteristics in Table A-5.

Buffer	$I_{OH}$ (mA)	$I_{OL}$ (mA)	$I_{IH}$ ( $\mu A$ )	$I_{IL}$ (mA)
<b>74LS244</b>	3	12	20	0.2
<b>74LS245</b>	3	12	20	0.2

*Note:  $V_{OL} = 0.4 V$  and  $V_{OH} = 2.4V$  are assumed.*

Table A-5: Electrical Specifications for Buffers

### Capacitance derating

Next we study what is called capacitance derating and its impact in system design. A pin of an IC has an input capacitance of 5 to 7 pF. This means that a single output that drives many inputs sees a large capacitance load since the inputs are in parallel and therefore added together. Look at the following equations.

$$Q = CT \quad (A-1)$$

$$Q / T = CV / T \quad (A-2)$$

$$F = 1 / T \quad (A-3)$$

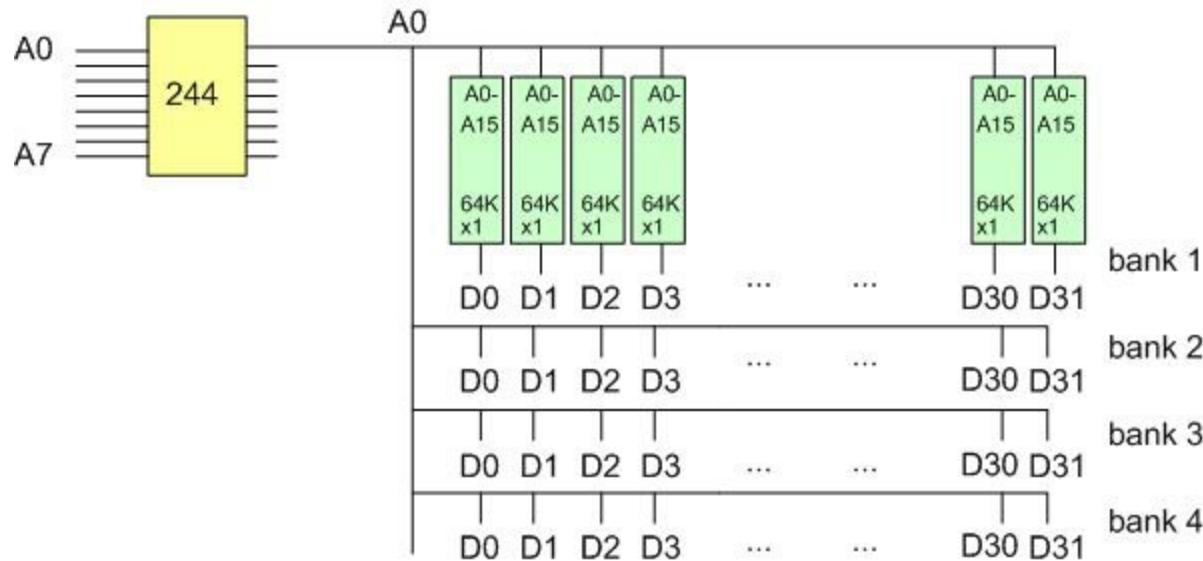
$$I = CVF \quad (A-4)$$

In Equation (A-4),  $I$  is the driving capability of the output pin,  $C$  is  $C_{IN}$  as seen by the output, and  $V$  is the voltage. The equation indicates that as the number of  $C_{IN}$  loads goes up, there must be a corresponding increase in  $I_O$ , the driving capability of the output. In other words, outputs with high values of  $I_{OL}$  and  $I_{OH}$  are desirable. Although there have been some logic families with  $I_{OL} = 64 mA$  and  $I_{OH} = 15 mA$ , their power consumption is high. Equation (A-4) indicates that if  $I = \text{constant}$ , as  $C$  goes up,  $F$  must come down, resulting in lower speed. The most widely accepted solution is the use of a large number of drivers to reduce the load capacitance seen by a given output. Assume that we have a single address bus line driving 16 banks of 32-bit-wide memory. Each bank has 4 chips of  $64K \times 8$  organization, which results in  $16 \times 4 = 64$  memory chips, or  $16 \times 64K \times 32 = 32M$  bytes of SRAM. Depending on how many 244s are used to drive the memory addresses, the delay due to the address path varies substantially.

To understand this we examine three cases.

### Case 1: Two 244 drivers

This option uses two 244 drivers, one for A0–A7 and one for A8–A15. An output of the 244 drives 16 banks of memory, each with 4 inputs. Assuming that each memory input has 5 pF capacitance, this results in a total of  $4 \times 16 \times 5 = 320$  pF capacitance load seen by the 244 output. However, the 244 output can handle no more than 50 pF. As a result, the delay due to this extra capacitance must be added to the address path delay. For each 50 to 100 pF of capacitance, an extra 3 ns delay is added to the address path delay. In our calculation, we use 3 ns for each 100 pF of capacitance. Figure A-6 shows driving memory inputs by two 244 chips. See Example A-4.



*Note: the second 244 for A8-A15 is not shown*

Figure A-6: Case 1, Two 244 Address Drivers

### Example A-4

Calculate the following for Figure A-6, assuming a memory access time of 25 ns and a propagation delay of 10 ns for the 244.

- delay due to capacitance derating on the address path
- the total address path delay for case 1

### Solution:

- Of the 320 pF capacitance seen by the 244, only 50 pF is taken care of; the rest, which is 270 ( $320 - 50 = 270$ ), causes a delay. Since there are 3 ns for each extra 100 pF, we have the following delay due to capacitance derating,  $(270/100) \times 3$  ns = 8.1 ns.
- Address path delay = 244 buffer propagation delay + capacitance derating delay + memory access time = 10 ns + 8.1 ns + 25 ns = 43.1 ns.

## Case 2: Doubling the number of 244 buffers

Doubling the number of 244 buffers will reduce the address path delay. A single 244 drives only 8 banks, or a total of 32 inputs, since there are 4 inputs in each bank. As a result, a 244 output will see a capacitance load of  $32 \times 5 = 160$  pF. In this case, we use only four 244 buffer chips, as shown in Figure A-7 and Example A-5.

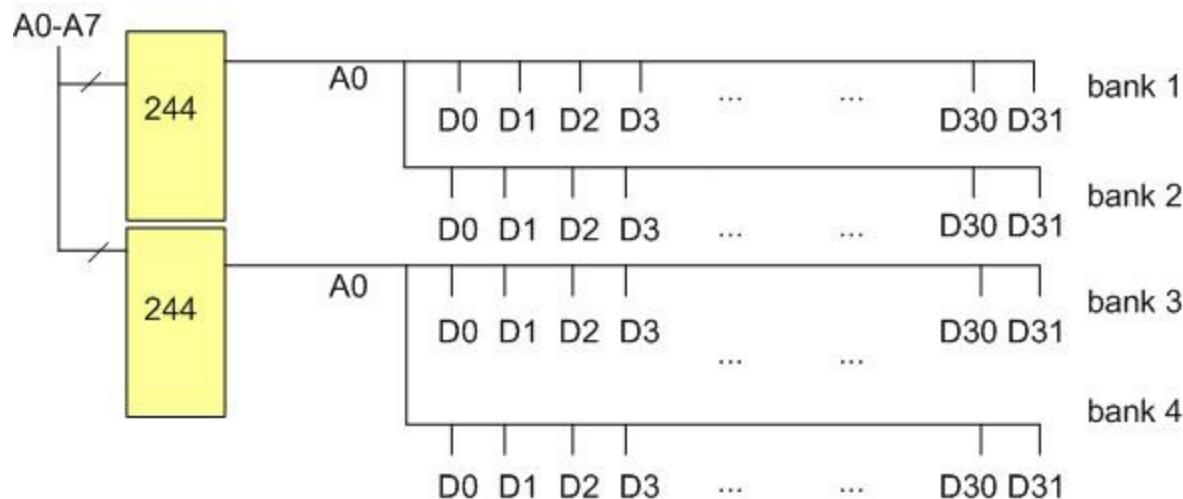


Figure A-7: Case 2, Four 244 Address Drivers

## Example A-5

Calculate (a) delay due to capacitance derating on the address path, and (b) total address path delay for case 2. Assume a memory access time of 25 ns and a propagation delay of 10 ns for the 244.

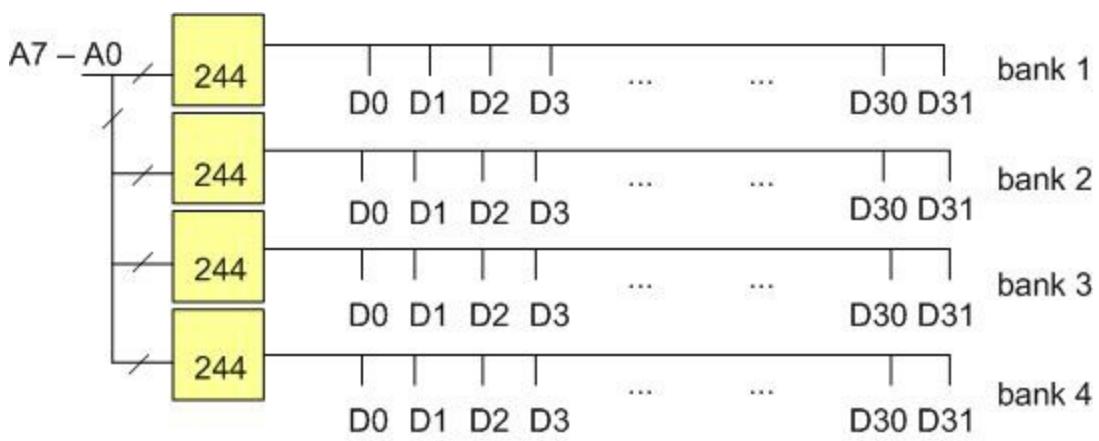
### Solution:

(a) Of the 160 pF capacitance seen by the 244, only 50 pF is taken care of; the rest, which is 110 pF, causes a delay. Since there are 3 ns for each extra 100 pF, we have  $(110/100) \times 3$  ns = 3.1 ns delay due to capacitance derating.

(b) The address path delay = 244 buffer propagation delay + capacitance derating delay + memory access time = 10 ns + 3.1 ns + 25 ns = 28.1 ns.

## Case 3: Doubling again

In this case, we double the number of 244 buffers again, so that an output of the 244 drives four banks, each with 4 inputs. This results in a total capacitance load of  $4 \times 4 \times 5 = 80$  pF. Only 50 pF of it is taken care of by the 244, leaving 30 pF, causing a delay. See Figure A-8.



**Figure A-8: Case 3, A Single 244 Address Driver for Each Bank**

Examining cases 1 through 3 shows that for high-speed system design we must accept a higher cost due to extra parts and higher power consumption.

### Power dissipation considerations

Power dissipation of a system is a major concern of system designers, especially for laptop and hand-held systems. Although power dissipation is a function of the total current consumption of all components of a system, the impact of  $V_{CC}$  is much more pronounced, as shown next. Earlier we showed in Equation (26-4) that  $I = CFV$ . Substituting this in equation  $P = VI$  yields the following:

$$P = VI = CFV^2 \quad (A-5)$$

In Equation (A-5), the effects of frequency and VCC voltage should be noted. While the power dissipation goes up linearly with frequency, the impact of the power supply voltage is much more pronounced (squared). See Example A-6.

### Example A-6

Prove that a 3.3 V system consumes 56% less power than a system with a 5 V power supply.

#### Solution:

Since  $P = VI$ , by substituting  $I = V/R$ , we have  $P = V^2/R$ . Assuming that  $R = 1$ , we have  $P(3.3)^2 = 10.89 \text{ W}$  and  $P(5)^2 = 25 \text{ W}$ . This results in using 14.11 W less ( $25 - 10.89 = 14.11$ ), which means a 56% power saving ( $14.11 \text{ W}/25 \text{ W} \times 100 = 56\%$ ).

### Dynamic and static currents

There are two major types of currents flowing through an IC: dynamic and static. A dynamic current is a function of the frequency under which the component is working, as seen in Equation (A-4). This means that as the frequency goes up, the dynamic current and power

dissipation go up. The static current, also called dc, is the current consumption of the component when it is inactive (not selected).

## Power-down option

The popularity of laptops and tablets have led microprocessor designers to make an all-out effort to conserve battery power. Today processors have what is called *system management mode (SMM)*, which reduces energy consumption by turning off peripherals or the entire system when not in use. The SMM can put the entire system, including the monitor, into sleep mode during periods of inactivity, thereby reducing "power from 250 watts to less than 30 watts." The effects on the 3.3 V power supply alone translate into a power savings of up to 56% over systems with a 5 V power supply, as was shown in Example A-6.

## Ground bounce

One of the major issues that designers of high-frequency systems must grapple with is *ground bounce*. Before we define ground bounce, we will discuss lead inductance of IC pins. There is a certain amount of capacitance, resistance, and inductance associated with each pin of the IC. The size of these elements varies depending on many factors such as length, area, and so on. Figure A-9 shows the lead inductance and capacitance of the 24 pins of a DIP IC.

Pin	Self-inductance	Capacitance
1	15.10 nH	1.86 pF
2	12.20 nH	1.70 pF
3	9.54 nH	1.29 pF
4	7.44 nH	0.95 pF
5	5.31 nH	0.61 pF
6	3.73 nH	0.43 pF
7	3.41 nH	0.43 pF
8	4.66 nH	0.61 pF
9	6.95 nH	0.95 pF
10	8.96 nH	1.29 pF
11	11.70 nH	1.70 pF
12	14.50 nH	1.86 pF
13	14.50 nH	1.86 pF
14	11.70 nH	1.70 pF
15	8.96 nH	1.29 pF
16	6.95 nH	0.95 pF
17	4.66 nH	0.61 pF
18	3.41 nH	0.43 pF
19	3.73 nH	0.43 pF
20	5.31 nH	0.61 pF
21	7.44 nH	0.95 pF
22	9.54 nH	1.29 pF
23	12.20 nH	1.70 pF
24	15.10 nH	1.86 pF

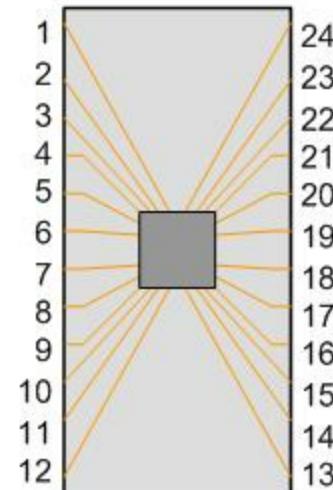


Figure A-9: Inductance and Capacitance of 24-pin DIP

The inductance of the pins is commonly referred to as *self-inductance* since there is also what is called *mutual inductance*, as we will show below. Of the three components of

capacitance, resistance, and inductance, self-inductance is the one that causes the most problems in high-frequency system design since it can result in ground bounce. Ground bounce is caused when a large amount of current flows through the ground pin when multiple outputs change from high to low all at the same time. The voltage relation to the inductance of the ground lead follows:

$$V = L \frac{di}{dt} \quad (A-6)$$

As we increase the system frequency, the rate of dynamic current,  $di/dt$ , is also increased, resulting in an increase in the inductance voltage  $L (di/dt)$  of the ground pin. Since the low state (ground) has a small noise margin, any extra voltage due to the inductance voltage can cause a false signal. To reduce the effect of ground bounce, the following steps must be taken where possible.

1. The  $V_{CC}$  and ground pins of the chip must be located in the middle rather than at the opposite ends of the IC chip (the 14-pin TTL logic IC uses pins 14 and 7 for ground and  $V_{CC}$ ). This is exactly what we see in high-performance logic gates such as Texas Instrument's advanced logic AC11000 and ACT11000 families. For example, the ACT11013 is a 14-pin DIP chip where pins 4 and 11 are used for the ground and  $V_{CC}$  instead of 7 and 14 as in the TTL. We can also use surface mount technology such as the SOIC packages instead of DIP. Surface mount devices have much smaller size and shorter leads. The self-inductance of the leads is shown in Table A-6.

Pins	DIP (nH)	SOIC (nH)
<b>1, 10, 11, 20</b>	13.7	4.2
<b>2, 9, 12, 19</b>	11.1	3.8
<b>3, 8, 13, 18</b>	8.6	3.3
<b>4, 7, 14, 17</b>	6.0	2.9
<b>5, 6, 15, 16</b>	3.4	2.4

*Courtesy of Texas Instruments*

Table A-6: 20-Pin DIP and SOIC Lead Inductance

2. Use logics with a minimum number of outputs. For example, a 4-output is preferable to an 8-output. This explains why many designers of high-performance systems avoid using memory chips or the drivers and buffers of 16- or 32-bit-wide outputs since all the outputs switching at the same time will cause a massive flow of current in the ground pin, and hence cause ground bounce (see Figure A-10).
3. Use as many pins for the ground and  $V_{CC}$  as possible to reduce the lead length, since the self-inductance of a wire with length  $l$  and a cross section of  $B \times C$  is:

$$L=0.002 \ln [2l / (B + C) + l / 2] \quad (A-7)$$

As seen in Equation (A-7), the wire length,  $l$ , contributes more to self-inductance than does the cross section. This explains why all high-performance microprocessors and logic families use several pins for the  $V_{CC}$  and ground. For example, in the case of Intel's Pentium

processor there are over 50 pins for the ground and another 50 pins for the  $V_{CC}$ .

The discussion of ground bounce is also applicable to  $V_{CC}$  when a large number of outputs changes from the low to high state and is referred to as  $V_{CC}$  bounce. However, the effect of  $V_{CC}$  bounce is not as severe as ground bounce since the high ("1") state has wider noise margin than the low ("0") state.

### Filtering the transient currents using decoupling capacitors

In the TTL family, the change of the output from low to high can cause what is called *transient current*. In totem-pole output, when the output is low, Q4 is on and saturated, whereas Q3 is off. By changing the output from the low to high state, Q3 becomes on and Q4 becomes off. It is faster to turn a transistor on than turn a transistor off. This means that there is a time that both transistors are on and drawing currents from the  $V_{CC}$ . The amount of current depends on the  $R_{ON}$  values of the two transistors, and that, in turn, depends on the internal parameters of the transistors. However, the net effect of this is a large amount of current in the form of a spike for the output current, as shown in Figure A-10.

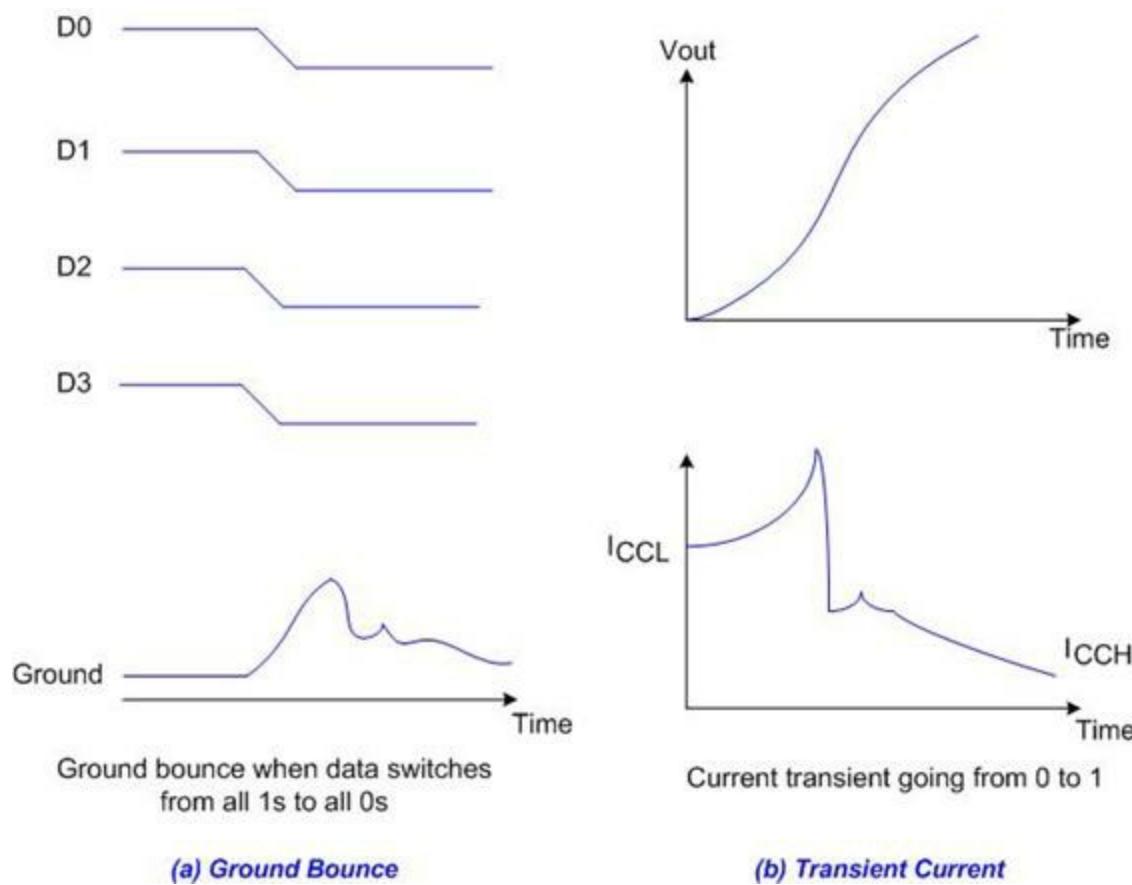


Figure A-10: (a) Ground Bounce (b) Transient Current

To filter the transient current, a 0.01 F or 0.1 F ceramic disk capacitor can be placed between the  $V_{CC}$  and ground for each TTL IC. However, the lead for this capacitor should be as small as possible since a long lead results in a large self-inductance and that results in a spike on the  $V_{CC}$  line [ $V = L \cdot (di/dt)$ ]. This is also called  $V_{CC}$  bounce. The ceramic capacitor for each IC is referred to as a decoupling capacitor. There is also a bulk decoupling capacitor, as described next.

## Bulk decoupling capacitor

As many IC chips change state at the same time, the combined currents drawn from the board's  $V_{CC}$  power supply can be massive and cause a fluctuation of  $V_{CC}$  on the board where all the ICs are mounted. To eliminate this, a relatively large (relative to an IC decoupling capacitor) tantalum capacitor is placed between the  $V_{CC}$  and ground lines. The size and location of this tantalum capacitor vary depending on the number of ICs on the board and the amount of current drawn by each IC, but it is common to have a single 22  $\mu F$  to 47  $\mu F$  capacitor for each of the 16 devices, placed between the  $V_{CC}$  and ground lines. See Technical Notes TN0006 and TN4602 from Micron Technology.

<http://www.micron.com/products/support/technical-notes>

## Crosstalk

Crosstalk is due to mutual inductance. See Figure A-11.

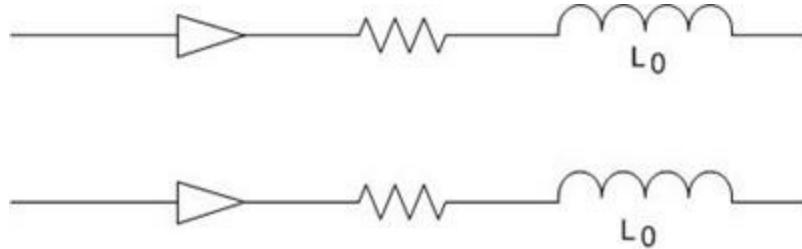


Figure A-11: Crosstalk (EMI)

Previously, we discussed self-inductance, which is inherent in a piece of conductor. *Mutual inductance* is caused by two electric lines running parallel to each other. It is calculated as follows:

$$M = 0.002l \times \ln(2l/d) - \ln(K - 1 + d/l - d/2l)^2 \quad (A-8)$$

where  $l$  is the length of two conductors running in parallel, and  $d$  is the distance between them, and the medium material placed in between affects  $K$ . Equation (A-8) indicates that the effect of crosstalk can be reduced by increasing the distance between the parallel or adjacent lines (in printed circuit boards, these will be traces). In many cases, such as printer and disk drive cables, there is a dedicated ground for each signal. Placing ground lines (traces) between signal lines reduces the effect of crosstalk. This method is used even in some ACT logic families where  $V_{CC}$  and GND pins are next to each other. Crosstalk is also called EMI (electromagnetic interference). This is in contrast to ESI (electrostatic interference), which is caused by capacitive coupling between two adjacent conductors.

## Transmission line ringing

The square wave used in digital circuits is in reality made of a single fundamental pulse and many harmonics of various amplitudes. When this signal travels on the line, not all the harmonics respond the same way to the capacitance, inductance, and resistance of the line. This causes what is called *ringing*, which depends on the thickness and the length of the line

driver, among other factors. To reduce the effect of ringing, the line drivers are terminated by putting a resistor at the end of the line. See Figure A-12.

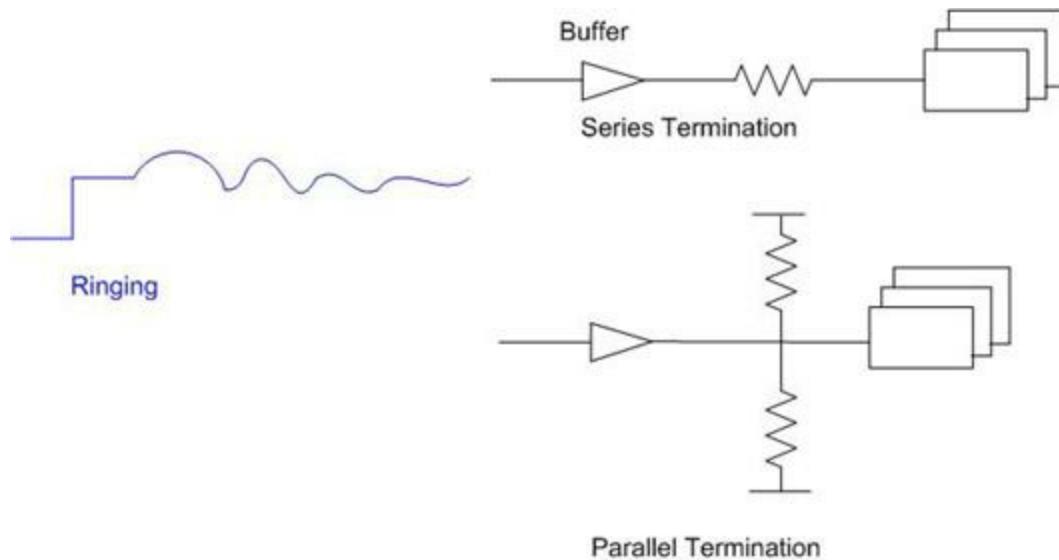


Figure A-12: Reducing Transmission Line Ringing

There are three major methods of line driver termination: parallel, serial, and Thevenin. In many systems resistors of 30–50 ohms are used to terminate the line. The parallel and Thevenin methods are used in cases where there is a need to match the impedance of the line with the load impedance. This requires a detailed analysis of the signal traces and load impedance, which is beyond the scope of this book. In high-frequency systems, wire traces on the printed circuit board (PCB) behave like transmission lines, causing ringing. The severity of this ringing depends on the speed and the logic family used. Table A-7 provides the length of the traces, beyond which the traces must be looked at as transmission lines.

Logic Family	Line Length (in.)
LS	25
S, AS	11
F, ACT	8
AS, ECL	6
FCT, FCTA	5

(Reprinted by permission of Integrated Device Technology, copyright IDT 1991)

Table A-7: Line Length Beyond Which Traces Behave Like Transmission Lines

## FIT and failure analysis

Chip manufacturers provide a parameter called *FIT* (*failure in time*) to measure the reliability for a single chip. The FIT of a single chip is the number of expected failures in a billion ( $10^9$ ) hours of operation. If a chip has FIT of 300, then there will be 300 failures per billion device hours of operation. To reduce the number of device failures, manufacturers use burn-in to eliminate the early failures before the product is shipped to the customer. This is commonly referred to as infant mortality since the failure rate starts high and eventually levels off to a constant level. See Figure A-13.

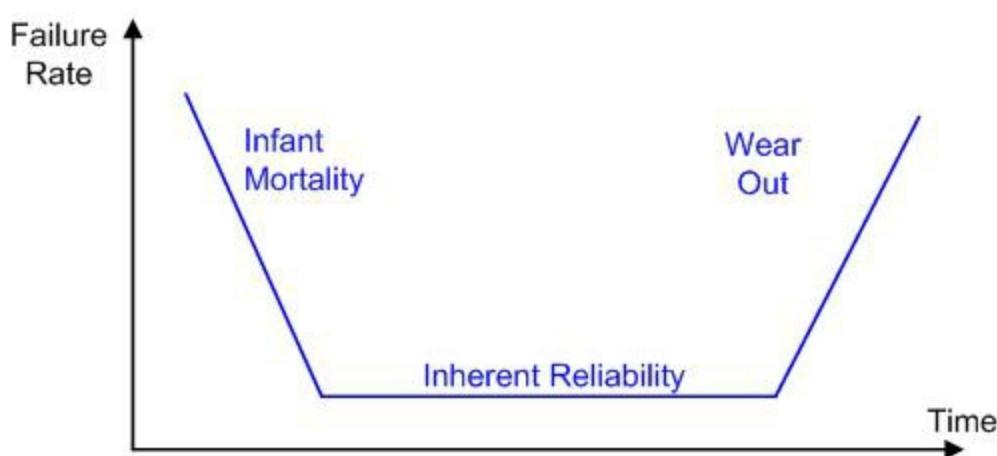


Figure A-13: Bathtub Failure Rate

Although we can eliminate the early failures using burn-in, we can never reduce the failure rate to zero due to wear out and other factors such as soft error. This is discussed next.

## Soft error and hard error

In memory there are two kinds of errors that can cause a bit to change: soft error and hard error. If the cell bit gets stuck permanently in a "high" or "low" state, this is referred to as a *hard error*. Hard error is due to deterioration of the cell caused by wear-out (see Figure A-13). There is no remedy for hard error except to replace the defective RAM chip since the damage is permanent. The other kind of error, a *soft error*, alters the cell bit from 1 to 0 or from 0 to 1, even though the cell is perfectly fine (no hard error). Soft error is caused by alpha particle radiation and power surges. The sources of the alpha particles are the radiation in the air or the materials in the plastic package enclosing the RAM die. The occurrence of a soft error as a result of alpha particles ionizing the charges in a RAM cell is a greater source of concern since it is 5 times more likely to happen than a hard error. As the density of RAM chips increases and the size of the RAM cell goes down, the probability of a soft error for a given cell goes up, but the relation is not linear.

## Mean time between failures (MTBF) for system

Reliability of system depends directly on two factors: a) the FIT (failures in time) value of a single part, and b) the number of parts in the system. We use these two factors to calculate what is called *MTBF (mean time between failures)*. The MTBF predicts the average time between the two consecutive failures. The MTBF for a single chip is calculated using the FIT as follows:

$$\text{MTBF} = 1,000,000,000 \text{ hours} / \text{FIT} \quad (\text{A-9}) \quad (\text{A-9})$$

To get the MTBF rate for the system, we must divide the single-chip MTBF by the number of chips in the system.

$$\text{MTBF of system} = \text{MTBF of one chip} / \text{number of chips in system} \quad (\text{A-10}) \quad (\text{A-10})$$

See Examples A-7 and A-8.

### Example A-7

Assuming that the FIT for a single chip is 252, calculate the MTBF for:

- (a) a single chip
- (b) a system with 512 chips

**Solution:**

- (a) The MTBF for a single chip is as follows:  $MTBF \text{ for 1 chip} = 1,000,000,000 \text{ hr} / 252 = 3,968,254 \text{ hr} = 453 \text{ years}$
  - (b) The MTBF for 512 chips is  $= 453 \text{ years} / 512 \text{ chips} = 0.884 \text{ year} = 323 \text{ days}$
- 

### Example A-8

Calculate the system MTBF for the system in Example A-7 if FIT = 745.

**Solution:**

MTBF for a single chip  $= 109 / 745 \text{ hrs.} = 153 \text{ years}$ . For the system it is  $153 \text{ years} / 512 = 109 \text{ days}$ .

---

See Technical Notes TN-00-14 and TN-00-18 on the <http://www.micron.com> website.

<http://www.micron.com/products/support/technical-notes>

There is a paper called “Testing RAM for Embedded Systems” by Jack Ganssle and available from the following website:

<http://www.ganssle.com/testingram.pdf>

Also see the article “Thirteen feet of concrete won't shield your RAM from the perils of cosmic rays. What's the solution?” by Jack Ganssle in Dr. Dobb's Journal. It is available from the following website:

<http://www.ddj.com/dept/debug/196800160>

## ECL and gallium arsenide (GaAs) chips

The use of L3 cache and EDC (Error Detection and Correction) in systems with speeds of 200 MHz and higher is adding to the data and address path delay. This is forcing designers to resort to using ECL and GaAs chips. Due to the fact that ECL chips have a very high power dissipation, they are not used in low-cost x86 design. However, GaAs chips are showing up in high-speed x86 and RISC-based computers. This is especially the case for the GaAs EDC and cache controller chips. The mass of electrons in GaAs is lighter than in silicon, due to its

quantum mechanics structure. As a result, the electrons in GaAs have a much higher speed. This means that GaAs chips can achieve a much higher speed than silicon. The power dissipation of the GaAs transistor is comparable to the silicon-based MOS transistor. Therefore, GaAs technology might appear to provide the ideal chip since it has the speed of ECL (it is even faster than ECL) and the power dissipation of CMOS. However, it has the following disadvantages.

1. Unlike silicon, of which there is a plentiful supply in nature in the form of sand, GaAs is a rare commodity, and therefore more expensive.
2. GaAs is a compound made of two elements, Ga and As, and therefore is unstable at high temperatures.
3. It is very brittle, making it impossible to have large wafers. As a consequence, at this time no more than 100,000 transistors can be placed on a single chip. Contrast this to the millions of transistors for silicon-based chips.
4. The GaAs yields are much lower than for silicon, making the cost per chip much more expensive than for silicon chips.

These problems make the building of an entire computer based on GaAs a visionary project, if not an impossible one. This was the case for the CRAY III supercomputer, which was based on GaAs, and the buses ran at speeds of multiple GHz; but the project was also several years behind and millions of dollars over budget, so it was eventually abandoned and the company went out of business.

## Review Questions

1. What is the fan-out of the "0" state?
2. If the fan-out of "low" and "high" are 10 and 15, respectively, what is the fan-out?
3. If  $I_{OL} = 12 \text{ mA}$ ,  $I_{OH} = 3 \text{ mA}$  for the driver, and  $I_{IL} = 1.6 \text{ mA}$ ,  $I_{IH} = 40 \text{ A}$  for the load, find the fan-out.
4. Why do  $I_{IL}$  and  $I_{OH}$  have negative signs in many TTL books?
5. What are the 74xx244 and 74xx245 used for?
6. What is capacitive derating?
7. Ground bounce happens when the output makes a transition from \_\_\_\_\_ to \_\_\_\_\_.
8. Give one way to reduce ground bounce.
9. Transient current is due to transition of output from \_\_\_\_\_ to \_\_\_\_\_.
10. Why do high-speed logic gates using DIP packaging put the VCC and ground pins in the middle instead of the corners?
11. True or false. Soft error is permanent.
12. True or false. Hard error is permanent.
13. Alpha particle radiation causes \_\_\_\_\_ (soft, hard) errors.
14. FIT is in \_\_\_\_\_ (hours, months, years) of device operation.
15. What is the MTBF for 512 megabytes of memory if DRAM chips used are  $16M \times 8$  with FIT = 252?
16. What is the MTBF for 512 megabytes of memory if DRAM chips used are  $16M \times 8$  with FIT = 1000?

# Answers to Review Questions

## Section A.1

1. MOS is more power efficient, while bipolar is faster.
2. True
3. True
4. CMOS
5. NMOS
6. In the lower end, ALS, and in the higher end, FAST

## Section A.2

1. It is the number of loads that the driver can support and it is calculated by  $I_{OL}/I_{IL}$ .
2. 10
3.  $I_{OL}/I_{IL} = 12 \text{ mA}/1.6 \text{ mA} = 7$  and  $I_{OH}/I_{IH} = 3 \text{ mA}/40 \mu\text{A} = 75$ . Fan-out is 7, a lower number.
4. The negative sign indicates that these currents are flowing out of the IC (conventional current flow).
5. They are used for the line driver: the 74xx244 for unidirectional and 74xx245 for bidirectional lines.
6. It is signal delay caused by excessive load capacitance.
7. High, low
8. Make the ground pin length as small and short as possible.
9. Low, high
10. To make the self-inductance of pins VCC and GND small in order to reduce the ground and VCC bounce
11. False
12. True
13. Soft
14. Hours
15.  $453/32 = 14.1$  years since we have  $512\text{M} \times 8/16\text{M} \times 8 = 32$  chips
16. 3.56 years (114.15 years for one DRAM divided by 32 chips)