

# **Freescale ARM Cortex-M Embedded Programming**

**Using C Language**

*Muhammad Ali Mazidi  
Shujen Chen  
Sarmad Naimi  
Sepehr Naimi*



# **Freescale ARM Cortex-M Embedded Programming Using C Language**

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 Freescale ARM KL25Z chip. Throughout the book, we use C language to access the special function registers and program the Freescale ARM FRDM 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. Freescale provides Kinetis Software Development kit (SDK) [http://www.freescale.com/webapp/sps/site/prod\\_summary.jsp?code=KINETIS\\_SDK](http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=KINETIS_SDK) 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 Freescale Kinetis SDK.

## Freescale ARM Trainer

The Freescale has many inexpensive trainers for the ARM Cortex Kinetis. Among them is Freescale FRDM KL25Z board. Although we used the Freescale FRDM KL25Z board to test the programs, the programs runs on other Freescale kits as long as they are based on KL25 Series ARM® Cortex™-M0+ 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 and DAC 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 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 are 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. Appendix B provides a single source for KL25Z alternate pin functions. The CPU clock source is examined in Appendix C.

## 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 Freescale ARM FRDM 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 Freescale ARM FRDM 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 Freescale ARM FRDM Evaluation kit?

See the link below for Freescale ARM FRDM evaluation kit and KL25Z datasheet.

[http://www.freescale.com/webapp/sps/site/prod\\_summary.jsp?code=FRDM-KL25Z&tid=vanFRDM-KL25Z](http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=FRDM-KL25Z&tid=vanFRDM-KL25Z)

The above FRDM board uses the KL25Z128VLK4 chip. The KL25Z128VLK chip is part of the ARM Cortex-M0 from Freescales' and is often called Kinetis L series.

## **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)

## **Contact us**

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: Freescale ARM I/O Programming](#)

[Section 2.1: Freescale Freedom KL25Z128VLK4 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: Freescale ARM Timer Programming](#)

[Section 5.0: Introduction to counters and timers](#)

[Section 5.1: System Tick Timer](#)

[Section 5.2: Delay Generation with Freescale Timers](#)

[Section 5.3: Output Compare and TPM Channels](#)

[Section 5.4: Using Timer for Input Edge-time Capturing](#)

[Section 5.5: Using Timer as an Event Counter](#)

[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: Freescale 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 Freescale ARM](#)

[Answer to Review Questions](#)

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

[Section 7.1: ADC Characteristics](#)

[Section 7.2: ADC Programming with the Freescale KL25Z](#)

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

[Section 7.4: DAC Programming](#)

[Answers to Review Questions](#)

[Chapter 8: SPI Protocol and Devices](#)

[Section 8.1: SPI Bus Protocol](#)

[Section 8.2: SPI programming in Freescale ARM KL25Z](#)

[Section 8.3: MAX7221 SPI 7-Segment Driver](#)

[Answers to Review Questions](#)

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

[Section 9.1: I2C Bus Protocol](#)

[Section 9.2: I2C Programming in Freescale ARM KL25Z](#)

[Section 9.3: DS1337 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 Freescale ARM KL25Z](#)

[Answers to Review Questions](#)

[Chapter 12: Programming Graphic LCD](#)

[Section 12.1: Graphic LCDs](#)

[Section 12.2: Displaying Texts on Graphic 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](#)

### [Appendix B: KL25Z 80-pin Pinout](#)

### [Appendix C: System Clock Generation](#)

### [References](#)





# **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.

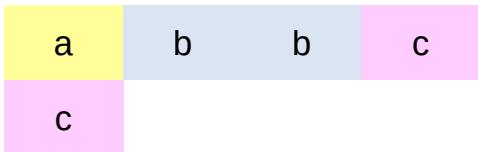
*unsigned char a;*

*unsigned short int b;*

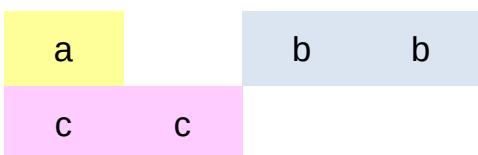
*unsigned short int c;*

## Solution:

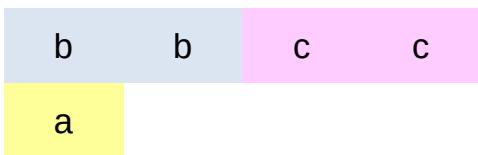
Unaligned allocation of variable c



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



Aligned allocation of variables by rearranging the variable sequence



---

## 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:

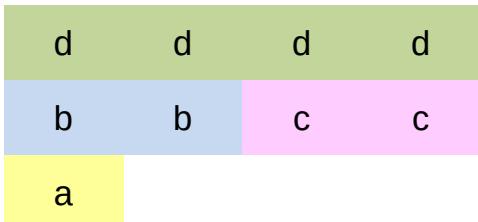
Unaligned allocation of variable c

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



## 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
<b>char</b>	1 byte	-128 to 127

<b>unsigned char</b>	1 byte	0 to 255
<b>short int</b>	2 bytes	-32,768 to 32,767
<b>unsigned int</b>	2 bytes	0 to 65,535
<b>long</b>	4 bytes	-2,147,483,648 to 2,147,483,647
<b>unsigned long</b>	4 bytes	0 to 4,294,967,295
<b>long long</b>	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<b>unsigned long long</b>	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
<b>int8_t</b>	1 byte	-128 to 127
<b>uint8_t</b>	1 byte	0 to 255
<b>int16_t</b>	2 bytes	-32,768 to 32,767
<b>uint16_t</b>	2 bytes	0 to 65,535
<b>int32_t</b>	4 bytes	-2,147,483,648 to 2,147,483,647
<b>uint32_t</b>	4 bytes	0 to 4,294,967,295
<b>int64_t</b>	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<b>uint64_t</b>	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 operands. 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: Freescale 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 Freescale ARM chip. Section 2.2 shows how to access the special function registers associated with the GPIO of Freescale ARM. In Section 2.2, we also use simple LEDs and switches 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: Freescale Freedom KL25Z128VLK4 Microcontroller

The FRDM-KL25Z is a low-cost development platform for Kinetis KL2x (KL24/25) MCUs built on ARM® Cortex™-M0+ processor. Freescale Freedom board uses the KL25Z128VLK4, an 80-pin microcontroller. The KL25Z128VLK4 chip has 128K bytes (128KB) of on-chip Flash memory for code, 16KB of on-chip SRAM for data, and a large number of on-chip peripherals, as shown in Figures 2-1 and 2-2.

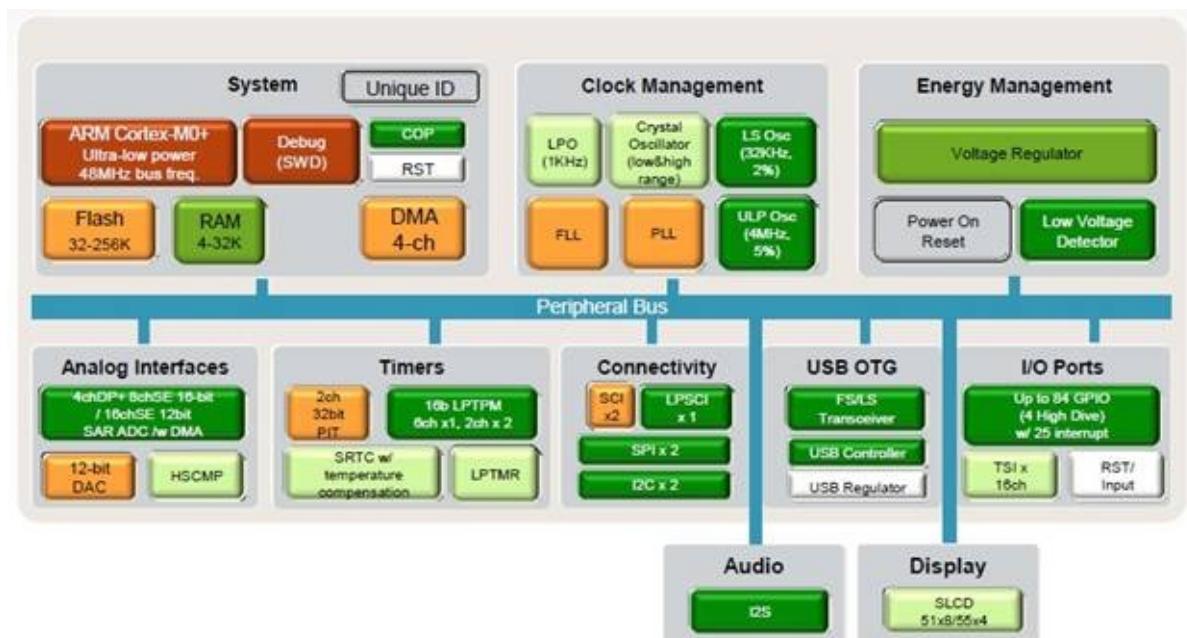


Figure 2-1: Freescale KL25Z128VLK4 Microcontroller High-Level Block Diagram

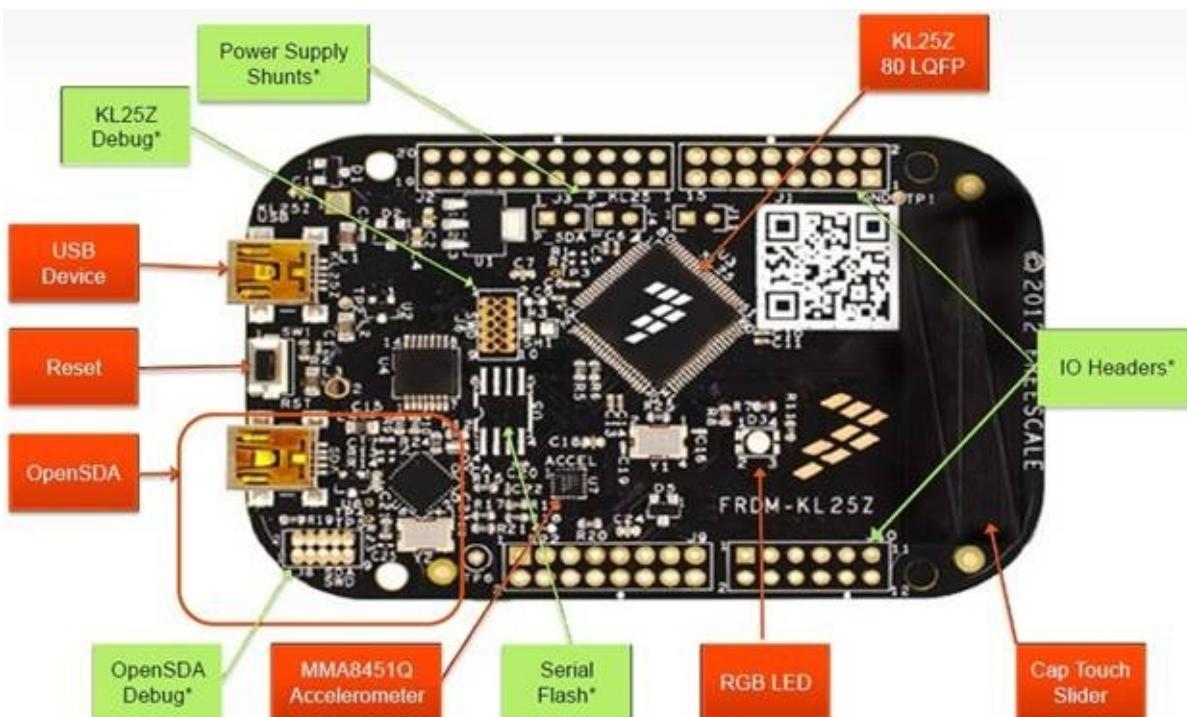


Figure 2-2: Freescale FRDM Trainer board

Note

For more information about this series see the following website:  
[http://www.freescale.com/webapp/sps/site/prod\\_summary.jsp?code=KL2x#](http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=KL2x#)

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 of KL25Z128VLK4 chip.

	Allocated size	Allocated address
<b>Flash</b>	128KB	0x00000000 to 0x0001FFFF
<b>SRAM</b>	16KB	0x1FFFF000 to 0x20002FFF
<b>I/O</b>	All the peripherals	0x400FF000 to 0x400FFFFFF

Table 2-1: Memory Map in KL25Z128VLK4

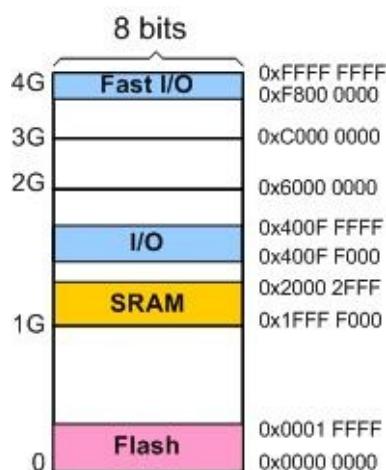


Figure 2-3: Memory Map

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

- 1) 128KB 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 16KB 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/Os, Timers, ADCs are mapped to addresses starting at 0x40000000. In KL25Z128VLK4 the upper limit is 0x400FFFFFF. For details see Table 2-4 in KL25Z128VLK4 reference manual. 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 KL25Z128VLK4 ARM are designated as port A to port E. The following shows the address range assigned to each GPIO port:

- GPIO Port A : 0x400F F000 to 0x400F F017
- GPIO Port B : 0x400F F040 to 0x400F F057
- GPIO Port C : 0x400F F080 to 0x400F F097
- GPIO Port D : 0x400F F0C0 to 0x400F F0D7
- GPIO Port E : 0x400F F100 to 0x400F F117

See Figure 2-4.

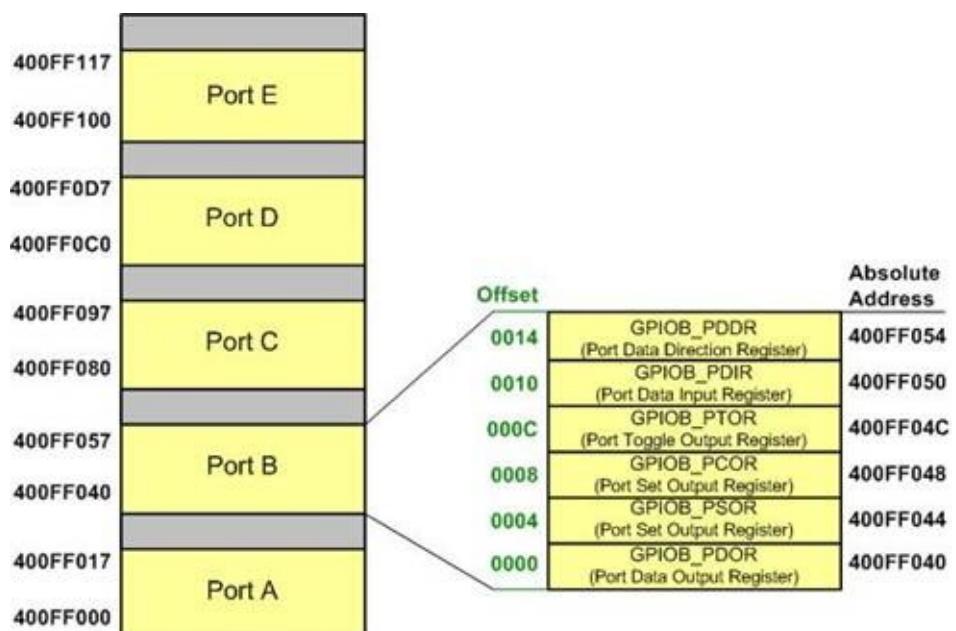


Figure 2-4: GPIO Memory Map

## Freescale's naming convention

Freescale part numbers have the following format:

Q KL## A FFF R T PP CC N

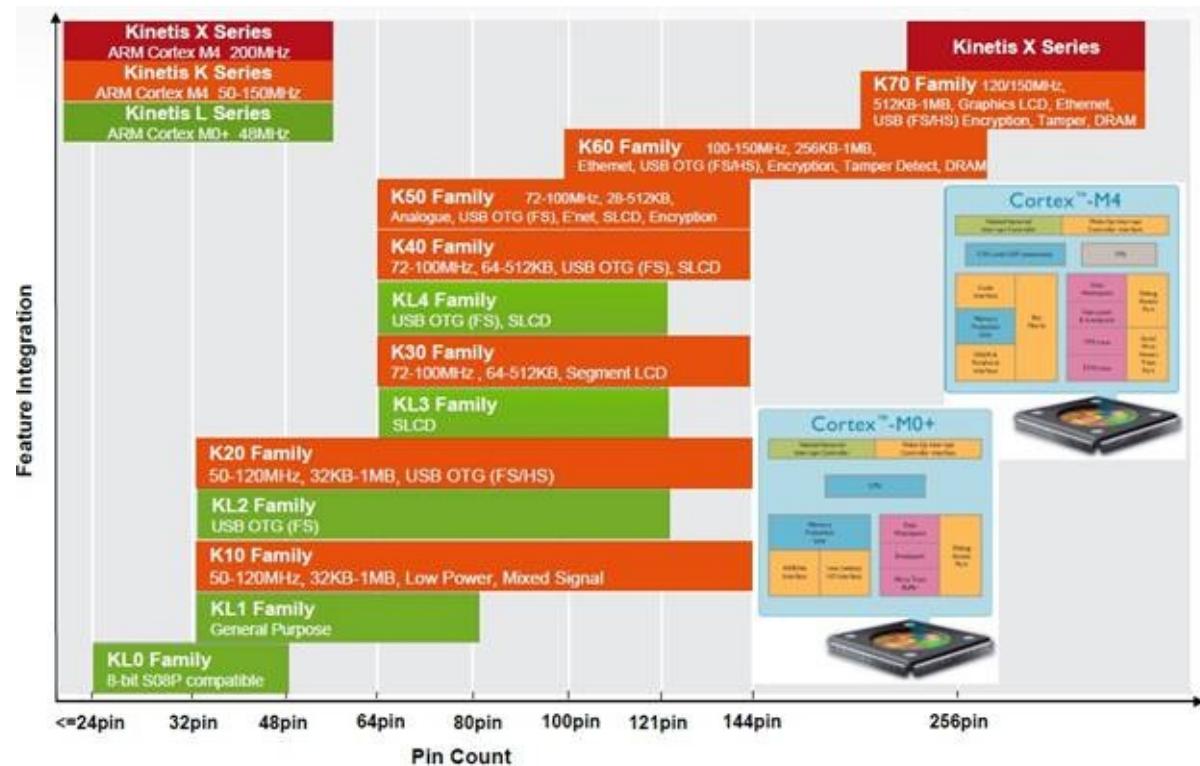
Table 2-2 lists the possible values for each field in the part number (not all combinations are valid):

Field	Description	Some Valid Values
Q	Qualification status	<ul style="list-style-type: none"> <li>• M = Fully qualified, general market flow</li> <li>• P = Prequalification</li> </ul>

<b>KL##</b>	Kinetis family	• KL25
<b>A</b>	Key attribute	• Z = Cortex-M0+
<b>FFF</b>	Program Flash memory size	<ul style="list-style-type: none"> <li>• 32 = 32 KB</li> <li>• 64 = 64 KB</li> <li>• 128 = 128 KB</li> </ul>
<b>T</b>	Temperature range (°C)	<ul style="list-style-type: none"> <li>• V = -40 to 105</li> </ul>
<b>PP</b>	Package identifier	<ul style="list-style-type: none"> <li>• FM = 32 QFN (5 mm x 5 mm)</li> <li>• FT = 48 QFN (7 mm x 7 mm)</li> <li>• LH = 64 LQFP (10 mm x 10 mm)</li> <li>• LK = 80 LQFP (12 mm x 12 mm)</li> </ul>
<b>CC</b>	Maximum CPU frequency (MHz)	<ul style="list-style-type: none"> <li>• 4 = 48 MHz</li> </ul>

**Table 2-2: Fields Values Description**

In Figure 2-5 the Kinetis Families and their features are shown.



**Figure 2-5: Kinetis MCU Portfolio**

## Review Questions

1. KL25Z128VLK4 has \_\_\_\_\_ KB of on-chip Flash memory.
2. KL25Z128VLK4 has \_\_\_\_\_ KB of on-chip SRAM memory.
3. KL25Z128VLK4 Flash memory is used mainly for (program code, data).
4. KL25Z128VLK4 SRAM memory is used for (program code, data).

5. Give the address space assigned to the Flash memory of KL25Z128VLK4.
6. Give the meaning of LK in KL25Z128VLK4.



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

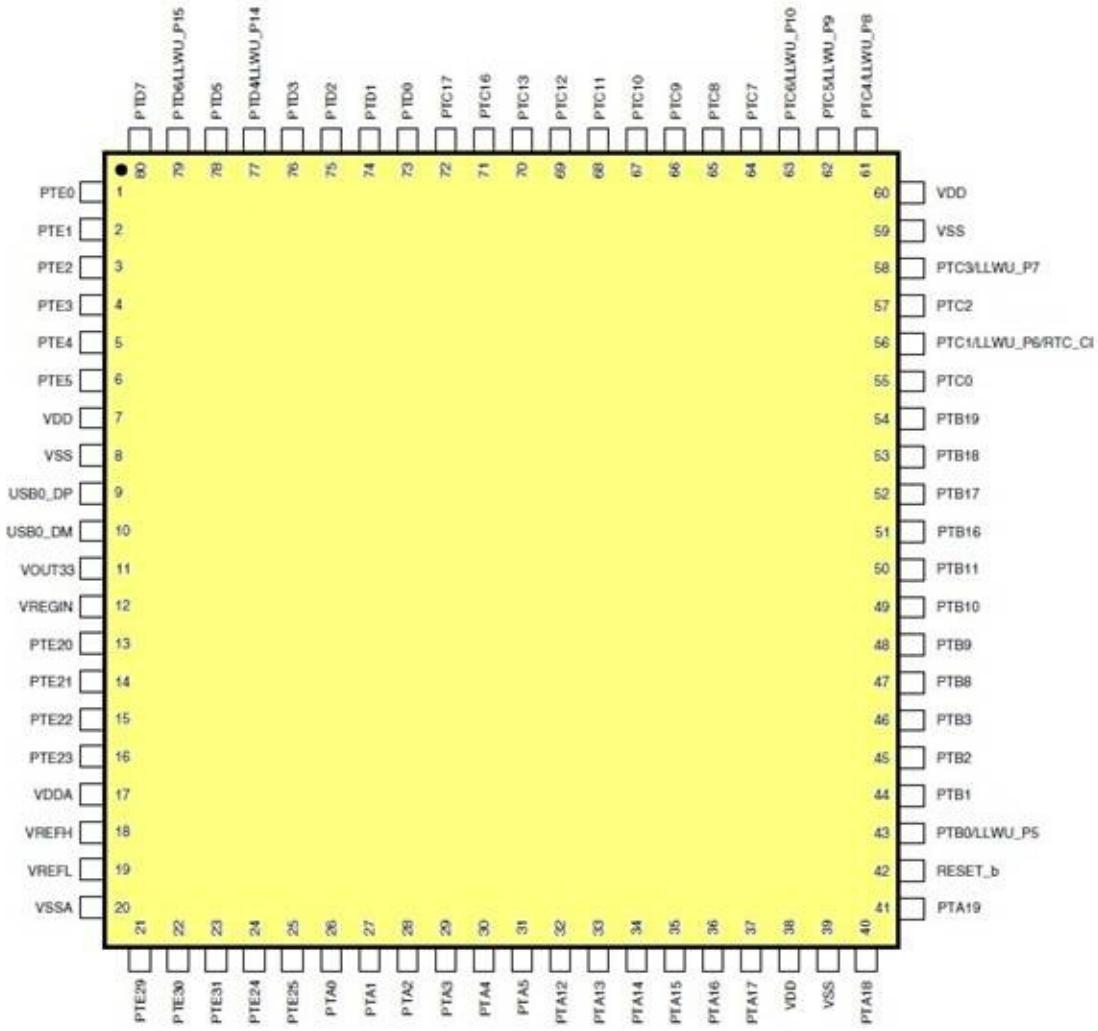
While memory holds code and data for the CPU to process, the I/O ports are used by the CPU to access input and output devices. In the microcontroller, we have two types of I/O. They are:

- 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 Freescale FRDM board

In Freescale ARM chips, I/O ports are named with alphabets A, B, C, and so on. Each port can have up to 32 pins and they are designated as PTA0-PTA31, PTB0-PTB31, and so on. It must be noted that not all 32 pins of each port are implemented. The ARM chip used in FRDM board is in Kinetis L series with part number KLxxZ128VLK4. It has ports A, B, C, D, and E. See Figure 2-6.



**Figure 2-6: KL25Z128VLK4 Pin-out**

Notice from Figure 2-6, for each port only a limited number of pins are implemented. For example, for Port A we have only PTA0-PTA5 and PTA12-PTA20 pins.

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 PTA-PTE for APB are as follow:

- GPIO Port A (APB): 0x400F F000
- GPIO Port B (APB): 0x400F F040
- GPIO Port C (APB): 0x400F F080
- GPIO Port D (APB): 0x400F F0C0
- GPIO Port E (APB): 0x400F F100

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

- GPIO Port A (AHB): 0xF80F F000
- GPIO Port B (AHB): 0xF80F F040

- GPIO Port C (AHB): 0xF80F F080
- GPIO Port D (AHB): 0xF80F F0C0
- GPIO Port E (AHB): 0xF80F F100

Notice that Freescale datasheet refers to AHB as FGPIO (Fast GPIO). 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 many 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*. 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 the device. See Figure 2-7.

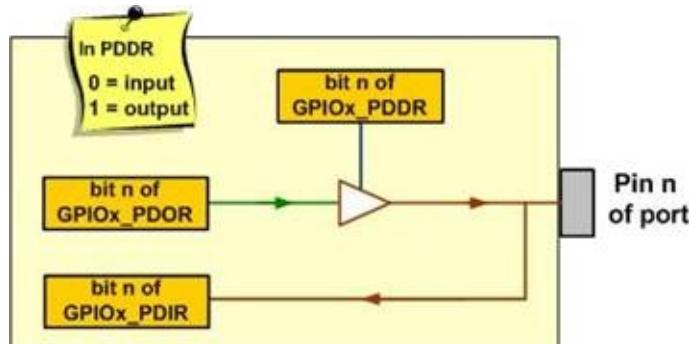


Figure 2-7: The Data and Direction Registers and a Simplified View of an I/O pin

## Port Data Output Register (GPIOx\_PDOR) in Freescale ARM

The Port Data Output Register (GPIOx\_PDOR) is located at the offset address of 0x0000 from the Base address of that port. This is shown below.

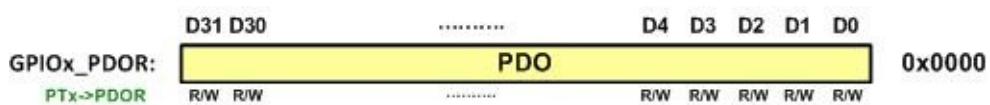


Figure 2-8: GPIOx\_PDOR (Port Data Output Register)

## Direction Register in Freescale ARM

In the case of Freescale 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. The address of the GPIO Direction register is located at the offset address of 0x0014 from the Base

address of that port. This is shown below.

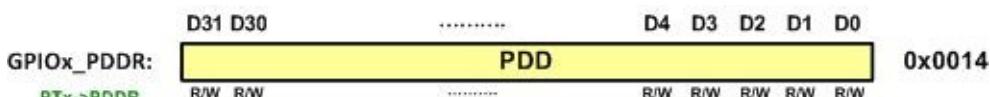


Figure 2-9: GPIOx\_PDDR (Port Data Direction Register)

For example, by writing 0x03 (00000011 in binary) into the GPIOA\_PDDR register, pins 0 and 1 of PORTA become outputs while the other pins become inputs.

See Example 2-1 and Table 2-3.

### Example 2-1

Find the physical address of the GPIO DATA and GPIO Direction registers for PORTA if the Base address of the PORTA is 0x400F F000.

#### Solution:

The physical address location of the GPIO Data Out for PORTA is 0x400F F000+0000=0x400F F000.

The physical address location of GPIO Direction for the PORTA is 0x400F F000+0x0014=0x400F F014.

Address	Name	Description	Type	Reset Value
0x400F F000	GPIOA_PDOR	Port Data Output Register	R/W	0x00000000
0x400F F004	GPIOA_PSOR	Port Set Output register	W (always reads 0)	0x00000000
0x400F F008	GPIOA_PCOR	Port Clear Output Register	W (always reads 0)	0x00000000
0x400F F00C	GPIOA_PTOR	Port Toggle Output Register	W (always reads 0)	0x00000000
0x400F F010	GPIOA_PDIR	Port Data Input Register	R	0x00000000
0x400F F014	GPIOA_PDDR	Port Data Direction Register	R/W	0x00000000

Table 2-3: Some GPIO Registers for PORTA

Table 2-3 shows some of the registers associated with PORTA. After we configure the Direction register for output, we can use Data Out, Set Out, Clear Out, and Toggle Out registers. The same way, after Direction register is configured for input, the Data Input register is used to bring data into CPU from the pins. It must be noted that we have these registers for all the ports A to E in the Freescale ARM. Now, to access the I/O pins of KL25Z128VLK4 ARM chip used in FRDM board, we need to examine Pin selection and clock enable registers.

## Alternate pin functions and the simple GPIO

Each pin of the Freescale ARM chip can be used for one of several functions including GPIO. We choose the function by programming a 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 of its on-chip features. For example, a given pin can be used as simple digital I/O (GPIO), analog input, or I2C pin. Of course not all at the same time. We must make sure that a pin is assigned to only one peripheral function at a time.

The PORTx\_PCRn (Portx Pin Control) special function register allows us to program a pin to be used for a given alternate function. It must be noted that each pin of ports A-E has its own PORTx\_PCRn register. The x is used for Ports A to E and n is used for pin number 0 to 31. As we mentioned earlier, each port of A to E can have up to 32 pins. Not all of the pins for a given port are implemented. The most important bits of PORTx\_PCRn are D10-D8 (Mux control). Upon reset, ports A to E are disabled. To use a pin as simple digital I/O, we must choose MUX=001 option. See Figure 2-10.

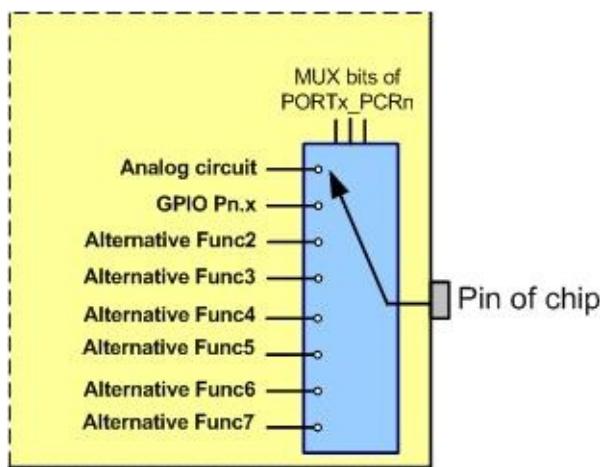


Figure 2-10: Alternative Functions of Pins

With the PORTx\_PCRn register, not only we select the alternate I/O function of a given pin, we can also control its Drive Strength and its internal Pull-up (or Pull-down) resistor. See Figure 2-11 and Table 2-4. The D1 (PE, Pull enable) bit of the PORTx\_PCRn is used to enable the internal Pull resistor option. If PE=1, then we use the D0 bit (PS, pull Select) to enable the pull-up (or pull-down) option. We

can also control the drive capability (fan-out and fan-in. See Appendix A) of a digital I/O pin with D6 (Drive Strength Enable) bit. These options are widely used when connecting a pin to a switch or LED. In future chapters, we will see how to use PORTx\_PCRn bits to choose other pin functions when we examine the on-chip peripherals of KL25Z chip.

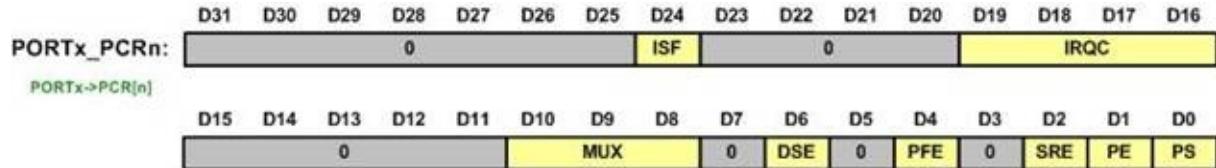


Figure 2-11: PORTx\_PCRn Register is used to select alternate pin functions (from Sec 11.5 of KL25 Ref. Man.)

BIT	Field	Description																
0	Pull Select (PS)	If the PE field is set, the field chooses between pull-up and pull-down resistors. 0: pull-down resistor, 1: pull-up resistor																
1	Pull Enable (PE)	0: Disable the internal pull resistors 1: Enable the internal pull resistors																
2	Slew Rate Enable (SRE)	0: Fast slew rate 1: Slow slew rate																
4	Passive Filter Enable (PFE)	0: Passive input filter is disabled 1: Passive input filter is enabled																
6	Drive Strength Enable (DSE)	0: Low drive strength 1: High drive strength																
10-8	Pin Mux Control (MUX)	<table border="1"> <thead> <tr> <th>value</th><th>Meaning</th></tr> </thead> <tbody> <tr> <td>000</td><td>Pin disabled (analog)</td></tr> <tr> <td>001</td><td>Alternative 1 (GPIO)</td></tr> <tr> <td>010</td><td>Alternative 2 (Chip-specific)</td></tr> <tr> <td>011</td><td>Alternative 3 (Chip-specific)</td></tr> <tr> <td>100</td><td>Alternative 4 (Chip-specific)</td></tr> <tr> <td>101</td><td>Alternative 5 (Chip-specific)</td></tr> <tr> <td>110</td><td>Alternative 6 (Chip-specific)</td></tr> </tbody> </table>	value	Meaning	000	Pin disabled (analog)	001	Alternative 1 (GPIO)	010	Alternative 2 (Chip-specific)	011	Alternative 3 (Chip-specific)	100	Alternative 4 (Chip-specific)	101	Alternative 5 (Chip-specific)	110	Alternative 6 (Chip-specific)
value	Meaning																	
000	Pin disabled (analog)																	
001	Alternative 1 (GPIO)																	
010	Alternative 2 (Chip-specific)																	
011	Alternative 3 (Chip-specific)																	
100	Alternative 4 (Chip-specific)																	
101	Alternative 5 (Chip-specific)																	
110	Alternative 6 (Chip-specific)																	

		<b>111</b>	Alternative 7 (Chip-specific)
			By setting the field, an interrupt (or a DMA request) is generated when the pin is triggered. (See Chapter 6)
		<b>Value</b>	<b>Meaning</b>
		<b>0000</b>	Interrupt and DMA request is disabled.
		<b>0001</b>	DMA request on rising edge
		<b>0010</b>	DMA request on falling edge
		<b>0011</b>	DMA request on each edge
		<b>1000</b>	Interrupt when logic 0
		<b>1001</b>	Interrupt on rising edge
		<b>1010</b>	Interrupt on falling edge
		<b>1011</b>	Interrupt on either edge
		<b>1100</b>	Interrupt when logic 1
		<b>Others</b>	Reserved
<b>19-16</b>	Interrupt Configuration (IRQC)		
<b>24</b>	Interrupt Status Flag (ISF)	0: interrupt is not detected 1: interrupt is detected (It is covered in Chapter 6)	
<b>Others</b>	Reserved	The gray fields are reserved and are not used.	

**Table 2-4: PORTx\_PCRn Register**

9	USB0_DP	USB0_DP	USB0_DP						
10	USB0_DM	USB0_DM	USB0_DM						
11	VOUT33	VOUT33	VOUT33						
12	VREGIN	VREGIN	VREGIN						
13	PTE20	ADC0_DP0/ ADC0_SE0	ADC0_DP0/ ADC0_SE0	PTE20		TPM1_CH0	UART0_TX		
14	PTE21	ADC0_DM0/ ADC0_SE4a	ADC0_DM0/ ADC0_SE4a	PTE21		TPM1_CH1	UART0_RX		
15	PTE22	ADC0_DP3/ ADC0_SE3	ADC0_DP3/ ADC0_SE3	PTE22		TPM2_CH0	UART2_TX		
16	PTE23	ADC0_DM3/ ADC0_SE7a	ADC0_DM3/ ADC0_SE7a	PTE23		TPM2_CH1	UART2_RX		
17	VDDA	VDDA	VDDA						
18	VREFH	VREFH	VREFH						
19	VREFL	VREFL	VREFL						
20	VSSA	VSSA	VSSA						
21	PTE29	CMP0_IN5/ ADC0_SE4b				TPM0_CH2	TPM_CLKIN0		
22	PTE30	DAC0_OUT/ ADC0_SE23/ CMP0_IN4				TPM0_CH3	TPM_CLKIN1		
23	PTE31	DISABLED		PTE31		TPM0_CH4			
24	PTE24	DISABLED		PTE24		TPM0_CH0		I2C0_SCL	
25	PTE25	DISABLED		PTE25		TPM0_CH1		I2C0_SDA	
26	PTA0	SWD_CLK	TSI0_CH1	PTA0		TPM0_CH5			S
27	PTA1	DISABLED	TSI0_CH2	PTA1	UART0_RX	TPM0_CH0			
28	PTA2	DISABLED	TSI0_CH3	PTA2	UART0_TX	TPM2_CH1			
29	PTA3	SWD_DIO	TSI0_CH4	PTA3	I2C1_SCL	TPM0_CH0			S
30	PTA4	NMI_b	TSI0_CH5	PTA4	I2C1_SDA	TPM0_CH1			
31	PTA5	DISABLED		PTA5	USB_CLKIN	TPM0_CH2			
32	PTA12	DISABLED		PTA12		TPM1_CH0			
33	PTA13	DISABLED		PTA13		TPM1_CH1			
34	PTA14	DISABLED		PTA14	SPI0_PCS0	UART0_TX			
35	PTA15	DISABLED		PTA15	SPI0_SCK	UART0_RX			
36	PTA16	DISABLED		PTA16	SPI0_MOSI			SPI0_MISO	
37	PTA17	DISABLED		PTA17	SPI0_MISO			SPI0_MOSI	
38	VDD	VDD	VDD						

39	VSS	VSS	VSS						
40	PTA18	EXTAL0	EXTAL0	PTA18		UART1_RX	TPM_CLKIN0		
41	PTA19	XTAL0	XTAL0	PTA19		UART1_TX	TPM_CLKIN1		LPTMR0_ALT1
42	RESET_b	RESET_b		PTA20					
43	PTB0/ LLWU_P5	ADC0_SE8/ TSI0_CH0	ADC0_SE8/ TSI0_CH0		I2C0_SCL	TPM1_CH0			
44	PTB1	ADC0_SE9/ TSI0_CH6	ADC0_SE9/ TSI0_CH6	PTB1	I2C0_SDA	TPM1_CH1			
45	PTB2	ADC0_SE12/ TSI0_CH7	ADC0_SE12/ TSI0_CH7	PTB2	I2C0_SCL	TPM2_CH0			
46	PTB3	ADC0_SE13/ TSI0_CH8	ADC0_SE13/ TSI0_CH8	PTB3	I2C0_SDA	TPM2_CH1			
47	PTB8	DISABLED		PTB8		EXTRG_IN			
48	PTB9	DISABLED		PTB9					
49	PTB10	DISABLED		PTB10	SPI1_PCS0				
50	PTB11	DISABLED		PTB11	SPI1_SCK				
51	PTB16	TSI0_CH9	TSI0_CH9	PTB16	SPI1_MOSI	UART0_RX	TPM_CLKIN0	SPI1_MISO	
52	PTB17	TSI0_CH10	TSI0_CH10	PTB17	SPI1_MISO	UART0_TX	TPM_CLKIN1	SPI1_MOSI	
53	PTB18	TSI0_CH11	TSI0_CH11	PTB18		TPM2_CH0			
54	PTB19	TSI0_CH12	TSI0_CH12	PTB19		TPM2_CH1			
55	PTC0	ADC0_SE14/ TSI0_CH13	ADC0_SE14/ TSI0_CH13	PTC0		EXTRG_IN		CMP0_OUT	
56	PTC1/ LLWU_P6/ RTC_CLKIN	ADC0_SE15/ TSI0_CH14	ADC0_SE15/ TSI0_CH14	PTC1/ LLWU_P6/ RTC_CLKIN	I2C1_SCL		TPM0_CH0		
57	PTC2	ADC0_SE11/ TSI0_CH15	ADC0_SE11/ TSI0_CH15	PTC2	I2C1_SDA		TPM0_CH1		
58	PTC3/ LLWU_P7	DISABLED		PTC3/ LLWU_P7		UART1_RX	TPM0_CH2	CLKOUT	
59	VSS	VSS	VSS						
60	VDD	VDD	VDD						
61	PTC4/ LLWU_P8	DISABLED		PTC4/ LLWU_P8	SPI0_PCS0	UART1_TX	TPM0_CH3		
62	PTC5/ LLWU_P9	DISABLED		PTC5/ LLWU_P9	SPI0_SCK	LPTMR0_ALT2			CMP0_OUT
63	PTC6/ LLWU_P10	CMP0_IN0	CMP0_IN0	PTC6/ LLWU_P10	SPI0_MOSI	EXTRG_IN		SPI0_MISO	
64	PTC7	CMP0_IN1	CMP0_IN1	PTC7	SPI0_MISO			SPI0_MOSI	

65	PTC8	CMP0_IN2	CMP0_IN2	PTC8	I2C0_SCL	TPM0_CH4			
66	PTC9	CMP0_IN3	CMP0_IN3	PTC9	I2C0_SDA	TPM0_CH5			
67	PTC10	DISABLED		PTC10	I2C1_SCL				
68	PTC11	DISABLED		PTC11	I2C1_SDA				
69	PTC12	DISABLED		PTC12			TPM_CLKN0		
70	PTC13	DISABLED		PTC13			TPM_CLKN1		
71	PTC16	DISABLED		PTC16					
72	PTC17	DISABLED		PTC17					
73	PTD0	DISABLED		PTD0	SPI0_PCS0		TPM0_CH0		
74	PTD1	ADC0_SE5b	ADC0_SE5b	PTD1	SPI0_SCK		TPM0_CH1		
75	PTD2	DISABLED		PTD2	SPI0_MOSI	UART2_RX	TPM0_CH2	SPI0_MISO	
76	PTD3	DISABLED		PTD3	SPI0_MISO	UART2_TX	TPM0_CH3	SPI0_MOSI	
77	PTD4/ LLWU_P14	DISABLED		PTD4/ LLWU_P14	SPI1_PCS0	UART2_RX	TPM0_CH4		
78	PTD5	ADC0_SE6b	ADC0_SE6b	PTD5	SPI1_SCK	UART2_TX	TPM0_CH5		
79	PTD6/ LLWU_P15	ADC0_SE7b	ADC0_SE7b	PTD6/ LLWU_P15	SPI1_MOSI	UART0_RX		SPI1_MISO	
80	PTD7	DISABLED		PTD7	SPI1_MISO	UART0_TX		SPI1_MOSI	

Figure 2-12: KL25Z Alternative Pin Functions (Chapter 10 of KL25 Ref. Man.)

## Example 2-2

In a given FRDM board the PTB18 and PTB19 are connected to LEDs to be used as simple I/O.

- Configure the GPIOB Direction register for pins PTB18 and PTB19 to be digital output.
- Give the address of the register in part a.
- Configure the PORTx\_PCRn registers for PTB18 and PTB19 pins. Assume slow slew rate, high drive, and no pull-up.
- Give the address of each register in part C.

### Solution:

- For any pins of PORTB to be used as an output, we need to set bits in GPIOB\_PDDR (PORTB direction) to high. Now, we have 0x000C 0000 since D18 and D19 bit are set high. See Sec. 41.2.6 of KL25 Ref. Man. Notice for the bit D18 (pin 18) and bit D19 (pin 19) we have 0b0000 0000 0000 1100 0000 0000 0000 0000 in binary or 0x000C 0000.

b) The PORTB\_PDDIR register is located at address 0x4000 F054 location. See Section 41.2 KL25 Ref. manual.

c) We have a PORTn\_PCRx register for each implemented pin. Section 11.5.1 of KL25 Ref. Manual gives the description of the each bit for PORTn\_PCRx, which Figure 2-12 is taken from. Therefore for PTB18, we have PORTB\_PCR18=0x0000 0144 to configure it for I/O alternate, High Drive, slow slew rate, no pull resistor. The same is for PB19 which is PORTB\_PCR19=0x0000 0144.

d) Examine memory map table in Sec. 11.5 of KL25 Ref. Manual. Notice we have an address for a register (PORTn\_PCRx) to set the characteristics of each pin of each port. For PTB18 pin we have register PORTB\_PCR18 and is located at address 0x4004 A048. Register PORTB\_PCR19 for PTB19 pin is located at address 0x4004 A04C.

---

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

The System Clock Gating Control Register 5 (SIM\_SCGC5) is used to enable the clock source for the I/O port circuitry among other things. 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 SIM\_SCGC5 special function register for all the GPIO ports and some of the bits of this register are used to enable the clock source to Ports A to E. In the case of Freescale KL25Z128VLK4, since we have only ports A through E, many of the bits of this register are unused or used for other functions. Chapter 12 of KL25 Ref. Manual shows the details of SIM\_SCGC5 register. The register bits are also shown in Figure 2-13.

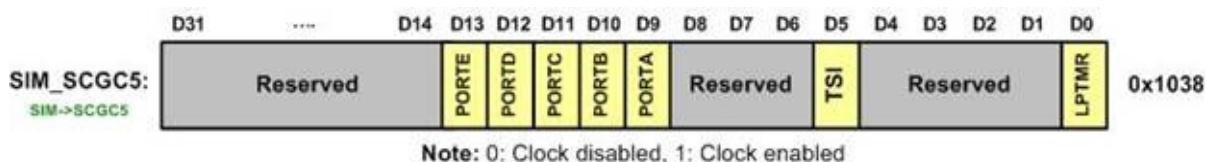


Figure 2-13: SIM\_SCGC5 (System Clock Gating Control Register 5) Register

In SIM\_SCGC5, the Base address is 0x4004 7000 and the offset is 0x1038; as a result, the physical address is  $0x4004\ 7000 + 0x1038 = 0x4004\ 8038$ . See Example 2-3.

---

### Example 2-3

Show how a) to enable the clock to PTB and b) enable the digital I/O feature of pins for PTB18 and PTB19.

#### Solution:

- To enable the clock source to PTB, we need to set HIGH the D10 of the SIM\_SCGC5 register. We can OR hex value 0x400 (0b0100 0000 0000) with

SIM\_SCGC5 register to make sure it leaves clock source to other ports unchanged. See Figure 2-13.

- b) We need to write value 0b0001 0000 0000 (0x0100 in hex) to PORTB\_PCRn register located at address 0x400F 8038. See Table 2-4. Also see SIM memory map in Chapter 12 of KL25 Ref. Manual.

---

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.

## LED connection in Freescale FRDM board

In the Freescale Freedom board we have a tri-color RGB LED connected to PTB18 (red), PTB19 (green), and PTD1 (blue). The tri-color RGB (red, blue, green) LED is popular in many trainer kit for embedded systems.

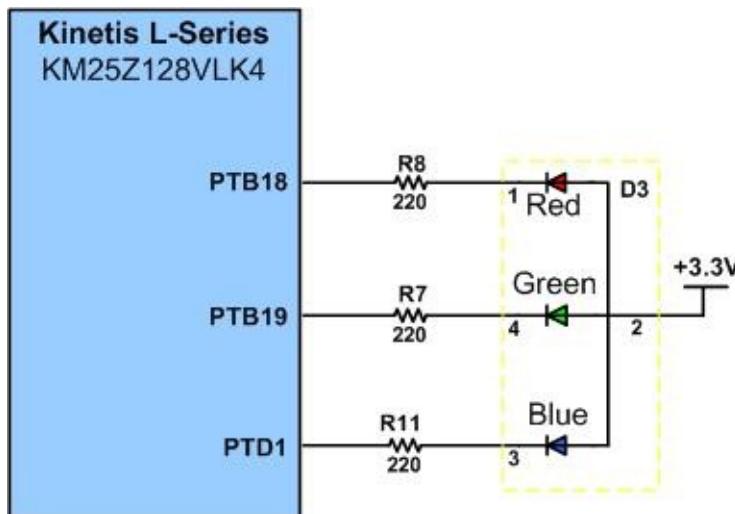


Figure 2-14: LED connection to PTB and PTD Freescale FRDM board

## Toggling LEDs in Freescale FRDM board in C

To toggle the green LEDs of the FRDM board, the following steps must be followed.

- 1) enable the clock to PORTB, since access is denied to the port registers until the clock is enabled,
- 2) configure PortB\_PCR19 (Pin Control Register) to select GPIO function for PTB19,
- 3) set the Direction register bit 19 of PTB as output,
- 4) write HIGH to PTB19 in data register,
- 5) call a delay function,
- 6) write LOW to PTB19 in data register,
- 7) call a delay function,

## 8) Repeat steps 4 to 7.

Program 2-1 shows one way to toggle the green LED continuously.

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

```
/* p2_1.c Toggling LED in C using registers by addresses
 * This program toggles green LED for 0.5 second ON and 0.5 second OFF.
 * The green LED is connected to PTB19.
 * The LEDs are low active (a '0' turns ON the LED).
 */

/* System Integration Module System Clock Gating Control Register 5*/
#define SIM_SCGC5    (*((volatile unsigned int*)0x40048038))
/* Port B Pin Control Register 19*/
#define PORTB_PCR19  (*((volatile unsigned int*)0x4004A04C))
/* Port B Data Direction Register */
#define GPIOB_PDDR   (*((volatile unsigned int*)0x400FF054))
/* Port B Data Output Register */
#define GPIOB_PDOR   (*((volatile unsigned int*)0x400FF040))

int main (void) {
    void delayMs(int n);
    SIM_SCGC5 |= 0x400;           /* enable clock to Port B */
    PORTB_PCR19 = 0x100;          /* make PTB19 pin as GPIO (See Table 2-4) */
    GPIOB_PDDR |= 0x80000;        /* make PTB19 as output pin */
    while (1) {
        GPIOB_PDOR &= ~0x80000;    /* turn on green LED */
        delayMs(500);
        GPIOB_PDOR |= 0x80000;     /* turn off green LED */
        delayMs(500);
    }
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
```

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

### Program 2-2: Toggling Green LED in Assembly Language

```
; p2_2.s Toggling LED using assembly language
; This program toggles green LED for 0.5 second ON and 0.5 second OFF.
; The green LED is connected to PTB19.
; The LEDs are low active (a '0' turns ON the LED).

; System Integration Module System Clock Gating Control Register 5
SIM_SCGC5 EQU 0x40048038
; Port B Pin Control Register 19
PORTB_PCR19 EQU 0x4004A04C
; Port B Pin Direction Register
GPIOB_PDDR EQU 0x400FF054
; Port B Pin Set Output Register
GPIOB_PDOR EQU 0x400FF040

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

__main
; enable clock to Port B
LDR     R0, =SIM_SCGC5 ; load SCGC5 Reg in R1
LDR     R1, [R0]
LDR     R2, =0x400        ; load bit 10 mask
ORRS   R1, R2          ; OR with bit mask
STR    R1, [R0]         ; store back to SCGC5
; make PTB19 pin as GPIO
LDR     R0, =PORTB_PCR19
LDR     R1, =0x100        ; pin mux GPIO (See Table 2-4)
STR    R1, [R0]         ; store in PCR19
; make PTB19 as output pin
LDR     R0, =GPIOB_PDDR ; load Dir Reg in R1
```

```

LDR      R1, [R0]
LDR      R2, =0x80000 ; load bit 19 mask
ORRS    R1, R2        ; OR with bit mask
STR      R1, [R0]      ; store back to Dir Reg

loop
; turn on green LED
LDR      R0, =GPIOB_PDOR ; load Data Reg in R1
LDR      R1, [R0]
LDR      R2, =0x80000 ; load bit 19 mask
MVNS    R2, R2        ; complement bit mask
ANDS    R1, R2        ; AND with bit mask
STR      R1, [R0]      ; store back to Data Reg
; delay for 0.5 second
LDR      R0, =500
BL      delayMs

; turn off green LED
LDR      R0, =GPIOB_PDOR ; load Data Reg in R1
LDR      R1, [R0]
LDR      R2, =0x80000 ; load bit 19 mask
ORRS    R1, R2        ; OR with bit mask
STR      R1, [R0]      ; store back to Data Reg
; delay for 0.5 second
LDR      R0, =500
BL      delayMs

; repeat the loop
B      loop

; This subroutine performs a delay of N ms
; (for 41.94 MHz CPU clock).
; N is the value in R0.

delayMs
MOVS    R0, R0        ; if N = 0, return
BNE     L1
BX      LR

L1      LDR      R1, =14022 ; do inner loop 14022 times

```

```

L2      SUBS    R1, #1           ; inner loop
BNE     L2
SUBS   R0, #1           ; do outer loop N times
BNE     L1
BX     LR
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 KL25Z128VLK4.

Freescale provides the header file in either CodeWarrior IDE or Kinetis Design Studio IDE. If you have downloaded either of them, you will find the header file at:

```

C:\Freescale\CW MCU v10.5\MCU\ProcessorExpert\lib\Kinetis\iofiles\MKL25Z4.H
Or
C:\Freescale\_1.0\eclipse\ProcessorExpert\lib\Kinetis\iofiles\MKL25Z4.H

```

The version numbers may vary depending on when the IDE is downloaded. Program 2-3 is the same program as Program 2-1 except it uses the register definitions from Freescale.

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

```

/* p2_3.c Toggling LED in C using Freescale header file register definitions.
 * This program toggles green LED for 0.5 second ON and 0.5 second OFF.
 * The green LED is connected to PTB19.
 * The LEDs are low active (a '0' turns ON the LED).
 */

#include "C:\Freescale\CW MCU
v10.5\MCU\ProcessorExpert\lib\Kinetis\iofiles\MKL25Z4.H"
/*
#include
"C:\Freescale\_1.0\eclipse\ProcessorExpert\lib\Kinetis\iofiles\MKL25Z4.H"
*/
int main (void) {

```

```

void delayMs(int n);

SIM_SCGC5 |= 0x400;           /* enable clock to Port B */

PORTB_PCR19 = 0x100;          /* make PTB19 pin as GPIO */

GPIOB_PDDR |= 0x80000;        /* make PTB19 as output pin */

while (1) {

    GPIOB_PDOR &= ~0x80000; /* turn on green LED */

    delayMs(500);

    GPIOB_PDOR |= 0x80000; /* turn off green LED */

    delayMs(500);

}

}

/* Delay n milliseconds

 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().

 */

void delayMs(int n) {

    int i;

    int j;

    for(i = 0 ; i < n; i++)

        for (j = 0; j < 7000; j++) {}

}

```

Keil MDK-ARM uses a different syntax to define the registers to be in compliant with CMSIS (Cortex Microcontrollers Software Interface Standard). In this syntax, 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 B is referred to as `PTB->PDDR` and the Data Register of Port B is referred to as `PTB->PDOR` and so on.

With Keil MDK-ARM version 5, the header file is provided in the Device Family Pack download. You will find it at

```
C:\Keil_v5\ARM\Pack\Keil\Kinetis_KLxx_DFP\1.2.0\Device\Include\MKL25Z4.h.
```

Program 2-1 is rewritten with this header file as Program 2-4. When you start a Keil MDK-ARM project and choose a Freescale MKL25Z device, the project wizard will add the location of this header file in the compiler search path. In the program, you only need to specify the file name as:

```
#include <MKL25Z4.H>
```

Throughout the rest of this book, we will use the Keil MDK-ARM header file for the programs.

## Program 2-4: Toggling LEDs in C (using special function registers by their names in Keil header file)

```
/* p2_4.c Toggling LED in C using Keil header file register definitions.

 * This program toggles green LED for 0.5 second ON and 0.5 second OFF.

 * The green LED is connected to PTB19.

 * The LEDs are low active (a '0' turns ON the LED).

 */

#include <MKL25Z4.H>

int main (void) {
    void delayMs(int n);

    SIM->SCGC5 |= 0x400;           /* enable clock to Port B */
    PORTB->PCR[19] = 0x100;        /* make PTB19 pin as GPIO (See Table 2-4) */
    PTB->PDDR |= 0x80000;          /* make PTB19 as output pin */

    while (1) {
        PTB->PDOR &= ~0x80000;    /* turn on green LED */
        delayMs(500);
        PTB->PDOR |= 0x80000;     /* turn off green LED */
        delayMs(500);
    }
}

/* Delay n milliseconds

 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().

 */

void delayMs(int n) {
    int i;
    int j;

    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
}
```

The Kinetis Family GPIO ports have three additional registers that make turning a pin (or more) on and off easier. They are PSOR (Port Set Output Register), PCOR (Port Clear Output Register), PTOR (Port Toggle Output Register). Writing to these registers only affects the pin(s) that the corresponding

bit(s) in the value written. This makes it easier to turn on or off a single pin or a few pins without affecting the other pins. For example, writing a value 4 (0100 in binary) to PCOR will turn off bit 2 of that port without modifying any other pins. See Figures 2-15 through 2-17. For more information, see Chapter 41 of KL25 reference Manual.

<b>GPIOx_PSOR:</b> $\text{PTx}\rightarrow\text{PSOR}$	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">D31</td><td style="width: 10%;">D30</td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;">D4</td><td style="width: 10%;">D3</td><td style="width: 10%;">D2</td><td style="width: 10%;">D1</td><td style="width: 10%;">D0</td></tr> <tr> <td colspan="6"></td><td colspan="6"></td></tr> <tr> <td colspan="6" style="background-color: #ffff00;">PTSO</td><td colspan="6"></td></tr> <tr> <td>R/W</td><td>R/W</td><td></td><td></td><td></td><td></td><td></td><td></td><td>R/W</td><td>R/W</td><td>R/W</td><td>R/W</td><td>R/W</td></tr> </table>	D31	D30							D4	D3	D2	D1	D0													PTSO												R/W	R/W							R/W	R/W	R/W	R/W	R/W	<b>0x0004</b>
D31	D30							D4	D3	D2	D1	D0																																								
PTSO																																																				
R/W	R/W							R/W	R/W	R/W	R/W	R/W																																								

Figure 2-15: PSOR (Port Set Output Register)

<b>GPIOx_PCOR:</b> $\text{PTx}\rightarrow\text{PCOR}$	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">D31</td><td style="width: 10%;">D30</td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;">D4</td><td style="width: 10%;">D3</td><td style="width: 10%;">D2</td><td style="width: 10%;">D1</td><td style="width: 10%;">D0</td></tr> <tr> <td colspan="6"></td><td colspan="6"></td></tr> <tr> <td colspan="6" style="background-color: #ffff00;">PTCO</td><td colspan="6"></td></tr> <tr> <td>R/W</td><td>R/W</td><td></td><td></td><td></td><td></td><td></td><td>R/W</td><td>R/W</td><td>R/W</td><td>R/W</td><td>R/W</td></tr> </table>	D31	D30						D4	D3	D2	D1	D0													PTCO												R/W	R/W						R/W	R/W	R/W	R/W	R/W	<b>0x0008</b>
D31	D30						D4	D3	D2	D1	D0																																							
PTCO																																																		
R/W	R/W						R/W	R/W	R/W	R/W	R/W																																							

Figure 2-16: PCOR (Port Clear Output Register)

<b>GPIOx_PTOR:</b> $\text{PTx}\rightarrow\text{PTOR}$	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">D31</td><td style="width: 10%;">D30</td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;"></td><td style="width: 10%;">D4</td><td style="width: 10%;">D3</td><td style="width: 10%;">D2</td><td style="width: 10%;">D1</td><td style="width: 10%;">D0</td></tr> <tr> <td colspan="6"></td><td colspan="6"></td></tr> <tr> <td colspan="6" style="background-color: #ffff00;">PTTO</td><td colspan="6"></td></tr> <tr> <td>R/W</td><td>R/W</td><td></td><td></td><td></td><td></td><td></td><td>R/W</td><td>R/W</td><td>R/W</td><td>R/W</td><td>R/W</td></tr> </table>	D31	D30						D4	D3	D2	D1	D0													PTTO												R/W	R/W						R/W	R/W	R/W	R/W	R/W	<b>0x000C</b>
D31	D30						D4	D3	D2	D1	D0																																							
PTTO																																																		
R/W	R/W						R/W	R/W	R/W	R/W	R/W																																							

Figure 2-17: PTOR (Port Toggle Output Register)

Program 2-5 shows how to use these registers to toggle the green LED on Freescale FRDM board.

#### Program 2-5: Toggling a single LED using PSOR, PCOR, and PTOR registers

```
/*
 * p2_5.c Toggling LED in C using PSOR, PCOR, and PTOR registers.
 *
 * This program toggles green LED for 0.5 second ON and 0.5 second OFF.
 *
 * The green LED is connected to PTB19.
 *
 * The LEDs are low active (a '0' turns ON the LED).
 */

#include <MKL25Z4.H>

int main (void) {
    void delayMs(int n);
    SIM->SCGC5 |= 0x400;           /* enable clock to Port B */
    PORTB->PCR[19] = 0x100;        /* make PTB19 pin as GPIO (See Table 2-4)*/
    PTB->PDDR |= 0x80000;          /* make PTB19 as output pin */
    while (1) {
        PTB->PCOR = 0x80000;      /* turn on green LED */
        delayMs(500);
        PTB->PSOR = 0x80000;      /* turn off green LED */
    }
}
```

```

    delayMs(500);

    PTB->PTOR = 0x80000;      /* Toggle green LED */

    delayMs(500);

    PTB->PTOR = 0x80000;      /* Toggle green LED */

    delayMs(500);

}

}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;

    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++)
}

```

Program 2-6 shows how to toggle all three LEDs on Freescale FRDM board. The three color LEDs are not connected to the same port of the microcontroller.

#### Program 2-6: Toggling all three LEDs on FRDM-KL25Z board

```

/* p2_6.c Toggling all three LEDs on FRDM-KL25Z board.

 * This program toggles all three LEDs on the FRDM-KL25Z board.

 * The red LED is connected to PTB18.

 * The green LED is connected to PTB19.

 * The blue LED is connected to PTD1.

 * The LEDs are low active (a '0' turns ON the LED).

 */

#include <MKL25Z4.H>

int main (void) {
    void delayMs(int n);

    SIM->SCGC5 |= 0x400;          /* enable clock to Port B */

    SIM->SCGC5 |= 0x1000;         /* enable clock to Port D */

    PORTB->PCR[18] = 0x100;       /* make PTB18 pin as GPIO (See Table 2-4) */

```

```

PORTB->PCR[19] = 0x100;      /* make PTB19 pin as GPIO */
PTB->PDDR |= 0xC0000;        /* make PTB18, 19 as output pin */
PORTD->PCR[1] = 0x100;       /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02;          /* make PTD1 as output pin */

while (1) {
    PTB->PDOR &= ~0xC0000; /* turn on red and green LED */
    PTD->PDOR &= ~0x02;     /* turn on blue LED */
    delayMs(500);
    PTB->PDOR |= 0xC0000;   /* turn off red and green LED */
    PTD->PDOR |= 0x02;      /* turn off blue LED */
    delayMs(500);
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
}

```

Program 2-7 shows how to generate all 8 color combinations of the tri-color LEDs. An incrementing counter is used. The bit 0 of the counter is used to control the red LED. The bit 1 of the counter is used to control the green LED. The bit 2 of the counter is used to control the blue LED. More colors may be generated by using the PWM (pulse width modulation) but that is a subject of a later chapter.

#### Program 2-7: Cycle through all color combinations of LEDs

```

/* p2_7.c Cycle through all color combinations of LEDs on FRDM-KL25Z board.
 * This program displays all eight combinations of the
 * three LEDs on the FRDM-KL25Z board.
 * The red LED is connected to PTB18.
 * The green LED is connected to PTB19.
 * The blue LED is connected to PTD1.

```

```

 * The LEDs are low active (a '0' turns ON the LED).
 */

#include <MKL25Z4.H>

int main (void) {
    void delayMs(int n);

    int counter = 0;

    SIM->SCGC5 |= 0x400;           /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTB->PCR[18] = 0x100;        /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x40000;          /* make PTB18 as output pin */
    PORTB->PCR[19] = 0x100;        /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0x80000;          /* make PTB19 as output pin */
    PORTD->PCR[1] = 0x100;         /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;            /* make PTD1 as output pin */

    while (1) {
        if (counter & 1)      /* use bit 0 of counter to control red LED */
            PTB->PCOR = 0x40000;    /* turn on red LED */
        else
            PTB->PSOR = 0x40000;    /* turn off red LED */
        if (counter & 2)      /* use bit 1 of counter to control green LED */
            PTB->PCOR = 0x80000;    /* turn on green LED */
        else
            PTB->PSOR = 0x80000;    /* turn off green LED */
        if (counter & 4)      /* use bit 2 of counter to control blue LED */
            PTD->PCOR = 0x02;       /* turn on blue LED */
        else
            PTD->PSOR = 0x02;       /* turn off blue LED */
        counter++;
        delayMs(500);
    }
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {

```

```

int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}

```

## CPU clock frequency and time delay

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

- 1) The on-chip RC oscillator circuit. This is the least precise clock source for the CPU. But it does not require additional external devices.
- 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.

Freescale FRDM board is connected externally to an 8MHz XTAL oscillator and one can program its KL25Z128VLK4 chip clock generator to implement all three options.

When you start a new project in Keil MDK-ARM v5 with Device Family Support Pack, the Project Wizard automatically add a file system\_MKL25Z4.c to the project. In this file, there are three system clock initialization modes (see Table 2-5 below). By default, clock setup mode 0 is used. This will be the clock setup we use in all the examples throughout this book unless stated otherwise. You may choose a different clock setup mode by editing the #define CLOCK\_SETUP in the system\_MKL25Z4.c file.

<b>CLOCK_SETUP</b>	<b>Mode</b>	<b>Reference clock source</b>	<b>Core clock</b>	<b>Bus clock</b>
0	FLL Engaged Internal	slow internal clock 32.768kHz	41.94MHz	13.98MHz
1	PLL Engaged External	external crystal 8MHz	48MHz	24MHz
2	Bypassed Low Power External	external crystal 8MHz	8MHz	8MHz

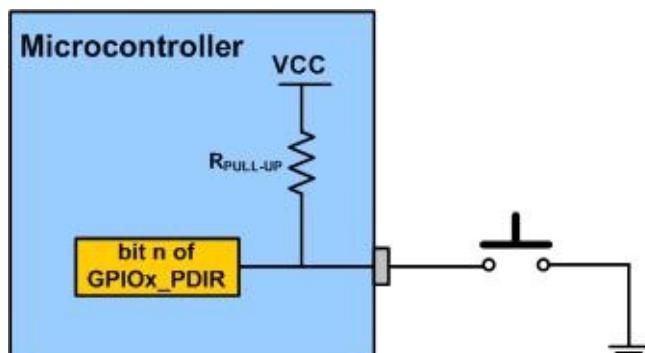
**Table 2-5: Clock setup modes in Keil MDK-ARM v5 Freescale Kinetis KLxx Series Device Family Support Pack**

## Measuring time delay in a C program loop

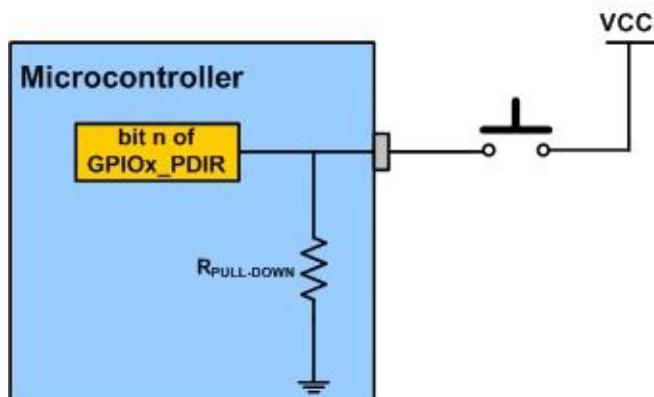
One simple way of creating a time delay is using a **for** loop in C language. The length of time delay loop for a given system is function of two factors: a) the CPU frequency and b) the compiler. It must be noted that a time delay C loop measured using a given compiler (e.g. Keil) may not give the same result if a different compiler such as CodeWarrior 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 2-4 and Program 2-5 using an oscilloscope.

## Reading a switch in Freescale FRDM board

The FRDM KL25Z board does not come with any user programmable push-button switches. We can connect an external SW to the board and experiment with the concept of inputting data via a port. Depending on how we connect an external SW to a pin, we need to enable the internal pull-up or pull-down resistor for a pin. See Figure 2-18 for connecting external switches to microcontroller.



(a) Using Pull-up Resistor



(b) Using Pull-down Resistor

**Figure 2-18: Connecting External Switches to the Microcontroller**

Using the PORTx\_PCRn register, not only we select the alternate I/O function of a given pin, we can also control its Drive Strength and its internal Pull-

up (or Pull-down) resistor. The D1 (PE, pull enable) bit of the PORTx\_PCRn is used to enable the internal Pull resistor option. If PE=1, then we use the D0 bit (PS, pull select) to enable the pull-up (or pull-down) option. See Figure 2-19 and See Example 2-4.

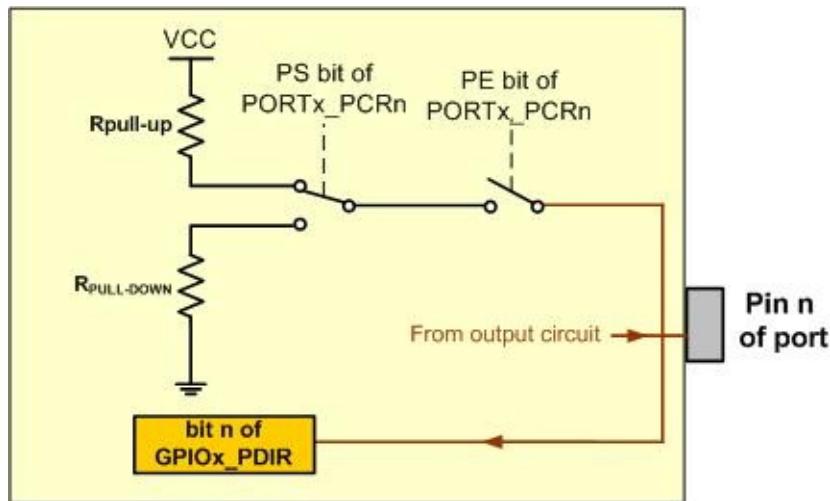


Figure 2-19: PS and PE bits

### Example 2-4

Find the contents of the PORTA\_PCR1 register for PTA1 to use PTA1 pin as input connected to SW. Use the pull-up resistor.

**Solution:**

0	0	0	0000	00000	001	0	0	0	0	1	1	1
Reserved	ISF	Reserved	IRQC	Res.	MUX	0	DSE	0	PFE	0	SRE	PE

To read a switch on PTA1 and display it on the LED on PTB19, the following steps must be taken.

- 1) enable the clock to PORTB,
- 2) configure PTB19 as GPIO in PORTB\_PCR19 register,
- 3) make PTB19 output in PDDR register,
- 4) enable the clock to PORTA,
- 5) configure PTA1 as GPIO and enable the pull-up resistor in PORTA PCR1 register,
- 6) make PTA1 input in PDDR register,
- 7) read switch from PORTA,
- 8) if PTA1 is high, set PTB1
- 9) else clear PTB1

10) Repeat steps 7 to 9.

See Programs 2-8.

#### Program 2-8: Reading a switch and displaying it on the green LED

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

 * This program reads an external SW connected to PTA1
 * of FRDM board and writes the value to the green LED.
 * When switch is pressed, it connects PTA1 to ground.
 * PTA1 pin pull-up is enabled so that it is high when
 * the switch is not pressed.
 * LED is on when low.
 */

#include <MKL25Z4.H>

int main (void) {
    SIM->SCGC5 |= 0x400;          /* enable clock to Port B */
    PORTB->PCR[19] = 0x100;       /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0x80000;         /* make PTB19 as output pin */
    SIM->SCGC5 |= 0x200;          /* enable clock to Port A */
    PORTA->PCR[1] = 0x103;        /* make PTA1 pin as GPIO and enable pull-up */
    PTA->PDDR &= ~0x02;           /* make PTA1 as input pin */

    while (1) {
        if (PTA->PDIR & 2)      /* check to see if switch is pressed */
            PTB->PSOR = 0x80000;   /* if not, turn off green LED */
        else
            PTB->PCOR = 0x80000;   /* turn on green LED */
    }
}
```

### Review Questions

1. KL25Z128VLK4 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 KL25Z128VLK4 has \_\_\_\_ pins.

4. Give the address location assigned to Data Output register of PORTA (GPIOA\_PODR). See Chapter 41 of KL25 Reference Manual.
5. Give the address location assigned to Data Direction register of PORTB (GPIO\_PDDR). See Chapter 41 of KL25 Reference Manual.



## 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 2-20.

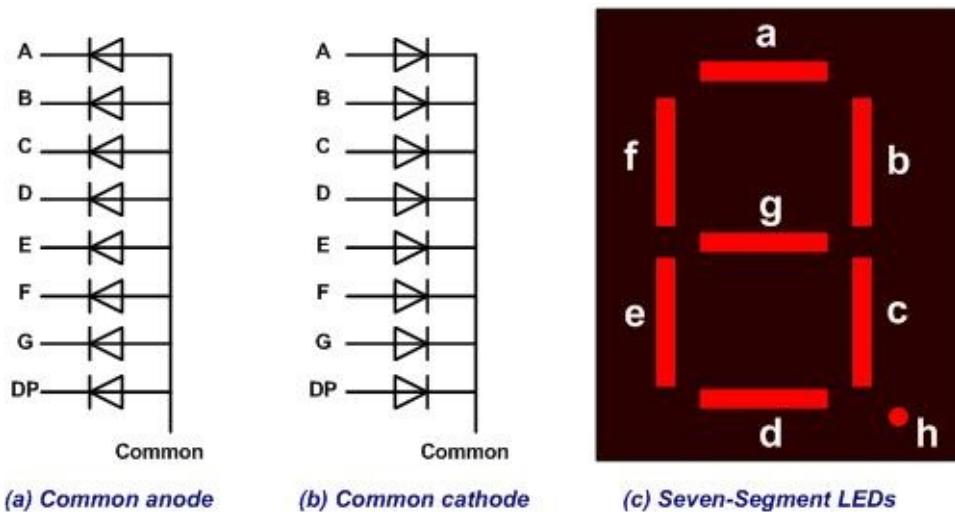


Figure 2-20: Seven-Segment

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 2-6: Assignments of port pins to each segments of a 7-seg LED

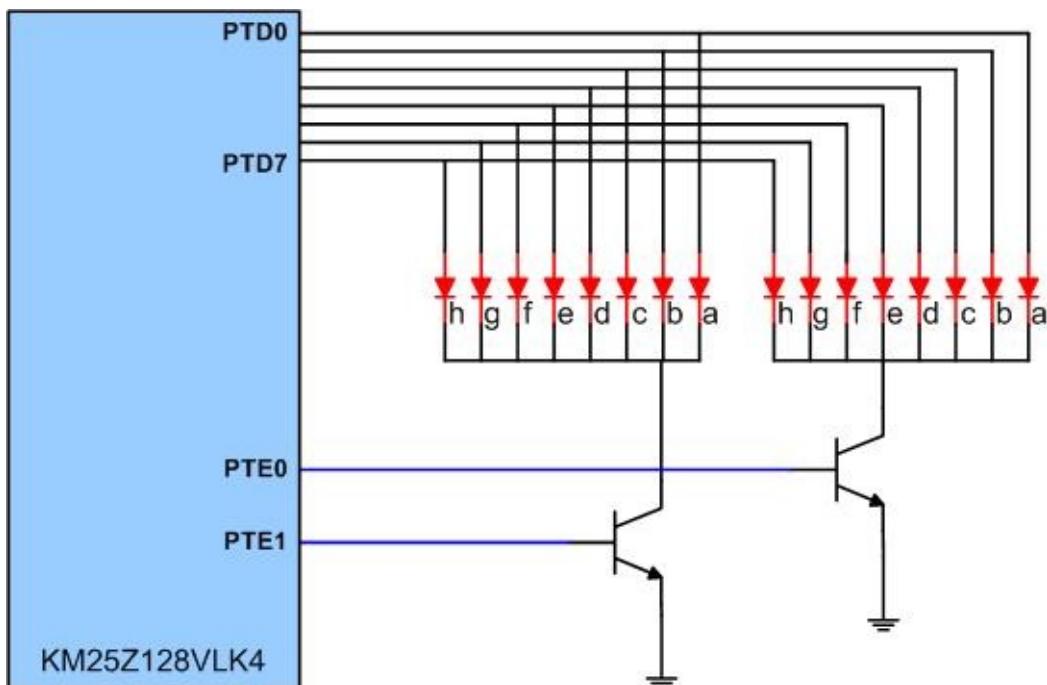
The D7 bit is assigned to decimal point. One can create the following patterns for numbers 0 to 9 for the common cathode configuration:

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

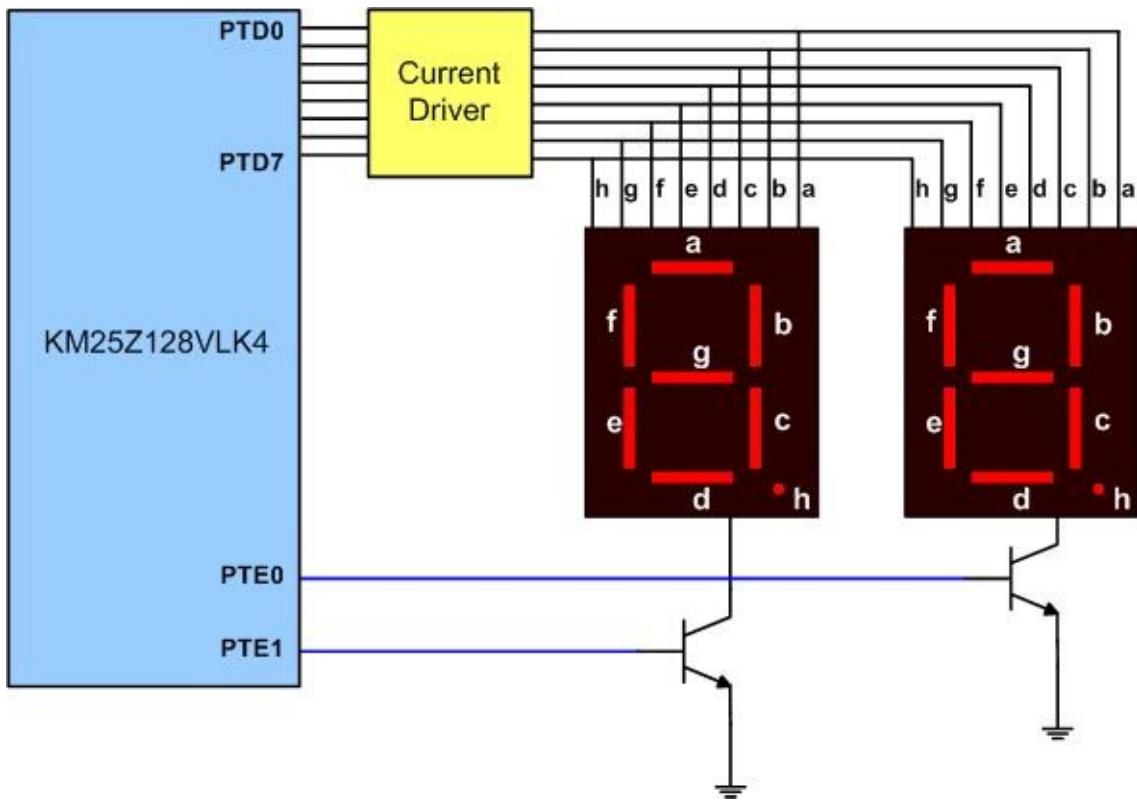
<b>0</b>	0	0	1	1	1	1	1	1	0x3F
<b>1</b>	0	0	0	0	0	1	1	0	0x06
<b>2</b>	0	1	0	1	1	0	1	1	0x5B
<b>3</b>	0	1	0	0	1	1	1	1	0x4F
<b>4</b>	0	1	1	0	0	1	1	0	0x66
<b>5</b>	0	1	1	0	1	1	0	1	0x6D
<b>6</b>	0	1	1	1	1	1	0	1	0x7D
<b>7</b>	0	0	0	0	0	1	1	1	0x07
<b>8</b>	0	1	1	1	1	1	1	1	0x7F
<b>9</b>	0	1	1	0	1	1	1	1	0x6F

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

In Figures 2-21 and 2-22 the connection for 2-digit 7-seg LED and the microcontroller is shown. The Program 2-9 shows the code.



**Figure 2-21: Microcontroller Connection to 7-segment LED**



**Figure 2-22: Microcontroller Connection to 7-segment LED with Buffer Driver**

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 25 on the 7-seg LED, the following steps should be used:

- 1) enable the system clock to PORTD and PORTE,
- 2) Configure Port D as output port to drive the segments,
- 3) Configure Port E as output port to select the digits,
- 4) Write the pattern of numeral 2 from Table 2-7 to Port D,
- 5) Set the PTE1 pin to HIGH to activate the tens digit,
- 6) Delay for some time,
- 7) Write the pattern of numeral 5 from Table 2-7 to Port D,
- 8) Set the PTE0 pin to HIGH to activate the ones digit,
- 9) Delay for some time,
- 10) Repeat from step 4 to 9.

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 6 and 9 should be 8 milliseconds or less.

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

See Program 2-9.

### Program 2-9: Displaying “25” on 2-digit 7-segment LED display

```
/* p2_9.c: Display number 25 on a 2-digit 7-segment LED.

 * Two common cathode 7-segment LEDs are used.
 * PTDO-6 are connected to segment A-G respectively.
 * PTE0 is used to control right digit (low for digit on).
 * PTE1 is used to control left digit (low for digit on).
 */

#include <MKL25Z4.H>

void delayMs(int n);

int main(void)
{
    unsigned char digitPattern[] =
    {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};

    /*from Table 2-7 */

    SIM->SCGC5 |= 0x1000;      /* enable clock to Port D */
    SIM->SCGC5 |= 0x2000;      /* enable clock to Port E */
    PORTD->PCR[0] = 0x100;     /* make PTDO pin as GPIO */
    PORTD->PCR[1] = 0x100;     /* make PTD1 pin as GPIO */
    PORTD->PCR[2] = 0x100;     /* make PTD2 pin as GPIO */
    PORTD->PCR[3] = 0x100;     /* make PTD3 pin as GPIO */
    PORTD->PCR[4] = 0x100;     /* make PTD4 pin as GPIO */
    PORTD->PCR[5] = 0x100;     /* make PTD5 pin as GPIO */
    PORTD->PCR[6] = 0x100;     /* make PTD6 pin as GPIO */
    PTD->PDDR |= 0x7F;        /* make PTD6-0 as output pins */
    PORTE->PCR[0] = 0x100;     /* make PTE0 pin as GPIO */
    PORTE->PCR[1] = 0x100;     /* make PTE1 pin as GPIO */
    PTE->PDDR |= 0x03;        /* make PTE1-0 as output pin */

    for(;;)
    {

```

```

PTD->PDOR = digitPattern[2];      /* drive pattern of 2 on the segments */
PTE->PSOR = 0x01;                  /* turn off right digit */
PTE->PCOR = 0x02;                  /* turn on left digit */
/* delay 8 ms will result in 16 ms per loop or 62.5 Hz */
delayMs(8);

PTD->PDOR = digitPattern[5];      /* drive pattern of 5 on the segments */
PTE->PCOR = 0x01;                  /* turn on right digit */
PTE->PSOR = 0x02;                  /* turn off left digit */
delayMs(8);

}

}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
}

```

Notice in Figure 2-21, 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 2-22)? How about if we use 3-8 decoder for digit selection?

# **Answer to Review Questions**

## **Section 2-1**

1. 128KB
2. 16KB
3. Program code
4. Data
5. 0x0000 0000 to 0x0001 FFFF
6. LK is packaging designation meaning 80-pin LQFP

## **Section 2-2**

1. 5 (A to E)
2. True
3. 12
4. 0x400F F000
5. 0x400F F054

## **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 Freescale FRDM board.

### 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

<b>8</b>	DB1	I/O	The 8-bit data bus
<b>9</b>	DB2	I/O	The 8-bit data bus
<b>10</b>	DB3	I/O	The 8-bit data bus
<b>11</b>	DB4	I/O	The 4/8-bit data bus
<b>12</b>	DB5	I/O	The 4/8-bit data bus
<b>13</b>	DB6	I/O	The 4/8-bit data bus
<b>14</b>	DB7	I/O	The 4/8-bit data bus

Table 3-1: Pin Descriptions for LCD

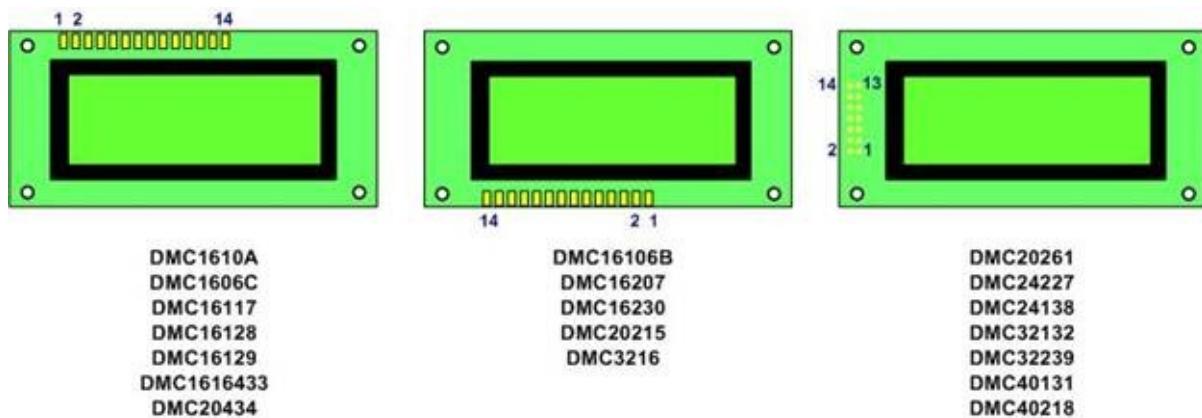


Figure 3-1: Pin Positions for Various LCDs from Optrex

**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 pulse (Low-to-High-to-Low) 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)
28	2 lines and 5x7 character (4-bit data, D4 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 pins RS = 0, R/W = 0, and send a pulse (L-to-H-to-L) 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.

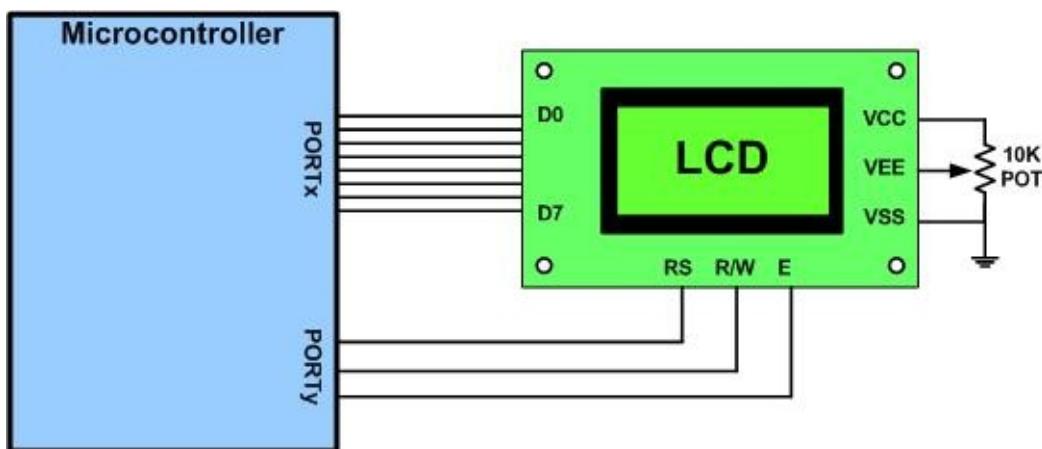


Figure 3-2: LCD Connection to Microcontroller

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

1. The LCD's data pins are connected to PORTD of the microcontroller.
2. The LCD's RS pin is connected to Pin 2 of PORTA of the microcontroller.
3. The LCD's R/W pin is connected to Pin 4 of PORTA of the microcontroller.
4. The LCD's E pin is connected to Pin 5 of PORTA of the microcontroller.
5. Both Ports D and A are configured as output ports.

## Sending data to the LCD

In order to send data to the LCD to be displayed, we must set pins RS = 1, R/W = 0, and also send a pulse (L-to-H-to-L) 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, MKL25Z4 can execute more than 1,000 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 use much more relaxed timing than the original HD44780 datasheet (See Table 3-4) to accommodate the variations of different LCD modules. You may want adjust the delay time for the LCD module you use.

**Program 3-1: This program displays a message on the LCD using 8-bit mode and delay.**

```
/* p3_1.c: Initialize and display "Hello" on the LCD using 8-bit data mode.
```

```
* Data pins use Port D, control pins use Port A.
```

```
* This program does not poll the status of the LCD.
```

```
* It uses delay to wait out the time LCD controller is busy.
```

```
* Timing is more relax than the HD44780 datasheet to accommodate the
```

```
* variations among the LCD modules.
```

```
* You may want to adjust the amount of delay for your LCD controller.
```

```
*/
```

```
#include <MKL25Z4.H>
```

```
#define RS 0x04      /* PTA2 mask */
```

```
#define RW 0x10      /* PTA4 mask */
```

```
#define EN 0x20      /* PTA5 mask */
```

```

void delayMs(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 */

delayMs(500);

LCD_command(0x80);        /* set cursor at first line */

LCD_data('H');            /* write the word */

LCD_data('e');

LCD_data('l');

LCD_data('l');

LCD_data('o');

delayMs(500);

}

}

void LCD_init(void)

{

SIM->SCGC5 |= 0x1000;      /* enable clock to Port D */

PORTD->PCR[0] = 0x100;     /* make PTD0 pin as GPIO */

PORTD->PCR[1] = 0x100;     /* make PTD1 pin as GPIO */

PORTD->PCR[2] = 0x100;     /* make PTD2 pin as GPIO */

PORTD->PCR[3] = 0x100;     /* make PTD3 pin as GPIO */

PORTD->PCR[4] = 0x100;     /* make PTD4 pin as GPIO */

PORTD->PCR[5] = 0x100;     /* make PTD5 pin as GPIO */

PORTD->PCR[6] = 0x100;     /* make PTD6 pin as GPIO */

PORTD->PCR[7] = 0x100;     /* make PTD7 pin as GPIO */

PTD->PDDR = 0xFF;         /* make PTD7-0 as output pins */

SIM->SCGC5 |= 0x0200;     /* enable clock to Port A */

PORTA->PCR[2] = 0x100;     /* make PTA2 pin as GPIO */

PORTA->PCR[4] = 0x100;     /* make PTA4 pin as GPIO */

```

```

PORTA->PCR[5] = 0x100;           /* make PTA5 pin as GPIO */
PTA->PDDR |= 0x34;              /* make PTA5, 4, 2 as output pins */

delayMs(30);                   /* initialization sequence */
LCD_command(0x30);
delayMs(10);
LCD_command(0x30);
delayMs(1);
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)
{
    PTA->PCOR = RS | RW;        /* RS = 0, R/W = 0 */
    PTD->PDOR = command;
    PTA->PSOR = EN;            /* pulse E */
    delayMs(0);
    PTA->PCOR = EN;
    if (command < 4)
        delayMs(4);            /* command 1 and 2 needs up to 1.64ms */
    else
        delayMs(1);            /* all others 40 us */
}

void LCD_data(unsigned char data)
{
    PTA->PSOR = RS;            /* RS = 1, R/W = 0 */
    PTA->PCOR = RW;
    PTD->PDOR = data;
    PTA->PSOR = EN;            /* pulse E */
    delayMs(0);
    PTA->PCOR = EN;
    delayMs(1);
}

```

```

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for(j = 0 ; j < 7000; j++) { }
}

```

## 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. The next program example uses polling of the busy bit in the status register.

**Program 3-2: This program displays a message on the LCD using 8-bit mode and polling of the status register**

```

/* p3_2.c: Initialize and display "hello" on the LCD using 8-bit data mode.

 * Data pins use Port D, control pins use Port A.
 * Polling of the busy bit of the LCD status bit is used for timing.
 */
#include <MKL25Z4.H>

```

```

#define RS 0x04      /* PTA2 mask */
#define RW 0x10      /* PTA4 mask */
#define EN 0x20      /* PTA5 mask */

void delayMs(int n);

void LCD_command(unsigned char command);
void LCD_command_noWait(unsigned char command);
void LCD_data(unsigned char data);
void LCD_init(void);
void LCD_ready(void);

int main(void)
{
    LCD_init();
    for(;;)
    {
        LCD_command(1);          /* clear display */
        delayMs(500);
        LCD_command(0xC0);      /* set cursor at 2nd line */
        LCD_data('h');          /* write the word on LCD */
        LCD_data('e');
        LCD_data('l');
        LCD_data('l');
        LCD_data('o');
        delayMs(500);
    }
}

void LCD_init(void)
{
    SIM->SCGC5 |= 0x1000;      /* enable clock to Port D */
    PORTD->PCR[0] = 0x100;     /* make PTD0 pin as GPIO */
    PORTD->PCR[1] = 0x100;     /* make PTD1 pin as GPIO */
    PORTD->PCR[2] = 0x100;     /* make PTD2 pin as GPIO */
    PORTD->PCR[3] = 0x100;     /* make PTD3 pin as GPIO */
    PORTD->PCR[4] = 0x100;     /* make PTD4 pin as GPIO */
    PORTD->PCR[5] = 0x100;     /* make PTD5 pin as GPIO */
    PORTD->PCR[6] = 0x100;     /* make PTD6 pin as GPIO */
}

```

```

PORTD->PCR[7] = 0x100;           /* make PTD7 pin as GPIO */
PTD->PDDR = 0xFF;                /* make PTD7-0 as output pins */
SIM->SCGC5 |= 0x0200;            /* enable clock to Port A */
PORTA->PCR[2] = 0x100;           /* make PTA2 pin as GPIO */
PORTA->PCR[4] = 0x100;           /* make PTA4 pin as GPIO */
PORTA->PCR[5] = 0x100;           /* make PTA5 pin as GPIO */
PTA->PDDR |= 0x34;              /* make PTA5, 4, 2 as output pins */

delayMs(20);                    /* initialization sequence */

LCD_command_noWait(0x30);        /* LCD does not respond to status poll */
delayMs(5);
LCD_command_noWait(0x30);
delayMs(1);
LCD_command_noWait(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 */

}

/* This function waits until LCD controller is ready to
 * accept a new command/data before returns.
 */
void LCD_ready(void)
{
    char status;

    PTD->PDDR = 0;                /* PortD input */
    PTA->PCOR = RS;               /* RS = 0 for status */
    PTA->PSOR = RW;               /* R/W = 1, LCD output */
    do { /* stay in the loop until it is not busy */
        PTA->PSOR = EN;           /* raise E */
        delayMs(0);

        status = PTD->PDIR; /* read status register */
        PTA->PCOR = EN;
        delayMs(0);              /* clear E */
        } while (status & 0x80);   /* check busy bit */
    PTA->PCOR = RW;               /* R/W = 0, LCD input */
    PTD->PDDR = 0xFF;             /* PortD output */
}

```

```

}

void LCD_command(unsigned char command)
{
    LCD_ready();                      /* wait until LCD is ready */
    PTA->PCOR = RS | RW;           /* RS = 0, R/W = 0 */
    PTD->PDOR = command;
    PTA->PSOR = EN;                /* pulse E */
    delayMs(0);
    PTA->PCOR = EN;
}

void LCD_command_noWait(unsigned char command)
{
    PTA->PCOR = RS | RW;           /* RS = 0, R/W = 0 */
    PTD->PDOR = command;
    PTA->PSOR = EN;                /* pulse E */
    delayMs(0);
    PTA->PCOR = EN;
}

void LCD_data(unsigned char data)
{
    LCD_ready();                      /* wait until LCD is ready */
    PTA->PSOR = RS;                 /* RS = 1, R/W = 0 */
    PTA->PCOR = RW;
    PTD->PDOR = data;
    PTA->PSOR = EN;                /* pulse E */
    delayMs(0);
    PTA->PCOR = EN;
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i, j;
    for(i = 0 ; i < n; i++)

```

```
for(j = 0 ; j < 7000; j++) { }
```

## 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 in Program 3-3. After that, every command or data needs to be broken down to two 4-bit operations, upper nibble first. In Program 3-3, the upper nibble is extracted using `command & 0xF0` and the lower nibble is shifted into place by `command << 4`.

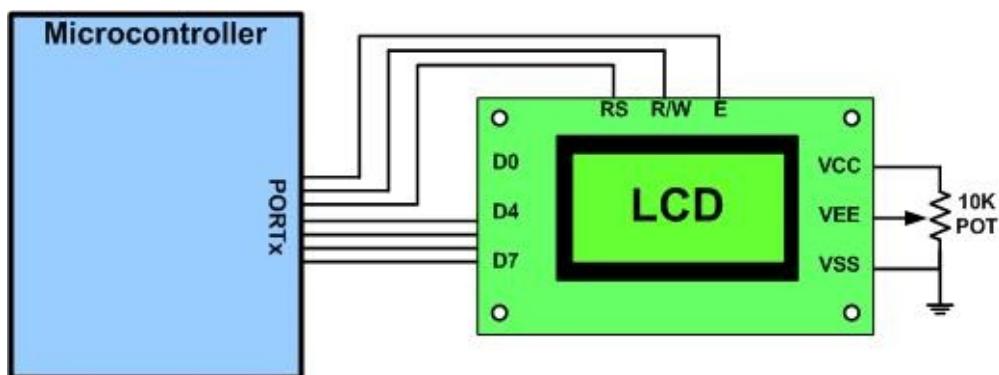


Figure 3-3: LCD Connection for 4-bit Data

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

```
/* p3_3.c: Initialize and display "hello" on the LCD using 4-bit data mode.

* All interface uses Port D. Bit 7-4 are used for data.
* Bit 4, 2, 1 are used for control.
* This program does not poll the status of the LCD.
* It uses delay to wait out the time LCD controller is busy.
* Timing is more relax than the HD44780 datasheet to accommodate the
* variations of the devices.
* You may want to adjust the amount of delay for your LCD controller.
*/
#include <MKL25Z4.H>
```

```

#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 */
        delayMs(500);
        LCD_command(0x85);       /* set cursor at first line */
        LCD_data('h');           /* write the word */
        LCD_data('e');
        LCD_data('l');
        LCD_data('l');
        LCD_data('o');
        delayMs(500);
    }
}

void LCD_init(void)
{
    SIM->SCGC5 |= 0x1000;      /* enable clock to Port D */
    PORTD->PCR[0] = 0x100;     /* make PTD0 pin as GPIO */
    PORTD->PCR[1] = 0x100;     /* make PTD1 pin as GPIO */
    PORTD->PCR[2] = 0x100;     /* make PTD2 pin as GPIO */
    PORTD->PCR[4] = 0x100;     /* make PTD4 pin as GPIO */
    PORTD->PCR[5] = 0x100;     /* make PTD5 pin as GPIO */
    PORTD->PCR[6] = 0x100;     /* make PTD6 pin as GPIO */
}

```

```

PORTD->PCR[7] = 0x100;           /* make PTD7 pin as GPIO */
PTD->PDDR |= 0xF7;             /* make PTD7-4, 2, 1, 0 as output pins */

delayMs(30);                  /* initialization sequence */
LCD_nibble_write(0x30, 0);
delayMs(10);
LCD_nibble_write(0x30, 0);
delayMs(1);
LCD_nibble_write(0x30, 0);
delayMs(1);
LCD_nibble_write(0x20, 0);    /* use 4-bit data mode */
delayMs(1);
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 */
    PTD->PDOR = data | control;      /* RS = 0, R/W = 0 */
    PTD->PDOR = data | control | EN;  /* pulse E */
    delayMs(0);
    PTD->PDOR = data;
    PTD->PDOR = 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(4);          /* commands 1 and 2 need up to 1.64ms */
    else
        delayMs(1);          /* 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 */
    delayMs(1);
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for(j = 0 ; j < 7000; j++) { }
}

```

## 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:



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.

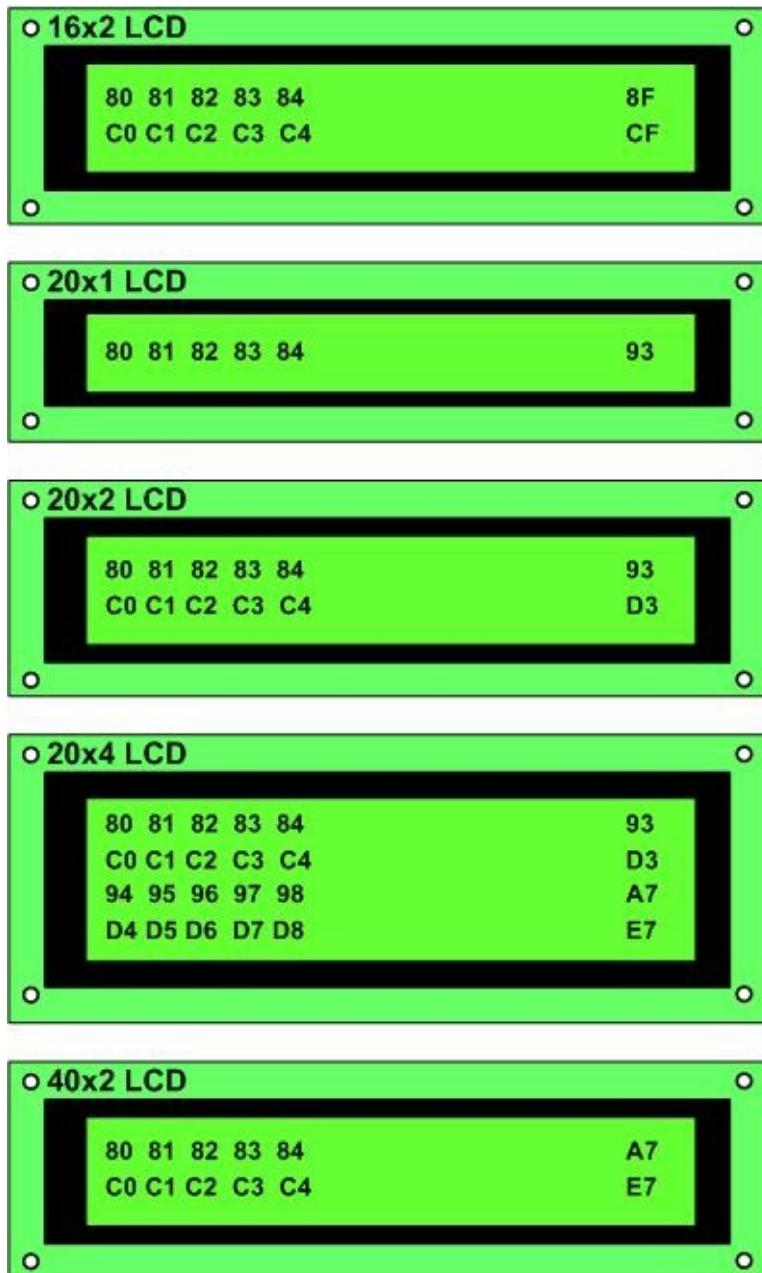


Figure 3-4: Cursor Addresses for Some LCDs

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:

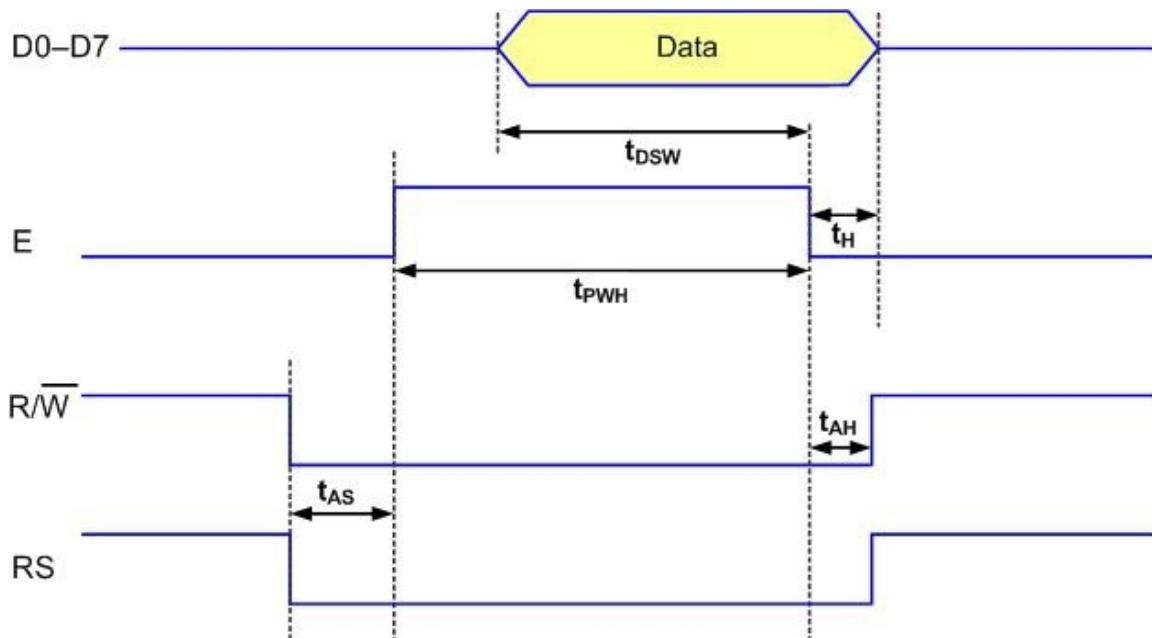
```
LCD_command(0xC5);
```

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.



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

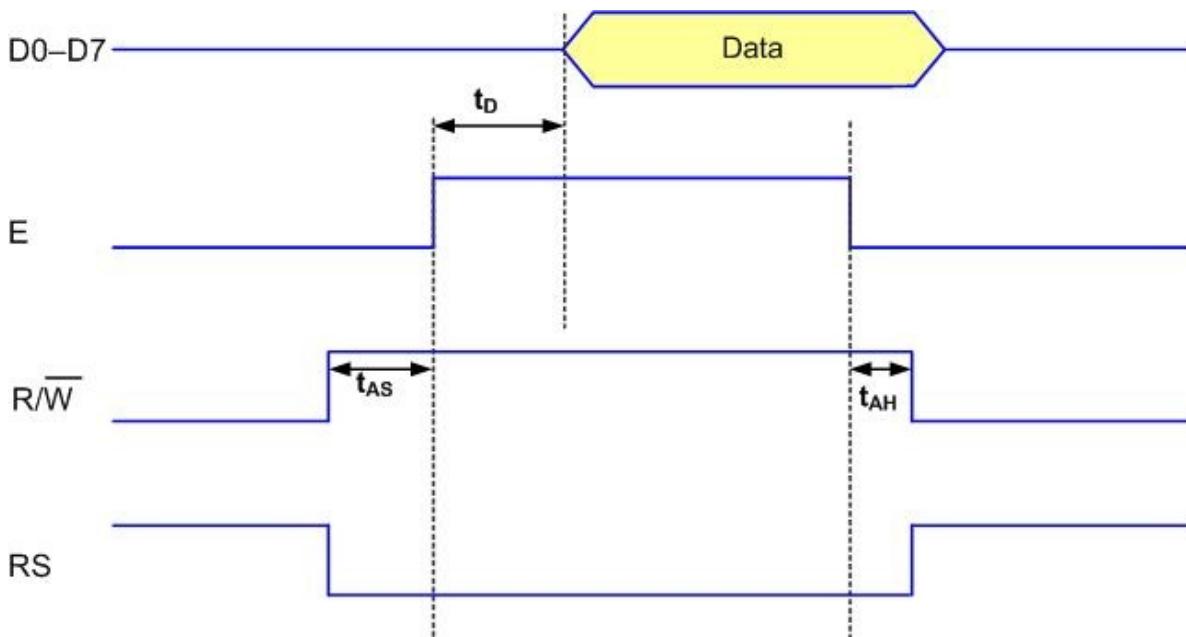
$t_{DSW}$  = Data setup time = 80 ns (minimum)

$t_H$  = Data hold time = 10 ns (minimum)

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

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

Figure 3-5: LCD Write Timing



$t_D$  = Data output delay time

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

$t_{AH}$  = Hold time after E has come down for both RS and R/W = 10 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)
<b>Clear display</b>	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
<b>Return Home</b>	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
<b>Entry Mode Set</b>	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
<b>Display On/Off Control</b>	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

<b>Cursor or Display shift</b>	0	0	0	0	0	1	S/C	R/L	-	-	Moves cursor and shifts display without changing DD RAM contents.	40µs
<b>Function Set</b>	0	0	0	0	1	DL	N	F	-	-	Sets interface data length (DL), number of display lines (L), and character font (F)	40µs
<b>Set CG RAM Address</b>	0	0	0	1	AGC						Sets CG RAM address. CG RAM data is sent and received after this setting.	40µs
<b>Set DD RAM Address</b>	0	0	1	ADD						Sets DD RAM address. DD RAM data is sent and received after this setting.		40µs
<b>Read Busy Flag &amp; Address</b>	0	1	BF	AC						Reads Busy flag (BF) indicating internal operation is being performed and reads address counter contents.		40µs
<b>Write Data CG or DD RAM</b>	1	0	Write Data						Writes data into DD or CG RAM.		40µs	
<b>Read Data CG or DD RAM</b>	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 3-7 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.

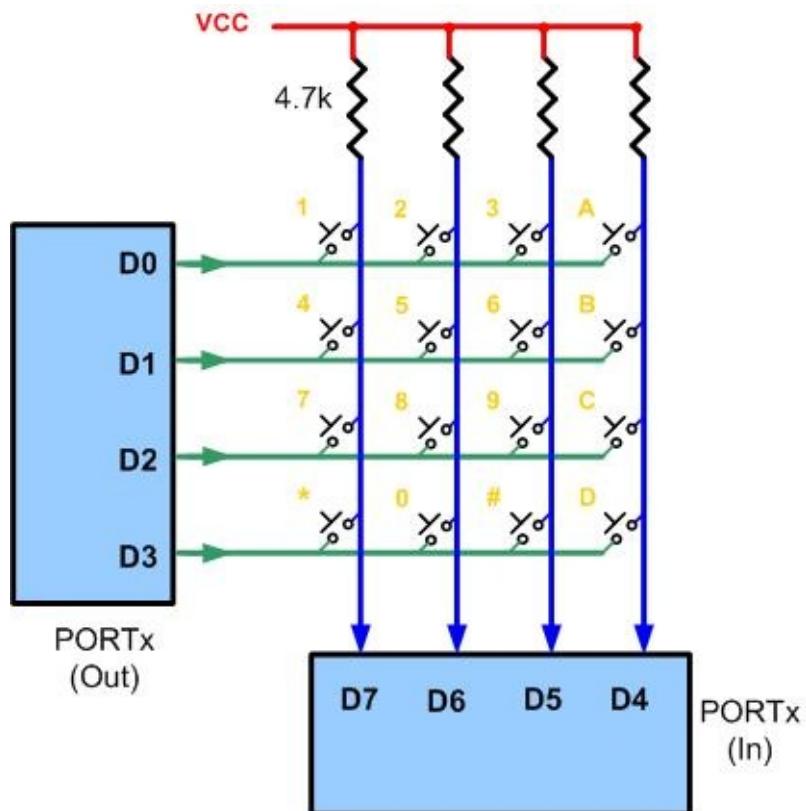


Figure 3-7: Matrix Keyboard Connection to Ports

### 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 3-4: This program turns on the blue LED when a key is pressed.**

```
/* p3_4.c: Matrix keypad detect

* This program checks a 4x4 matrix keypad to see whether
* a key is pressed or not. When a key is pressed, it turns
* on the blue LED.
*
* PortC 7-4 are connected to the columns and PortC 3-0 are connected
* to the rows.
*/
#include <MKL25Z4.H>

void delayUs(int n);

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

int main(void)
{
    keypad_init();
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[1] = 0x100;         /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;            /* make PTD1 as output pin */
    while(1)
    {
        if (keypad_kbhit() != 0)   /* if a key is pressed? */
            PTD->PCOR |= 0x02;     /* turn on blue LED */
        else
            PTD->PSOR |= 0x02;     /* turn off blue LED */
    }
}
```

```

/* this function initializes PortC that is connected to the keypad.
 * All pins are configured as GPIO input pin with pullup enabled.
 */
void keypad_init(void)
{
    SIM->SCGC5 |= 0x0800;          /* enable clock to Port C */
    PORTC->PCR[0] = 0x103;        /* make PTD0 pin as GPIO and enable pullup*/
    PORTC->PCR[1] = 0x103;        /* make PTD1 pin as GPIO and enable pullup*/
    PORTC->PCR[2] = 0x103;        /* make PTD2 pin as GPIO and enable pullup*/
    PORTC->PCR[3] = 0x103;        /* make PTD3 pin as GPIO and enable pullup*/
    PORTC->PCR[4] = 0x103;        /* make PTD4 pin as GPIO and enable pullup*/
    PORTC->PCR[5] = 0x103;        /* make PTD5 pin as GPIO and enable pullup*/
    PORTC->PCR[6] = 0x103;        /* make PTD6 pin as GPIO and enable pullup*/
    PORTC->PCR[7] = 0x103;        /* make PTD7 pin as GPIO and enable pullup*/
    PTD->PDDR = 0x0F;           /* make PTD7-0 as input pins */
}

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

    PTC->PDDR |= 0x0F;          /* enable all rows */
    PTC->PCOR = 0x0F;
    delayUs(2);                 /* wait for signal return */
    col = PTC->PDIR & 0xF0;     /* read all columns */
    PTC->PDDR = 0;              /* disable all rows */
    if (col == 0xF0)
        return 0;                /* no key pressed */
    else
        return 1;                /* a key is pressed */
}

void delayUs(int n)
{
    int i; int j;
    for(i = 0 ; i < n; i++) {
        for(j = 0; j < 8; j++) ;
    }
}

```

## 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 pressed 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 4 was pressed.

---

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

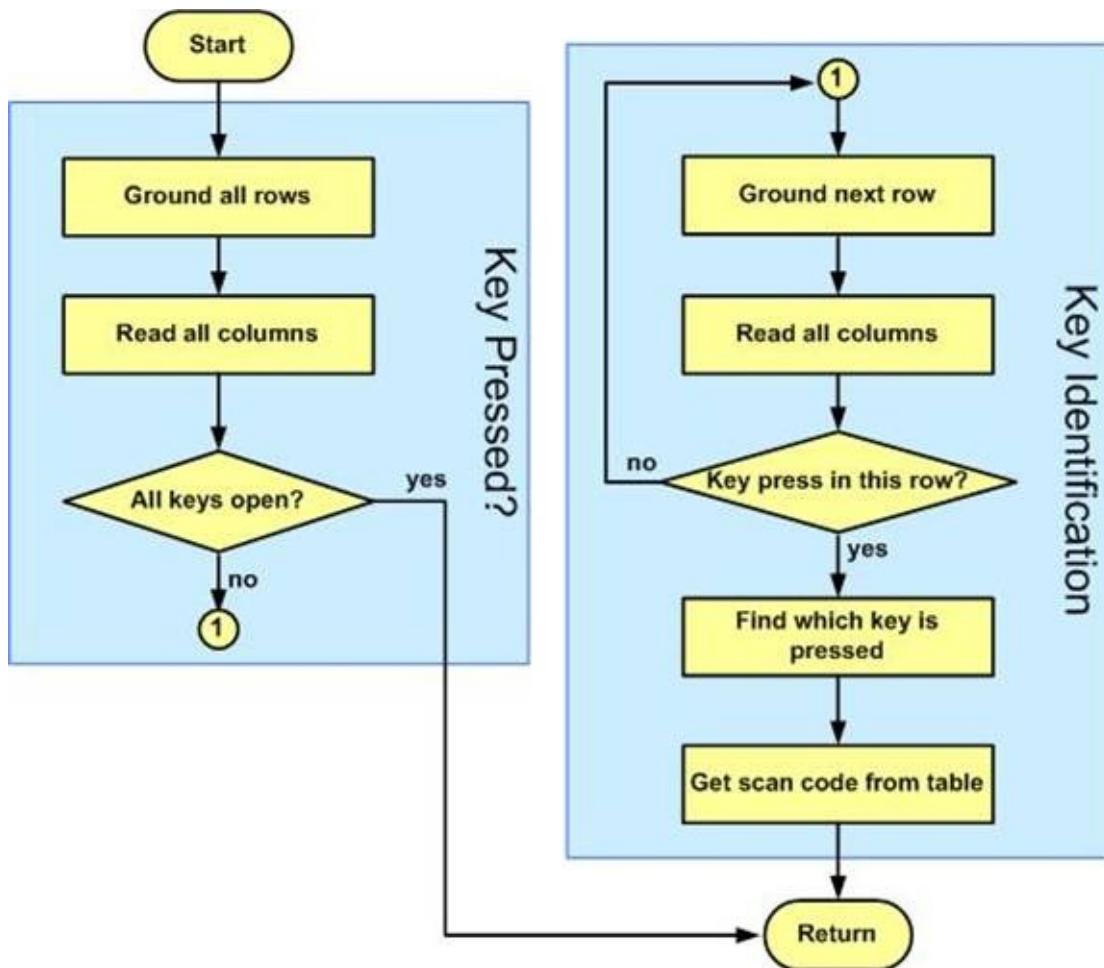


Figure 3-8: The Flowchart for Key Press Detection and Identification

Program 3-5 provides an implementation of the detection and identification algorithm in C language. We will exam it in details here. First for the initialization of the ports, Port C pins 3-0 are used for rows. The Port C pins 7-4 are used for columns. They are all configured as input digital pin to prevent accidental short circuit of two output pins. If output pins are driven high and low and two keys of the same column are pressed at the same time by accident, they will short the output low to output high of the adjacent pins and cause damages to these pins. To prevent this, all pins are configured as input pin and only one pin is configured as output pin at a time. Since only one pin is actively driving the row, shorting two rows will not damage the circuit. The input pins are configured with pull-up enabled so that when the connected keys are not pressed, they stay high and read as 1.

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 return a unique numeric code. The program below reads a 4x4 keypad and use the key code returned to set the tri-color LEDs. LED program is borrowed from P2-7 in

## Chapter 2.

### Program 3-5: This program displays the pressed key on the tri-color LED.

```
/* p3_5.c: Matrix keypad scanning

 * This program scans a 4x4 matrix keypad and returns a unique number
 * for each key pressed. The number is displayed on the tri-color
 * LEDs using the code from P2-7.
 *
 * PortC 7-4 are connected to the columns and PortC 3-0 are connected
 * to the rows.
 */

#include <MKL25Z4.H>

void delayMs(int n);
void delayUs(int n);
void keypad_init(void);
char keypad_getkey(void);
void LED_init(void);
void LED_set(int value);

int main(void)
{
    unsigned char key;
    keypad_init();
    LED_init();

    while(1)
    {
        key = keypad_getkey();
        LED_set(key);           /* set LEDs according to the key code */
    }
}

/* this function initializes PortC that is connected to the keypad.
 * All pins are configured as GPIO input pin with pull-up enabled.
 */
void keypad_init(void)
{
    SIM->SCGC5 |= 0x0800;      /* enable clock to Port C */
```

```

PORTC->PCR[0] = 0x103; /* make PTD0 pin as GPIO and enable pullup*/
PORTC->PCR[1] = 0x103; /* make PTD1 pin as GPIO and enable pullup*/
PORTC->PCR[2] = 0x103; /* make PTD2 pin as GPIO and enable pullup*/
PORTC->PCR[3] = 0x103; /* make PTD3 pin as GPIO and enable pullup*/
PORTC->PCR[4] = 0x103; /* make PTD4 pin as GPIO and enable pullup*/
PORTC->PCR[5] = 0x103; /* make PTD5 pin as GPIO and enable pullup*/
PORTC->PCR[6] = 0x103; /* make PTD6 pin as GPIO and enable pullup*/
PORTC->PCR[7] = 0x103; /* make PTD7 pin as GPIO and enable pullup*/
PTD->PDDR = 0x0F; /* make PTD7-0 as input pins */

}

/*
 * This is a non-blocking function to read the keypad.
 * If a key is pressed, it returns a key code. Otherwise, a zero
 * is returned.
 * The upper nibble of Port C is used as input. Pull-ups are enabled
 * when the keys are not pressed, these pins are pull up high.
 * The lower nibble of Port C is used as output that drives the keypad rows.
 * First all rows are driven low and the input pins are read. If no
 * key is pressed, it will read as all ones. Otherwise, some key is pressed.
 * If any key is pressed, the program drives one row low at a time and
 * leave the rest of the rows inactive (float) then read the input pins.
 * Knowing which row is active and which column is active, the program
 * can decide which key is pressed.
 */
char keypad_getkey(void)
{
    int row, col;
    const char row_select[] = {0x01, 0x02, 0x04, 0x08}; /* one row is active */

    /* check to see any key pressed */
    PTC->PDDR |= 0x0F; /* enable all rows */
    PTC->PCOR = 0x0F;
    delayUs(2); /* wait for signal return */
    col = PTC->PDIR & 0xF0; /* read all columns */
    PTC->PDDR = 0; /* disable all rows */
    if (col == 0xF0)
        return 0; /* no key pressed */

    /* If a key is pressed, it gets here to find out which key.
     * It activates one row at a time and read the input to see
     * which column is active. */

```

```

for (row = 0; row < 4; row++)
{
    PTC->PDDR = 0;                      /* disable all rows */
    PTC->PDDR |= row_select[row];       /* enable one row */
    PTC->PCOR = row_select[row];       /* drive the active row low */
    delayUs(2);                         /* wait for signal to settle */
    col = PTC->PDIR & 0xF0;             /* read all columns */

    if (col != 0xF0) break;              /* if one of the input is low, some key
is pressed. */

}

PTC->PDDR = 0;                      /* disable all rows */
if (row == 4)
return 0;                            /* if we get here, no key is pressed */

/* gets here when one of the rows has key pressed, check which column it is
*/
if (col == 0xE0) return row * 4 + 1;   /* key in column 0 */
if (col == 0xD0) return row * 4 + 2;   /* key in column 1 */
if (col == 0xB0) return row * 4 + 3;   /* key in column 2 */
if (col == 0x70) return row * 4 + 4;   /* key in column 3 */

return 0;    /* just to be safe */
}

/* initialize all three LEDs on the FRDM board */
void LED_init(void)
{
    SIM->SCGC5 |= 0x400;                /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000;               /* enable clock to Port D */
    PORTB->PCR[18] = 0x100;             /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x40000;              /* make PTB18 as output pin */
    PTB->PSOR |= 0x40000;              /* turn off red LED */
    PORTB->PCR[19] = 0x100;             /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0x80000;              /* make PTB19 as output pin */
    PTB->PSOR |= 0x80000;              /* turn off green LED */
    PORTD->PCR[1] = 0x100;              /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;                 /* make PTD1 as output pin */
    PTD->PSOR |= 0x02;                 /* turn off blue LED */
}

/* turn on or off the LEDs according to bit 2-0 of the value */
void LED_set(int value)

```

```

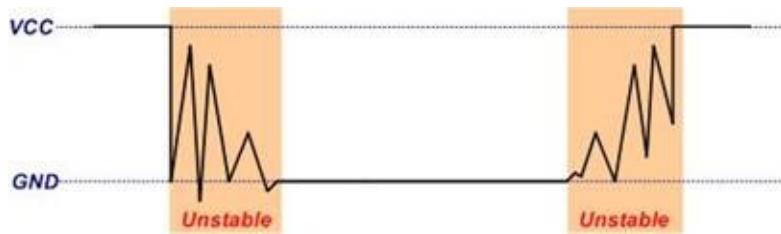
{
    if (value & 1)      /* use bit 0 of value to control red LED */
        PTB->PCOR = 0x40000;      /* turn on red LED */
    else
        PTB->PSOR = 0x40000;      /* turn off red LED */
    if (value & 2)      /* use bit 1 of value to control green LED */
        PTB->PCOR = 0x80000;      /* turn on green LED */
    else
        PTB->PSOR = 0x80000;      /* turn off green LED */
    if (value & 4)      /* use bit 2 of value to control blue LED */
        PTD->PCOR = 0x02;        /* turn on blue LED */
    else
        PTD->PSOR = 0x02;        /* turn off blue LED */
}

/* delay n microseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayUs(int n)
{
    int i; int j;
    for(i = 0 ; i < n; i++) {
        for(j = 0; j < 5; j++) ;
    }
}

```

## Contact Bounce and 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. Contact bounce also occurs when the switch is released. Because the switch contacts open and close several times before they settle, the program may detect a key press when the key is released.



**Figure 3-9: Switch contact bounces**

For many applications, it is important that each key press is only recognized as one action. When you press a numeral key of a calculator, you expect to get only one digit. A contact bounce results in multiple digits entered with a single key press. A simple software solution is that when a transition of the contact state change is detected such as a key pressed or a key released, the software does a delay for about 10 – 20 ms to wait out the contact bounce. After the delay, the contacts should be settled and stable.

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 3-4 and 3-5.

## Review Questions

1. True or false. To see if any key is pressed, all rows are driven low.
2. If  $D_3-D_0 = 0111$  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  $D_3-D_0 = 1110$  and the columns are  $D_7-D_4 = 1110$ , which key is pressed?
5. True or false. To identify the key pressed, one row at a time is driven low.

## **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. A
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.

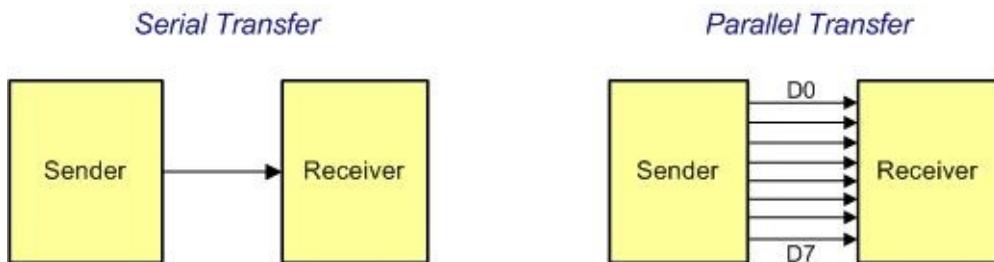


Figure 4-1: Serial vs. Parallel Data Transfer

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.

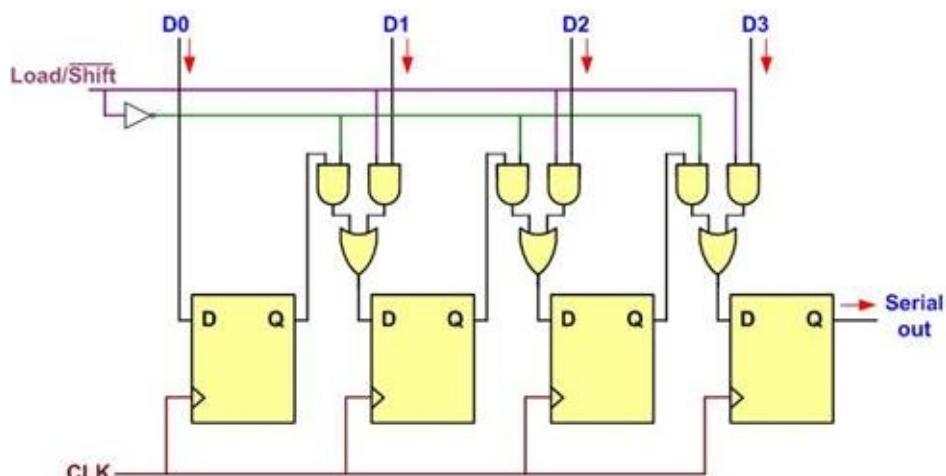


Figure 4-2: Parallel In Serial Out

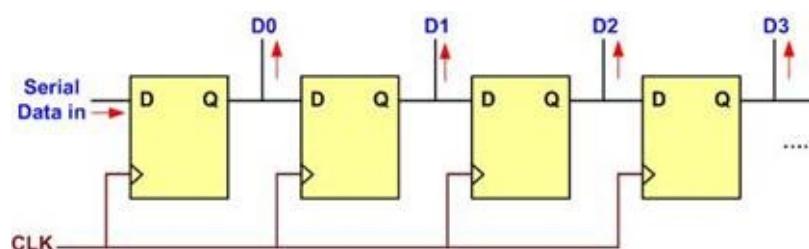


Figure 4-3: Serial In Parallel Out

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 incorporated into 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.

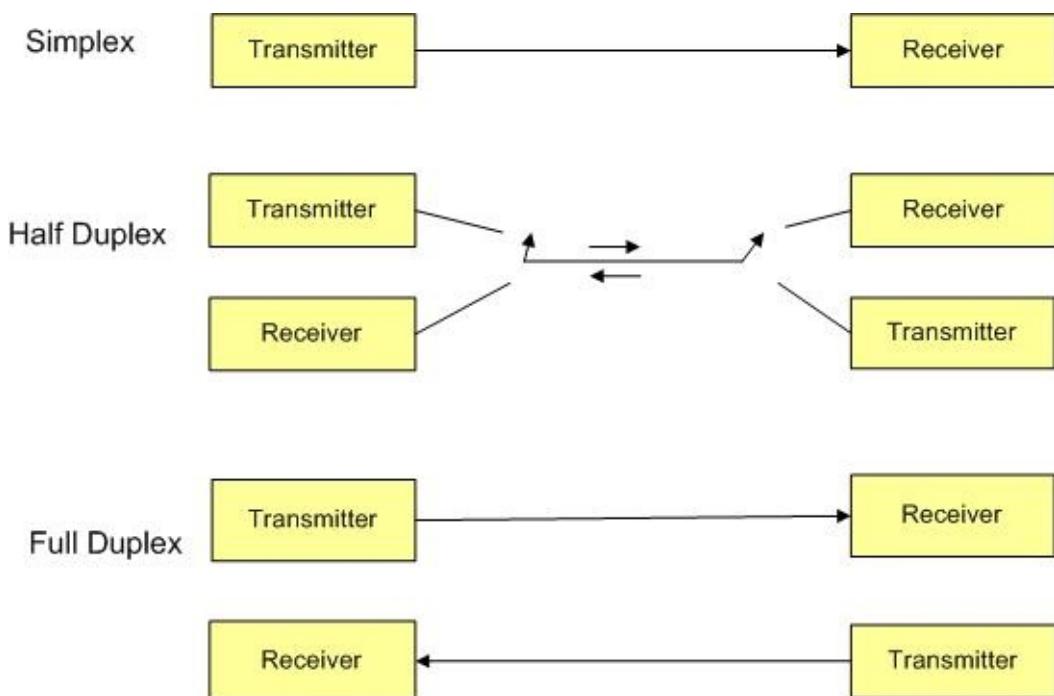


Figure 4-4: Simplex, Half-, and Full-Duplex Transfers

## 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.

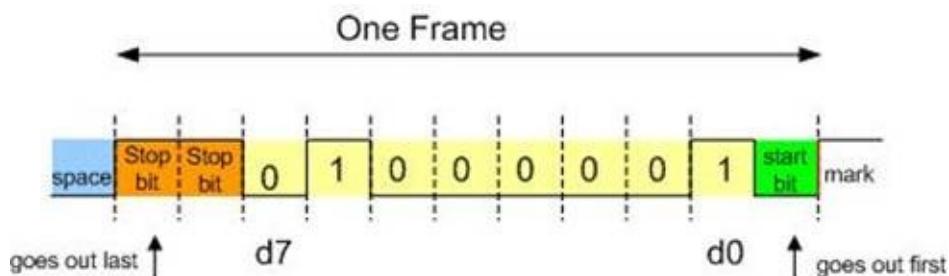


Figure 4-5: Framing ASCII “A” (0x41)

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 Baud 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)  $1,000,000 / 9600 = 104$  seconds  
(b)  $1,000,000 / 57,600 = 17$  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 4-6 and 4-7. The MAX232 has two sets of line drivers for transferring and receiving data, as shown in Figure 4-6. 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.

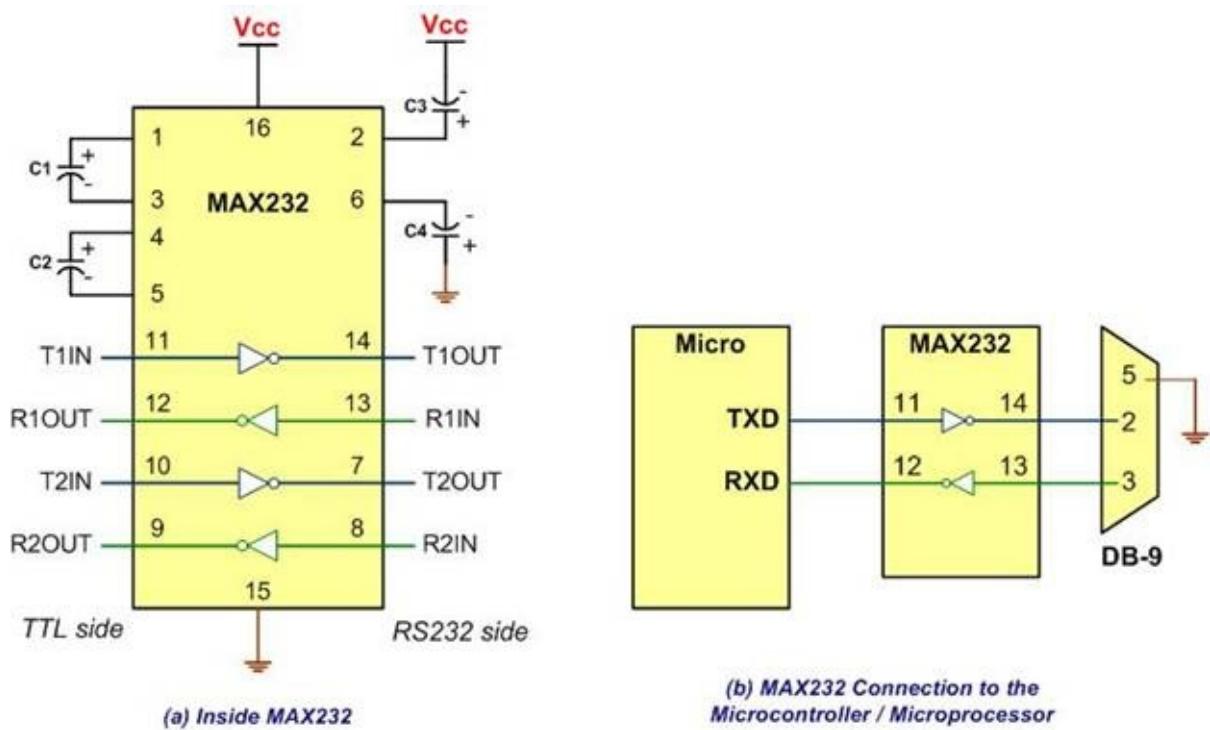


Figure 4-6: MAX232

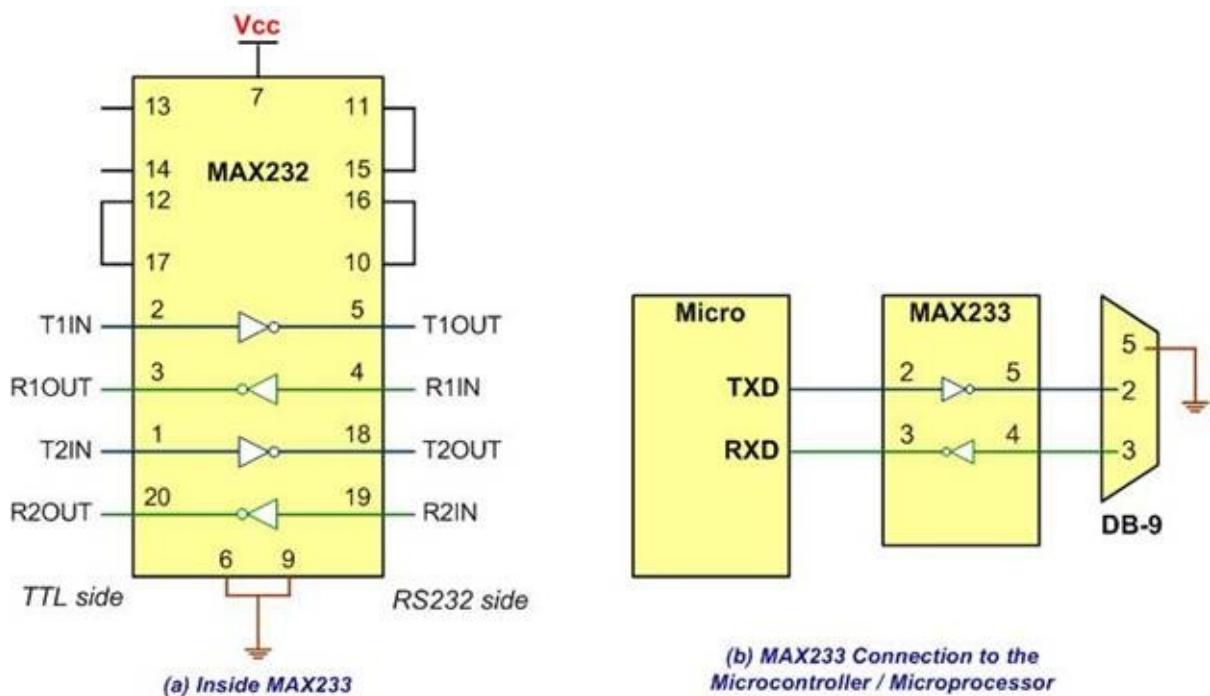


Figure 4-7: MAX233

MAX232 requires four capacitors of  $1 \mu\text{F}$ . 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

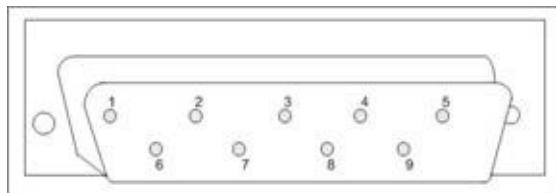


Figure 4-8: 9-Pin Male Connector

## 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.

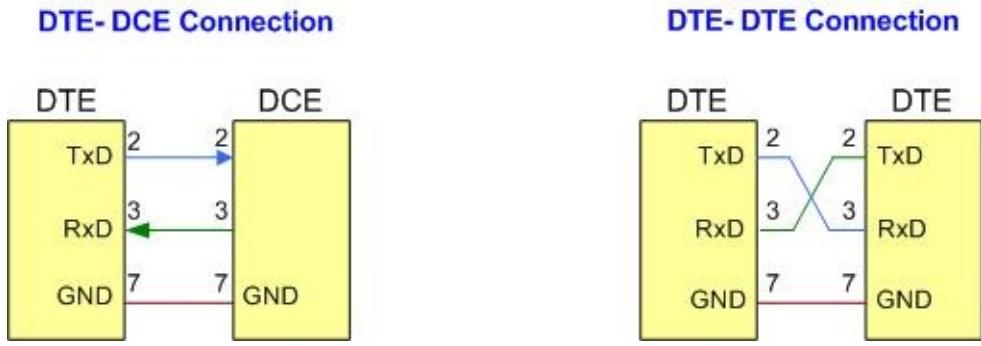


Figure 4-9: DTE-DCE and DTE-DTE Connections

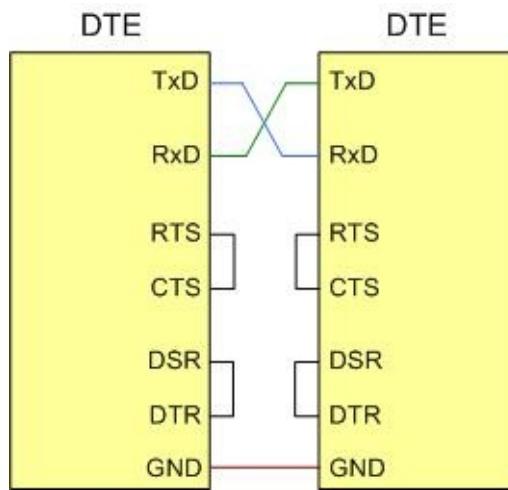
## Examining the RS232 handshaking signals

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 Freescale ARM KL25Z and show how to program them to transmit and receive data serially. Many of the Freescale ARM chips come with up to 3 on-chip UART ports. They are designated as UART0-UART2. In the Freescale FRDM, the UART0 port of the KL25Z is connected to the OpenSDA (Open Serial Debug Adaptor), which is connected to a USB connector. The OpenSDA USB is located below Reset switch and is labeled as SDA. It is on the left side of the board. See Figure 4-11. This OpenSDA USB connection contains three distinct functions:

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

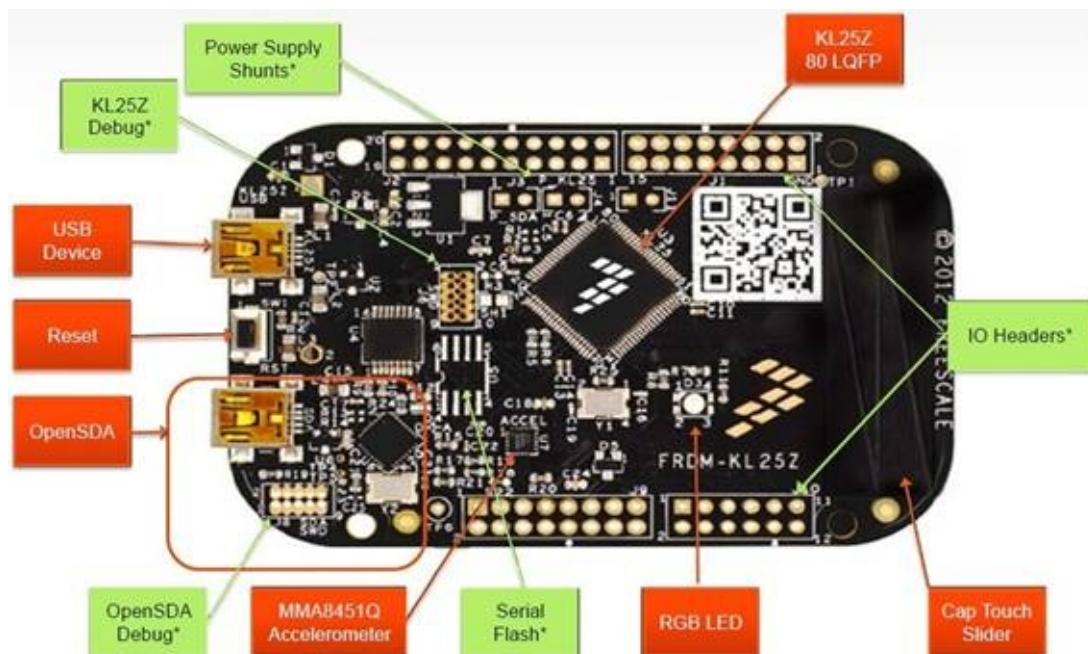


Figure 4-11: Freescale FRDM board

When the USB cable connects the PC to the FRDM board, the device driver at the host PC establishes a virtual connection between the PC and the UART0 of the KL25Z device. On the FRDM, 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 KL25Z on FRDM board, we see the UART0 uses PTA1 and PTA2 pins as alternate functions for UART0\_Tx and UART0\_Rx, respectively. See Figure 4-12.

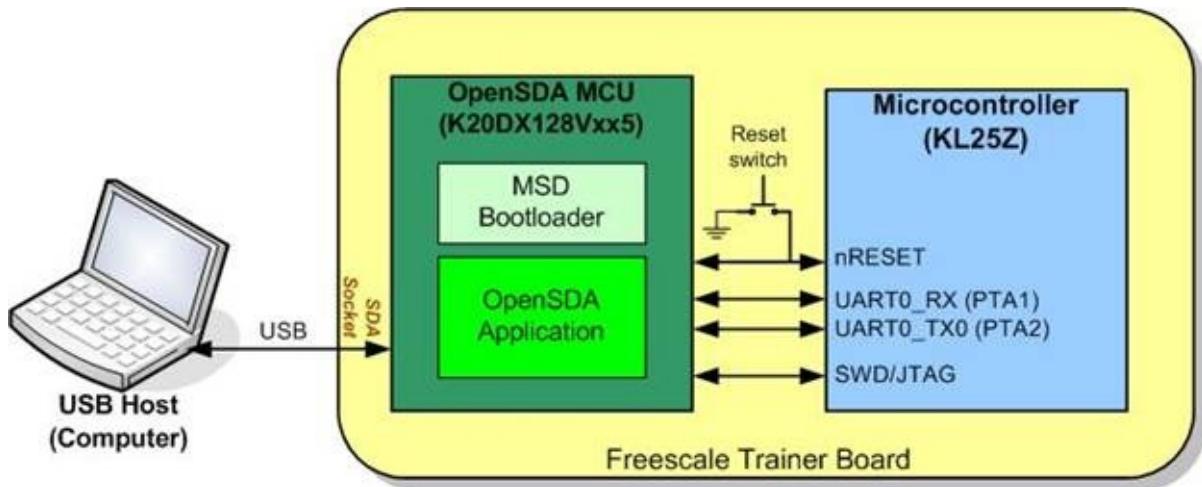


Figure 4-12: OpenSDA USB Port

Notice that there is a second USB connector on the Freescale FRDM right above the Reset switch. It is labeled as *KL25Z USB*. 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 Freescale KL25Z can have up to 3 UART ports. They are designated as UART0 to UART2. The following shows their Base addresses in the memory map:

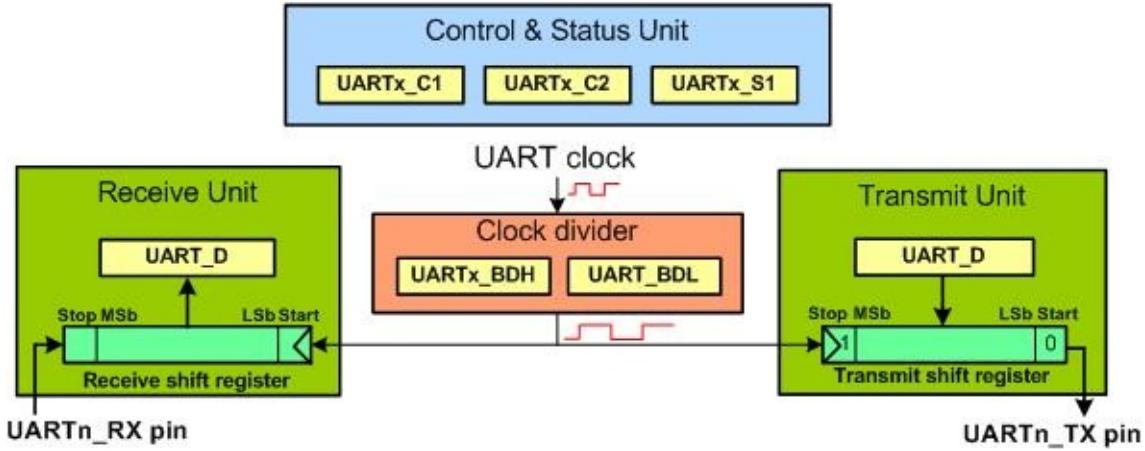
- UART0 base: 0x4006 A000
- UART1 base: 0x4006 B000
- UART2 base: 0x4006 C000

The exact address locations for some of the UART0 registers are shown below:

Register Name	Register Function	Register Address
UART0_BDH	Baud Rate High	4006 A000
UART0_BDL	Baud Rate Low	4006 A001
UART0_C1	Control 1	4006 A002
UART0_C2	Control 2	4006 A003
UART0_S1	Status 1	4006 A004

Table 4-2: Partial list of UART0 Registers and their addresses

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



**Figure 4-13: a Simplified Block Diagram of UARTn**

In all microcontrollers, there are 3 groups of registers in UART peripherals:

1. **Configuration registers:** Before using the UART peripheral the configuration registers must be initialized. This sets some parameters of the communication including: Baud rate, word length, stop bit, serial interrupts (if needed). In Freescale K25Z microcontroller, some of the configuration registers are: `UARTx_BDH`, `UARTx_BDL`, `UARTx_C1`, and `UARTx_C2`.
2. **Transmit and receive register:** To send data, we simply write to the transmit register. The UART peripheral sends out the contents of the transfer register through the serial transmit pin (`UARTx_TX`). The received data is stored in the receive register. In Freescale ARM, the transfer and receive registers are named as `UART_D` (UART Data register).
3. **Status register:** the status register contains some flags which show the state of sending and receiving data including: the existence of new received data, the existence of error in received data; the sending unit is ready for new data, etc. The status register is named as `UARTx_S1` (UART status register) in the Freescale ARMs.

There are many special function registers associated with each of the UARTs. In this section, first, we will be using the `UART0` as an example since a virtual connection is available on the Freescale FRDM board.

First, we will examine the baud-rate generator registers.

## Transmit clock and receive clock

The transmitter operates on the clock that runs at the Baud rate. For each clock pulse, one bit is transmitted. Because UART is asynchronous, the receiver needs to detect the falling edge of the start bit so it has to run on a faster clock. This is called oversampling. The `UART0` has more flexible configurations of the clocks and oversampling and is different from `UART1` and `UART2`.

## ***UART0 Baud clock and oversampling***

The source of the Baud rate generator clock for UART0 is programmable in SIM\_SOPT2 register to select from the output of FLL, PLL, the external oscillator, or the internal oscillator. Coming out of power up or reset the clock is disabled. One must program the SIM\_SOPT2 register to use UART0. For the default system clock configuration of Keil projects, the FLL output is 41.94 MHz and it should be used. The internal 32.768kHz clock is too slow for UART to generate a precise clock for higher baud rate. See Figures 4-14 and 4-15. Also see Appendix C.

SIM_SOPT2:	D31 ... D28	D27	D26	D25	D24	D23 ... D19	D18	D17	D16	D15	...	D8	D7 ... D5	D4	D3	...	D0	0x1004
	UART0SRC	TPM SRC	0	USBSRC	C	0	PLLSEL	0	0	CLKOUTSEL	RTCCCLKOUTSEL	0						
bit	Name	Description																
27-26	UART0SRC	<i>UART0SRC (UART0 clock source select) selects the clock source for UART0.</i>																
		UART0SRC	Selected Clock															
		00	Clock disabled															
		01	MCGFLLCLK clock or MCGFLLCLK/2															
		10	OSCERCLK clock															
		11	MCGIRCLK clock															
16	PLLSEL	<i>It selects between MCGPLLCLK/2 and MCGPLLCLK for various peripherals:</i> 0: MCGPLLCLK 1: MCGPLLCLK/2																

Figure 4-14: SIM\_SOPT2 Register

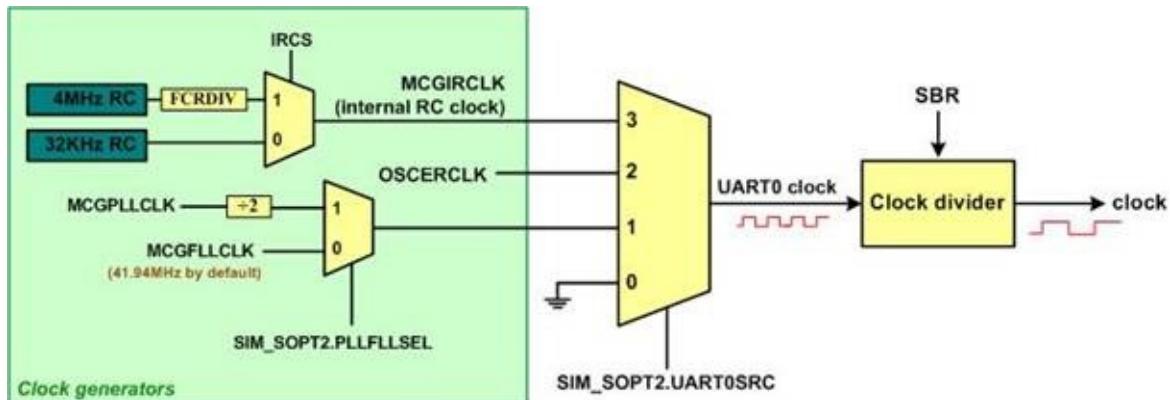


Figure 4-15: Clock Circuit of UART0

### Note

The Clock Distribution chapter in KL25 reference manual discusses the clock sources in detail.

## ***UARTx\_BDH and UARTx\_BDL Registers and the SBR Value***

Two registers are used to set the baud rate: They are UART Baud Rate High (UARTx\_BDH) and UART Baud Rate Low (UARTx\_BDL) in which x=0, 1, or 2 referring to UART0, UART1, or UART2. The details of these two registers are shown below:

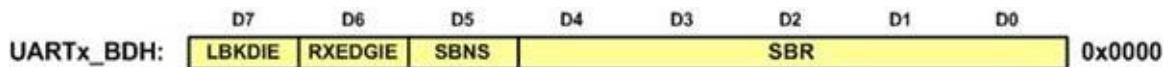


Figure 4-16: UARTx\_BDH

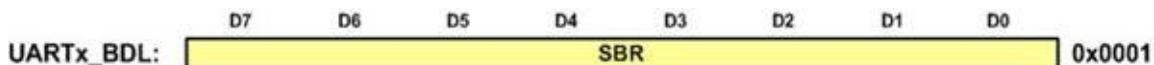


Figure 4-17: UARTx\_BDL

For the UART0 used in Freescale FRDM board, their physical addresses are located at 0x4006 A000 and 0x4006 A001, respectively. Notice these registers are only 8 bits and each register takes a single address location in the memory map. Of the 8-bit of the UARTx\_BDH, only lower 5 bits are used and for the UARTx\_BDL, all the 8 bits are used. That gives us total of 13 bits.

The transmit clock of UART0 is calculated by the clock source by the content of UART0\_BDH:UART0\_BDL and then by the Over Sampling Ratio. The transmit clock runs at the baud rate is

$$\text{Baud Rate} = \frac{\text{Clock Source}}{(\text{OSR} + 1) \times \text{SBR}}$$

SBR is the concatenation of UART0\_BDH and UART0\_BDL. The OSR is the D4-D0 of UART0\_C4 register that determines the oversampling rate. For example, if we set UART0\_BDH = 0, UART0\_BDL = 23 and use the default value of OSR = 15, then

$$\text{Baud Rate} = \frac{41,940,000}{(15 + 1) \times 23} = 113967$$

which is only 1% off the Baud rate of 115200.

The OSR can be set to values 3 to 31. See Figure 4-18. So, the Over Sampling Ratio has the range of 4–32. See Figure 4-18 and Table 4-3.

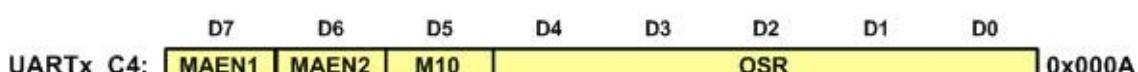


Figure 4-18: UARTx\_C4 Register

Bit	Field	Descriptions
7	MAEN1	Match Address Enable 1: In your programs set the bit to 0. For more information see the KL25 user manual.
6	MAEN2	Match Address Enable 2: In your programs set the bit to 0.
5	M10	10-bit Mode select: 0: Receiver and transmitter use 8-bit or 9-bit data characters 1: Receiver and transmitter use 10-bit data characters
		Over Sampling Ration (the value can be between 00011 to

**Table 4-3: UARTx\_C4 Register**

For OSR less than 7, the receiver must be sampled at both rising edge and falling edge of the clock. This is accomplished by setting the BOTHEDGE (bit 1) of the UART0\_C5 register. See Example 4-4.

### Example 4-4

- (a) Assume the clock source of 41.94 MHz is fed to UART0 Baud rate generator and OSR is set to 7 in UART0\_C4. Find the values for the divisor registers of UART0\_BDH and UART0\_BDL for 9600 Baud.
- (b) Assume the clock source of 41.94 MHz is fed to UART0 Baud rate generator and OSR is set to 15 in UART0\_C4. Find the values for the divisor registers of UART0\_BDH and UART0\_BDL for 115200 Baud.
- (c) Assume the bus clock is 13.98 MHz, find the values for the divisor registers of UART2\_BDH and UART2\_BDL for 38400 Baud.
- (d) Assume the bus clock is 13.98 MHz, find the values for the divisor registers of UART2\_BDH and UART2\_BDL for 115200 Baud.

### **Solution:**

- (a)  $41,940,000 / (7 + 1) / 9600 = 546 = 0x0222$ , UART0\_BDH = 0x02 and UART0\_BDL = 0x22
- (b)  $41,940,000 / (15 + 1) / 115200 = 23 = 0x0017$ , UART0\_BDH = 0x00 and UART0\_BDL = 0x17
- (c)  $13,980,000 / 16 / 38400 = 23 = 0x0017$ , UART2\_BDH = 0x00 and UART2\_BDL = 0x17
- (d)  $13,980,000 / 16 / 115200 = 8 = 0x0008$ , UART2\_BDH = 0x00 and UART2\_BDL = 0x08

**Note:** It must be noted that we have rounded up or rounded down the value loaded into the BDL register. Both registers of UARTx\_BDH and UARTx\_BDL must be loaded, even if the UARTx\_BDH value is zero.

115,200. Table 4-4 shows the SBR values for the different baud rate for UART0 using default OSR = 15 and FLL clock output of 41.94 MHz.

Baud rate	SBR (in decimal)	SBR (in hex)
4,800	546	0x0222
9,600	273	0x0111
19,200	137	0x0089
38,400	68	0x0044
115,200	23	0x0017

Table 4-4: UART0 SBR Values for Some Baud Rates using default OSR=15 and FLL clock output of 41.94 MHz.

## UART Control 1 (UARTx\_C1) register

The next important register in UART is the control register. We have several UART Control Registers. The most important among them are UART Control Register 1 (UARTx\_C1) and UART Control Register 2 (UARTx\_C2). They are 8-bit registers. The Control Register 1 is used to select the data framing size among other things. See Figure 4-19. Notice that the M bit (D4) of the C1 register determines the framing of data by specifying the number of bits per character. In this textbook, we use the no parity option with a data size of 8 bits. The Control Register 2 is used to enable the serial port to send and receive data, among other things. The Control Register 4 and 5 are only used in UART0 for over sampling ratio and both edges as described previously on Baud rate generation.

Control Register 1 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.

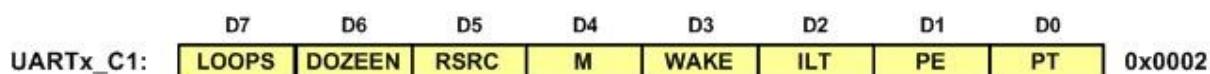


Figure 4-19: UART Control 1 (UARTx\_C1) register

Field	Bit	Description
LOOPS	D7	0 = Normal operation. RX and TX use separate pins. 1 = LOOP operation enabled. See KL25Z manual
DOZEEN	D6	Doze Mode Using this we can disable or enable UARTx in wait mode 0 = UART enabled in Wait mode 1 = UART disabled in Wait mode

		Receiver source bit. Used only when LOOPS=1 (see KL25Z manual)
<b>RSRC</b>	D5	0 = for internally connected loop 1 = for externally connected loop
		Data format mode bit. We must use this to select 8-bit data frame size
<b>M</b>	D4	0 = select 8-bit data frame, one stop bit and one start bit 1 = Select 9-bit data frame, one stop bit and one start bit
		Wake-up condition bit. See the KL25Z manual
<b>WAKE</b>	D3	0 = Idle line wakeup 1 = Address mark wake-up
		Idle line type bit. See the KL25Z manual
<b>ILT</b>	D2	0 = Idle character bit count begins after start bit 1 = Idle character bit count begins after stop bit
		Parity Enable bit. This will allow us to insert a parity bit right after the 8th (MSB) bit.
<b>PE</b>	D1	0 = no parity bit 1 = parity bit
		Parity bit type (used only if PE is one.)
<b>PT</b>	D0	0 = even parity bit 1 = odd parity bit
<b>Note:</b> The most important bit in this register is the M bit. The vast majority of the applications use M=0 for 8-bit data size. The rest of the bits are for testing purpose and we do not use them unless we are writing UART diagnostic test software. For that reason, we make them all zeros and we use <b>UARTx_C1 = 0x00</b> .		

Table 4-5: UART Control 1 (UARTx\_C1) register

## UART Control register 2 (UARTx\_C2)

Figure 4-20 describes various bits of the Control Register 2. Several of the C2 register bits are widely used by the UART. The TE (transmit enable) and RE (receive enable) are the most important bits in this register. The rest of the bits are used for interrupt driven serial communication. In Chapter 6 we will see how these flags are used with interrupts instead of polling.

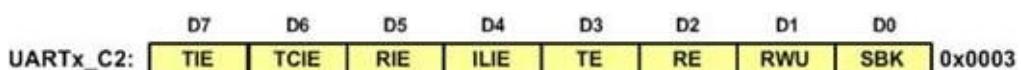


Figure 4-20: UART Control 2 (UARTx\_C2) register

Field	Bit	Description
<b>TIE</b>	D7	Transmit Interrupt Enable bit. Used for interrupt-driven UART. See Chapter 6. 0 = TDRE Interrupt Request is disabled. 1 = TDRE Interrupt Request is enabled.
<b>TCIE</b>	D6	Transmission Complete Interrupt Enable bit. Used for interrupt-driven UART. See Chapter 6. 0 = TC Interrupt Request is disabled. 1 = TC Interrupt Request is enabled.
<b>RIE</b>	D5	Receiver Full Interrupt Enable bit. Used for interrupt-driven UART. See Chapter 6. 0 = RDRF Interrupt Request is disabled. 1 = RDRF Interrupt Request is enabled.
<b>ILIE</b>	D4	Idle Line Interrupt Enable bit. Used for interrupt-driven UART. 0 = IDLE Interrupt Request is disabled. 1 = IDLE Interrupt Request is enabled.
<b>TE</b>	D3	Transmitter Enable bit. We must enable this bit to transmit data. 0 = Transmitter is disabled. 1 = Transmitter is enabled.
<b>RE</b>	D2	Receiver Enable bit. We must enable this bit to receive data. 0 = Receiver is disabled. 1 = Receiver is enabled.
<b>RWU</b>	D1	Used for wake-up condition in stand-by mode. See the KL25Z manual. 0 = Normal operation 1 = RWU is enabled.
<b>SBK</b>	D0	Used for break bit. See the KL25Z manual. 0 = No break character 1 = Transmit break character
<b>Note:</b> The most important bits in this register are the TE and RE bits. In applications using the polling method we make the interrupt request bits all zeros. For the polling method, we use <b>UARTx_C2 = 0x0C</b> . The		

rest of the bits are for testing purposes. To use interrupt-driven UART, see Chapter 6.

Table 4-6: UART Control 2 (UARTx\_C2) register

## UART Data Register

To transmit a byte of data we must place it in UART Data register. 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 this is an 8-bit register and located at address 0x4000 A007 for the UART0.

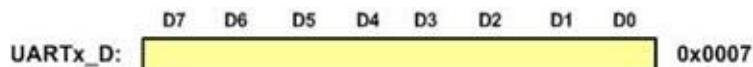
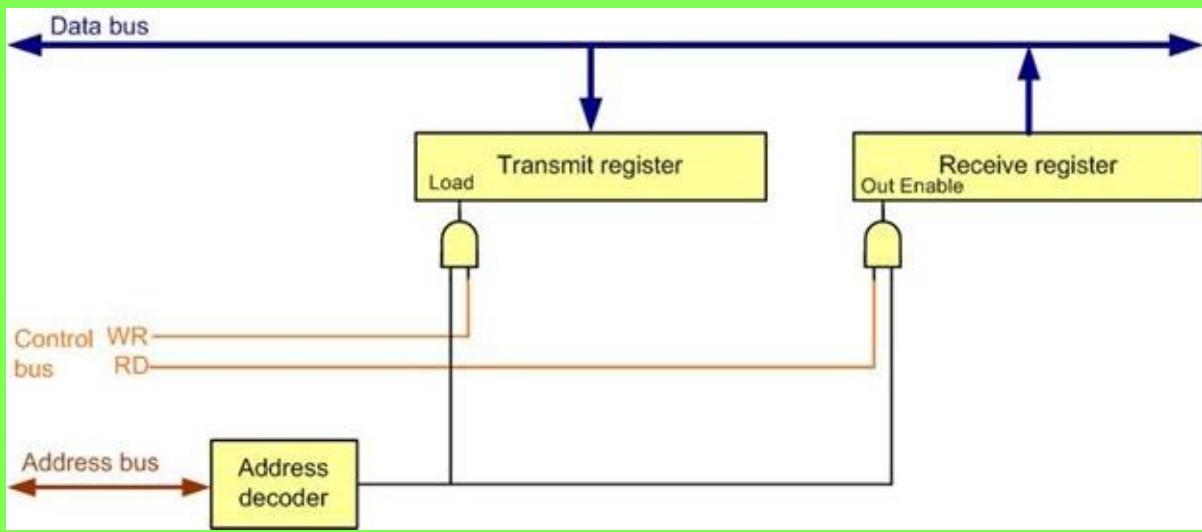


Figure 4-21: UART Data (UART\_D) register

### Note about UART\_D (advanced information)

There are in fact 2 separate registers with the same address and the same name, for transmitting and receiving data. Writing to the memory address leads to write to the transmit register and reads from the memory address return the received data. (Writing to the receive register and reading from the transmit register are meaningless and impossible.)



## UART Status 1 (UARTx\_S1) Register

We have two UART Status Registers: UARTx\_S1 and UARTx\_S2. They are 8-bit registers. The most important UART status register is the S1 and is used to monitor the arrival of data among other things. Figure 4-22 describes various bits of the S1. Several of the S1 register bits are widely used by the UART. We monitor (poll) the TC flag bit to make sure that all the bits of the last byte are transmitted. By the same logic, we monitor (poll) the RDRF flag to see if a byte of data is received. The transmitter is double buffered. That means there is a data register in addition to the shift register that shifts the bits out. While the shift

register is shifting the last byte out, the program may write another byte of data to the Data Register to wait for the shift register to be ready. The transfer of data between the data register and the shift register is automatic and the program does not have to worry about it. The TDRE flag indicates that the Data Register is empty and ready to accept another byte. The TC flag mentioned above actually indicates whether the shift register is empty or not. When the shift register is empty, the Data Register must be empty too and the UART has no data to transmit. In Chapter 6, we will see how these flags are used with interrupts instead of polling. The UART Status Register 2 is used for single-wire operation and is discussed in the KL25Z manual. We do not cover it in this textbook.

UARTx_S1:	D7	D6	D5	D4	D3	D2	D1	D0	
Type	TDRE	TC	RDRF	IDLE	OR	NF	OR	PF	0x0004
Reset	RO	RO	RO	RO	RO	RO	RO	RO	

Figure 4-22: UART Status Register (UARTx\_S1)

Field	Bit	Description
TDRE	D7	<p>Transmit Data Register Empty</p> <p>0 = The shift register is loaded and shifting. An additional byte is waiting in the Data Register.</p> <p>1 = The Data Register is empty and ready for the next byte.</p>
TC	D6	<p>Transmit Complete flag</p> <p>0 = Transmission is in progress (shift register is occupied)</p> <p>1 = No transmission in progress (both shift register and Data Register are empty)</p>
RDRF	D5	<p>Receive Data Register Full flag. This indicates a byte has been received and is sitting in UART Data Register and ready to be picked up.</p> <p>0 = No data is available in UART Data Register.</p> <p>1 = Data is available in UART Data Register and ready to be picked up.</p>
IDLE	D4	Idle line flag. See the KL25Z manual.
OR	D3	<p>D3 Overrun error</p> <p>0 = No overrun</p> <p>1 = Overrun error</p>
NF	D2	<p>Noise Flag error bit</p> <p>0 = No noise</p>

		1 = Noise error
		Framing Error bit
<b>FE</b>	D1	0 = No framing error 1 = Framing error
<b>PF</b>	D0	Parity flag error bit 0 = No parity error 1 = Parity error

Table 4-7: UART Status Register (UARTx\_S1)

### ***The importance of the TDRE***

To transmit a byte of data serially via the TXD pin, we must write it into the UART Data Register (UARTx\_D). The transmit shift register is an internal register whose job is to get the data from the UART Data Register (UARTx\_D), frame it with the start and stop bits, and send it out one bit at a time via the UARTx\_TX pin. Notice that the transmit shift register is a parallel-in-serial-out shifter and is not accessible to the programmer. We can only write to the UART Data Register. Whenever the shifter is empty, it gets its new data from the UART Data Register and clears the UART Data Register immediately, so it does not send out the same data twice. When the shifter fetches the data from the UART Data Register, it clears the TDRE flag to indicate it is empty and the UART Data Register is ready for the next character. We must check the TDRE flag before we write another byte to the UART Data Register.

### ***The importance of the TC***

The TC flag indicates that both the Data Register and the shift register are empty and there is no data left for the transmitter to send. This bit is used when the program needs to know that all the data is sent before starting the next task. When TDRE flag is set, the Data Register is empty but the last byte of data may still be in the shift register. If the program is to shut down the transmitter, the last byte of data will be lost. Checking TC ensures that all the data written to the UART is transmitted already.

### ***The importance of the RDRF***

The internal serial-in-parallel-out receive register receives data via the UARTx\_RX pin. It gets rid of the start and stop bits and writes the received byte to the UART Data Register and makes RDRF high. We must check the RDRF flag to see if we need to pick up the received byte.

### ***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 System Clock Gating Control register 4

(SIM\_SCGC4) register is used to enable the clock to the UARTs. In this register, there is a bit for each of the UART0 to UART2 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 D10 bit of this register. The other bits of this register are used for enabling the clock to other on-chip peripherals. See Figure 4-23.

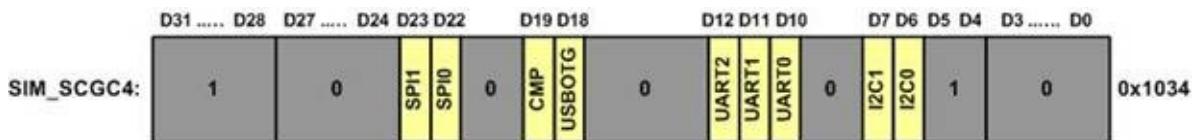


Figure 4-23: SIM\_SCGC4 (System Clock Gating Control) Register

## 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 I/O 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 simple I/O and provided the Clock to it. When I/O pins are used for their alternate peripheral functions such as UART, Timer, and ADC, we need to use the PORTx\_PCRn (Portx Pin Control) special function register. As we showed in the last two chapters, each pin of ports A-E has its own PORTx\_PCRn register. The x is used for Ports A to E and n is used for pin number 0 to 31. The most important bits of PORTx\_PCRn are D10-D8 (MUX control). Upon reset, ports A to E are disabled (default). To use a pin as simple I/O, we must choose MUX=001 option or ALT1 in the PORTx\_PCRn register. The ALT2 option allows PTA1 and PTA2 pins to be used for UART0\_Rx and UART0\_Tx signals, respectively. For others pin and UART combination, other ALT numbers should be used according to Table 4-8. Appendix B shows the pin function selection for all the alternate of ALT0 to ALT7. In the case of PTA1, we must use the PORTA\_PCR1 register and for PTA2 we use the PORTA\_PCR2 register.

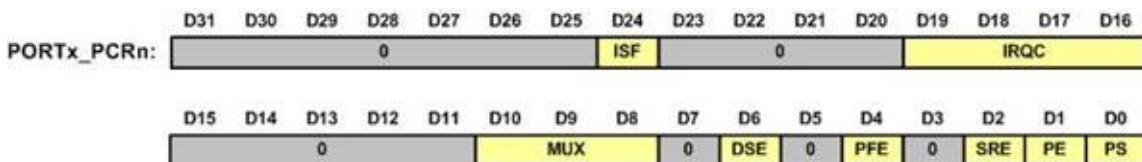


Figure 4-24: PORTx\_PCRn Alternate Function Selection register

FRDM I/O Pin	KL25Z Pin	ALT2	ALT3	ALT4
J1 02	PTA1	UART0_RX		
J1 04	PTA2	UART0_TX		
-	PTA14		UART0_TX	
-	PTA15		UART0_RX	
-	PTA18		UART1_RX	

	PTA19	UART1_TX
	PTB16	UART0_RX
	PTB17	UART0_TX
J1 05	PTC3	UART1_RX
J1 07	PTC4	UART1_TX
J2 08	PTD2	UART2_RX
J2 10	PTD3	UART2_TX
J1 06	PTD4	UART2_RX
J2 04	PTD5	UART2_TX
J2 17	PTD6	UART0_RX
J2 19	PTD7	UART0_TX
J2 20	PTE0	UART1_TX
J2 18	PTE1	UART1_RX
J10 01	PTE20	UART0_TX
J10 03	PTE21	UART0_RX
J10 05	PTE22	UART2_TX
J10 07	PTE23	UART2_RX

Table 4-8: Pins available for UARTs

## Steps for transmitting data

Here are the steps to configure the UART0 and transmit a byte of data for Freescale FRDM board:

- 1) Provide clock to UART0 by writing a 1 to D10 bit of SIM\_SCGC4 register.
- 2) Select FLL as UART0 clock source by setting bit 27-26 to 01 and bit 16 to 0 in SIM\_SOPT2.
- 3) Turn off the UART0 before changing configurations by clearing UART0\_C2 register.
- 4) Set the baud rate for UART0 by using UART0\_BDH and UART0\_BDL registers.
- 5) Writing 0x0F to UART0\_C4 register to select 16 for Over Sampling Ratio.
- 6) Configure the control register value for 1 stop bit, no parity, and 8-bit data size by writing 0x00 for the UART0\_C1 register.
- 7) Write 0x08 to UART0\_C2 register to enable the transmitter of UART0.
- 8) Set bit 9 of SIM\_SCGC5 to 1 to enable clock to PORTA.

- 9) Select the alternate functions 2 for PA2 (UART0\_Tx) pins using the PORTA\_PCR2.
- 10) Monitor the TDRE bit of the Status Register 1 (UART0\_S1) and wait for UART0 transmit buffer empty.
- 11) Write a byte to UART0 Data Register to be transmitted.
- 12) To transfer the next character, go to step 10.

Program 4-1 sends the characters “YES” to the terminal emulator program (TeraTerminal) on 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 UART0 transmit

 * Sending "YES" to UART0 on Freescale FRDM-KL25Z board.
 * UART0 is connected to openSDA debug agent and has
 * a virtual connection to the host PC COM port.
 * Use TeraTerm to see the message "YES" on a PC.

 * By default in SystemInit(), FLL clock output is 41.94 MHz.
 * Setting BDH=0, BDL=0x17, and OSR=0x0F yields 115200 Baud.

 */

#include <MKL25Z4.H>

void UART0_init(void);
void delayMs(int n);

int main (void) {
    UART0_init();
    while (1) {
        while (!(UART0->S1 & 0x80)) {
            /* wait for transmit buffer empty */
        }
        /* send character here */
    }
}
```

```

UART0->D = 'Y'; /* send a char */

while(!(UART0->S1 & 0x80)) { }

UART0->D = 'e'; /* send a char */

while(!(UART0->S1 & 0x80)) { }

UART0->D = 's'; /* send a char */

delayMs(2); /* leave a gap between messages */

}

}

/* initialize UART0 to transmit at 115200 Baud */

void UART0_init(void) {

    SIM->SCGC4 |= 0x0400; /* enable clock for UART0 */

    SIM->SOPT2 |= 0x04000000; /* use FLL output for UART Baud rate generator */

    UART0->C2 = 0; /* turn off UART0 while changing configurations */

    UART0->BDH = 0x00;

    UART0->BDL = 0x17; /* 115200 Baud */

    UART0->C4 = 0x0F; /* Over Sampling Ratio 16 */

    UART0->C1 = 0x00; /* 8-bit data */

    UART0->C2 = 0x08; /* enable transmit */

}

SIM->SCGC5 |= 0x0200; /* enable clock for PORTA */

PORTA->PCR[2] = 0x0200; /* make PTA2 UART0_Tx pin */

}

/* Delay n milliseconds */

/* The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit(). */

void delayMs(int n) {

    int i;

    int j;

    for(i = 0 ; i < n; i++)

        for (j = 0; j < 7000; j++) {}

}

```

## Importance of the TDRE (Transmit Data Register Empty) flag

To understand the importance of the role of TDRE, look at the following

sequence of steps that the KL25Z goes through in transmitting a character via TXD:

1. The byte character to be transmitted is written into the UARTx Data Register.
2. The TDRE flag is set to 0 internally to indicate that the UARTx Data Register has a byte and will not accept another byte until this one is transmitted.
3. The transmit shift register reads the byte from the UARTx Data Register and begins to transfer the byte starting with the start bit.
4. The TDRE flag is set to 1 to indicate that the last byte is being transmitted and the UARTx Data Register is ready to accept another byte.
5. The 8-bit character is transferred one bit at a time.

By monitoring the TDRE flag, we make sure that we are not overloading the UARTx Data Register. If we write another byte into the UARTx Data Register before the shifter has fetched the last one, the old byte could be lost before it is transmitted.

Notice that we can also check the TC (transmit complete) flag before loading the UARTx data register with a new byte.

## Steps for receiving data

Here are the steps to receive a byte of data for UART0 in Freescale ARM FRDM board:

Provide clock to UART0 by writing a 1 to D10 bit of SIM\_SCGC4 register.

Select FLL as UART0 clock source by setting bit 27-26 to 01 and bit 16 to 0 in SIM\_SOPT2.

Turn off the UART0 before changing configurations by clearing UART0\_C2 register.

Set the baud rate for UART0 by using UART0\_BDH and UART0\_BDL registers.

Writing 0x0F to UART0\_C4 register to select 16 for Over Sampling Ratio.

Configure the control register value for 1 stop bit, no parity, and 8-bit data size by writing 0x00 for the UART0\_C1 register.

Write 0x04 to UART0\_C2 register to enable the receiver of UART0.

Set bit 9 of SIM\_SCGC5 to 1 to enable clock to PORTA.

Select the alternate functions 2 for PA1 (UART0\_Rx) pins using the PORTA\_PCR1.

Monitor the RDRE (receive data register full) bit of the Status Register 1

(UART0\_S1) and wait for UART0 receive buffer full.

When RDRE is set, read a byte of data from UART0\_D register and use it to set the tri-color LEDs.

To receive another character, go to step 10.

### Note

The configuration steps (steps 1 to 6) are identical for receiving and sending data.

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

### Program 4-2: UART0 Receive

```
/* p4_2.c UART0 Receive

 * Receiving any key from terminal emulator (TeraTerm) of the
 * host PC to the UART0 on Freescale FRDM-KL25Z board.
 * UART0 is connected to openSDA debug agent and has
 * a virtual connection to the host PC COM port.
 * Launch TeraTerm on a PC and hit any key.
 * The LED program from P2_7 of Chapter 2 is used to turn
 * on the tri-color LEDs according to the key received.
 *
 * By default in SystemInit(), FLL clock output is 41.94 MHz.
 * Setting BDH=0, BDL=0x17, and OSR=0x0F yields 115200 Baud.
 */


```

```
#include <MKL25Z4.H>

void UART0_init(void);
void delayMs(int n);
void LED_init(void);
void LED_set(int value);
```

```

int main (void) {
    char c;

    UART0_init();
    LED_init();

    while (1) {
        while (!(UART0->S1 & 0x20)) {
            /* wait for receive buffer full */

            c = UART0->D; /* read the char received */

            LED_set(c);
        }
    }

/* initialize UART0 to receive at 115200 Baud */
void UART0_init(void) {
    SIM->SCGC4 |= 0x0400; /* enable clock for UART0 */

    SIM->SOPT2 |= 0x04000000; /* use FLL output for UART Baud rate generator */
/*
    UART0->C2 = 0;           /* turn off UART0 while changing configurations */
    UART0->BDH = 0x00;
    UART0->BDL = 0x17;       /* 115200 Baud */
    UART0->C4 = 0x0F;         /* Over Sampling Ratio 16 */
    UART0->C1 = 0x00;         /* 8-bit data */
    UART0->C2 = 0x04;         /* enable receive */

    SIM->SCGC5 |= 0x0200; /* enable clock for PORTA */
    PORTA->PCR[1] = 0x0200; /* make PTA1 UART0_Rx pin */
}

/* initialize all three LEDs on the FRDM board */
void LED_init(void)
{
    SIM->SCGC5 |= 0x400;      /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000;     /* enable clock to Port D */
    PORTB->PCR[18] = 0x100;   /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x40000;    /* make PTB18 as output pin */
    PTB->PSOR = 0x40000;     /* turn off red LED */
    PORTB->PCR[19] = 0x100;   /* make PTB19 pin as GPIO */
}

```

```

PTB->PDDR |= 0x80000;           /* make PTB19 as output pin */
PTB->PSOR = 0x80000;           /* turn off green LED */
PORTD->PCR[1] = 0x100;         /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02;            /* make PTD1 as output pin */
PTD->PSOR = 0x02;             /* turn off blue LED */
}

/* turn on or off the LEDs according to bit 2-0 of the value */
void LED_set(int value)
{
    if (value & 1)      /* use bit 0 of value to control red LED */
        PTB->PCOR = 0x40000;    /* turn on red LED */
    else
        PTB->PSOR = 0x40000;    /* turn off red LED */
    if (value & 2)      /* use bit 1 of value to control green LED */
        PTB->PCOR = 0x80000;    /* turn on green LED */
    else
        PTB->PSOR = 0x80000;    /* turn off green LED */
    if (value & 4)      /* use bit 2 of value to control blue LED */
        PTD->PCOR = 0x02;       /* turn on blue LED */
    else
        PTD->PSOR = 0x02;       /* turn off blue LED */
}

```

## ***Importance of the RDRF (Receive Data Register Full) flag bit***

In receiving bits via its RXD pin, the KL25Z goes through the following steps:

1. The receiver's shift register receives the start bit indicating that the next bit is the first bit of the character byte it is about to receive.
2. The 8-bit character is received one bit at time. When the last bit is received, a byte is formed and placed in UARTx Data Register (UARTx\_D) and the KL25Z makes RDRF = 1, indicating that an entire character byte has been received and must be picked up before it gets overwritten by another incoming character.
3. By checking the RDRF flag bit when it is raised, we know that a character has been received and is sitting in the UARTx Data Register. We copy the UARTx Data register contents to a safe place in some other register or memory before it is lost. Notice that for receiving data we use the same register as when sending data.

## Using UART1 or UART2 port

The previous two programs showed how to use the UART0 on Freescale FRDM board, which is connected to the host computer through the OpenSDA USB cable. Now, you can buy a USB-to-Serial module (or cable) and connect to either UART1 or UART2 port. One side of the USB-to-Serial module should be 3.3V logic level signals for TxD and RxD and is connected to the UARTx pins on Freescale ARM FRDM board. The other side is USB port connected to the PC USB port. See these links:

<https://www.sparkfun.com/products/9893>,  
<https://www.sparkfun.com/products/9717>,

<http://www.adafruit.com/products/284> or <http://www.adafruit.com/products/70>.

Make sure you are using a 3.3V logic level converter. Many TTL level serial to USB converters produce output higher than 3.6V. They may appear functional but will damage the KL25Z input pins.

## UART1 and UART2 Baud clock and oversampling

Compared to UART0, the other two UART modules have simpler Baud rate generator clock scheme. Their clock source is connected to the system bus clock, which is configured to 13.98MHz by Keil default startup code. The Over Sampling Ratio is fixed to 16.

$$\text{Baud Rate} = \frac{\text{Bus Clock}}{16 \times \text{SBR}}$$

So if we set `UART0_BDH` = 0 and `UART0_BDL` = 91, then

$$\text{Baud Rate} = \frac{13,980,000}{16 \times 91} = 9601$$

Table 4-9 shows the SBR values for the different baud rate for UART1 using bus clock of 13.98 MHz.

Baud rate	SBR (in decimal)	SBR (in hex)
4,800	182	0x00B6
9,600	91	0x005B
19,200	46	0x002E
38,400	23	0x0017

Table 4-9: UART1 and UART2 SBR Values for Some Baud Rates using bus clock of 13.98 MHz

Program 4-3 is modified from Program 4-1 to use UART1. Comparing them you will find that the initialization of the associated port is changed. Besides the change of UART module and the port pin assignments, the Baud rate generation

configuration is different from UART0. 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 TeraTerm via UART2

```
/* p4_3.c Sending "Hello" through UART2

 * This program sends a message "Hello" through UART2.
 * The bus clock is set to 13.98 MHz in SystemInit().
 * Baud rate = bus clock / BDH:BDL / 16 = 9600
 * A terminal emulator (TeraTerm) should be launched
 * on the host PC to display the message.
 * The UART2 transmit line is connected to PTD5.
 */

#include <MKL25Z4.H>

void UART2_init(void);
void delayMs(int n);

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

    UART2_init();
    while (1) {
        for (i = 0; i < 7; i++) {
            while (!(UART2->S1 & 0x80)) {
                /* wait for transmit buffer empty */
            }
            UART2->D = message[i]; /* send a char */
        }
        delayMs(10); /* leave a gap between messages */
    }
}

/* initialize UART2 to transmit at 9600 Baud */
void UART2_init(void) {
```

```

SIM->SCGC4 |= 0x1000; /* enable clock to UART2 */
UART2->C2 = 0;           /* disable UART during configuration */
UART2->BDH = 0x00;
UART2->BDL = 0x5B;       /* 9600 Baud */
UART2->C1 = 0x00;         /* normal 8-bit, no parity */
UART2->C3 = 0x00;         /* no fault interrupt */
UART2->C2 = 0x08;         /* enable transmit, no interrupt */
SIM->SCGC5 |= 0x1000;    /* enable clock to PORTD */
PORTD->PCR[5] = 0x300;   /* PTD5 for UART2 transmit */
}

/* Delay n milliseconds */
/* The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit(). */

void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
}

```

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

#### Program 4-4: Echoing the received data from UART2

```

/* p4_4.c UART2 echo

 * This program receives a character from UART2 receiver
 * then sends it back through UART2.

 * The bus clock is set to 13.98 MHz in SystemInit().
 * Baud rate = bus clock / BDH:BDL / 16 = 9600
 * A terminal emulator (TeraTerm) should be launched
 * on the host PC. Typing on the PC keyboard sends
 * characters to the FRDM board. The FRDM board echoes
 * the character back to the terminal emulator.

```

```

 * The UART2 transmit line is connected to PTD5.
 *
 * The UART2 receive line is connected to PTD4.
 */
#include <MKL25Z4.H>

void UART2_init(void);
void delayMs(int n);

int main (void) {
    char c;
    UART2_init();
    while (1) {
        while (!(UART2->S1 & 0x20)) {
            /* wait for receive buffer full */
            c = UART2->D ; /* read the char received */

        while (!(UART2->S1 & 0x80)) {
            /* wait for transmit buffer empty */
            UART2->D = c; /* send the char received */
        }
    }
}

/* initialize UART2 to transmit and receive at 9600 Baud */
void UART2_init(void) {
    SIM->SCGC4 |= 0x1000; /* enable clock to UART2 */
    UART2->C2 = 0; /* disable UART during configuration */
    UART2->BDH = 0x00;
    UART2->BDL = 0x5B; /* 9600 Baud */
    UART2->C1 = 0x00; /* normal 8-bit, no parity */
    UART2->C3 = 0x00; /* no fault interrupt */
    UART2->C2 = 0x0C; /* enable transmit and receive */
    SIM->SCGC5 |= 0x1000; /* enable clock to PORTD */
    PORTD->PCR[5] = 0x300; /* PTD5 for UART2 transmit */
    PORTD->PCR[4] = 0x300; /* PTD5 for UART2 receive */
}

```

## Baud rate error calculation

In calculating the baud rate, we have used the integer number for the UARTx\_BDH:UARTx\_BDL register (SBR) values because KL25Z microcontrollers can only use integer values. By dropping the decimal fraction portion of the calculated values we run the risk of introducing error into the baud rate. One way to calculate this error is to use the following formula.

$$\text{Error} = (\text{Calculated value for the SBR} - \text{Integer part}) / \text{Integer part}$$

See Example 4-5.

### Example 4-5

Calculate the baud rate error for Example 4-4.

**Solution:**

(a)  $\text{SBR} = 41,940,000 / (7 + 1) / 9600 = 546.094$

$$\text{Error} = (546.094 - 546) / 546 \times 100 = 0.02\%$$

(b)  $\text{SBR} = 41,940,000 / (15 + 1) / 115200 = 22.754$

$$\text{Error} = (22.754 - 23) / 23 \times 100 = -1.07\%$$

(c)  $\text{SBR} = 13,980,000 / 16 / 38400 = 22.754$

$$\text{Error} = (22.754 - 23) / 23 \times 100 = -1.07\%$$

(d)  $\text{SBR} = 41,940,000 / (7 + 1) / 9600 = 7.5846$

$$\text{Error} = (7.5846 - 8) / 8 \times 100 = -5.19\%$$

## Interrupt-based data transfer

By now you might have noticed that it is a waste of the microcontroller's time to poll the TDRE and RDRF flags. In order to avoid wasting the microcontroller's time we use interrupts instead of polling. In Chapter 6, we will show how to use interrupts to program the KL25Z's serial communication port.

## Idle and break characters

In the KL25Z manual we see the mention of some terminology such as idle and break. The idle is when the UART output is high with no start bit. That is ten ones when the data frame size is 8-bit. The break character is when the UART sends out a low signal much longer than one frame, that is ten zeros when the data size is 8-bit. The break character is used to force a framing error at the receiver for the testing purpose and writing diagnostic software.

## Review Questions

1. The Freescale FRDM comes with maximum of \_\_\_\_\_ on-chip UARTs.
2. In Freescale FRDM-KL25Z board, 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) stop bit)	(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. 3
2. PTA0 and PTA1
3. UARTx\_C1
4. The TDRE flag from the UART0\_S1 register goes low.
5. The RDRE flag from the UART0\_S1 register goes low.





# **Chapter 5: Freescale 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-bit Freescale timers. Section 5-3 shows Output Compare mode. In Section 5-4, input edge-time mode is discussed and the pulse width and frequency measuring are covered. The event counter feature is studied in Section 5-5.



## 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 flip flops.

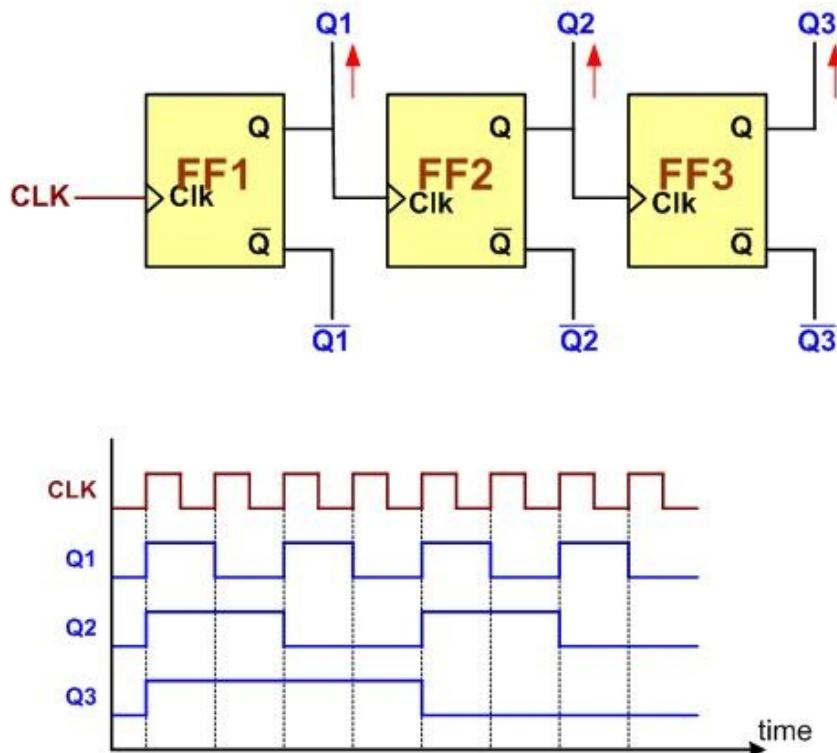


Figure 5-1: A 3-bit Counter

Regarding Figure 5-1, notice the following points:

- 1) The Q outputs give the down counter.
- 2) The  $\bar{Q}$  (Q not) outputs give us up counter.
- 3) The frequency on Q3 is  $\frac{1}{8}$  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.

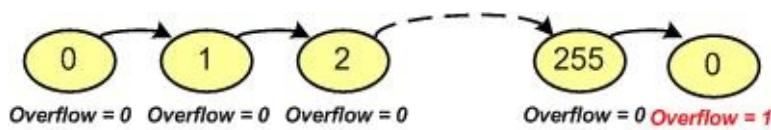


Figure 5-2: an 8-bit Up-Counter Stages

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.

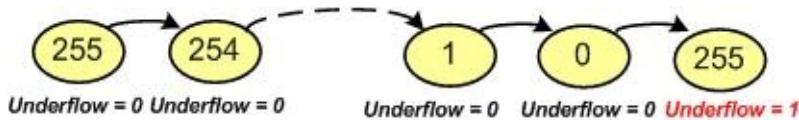


Figure 5-3: an 8-bit Down-Counter Stages

## 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 packages 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.

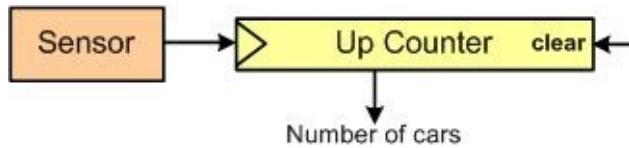


Figure 5-4: Counting Events Using a Counter

### **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.

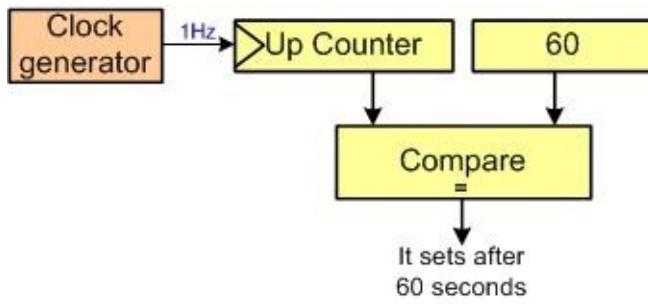


Figure 5-5: Using Counter as a Timer

### **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:

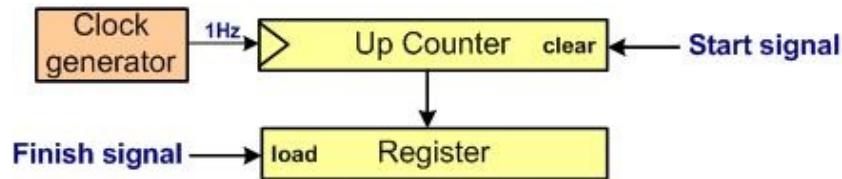


Figure 5-6: Capturing

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.

## Counters and Timers in microcontrollers

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

With 5 FFs we can get maximum of \_\_\_\_\_ count.

With 5 FFs we can divide the frequency by maximum of \_\_\_\_\_.

When pulses are fed to a timer from the outside it is called \_\_\_\_\_.

When clocks pulses are fed to a timer from inside it is called \_\_\_\_\_.

If we need to divide a frequency by 500, we need \_\_\_\_\_ flip flops.



## Section 5.1: System Tick Timer

Every ARM Cortex-M comes with a 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.

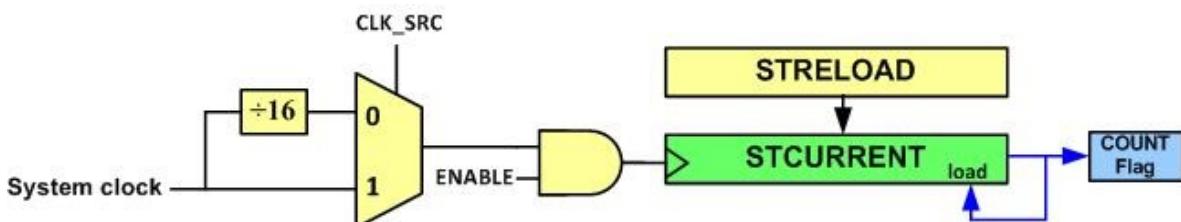


Figure 5-7: System Tick Timer Internal Structure

The down counter is named as STCURRENT (SysTick->VAL) in Freescale 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.

SysTick->CTRL

STCTRL:	D31	....	D17	D16	D15	...	D3	D2	D1	D0	0x010
	Reserved		COUNT		Reserved		CLK_SRC	INTEN	ENABLE		
<b>SysTick-&gt;CTRL</b>											
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									
2	CLK_SRC	Clock Source 0: System clock divided by 16 1: System clock									
16	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 STCURRENT register.</i>									

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.

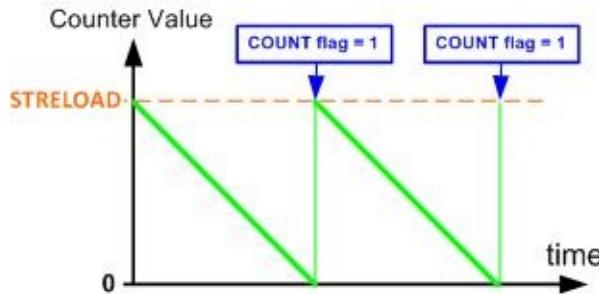


Figure 5-9: System Tick Counting

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

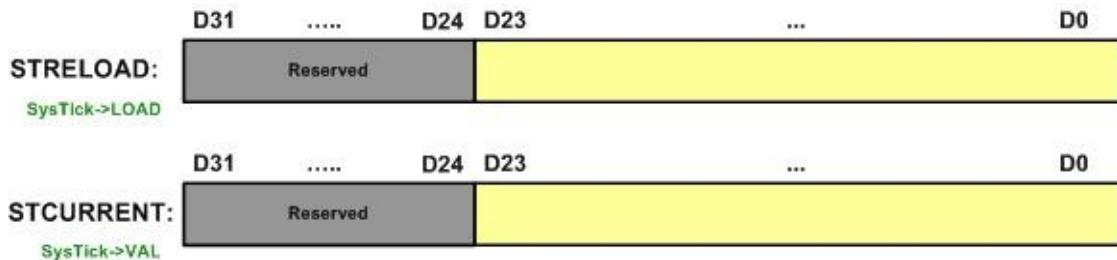
**CLK\_SRC (Clock Source D2):** We have the choice of clock coming from System clock or Precision Internal Oscillator (PIOOSC). If CLK\_SRC=0 then the clock comes from PIOOSC/4. If CLK\_SRC=1, then the system clock provides the clock source to SysTick down counter.

**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 0xFFFFFFF or 16,777,216 decimal. 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 PORTB as it counts down. The value of STCURRENT (SysTick->VAL) is shifted 4 places to the right so that the most significant bit is aligned with PTB19, which is connected to the green LED. SysTick is configured to use default system clock at 41.94 MHz. The STRELOAD (SysTick->LOAD) register has 24 bits and is set to the maximal value. So the counter has the frequency of

$$\frac{41,940,000 \text{ Hz}}{2^{24}} = \frac{41,940,000 \text{ Hz}}{16,777,216} \approx 2.5 \text{ Hz}$$

And that is the frequency of the green LED. The red LED is connected to PTB18 therefore runs twice as fast as the green LED at PTB19.

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

```
/* p5_1.c Toggling LEDs using SysTick counter
```

This program let the SysTick counter run freely and dumps the counter values to the tri-color LEDs continuously.

The counter value is shifted 4 places to the right so that the changes of LEDs will be slow enough to be visible.

SysTick counter has 24 bits.

The red LED is connected to PTB18.

The green LED is connected to PTB19.

\*/

```
#include <MKL25Z4.H>
```

```
int main (void) {
```

```
    int c;
```

```
    SIM->SCGC5 |= 0x400; /* enable clock to Port B */
```

```

PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
PTB->PDDR |= 0xC0000; /* make PTB18, 19 as output pin */

/* Configure SysTick */

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

while (1) {
    c = SysTick->VAL; /* read current value of down counter */
    PTB->PDOR = c >> 4; /* line up counter MSB with LED */
}

}

```

See the following examples.

### Example 5-1

In an ARM microcontroller system clock = 8 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 */

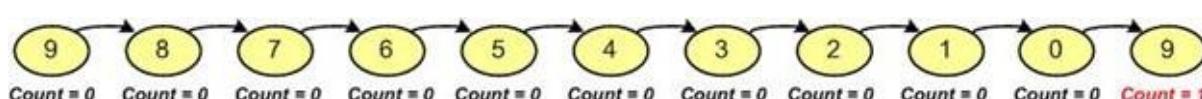
    while((SysTick->CTRL &0x10000) == 0) /*wait until the Count flag is set */
    {}

    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

$$\frac{1}{8 \text{MHz}} = \frac{1}{8,000,000} = 0.125\mu\text{s}$$

So, the program makes a delay of  $10 \times 0.125\mu\text{s} = 1.25\mu\text{s} = 1.250\text{ms}$ .

**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 clock 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 clock is chosen as the clock source, each clock lasts  $1 / \text{sysclk}$ .  
So, the program makes a delay of  $(N + 1) \times (1 / \text{sysclk}) = (N + 1) / \text{sysclk}$ .

---

## Example 5-3

Using the System Tick timer, write a function that makes a delay of 1 ms. Assume sysclk = 41.94 MHz.

### Solution:

From the equation derived in Example 5-2

$$\text{delay} = (N + 1) / \text{sysclk}$$

$$(N + 1) = \text{delay} \times \text{sysclk} = 0.001 \text{ sec} \times 41.94 \text{ MHz} = 41,940 \Rightarrow N = 41,940 - 1 = 41939$$

```
void delay1ms(void) {  
    SysTick->LOAD = 41939;  
    SysTick->CTRL = 0x5;      /* Enable the timer and choose sysclk 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 PTB18 every 200 milliseconds. We need the RELOAD value of 8,387,999 since  $0.200 \text{ sec} * 41.94$

MHz = 8,388,000. We assume the system clock is 41.94 MHz. Notice, every 8,388,000 clocks the down counter reaches 0, and COUNT flag is raised. Then the RELOAD register is loaded with 8,388,000 automatically. The COUNT flag is clear when the STCTRL (SysTick->CTRL) register is read.

### Program 5-2: Toggle red LED at 5 Hz using the SysTick Counter

```
/* p5_2.c Toggling red LED at 5 Hz using SysTick

 * This program uses SysTick to generate 200 ms delay to
 * toggle the red LED.
 * System clock is running at 41.94 MHz. SysTick is configured
 * to count down from 8387999 to give a 200 ms delay.
 * The red LED is connected to PTB18.
 */

#include <MKL25Z4.H>

int main (void) {
    SIM->SCGC5 |= 0x0400;          /* enable clock to Port B */
    PORTB->PCR[18] = 0x100;        /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x040000;         /* make PTB18 as output pin */
    /* Configure SysTick */
    SysTick->LOAD = 8388000 - 1; /* reload with number of clocks per 200 ms */
    SysTick->CTRL = 5;           /* enable it, no interrupt, use system clock */

    while (1)
    {
        if (SysTick->CTRL & 0x10000) /* if COUNT flag is set */
            PTB->PTOR = 0x040000;   /* toggle red LED */
    }
}
```

In Program 5-3, SysTick is used to generate multiple of 1 millisecond delay. RELOAD value of 41,939 is used since  $0.001 \text{ sec} * 41.94 \text{ MHz} = 41,940$ .

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

```
/* p5_3.c Toggling green LED using SysTick delay
```

```

 * This program uses SysTick to generate one second delay to
 * toggle the green LED.
 *
 * System clock is running at 41.94 MHz. SysTick is configured
 * to count down from 41939 to give a 1 ms delay.
 *
 * For every 1000 delays (1 ms * 1000 = 1 sec), toggle the
 * green LED once. The green LED is connected to PTB19.
 */

#include <MKL25Z4.H>

int main (void) {
    void delayMs(int n);

    SIM->SCGC5 |= 0x400;           /* enable clock to Port B */
    PORTB->PCR[19] = 0x100;        /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0x080000;         /* make PTB19 as output pin */

    while (1) {
        delayMs(1000);           /* delay 1000 ms */
        PTB->PTOR = 0x080000;     /* toggle green LED */
    }
}

void delayMs(int n) {
    int i;
    SysTick->LOAD = 41940 - 1;
    SysTick->CTRL = 0x5; /* Enable the timer and choose sysclk as the clock
source */

    for(i = 0; i < n; i++) {
        while((SysTick->CTRL & 0x10000) == 0) /* wait until the COUTN flag is set */
    }
    SysTick->CTRL = 0; /* Stop the timer (Enable = 0) */
}

```

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, Freescale has its own timers which are covered in the next section.

## Review Questions

True or false. The highest number we can place in RELOAD register is \_\_\_\_\_.

Assume CPU frequency of 16MHz. Find the value for RELOAD register if we want 5 ms elapsed time.

The SysTick is \_\_\_\_\_-bit wide.

Which bit of STCTRL is used to enable the SysTick.

The SysTick is (down or up) counter.



## Section 5.2: Delay Generation with Freescale Timers

In this section we examine the timers for Freescale KL25Z ARM chip. We will use KL25Z timer to create time delay on Freescale FRDM board.

### Timer Registers

In Freescale KL25Z microcontrollers, the timers are called *Timer/ PWM Module (TPM)*. There are 3 Timer Modules in the KL25Z, each supporting up to 6 channels. The Timer modules support Output Compare, Input capture, and PWM. The Input Capture and event counter are covered in the next section and PWM (pulse width module) feature is covered in Chapter 11.

The Timer Modules in KL25Z are designated as TPMx in which x = 0, 1, or 2. In other words, there are TPM0, TPM1, and TPM2. The following shows the base addresses for the Timer modules:

- TPM0 base: 0x4003 8000
- TPM1 base: 0x4003 9000
- TPM2 base: 0x4003 A000

### Enabling Clock to TPMx

Before we can use any of the Timer Modules, we must enable the clock to it. This is done with the System Clock Gating Register 6 (SIM\_SCGC6 register), as shown below:

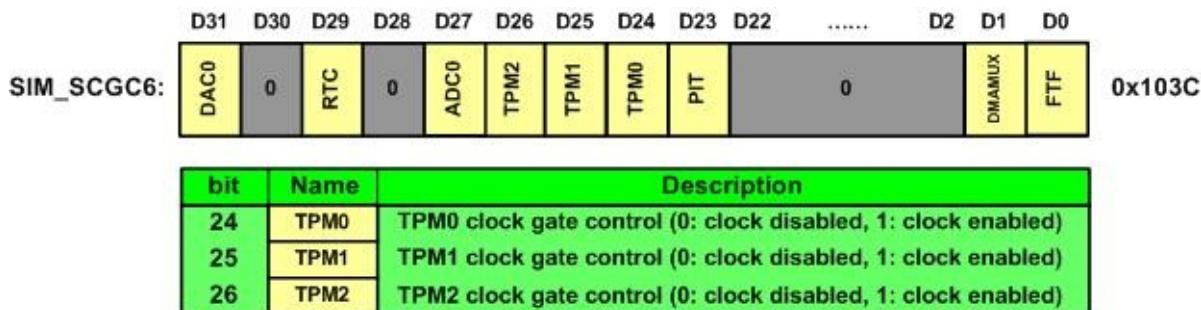


Figure 5-11: SIM\_SCGC6 (SIM Clock Gating Control Register 6)

The SIM\_SCGC6 is part of the SIM (System Integration Module). The details of SIM are shown in Chapter 12 of KL25Z reference manual. Just like GPIO and UART, we must enable the clock to TPM0 –TPM2 modules before we can use them. Notice, in SIM\_SCGC6 registers, bit D24 is for TPM0 module, bit D25 is for TPM1 module, and bit D26 is for TPM2 module.

This clock is used to operate the timer module circuit. The core of the timer is a counter, which receives a different clock.

### Enable Timer Counter Clock

The clock that drives the timer counter is selected by the TPMSRC bits and the PLLFLLSEL bit of SIM\_SOPT2 register in System Integration Module.

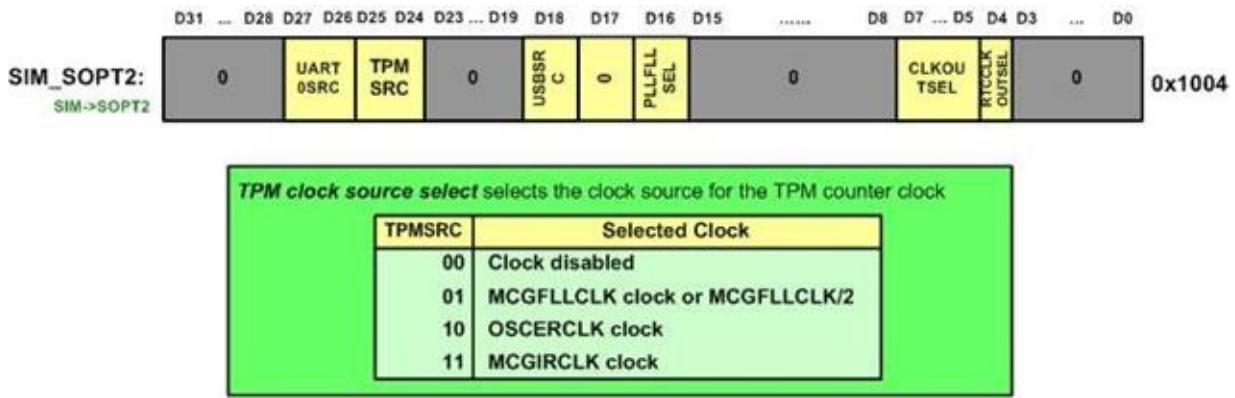


Figure 5-12: SIM\_SOPT2 (System Options 2)

Upon reset, the timer counter clock is disabled. The possible clock sources are MCGFLLCLK, MCGPLLCLK/2, OSCERCLK and MCGIRCLK (KL25Z Ref Man Section 5.7.5). The clock source availability depends on the configuration of the MCG (Multiple Clock Generation) Module. The Keil MDK-ARM v5 supports three MCG configurations in the startup code of system\_MKL25Z4.c file and is default to Mode 0. The available clock sources and frequency for FRDM-MKL25Z are shown in the table below.

MCG Mode	MCGFLLCLK	MCGPLLCLK/2	OSCERCLK	MCGIRCLK
0	41.94 MHz	N.A.	N.A.	32.768 kHz
1	N.A.	48.0 MHz	8.0 MHz	N.A.
2	N.A.		8.0 MHz	N.A.

**Note:** N.A.: Not Applicable

Table 5-1: Clock Sources in FRDM-MKL25Z

### TPM COUNT Register (TPMx\_CNT)

Each of the Timer modules has a 16-bit counter. It is called TPMx\_CNT in which  $x = 0, 1, \text{ or } 2$ . That means we have TPM0\_CNT, TPM1\_CNT, and TPM2\_CNT. When the clock is fed to TPMx\_CNT, it keeps counting up (or counting down). TPMx\_CNT is a 16-bit counter register. Although the TPMx\_CNT is a 32-bit register only 16-bits are used. We can read its content as it counts. Upon Reset TPMx\_CNT=0000. See Figure 5-13. The discussion about TPM0 also applies equally to TPM1 and TPM2.

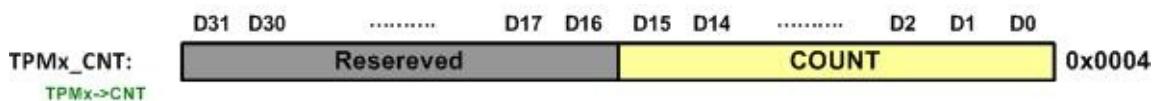


Figure 5-13: TPMx\_CNT Register

### TPM Modulo Register (TPMx\_MOD)

Each TPM has a Modulo (TPMx\_MOD) register. It is a 16-bit register whose

value is continuously compared with the TPMx\_CNT register. See Figures 5-14 and 5-15.

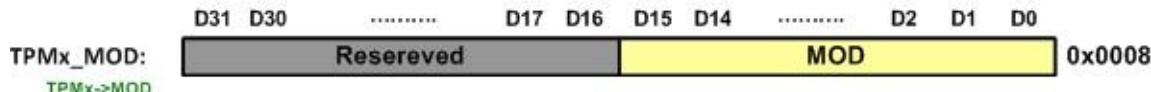


Figure 5-14: TPMx\_CNT and TPMx\_MOD registers

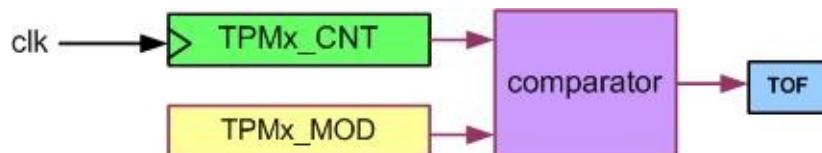


Figure 5-15: The TPM\_CNT and TPM\_MOD register and TOF flag

When TPMx\_CNT counter register is counting up, it is compared with the contents of this register. Whenever the contents of free-running TPMx\_CNT counter and TPMx\_MOD register are equal, the TOF flag (Timer Over Flow flag) goes up indicating there is a match and TPMx\_CNT rolls over to zero. See Figure 5-16. A smaller value of the TPMx\_MOD register leads the timer times out faster and the TOF flag sets sooner. In other words, delays can be made by setting the TPMx\_MOD register and monitoring the TOF flag.

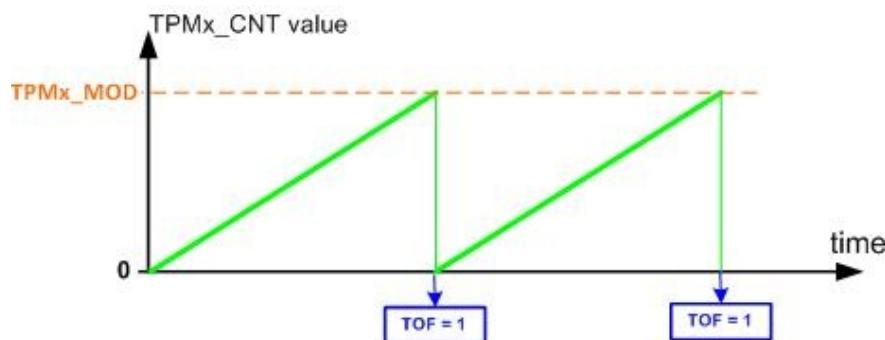


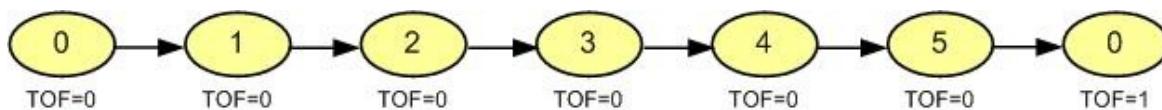
Figure 5-16: The role of TPMx\_MOD

### Example 5-4

Assume TPM0\_MOD = 5 and TPM0\_CNT is counting up. Explain when the TOF flag is raised.

#### Solution:

The timer counts up with the passing of each clock provided by the oscillator. As the timer counts up, it goes through the states of 0, 1, 2, 3, 4, and 5. Now since TPM0\_MOD=TPM0\_CNT match it raises the TOF flag.



The D7 bit of TPMx\_SC (TPMx Status Control) register belongs to the TOF flag, as we will see soon. Although the Timer Modulo is a 32-bit register, only the lower 16 bits are used. We can initialize this register with values ranging from 0x0000 to 0xFFFF. It must be noted that upon Reset TPMx\_MOD=0xFFFF. That means, if we do not initialize the TPMx\_MOD register, the TPMx\_CNT keeps counting up to 0xFFFF and rolls over to zero when it reaches 0xFFFF.

### Note

In Freescale ARM KL25Z microcontroller, all the timer registers begin with TPM. So, for simplicity, just consider the letters which come after TPM. For example, consider TPMx\_CNT as CNT (Counter). That means, TPM0\_CNT is Counter register for TPM0 and TPM0\_MOD is Modulus register for TPM0.

## TPMx Status Control (TPMx\_SC) register

Each of the TPMx has its own Status Control register. It is called TPMx\_SC in which x = 0, 1, or 2. During the initialization of the timers we must disable them. Modifying the configurations of a running timer may cause unpredictable results. We use D4:D3 (CMOD) bits of TPMx\_SC (TPM Status Control) register to disable or enable the Counter. This must be done in addition to allowing clock to the TPMx module using the SIM\_SCGC6 register and selecting the clock source for timer counter using SIM\_SOPT2 register. See Figures 5-17 and 5-18. Among other important bits of this register are TOF (Timer Over Flow flag), PS (Prescaler), and TOIE. The TOIE (Timer Overflow Interrupt Enable) is covered in Chapter 6 when we cover interrupts. Next, we examine the major bits of the TPMx\_SC register.

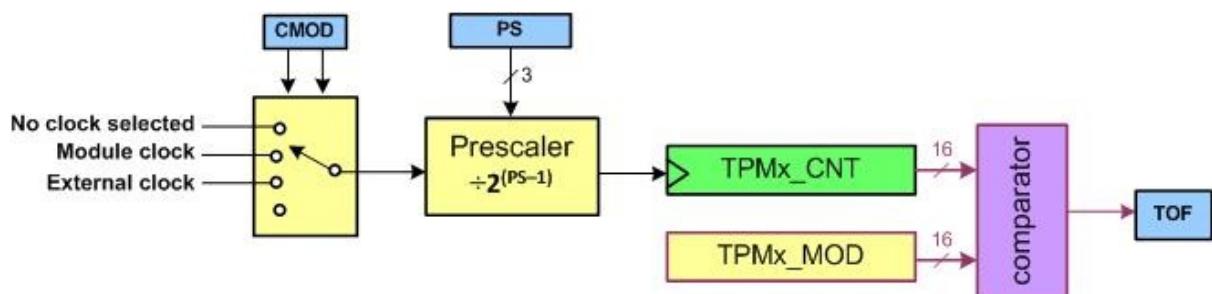


Figure 5-17: CMOD and PS (Prescaler) bits

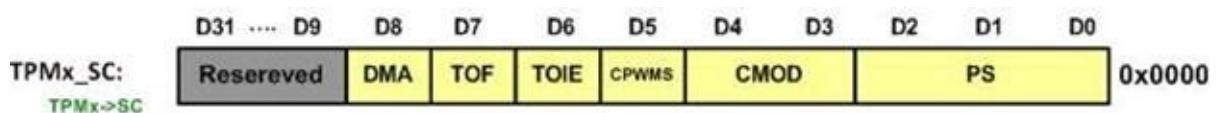


Figure 5-18: Timer Status and Control (TPMx\_SC) Register

Field	Bits	Description

In the prescaler, the clock is divided by $2^{PS}$ .											
<b>PS</b>	0–2	<b>PS value</b>	000	001	010	011	100	101	110	111	
		<b>Division</b>	1	2	4	8	16	32	64	128	
Clock Mode Selection											
<b>CMOD</b>	3–4	<b>CMOD value</b>	<b>Selected clock</b>								
		00	Timer stopped (No clock selected): In the mode, the TPM_CNT register receives no clock and it is stopped.								
		01	Timer mode (clock selected at SIM_SOPT2): This mode can be used to generate delays, periodic interrupts, or PWM.								
		10	Counter mode (clocked by LPTPM_EXTCLK pin): This mode is used to count an external event.								
<b>CPWMS</b>	5	11 Reserved									
		Center-aligned PWM select (0: Up counter mode, 1: up-down counter mode). For generating delays use the Up counter mode.									
<b>TOIE</b>	6	Time Overflow Interrupt Enable (0: Disabled, 1: Enabled). It is discussed in Chapter 6.									
<b>TOF</b>	7	Timer Overflow Flag									
<b>DMA</b>	8	DMA Enable (0: Disabled, 1: Enabled)									

Table 5-2: Timer Status and Control (TPMx\_SC) Register

### TOF flag bit

The TOF (Timer Overflow Flag) is bit D7 of the TPM\_SC register. When the CNT register counts up and matches the value in TPMx\_MOD, TOF is set to 1. We can monitor this flag and perform an action such as turning on a port bit. See Program 5-4. In order to clear the TOF bit for the next round we need to write a 1 to it. In other words, writing 0 to TOF has no effect. Indeed this rule applies to many flags of the KL25Z chip. We can also use the TOF in conjunction with TOIE to generate an interrupt. This will be covered in Chapter 6.

### Making delays using the TPM timer

The steps to program the timer for TPMx\_CNT are:

- 1) enable the clock to TPMx module in SIM\_SCAGC6,
- 2) select the clock source for timer counter in SIM\_SOPT2,
- 3) disable timer while the configuration is being modified,
- 4) set the mode as up-counter timer mode with TPMx\_SC register,
- 5) load TPMx\_MOD register with proper value,
- 6) clear TOF flag,
- 7) enable timer,
- 8) wait for TOF flag to go HIGH.

**Program 5-4: Toggle blue LED (PTD1 pin) every 320 times TPM0\_CNT matches the TPM0\_MOD.**

```
/* p5_4.c Toggling blue LED using TPM0 delay

This program uses TPM0 to generate maximal delay to toggle the blue LED.
MCGFLLCLK (41.94 MHz) is used as timer counter clock.

The Modulo register is set to 65,535. The timer counter overflows at
41.94 MHz / 65,536 = 640 Hz

We put the time out delay in a for loop and repeat it for 320 times before we
toggle the LED. This results in the LED flashing at half second on and half
second off.

The blue LED is connected to PTD1.

*/
#include <MKL25Z4.H>

int main (void) {
    int i;
    SIM->SCGC5 |= 0x1000;      /* enable clock to Port D */
    PORTD->PCR[1] = 0x100;     /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;        /* make PTD1 as output pin */
    SIM->SCGC6 |= 0x01000000;  /* enable clock to TPM0 */
    SIM->SOPT2 |= 0x01000000;  /* use MCGFLLCLK as timer counter clock */
    TPM0->SC = 0;              /* disable timer while configuring */
    TPM0->MOD = 0xFFFF;        /* max modulo value */
    TPM0->SC |= 0x80;          /* clear TOF */
    TPM0->SC |= 0x08;          /* enable timer free-running mode */
    while (1) {
        for(i = 0; i < 320; i++) { /* repeat timeout for 320 times */
            if(TPM0->SC == 0) { /* if timer overflowed */
                if(PTD->PDDR & 0x02) /* if PTD1 was high */
                    PTD->PDDR = 0; /* make PTD1 low */
                else
                    PTD->PDDR |= 0x02; /* make PTD1 high */
            }
        }
    }
}
```

```

while( (TPM0->SC & 0x80) == 0) { } /* wait until the TOF is set */
TPM0->SC |= 0x80; /* clear TOF */
}
PTD->PTOR = 0x02; /* toggle blue LED */
}
}

```

### Example 5-5

- (a) Show time delay calculation for Program 5-4,
- (b) Find the TPMx\_MOD value to make a delay of 142 ms.

#### Solution:

a)  $1 / 41.94\text{MHz} = 23.84\text{ns}$  since the FRDM board working clock is 41.94MHz.

$$23.84\text{ns} \times 65,536 = 1.56 \text{ msec}$$

The timer overflows every 1.56 msec. The delay contains 320 timer overflows in the for-loop:

$$1.56\text{msec} \times 320 = 0.5 \text{ second}$$

b)  $1 / 41.94\text{MHz} = 23.84\text{ns}$  since the FRDM board working clock is 41.94MHz.

$$142 \text{ ms} / 23.84\text{ns} = 5956. \text{ Thus } \text{TPMx\_MOD} = 5955$$

### *Prescaler options of timer*

The clock source of the timer counter is selected in SIM\_SOPT2 register. The prescaler sits between the clock source and the timer counter. It can be configured to divide the clock source by a number before feeding it to the timer counter. The lowest 3 bits of the TPMx\_SC register give the options of the number we can divide by. As shown in Figures 5-17 and 5-18, and Table 5-2, this number can be 1, 2, 4, 8, 16, 32, 64, and 128. Notice that the lowest factor is 1 and the highest factor is 128. That means at the lowest number 1, the clock source bypasses the prescaler and feed into the timer counter directly. Next, we will examine how the prescaler options are programmed.

#### Prescaler register for TPMx

Because TPM Modulo register has only 16 bits, the time interval is limited to

1.56 ms with 41.94 MHz clock as seen in Program 5-4. For longer delay, we will need to incorporate prescaler. The Program 5-5 below sets the prescalers to divide by 128 that will extend the period to 200 ms.

**Program 5-5: Toggle PTD1 pin on FRDM board every time register TPM0\_CNT matches the TPM0\_MOD register. Make TPM0\_MOD=65,535 and set prescaler to 128.**

```
/* p5_5.c Toggling blue LED using TPM0 delay (prescaler)

 * This program uses TPM0 to generate maximal delay to
 * toggle the blue LED.
 * MCGFLLCLK (41.94 MHz) is used as timer counter clock.
 * Prescaler is set to divided by 128 and the Modulo register
 * is set to 65,535. The timer counter overflows at
 * 41.94 MHz / 128 / 65,536 = 5.0 Hz
 *
 * The blue LED is connected to PTD1.
 */

#include <MKL25Z4.H>

int main (void) {
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[1] = 0x100;         /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;            /* make PTD1 as output pin */
    SIM->SCGC6 |= 0x01000000;     /* enable clock to TPM0 */
    SIM->SOPT2 |= 0x01000000;     /* use MCGFLLCLK as timer counter clock */
    TPM0->SC = 0;                 /* disable timer while configuring */
    TPM0->SC = 0x07;               /* prescaler /128 */
    TPM0->MOD = 0xFFFF;            /* max modulo value */
    TPM0->SC |= 0x80;              /* clear TOF */
    TPM0->SC |= 0x08;              /* enable timer free-running mode */
    while (1) {
        while((TPM0->SC & 0x80) == 0) { } /* wait until the TOF is set */
        TPM0->SC |= 0x80;                /* clear TOF */
        PTD->PTOR = 0x02;                /* toggle blue LED */
    }
}
```

## Example 5-6

- (a) Show time delay calculation for Program 5-5,
- (b) calculate the largest delay size without prescaler
- (c) Find the TPMx\_MOD value to generate a delay of 0.1 second. Use the prescaler of 128.

### Solution:

(a)  $41.94 \text{ MHz} / 128 = 327,656 \text{ Hz}$  with prescaler of 128.

$$1 / 327,656 \text{ Hz} = 3.05 \mu\text{sec}$$

$$3.05 \mu\text{sec} \times 65,535 = 200 \text{ ms}$$

(b)  $41.94 \text{ MHz} / 1 = 41.94 \text{ MHz}$  with no prescaler.

$$1 / 41.94 \text{ MHz} = 23.84 \text{ ns.}$$

The largest possible delay is  $\text{TPMx\_MOD}=65,535=0xFFFF$ .

$$\text{Now, } 65,536 \times 23.84 \text{ ns} = 1,562,613 \text{ ns} = 1.56 \text{ ms} = 0.00156 \text{ sec.}$$

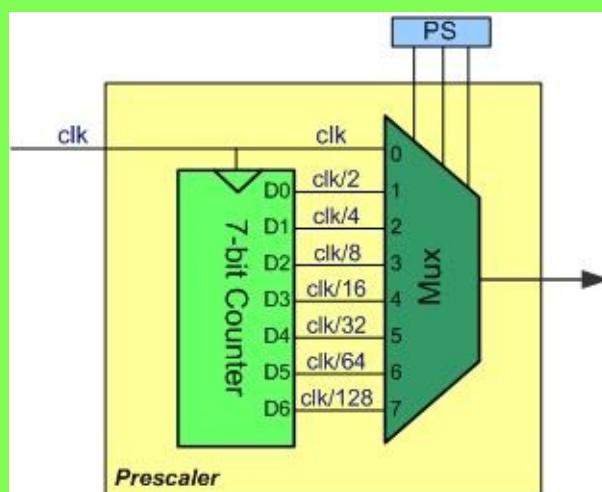
(c)  $41.94 \text{ MHz} / 128 = 327,656 \text{ Hz}$  with prescaler of 128.

$$1 / 327,656 \text{ Hz} = 3.05 \mu\text{sec}$$

$$0.1 \text{ sec} / 3.05 \mu\text{sec} = 32766. \text{ TPMx\_MOD is } 32,766 - 1 = 32,765.$$

### Prescaler internal circuit (Case study)

The prescaler is made of a 7-bit counter and a multiplexer, as shown in the following figure.



## Using TPM1 and TPM2

Program 5-6 shows the Timer1 version of Program 5-5. The only changes required are enabling the clock in SIM->SCGC6 and replacing all the references to TPM0 by TPM1.

Program 5-6: Delay using Timer1

```
/* p5_6.c Toggling blue LED using TPM1 delay

 * This program uses TPM1 to generate maximal delay to
 * toggle the blue LED. It is the same as p5_5 except
 * a different timer is used.

 *
 * MCGFLLCLK (41.94 MHz) is used as timer counter clock.
 * Prescaler is set to divided by 128 and the Modulo register
 * is set to 65,535. The timer counter overflows at
 * 41.94 MHz / 128 / 65,536 = 5.0 Hz
 *
 * The blue LED is connected to PTD1.
 */

#include <MKL25Z4.H>

int main (void) {
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[1] = 0x100;         /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;            /* make PTD1 as output pin */
    SIM->SCGC6 |= 0x02000000;     /* enable clock to TPM1 */
    SIM->SOPT2 |= 0x01000000;     /* use MCGFLLCLK as timer counter clock */
    TPM1->SC = 0;                 /* disable timer while configuring */
    TPM1->SC = 0x07;               /* prescaler /128 */
    TPM1->MOD = 0xFFFF;            /* max modulo value */
    TPM1->SC |= 0x80;              /* clear TOF */
    TPM1->SC |= 0x08;              /* enable timer free-running mode */
    while (1) {
        while ((TPM1->SC & 0x80) == 0) { } /* wait until the TOF is set */
        TPM1->SC |= 0x80;                /* clear TOF */
        PTD->PTOR = 0x02;                /* toggle blue LED */
    }
}
```

}
}

TPM0 and TPM1 run 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 TPM0, it is difficult to monitor TPM1 at the same time. One solution to that is to use timer interrupt, which is discussed in the next chapter.

## TPMx Registers and their addresses

Table below shows the addresses of major registers for TPM0, TPM1, and TPM2 modules.

Absolute address	Register Name
<b>4003 8000</b>	Status and Control (TPM0_SC)
<b>4003 8004</b>	Counter (TPM0_CNT)
<b>4003 8008</b>	Modulo (TPM0_MOD)
<b>4003 800C</b>	Channel 0 Status and Control (TPM0_C0SC)
<b>4003 8010</b>	Channel 0 Value (TPM0_C0V)
<b>4003 8014</b>	Channel 1 Status and Control (TPM0_C1SC)
<b>4003 8018</b>	Channel 1 Value (TPM0_C1V)
<b>4003 801C</b>	Channel 2 Status and Control (TPM0_C2SC)
<b>4003 8020</b>	Channel 2 Value (TPM0_C2V)
<b>4003 8024</b>	Channel 3 Status and Control (TPM0_C3SC)
<b>4003 8028</b>	Channel 3 Value (TPM0_C3V)
<b>4003 802C</b>	Channel 4 Status and Control (TPM0_C4SC)
<b>4003 8030</b>	Channel 4 Value (TPM0_C4V)
<b>4003 8034</b>	Channel 5 Status and Control (TPM0_C5SC)
<b>4003 8038</b>	Channel 5 Value (TPM0_C5V)
<b>4003 8050</b>	Capture and Compare Status (TPM0_STATUS)
<b>4003 9000</b>	Status and Control (TPM1_SC)

<b>4003 9004</b>	Counter (TPM1_CNT)
<b>4003 9008</b>	Modulo (TPM1_MOD)
<b>4003 900C</b>	Channel 0 Status and Control (TPM1_C0SC)
<b>4003 9010</b>	Channel 0 Value (TPM10_C0V)
<b>4003 9014</b>	Channel 1 Status and Control (TPM1_C1SC)
<b>4003 9018</b>	Channel 1 Value (TPM1_C1V)
<b>4003 901C</b>	Channel 2 Status and Control (TPM1_C2SC)
<b>4003 9020</b>	Channel 2 Value (TPM1_C2V)
<b>4003 9024</b>	Channel 3 Status and Control (TPM1_C3SC)
<b>4003 9028</b>	Channel 3 Value (TPM1_C3V)
<b>4003 902C</b>	Channel 4 Status and Control (TPM1_C4SC)
<b>4003 9030</b>	Channel 4 Value (TPM1_C4V)
<b>4003 9034</b>	Channel 5 Status and Control (TPM1_C5SC)
<b>4003 9038</b>	Channel 5 Value (TPM1_C5V)
<b>4003 9050</b>	Capture and Compare Status (TPM1_STATUS)
<b>4003 A000</b>	Status and Control (TPM2_SC)
<b>4003 A004</b>	Counter (TPM2_CNT)
<b>4003 A008</b>	Modulo (TPM2_MOD)
<b>4003 A00C</b>	Channel 0 Status and Control (TPM2_C0SC)
<b>4003 A010</b>	Channel 0 Value (TPM2_C0V)
<b>4003 A014</b>	Channel 1 Status and Control (TPM2_C1SC)
<b>4003 A018</b>	Channel 1 Value (TPM2_C1V)
<b>4003 A01C</b>	Channel 2 Status and Control (TPM2_C2SC)
<b>4003 A020</b>	Channel 2 Value (TPM2_C2V)
<b>4003 A024</b>	Channel 3 Status and Control (TPM2_C3SC)
<b>4003 A028</b>	Channel 3 Value (TPM2_C3V)
<b>4003 A02C</b>	Channel 4 Status and Control (TPM2_C4SC)

<b>4003 A030</b>	Channel 4 Value (TPM2_C4V)
<b>4003 A034</b>	Channel 5 Status and Control (TPM2_C5SC)
<b>4003 A038</b>	Channel 5 Value (TPM2_C5V)
<b>4003 A050</b>	Capture and Compare Status (TPM2_STATUS)

**Table 5-3: TPM Registers and their addresses**

## Longer Timer Interval

As shown in Program 5-5 and Program 5-6, with 41.94 MHz system clock the longest time interval we could get was 200 ms. To achieve a longer time interval, we may repeat the short time interval multiple times as we have done in Program 5-4. An alternative is to drive the timer with a slower clock. The benefit of slow clock is that the circuit consumes much less power when it is switching fewer times. This is important if it is used in mobile device when battery charge is precious.

The Freescale ARM KL25Z has an internal reference clock at 32.768 kHz that may be used as the clock source for the timers. Recall the timer clock source selection is made in SIM->SOPT2 register. Program 5-7 uses the 32.768 kHz to generate 5 second timeout interval. The longest timeout interval from the timers with the 32.768 kHz clock is  $32.768 \text{ kHz} / 128 / 65536 = 0.0039 \text{ Hz}$  or 256 second.

### Program 5-7: Toggling blue LED every five seconds

```
/*
 * p5_7.c Toggling blue LED using TPM0 delay
 *
 * This program uses TPM0 to generate long delay to
 * toggle the blue LED.
 *
 * MCGIRCLK (32.768 kHz) is used as timer counter clock.
 *
 * Prescaler is set to divided by 4 and the Modulo register
 * is set to 40,959. The timer counter overflows at
 * 32,768 Hz / 40,960 / 4 = 0.2 Hz
 *
 * The blue LED is connected to PTD1.
 */
#include <MKL25Z4.H>
int main (void) {
    SIM->SCGC5 |= 0x1000;      /* enable clock to Port D */
}
```

```

PORTD->PCR[1] = 0x100;           /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02;              /* make PTD1 as output pin */
SIM->SCGC6 |= 0x01000000;       /* enable clock to TPM0 */
SIM->SOPT2 |= 0x03000000;       /* use MCGIRCLK as timer counter clock */
TPM0->SC = 0;                  /* disable timer while configuring */
TPM0->SC = 0x02;                /* prescaler /4 */
TPM0->MOD = 40960 - 1;          /* modulo value */
TPM0->SC |= 0x80;               /* clear TOF */
TPM0->SC |= 0x08;               /* enable timer free-running mode */
while (1) {
    while((TPM0->SC & 0x80) == 0) { } /* wait until the TOF is set */
    TPM0->SC |= 0x80;               /* clear TOF */
    PTD->PTOR = 0x02;               /* toggle blue LED */
}
}

```

For even longer timeout interval, the Freescale ARM KL25Z has a low power timer (LPTMR) that may use the 32.768 kHz clock or a 1 kHz clock. The LPTMR has a prescaler that will divide up to 65,536 and a 16-bit counter that will count up to 65,536. The longest timeout interval for LPTMR is 4,294,967 seconds or about 50 days. We will leave the programming of the LPTMR to the readers.

## Review Questions

1. True or false. We can use only 16 bits of the TPMx\_CNT even though the register is 32-bit.
2. We have \_\_\_\_\_ Timer Module in KL25Z and they are designated as \_\_\_\_\_.
3. True or false. Each of the TPM0, TPM1, and TPM2 has its own TPMx\_CNT.
4. Which register is used to enable the clock to the TPM0?
5. Which register is used to select the clock source for timer counters?
6. In Freescale ARM KL25Z, each Timer has \_\_\_\_\_ channels.



## Section 5.3: Output Compare and TPM Channels

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. In this section, we will study the Output Compare feature of the KL25Z Timers. We examine the channels of TPMs, as well.

### Programming Output Compare option

In some applications we need to control the digital pin output transition with precision timing. To do that, we use the Output Compare function of the timer. In the KL25Z, each of the TPMx module has 6 channels for the Output Compare function. See Figure 5-19.

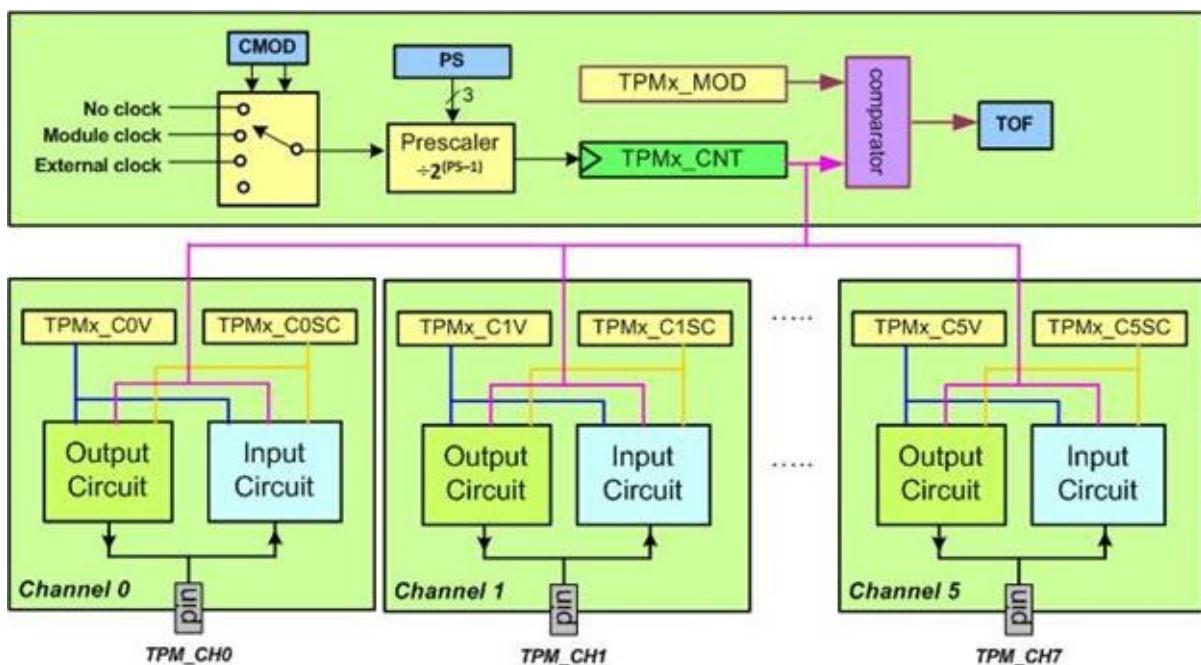


Figure 5-19: The Channels of TPMx

Each channel has its own 16-bit register for the compare purpose. The registers are called TPMx Channel Value (TPMx\_CnV) and are designated as TPMxC0V to TPMxC5V. See Figure 5-20.

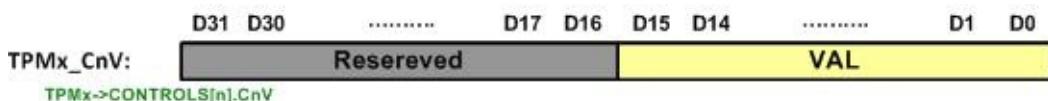


Figure 5-20: TPMx\_CnV (TPMx Channel Value) Register

The 16-bit registers of TPMx\_CnV are readable and writable, which means we can initialize them to a desired value. After the initialization, the TPMx\_CnV content is compared with the value in TPMx\_CNT after each clock cycle as TPM\_CNT is counting up. When the value of the CNT register and CnV register match, the CHF flag is set high. See Figure 5-21. It can also perform some actions such as toggling a pin, making a pin to go Low or High. We choose one of these options using the Channel Status and Control Register (TPMx\_CnSC), which is discussed next.

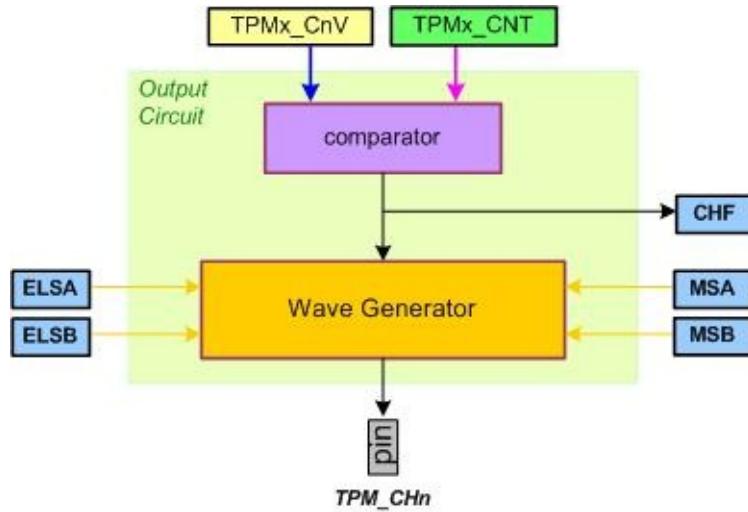


Figure 5-21: Output Circuit

## ***Channel Status and Control Selection (TPMx\_CnSC) register***

As we just stated, each TPM module has six channels. There are two registers associated with each channel. They are the Channel Value register (TPMxCnV) and Channel Status and Control register (TPMxCnSC). Notice the x can be 0, 1, or 2 for Timer modules of 0, 1, and 2. The n can be 0 to 5 for one of the six channels inside each Timer module. Now, the mode and edge selections for Output Compare of a given channel are done with TPMx Channel Status and Control (TPMxCnSC). Bits D5:D4 is used to choose the Output Compare option of the timer. See Figure 5-22 and Table 5-4.



Figure 5-22: TPMxCnSC (TPMx Channel Status and Control)

Field	Bit	Description
<b>CHF</b>	7	Channel Flag
<b>CHIE</b>	6	Channel interrupt enable
<b>MSB and MSA</b>	5-4	Channel mode select
		<b>D5:D4 (MSB:MSA)      Output mode</b>
		<b>00</b> Channel disabled
		<b>01</b> Output compare
		<b>10</b> PWM
		<b>11</b> Output compare

<b>ELSB and ELSA</b>	3-2	Edge or Level Select
<b>DMA</b>	0	DMA enable (0: Disabled, 1: Enabled)

**Table 5-4: TPMxCnSC Register**

After selecting the Output Compare with D5:D4=01, we use the D3:D2 bits to choose the following action for a given channel:

D5:D4 (MSB:MSA)	D3 (ELSB)	D2 (ELSA)	Output Action
01	0	1	Toggle Output on Match
01	1	0	Clear Output on Match (make it Low)
01	1	1	Set Output on Match (make output High)
11	1	0	Pulse Output Low on Match
11	X	1	Pulse Output High on Match

**Table 5-5: Mode Selection for Output Compare**

### **Output Compare mode**

If a timer channel is in the output compare mode, when the timer is counting up, the TPMx\_CNT counter begins counting from 0 and goes up until it reaches the TPMx\_CnV value. Then, the Channel Flag (CHF) for that channel is set and the channel output is changed. The timer continues counting until it reaches to the TPMx\_MOD value and rolls over. Figures 5-23 through 5-25 show the output pin in toggle, set, and clear modes.

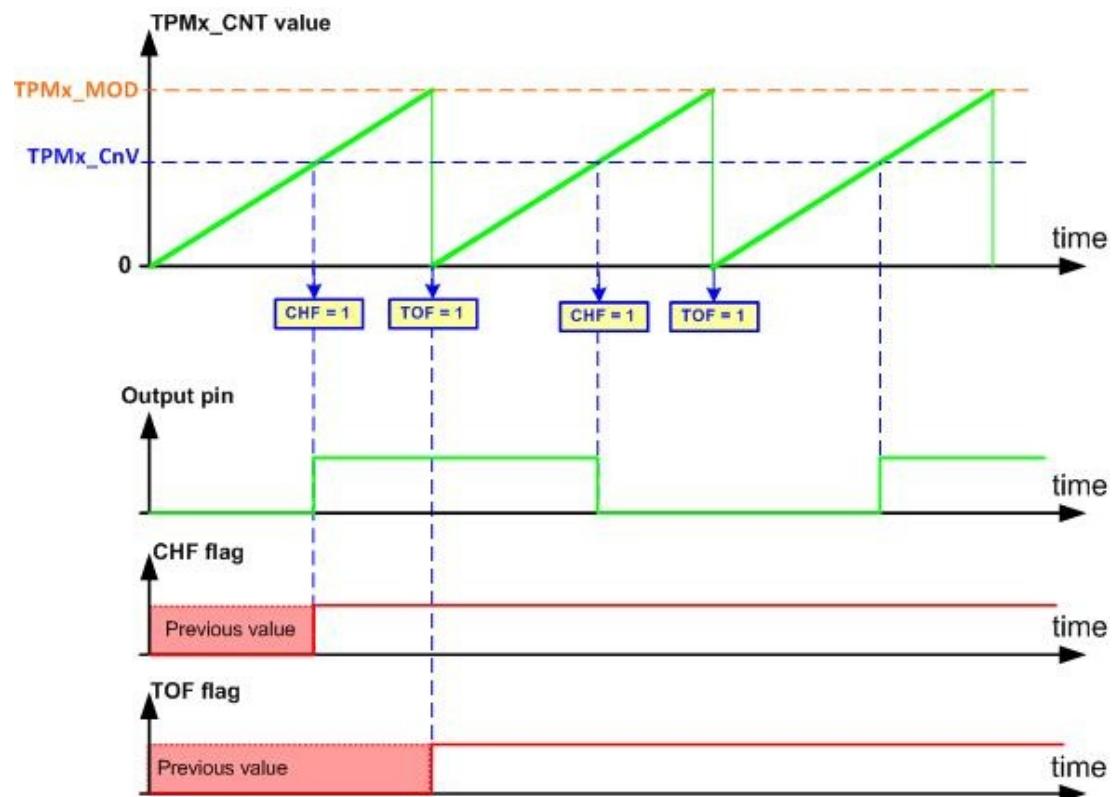


Figure 5-23: In Toggle Mode

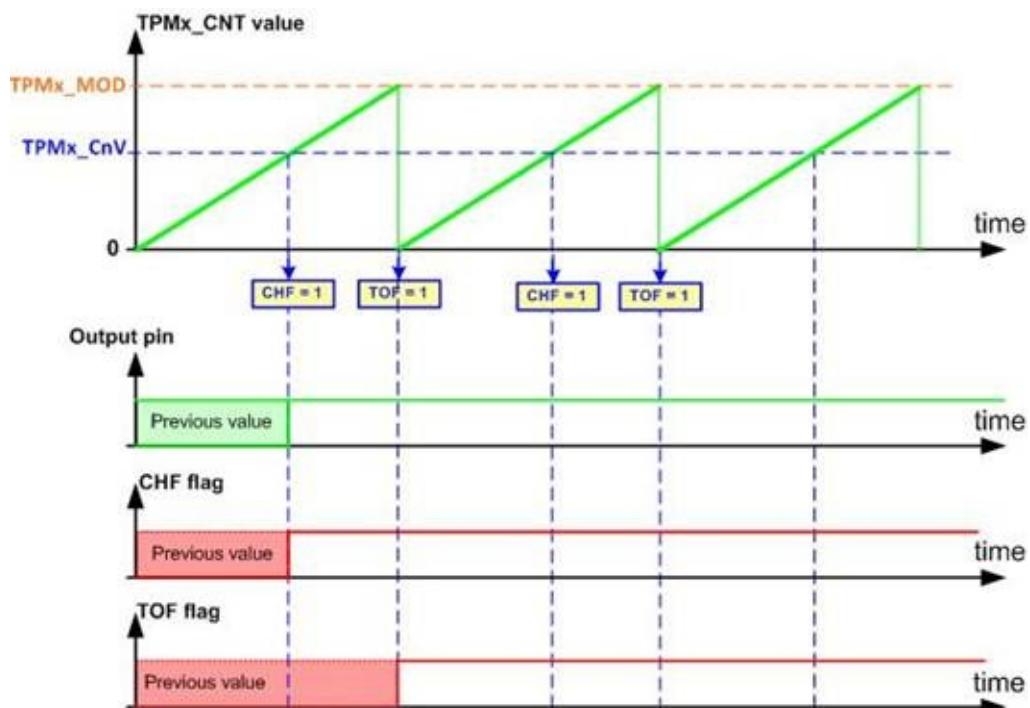


Figure 5-24: In Set Mode

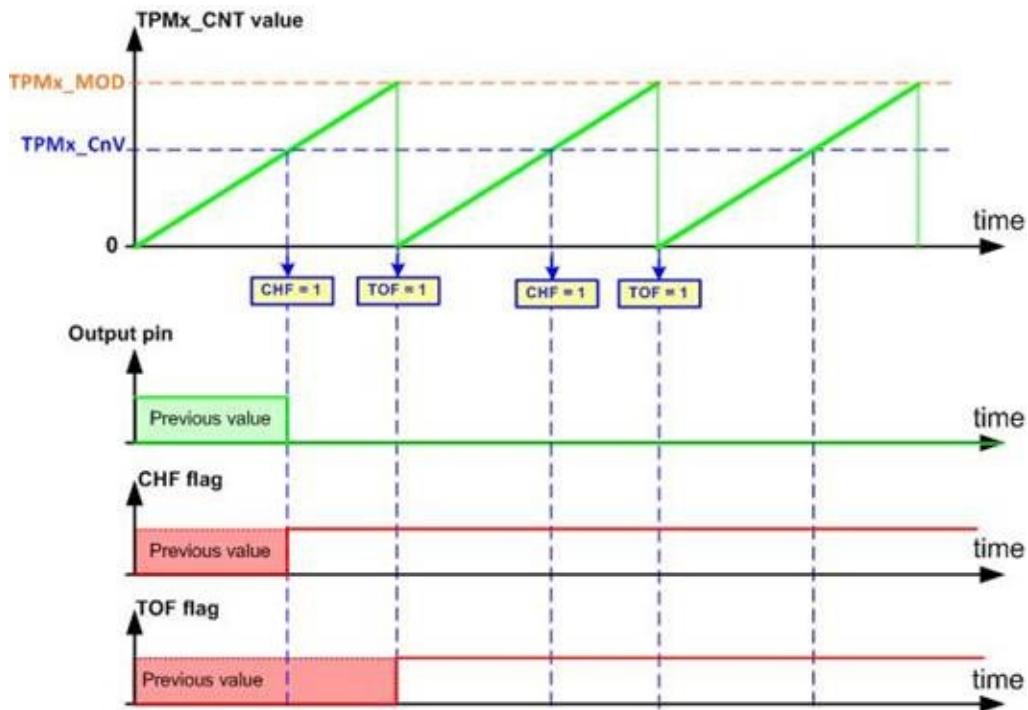


Figure 5-25: Clear Mode

Upon reset, all bits of the TPMxCnV register are initialized to 0s. But the software can change the value of TPMx\_CnV.

## Channel pins

There are CH0 to CH5 for each TPMx. Each channel has its own designated output pins. For example, Channel 0 of TPM0 may be connected to PTA3, PTC1, PTD0, or PTE24 for output and Channel 1 of TPM0 may use PTA4, PTC2, PTD1 or PTE25 as output pin. The choices of the output pin designations are made in the alternate function of the pin control register (PORTx\_PCR) of each port. It is possible to have multiple output pins connected to a single channel at the same time. Table 5-5 shows the pin designations for all Timer modules.

## Selecting alternate function for Timers pin

Upon Reset, the PORTx\_PCRn register has all 0s meaning the I/O pins are not defined yet. To use an alternate function, we first must configure the bits in the PORTx\_PCRn register for that pin. As we mentioned in previous chapters, each pin has its own PORTx\_PCRn register. For example, for the PTB18 to be used by TPM2\_CH1, we need to write 0x0300 (0000 0011 0000 0000 in binary) to PORTB\_PCR18 register. See Tables 5-5. To do that, we need to use the information in Section 10.3.1 of Freescale KL25Z Ref. Manual. Table 5-6 provides the summary for the Timers pins.

### Using ALT3 pin options for TPM0 Channel Output

TPM0 Channels	Pins	Pin Control Register
TPM0 CH0 Output		

<b>Pins</b>	PTA3	PORTA_PCR3=0x0300
	PTE24	PORTE_PCR24=0x0300
<b>TPM0 CH1 Output Pins</b>	PTA4	PORTA_PCR4=0x0300
	PTE25	PORTE_PCR25=0x0300
<b>TPM0 CH2 Output Pins</b>	PTA5	PORTA_PCR5=0x0300
	PTE29	PORTE_PCR29=0x0300
<b>TPM0 CH3 Output Pins</b>	PTE30	PORTE_PCR30=0x0300
<b>TPM0 CH4 Output Pins</b>	PTC8	PORTC_PCR8=0x0300
	PTE31	PORTE_PCR31=0x0300
<b>TPM0 CH5 Output Pins</b>	PTA0	PORTA_PCR0=0x0300
	PTC9	PORTC_PCR9=0x0300

### Using ALT4 pin options for TPM0 Channel Output

<b>TPM0 CH0 Output Pins</b>	PTC1	PORTC_PCR1=0x0400
	PTD0	PORTD_PCR0=0x0400
<b>TPM0 CH1 Output Pins</b>	PTC2	PORTC_PCR2=0x0400
	PTD1	PORTD_PCR1=0x0400
<b>TPM0 CH2 Output Pins</b>	PTC3	PORTC_PCR3=0x0400
	PTD2	PORTD_PCR2=0x0400
<b>TPM0 CH3 Output Pins</b>	PTC4	PORTC_PCR4=0x0400
	PTD3	PORTD_PCR3=0x0400
<b>TPM0 CH4 Output</b>	PTD4	PORTD_PCR4=0x0400

Pins		
TPM0 CH5 Output Pins	Pins	Pin Control Register
<b>Using ALT3 pin options for TPM1 Channel Output</b>		
TPM1 Channels	Pins	Pin Control Register
TPM1 CH0 Output Pins	PTA12	PORTA_PCR12=0x0300
	PTB0	PORTB_PCR0=0x0300
	PTE20	PORTE_PCR20=0x0300
TPM1 CH1 Output Pins	PTA13	PORTA_PCR13=0x0300
	PTB1	PORTB_PCR1=0x0300
	PTE21	PORTE_PCR21=0x0300
<b>Using ALT3 pin options for TPM2 Channel Output</b>		
TPM2 Channels	Pins	Pin Control Register
TPM2 CH0 Output Pins	PTA1	PORTA_PCR1=0x0300
	PTB18	PORTB_PCR18=0x0300
	PTE22	PORTE_PCR22=0x0300
TPM2 CH1 Output Pins	PTA2	PORTA_PCR2=0x0300
	PTB3	PORTB_PCR3=0x0300
	PTB19	PORTB_PCR19=0x0300
	PTE23	PORTE_PCR23=0x0300

Table 5-6: Timers Channel Output alternate pin assignment

See Example 5-7.

### Example 5-7

Write the code to provide the TPM0\_CH1 function on PTD1.

**Solution:**

PTD1 uses the ALT4, as shown in Table 5-6. The I/O pin is used for an alternate peripheral function pin by setting the PORTx\_PCRn register of PORTx:

`PORTD->PCR[1] |= 0x0400;`

---

### ***Toggling a pin using output compare***

The steps to program the timer for Output Compare are:

enable the clock to the output pin GPIO port,

select the alternate function for the output pin,

enable the clock to TPMx module,

Select the clock source for timer counter,

disable timer while the configuration is being modified,

select prescaler value with TPMx\_SC register,

set modulo value in TPMx\_MOD register, set the CnSC register to toggle mode  
(MSA = 1, MSB = 0, ELSA = 1, ELSB = 0),

clear CHFn (channel n flag) flag,

set TPMx\_CnV register based on its current value with the interval count added,

enable timer

See Program 5-8. This program uses output compare mode to toggle the PTD1 pin, which is connected to the blue LED on the FRDM board. Every time there is match between TPM0\_CNT and TPM0\_C1V registers, the CHF bit is set in the TPM0\_C1SC register and the output is toggled. The program reads the TPM0\_C1V value and adds 32766 to it that scheduled the next match to be 32,766 clock cycles later. The timer counter clock is running at 41.94 MHz and the prescaler is set to divide the clock source by 128 so the timer counter is counting at  $41.94 \text{ MHz} / 128 = 367,656 \text{ Hz}$  and the period is  $3.05 \mu\text{s}$ . To schedule next output compare match for 32,766 clock cycles results in

$$3.05 \mu\text{s} \times 32,766 = 0.1 \text{ sec.}$$

**Program 5-8: Toggling blue LED using Timer 0 Channel 1 in Output Compare mode**

```

/* p5_8.c using TPM0 Output Compare

 * This program uses TPM0 CH1 OC to generate periodic output.
 * MCGFLLCLK (41.94 MHz) is used as timer counter clock.
 * Prescaler is set to divided by 128.
 * Timer0 Channel 1 is configured as Output Compare Toggle mode
 * and the output is on PTD1 (blue LED). Every time there is a
 * match between TPM0_CNT and TPM0_C1V, the output is toggled and
 * the value in TPM0_C1V is incremented by 2097 that schedules
 * the next match to be in (1 / 41.94 MHz) * 128 * 32766 = 100 ms.
 * The output will toggle every 100 ms or 5 Hz.
 */

#include <MKL25Z4.H>

int main (void) {
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[1] = 0x400;         /* set PTD1 pin for TPM0CH1 */
    SIM->SCGC6 |= 0x01000000;      /* enable clock to TPM0 */
    SIM->SOPT2 |= 0x01000000;      /* use MCGFLLCLK as timer counter clock */
    TPM0->SC = 0;                 /* disable timer while configuring */
    TPM0->SC = 0x07;              /* prescaler /128 */
    TPM0->MOD = 0xFFFF;            /* max modulo value */
    TPM0->CONTROLS[1].CnSC = 0x14; /* OC toggle mode */
    TPM0->CONTROLS[1].CnSC |= 0x80; /* clear CHF */
    TPM0->CONTROLS[1].CnV = TPM0->CNT + 32766; /* schedule next transition */
    TPM0->SC |= 0x08;              /* enable timer */

    while (1) {
        while (!(TPM0->CONTROLS[1].CnSC & 0x80)) { } /* wait until the CHF is set */
        TPM0->CONTROLS[1].CnSC |= 0x80;                  /* clear CHF */
        TPM0->CONTROLS[1].CnV = TPM0->CNT + 32766; /* schedule next transition */
    }
}

```

## **Channel Status for each timer module (**TPM<sub>x</sub>\_STATUS**) register**

As we just stated that, each TPM module has six channels. There are two

registers associated with each channel. They are the Channel Value register (TPMxCnV) and Channel Status and Control register (TPMxCnSC). However, we also have a single status register for all the channels. This is called TPMx\_STATUS. This allows us to monitor the status of all the 6 channels with a single read of the register to see whether any given CHF flag has been raised. Notice from Figure 5-26, that we have D5 bit for Channel 5 flag and D0 bit for Channel 0 flag. Also notice that, in addition to the CHF (Channel flag) for each channel, we also have the TOF belonging to the TPMx\_CNT and TPMx\_MOD all of them in one register. The D8 bit of the TPMx\_STATUS register is for TOF.

D31	.....	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
TPMx_STATUS:	Reserved		TOF		0	CH5F	CH4F	CH3F	CH2F	CH1F	CH0F
TPMx->STATUS											

Figure 5-26: TPMx\_STATUS Register

## Review Questions

1. True or false. Each Channel has its own designated pins.
2. True or false. Upon Reset, all the pins are designated as simple I/O.
3. True or false. We have a single register for selection of the alternate function for all the I/O ports.
4. True or false. Each pin has its own PCRn register.



## 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 TPMx\_CNT timer counter is captured and saved in a register while the counter keeps counting.

To configure TPM as Input Capture mode, bits MSnB:MSnA of the TPMx\_CnSC should be 00 (binary). We use ELSnB:ELSnA bits to choose the rising or falling edge. See Figure 5-22. In this mode, the counter value is stored in the Channel register (TPMx\_CnV) whenever the input pin is triggered by an external event (falling or rising edge-triggered) fed to the TPM\_CHn pin. See Figure 5-27.

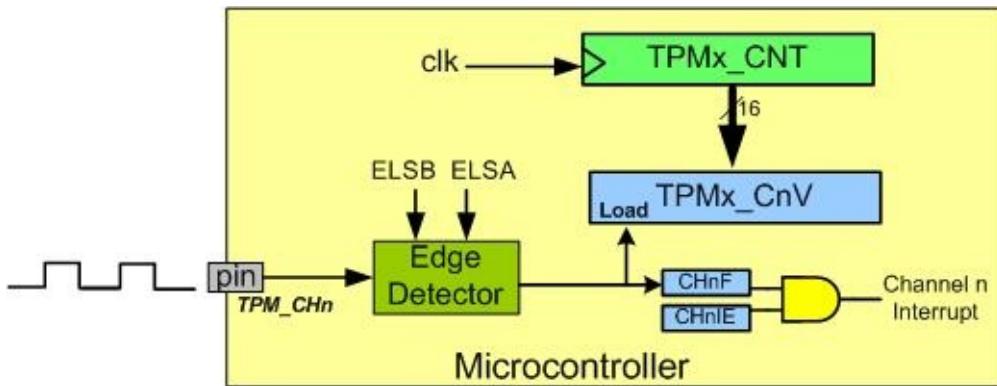


Figure 5-27: Input Edge Time Capturing

Notice that the Channel can be configured to capture on the falling edge, rising edge, or both. To determine the type of edge that is captured, the ELSn:BLELSnA bits of the TPMx\_CnSC register should be initialized. See Table 5-7. Also notice that capturing has no effect on counting and the timer counter continues counting when the capture event takes place.

ELSB	ELSA	Capture mode
0	0	Channel disabled
0	1	Capture on rising edge
1	0	Capture on falling edge
1	1	Capture on both edges

Table 5-7: Choosing the Capture Edge

### Pin Selection for Input Capture

To measure the edge time we must feed the pulse into the TPM\_CHn pin. The input capture timer channel-pin designation is identical to the output compare

timer channel-pin designation in Table 5-6. So we will not repeat it here.

On the FRDM-KL25Z board, the power supply is regulated at 3.3V when it is powered by the OpenSDA USB cable (J7). Input signal shall not exceed 3.6V otherwise damage to the pin or device may happen. For an input signal to be recognized by the input pin, a high signal should be above 2.31V and below 3.6V, a low signal should be below 1.15V and above 0V. For more details, check out the KL25 datasheet.

## **Input edge-time mode usages**

The input edge time capturing can be used for many applications; e.g. recording the arrival time of an event, measuring the frequency and pulse width of a signal.

### ***Steps to program the Input Capture function***

Perform the following steps to measure the period of a periodic waveform based on the edge arrival time of the Input Capture function.

- 1) Enable the clock to the input pin GPIO port,
- 2) select the alternate function for the input pin at the PORTX\_PCR register,
- 3) enable the clock to TPMx module,
- 4) select the clock source for timer counter,
- 5) disable timer while the configuration is being modified,
- 6) select prescaler value with TPMx\_SC register,
- 7) set modulo value in TPMx\_MOD register,
- 8) set the CnSC register to capture rising edge,
- 9) enable timer,
- 10) wait until the CHF bit is set in CnSC register,
- 11) read the current counter value captured,
- 12) calculate the current counter value difference from the last value,
- 13) save the current value for next calculation,
- 14) repeat from step 10.

As shown in Figure 5-28, to measure the period of a signal we must measure the time between two falling edges or two rising edges. Program 5-9 measures the period of the square wave.

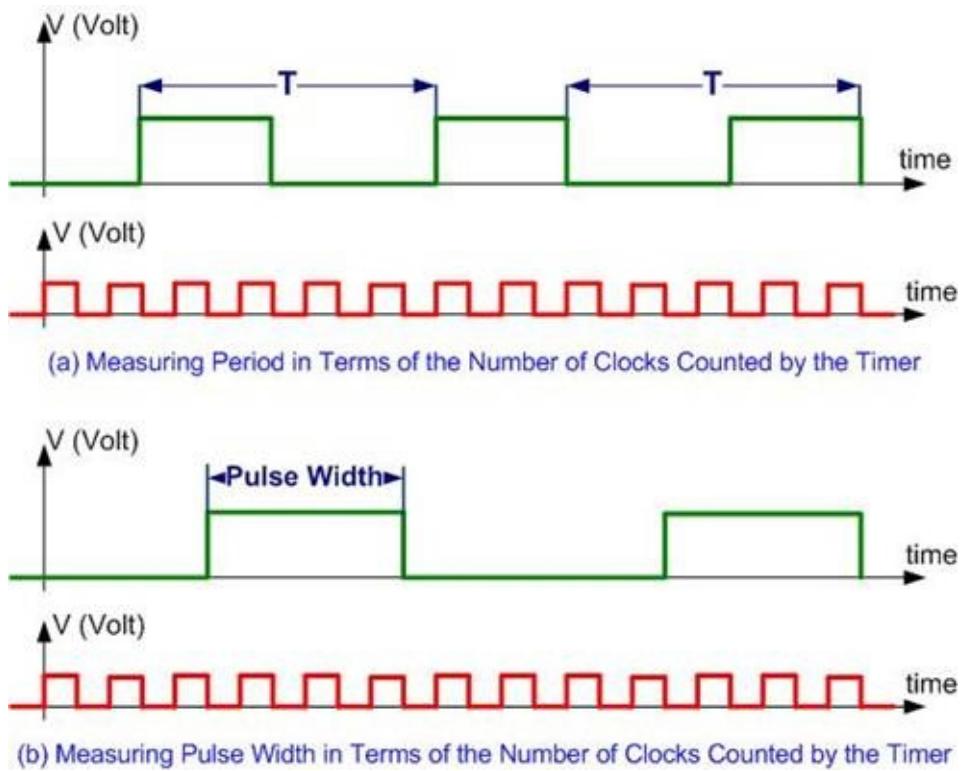


Figure 5-28: Measuring Period and Pulse Width

### Program 5-9: Functions to initialize Channel 0 of Timer0 for edge-time capture mode to measure the period of a square wave input signal

```
/*
 * p5_9.c Using TPM2 Channel 0 to measure input period.

 * This program uses TPM2 CH1 Input Edge-time Capture to measure
 * the period of a periodic waveform.
 * MCGFLLCLK (41.94 MHz) is used as timer counter clock.
 * Prescaler is set to divided by 128. So the timer counter is
 * counting at 41.94 MHz / 128 = 327,656 Hz.
 * Timer 2 Channel 0 is configured as Input Edge-time Capture mode.
 * and the input is using PTA1.
 * When a rising edge occurs at PTA1, the timer counter value is
 * copied to TPM2_COV and the CHF is set.
 * The program waits for CHF flag to set then calculates the
 * difference of the current value to the previous recorded value.
 * Bit 11-9 are used to control the tri-color LEDs.
 * The LED should change color when the input frequency is changing
 * below 642 Hz. Above 642 Hz, the number of clock cycles between
 * rising edges is too small to reach bit 9.
 */

#include <MKL25Z4.H>
```

```

int main (void) {
    unsigned short then = 0;
    unsigned short now = 0;
    unsigned short diff;

    /* Initialize GPIO pins for tri-color LEDs */
    SIM->SCGC5 |= 0x400;           /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTB->PCR[18] = 0x100;        /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x40000;          /* make PTB18 as output pin */
    PORTB->PCR[19] = 0x100;        /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0x80000;          /* make PTB19 as output pin */
    PORTD->PCR[1] = 0x100;         /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;            /* make PTD1 as output pin */

    /* end GPIO pin initialization for LEDs */

    /* Start of Timer code */
    SIM->SCGC5 |= 0x0200;          /* enable clock to Port A */
    PORTA->PCR[1] = 0x0300;         /* set PTA1 pin for TPM2CH0 */
    SIM->SCGC6 |= 0x04000000;       /* enable clock to TPM2 */
    SIM->SOPT2 |= 0x01000000;       /* use MCGFLLCLK as timer counter clock */
    TPM2->SC = 0;                 /* disable timer while configuring */
    TPM2->SC = 0x07;               /* prescaler /128 */
    TPM2->MOD = 0xFFFF;             /* max modulo value */
    TPM2->CONTROLS[0].CnSC = 0x04; /* IC rising edge */
    TPM2->SC |= 0x08;              /* enable timer */

    while (1) {

        while (!(TPM2->CONTROLS[0].CnSC & 0x80)) { } /* wait until the CHF is set */
        TPM2->CONTROLS[0].CnSC |= 0x80;                  /* clear CHF */
        now = TPM2->CONTROLS[0].CnV;

        diff = now - then; /* you may put a breakpoint here and examine the values */
        then = now;          /* save the current counter value for next calculation */
    }

    /* change LEDs according to bit 11-9 of the value of diff */
    diff = diff >> 9;

    if (diff & 1)      /* use bit 0 of diff to control red LED */
        PTB->PCOR = 0x40000;    /* turn on red LED */
    else
        PTB->PSOR = 0x40000;    /* turn off red LED */
    if (diff & 2)      /* use bit 1 of diff to control green LED */
        PTB->PCOR = 0x80000;    /* turn on green LED */
    else

```

```
PTB->PSOR = 0x80000; /* turn off green LED */  
if (diff & 4) /* use bit 2 of diff to control blue LED */  
PTD->PCOR = 0x02; /* turn on blue LED */  
else  
PTD->PSOR = 0x02; /* turn off blue LED */  
/* end of LED code */  
}  
}
```

## Review Questions

1. True or false. To capture the input edge time, the TPM\_CnSC register must be configured for Input Capture mode.
2. True or false. To measure the frequency of a signal, the time interval between a falling edge and a rising edge are needed.
3. True or false. If the time interval between two consecutive falling edges is measured, the frequency of the periodic signal can be calculated.
4. True or False. The Freescale ARM KL25Z supports both rising and falling edge detection.
5. A Timer must be disabled (before, after) it is initialized.



## Section 5.5: Using Timer as an Event Counter

TPMx works as a counter when the CMOD field of TPMx\_SC is set to 10 (binary). In this mode, the timer counts the rising edges at the input pin synchronized to the timer counter clock. See Figure 5-29. For the timer to count the external edges the timer counter clock must be present and the external signal at the input pin should have the frequency half of the timer counter clock or lower. The timer counter in event counter mode operates the same as the counter described in Section 5.2. When the counter value reaches the Modulo register value, the TOF bit is set in TPMx\_SC register and the TPMx\_CNT value restarts from zero again. As shown in Figure 5-29, the external clock signal also passes through the prescaler. If the prescaler is set to divide by a number greater than 1, the external pulses are divided by the prescaler before incrementing the counter.

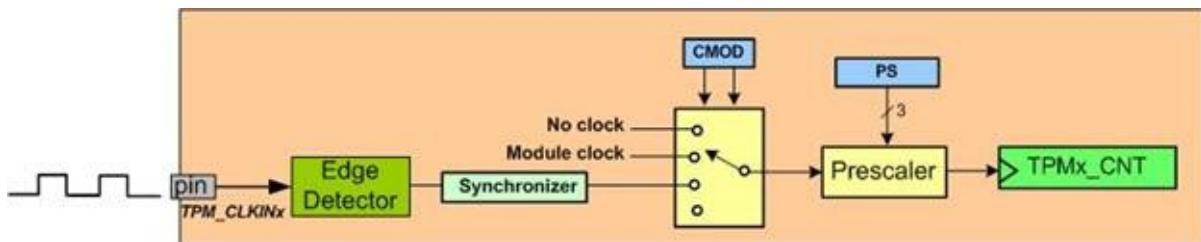


Figure 5-29: Counter Diagram

### Pin Selection for Event Counter

There are eight pins available to be used for external event counter. These pins are grouped as TPM\_CLKIN0 and TPM\_CLKIN1. The available pins for each group are listed in Table 5-8. The selected pin should have the clock enabled and the alternate port pin function set to 4 in PORTx\_PCR register. Each timer module in event counter mode may select one input pin from either TPM\_CLKIN0 or TPM\_CLKIN1. The selection is made in SIM\_SOPT4 register. Bit 26 of SIM\_SOPT4 is used for TPM2, bit 25 is used for TPM1, and bit 24 is used for TPM0. When the bit is 0, TPM\_CLKIN0 is used. When the bit is 1, TPM\_CLKIN1 is used.

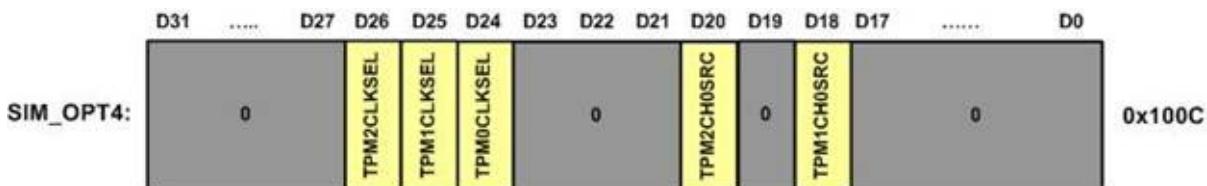


Figure 5-30: SIM\_SOPT4

On the FRDM-KL25Z board, the power supply is regulated at 3.3V when it is powered by the OpenSDA USB cable (J7). Input signal shall not exceed 3.6V otherwise damage to the pin or device may happen. For an input signal to be recognized by the input pin, a high signal should be above 2.31V and below 3.6V, a low signal should be below 1.15V and above 0V. For more details, check out the KL25 datasheet.

<b>TPM_CLKIN0 Pins</b>	PTA18	PORTA_PCR18=0x0400
	PTB16	PORTB_PCR16=0x0400
	PTC12	PORTC_PCR12=0x0400
	PTE29	PORTE_PCR29=0x0400
<b>TPM_CLKIN1 Pins</b>	PTA19	PORTA_PCR19=0x0400
	PTB17	PORTB_PCR17=0x0400
	PTC13	PORTC_PCR13=0x0400
	PTE30	PORTE_PCR30=0x0400

**Table 5-8: Input Clock Pins**

Program 5-10 uses Timer 0 to count the pulses present at PTC12. The least significant three bits of the counter value is displayed on the tri-color LEDs.

#### Program 5-10: Use Timer0 to count external pulse rising edges

```
/*
 * p5_10.c Counting pulses from PTC12.
 * This is used as the base for P5_10.
 * This program uses TPM0 to count the number of pulses
 * from PTC12.
 * The tri-color LEDs are used to display bit2-0 of
 * the counter. At low frequency input, the change of
 * LED color should be visible.
 * Although the counter is counting pulses from PTC12,
 * timer counter clock must be present.
 */
```

```
#include <MKL25Z4.H>
#include <stdio.h>

int main (void) {

    unsigned short count;

    /* Initialize GPIO pins for tri-color LEDs */
    SIM->SCGC5 |= 0x400;           /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTB->PCR[18] = 0x100;        /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x40000;          /* make PTB18 as output pin */
```

```

PORTB->PCR[19] = 0x100;      /* make PTB19 pin as GPIO */
PTB->PDDR |= 0x80000;        /* make PTB19 as output pin */
PORTD->PCR[1] = 0x100;       /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02;          /* make PTD1 as output pin */
/* end GPIO pin initialization for LEDs */

/* Start of Timer code */
SIM->SCGC5 |= 0x0800;        /* enable clock to Port C */
PORTC->PCR[12] = 0x400;       /* set PTC12 pin for TPM0 */
SIM->SOPT4 &= ~0x01000000;   /* use TPM_CLKIN0 as timer counter clock */
SIM->SCGC6 |= 0x01000000;    /* enable clock to TPM0 */
SIM->SOPT2 |= 0x01000000;    /* counter clock must be present */
TPM0->SC = 0;                /* disable timer while configuring */
TPM0->SC = 0x80;              /* prescaler /1 and clear TOF */
TPM0->MOD = 0xFFFF;           /* max modulo value */
TPM0->CNT = 0;                /* clear counter */
TPM0->SC |= 0x10;             /* enable timer and use LPTPM_EXTCLK */
while (1) {
    count = TPM0->CNT;
    /* change LEDs according to bit 2-0 of the value of count */
    if (count & 1)      /* use bit 0 of count to control red LED */
        PTB->PCOR = 0x40000;    /* turn on red LED */
    else
        PTB->PSOR = 0x40000;    /* turn off red LED */
    if (count & 2)      /* use bit 1 of count to control green LED */
        PTB->PCOR = 0x80000;    /* turn on green LED */
    else
        PTB->PSOR = 0x80000;    /* turn off green LED */
    if (count & 4)      /* use bit 2 of count to control blue LED */
        PTD->PCOR = 0x02;       /* turn on blue LED */
    else
        PTD->PSOR = 0x02;       /* turn off blue LED */
    /* end of LED code */
}
}

```

## Review Questions

1. True or false. The Timer can also be used as event counter.
2. True or false. The Freescale KL25Z timer can only count the falling edges.
3. True or false. In edge count mode the incoming pulses must be minimum of 3.3V to 5V to be detected.

4. True or false. With prescaler, we can divide the incoming pulses by prescaler value.
5. True or false. To use the timer as even-counter, we must configure it in capture mode.

# 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,  
 $\text{RELOAD} = 80,000 - 1 = 79,999$
3. 24
4. The D0 of STCTRL (the Enable)
5. Down counter

## Section 5.2

1. True
2. 3, Timer0 to Timer2
3. True
4. SIM\_SCGC6
5. SIM\_SOPT2
6. 6

## Section 5.3

1. True
2. False
3. False
4. True

## Section 5.4

1. True
2. False
3. True
4. True

5. before

## Section 5.5

1. True
2. False
3. False, a high signal should be above 2.31V but never exceed 3.6V and a low signal should be below 1.15V.
4. True
5. False





# Chapter 6: Interrupt and Exception Programming

This chapter examines the interrupts in ARM. We also discuss sources of hardware interrupts in the Freescale ARM KL25Z device. In Section 6.1 we discuss the concept of interrupts in the ARM CPU, and 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.

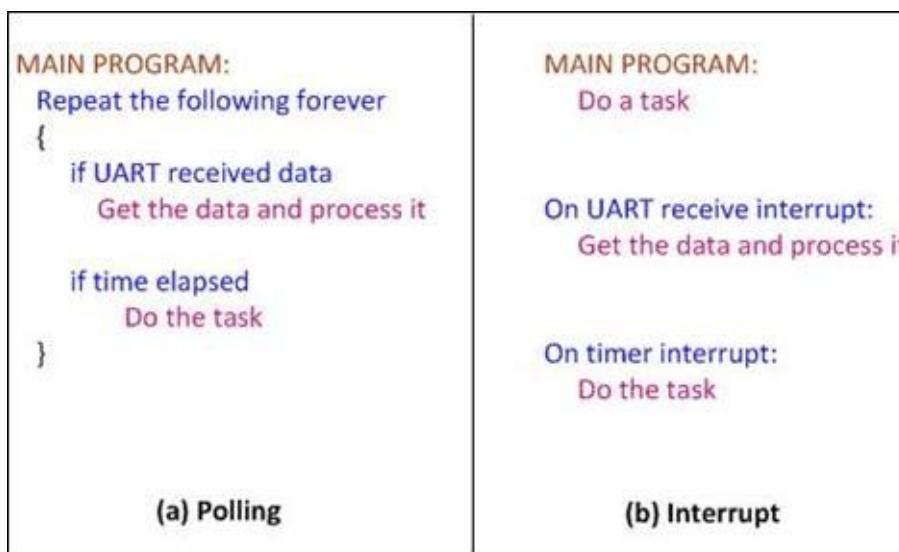


Figure 6-1: Polling vs. Interrupts

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 when they 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.

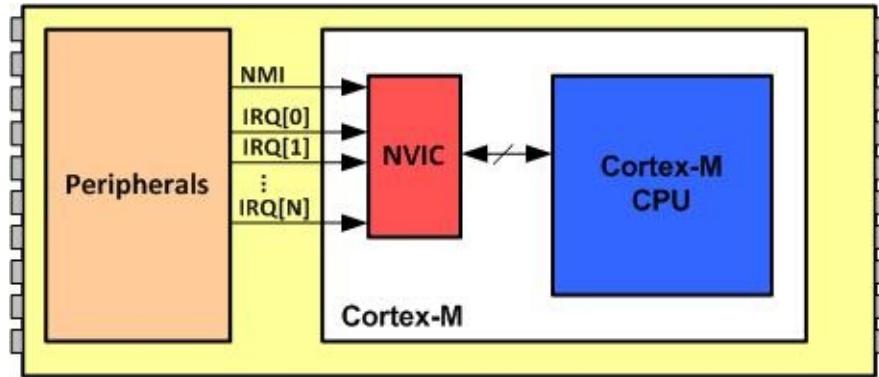


Figure 6-2: NVIC in ARM Cortex-M

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.

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

Table 6-1: Interrupt Vector Table for ARM Cortex-M

## The predefined Interrupts (INT 0 to INT 15)

The followings are the first 15 interrupts in ARM Cortex-M:

### Reset

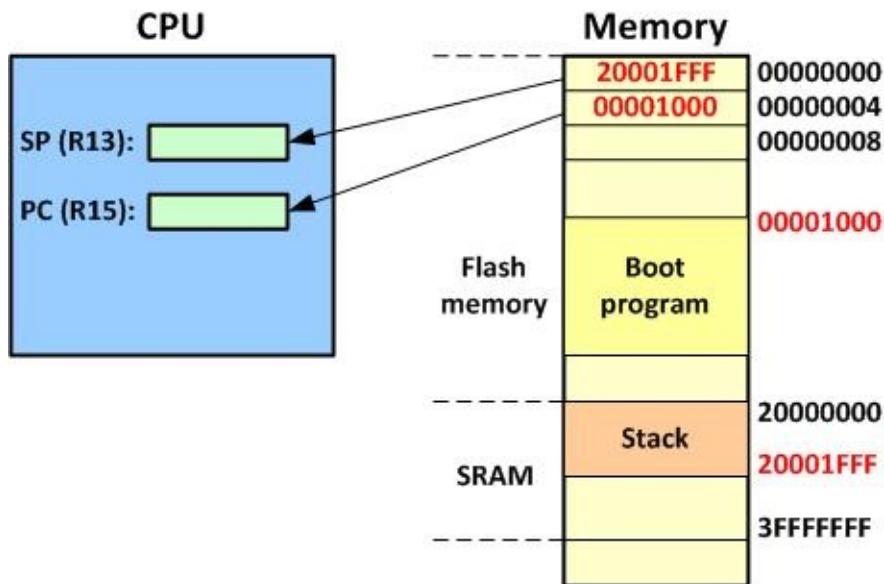


Figure 6-3: Going from Reset to Boot Program

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 ensures 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.

## **SVC**

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 SVC 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 belongs 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 KL25Z ARM chip allows enabling lazy stacking of the FPU registers. 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.

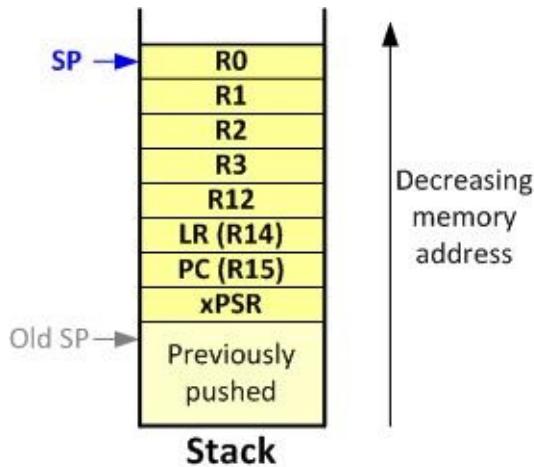


Figure 6-4: ARM Cortex-M Stack Frame upon Interrupt

## ***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.

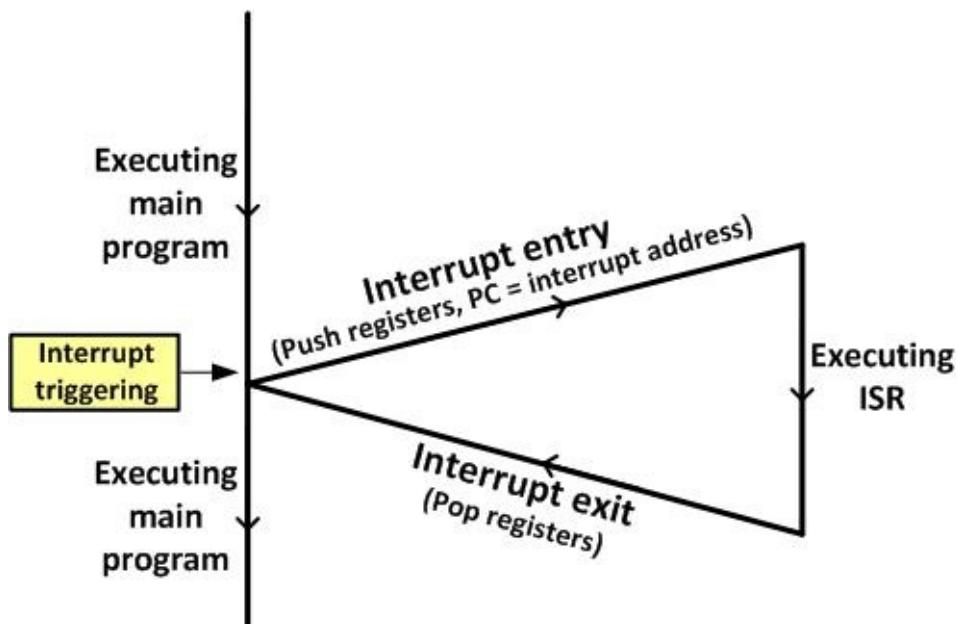


Figure 6-5: Main Program gets Interrupted

## 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 3 with 3 has the lowest priority.

Interrupt #	Interrupt	Priority Level
0	<i>Stack Pointer initial value</i>	
1	Reset	-3 Highest
2	NMI	-2
3	Hard Fault	-1
4	Memory Management Fault	Programmable
5	Bus Fault	Programmable
6	Usage Fault (undefined instructions, divide by zero, unaligned memory access, ....)	Programmable
7	Reserved	Programmable
8	Reserved	Programmable
9	Reserved	Programmable
10	Reserved	Programmable
11	SVCALL	Programmable
12	Debug Monitor	Programmable
13	Reserved	Programmable
14	PendSV	Programmable
15	SysTick	Programmable

16	IRQ for peripherals	Programmable
17	IRQ for peripherals	Programmable
...	...	Programmable
255	IRQ for peripherals	Programmable

Table 6-2: Interrupt Priority for ARM Cortex-M

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. Also, not all the interrupts are used in all the chips. The Freescale KL25Z uses only interrupt #1-#47 (or up to IRQ31).

## 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 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 peripheral 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.

<b>nPRIV (Privilege):</b>	Defines the Thread mode privilege level
0:	Privileged
1:	Unprivileged
<b>Active Stack Pointer (ASP):</b>	Defines the currently active stack pointer (ASP = SPSEL)
0:	MSP is the current stack pointer.
1:	PSP is the current stack pointer.
<b>Floating Point Context Active (FPCA)</b>	
0:	No floating point context active.
1:	Floating point context active.

Figure 6-6: CONTROL Register in ARM Cortex-M4

Processor Mode	Software	Privilege level
<b>Thread</b>	Applications	Privileged and Unprivileged
<b>Handler</b>	ISR for Exceptions and IRQs	Always Privileged
<i>In Thread mode, use bit 0 of the CONTROL register to select Privileged or Unprivileged</i>		

Table 6-3: Privileged level Execution and Processor Modes in ARM Cortex-M

## 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 has access to all registers including the special function registers for interrupts.
2. Privileged level software has access to every region of memory.
3. Privileged level software has 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.

Processor Mode	Software	Stack Usage
Thread	Applications	MSP or PSP
Handler	ISR for Exceptions and IRQs	MSP
<i>Note: In Thread mode, use bit 1 of the Control register to select MSP or PSP for stack pointer.</i>		

Table 6-4: Processor Modes and Stack Usage in ARM Cortex-M

Here are summary of the Unprivileged level software:

1. Unprivileged level software has no access to some registers such the special function registers for interrupts.
2. Unprivileged level software has limited access to some regions of memory.
3. Unprivileged level software is 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.

Mode	Privilege	Stack Pointer	Typical Example usage
<b>Handler</b>	Privileged	Main	Exception Handling
<b>Handler</b>	Unprivileged	Any	Reserved since Handler is always Privileged
<b>Thread</b>	Privileged	Main	Operating system kernel
<b>Thread</b>	Privileged	Process	
<b>Thread</b>	Unprivileged	Main	
<b>Thread</b>	Unprivileged	Process	Application threads

Table 6-5: Processor Mode, Privilege, and Stack in ARM Cortex

## 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.

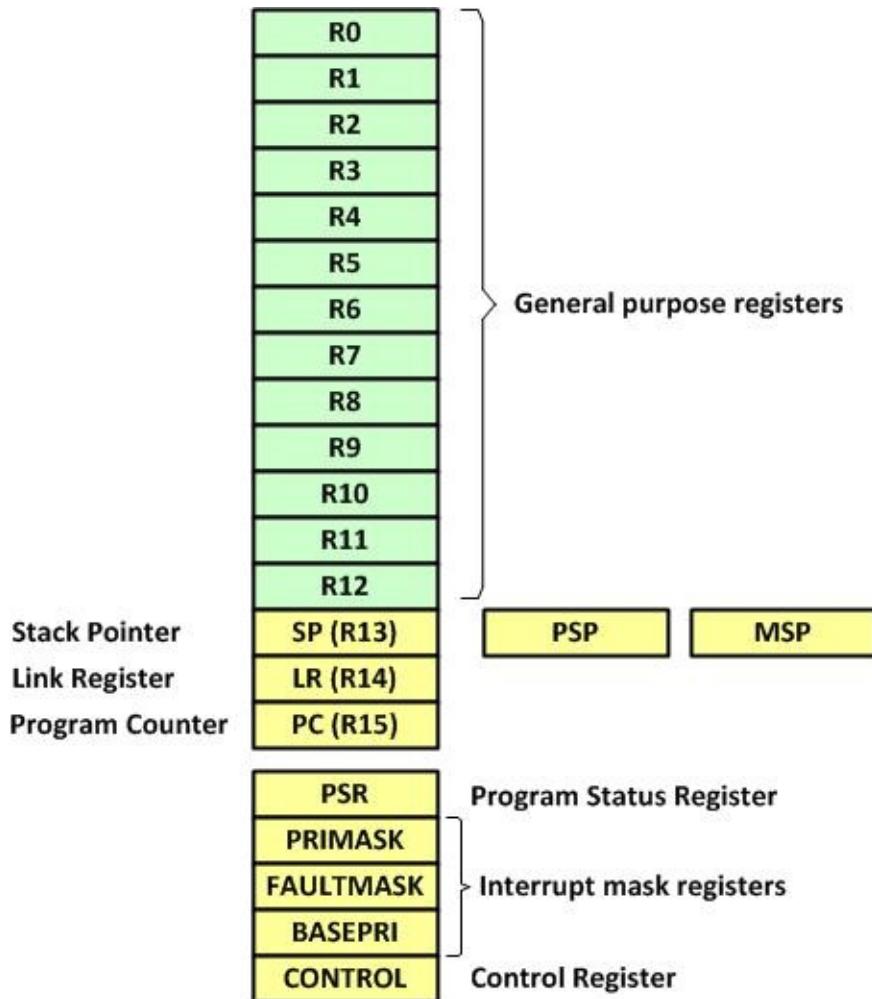


Figure 6-7: ARM Cortex-M Registers

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.

Register name	Privilege Usage
MSP (main stack pointer)	Privileged
PSP (processor stack pointer)	Privileged or Unprivileged
PSR (Processor status register)	Privileged
APSR (application processor status register)	Privileged or Unprivileged
ISPR (interrupt processor status register)	Privileged
EPSR (execution processor status register)	Privileged
PRIMASK (Priority Mask register)	Privileged
FAULTMASK (fault mask register)	Privileged
BASEPRI (base priority register)	Privileged
CONTROL (control register)	Privileged
<i>Note: We must use MSR and MRS instructions to access the above registers</i>	

Table 6-6: Special function registers of ARM Cortex-M

## Review Questions

- True or false. When a Reset pin is activated, the ARM CPU wakes up in Thread mode.
- There are only \_\_\_\_\_ processor modes in the ARM Cortex. Give their names
- How many bytes of data are fetched into CPU from interrupt vector table when ARM Cortex-M is Reset, and what are they?
- Another name for ISR is \_\_\_\_\_.
- True or false. When an interrupt comes in from exception fault or IRQ, the ARM CPU switches to Handler mode automatically.



## Section 6.3: Freescale I/O Port Interrupt Programming

Freescale KL25Z is an ARM Cortex-M chip. 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 I/O ports in KL25Z chip.

### PORTA and PORTD Interrupt Registers

In this section, we will use a switch to show an example of external interrupt programming using GPIO pins. However, before we do that, we need to examine the interrupt vector table for the Freescale KL25Z microcontroller. For Freescale KL25Z, only PORTA and PORTD are capable of generating interrupts. Table 6-7 shows interrupt assignment in KL25Z used by FRDM board. See Section 3.3 of KL25Z reference manual.

INT#	IRQ#	Vector location	Device
1-15	None	0000 0000 to 0000 003C	CPU Exception (set by ARM)
16	0	0000 0040	DMA
17	1	0000 0044	DMA
18	2	0000 0048	DMA
19	3	0000 004C	DMA
20	4	0000 0050	--
21	5	0000 0054	FTFA
22	6	0000 0058	PMC
23	7	0000 005C	LLWU
24	8	0000 0060	I2C0
25	9	0000 0064	I2C1
26	10	0000 0068	SPI0
27	11	0000 006C	SPI1
28	12	0000 0070	UART0
29	13	0000 0074	UART1
30	14	0000 0078	UART2
31	15	0000 007C	ADC0

32	16	0000 0080	CMP0
33	17	0000 0084	TPM0
34	18	0000 0088	TPM1
35	19	0000 008C	TPM2
36	20	0000 0090	RTC
37	21	0000 0094	RTC
38	22	0000 0098	PIT
39	23	0000 009C	—
40	24	0000 00A0	USB OTG
41	25	0000 00A4	DAC0
42	26	0000 00A8	TSI0
43	27	-	MCG
44	28	0000 00B0	LPTMR0
45	29	0000 00B4	—I
46	30	0000 00B8	I/O PORTA
47	31	0000-00BC	I/O PORTD

**Table 6-7: IRQ assignment in KL25Z of FRDM board**

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. In the Freescale KL25Z, the INT 46 is assigned to the I/O port of PORTA and INT47 to I/O PORTD. See Table 6-7. Although PORTA has many pins, we have only one interrupt assigned to the entire PORTA. This is common in many microcontrollers. In other words, when any of the PORTA pins trigger an interrupt, they all go to the same address location in the interrupt vector table. It is the 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 PORTA interrupt.

Upon Reset, all the interrupts are disabled at the peripheral modules and NVIC but enabled globally. To enable any interrupt we need these steps:

- 1) Enable the interrupt for a specific peripheral module.
- 2) Enable the interrupts at the NVIC module.

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 PORTx\_PCR register, as we will see soon.

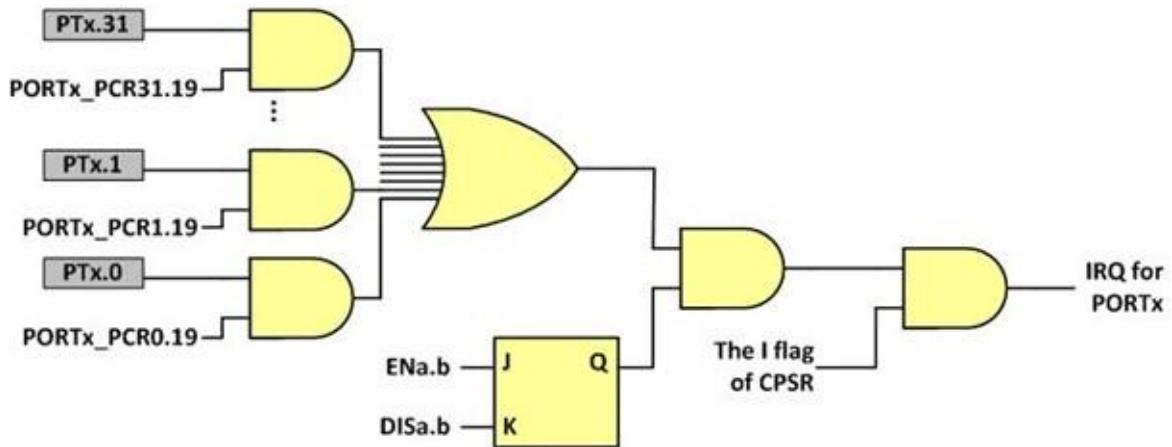


Figure 6-8: Interrupt enabling with all 3 levels

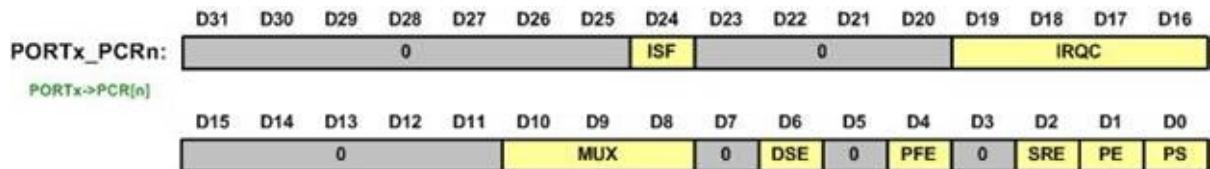


Figure 6-9: PORTx\_PCRn register

Notice that, the IRQC field (bits D19-D16) of PORTx\_PCR register are used to enable the interrupt capability of each pin of the I/O port. They allow you to select the trigger of interrupt by an edge or a level (which we will describe in more details later). For example, to enable the interrupts for PTA1 on the falling edge, we will need the following:

```
PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */
```

```
PORTA->PCR[1] |= 0xA0000; /* enable falling edge interrupt */
```

- 2) In ARM Cortex, 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 Figure 6-10.

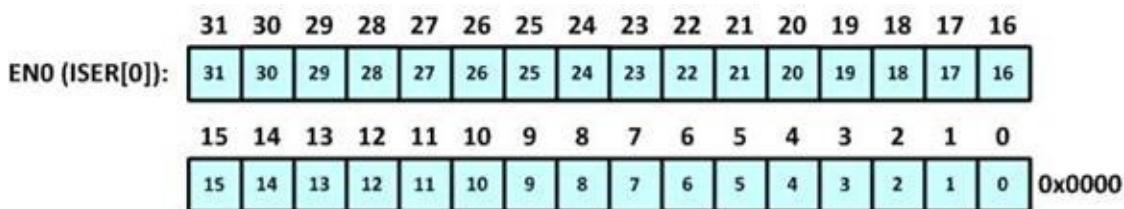


Figure 6-10: Interrupts 0–31 Set Enable (EN0)

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. The KL25Z has a total of 32 IRQs and only ISER[0] is used.

As we can see in the interrupt vector table in Table 6-7, the PORTA interrupt is assigned to IRQ30. Therefore, to enable the interrupt associated with PORTA 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\_cm0plus.h* file which is included in the *MKL25Z4.h* header file. To enable an interrupt using this function, the IRQ number of the interrupt should be passed as the argument to the function. For example, the following statement enables PORTA interrupt:

```
NVIC_EnableIRQ(30);
```

Since the IRQ numbers of all the interrupts are defined in the *MKL25Z4.h*, we can use their names instead of their numbers. For example, to enable PORTA interrupt the following can be used as well:

```
NVIC_EnableIRQ(PORTA_IRQn);
```

For more information, open the *MKL25Z4.h* file and find “typedef enum IRQn” in the file.

To disable interrupts there are other registers: ICER0 to ICER3. Again because KL25Z device has 32 IRQs, only ICER0 is used. See the Figure 6-11.



Figure 6-11: Interrupts 0–31 Clear Enable (DIS0)

Each interrupt can be disabled by writing a 1 to the corresponding bit in 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] = 0x1000; /*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 is connected to a J-K Flip-Flop, as shown below:

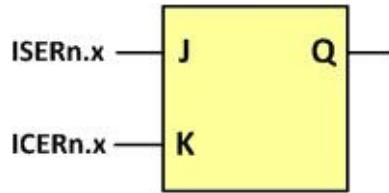


Figure 6-12: Enabling and Disabling an Interrupt

- 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:

```
_enable_irq(); /* Enable interrupt Globally */
```

and

```
_disable_irq(); /* Disable interrupt Globally */
```

It is a good idea to disable all interrupts during the initialization of the program and enable interrupts after all the initializations are complete. Now, using the following lines of code, we enable the interrupts for PTA1 pin at all three levels:

```
PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */
```

```
PORTA->PCR[1] |= 0xA0000; /* enable falling edge interrupt */
```

```
NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */
```

```
_enable_irq(); /* global enable IRQs */
```

## IRQ Priority

As described earlier, 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 IPR 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 3 in a KL25Z device and they are defined by two bits left justified in that byte. There are eight IPR registers to hold the priority of 32 IRQs. One needs to identify the byte and the register to set the IRQ priority. We will describe it in more details in Section 6.7.

## 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.

In Freescale KL25Z, we must use PORTX\_PCRn register to decide the level or edge for I/O interrupts.

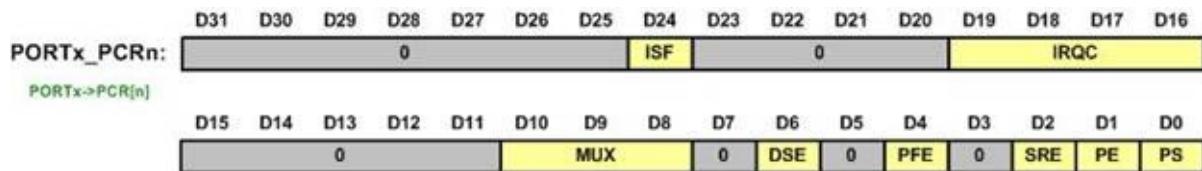


Figure 6-13: PORTx\_PCR Interrupt activation bits

From above figure, we see the D19-D16 bits are used to decide low-level, high-level, falling-edge, rising-edge, or both-edge activation.

D19	D18	D17	D16	
1	0	0	0	Interrupt when logic zero (Active Low-level).
1	0	0	1	Interrupt on rising edge.
1	0	1	0	Interrupt on falling edge.
1	0	1	1	Interrupt on either edge.
1	1	0	0	Interrupt when logic one (Active High-level)

Table 6-8: I/O Interrupt Trigger

Now, the following lines of code make the PTA1 interrupt trigger on logic zero.

```
PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */
PORTA->PCR[1] |= 0x80000; /* enable low-level interrupt */
```

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 on PTB18 continuously. When an interrupt comes from an external SW connected to PTA1, it toggles the green LED on PTB19 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\_MKL25Z4.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 address of the new interrupt handler. In the case of PORTA interrupt, the interrupt handler name is PORTA\_IRQHandler. 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 GPIO pin interrupt posts an interrupt flag at its corresponding bit in the Interrupt Status Flag Register (PORTx\_ISFR). For example, PTA1 interrupt posts the interrupt flag in bit 1 of PORTA\_ISFR. To clear the interrupt flag, the program writes a 1 to the location of the flag. Writing a zero to the flag has no effect. So to clear the interrupt flag of PTA1, the following statement is used:

**PORTA->ISFR |= 0x00000002;**

The following program configures PTA1 and PTA2 pins as external interrupt source. The pins are set to trigger on the falling edge of the signal and the internal pull-up is enabled. A push button switch should be connected between either PTA1 pin or PTA2 pin to ground. Pressing the push button switch will generate a falling edge and trigger a PORTA interrupt. The interrupt handler blinks the green LED three times. The infinite loop in main blinks the red LED continuously but when the interrupt is acknowledged and the green LED is blinking, the blinking of the red LED is halted.

#### Program 6-1: PORTA interrupt from a switch

```
/* p6_1: PORTA interrupt from a switch */  
/* Upon pressing a switch connecting either PTA1 or PTA2 to ground, the green  
LED will toggle for three times. */  
/* Main program toggles red LED while waiting for interrupt from switches. */
```

```

#include "MKL25Z4.h"

void delayMs(int n);

int main(void) {
    __disable_irq();          /* disable all IRQs */
    SIM->SCGC5 |= 0x400;     /* enable clock to Port B */
    PORTB->PCR[18] = 0x100;   /* make PTB18 pin as GPIO */
    PORTB->PCR[19] = 0x100;   /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0xC0000;    /* make PTB18, 19 as output pin */
    PTB->PDOR |= 0xC0000;    /* turn off LEDs */
    SIM->SCGC5 |= 0x200;     /* enable clock to Port A */
    /* configure PTA1 for interrupt */
    PORTA->PCR[1] |= 0x00100; /* make it GPIO */
    PORTA->PCR[1] |= 0x00003; /* enable pull-up */
    PTA->PDDR &= ~0x0002;    /* make pin input */
    PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */
    PORTA->PCR[1] |= 0xA0000; /* enable falling edge interrupt */
    // configure PTA2 for interrupt
    PORTA->PCR[2] |= 0x00100; /* make it GPIO */
    PORTA->PCR[2] |= 0x00003; /* enable pull-up */
    PTA->PDDR &= ~0x0004;    /* make pin input */
    PORTA->PCR[2] &= ~0xF0000; /* clear interrupt selection */
    PORTA->PCR[2] |= 0xA0000; /* enable falling edge interrupt */
    NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */
    __enable_irq();           /* global enable IRQs */

    /* toggle the red LED continuously */
    while(1) {
        PTB->PTOR |= 0x40000; /* toggle red LED */
    }
}

```

```

    delayMs(500);
}

}

/* A pushbutton switch is connecting either PTA1 or PTA2 to ground to trigger
PORTA interrupt */

void PORTA_IRQHandler(void) {
    int i;

    /* toggle green LED (PTB19) three times */
    for (i = 0; i < 3; i++) {
        PTB->PDOR &= ~0x80000; /* turn on green LED */
        delayMs(500);
        PTB->PDOR |= 0x80000; /* turn off green LED */
        delayMs(500);
    }
    PORTA->ISFR = 0x00000006; /* clear interrupt flag */
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++)
}

```

Notice in Program 6-1, if we have two switches connected to PTA1 and PTA2, the program does not make any distinction which switch is pressed. The reason is that only one interrupt (IRQ30) is associated with the entire PORTA. In

other words, whichever pin of PORTA interrupt is activated it goes to the same interrupt handler belonging to PORTA. Now, we can modify the Program 6-1 to distinguish between various pins of PORTA. That means, by pressing SW1 (PTA1) we can toggle the green LED (PTB19) and when SW2 (PTA2) is pressed blue LED (PTD1) is toggled. Each of the PORTA pin interrupt sets a bit in the PORTA\_ISFR. The interrupt handler reads the PORTA\_ISFR register to find out which interrupt flag bit is posted and which pin is requesting interrupt. See Program 6-2. This is like using a 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 is opened. Notice when both switches are pressed and released, both interrupts will be served sequentially. The sequence the interrupts are served depends on the sequence their interrupt flags are checked in the interrupt handler. For example in Program 6-2, PTA1 interrupt flag is checked first therefore PTA1 interrupt is serviced before PTA2 interrupt.

#### Program 6-2: Rewrite of the interrupt handler in Program 6-1 to distinguish the interrupt pin

```

/* p6_2: PORTA interrupt from one of the two switches */
/* Upon pressing a switch connecting PTA1 to ground, the green LED will toggle
for three times. If a switch between PTA2 and ground is pressed, the blue LED
will toggle for three times. */

/* Main program toggles red LED while waiting for interrupt from switches.

*/
#include "MKL25Z4.h"

void delayMs(int n);

int main(void)
{
    __disable_irq();          /* disable all IRQs */
    SIM->SCGC5 |= 0x400;     /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000;    /* enable clock to Port D */
    PORTB->PCR[18] = 0x100;  /* make PTB18 pin as GPIO */
    PORTB->PCR[19] = 0x100;  /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0xC0000;   /* make PTB18, 19 as output pin */
    PORTD->PCR[1] = 0x100;   /* make PTD1 pin as GPIO */
}

```

```

PTD->PDDR |= 0x02;      /* make PTD1 as output pin */
PTB->PDOR |= 0xC0000;    /* turn off red/green LEDs */
PTD->PDOR |= 0x02;      /* turn off blue LED */
SIM->SCGC5 |= 0x200;    /* enable clock to Port A */

// configure PTA1 for interrupt
PORTA->PCR[1] |= 0x00100; /* make it GPIO */
PORTA->PCR[1] |= 0x00003; /* enable pull-up */
PTA->PDDR &= ~0x0002;    /* make pin input */
PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */
PORTA->PCR[1] |= 0xA0000; /* enable falling edge interrupt */

// configure PTA2 for interrupt
PORTA->PCR[2] |= 0x00100; /* make it GPIO */
PORTA->PCR[2] |= 0x00003; /* enable pull-up */
PTA->PDDR &= ~0x0004;    /* make pin input */
PORTA->PCR[2] &= ~0xF0000; /* clear interrupt selection */
PORTA->PCR[2] |= 0xA0000; /* enable falling edge interrupt */

NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */
__enable_irq();           /* global enable IRQs */

/* toggle the red LED continuously */
while(1)
{
    PTB->PTOR |= 0x40000; /* toggle red LED */
    delayMs(500);
}

/* A pushbutton switch is connected to PTA1 pin to trigger PORTA interrupt */
void PORTA_IRQHandler(void)
{
    int i;
}

```

```

while (PORTA->ISFR & 0x00000006) {
    if (PORTA->ISFR & 0x00000002) {
        /* toggle green LED (PTB19) three times */
        for (i = 0; i < 3; i++) {
            PTB->PDOR &= ~0x80000; /* turn on green LED */
            delayMs(500);
            PTB->PDOR |= 0x80000; /* turn off green LED */
            delayMs(500);
        }
        PORTA->ISFR = 0x00000002; /* clear interrupt flag */
    }
    if (PORTA->ISFR & 0x00000004) {
        /* toggle blue LED (PTD1) three times */
        for (i = 0; i < 3; i++) {
            PTD->PDOR &= ~0x02; /* turn on blue LED */
            delayMs(500);
            PTD->PDOR |= 0x02; /* turn off blue LED */
            delayMs(500);
        }
        PORTA->ISFR = 0x00000004; /* clear interrupt flag */
    }
}
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
}

```

```

for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}

```

## Interrupting on both edge

The next example Program 6-3 is to connect a square wave of 3 V to PTD4 pin and let an LED toggle at the same rate as the input frequency. The PORTD interrupt is configured to trigger on both rising edge and falling edge. Because there is no delay in the interrupt handler and the interrupt is triggered by rising edge and falling edge, it is necessary to trigger the interrupts using a function generator. If you connect PTD4 to a switch to the ground, the program does not work well because of contact bounce (as described in Chapter 3). During the contact bounce, it may trigger interrupt several times. The interrupt handler supposed to toggle the blue LED. If even numbers of interrupts are acknowledged when the switch is pressed or released because of contact bounce, the LED will not change. Also notice in Table 6-7, IRQ31 is assigned to PORTD.

**Program 6-3: External interrupt on both edges**

```
/* p6_3: Toggle blue LED by PTD4 interrupt on both edges */
```

```
/* PTD4 is configured to interrupt on both rising edge and falling edge. In the
interrupt handler, the blue LED (PTD1) is toggled. */
```

```
#include "MKL25Z4.h"
```

```
int main(void)
```

```
{
```

```
    __disable_irq();      /* disable all IRQs */
```

```
    SIM->SCGC5 |= 0x1000;    /* enable clock to Port D */
```

```
    PORTD->PCR[1] = 0x100;    /* make PTD1 pin as GPIO */
```

```
    PTD->PDDR |= 0x02;      /* make PTD1 as output pin */
```

```
    PTD->PDOR |= 0x02;      /* turn off blue LED */
```

```
// configure PTD4 for interrupt
```

```
    PORTD->PCR[4] |= 0x00100; /* make it GPIO */
```

```

PORTD->PCR[4] |= 0x00003; /* enable pull-up */
PTD->PDDR &= ~0x0010;    /* make pin input */
PORTD->PCR[4] &= ~0xF0000; /* clear interrupt selection */
PORTD->PCR[4] |= 0xB0000; /* enable both edge interrupt */
NVIC->ISER[0] |= 0x80000000; /* enable INT31 (bit 31 of ISER[0]) */
__enable_irq();           /* global enable IRQs */

while(1)
{
}

/* A pushbutton switch is connected to PTD4 pin to trigger PORTD interrupt */
void PORTD_IRQHandler(void)
{
    if (PORTD->ISFR & 0x00000010) {
        /* toggle blue LED (PTD1) */
        PTD->PTOR |= 0x0002; /* toggle blue LED */
        PORTD->ISFR = 0x0010; /* clear interrupt flag */
    }
}

```

## 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 Freescale KL25Z support 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 Freescale ARM KL25Z using polling. This chapter shows how to do the same thing using interrupt. Using interrupt frees up the CPU from having to poll the status of UART.

### UART0 Interrupt Programming to receive data

Program 4-2 showed how UART0 receives data by polling the RDRF status flag. The disadvantage with that program is that it ties down the CPU while polling the status flag. We can modify it to make it an interrupt driven program. Examining the UARTx\_C2 (UARTx Control 2) register, we see bit 5 allows us to enable the receiver interrupt. If the receiver interrupt for UART is enabled when a byte is received, the receiver RDRF flag is directed to NVIC and that causes the interrupt handler associated with the UART0 to be executed. In the UART0 handler we must read the received character. Reading the received character from the data register clears the RDRF flag.

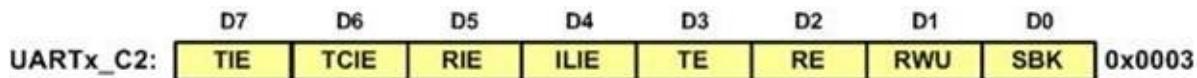


Figure 6-14: UARTx\_Control2 (UARTx\_C2)

Field	Bit	Description
TIE	D7	Transmit Interrupt Enable bit. Used for interrupt-driven UART. 0 = TDRE Interrupt Request is disabled. 1 = TDRE Interrupt Request is enabled.
TCIE	D6	Transmission Complete Interrupt Enable bit. Used for interrupt-driven UART. 0 = TC Interrupt Request is disabled. 1 = TC Interrupt Request is enabled.
RIE	D5	Receiver Full Interrupt Enable bit. Used for interrupt-driven UART. 0 = RDRF Interrupt Request is disabled. 1 = RDRF Interrupt Request is enabled.
ILIE	D4	Idle Line Interrupt Enable bit. Used for interrupt-driven UART. 0 = IDLE Interrupt Request is disabled. 1 = IDLE Interrupt Request is enabled.
		Transmitter Enable bit. We must enable this bit to transmit

<b>TE</b>	D3	data. 0 = Transmitter is disabled. 1 = Transmitter is enabled.
<b>RE</b>	D2	Receiver Enable bit. We must enable this bit to receive data. 0 = Receiver is disabled. 1 = Receiver is enabled.
<b>RWU</b>	D1	Used for wake-up condition in stand-by mode. See the KL25Z manual. 0 = Normal operation 1 = RWU is enabled.
<b>SBK</b>	D0	Used for break bit. See the KL25Z manual. 0 = No break character 1 = Transmit break character

Table 6-9: UART Control 2 (UARTx\_C2) register

From Table 6-7 we see IRQ12 is assigned to UART0. We enable the receiver interrupt in UART0 as follow:

```
UART0->C2 = 0x24; /* enable receive and receive interrupt*/
```

```
NVIC->ISER[0] |= 0x00001000; /* enable INT12 (bit 12 of ISER[0]) */
```

```
_enable_irq(); /* global enable IRQs */
```

In Program 6-4, pressing a key at the terminal emulator causes the PC to send the ASCII code of the key to the FRDM board. When the character is received by UART0, the interrupt handler reads the character and writes it on the LEDs.

#### Program 6-4: Using the UART0 interrupt

```
/* p6_4.c UART0 Receive using interrupt */
```

```
/* This program modifies p4_2.c to use interrupt to handle the UART0 receive.
```

Receiving any key from terminal emulator (TeraTerm) of the host PC to the UART0 on Freescale FRDM-KL25Z board.

UART0 is connected to openSDA debug agent and has a virtual connection to the host PC COM port.

Launch TeraTerm on a PC and hit any key.

The LED program from P2\_7 of Chapter 2 is used to turn on the tri-color LEDs

according to the key received.

By default in SystemInit(), FLL clock output is 41.94 MHz.

Setting BDH=0, BDL=0x17, and OSR=0x0F yields 115200 Baud. \*/

```
#include <MKL25Z4.H>
```

```
void UART0_init(void);
```

```
void delayMs(int n);
```

```
void LED_init(void);
```

```
void LED_set(int value);
```

```
int main (void) {
```

```
    __disable_irq();      /* global disable IRQs */
```

```
    UART0_init();
```

```
    LED_init();
```

```
    __enable_irq();      /* global enable IRQs */
```

```
    while (1) {
```

```
        /* UART0 receive is moved to interrupt handler*/
```

```
    }
```

```
}
```

```
/* UART0 interrupt handler */
```

```
void UART0_IRQHandler(void) {
```

```
    char c;
```

```
    c = UART0->D;      /* read the char received */
```

```
    LED_set(c);        /* and use it to set LEDs */
```

```
}
```

```
/* initialize UART0 to receive at 115200 Baud */
```

```

void UART0_init(void) {
    SIM->SCGC4 |= 0x0400; /* enable clock for UART0 */
    SIM->SOPT2 |= 0x04000000; /* use FLL output for UART Baud rate
generator */

    UART0->C2 = 0; /* turn off UART0 while changing configurations */
    UART0->BDH = 0x00;
    UART0->BDL = 0x17; /* 115200 Baud */
    UART0->C4 = 0x0F; /* Over Sampling Ratio 16 */
    UART0->C1 = 0x00; /* 8-bit data */
    UART0->C2 = 0x24; /* enable receive and receive interrupt*/
    NVIC->ISER[0] |= 0x00001000; /* enable INT12 (bit 12 of ISER[0]) */

}

/* initialize all three LEDs on the FRDM board */

```

```

void LED_init(void)
{
    SIM->SCGC5 |= 0x400; /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
    PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x40000; /* make PTB18 as output pin */
    PTB->PSOR |= 0x40000; /* turn off red LED */
    PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0x80000; /* make PTB19 as output pin */
    PTB->PSOR |= 0x80000; /* turn off green LED */
    PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02; /* make PTD1 as output pin */
    PTD->PSOR |= 0x02; /* turn off blue LED */
}

```

```

/* turn on or off the LEDs according to bit 2-0 of the value */

void LED_set(int value)
{
    if (value & 1) /* use bit 0 of value to control red LED */
        PTB->PCOR = 0x40000; /* turn on red LED */
    else
        PTB->PSOR = 0x40000; /* turn off red LED */
    if (value & 2) /* use bit 1 of value to control green LED */
        PTB->PCOR = 0x80000; /* turn on green LED */
    else
        PTB->PSOR = 0x80000; /* turn off green LED */
    if (value & 4) /* use bit 2 of value to control blue LED */
        PTD->PCOR = 0x02; /* turn on blue LED */
    else
        PTD->PSOR = 0x02; /* turn off blue LED */
}

```

Also notice that, in Program 6-4 there is only a single interrupt for both receiver and transmitter of UART0. If we want to implement both transmitter and receiver interrupts, then we have to test the TDRE and RDRF bits in register UART0\_S1 to see which one caused the interrupt.

## Review Questions

1. In Freescale KL25Z, Which IRQ is assigned to UART0?
2. True or false. There is only one interrupt for both Receiver and Transmitter.
3. Which pins are assigned to UART0\_TX and UART0\_RX of FRDM board?
4. We use register \_\_\_\_\_ to 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 time. To do that, we need to enable the timer interrupt using the TOIE (Timer Overflow Interrupt Enable) in TPMx\_SC register.

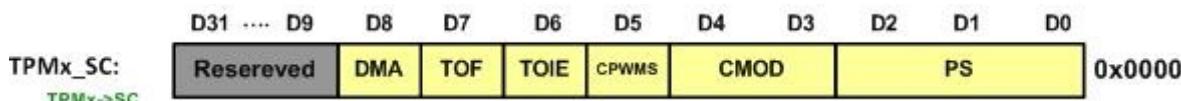


Figure 6-15: TOIE in TPMx\_SC (Timer Status Control) register

From Table 6-7, notice that IRQ17, IRQ18, and IRQ19 are assigned to TPM0, TPM1, and TPM2, respectively. The following will enable these timers in NVIC:

```
NVIC->ISER[0] |= 0x0020000; /* enable IRQ17 (D17 of ISER[0]) */  
NVIC->ISER[0] |= 0x0040000; /* enable IRQ18 (D18 of ISER[0]) */  
NVIC->ISER[0] |= 0x0080000; /* enable IRQ19 (D19 of ISER[0]) */
```

In Program 6-5, the main program toggles the blue LED of PTD1 continuously. Using interrupts, TPM0 Toggles red LED (PTB18) and TPM1 toggles green LED (PTB19), every so often.

### Program 6-5: Toggling the green LED using the Timer interrupt

```
/* p6_5.c Toggling red and green LED using TPM0 and TPM1 interrupts
```

This program is modified from p5-5 and p5-6 but instead of using polling, the timeout interrupts are used.

TPM1 is configured to run twice the frequency of TPM0.

The infinite loop in main blinks the blue LED. The TPM0 interrupt handler toggles the red LED and the TPM1 green LED. \*/

```
#include <MKL25Z4.H>
```

```
void delayMs(int n);
```

```
int main (void) {
    __disable_irq();      /* global disable IRQs */

    SIM->SCGC5 |= 0x400;      /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000;     /* enable clock to Port D */
    PORTB->PCR[18] = 0x100;   /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x40000;    /* make PTB18 as output pin */
    PORTB->PCR[19] = 0x100;   /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0x80000;    /* make PTB19 as output pin */
    PORTD->PCR[1] = 0x100;    /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;       /* make PTD1 as output pin */
    SIM->SOPT2 |= 0x01000000; /* use MCGFLLCLK as timer counter clock */
    SIM->SCGC6 |= 0x01000000; /* enable clock to TPM0 */
    TPM0->SC = 0;           /* disable timer while configuring */
    TPM0->SC = 0x07;         /* prescaler /128 */
    TPM0->MOD = 0xFFFF;      /* max modulo value */
    TPM0->SC |= 0x80;        /* clear TOF */
    TPM0->SC |= 0x40;        /* enable timeout interrupt */
    TPM0->SC |= 0x08;        /* enable timer */
    NVIC->ISER[0] |= 0x00020000; /* enable IRQ17 (bit 17 of ISER[0]) */

    SIM->SCGC6 |= 0x02000000; /* enable clock to TPM1 */
    TPM1->SC = 0;           /* disable timer while configuring */
    TPM1->SC = 0x07;         /* prescaler /128 */
    TPM1->MOD = 0x7FFF;      /* half of the max modulo value */
    TPM1->SC |= 0x40;        /* enable timeout interrupt */
    TPM1->SC |= 0x08;        /* enable timer */
    NVIC->ISER[0] |= 0x00040000; /* enable IRQ18 (bit 18 of ISER[0]) */

    __enable_irq();          /* global enable IRQs */

    while (1) {
```

```

PTD->PTOR = 0x02;      /* toggle blue LED */

delayMs(1500);

}

}

void TPM0_IRQHandler(void) {
    PTB->PTOR = 0x40000;      /* toggle red LED */
    TPM0->SC |= 0x80;        /* clear TOF */
}

void TPM1_IRQHandler(void) {
    PTB->PTOR = 0x80000;      /* toggle green LED */
    TPM1->SC |= 0x80;        /* clear TOF */
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++)
}

```

## Review Questions

1. For KL25Z chip, which IRQ is assigned to TPM0?
2. True or false. There is only one interrupt for all the TPM0 through TPM2.
3. We use register \_\_\_\_\_ to enable the interrupt associated with TPM0.
4. Show the contents of EN0 register for enabling TPM0.
5. True or false. Upon Reset, TPM0 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 6-16.

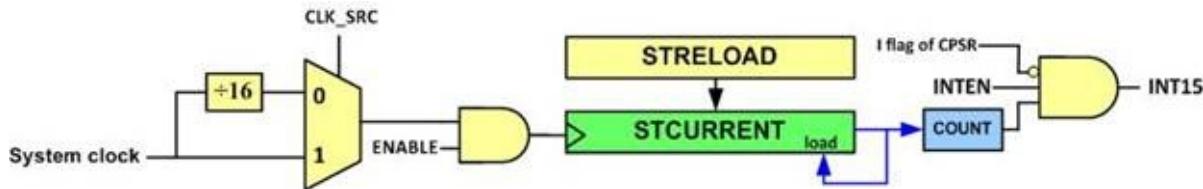


Figure 6-16: SysTick Internal Structure

If INTEN=1, when the COUNT flag is set, it generates an interrupt. INTEN is D1 of the STCTRL register, as shown in Figure 6-17.

STCTRL: SysTick->CTRL								0x010
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						
2	CLK_SRC	Clock Source 0: System clock divided by 16 1: System clock						
16	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 STRCTRL or writing to STCURRENT register.</i>						

Figure 6-17: 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 at 10 ms interval) to monitor and control the system operations. See Figure 6-18.

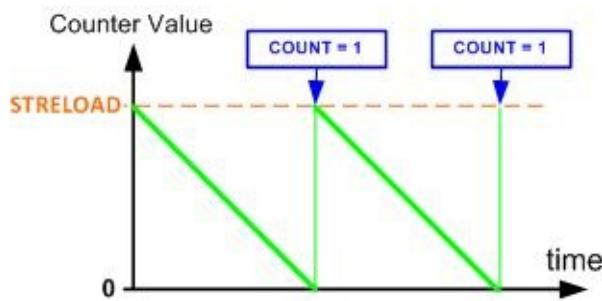


Figure 6-18: SysTick Counting

### Using SysTick with Interrupt

Examining interrupt vector table for ARM Cortex, we see the SysTick is the interrupt #15.

The Program 6-6 uses the SysTick to toggle the red LED of PTB18 every second. We need the RELOAD value of  $41,940,000/16-1$ . The system clock is running at 41.94 MHz. The CLK\_SRC bit of CTRL register is cleared so system clock/16 is used as the clock source of SysTick. The COUNT flag is raised every 41,940,000 clocks and an interrupt occurs. Then the RELOAD register is loaded with  $41,940,000/16-1$  automatically. ( $41,940,000-1$  is too large a number to fit in the 24-bit counter of SysTick so the system clock is divided by 16 first.)

Notice the interrupt is enabled in SysTick Control register. Because SysTick is an interrupt below interrupt # 16, the enable/disable and the priority are not managed by the registers in the NVIC. Its priority is controlled in the most significant byte of System Handler Priority 3 register (SHPR3) of System Control Block (SCB->SHP[1]). There is no need to clear interrupt flag in the interrupt handler for SysTick.

#### Program 6-6: SysTick interrupt

```
/* p6_6.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 41.94 MHz.
```

```
* In the interrupt handler, the red LED is toggled.
```

```
*/
```

```
#include <MKL25Z4.H>
```

```
int main (void) {
```

```
    __disable_irq();      /* global disable IRQs */
```

```
    SIM->SCGC5 |= 0x400;    /* enable clock to Port B */
```

```
    PORTB->PCR[18] = 0x100;  /* make PTB18 pin as GPIO */
```

```
    PTB->PDDR |= 0x40000;   /* make PTB18 as output pin */
```

```
    SysTick->LOAD = 41940000/16-1; /* reload with number of clocks per second */
```

```
    SysTick->CTRL = 3;        /* enable SysTick interrupt, use system clock/16 */
```

```
__enable_irq();          /* global enable interrupt */  
while (1) { /*wait here for interrupt */  
}  
}  
  
void SysTick_Handler(void) {  
    PTB->PTOR = 0x40000;    /* toggle red LED */  
}
```

## Review Questions

1. Which interrupt is assigned to SysTick in KL25Z chip?
2. The highest number we can place in TPMx\_MOD register is \_\_\_\_\_.
3. We use register \_\_\_\_\_ to enable the interrupt associated with SysTick.
4. True or false. We use EN0 register to enable SysTick interrupt.
5. Assume TPM1 clock source frequency of 32.768 kHz. Find the value for TPM1\_MOD register if we want 4 msec elapsed time.



## Section 6.7: Interrupt Priority Programming in Freescale 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 ARM Cortex, 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 ARM Cortex data sheet. In this section, we deal with the priority of peripheral interrupts of INT16 (IRQ0) to INT47 (IRQ31).

### IRQ0 to IRQ31 in Freescale ARM KL25Z

The INT16 is assigned to IRQ0 since the first 16 interrupts (INT0-INT15) are used by the ARM core itself. Not all the IRQs are implemented in all ARM chips. For example, The Freescale ARM KL25Z chip implements up to IRQ31 (or INT47). In other words, KL25Z has IRQ0 to IRQ31. 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 ARM Cortex 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.”

When more than one interrupts is pending, the interrupt with the highest priority is acknowledged first. While an interrupt handler is running, another interrupt with higher priority will interrupt the current interrupt handler and start its own interrupt handler (nested interrupts).

## The IPR registers and Priority Grouping in ARM Cortex

The priority of an IRQ is assigned in one of the interrupt priority registers called *IPRx* (*Interrupt PRorityx*) in NVIC. If we do not assign a priority to an IRQ, by default, it has priority 0. Each IRQ uses one byte in an interrupt priority register. Therefore, each interrupt priority register holds priorities for four IRQ. For example, IPR0 (IPR zero) holds the priorities of IRQ0, IRQ1, IRQ2 and IRQ3. In the same way, the priorities of IRQ4, IRQ5, IRQ6 and IRQ7 are assigned in IPR1. For 32 IRQs, eight interrupt priority registers are used. The KL25Z device uses only the two most significant bits of the byte in the interrupt priority register. With two bits, there can be four different priorities, 0 to 3. The lower the number the higher the priority is. See Figure 6-19.

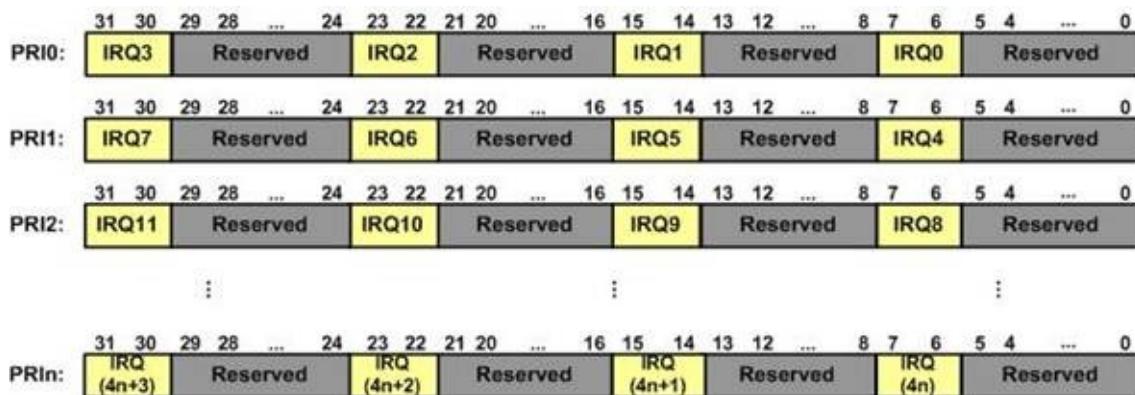


Figure 6-19: IPRn Registers

Notice, there is a pattern in the IPR# and IRQ# assignments. It follows the following formula:

**IPR<sub>n</sub>**    IRQ(4n), IRQ(4n+1), IRQ(4n+2), and IRQ(4n+3)

In other words, we multiply the IPR# 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 has a macro `NVIC_SetPriority` defined in `core_cm0plus.h` for programmers to set the priority of an IRQ.

For example, if we want to set the Timer 2 interrupt priority to 2, first we need to find out the IRQ number of Timer 2 interrupt, which is 19. To locate the register for IRQ19, we will divide 19 by 4, which results in a quotient of 4 and remainder of 3. The byte that holds the priority of Timer 2 is byte 3 of IPR4. To get to byte 3, we need to shift the priority 24 bits (8 x 3) to the left and to get the two most significant bits, we need to shift it 6 more bits to the left. The statement will look like:

`NVIC->IP[IRQn] |= PRIO << (8 * (IRQn % 4) + 6);`

or

`NVIC->IP[4] |= 2 << (8 * 3 + 6);`

This is tedious and error prone. I would be easier to use the macro:

```
NVIC_SetPriority (TPM2_IRQn, 2);
```

TPM2\_IRQn is defined in MKL25Z4.h.

Program 6-7 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.) TPM0 is programmed to interrupt at 1 second interval. In the interrupt handler, the red LED is turned on for 500 ms. TPM1 is programmed to interrupt at 100 ms interval and in its interrupt handler, the green LED is turned on for 20 ms. Since TPM0 has higher priority, you will observe that the green LED is not blinking when TPM1 interrupt is running (when the red LED is on). Now change the priority of the TPM1 to be higher than TPM0 by changing the line in TPM1\_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 TPM1 (green LED) preempts TPM0 interrupt handler (red LED).

Program 6-7: Interrupt priority demonstration	
/* P6_7.c Testing nested interrupts	
* Timer1 is setup to interrupt at 1 Hz.	
* In timer interrupt handler, the red LED is turned on and a delay function of 350 ms is called. The LED is turned off at the end of the delay.	
*	
* Timer2 is setup to interrupt at 10 Hz.	
* In timer interrupt handler, the blue LED is turned on and a delay function of 20 ms is called. The LED is turned off at the end of the delay.	
*	
* When Timer1 has higher priority, the Timer2 interrupts are blocked by Timer1 interrupt handler. You can see that when the red LED is on, the blue LED stops blinking.	
* When Timer2 has higher priority, the Timer1 interrupt handler is preempted by Timer2 interrupts and the blue LED is blinking all the time.	
* Timer2 interrupt handler also turns PD2 low and high so that it may be probed by the scope.	

```

*/



#include "MKL25Z4.h"



void Timer1_init(void);
void Timer2_init(void);
void delayMs(int n);



int main (void)
{
    __disable_irq();

    SIM->SCGC5 |= 0x400;      /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000;     /* enable clock to Port D */
    PORTB->PCR[18] = 0x100;   /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x40000;    /* make PTB18 as output pin */
    PORTD->PCR[1] = 0x100;    /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;       /* make PTD1 as output pin */
    PORTD->PCR[2] = 0x100;    /* make PTD2 pin as GPIO */
    PTD->PDDR |= 0x04;       /* make PTD2 as output pin */

    Timer1_init();
    Timer2_init();

    __enable_irq();



    while(1) /*wait here for interrupt */

    {
    }
}

void TPM1_IRQHandler(void)
{
    PTD->PCOR |= 0x02;      /* turn on blue LED */
}

```

```

delayMs(350);
PTD->PSOR |= 0x02; /* turn off blue LED */
TPM1->SC |= 0x80; /* clear TOF */
}

void TPM2_IRQHandler(void)
{
    PTB->PCOR |= 0x40000; /* turn on red LED */
    PTD->PCOR |= 0x04; /* make PTD2 low */
    delayMs(50);
    PTB->PSOR |= 0x40000; /* turn off red LED */
    PTD->PSOR |= 0x04; /* make PTD2 high */
    TPM2->SC |= 0x80; /* clear TOF */
}

/* priority of TPM1 and TPM2 should be between 0 and 3 */
#define PRIOTPM1 2U
#define PRIOTPM2 3U

void Timer1_init(void)
{
    SIM->SCGC6 |= 0x02000000; /* enable clock to TPM1 */
    SIM->SOPT2 |= 0x03000000; /* use MCGIRCLK as timer counter clock */
    TPM1->SC = 0; /* disable timer while configuring */
    TPM1->SC = 0x02; /* prescaler /4 */
    TPM1->MOD = 8192 - 1; //16384 - 1; /* modulo value */
    TPM1->SC |= 0x80; /* clear TOF */
    TPM1->SC |= 0x48; /* enable timer free-running mode and interrupt */

    /* set interrupt priority for TPM1 */
    /* NVIC->IP[TPM1_IRQn / 4] |= PRIOTPM1 << ((TPM1_IRQn % 4) * 8 + 6); */
}

```

```

/* NVIC->IP[4] |= PRIOTPM1 << 22; */
NVIC_SetPriority(TPM1_IRQn, PRIOTPM1);
NVIC_EnableIRQ(TPM1_IRQn); /* enable Timer1 interrupt in NVIC */
}

void Timer2_init(void)
{
    SIM->SCGC6 |= 0x04000000; /* enable clock to TPM2 */
    SIM->SOPT2 |= 0x03000000; /* use MCGIRCLK as timer counter clock */
    TPM2->SC = 0;           /* disable timer while configuring */
    TPM2->SC = 0x02;        /* prescaler /4 */
    TPM2->MOD = 819 - 1;    /* modulo value */
    TPM2->SC |= 0x80;       /* clear TOF */
    TPM2->SC |= 0x48;       /* enable timer free-running mode and interrupt */

    /* set interrupt priority for TMP2 */
    /* NVIC->IP[TPM2_IRQn / 4] |= PRIOTPM2 << ((TPM2_IRQn % 4) * 8 + 6); */
    /* NVIC->IP[4] |= PRIOTPM2 << 30; */
    NVIC_SetPriority(TPM2_IRQn, PRIOTPM2);
    NVIC_EnableIRQ(TPM2_IRQn); /* enable Timer2 interrupt in NVIC */
}

// 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 < 7000; j++)
            {} /* do nothing for 1 ms */
}

```

## 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 IRQ21, we need to program the IPR\_\_ 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. INT16
2. False
3. True
4. `__enable_irq();`
5. (a) on the peripheral device level. (b) on the system level with ISER register in NVIC. (c) on the global level with the `__enable_irq();` statement.

## Section 6.4

1. IRQ12
2. True
3. PTA0 and PTA1
4. NVIC\_SER0
5. False

## Section 6-5

1. IRQ17
2. False
3. NVIC\_SER0
4. IRQ17 is assigned to TPM0. Therefore, EN0=0x0020000
5. False

## Section 6-6

1. INT15
2. 0xFFFFFFF
3. STCTRL
4. False. NVIC\_EN0 register is used for IRQs (external interrupts) and SysTick is not part of them
5.  $1 / 32.768 \text{ kHz} = 30.52 \mu\text{sec}$ . Now,  $4 \text{ msec} / 30.52 \mu\text{sec} = 131$ . Therefore, TPM1\_MOD = 130.

## Section 6-7

1. Reset
2. False
3. IPR2
4. IPR5





# **Chapter 7: ADC, DAC, and Sensor Interfacing**

This chapter explores more real-world devices such as ADCs (analog-to-digital converters), DACs (digital-to-analog converters), and sensors. We will also explain how to interface the Freescale ARM KL25Z to these devices. In Section 7.1, we describe analog-to-digital converter (ADC) chips. We will program the ADC module of the KL25Z chip in Section 7.2. In Section 7.3, we show the interfacing of sensors and discuss the issue of signal conditioning. The characteristics and programming 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

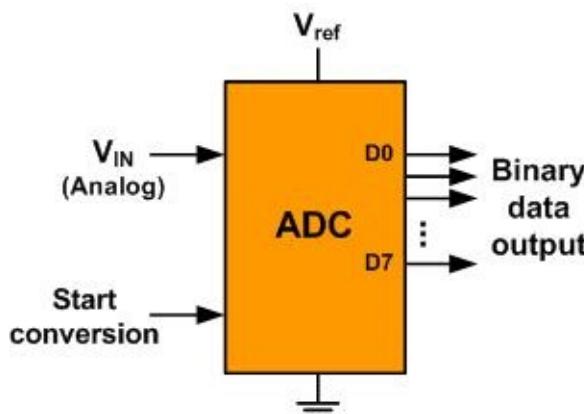


Figure 7-2: An 8-bit ADC Block Diagram

### 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: <math>V_{ref} = 5V</math></b>		

Table 7-1: Resolution versus Step Size for ADC ( $V_{ref} = 5V$ )

## $V_{ref}$

$V_{ref}$  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  $V_{ref} / 256$  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,  $V_{ref}$  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  $2.56V / 256 = 10 \text{ mV}$ . 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  $V_{ref}$  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  $V_{ref}$ .

$V_{ref} (\text{V})$	$V_{in}$ in Range (V)	Step Size (mV)
<b>5.00</b>	0 to 5	$5 / 256 = 19.53$
<b>4.00</b>	0 to 4	$4 / 256 = 15.62$
<b>3.00</b>	0 to 3	$3 / 256 = 11.71$
<b>2.56</b>	0 to 2.56	$2.56 / 256 = 10$
<b>2.00</b>	0 to 2	$2 / 256 = 7.81$
<b>1.28</b>	0 to 1.28	$1.28 / 256 = 5$
<b>1.00</b>	0 to 1	$1 / 256 = 3.90$
<b>Note: In an 8-bit ADC, step size is <math>V_{ref}/256</math></b>		

Table 7-2:  $V_{ref}$  Relation to  $V_{in}$  Range for an 8-bit ADC

Vref (V)	V <sub>in</sub> Range (V)	Step Size (mV)
<b>5.00</b>	0 to 5	5 / 1024 = 4.88
<b>4.96</b>	0 to 4.096	4.096 / 1024 = 4
<b>3.00</b>	0 to 3	3 / 1024 = 2.93
<b>2.56</b>	0 to 2.56	2.56 / 1024 = 2.5
<b>2.00</b>	0 to 2	2 / 1024 = 2
<b>1.28</b>	0 to 1.28	1.28 / 1024 = 1.25
<b>1.024</b>	0 to 1.024	1.024 / 1024 = 1

*Note: In a 10-bit ADC, step size is V<sub>ref</sub>/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.

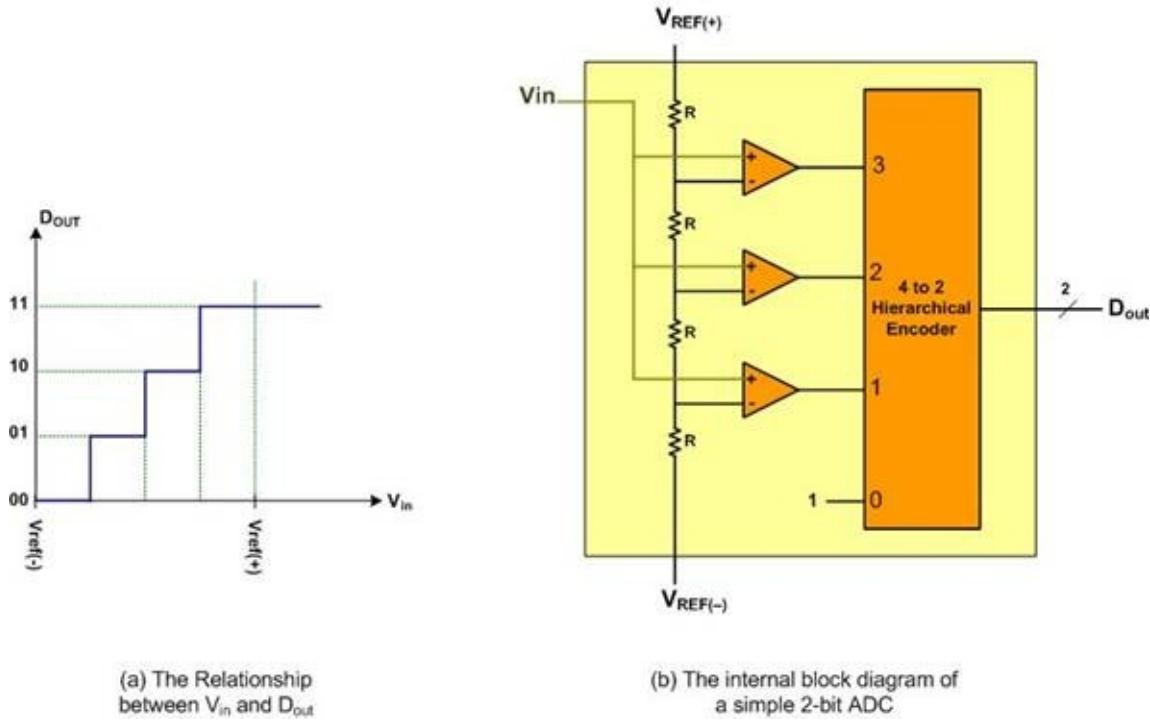
## Digital data output

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:

$$D_{OUT} = V_{IN} / \text{StepSize}$$

where  $D_{out}$  = digital data output (in decimal),  $V_{in}$  = analog input voltage, and step size (resolution) is the smallest change, which is  $V_{ref}/256$  for an 8-bit ADC.

Figure 7-3 shows a simple 2-bit ADC. In the circuit, the voltage between V<sub>ref(+)</sub> and V<sub>ref(-)</sub> is divided into 4 since resistors have the same values. As a result, the step size is  $(V_{ref(+)} - V_{ref(-)}) / 4$ .



**Figure 7-3: A Simultaneous 2-bit ADC**

If  $V_{in}$  is below step size all the comparators send out zeros. When  $V_{in}$  is between step size and step size  $\times 2$ , the lowest comparator sends out 1 and the encoder gives 01.

If  $V_{in}$  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  $V_{in}$  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.

### Example 7-1

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, respectively.

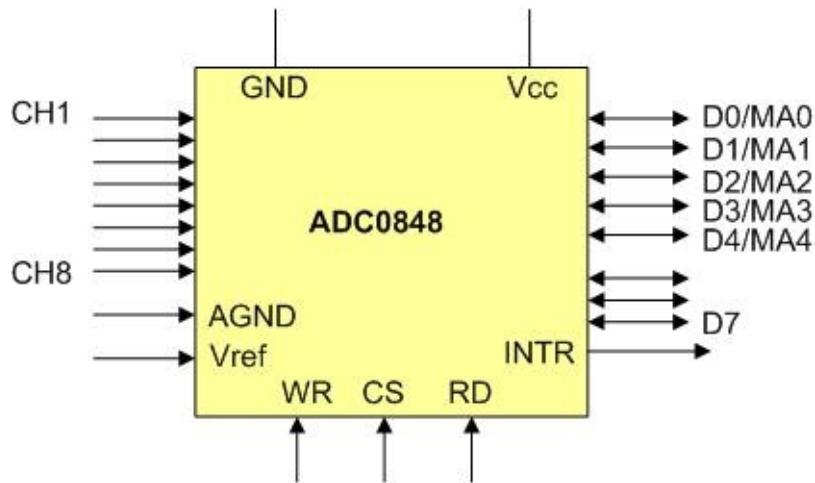


Figure 7-4: ADC0848 Parallel ADC Block Diagram

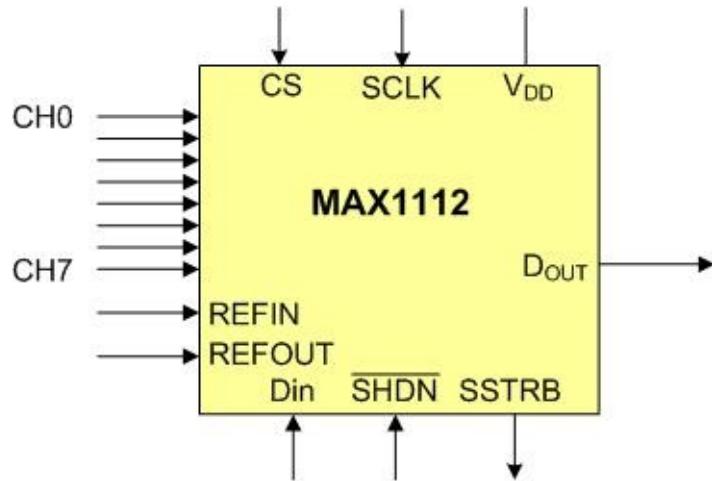


Figure 7-5: MAX1112 Serial ADC Block Diagram

## 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.

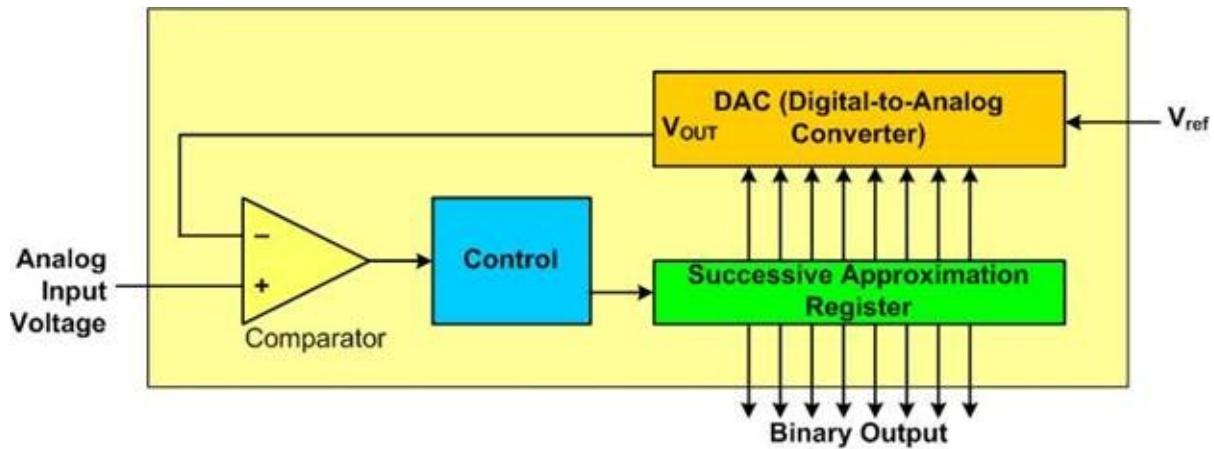


Figure 7-6: Successive Approximation ADC

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 Vref = 1.28 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 Freescale KL25Z

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 Freescale KL25Z and show how it is programmed.

The Freescale KL25Z ARM chip has a single ADC module which can support up to 31 ADC channels. These ADC channels have 16-bit resolution. To program them, we need to understand some of the major registers. Figure 7-7 shows a simplified block diagram of a KL25Z chip.

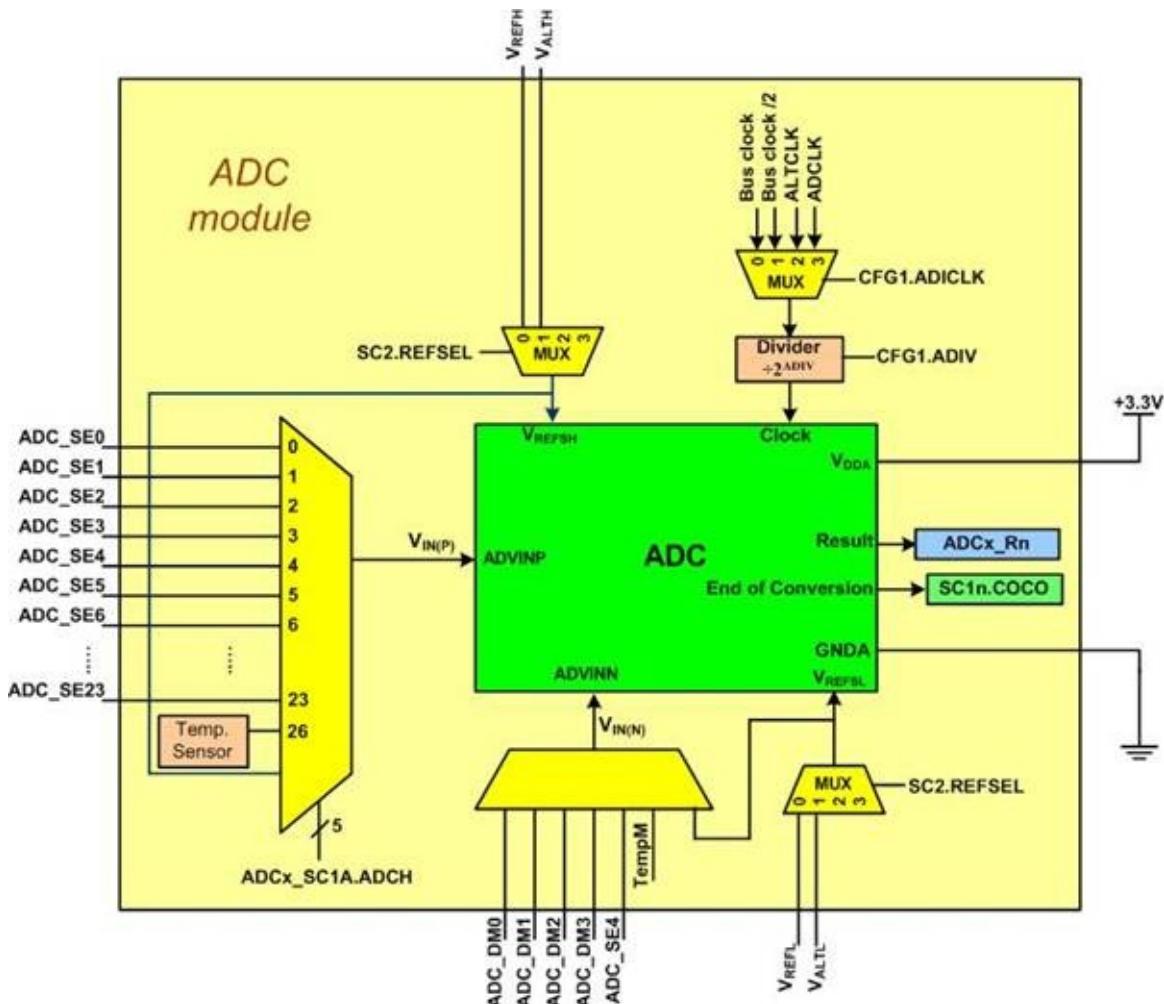


Figure 7-7: Simplified Block Diagram of a Freescale KL25Z chip

In this section, we examine some of these registers and show how to program the ADC. Below is some major registers of KL25Z ADC from KL25Z reference manual.

Absolute Address	Register
4003 B000	ADC Status and Control Registers 1 (ADC0_SC1A)
	ADC Status and Control Registers 1

4003 B004	(ADC0_SC1B)
4003 B008	ADC Configuration Register 1 (ADC0_CFG1)
4003 B00C	ADC Configuration Register 2 (ADC0_CFG2)
4003 B010	ADC Data Result Register (ADC0_RA)
4003 B014	ADC Data Result Register (ADC0_RB)

Table 7-4: KL25Z ADC Registers

## Enabling Clock to ADC

First thing we need to do is to enable the clock to the ADC0 module. Bit D27 of SIM\_SCGC6 register is used to enable the clock to ADC0 module. The SIM\_SCGC6 is part of the System Integration Module and located at physical address  $0x4004\ 7000 + 103C = 0x4000\ 803C$ . See Figure 7-8.

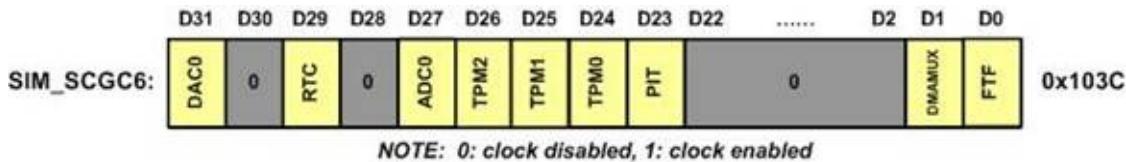


Figure 7-8: SIM\_SCGC6 register t for Enableing to ADC0

## Start Conversion trigger options

There are two start-conversion (trigger) options. They are hardware trigger and software trigger. The selection of hardware or software trigger for conversion is done via the bit D6 (ADTRG, ADC Trigger) of ADC0\_SC2 (ADC0 Status Control 2) register. The hardware trigger may be external pin, comparator, or timers (TPMx, LPTMR0, PIT, or RTC). The selection of hardware trigger is done in SIM\_SOPT7 register. The default trigger is software and that is what we will use in this section.

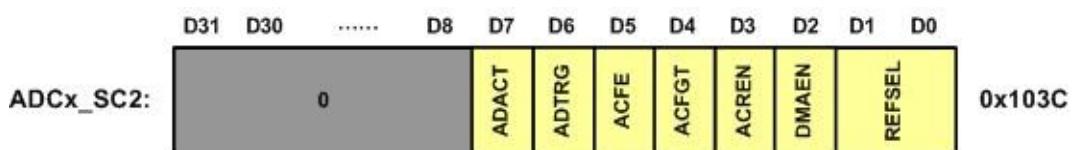


Figure 7-9: ADCx\_SC2 Register

Bit	Field	Descriptions
7	ADACT	Conversion active: Indicates that the ADC is converting data (0: Conversion not in progress, 1: Conversion in progress)
6	ADTRG	ADC conversion trigger select (0: software trigger, 1: hardware trigger)

5	ACFE	Compare Function Enable (0: compare function disabled, 1: enabled)
4	ACFGT	Compare Function Greater Than Enable
3	ACREN	Compare range Enable
2	DMAEN	DMA Enable
Voltage Reference Select		
1-0	REFSEL	<b>REFSEL</b> <b>Voltage Reference</b>
		<b>00</b> The VREFH and VREFL pins are used as VREF(+) and VREF(-), respectively.
		<b>01</b> VALTH and VALTL pair is used as references.
		<b>Others</b> Reserved

Table 7-5: ADCx\_SC2 Register

## Choosing $V_{in}$ input channel

The channel selection is done through the ADC0\_SC1A (ADC Status and Control 1A) register. (There are more ADC0\_SC1n registers but only ADC0\_SC1A can be used for software trigger. The other registers are for hardware trigger only and will not be discussed here.) The lowest 5 bits of ADC0\_SC1A register are used to select one of the 31 single-ended channels to be converted. See Figure 7-10.



Figure 7-10: ADCx\_SC1A Register

Bit	Field	Descriptions
7	COCO	Conversion Complete Flag: (0: Conversion is not completed, 1: Conversion is completed)  The COCO is cleared when the ADCx_SC1n register is written or the ADCx_Rn register is read.
6	AIEN	Interrupt Enable: The ADC interrupt is enabled by setting the bit to HIGH. If the interrupt enable is set, an

interrupt is triggered when the COCO flag is set.

5	DIFF	Differential mode (0: Single-ended mode, 1: Differential mode)
4-0	ADCH	<p>ADC input channel: The field selects the input channel as shown in Figure 7-7.</p> <p>When DIFF = 0 (single-ended mode), values 0 to 23 choose between the 24 input channels (ADC_SE0 to ADC_SE23).</p> <p>When DIFF = 1 (Differential mode), values 0 to 3 select between the 4 differential channels. See the reference manual for more information.</p> <p>When ADCH = 1111, the module is disabled.</p>

**Table 7-6: ADCx\_SC1 Register**

Not all the channels are connected to the input pins. The number of available channels in the Freescale KL25Z varies among the family members. In the case of KL25Z128VLK4 ARM chip used in FRDM board, there are 14 channels connected to the input pins and additional 4 channels are connected internally. See Table 7-7.

Pin Name	Description	Pin
ADC_SE0	ADC input 0	PTE20
ADC_SE3	ADC input 3	PTE22
ADC_SE4	ADC input 4	PTE21, PTE29
ADC_SE5	ADC input 5	PTD1
ADC_SE6	ADC input 6	PTD5
ADC_SE7	ADC input 7	PTD6, PTE23
ADC_SE8	ADC input 8	PTB0
ADC_SE9	ADC input 9	PTB1
ADC_SE11	ADC input 11	PTC2
ADC_SE12	ADC input 12	PTB2
ADC_SE13	ADC input 13	PTB3
ADC_SE14	ADC input 14	PTC0

<b>ADC_SE15</b>	ADC input 15	PTC1
<b>ADC_SE23</b>	ADC input 23, DAC0 output	PTE30
<b>ADC_SE26</b>	Temperature sensor	
<b>ADC_SE27</b>	Bandgap reference	
<b>ADC_SE29</b>	$V_{REFH}$	
<b>ADC_SE30</b>	$V_{REFL}$	
<b>ADC_SE31</b>	Module disabled	

**Table 7-7: Analog input pin assignment in Freescale KL25Z**

## Polling or interrupt

The end-of-conversion is indicated by a flag bit in the ADC0\_SC1A (ADC Status Control 1A) register. Upon the completion of conversion, the D7 bit Conversion Complete (COCO) flag goes high. By polling this flag, we know if the conversion is complete and we can read the value in ADC0\_R0 data result register. We can also use an interrupt to inform us that the conversion is complete but that will require us to set the AIEN (Interrupt Enable) bit (bit 6) high in ADC0\_SC1A register. By default, the interrupt is not enabled.

## ADC Data result

Upon the completion of conversion, the binary result is placed in the ADC0\_RA register. (There are many ADC0\_Rn registers corresponding to the ADC0\_SC1n registers. Because we can only use ADC0\_SC1A for software trigger, the data will be in ADC0\_RA register. This is a 32-bit register but only the lower 16 bits are used. For the ADC, we have the options of 8-, 10-, 12-, and 16-bit for single-ended unsigned result. In any case, always the result is right-justified and the rest of the bits up to bit D15 are unused. If the result is in 2's complement for differential, then it is signed-extended to bit D15. For the concept of sign-extend, see the "ARM Assembly Language Programming" volume in this series.



**Figure 7-11: ADC Result Reister (ADCx\_Rn) See Table 28-43 in KL25Z Ref man**

## Clearing conversion complete flag

The conversion complete (COCO) flag bit in ADCx\_SC1n register is cleared automatically when the data from the respective ADCx\_Rn register is read.

## 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 KL25Z has the option of converting the differential voltages of two analog channels. The bit D5 (DIFF) of ADC0\_SC1A register allows us to enable the differential option. Upon Reset, the default is the single-ended input and we will leave it at that for the discussion here. See the KL25Z reference manual for further information on differential options.

## Selection Bit Resolution

We use ADCx\_CFG1 (ADC configuration 1) register to select 8, 10, 12, or 16-bit ADC resolution. This register is also used to select the speed of the clock source to the ADC.

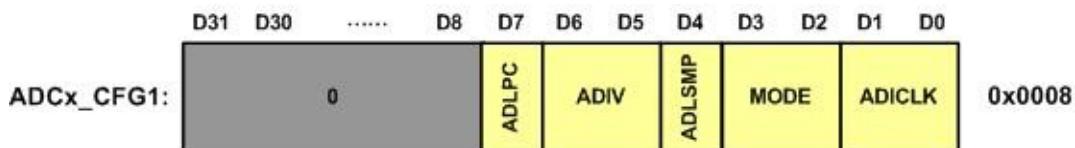


Figure 7-12: ADCx\_CFG1 Register

Bit	Field	Descriptions															
7	ADLPC	Low-Power Configuration															
6-5	ADIV	Clock Divide Select: The clock is divided by $2^{ADIV}$ as shown in Figure 7-7.															
4	ADLSMP	Sample time configuration (0: Short sample time, 1: Long sample time)															
3-2	MODE	<p>Conversion mode selection</p> <table border="1"><thead><tr><th>MODE</th><th>In single-ended mode (DIFF = 0)</th><th>In differential mode (DIFF = 1)</th></tr></thead><tbody><tr><td>00</td><td>8-bit conversion</td><td>9-bit conversion with 2's complement output</td></tr><tr><td>01</td><td>12-bit conversion</td><td>13-bit conversion with 2's complement output</td></tr><tr><td>10</td><td>10-bit conversion</td><td>11-bit conversion with 2's complement output</td></tr><tr><td>11</td><td>16-bit conversion</td><td>16-bit conversion with 2's complement output</td></tr></tbody></table>	MODE	In single-ended mode (DIFF = 0)	In differential mode (DIFF = 1)	00	8-bit conversion	9-bit conversion with 2's complement output	01	12-bit conversion	13-bit conversion with 2's complement output	10	10-bit conversion	11-bit conversion with 2's complement output	11	16-bit conversion	16-bit conversion with 2's complement output
MODE	In single-ended mode (DIFF = 0)	In differential mode (DIFF = 1)															
00	8-bit conversion	9-bit conversion with 2's complement output															
01	12-bit conversion	13-bit conversion with 2's complement output															
10	10-bit conversion	11-bit conversion with 2's complement output															
11	16-bit conversion	16-bit conversion with 2's complement output															

Input Clock Select		
	ADICLK	Clock source
1-0	00	Bus clock
	01	(Bus clock)/2
	10	Alternate clock (ALTCLK)
	11	Asynchronous clock (ADACK)

Table 7-8: ADCx\_CFG1 Register

Notice in ADCx\_CFG1, the MODE bits (D3:D2) selects the resolution. Also notice, if we use the Low Power option with D7 bit, then the conversion speed is limited.

## ADC Conversion Time

The conversion time for the ADC has three parts. They are as follows:

- 1) In the first phase, a sample amplifier of unity gain samples the analog input for a total of n clock cycles. This buffering of the analog input charges the sample capacitor up to the input potential.
- 2) In the second phase, the sample buffer is disconnected and connected to the storage node for a certain number of clock cycles. The number of clock cycles can be 4, 6, 10, 16, or 24. We program this number via the ADLSMP bit in ADCx\_CFG1 register and the ADLSTS bits in ADCx\_CFG2 register. Longer sampling time ensures that the voltage of the sample capacitor is brought closer to the input voltage. This is important when the input voltage differs significantly from sample to sample. But it prolongs the conversion time of each sample.
- 3) In the third phase, the analog input is converted to binary numbers using the successive approximation method. In this phase, the number of clocks used depends on how many bits are in the binary output. For each bit we need one clock. That means we need 8 clocks for the 8-bit output, 10 clocks for the 10-bit output, and so on. We choose the  $n$ -bit resolution option using the MODE bits of ADCx\_CFG1 register.

The ADCx\_CFG1 gives us many options for the clock fed to ADC module. We can choose the Bus Clock or a fraction of it. Using the ADICLK (AD input Clock) and ADIV (AD divide) bits of ADCx\_CFG1 registers we can control the speed of clock source fed to the ADC. These bits along with the bits in ADCx\_CFG2, we can control the conversion time.

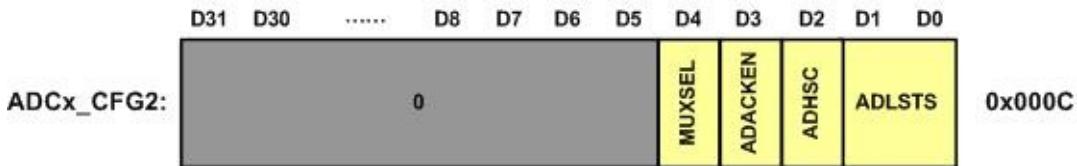


Figure 7-13: ADCx\_CFG2 Register

## V<sub>ref</sub> in FRDM board

In the Freescale ARM KL25Z chip, the pin for V<sub>ref</sub> (+) is called VREFH (Vref High) and V<sub>ref</sub> (-) pin is called VREFL (Vref Low). In the FRDM board, the VREFH pin is connected to 3.3V, the same supply voltage as the digital part of the chip. The circuit may be altered to use the AREF pin for an external reference voltage. Even if we connect the VREFH to an external reference other than the VDD of the chip, it cannot go beyond the VDD voltage. With VREFH=3.3V, we have the step size of  $3.3V / 65,536 = 0.05\text{ mV}$  since the maximum ADC resolution for KL25Z is 16 bits. See Example 7-2.

### Example 7-2

Give the digital converted output if the analog input voltage is 1.2V for the Freescale FRDM board.

#### Solution:

Since the step size is  $3.3V / 65,536 = 0.05\text{ mV}$ , we have  $1.2V / 0.05\text{ mV} = 23,831 = 0x5D17$  as ADC output.

## Configuring ADC and reading ADC channel

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. We need to take the following steps to configure the ADC:

1. Enable the clock to I/O pin used by the ADC channel. Table 7-7 shows the I/O pins used by various ADC channels.
2. Set the PORTX\_PCRn MUX bit for ADC input pin to 0 to use the pin for analog input channel. This is actually the power-on default.
3. Enable the clock to ADC0 modules using SIM\_SCGC6 register.
4. Choose the software trigger using the ADC0\_SC2 register.
5. Choose clock rate and resolution using ADC0\_CFG1 register.
6. Select the ADC input channel using the ADC0\_SC1A register. Make sure to use Table 7-7 to choose the right pin and channel. Also makes sure interrupt is not enabled and single-ended option is used when you

select the channel with this register.

7. Keep monitoring the end-of-conversion COCO flag in ADC0\_SC1A register.
8. When the COCO flag goes HIGH, read the ADC result from the ADC0\_RA and save it.
9. Repeat steps 6 through 8 for the next conversion.

Program 7-1 illustrates the steps for ADC conversion shown above. Figure 7-14 shows the hardware connection of Program 7-1.

#### Program 7-1: Using ADC0 to convert input from channel 0

```
/* p7_1.c: A to D conversion of channel 0

 * This program converts the analog input from channel 0 (PTE20)
 * using software trigger continuously.
 * Bits 10-8 are used to control the tri-color LEDs. LED code is
 * copied from p2_7. Connect a potentiometer between 3.3V and
 * ground. The wiper of the potentiometer is connected to PTE20.
 * When the potentiometer is turned, the LEDs should change color.
 */

#include "MKL25Z4.h"

void ADC0_init(void);
void LED_set(int s);
void LED_init(void);

int main (void)
{
    short int result;
    LED_init();           /* Configure LEDs */
    ADC0_init();          /* Configure ADC0 */
    while (1) {
        ADC0->SC1[0] = 0;          /* start conversion on channel 0 */
        while (!(ADC0->SC1[0] & 0x80)) { } /* wait for conversion complete */
        result = ADC0->R[0];       /* read conversion result and clear COCO flag */
        LED_set(result >> 7);     /* display result on LED */
    }
}
```

```

    }

}

void ADC0_init(void)
{
    SIM->SCGC5 |= 0x2000;           /* clock to PORTE */
    PORTE->PCR[20] = 0;           /* PTE20 analog input */

    SIM->SCGC6 |= 0x8000000;      /* clock to ADC0 */
    ADC0->SC2 &= ~0x40;          /* software trigger */
    /* clock div by 4, long sample time, single ended 12 bit, bus clock */
    ADC0->CFG1 = 0x40 | 0x10 | 0x04 | 0x00;
}

void LED_init(void) {
    SIM->SCGC5 |= 0x400;          /* enable clock to Port B */
    SIM->SCGC5 |= 0x1000;         /* enable clock to Port D */
    PORTB->PCR[18] = 0x100;       /* make PTB18 pin as GPIO */
    PTB->PDDR |= 0x40000;        /* make PTB18 as output pin */
    PORTB->PCR[19] = 0x100;       /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0x80000;        /* make PTB19 as output pin */
    PORTD->PCR[1] = 0x100;        /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;           /* make PTD1 as output pin */
}

void LED_set(int s) {
    if (s & 1)      /* use bit 0 of s to control red LED */
        PTB->PCOR = 0x40000;    /* turn on red LED */
    else
        PTB->PSOR = 0x40000;    /* turn off red LED */
    if (s & 2)      /* use bit 1 of s to control green LED */
        PTB->PCOR = 0x80000;    /* turn on green LED */
    else
        PTB->PSOR = 0x80000;    /* turn off green LED */
    if (s & 4)      /* use bit 2 of s to control blue LED */
        PTD->PCOR = 0x02;       /* turn on blue LED */
    else
        PTD->PSOR = 0x02;       /* turn off blue LED */
}

```

}

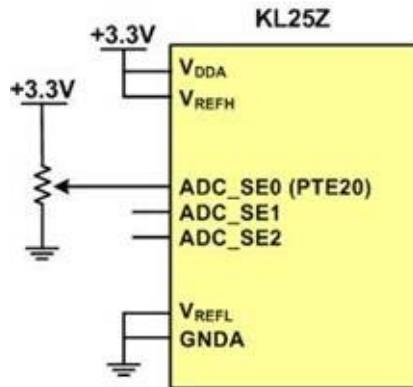


Figure 7-14: ADC Connection for Program 7-1

## Temperature sensor

There are several internal analog channels as seen in Table 7-7. The next program shows how to convert the internal temperature sensor output. In the ADC initialization, there is no need to initialize the input pin because the channel is connected internally.

### Program 7-2: Converting the on-chip temperature sensor with timer trigger

```
/* p7_2.c: A to D conversion of internal temperature

 * This program converts the analog input from channel 26
 * (internal temperature sensor) using software trigger continuously.

 * Bits 2-0 are used to control the tri-color LEDs. LED code is
 * copied from p2_7. Put your finger on the target MCU and watch
 * LEDs change color.

 */

#include "MKL25Z4.h"

void ADC0_init(void);
void LED_set(int s);
void LED_init(void);

int main (void)
```

```

{

    short int result;

    LED_init();                      /* Configure LEDs */

    ADC0_init();                     /* Configure ADC0 */

    while (1) {

        ADC0->SC1[0] = 26;           /* start conversion on channel 26 temperature */

        while (!(ADC0->SC1[0] & 0x80)) { } /* wait for COCO */

        result = ADC0->R[0];          /* read conversion result and clear COCO flag */

        LED_set(result);             /* display result on LED */

    }

}

void ADC0_init(void)

{

    SIM->SCGC6 |= 0x8000000;      /* clock to ADC0 */

    ADC0->SC2 &= ~0x40;           /* software trigger */

    /* clock div by 4, long sample time, single ended 12 bit, bus clock */

    ADC0->CFG1 = 0x40 | 0x10 | 0x04 | 0x00;

}

void LED_init(void) {

    SIM->SCGC5 |= 0x400;           /* enable clock to Port B */

    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */

    PORTB->PCR[18] = 0x100;         /* make PTB18 pin as GPIO */

    PTB->PDDR |= 0x40000;          /* make PTB18 as output pin */

    PORTB->PCR[19] = 0x100;         /* make PTB19 pin as GPIO */

    PTB->PDDR |= 0x80000;          /* make PTB19 as output pin */

    PORTD->PCR[1] = 0x100;          /* make PTD1 pin as GPIO */

    PTD->PDDR |= 0x02;             /* make PTD1 as output pin */

}

void LED_set(int s) {

    if (s & 1) /* use bit 0 of s to control red LED */

        PTB->PCOR = 0x40000;        /* turn on red LED */

    else

        PTB->PSOR = 0x40000;        /* turn off red LED */

    if (s & 2) /* use bit 1 of s to control green LED */

        PTB->PCOR = 0x80000;        /* turn on green LED */
}

```

```

    else
        PTB->PSOR = 0x80000; /* turn off green LED */

    if (s & 4) /* use bit 2 of s to control blue LED */
        PTD->PCOR = 0x02; /* turn on blue LED */
    else
        PTD->PSOR = 0x02; /* turn off blue LED */
}

```

## Analog Comparator unit

Many microcontrollers come with analog comparator for monitoring an analog input voltage. Using the analog comparator, we can monitor an analog input against two threshold values to determine whether the analog input value is above the high threshold, between the two threshold values, or below the 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 analog 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 handle the interrupt. It does not have to use the ADC to monitor the temperature continuously.

For the details of programming the analog comparator unit, we will leave it to the reader.

## Review Questions

1. The ADC in Freescale ARM KL25Z is \_\_\_\_\_ bit.
2. In KL25Z, the highest number we can get for the ADC output is \_\_\_\_\_ in hex.
3. Assume VREFH = 3.3V. Find the ADC output in decimal and hex if  $V_{in}$  of analog input is 1.9V. Assume 10-bit resolution.
4. In ARM KL25Z, which register provides the ADC output converted data?
5. In KL25Z, we have resolution choices of \_\_\_\_\_.



## 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 7-9 and Figure 7-15.

Temperature ('C)	T <sub>f</sub> (K ohms)
0	29.490
25	10.000
50	3.893
75	1.700
100	0.817

Table 7-9: Thermistor Resistance vs. Temperature

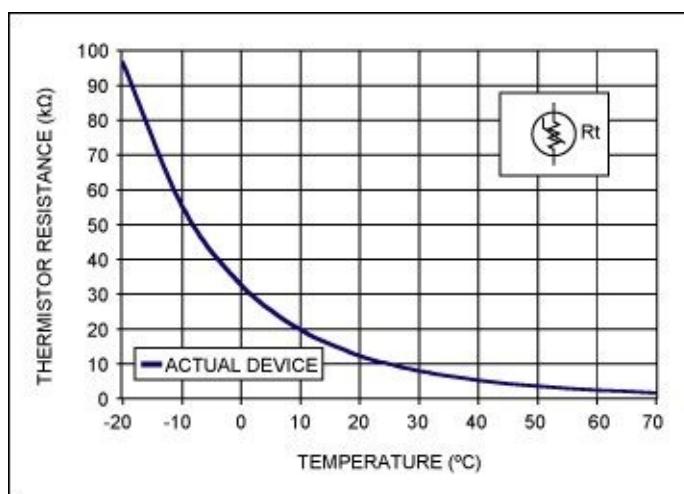


Figure 7-15: Thermistor (Copied from <http://www.maximintegrated.com>)

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 7-16. The LM34 requires no external calibration because it is internally calibrated. It outputs 10 mV for each degree of Fahrenheit temperature.

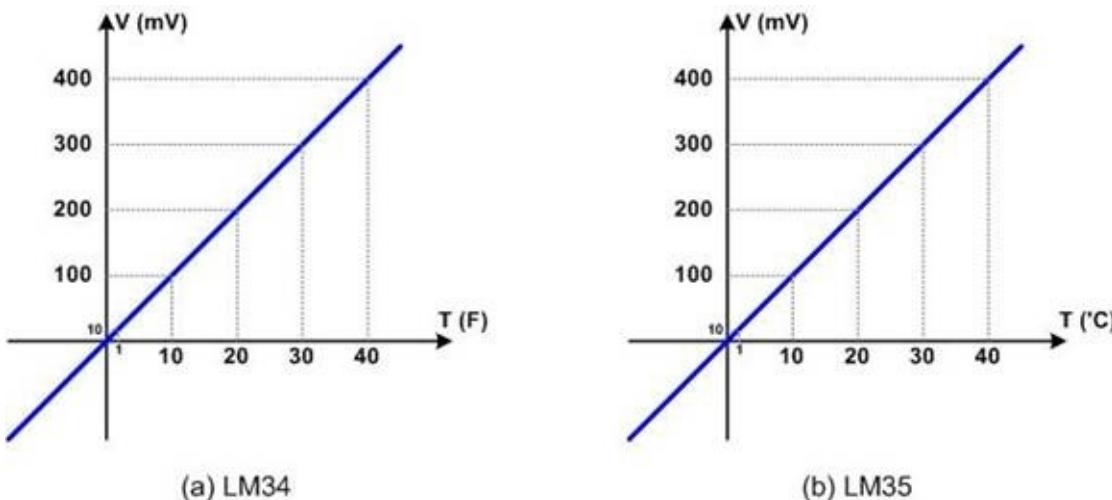


Figure 7-16: LM34 and LM35

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 7-16.

## 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 7-17 shows a block diagram of the input of a data acquisition system.

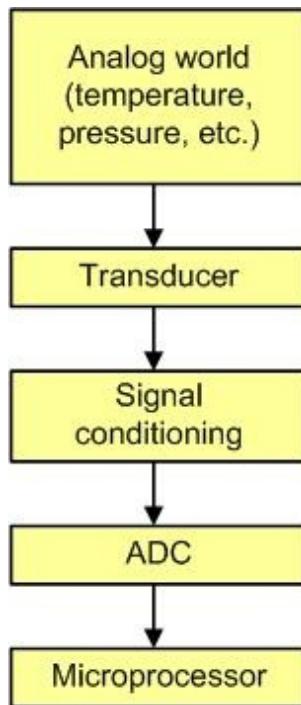


Figure 7-17: Getting Data to the CPU

## Interfacing the LM34 to the Freescale ARM Microcontroller

The A/D of Freescale ARM Microcontroller has 16-bit resolution with a maximum of 65,536 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 7-18.

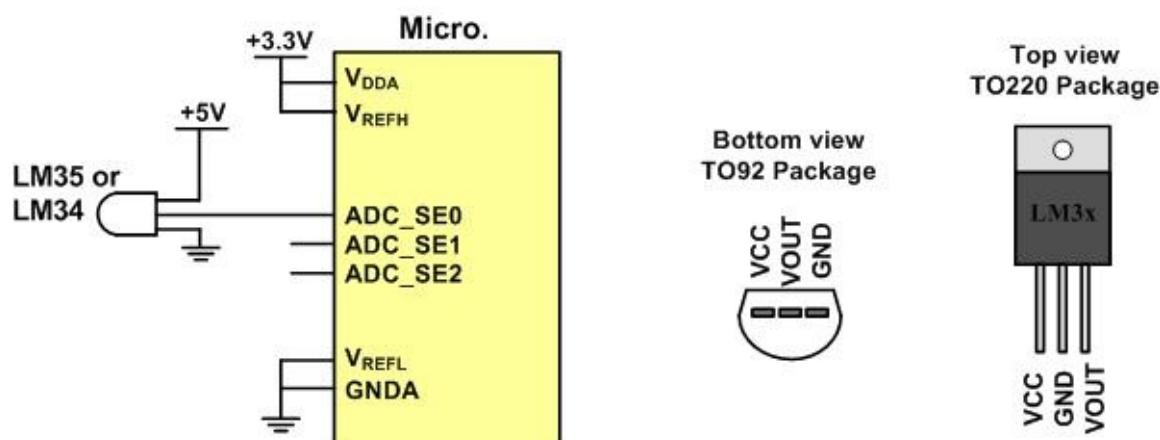


Figure 7-18: LM34/35 Connection to ARM and Its Pin Configuration

To convert the ADC result to temperature in degree, use the following equation:

$$\text{temperature} = \text{result} * 330.0 / 65536;$$

## Reading and displaying temperature

Programs 7-3 shows code for reading and displaying temperature in C.

Notice that in Figure 7-18, the LM34 (or LM35) is connected to channel 0 (ADC0 pin). 16-bit conversion is used to enhance the precision. At 16-bit, the noise is much more apparent so we turn on the hardware averaging to reduce the output fluctuation.

### Program 7-3: Reading Temperature Sensor in F

```
/* p7_3.c: A to D conversion of channel 0

 * This program converts the analog input from channel 0 (PTE20)
 * using software trigger continuously. PTE20 is connected to
 * an LM34 Fahrenheit temperature sensor. The conversion result
 * is displayed as temperature through UART0 virtual serial port.
 * 16-bit precision is used for conversion. At higher precision,
 * the noise is more significant so 32 samples averaging is used.
 * The LM34 output voltage is 10mV/degreeF. The ADC of FRDM-KL25Z
 * uses 3.3V as Vref so:
 * temperature = result * 330.0 / 65536
 * Open a terminal emulator at 115200 Baud rate at the host PC and
 * observe the output.

 */

#include "MKL25Z4.h"
#include <stdio.h>

void ADC0_init(void);
void delayMs(int n);
void UART0_init(void);
void UART0Tx(char c);
void UART0_puts(char* s);

int main (void) {
    int result;
    float temperature;
    char buffer[16];

    ADC0_init();          /* Configure ADC0 */
    UART0_init();         /* initialize UART0 for output */
    while (1) {
        ADC0->SC1[0] = 0;      /* start conversion on channel 0 */
        while (!(ADC0->SC1[0] & 0x80)) { } /* wait for COCO */
```

```

    result = ADC0->R[0];           /* read conversion result and clear COCO flag */
    temperature = result * 330.0 / 65536; /* convert voltage to temperature */
    sprintf(buffer, "\r\nTemp = %6.2fF", temperature); /* convert to string */
    UART0_puts(buffer);           /* send the string through UART0 for display */
    delayMs(1000);
}

void ADC0_init(void)
{
    SIM->SCGC5 |= 0x2000;          /* clock to PORTE */
    PORTE->PCR[20] = 0;           /* PTE20 analog input */

    SIM->SCGC6 |= 0x8000000;      /* clock to ADC0 */
    ADC0->SC2 &= ~0x40;           /* software trigger */
    ADC0->SC3 |= 0x07;            /* 32 samples average */
    /* clock div by 4, long sample time, single ended 16 bit, bus clock */
    ADC0->CFG1 = 0x40 | 0x10 | 0x0C | 0x00;
}

/* initialize UART0 to transmit at 115200 Baud */
void UART0_init(void) {
    SIM->SCGC4 = 0x0400;          /* enable clock for UART0 */
    SIM->SOPT2 = 0x04000000;      /* use FLL output for UART Baud rate generator */
    UART0->C2 = 0;                /* turn off UART0 while changing configurations */
    UART0->BDH = 0x00;
    UART0->BDL = 0x17;            /* 115200 Baud */
    UART0->C4 = 0x0F;              /* Over Sampling Ratio 16 */
    UART0->C1 = 0x00;              /* 8-bit data */
    UART0->C2 = 0x08;              /* enable transmit */

    SIM->SCGC5 = 0x0200;          /* enable clock for PORTA */
    PORTA->PCR[2] = 0x0200; /* make PTA2 UART0_Tx pin */
}

void UART0Tx(char c) {
    while(!(UART0->S1 & 0x80)) {
        /* wait for transmit buffer empty */
        UART0->D = c; /* send a char */
}

```

```

void UART0_puts(char* s) {
    while (*s != 0)           /* if not end of string */
        UART0Tx(*s++);      /* send the character through UART0 */
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
}

```

## Review Questions

- True or false. The transducer must be connected to signal conditioning circuitry before its signal is sent to the ADC.
- The LM35 provides \_\_\_\_ mV for each degree of \_\_\_\_\_ (Fahrenheit, Celsius) temperature.
- The LM34 provides \_\_\_\_ mV for each degree of \_\_\_\_\_ (Fahrenheit, Celsius) temperature.
- What is the temperature if the ADC output is 0000 0011 1110?



## Section 7.4: DAC Programming

This section will discuss the fundamentals of a DAC (digital-to-analog converter). Then we demonstrate how to generate a sawtooth wave and a sine wave. The analog conversion output can be observed on the oscilloscope at DAC output pin PTE30.

### 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 discussed 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 7-19.

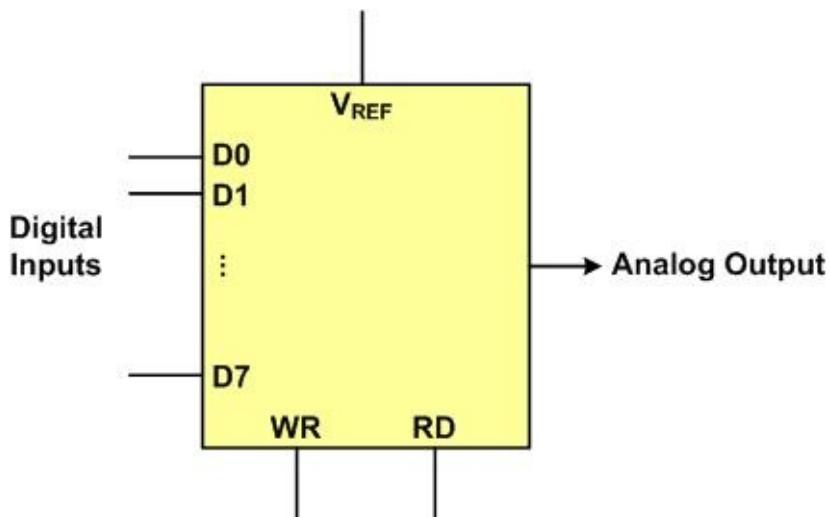


Figure 7-19: DAC Block Diagram

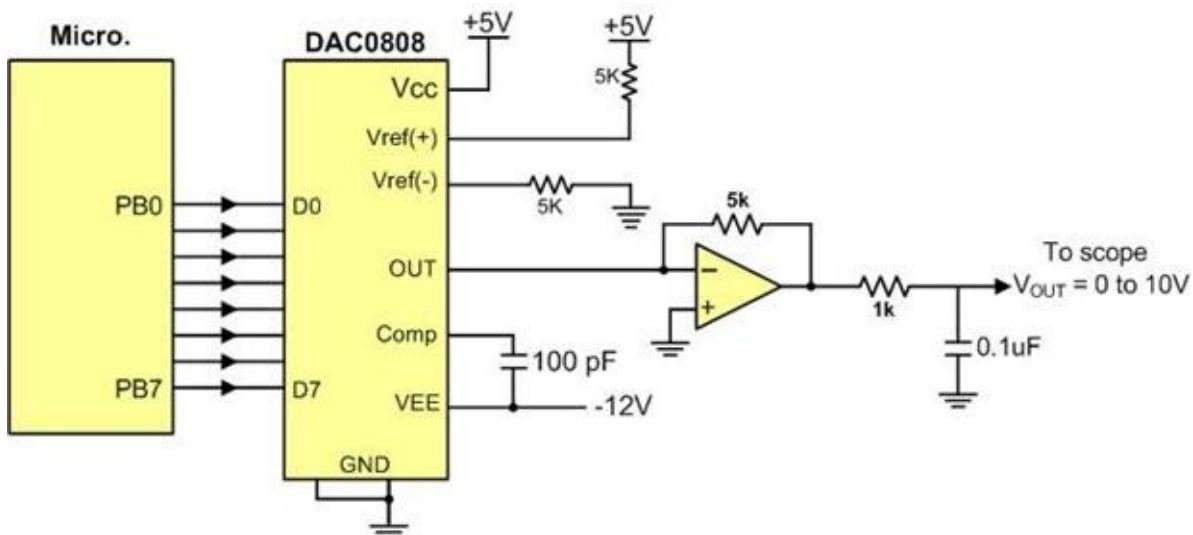
Similarly, the 12-bit DAC provides 4096 discrete voltage levels. Although there are 16-bit DACs, they are much more expensive.

### 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 7-20 shows the generation of current reference (setting  $I_{ref} = 2$  mA) by using the standard 5-V power supply and 5K ohm resistors.



**Figure 7-20: 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 7-3.

### Example 7-3

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)  $I_{out} = 2 \text{ mA} (153/255) = 1.195 \text{ mA}$  and  $V_{out} = 1.195 \text{ mA} \times 5\text{K} = 5.975 \text{ V}$   
(b)  $I_{out} = 2 \text{ mA} (200/256) = 1.562 \text{ mA}$  and  $V_{out} = 1.562 \text{ mA} \times 5\text{K} = 7.8125 \text{ V}$
- 

## DAC Features of KL25Z

The Freescale KL25Z comes with an on-chip DAC. The on-chip DAC is 12-bit string converter. A string DAC uses a resistor ladder and an analog multiplexer. See Figure 7-21.

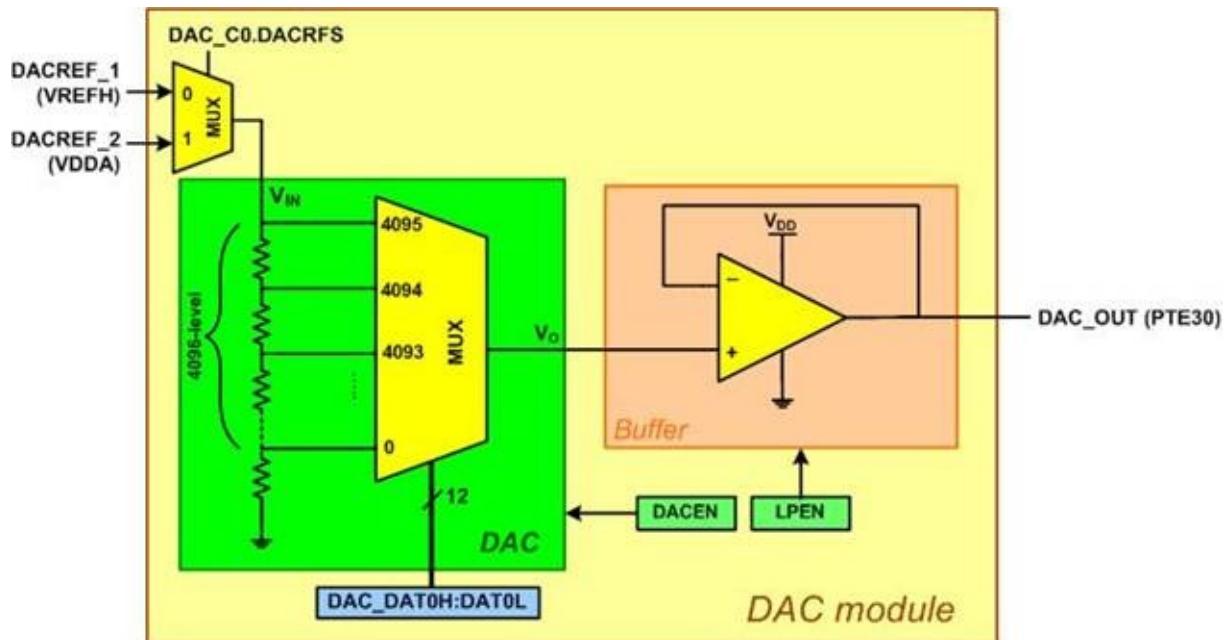


Figure 7-21: Simplified Block Diagram of KL25Z on-chip DAC

Below is a list of the major registers of KL25Z DAC from KL25Z ref. manual.

Absolute Address	Register
4003 F000	DAC Data Low Register (DAC0_DAT0L)
4003 F001	DAC Data High Register (DAC0_DAT0H)
4003 F002	DAC Data Low Register (DAC0_DAT1L)
4003 F003	DAC Data High Register (DAC0_DAT1H)
4003 F020	DAC Status Register (DAC0_SR)
4003 F021	DAC Control Register (DAC0_C0)

Table 7-10: DAC Registers

Next, we will examine some of the registers and use them to create

staircase ramp and sine wave signals. First, we must provide clock to the DAC. This is done with SIM\_SCGC6 register. In Figure 7-22, notice bit D31 is used to provide the clock to on-chip DAC0.

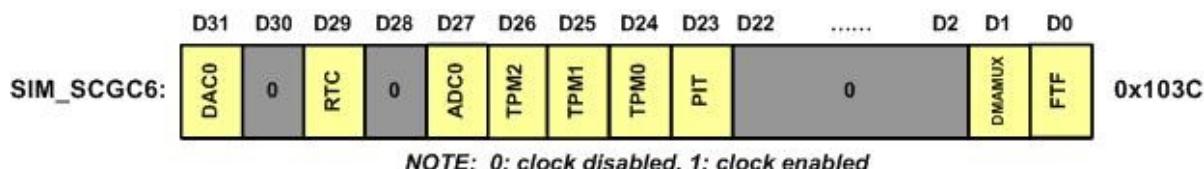


Figure 7-22: SIM\_SCGC6 register to provide clock to DAC

## DAC Control 0 register

We also need to enable the DAC itself before we can use it. This is done with bit D7 of DAC Control 0 (DAC0\_C0) register.



Figure 7-23: DAC0\_C0 Register to enable on-chip DAC

Bit	Field	Descriptions
7	DACEN	DAC Enable (0: DAC is disabled, 1: DAC is enabled)
6	DACRFS	DAC Reference Select (0: DACREF_1, 1: DACREF_2)
5	DACTRGSel	DAC Trigger Select (0: hardware trigger, 1: software trigger)
4	DACSWTRG	DAC Software Trigger
3	LPEN	DAC Low Power Control (0: High-Power mode, 1: Low-Power mode)
1	DACBTIEN	DAC Buffer read pointer Top flag Interrupt Enable
0	DACBBIEN	DAC Buffer read pointer Bottom flag Interrupt Enable

Table 7-11: DAC0\_C0 Register to enable on-chip DAC

Other important bits in DAC\_C0 registers are D5 (DACTRGSel: DAC Trigger select) and D6 (DAC0\_DACRFS: DAC Reference Select). The choices of reference voltages are DACREF\_1 for VREFH and DACREF\_2 for VDDA. In the FRDM\_KL25Z board, VREFH is tied to VDDA so there is no difference in using either reference. We will use software trigger.

## DAC Buffer register

The data registers DAC0\_DATnH:DAC0\_DATnH hold the 12-bit digital (binary) value needed to be converted to the analog. The DAC0\_DATnL register is used for the lower 8 bits and DAC0\_DATnH for the upper 4 bits of the 12-bit binary value. See Figures 7-24 and 7-25. For KL25Z, there are two sets of buffer data registers, DAC0\_DAT0 and DAC0\_DAT1. The use of buffer register is more

meaningful when hardware trigger is used. When buffer is disabled in DAC0\_C1, only DAC0\_DAT0 is used.



Figure 7-24: DAC0\_DAT0L Register

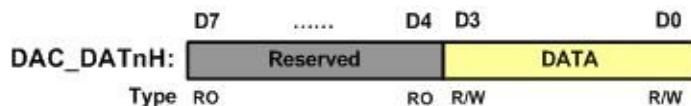


Figure 7-25: DAC0\_DAT0H Register

## Generating a staircase Ramp

In order to generate a staircase ramp, you can run Program 7-4 on the KL25Z microcontroller. To see the result waveform, connect the output (pin PTE30) to an oscilloscope. Table 7-7 shows pin designation for DAC. Figure 7-26 shows the output for Program 7-4.

### Program 7-4: Generating Saw Tooth Wave

```
/* p7_4.c: Use DAC to generate sawtooth waveform

 * The DAC is initialized with no buffer and use software trigger,
 * so every write to the DAC data registers will change the analog output.
 * The loop count i is incremented by 0x0010 every loop. The 12-bit DAC
 * has the range of 0-0xFFFF. Divide 0x1000 by 0x0010 yields 0x0100 or 256.
 * The sawtooth has 256 steps and each step takes 1 ms. The period of the
 * waveform is 256 ms and the frequency is about 3.9 Hz.
 */

#include "MKL25Z4.h"

void DAC0_init(void);
void delayMs(int n);

int main (void) {
    int i;
    DAC0_init();      /* Configure DAC0 */
    while (1) {
        for (i = 0; i < 0x1000; i += 0x0010) {
            /* write value of i to DAC0 */
        }
    }
}
```

```

    DAC0->DAT[0].DATL = i & 0xff;           /* write low byte */
    DAC0->DAT[0].DATH = (i >> 8) & 0x0f; /* write high byte */
    delayMs(1);                      /* delay 1ms */
}

}

}

void DAC0_init(void) {
    SIM->SCGC6 |= 0x80000000; /* clock to DAC module */
    DAC0->C1 = 0;             /* disable the use of buffer */
    DAC0->C0 = 0x80 | 0x20;   /* enable DAC and use software trigger */
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
}

```

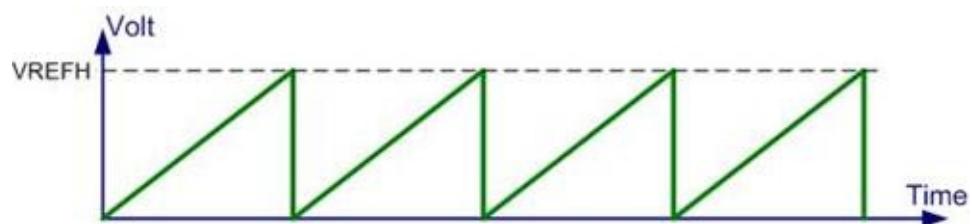


Figure 7-26: Saw Tooth WaveForm

## ***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 positive integer numbers representing the voltage magnitude for the Sine of the angle. This method ensures that only positive integer numbers are output to the DAC by the processor. Table 7-12 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) $(1.5V \times \sin \Theta) + 1.5V$	Values Sent to DAC (decimal) (Voltage Mag. ÷ 0.000806 V)
0	0.000	1.500	1862
30	0.500	2.250	2793
60	0.866	2.799	3474
90	1.000	3.000	3724
120	0.866	2.799	3474
150	0.500	2.250	2793
180	0.000	1.500	1862
210	-0.500	0.750	931
240	-0.866	0.201	249
270	-1.000	0.000	0
300	-0.866	0.201	249
330	-0.500	0.750	931
360	0.000	1.500	1862

Table 7-12: Angle vs. Voltage Magnitude for Sine Wave

To generate Table 7-12, we assumed the full-scale voltage of 3.3V for the DAC output. Full-scale output of the DAC is achieved when all the data input bits of the DAC are high. We will generate a sine wave with amplitude of 3.0V. Since DAC only accepts positive integers, the values of sine wave shall be  $1.5V \pm 1.5V$ . Therefore, to achieve the output amplitude of 3.0V, we use the following equation:

$$V_{OUT} = 1.5V + (1.5V \times \sin \Theta)$$

The DAC is 12 bit and the VREFH is 3.3V, so the step size if  $3.3V / 4096 = 0.000806V$ . To find the values sent to the DAC for various angles, we simply divide the  $V_{OUT}$  voltage by 0.000806V. To further clarify this, look at Example 7-4.

#### Example 7-4

Verify the values of Table 7-12 for the following angles: (a) 30 (b) 60.

**Solution:**

$$(a) V_{OUT} = 1.5 \text{ V} + (1.5 \text{ V} \times \sin \Theta) = 1.5 \text{ V} + 1.5 \text{ V} \times \sin 30 = 1.5 \text{ V} + 1.5 \text{ V} \times 0.5 = 2.25 \text{ V}$$

DAC input values =  $2.25 \text{ V} \div 0.000806 \text{ V} = 2793$  (decimal)

$$(b) V_{OUT} = 1.5 \text{ V} + (1.5 \text{ V} \times \sin \Theta) = 1.5 \text{ V} + 1.5 \text{ V} \times \sin 60 = 1.5 \text{ V} + 1.5 \text{ V} \times 0.866 = 2.799 \text{ V}$$

DAC input values =  $2.799 \text{ V} \div 0.000806 \text{ V} = 3474$  (decimal)

The following program sends the values of Table 7-12 to the DAC. See Figure 7-26.

#### Program 7-5: Generating Sine Wave

```
/*
 * p7_5.c: Use DAC to generate sine wave with look-up table
 */

/* This program uses a pre-calculated lookup table to generate a
 * sine wave output through DAC.
 */

#include "MKL25Z4.h"

void DAC0_init(void);
void delayMs(int n);

int main (void) {
    int i;
    const static int sineWave[] =
    {1862, 2793, 3474, 3724, 3474, 2793,
     1862, 931, 249, 0, 249, 931};

    DAC0_init(); /* Configure DAC0 */
    while (1) {
        for (i = 0; i < 12; i++) {
            /* write value to DAC0 */
            DAC0->DAT[0].DATL = sineWave[i] & 0xff; /* write low byte */
            DAC0->DAT[0].DATH = (sineWave[i] >> 8) & 0x0f; /* write high byte */
            delayMs(1); /* delay 1ms */
        }
    }
}
```

```

    }

}

}

void DAC0_init(void) {
    SIM->SCGC6 |= 0x80000000;          /* clock to DAC module */
    DAC0->C1 = 0;                    /* disable the use of buffer */
    DAC0->C0 = 0x80 | 0x20;          /* enable DAC and use software trigger */
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++)
}

```

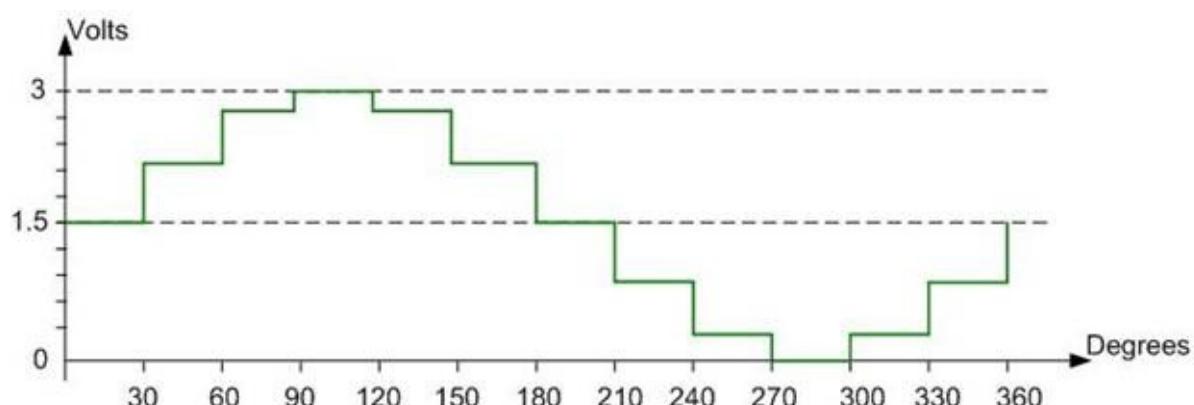


Figure 7-27: Angle vs. Voltage Magnitude for Sine Wave

Program 7-6 uses the C math library functions to generate sine wave lookup table.

#### Program 7-6: Generating Sine Wave Using Math Functions

```

/* p7_6.c: Use DAC to generate sine wave
 *
 * The program calculates the lookup table to generate

```

```

 * sine wave.

*/
#include "MKL25Z4.h"
#include <math.h>

void DAC0_init(void);

#define WAVEFORM_LENGTH 256
int sinewave[WAVEFORM_LENGTH];

int main(void) {
    void delayMs(int n);
    int i;
    float fRadians;
    const float M_PI = 3.1415926535897;

    /* construct data table for a sine wave */
    fRadians = ((2 * M_PI) / WAVEFORM_LENGTH);
    for (i = 0; i < WAVEFORM_LENGTH; i++) {
        sinewave[i] = 2047 * (sinf(fRadians * i) + 1);
    }
    DAC0_init(); /* Configure DAC0 */

    while (1) {
        for (i = 0; i < WAVEFORM_LENGTH; i++) {
            /* write value to DAC0 */
            DAC0->DAT[0].DATL = sinewave[i] & 0xff; /* write low byte */
            DAC0->DAT[0].DATH = (sinewave[i] >> 8) & 0x0f; /* write high byte */
            delayMs(1); /* delay 1ms */
        }
    }
}

void DAC0_init(void) {
    SIM->SCGC6 |= 0x80000000; /* clock to DAC module */
    DAC0->C1 = 0; /* disable the use of buffer */
    DAC0->C0 = 0x80 | 0x20; /* enable DAC and use software trigger */
}

```

```
/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
```

## 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 DAC808 is in \_\_\_\_\_ (current, voltage).
4. The on-chip DAC for KL25Z is \_\_\_\_\_ bit.

## 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. 16
2. 0xFFFF
3. Step size is  $3.3\text{V} / 4096 = 0.8057 \text{ mV}$  and  $1.9\text{V} / 0.809\text{mV} = 2,358$  in decimal or 0x936.
4. ADC0\_Rn
5. 8-,10-,12-,and 16-bit.

### Section 7.3

1. True
2. 10, Celsius
3. 10, Fahrenheit
4. 00111110 (binary) = 62 è Temperature =  $62 \times 330 / 4096 = 5$

### Section 7.4

1. Digital, analog
2. 8
3. current
4. 12-bit





## **Chapter 8: SPI Protocol and Devices**

The SPI (serial peripheral interface) is a bus interface incorporated in many devices such as ADC 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 Freescale ARM KL25Z SPI registers. In Section 8.3 we show 7-LED driver 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.

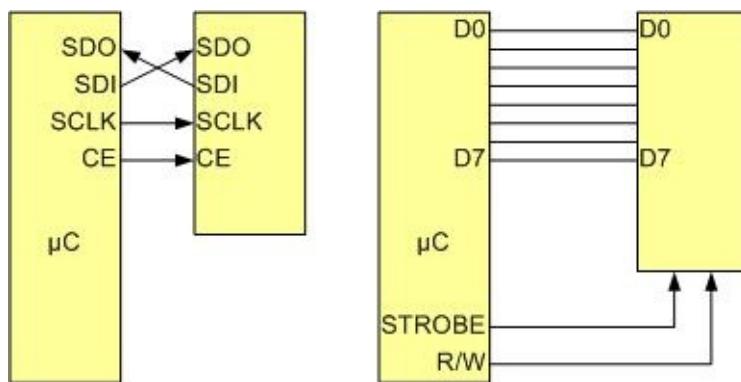
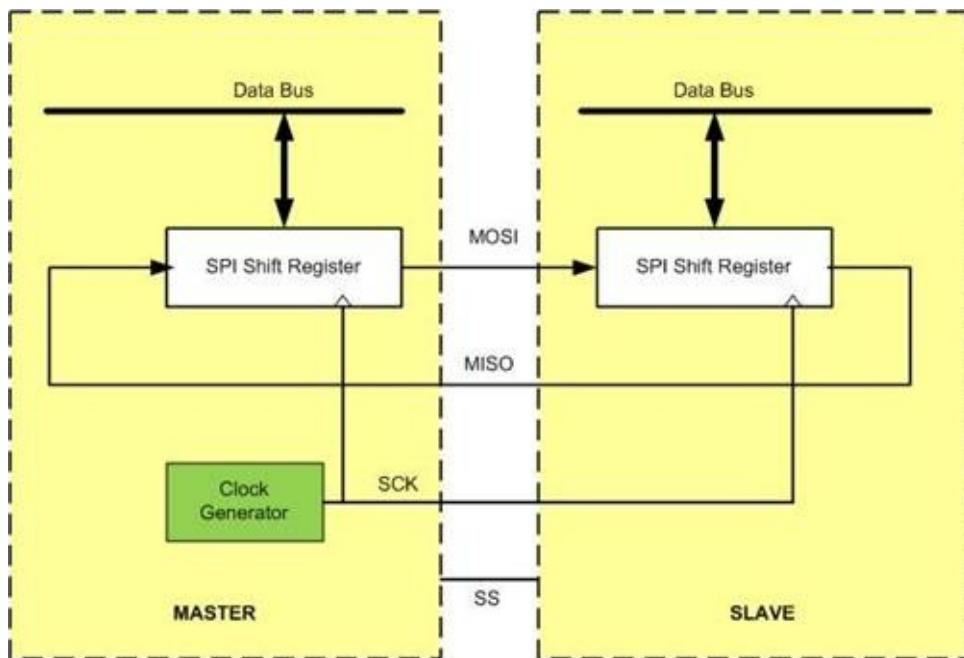


Figure 8-1: SPI Bus vs. Traditional Parallel Bus Connection to Microcontroller

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

In SPI communication, both master and slave use the same clock. The master must choose a clock rate that can be handled by the slave. If the master is driving the clock faster than the slave can handle, the transmission will fail. 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. CPOL determines the idle state of the clock. When CPOL= 0 the idle value of the clock is zero while at CPOL=1 the idle value of the clock is one. CPHA determines when to sample the data. CPHA=0 means data should be sampled on the leading (first) clock edge, while CPHA=1 means data should be sampled 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.

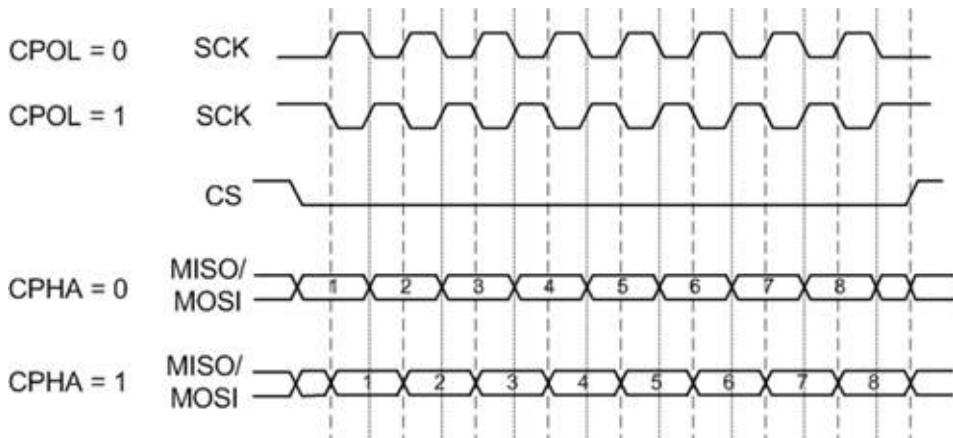


Figure 8-3: SPI Clock Polarity and phase

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

1. True or false. SPI is an Asynchronous protocol.
2. True or false. In the SPI protocol, the clock is always generated by the master device.



## Section 8.2: SPI programming in Freescale ARM KL25Z

The Freescale ARM KL25Z chip comes with two on-chip SPI modules. The SPI modules are located at the following base addresses:

SSI Module	Base Address
SPI0	0x4007 6000
SPI1	0x4007 7000

Table 8-2: SPI Module Base Address

### SPI Register addresses

The following table shows some of the registers in SPI modules with their addresses.

Absolute Address	Register
4007 6000	SPI control register 1 (SPI0_C1)
4007 6001	SPI control register 2 (SPI0_C2)
4007 6002	SPI baud rate register (SPI0_BR)
4007 6003	SPI status register (SPI0_S)
4007 6005	SPI data register (SPI0_D)
4007 7000	SPI control register 1 (SPI1_C1)
4007 7001	SPI control register 2 (SPI1_C2)
4007 7002	SPI baud rate register (SPI1_BR)
4007 7003	SPI status register (SPI1_S)
4007 7005	SPI data register (SPI1_D)

Table 8-3: Some of the KL25Z SPI Registers

### Enabling Clock to SPI

To enable and use any of the peripheral modules in the Freescale KL25Z chip, we must enable the clock to it. We use SIM\_SCGC4 register to enable the clock to SPI modules. Writing a one to D22 or D23, enables the corresponding SPI module. Notice, in addition to providing the Clock to SPI module we must also enable the SPI module. We enable SPI using the SPIx\_C1 register, as we will see next.

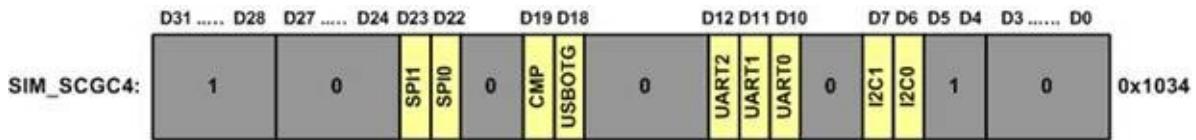


Figure 8-4: SIM\_SCGC4 register for enabling Clock to SPI

## SPI Control 1 register

The SPIx\_C1 (SPI Control 1) register sets SPI configuration. Figure 8-5 shows the bits of SPIx\_C1. We must use SPI\_C1 register to select the SPI mode operation of the KL25Z. Notice that the SPE bit in the SPIx\_C1 register must be set to HIGH to allow the use of the KL25Z pins for SPI data bus protocol. We choose the SPI Master mode by using the MSTR bit of SPIx\_C1 register. The CPOL bit is used for selecting an inverted or non-inverted SPI clock. In the active-HIGH (non-inverted) SCK, it is low in the idle state. We must make sure that the SPI slave device has the same SCK polarity as the KL25Z master. We use the CPHA bit in the SPIx\_C1 register to select the rising or falling edge of the SCK for sampling of data. We also have the option of sending out the LSB or the MSB first.

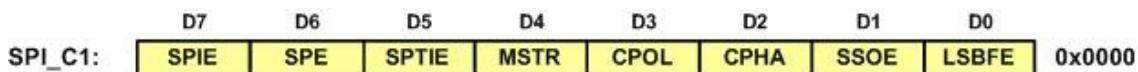


Figure 8-5: SPIx\_C1 Control 1 Register

Field	Bit	Descriptions
SPIE	D7	SPI Interrupt Enable. This bit enables SPI interrupt request for SPRF and MODF. 1 = SPRF and MODF interrupts enabled 0 = SPRF and MODF interrupts disabled
SPE	D6	SPI System Enable bit 1 = Enables SPI port and configures pins as serial port pins 0 = Disables SPI port and configures these pins as I/O ports
SPTIE	D5	SPI Transmit Interrupt Enable. This bit enables the SPI interrupt request if SPTEF = 1. 1 = SPTEF interrupt enabled 0 = SPTEF interrupt disabled
MSTR	D4	SPI Master/Slave mode Select bit. This bit selects master or slave mode. 1 = SPI in master mode 0 = SPI in slave mode
CPOL	D3	SPI Clock Polarity bit 1 = Active-LOW clocks selected. In idle state SCK is high.

		0 = Active-HIGH clocks selected. In idle state SCK is low.
		SPI Clock Phase bit
CPHA	D2	1 = Sampling of data occurs at even edges of the SCK clock. 0 = Sampling of data occurs at odd edges of the SCK clock.
SSOE	D1	Slave Select Output Enable. See the KL25Z manual.
		LSB First Enable
LSBFE	D0	1 = Data is transferred least significant bit first. 0 = Data is transferred most significant bit first.

Table 8-4: SPIx\_C1 Control 1 Register

## SPI Control 2 register

With the options available in SPIx\_C2 (SPI Control Register 2), many other features of the SPI module, such as bidirectional data transfer, can be used. See the KL25Z manual.

SPIx_C2:	D7	D6	D5	D4	D3	D2	D1	D0	0x0001
	SPMIE	0	TXDMAE	MODFEN	BIDIROE	RXDMAE	SPISWAI	SPC0	

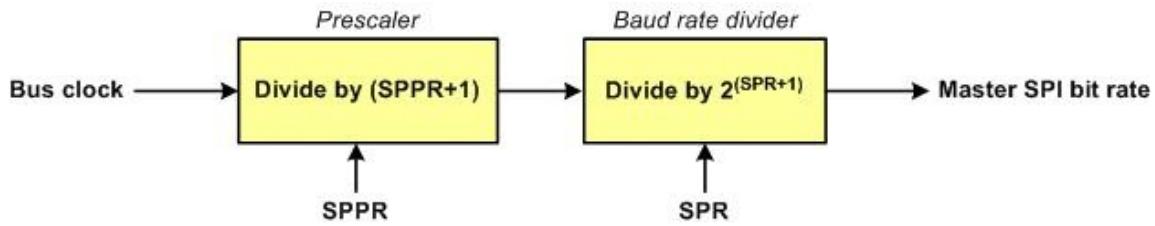
Figure 8-6: SPIxC2 (SPI Control 2) Register

Field	Bit	Descriptions
<b>MODFEN</b>	D4	Mode Fault Enable Bit. See the KL25Z reference manual. 1 = SS port pin with MODF feature 0 = SS port pin is not used by the SPI (default)
<b>BIDIROE</b>	D3	Bidirectional Output Enable. Used in bidirectional mode. 1 = Output Buffer enabled. See the HCS12 manual. 0 = Output Buffer enabled. (default)
<b>SPC0</b>	D0	Serial Pin Control bit 0. Used in bidirectional mode. 1 = Bidirectional. See the KL25Z reference manual. 0 = Normal (default)

Table 8-5: SPIxC2 (SPI Control 2) Register

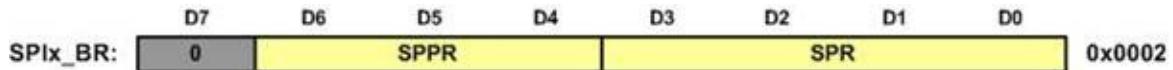
## Setting Bit Rate

We use the SPIx\_BR (SPI Baud Rate) register to set the SCLK rate for data transfer and receive. See Figure 8-7.



**Figure 8-7: SPI Baud Rate Generation**

Module clock source is generally Bus Clock. The selected frequency is fed to prescaler before it is used by the Bit Rate circuitry. The prescaler and divisor for SPIx\_BR register are shown in Figures 8-7 and 8-8 and Tables 8-6 and 8-7.



**Figure 8-8: SPIx\_BR Register**

Field	Bit	Description
SPPR2–SPPR0	D6–D4	SPI Baud Rate Prescaler Divisor bits
SPR3–SPR0	D3–D0	<p>SPI Baud Rate Divisor bits</p> <p>These bits specify the SPI baud rates as shown in the following two equations:</p> $\text{BaudRateDivisor} = (\text{SPPR} + 1) \times 2^{(\text{SPR} + 1)}$ $\text{Baud Rate} = \text{BusClock} / \text{BaudRateDivisor}$

**Table 8-6: SPIx\_BR Register**

SPPR2 SPPR1 SPPR0	SPR3 - SPR0	SPIx_BR (Hex)	BaudRateDivisor
0 0 0	0 0 0 0	00	2
0 0 0	0 0 0 1	01	4
0 0 0	0 0 1 0	02	8
0 0 0	0 0 1 1	03	16
0 0 0	0 1 0 0	04	32
0 0 0	0 1 0 1	05	64
....	....	....	....
1 1 0	0 1 1 1	67	1792
1 1 1	0 0 0 0	70	16

1 1 1	0 0 0 1	71	32
1 1 1	0 0 1 0	72	64
1 1 1	0 0 1 1	73	128
1 1 1	0 1 0 0	74	256
1 1 1	0 1 0 1	75	512
1 1 1	0 1 1 0	76	1024
1 1 1	0 1 1 1	77	2048

*Note:* The highest Baud Rate = BusFreq/2 and the lowest Baud Rate is BusFreq/2048

**Table 8-7: Some possible Values for SPIx\_BR Register**

See Examples 8-1 and 8-2.

### Example 8-1

Using BusFreq = 2 MHz, find the value for the SPI Baud Rate (SPIx\_BR) register for the following bit rates: a) 1 MHz, b) 500 kHz c) 7,812.5Hz

#### Solution:

- (a) For 2 MHz/1 MHz = 2, we have SPIx\_BR = 0000 0000 = 00 hex.
- (b) For 2 MHz/500 kHz = 4, we have SPIx\_BR = 0000 0001= 01 hex.
- (c) For 2 MHz/7,812.5 Hz = 256, we have SPIx\_BR = 0111 0100 = 74 hex.

### Example 8-2

In a given program, we have SPIx\_BR=0x72. Find baud rate if Bus Clock frequency is 13.98MHz.

#### Solution:

From Table 8-7, we have 0x72 for baud rate divisor of 64. Now, the BR=13.98MHz/64=218,437 Hz.

### Example 8-3

Find the value for the control register SPIx\_C1 register. Assume no interrupt, SPI enabled, SPI master, active-HIGH clock, sampling data on rising edge, and MSB first.

### Solution:

SPIx\_C1 = 0101 0100 or 0x54.

---

## Data Register

The data is placed in SPIx\_D (SPIx Data) register for transmission. The SPIx\_D register is also used for the received data buffer.

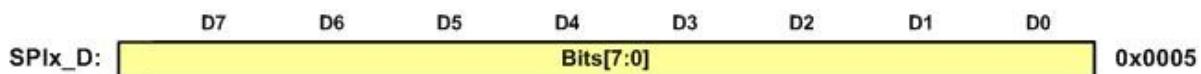


Figure 8-9: SPIx\_D Register

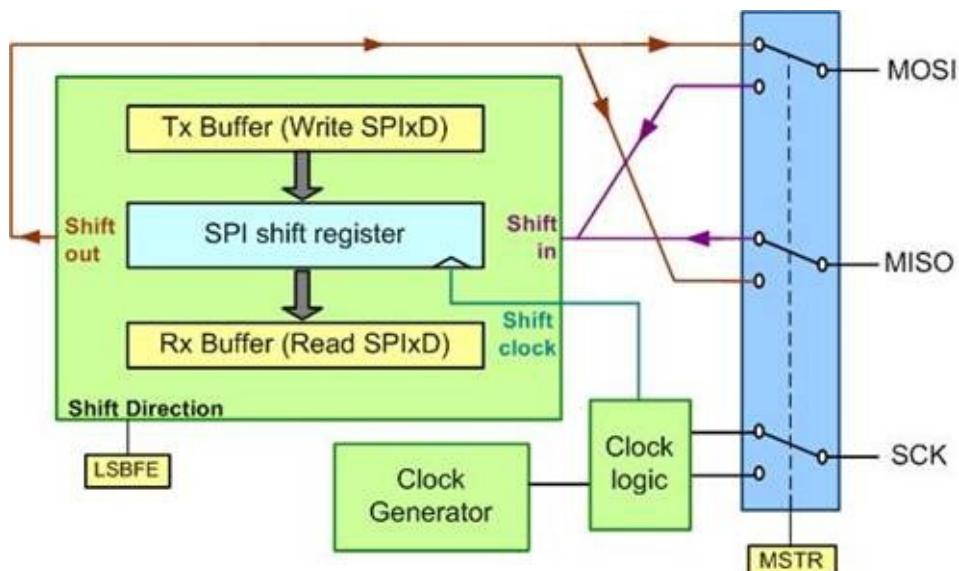


Figure 8-10: SPI Data (SPIx\_D) Register

## Status Flag Register

We use the SPI Status register (SPIx\_S) 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 8-8.

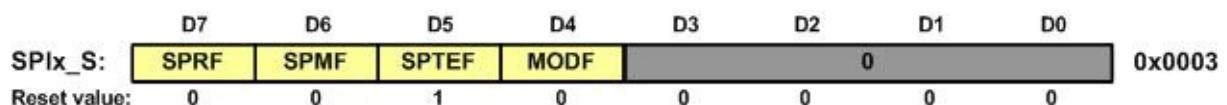


Figure 8-11: SPI Status (SPIx\_S) Register

Field	Bit	Descriptions
SPI read buffer full Flag. This bit is set after a received byte of		

SPRF	D7	<p>data has been placed into the SPI Data Register. This bit is cleared by reading the SPI Status Register (SPIx_S) followed by a read from the SPI Data Register.</p> <p>1 = New data has been received and placed in SPIx_D. 0 = Transfer not yet complete</p>
SPTEF	D5	<p>SPI Transmit Buffer Empty Flag. If set, this bit indicates that the transmit data register is empty and ready for a new byte of data.</p> <p>1 = SPI Data Register empty 0 = SPI Data Register not empty</p>
MODF	D4	<p>Mode Fault flag is used for mode selection error. See the KL25Z manual.</p> <p>1 = Mode fault has occurred. 0 = Mode fault has not occurred.</p>

Table 8-8: SPI Status (SPIx\_S) Register

## Configuring GPIO for SPI

In using SPI, we must also configure the GPIO pins to allow the connection of the CPU pins to SPI device pins. See Table 8-9. In this regard, it is the same as all other peripherals. The steps are as follow:

1. Enable the clock to GPIO pin using SIM\_SCGC5.
2. Assign the SPI signals to specific pins using ALT option in PORTx\_PCR register.

SPI Module Pin	I/O Pin	Other Alternate pins
SPI0_PCS0	PTC4 (ALT2)	PTD0(ALT2)
SPI0_SCK	PTC5 (ALT2)	PTD1(ALT2)
SPI0_MOSI	PTC6 (ALT2)	PTD2(ALT2)
SPI0_MISO	PTC5 (ALT2)	PTD3(ALT2)
SPI1_PCS0	PTB10 (ALT2)	PTD4(ALT2)
SPI1_SCK	PTB11 (ALT2)	PTD5(ALT2)
SPI1_MOSI	PTB16 (ALT2)	PTD6(ALT2)

SPI1\_MISO

PTB17 (ALT2)

PTD7(ALT2)

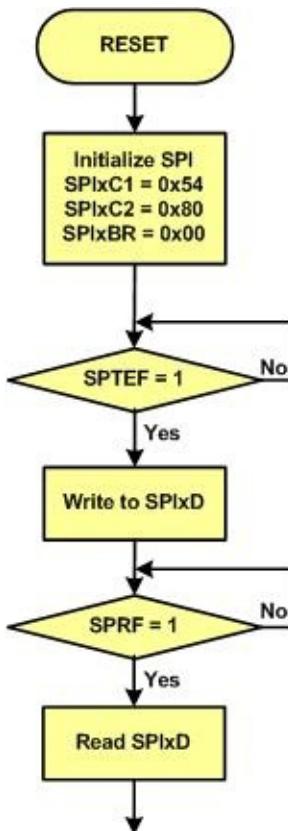
**Note:** Some of the SPI signals use ALT5 option in PORTx\_PCR register. See KL25Z Ref. Manual.

**Table 8-9: I/O Pin Assignment for both SPI0 and SPI1 Modules**

## Configuring SPI for data transmission

After the I/O configuration, we need to take the following steps to transmit a byte of data using the SPI protocol:

1. Enable the clock to SPI module using SIM\_SCGC4 register.
2. Disable the SPI via SPIx\_C1 register before initialization.
3. Set the Bit Rate with the SPIx\_BR registers.
4. Also select the SPI mode, phase, master, and polarity in SPIx\_C1 control register. Make sure the master mod in SPIx\_C1 register.
5. Enable SPI using SPIx\_C1 register.
6. Assert slave select signal.
7. Monitor the SPTEF flag (SPI Transmit buffer empty flag) in SPIx\_S (SPIx status) register until it goes high. Then, load a byte of data into SPIx\_D (SPI data) register to be transmitted.
8. Wait until SPRF (SPI read buffer full flag) bit in SPIx\_S register is set signaling that the transmission is complete. Read the SPIx\_D register to clear the SPRF.
9. Repeat step 6-7 above until all the data are transferred.
10. Deassert slave select signal.



**Figure 8-12: Initialization Flowchart for SPI Master Device**

### Program 8-1: sending 'A' to 'Z' characters via SPI0

```

/*P8_1.c: Send 'A' to 'Z' via SPI0

 * PTD1 pin as SPI SCK
 * PTD2 pin as SPI MOSI
 * PTDO pin as SPI SS
 */

#include "MKL25Z4.h"

void SPI0_init(void);
void SPI0_write(unsigned char data);

int main(void) {
    unsigned char c;
    SPI0_init(); /* enable SPI0 */
    while(1) {
        for(c = 'A'; c <= 'Z'; c++) {
            SPI0_write(c);
        }
    }
}

```

```

    }

}

}

void SPI0_init(void) {
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[1] = 0x200;        /* make PTD1 pin as SPI SCK */
    PORTD->PCR[2] = 0x200;        /* make PTD2 pin as SPI MOSI */
    PORTD->PCR[0] = 0x100;        /* make PTD0 pin as GPIO */
    PTD->PDDR |= 0x01;           /* make PTD0 as output pin for /SS */
    PTD->PSOR = 0x01;            /* make PTD0 idle high */
    SIM->SCGC4 |= 0x400000;       /* enable clock to SPI0 */
    SPI0->C1 = 0x10;             /* disable SPI and make SPI0 master */
    SPI0->BR = 0x60;              /* set Baud rate to 1 MHz */
    SPI0->C1 |= 0x40;             /* Enable SPI module */
}

void SPI0_write(unsigned char data) {
    volatile char dummy;
    PTD->PCOR = 1;                /* assert /SS */
    while(!(SPI0->S & 0x20)) { }   /* wait until tx ready */
    SPI0->D = data;               /* send data byte */
    while(!(SPI0->S & 0x80)) { }   /* wait until tx complete */
    dummy = SPI0->D;              /* clear SPRF */
    PTD->PSOR = 1;                /* deassert /SS */
}

```

## Review Questions

- True or false. The Freescale ARM KL25Z does not support SPI protocol.
- True or false. The Prescaler register of SPI\_BR can have any odd or even number between 1 and 255.
- In Freescale ARM KL25Z, which register is used to enable the clock to SPI module?
- In Freescale ARM KL25Z, which register is used to set the SPI baud rate?



## Section 8.3: MAX7221 SPI 7-Segment Driver

Chapter 2 examined 7-seg concepts. To save pins we can use MAX7219/21 chip. In this section we show an SPI-based 7-seg driver and its interfacing to ARM KL25Z. MAX7221 is an SPI serial 7-segment driver from Maxim Corporation. It can support up to 8-digit seven-segment display.

There are two types of 7-segments, common anode and common cathode. The MAX7221 supports common cathode only. See Figure 8-13.

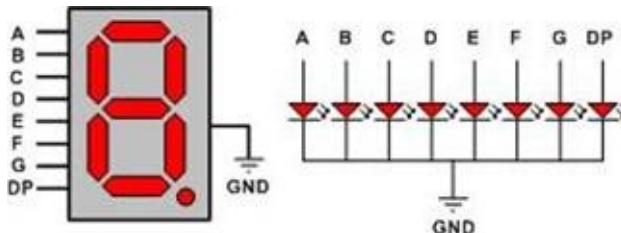


Figure 8-13: Common Cathode Connections in a 7-Segment Display

In many applications you need to connect two or more 7-segment LEDs to a microcontroller. For example, if you want to connect four 7-segment LEDs directly to a microcontroller you need  $4 \times 8 = 32$  pins. This is costly. The MAX7221 IC is an ideal chip for such applications since it supports up to eight 7-segment LEDs. We can connect the MAX7221 to the microcontroller using SPI protocol and control up to eight 7-segment LEDs. The MAX7221 contains an internal decoder that can be used to convert binary numbers to 7-segment codes. It activates the digits one at a time. That means the CPU does not need to refresh the 7-segment LEDs. All you need to do is to send a binary number to the MAX7221, and the chip decodes the binary data and displays the number. The device includes analog and digital brightness control, an  $8 \times 8$  static RAM that stores each digit, and a test mode that forces all LEDs on. Next, we will show how to interface an MAX7221 to the KL25Z and program it using SPI protocol.

### MAX7221 pins and connections

The MAX7221 is a 24-pin DIP chip. It can be directly connected to the microcontroller and control up to eight 7-segment LEDs. A resistor or a potentiometer is the only external component that you need. Next, we will discuss the pins of the MAX7221. See Figure 8-14.

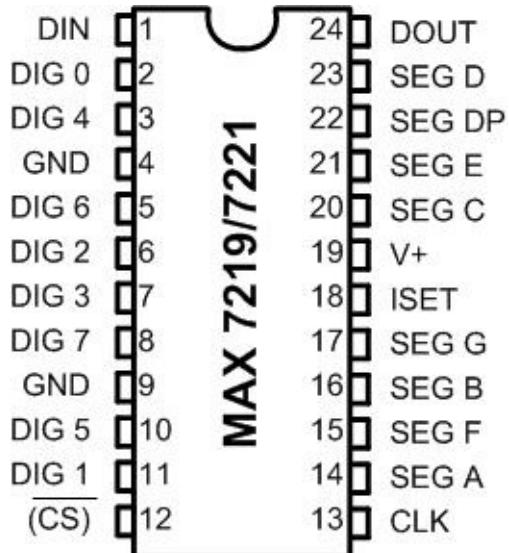


Figure 8-14: MAX7219 and MAX7221

## GND

Pin 4 and pin 9 are the ground. Notice that both of the ground pins should be connected to system ground and you cannot leave any of them unconnected.

## VCC

Pin 19 is the VCC and should be connected to the +5 V power supply. Notice that this pin also supplies the power to drive the 7-segments and the connecting wire to this pin should be able to handle 100–300 mA.

## ISET

Pin 18 is ISET and sets the maximum segment current. This pin should be connected to VCC through a resistor. A 10 kΩ resistor can be connected to this pin. If you want to manually control the segments' light intensity, you can replace the resistor with a 50K potentiometer. For more details about how to calculate the value of the resistor you can look at the datasheet of the chip.

## CS

Pin 12 is the chip select pin and should be connected to the SS (slave select) pin of the microcontroller. Serial data is loaded into the chip while CS is low, and the last 16 bits of the serial data are latched on the rising edge of CS.

## DIN

Pin 1 is the serial data input and should be connected to the MOSI pin of the microcontroller. On CLK's rising edge, data on this pin is loaded into the internal shift register. Notice that the MAX7221 reads the bit on rising edge.

## CLK

Pin 13 is the serial clock input and should be connected to the SCK pin of the microcontroller. On MAX7221 the clock input is inactive when CS is high.

## DOUT

Pin 24 is the serial data output and is used to connect more than one MAX7221 to a single SPI bus.

## DIG0-DIG7

The DIG pins are the 7-segment selector pins and should be connected to the 7-segments' common cathode pin. The MAX7221 chip can control up to eight 7-segment LEDs. These eight 7-segment displays are designated as DIG0 to DIG7.

## SEG<sub>A</sub>-SEG<sub>G</sub> and DP

These pins select each segment and should be connected to segments of each 7-segment accordingly. Figure 8-15 shows the connection for two 7-segments. You can connect up to eight 7-segments to MAX7221.

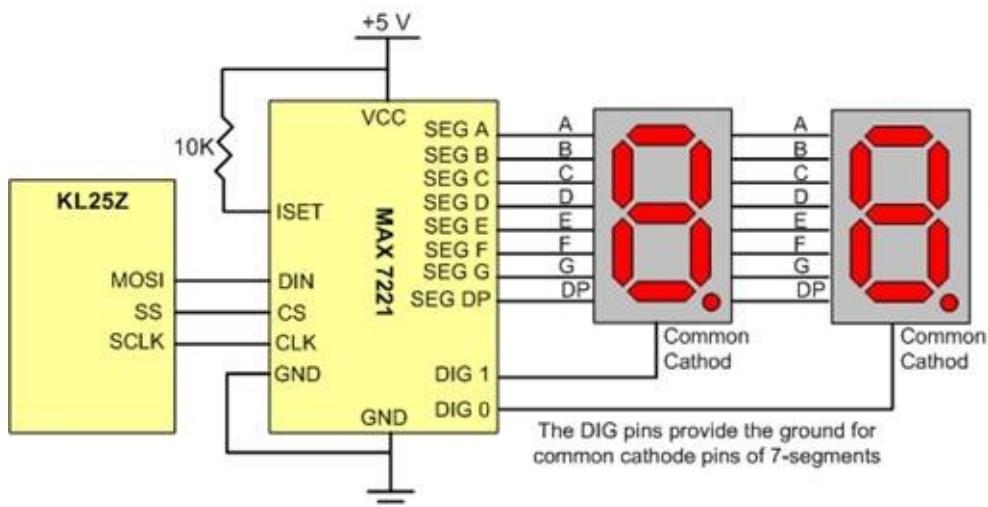


Figure 8-15: MAX7221 Connections to the Microcontroller

## MAX7221 data packet format

In MAX7221, data packets are 16 bits long (two bytes). You should first make CS low before transmitting; then you transmit two bytes of data and terminate the transmission by making CS high.

The first byte (MSBs) of each packet contains the command control bits, and the second byte is the data to be displayed. See Figure 8-16.

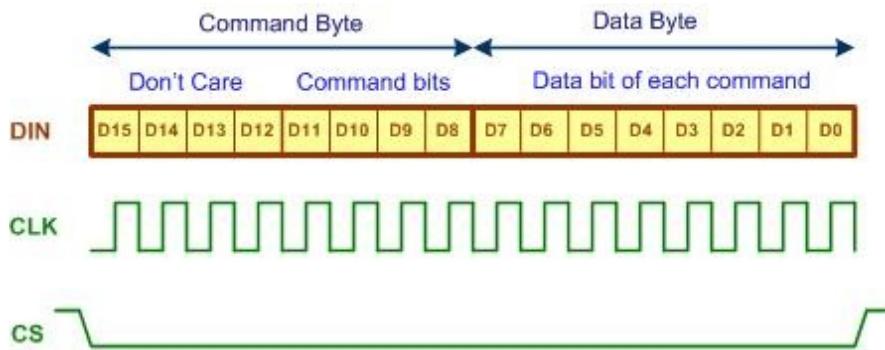


Figure 8-16: MAX7221 Packet Format

The upper four bits (D15–D12) of the command byte are “don’t cares” and the lower four bits (D11–D8) are used to identify the meaning of the data byte that

follows. The second byte (D7–D0) of the two-byte packet is called the data byte and is the actual data to be displayed or control the 7-segment driver. Table 8-9 shows the binary and hex values of each command. Next, we will discuss the commands in more detail.

Command	D15-12	D11	D10	D9	D8	Hex Code
No operation	X	0	0	0	0	X0
Set value of digit 0	X	0	0	0	1	X1
Set value of digit 1	X	0	0	1	0	X2
Set value of digit 2	X	0	0	1	1	X3
Set value of digit 3	X	0	1	0	0	X4
Set value of digit 4	X	0	1	0	1	X5
Set value of digit 5	X	0	1	1	0	X6
Set value of digit 6	X	0	1	1	1	X7
Set value of digit 7	X	1	0	0	0	X8
Set decoding mode	X	1	0	0	1	X9
Set intensity of light	X	1	0	1	0	XA
Set scan limit	X	1	0	1	1	XB
Turn on/off	X	1	1	0	0	XC
Display test	X	1	1	1	1	XF

**Notes:**

- 1) X means don't care.
- 2) Digits are designated as 0-7 to drive total of eight 7-segment LEDs.

Table 8-10: List of Commands in MAX7221/MAX7219

### Set value of digit 0-digit 7 (commands X1-X8)

These commands set what is to be displayed on each 7-segment. You can either send a binary number to the chip decoder and let it turn on/off the segments accordingly, or you may decide to turn on/off each segment of the 7-segment by yourself. The first way is useful when you do not want to deal with converting a binary number to 7-segment codes. The second way is useful when you want to show a character or any other thing that is not predefined. For example, if you want to show letter 'U', you should use the second way and turn on/off segments yourself. Next, you will see how to enable or bypass the decoder for each 7-

segment.

### **Set decoding mode (command X9)**

This command lets you enable or bypass the binary to 7-segment decoding function for each 7-segment digit. Each bit in the data byte (second byte) is assigned to one digit of 7-segment. D0 is assigned to Digit 0, D1 is assigned to Digit 1, and so on. If you want to enable the decoding function for a digit you should set to one the bit assigned to that digit, and if you want to disable the decoding function you should clear the bit for that digit. Figure 8-17 shows the structure of the set decoding mode command. See Examples 8-4 and 8-5.



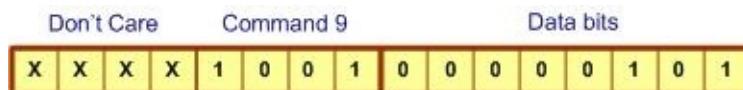
Figure 8-17: Set Decoding Mode Command Format

### **Example 8-4**

What sequence of bytes should be sent to the MAX7221 in order to enable the decoding function for digit 0 and digit 2, and disable the decoding function for other digits?

#### **Solution:**

The first byte should be xxxx 1001 (X9 hex) to execute the “Set decoding mode” command, and the second byte (argument of the command) should be 0000 0101 to enable the decoding function for digit 0 and digit 2.



### **Example 8-5**

After running Example 8-4, what sequence of numbers should be sent to the MAX7221 in order to write 5 on digit 2?

#### **Solution:**

The first byte should be xxxx 0011 (X3 hex) to execute the “Set value of digit 2” command, and the second byte (argument of the command) should be 0000 0101 (05 hex) to write 5 on digit 2. Notice that the decoding function for digit 2 has been enabled before.

Don't Care	Command 3	Data bits
x x x x	0 0 1 1	0 0 0 0 0 0 1 0 1

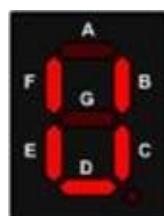
If you want to turn on/off each segment by yourself to display a specific letter on a 7-segment, you should bypass the decoding function and then use the “Set value of digit x” command to turn on/off each bit of a segment. As you see in Figure 8-18, each bit of the data bits is assigned to a segment of the 7-segment. For example, D0 is assigned to the G segment, D1 is assigned to the F segment, and so on. If you want to turn on a segment, you should write one to the corresponding bit, and if you want to turn off a segment, you should write zero to its bit. Figure 8-18 shows the bit assigned to each segment. See Example 8-6.

Bit:	D7	D6	D5	D4	D3	D2	D1	D0
Segment:	DP	A	B	C	D	E	F	G

Figure 8-18: Bits Assigned to Segments

### Example 8-6

After running Example 8-4, what sequence of numbers should be sent to the MAX7221 in order to write U on digit 1?



#### Solution:

The decoding function for digit 1 has been disabled before in Example 8-4, and we have to turn on/off each segment manually. As you see in the figure, segments B, C, D, E, and F should be turned on. To turn on these segments of digit 1, we should send the first byte xxxx 0010 (X2 hex) to execute the “Set value of digit 1” command and then we should send 0011 1110 (3E hex) to write U on digit 1. Notice that the decoding function for digit 1 has been enabled before. The figure below shows the bits.

Don't Care	Command 3	Data bits
x x x x	0 0 1 0	0 0 1 1 1 1 1 0

DP A B C D E F G

---

## ***Set Intensity of Light (command XA)***

This command sets the light intensity of the segments. The intensity can be any value between 0 and 16 (0F hex). 0 is the minimum value of intensity, and 16 is the maximum value of intensity. Notice that 0 does not mean off but it is the minimum intensity. As we mentioned before, you can also change the light intensity of segments by changing the resistor that connects the ISET pin to VCC.

## ***Set Scan Limit (command XB)***

This command sets the number of 7-segments that are connected to the chip. This number can vary from 1 to 8.

## ***Turn On/ Off (command XC)***

This command turns the display on or off. 1 (01 hex) turns the display on, while 0 (00 hex) turns off the display. This command is useful when you want to reduce the power consumption of your device.

## ***Display Test (command XF)***

This command is used to test the display. If you send 1 (01 hex) after sending the display test command to the chip, it enters display-test mode and turns on all segments. This lets you check to see if all segments work properly. When you want to return to normal operation mode, you should execute the command but send 0 (00 hex) as data to the chip.

## ***MAX7221 programming in the KL25Z***

To program MAX7221 in the KL25Z you should do the following steps. Notice that step 4 is optional and can be ignored:

1. Initialize the SPI to operate in master mode so that data is stable on rising edge and changes on falling edge.
2. Enable or disable decoding mode by executing command 9 (x9 hex).
3. Set the scan limit.
4. Set the intensity of light (optional).
5. Disable test mode
6. Turn on the display.
7. Set the values of each digit.

See Programs 8-2 and 8-3. Program 8-2 shows how to display 57 on the 7-segment display of Figure 8-15 using the decoding function.

**Program 8-2: Displaying 57 on the 7-segment**

```

/*P8_2.c: Programming MAX7219 via SPI with FRDM-KL25Z

 * MAX7219 is connected to a two-digit seven-segment LED
 * The program displays "57" on the seven-segment LED
 * PTD1 pin as SPI SCK
 * PTD2 pin as SPI MOSI
 * PTDO pin as chip select
 */

#include "MKL25Z4.h"

void SPI0_init(void);
void SPI0_write(unsigned char data);
void max7219_write(unsigned char command, unsigned char data);

#define DECODE 9
#define INTENSITY 10
#define SCANLIMIT 11
#define SHUTDOWN 12
#define TEST 15

int main(void) {
    SPI0_init(); /* enable SPI0 */
    max7219_write(DECODE, 3); /* enable decode for digit 1, 0 */
    max7219_write(SCANLIMIT, 2); /* scan two digits */
    max7219_write(INTENSITY, 4); /* set 1/4 intensity */
    max7219_write(TEST, 0); /* disable test mode */
    max7219_write(SHUTDOWN, 1); /* enable device */
    max7219_write(0x02, 5); /* display 5 */
    max7219_write(0x01, 7); /* display 7 */

    while(1) {
    }

}

void SPI0_init(void) {
    SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
}

```

```

PORTD->PCR[1] = 0x200;           /* make PTD1 pin as SPI SCK */
PORTD->PCR[2] = 0x200;           /* make PTD2 pin as SPI MOSI */
PORTD->PCR[0] = 0x100;           /* make PTDO pin as GPIO */
PTD->PDDR |= 0x01;             /* make PTDO as output pin for CS */
PTD->PSOR = 0x01;              /* make PTDO idle high */
SIM->SCGC4 |= 0x400000;         /* enable clock to SPI0 */
SPI0->C1 = 0x10;                /* disable SPI and make SPI0 master */
SPI0->BR = 0x60;                /* set Baud rate to 1 MHz */
SPI0->C1 |= 0x40;               /* Enable SPI module */
}

void max7219_write(unsigned char command, unsigned char data) {
    volatile char dummy;
    PTD->PCOR = 1;                  /* assert /CS */

    while(!(SPI0->S & 0x20)) { }    /* wait until tx ready */
    SPI0->D = command;              /* send command byte first */
    while(!(SPI0->S & 0x80)) { }    /* wait until tx complete */
    dummy = SPI0->D;                /* clear SPRF */
    while(!(SPI0->S & 0x20)) { }    /* wait until tx ready */
    SPI0->D = data;                /* send data byte */
    while(!(SPI0->S & 0x80)) { }    /* wait until tx complete */
    dummy = SPI0->D;                /* clear SPRF */
    PTD->PSOR = 1;                  /* de-assert /CS */
}

```

Program 8-3 shows how to display “2U” on the 7-segment of Figure 8-15 without using the decoding function for letter ‘U’.

### Program 8-3: Displaying 2U on the 7-segment

```

/*P8_3.c: Programming MAX7219 via SPI with FRDM-KL25Z

* MAX7219 is connected to a two-digit seven-segment LED
* Display "2U" on the seven-segment LED
* PTD1 pin as SPI SCK
* PTD2 pin as SPI MOSI
* PTDO pin as chip select

```

```

/*
#include "MKL25Z4.h"

void SPI0_init(void);
void SPI0_write(unsigned char data);
void max7219_write(unsigned char command, unsigned char data);

#define DECODE 9
#define INTENSITY 10
#define SCANLIMIT 11
#define SHUTDOWN 12
#define TEST 15

int main(void) {
    SPI0_init();
    max7219_write(DECODE, 2); /* enable decode for digit 1 */
    max7219_write(SCANLIMIT, 2); /* scan two digits */
    max7219_write(INTENSITY, 4); /* set 1/4 intensity */
    max7219_write(TEST, 0); /* disable test mode */
    max7219_write(SHUTDOWN, 1); /* enable device */
    max7219_write(0x02, 0x02); /* display 2 */
    max7219_write(0x01, 0x3E); /* display U (see Example 8-6) */

    while(1) {
    }
}

void SPI0_init(void) {
    SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
    PORTD->PCR[1] = 0x200; /* make PTD1 pin as SPI SCK */
    PORTD->PCR[2] = 0x200; /* make PTD2 pin as SPI MOSI */
    PORTD->PCR[0] = 0x100; /* make PTDO pin as GPIO */
    PTD->PDDR |= 0x01; /* make PTDO as output pin for CS */
    PTD->PSOR = 0x01; /* make PTDO idle high */
    SIM->SCGC4 |= 0x400000; /* enable clock to SPI0 */
    SPI0->C1 = 0x10; /* disable SPI and make SPI0 master */
    SPI0->BR = 0x60; /* set Baud rate to 1 MHz */
}

```

```

    SPI0->C1 |= 0x40;           /* Enable SPI module */

}

void max7219_write(unsigned char command, unsigned char data) {
    volatile char dummy;
    PTD->PCOR = 1;             /* assert /CS */

    while(!(SPI0->S & 0x20)) { } /* wait until tx ready */
    SPI0->D = command;         /* send command byte first */
    while(!(SPI0->S & 0x80)) { } /* wait until tx complete */
    dummy = SPI0->D;           /* clear SPRF */
    while(!(SPI0->S & 0x20)) { } /* wait until tx ready */
    SPI0->D = data;            /* send data byte */
    while(!(SPI0->S & 0x80)) { } /* wait until tx complete */
    dummy = SPI0->D;           /* clear SPRF */
    PTD->PSOR = 1;              /* de-assert /CS */
}

```

## Review Questions

1. How many 7-segments can be controlled by MAX7221?
2. What would happen if you do not set the scan limit?
3. True or False. If you want to show P on a 7-segment you can use the decoding function.
4. Which segments should be on to display P on a 7-segment?
5. What is the recommended value of the ISET resistor?

# **Answers to Review Questions**

## **Section 8.1**

1. False
2. True

## **Section 8.2**

1. False
2. False
3. SIM\_SCGC4
4. SPIx\_BR

## **Section 8.3**

1. 8
2. The scan limit would be 0 and nothing would be shown on the 7-segment.
3. False
4. A, B, E, F, G
5. 10 kΩ





# **Chapter 9: I2C Protocol and RTC Interfacing**

This chapter covers I2C bus interfacing and programming. Section 9.1 examines the I2C bus protocol. Section 9.2 shows the inner working of I2C module in Freescale ARM KL25Z devices. The DS1337 RTC and its I2C interfacing and programming are covered in Section 9.3.



## 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 some application notes, I2C is referred to as Two-Wire Serial Interface (TWI).

### I2C line electrical characteristics

I2C devices use only 2 bidirectional open-drain pin for data communication. To implement I2C, 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.

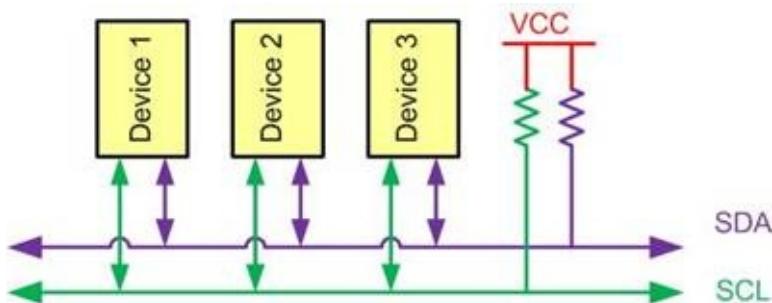


Figure 9-1: I2C Bus Characteristics

### 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 for each node. 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 any given time. See Example 9-1

### Example 9-1

Give an example to show how a device (node) can use 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.

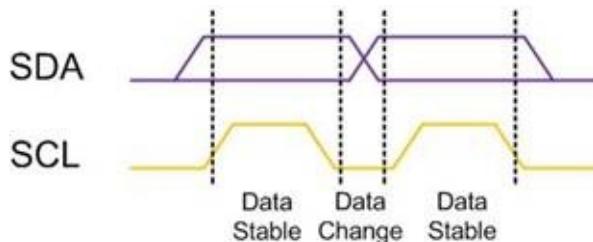


Figure 9-2: I2C Bit Format

#### 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 high. See Figure 9-3.

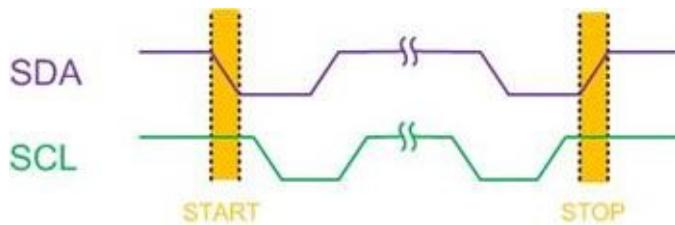


Figure 9-3: START and STOP Conditions

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.

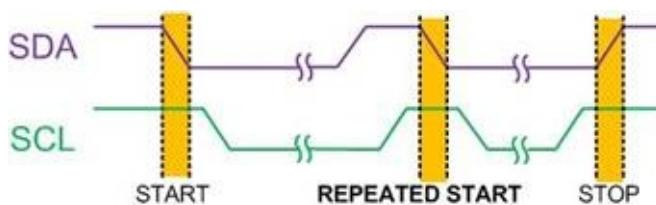


Figure 9-4: REPEATED START Condition

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 two microcontrollers (uA and uB) and an EEPROM with I<sup>2</sup>C, and the uA wants to display the sum of the contents at 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 acknowledgement 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 acknowledgement, the transmitter releases 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.

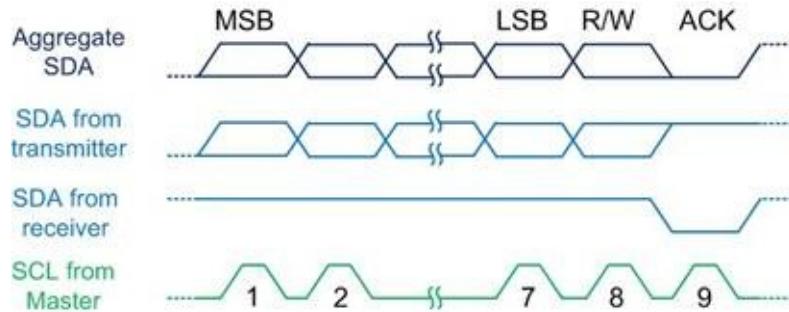


Figure 9-5: Byte Format in I2C

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)

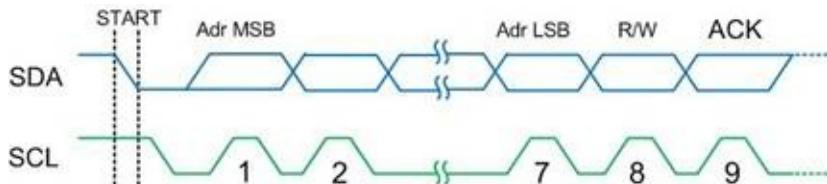


Figure 9-6: Address Byte Format in I2C

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. There are 8 more reserved addresses. That means  $111 = (128-1-8-8)$  device can share an I2C bus. In I2C bus the MSB of the address is transmitted first. The I2C bus also supports 10-bit address where the address is split into two frames at the beginning of the transmission. For the rest of the discussion, we will focus on 7 bit address only.

The 8th bit in the address 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.

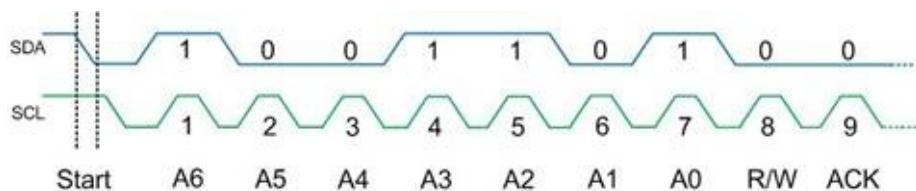
### Example 9-3

Show how a master initiates 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 respond 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 for 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)

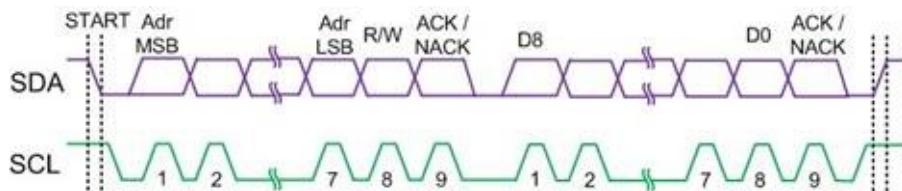


Figure 9-7: Typical Data Transmission

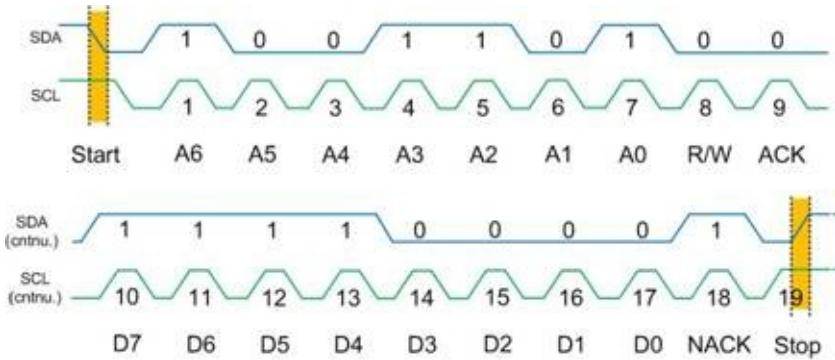
### Example 9-4

Show how a master writes data value 1111 0000 to a slave with an address 1001 101?

**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 1001 101 0 on the bus. The first seven bits (1001 101) indicates the slave address and the 8th bit (0) indicates a 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
- 4) After receiving the ACK, the master will transmit the data byte (1111 0000) 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 I2C protocol is clock stretching. It is used by a slow slave device to synchronize with the master. 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. Clock stretching can be used to slow down the clock for each bit or it can be used to temporarily halt the clock at the end of a byte while the receiver is processing the data.

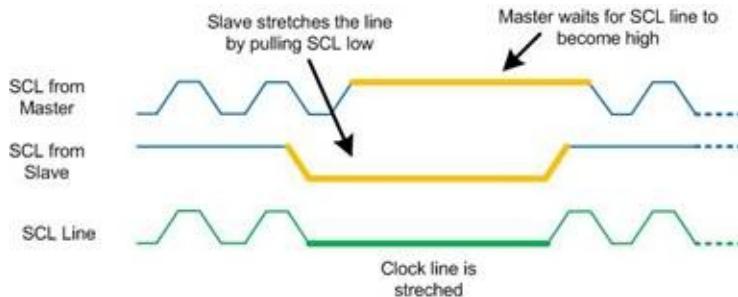


Figure 9-8: Clock Stretching

## 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.

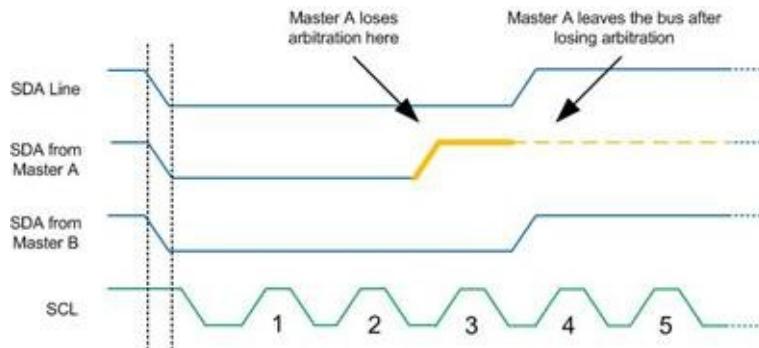
Each master has to check the level of the bus and compare it with the levels it is driving; if it doesn't match, that master has lost the arbitration, and will switch to slave mode. In the case of arbitration, the winning master will continue the transmission. Notice that neither the bus is corrupted 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, 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 memory address of the first location.
4. The master transmits the data for the first memory 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.

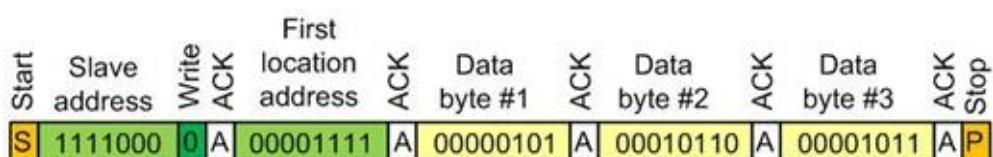


Figure 9-9: Multi-byte Burst Write

### 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 memory address).
3. The master transmits the memory 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 generates a STOP condition.

Figure 9-10 shows how to read three consecutive locations starting from location 00001111 of slave number 1111000.

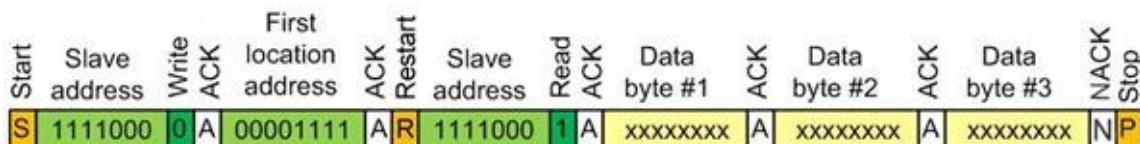


Figure 9-10: Multi-byte Burst Read

## 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 acknowledgement?
3. True or false. START and STOP conditions are generated when the SDA is high.
4. What is the name of the procedure a slow slave device uses to synchronize with a fast master?
5. True or false. After arbitration of two masters, both of them must start transmission from beginning.



## Section 9.2: I2C Programming in Freescale ARM KL25Z

The Freescale KL25Z chip comes with two on-chip I2C modules. In this section, we examine the registers and features of I2C module. The I2C modules are located at the following base addresses:

SSI Module	Base Address
I2C0	0x4006 6000
I2C1	0x4006 7000

Table 9-1: I2C Module Base Address for KL25Z

### Enabling Clock to I2C Module

To enable and use any of the peripherals, we must enable the clock to it. We use SIM\_SCGC4 register to enable the clock to I2C modules. We need to set bit D6 for I2C0 and bit D7 for I2C1 to enable the clock to I2C module. See Figure 9-11 and Table 9-2.

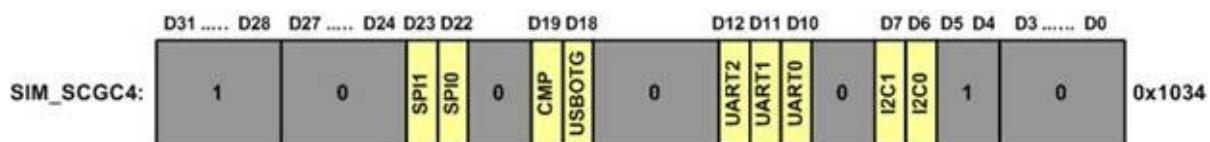


Figure 9-11: SIM\_SCGC4 register bits for enabling I2C clock

Bits	Name	Function	Description
6	I2C0	I2C 0 Clock Gating Control	1 to enable and 0 to disable
7	I2C1	I2C 1 Clock Gating Control	1 to enable and 0 to disable

Table 9-2: SIM\_SCGC4 Description

### I2C Clock speed

The I2Cx\_F (I2C Frequency divider) register allows us to set the clock rate for the SCL. The SCL clock comes from the bus clock and goes through clock divider circuit controlled by this register. The I2C provides a prescaler for the SCL. It does divide by 1, divide by 2, or divide by 4. The upper 2 bits (D7:D6) of I2Cx\_F register are used to select the above three options. See Figure 9-12 and Table 9-3.

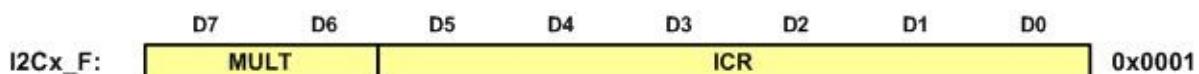


Figure 9-12: I2C\_F Register to Set I2C Baud Rate

Bits	Field	Descriptions
		This field defines the multiply factor to the SCL divider.

		MULT value	Description
6-7	MULT	00	Multiply by 1
		01	Multiply by 2
		10	Multiply by 4
		11	Reserved
0-5	ICR	I2C Clock Rate: this field defines the prescaler value.	

Table 9-3: I2C\_F Register

We use the following formula to set the I2C baud rate:

$$\text{I2C Baud Rate} = \text{Bus Clock} / (\text{MUL} \times \text{SCL divider})$$

In the above formula, I2C baud rate gives us the I2C clock rate for SCL, MUL is 1, 2, or 4, and the SCL divider is decided by the lower 6 bits of I2Cx\_F register. See Figure 9-12. The SCL divider is indexed by ICR bits of the I2Cx\_F register. The indexed values can be found in the following table.

ICR (Hex)	SCL divider	ICR (Hex)	SCL divider	ICR (Hex)	SCL divider	ICR (Hex)	SCL divider
00	20	10	48	20	160	30	640
01	22	11	56	21	192	31	768
02	24	12	64	22	224	32	896
03	26	13	72	23	256	33	1024
04	28	14	80	24	288	34	1152
05	30	15	88	25	320	35	1280
06	34	16	104	26	384	36	1536
07	40	17	128	27	480	37	1920
08	28	18	80	28	320	38	1280
09	32	19	96	29	384	39	1536
0A	36	1A	112	2A	448	3A	1792
0B	40	1B	128	2B	512	3B	2048
0C	44	1C	144	2C	576	3C	2304

0D	48	1D	160	2D	640	3D	2560
0E	56	1E	192	2E	768	3E	3072
0F	68	1F	240	2F	960	3F	3840

**Table 9-4: ICR and SCL Divider (From KL25Z Reference Manual)**

The I2C baud rate can go up to 100 kHz for Standard Mode, up to 400 kHz for Fast Mode, and up to 3.4 MHz for High-speed Mode. The smallest SCL divider value is 20 so the highest speed for KL25Z I2C module is (bus clock / 20).

See Examples 9-6 and 9-7.

### Example 9-6

Assume the Bus Clock frequency is 8MHz. Find the values for the I2C\_F register if we want I2C clock of (a) 100Kbps, (b) 400Kbps, and (c) 1Mbps.

#### Solution:

Using 8MHz for the Bus Frequency, we have:

$$I2C \text{ Baud Rate} = (\text{Bus\_Freq}) / (\text{MUL} \times \text{SCL Divider})$$

$$(a) 100,000 = (8\text{MHz}) / (1 \times 80)$$

That means the D7:D6=00 for MUL of 1 and D5-D0 = 0x14 for ICR to get SCL divider of 80. See Table 9-4. Now, I2Cx\_F=0001 0100

$$(b) 400,000 = (8\text{MHz}) / (1 \times 20)$$

That means the D7:D6=00 for MUL of 1 and D5-D0= 0x00 for ICR to get SCL divider of 20. Now, I2Cx\_F=0000 0000

$$(c) 1,000,000 = (8\text{MHz}) / (1 \times 8)$$

The smallest SCL divider value possible is 20. With 8MHz bus clock, it cannot achieve 1 MHz I2C clock.

### Example 9-7

Find the values for the I2C\_F for the (a) the lowest and (b) the Highest I2C baud rate if Bus Clock=13.98MHz.

## Solution:

$$I2C \text{ Baud Rate} = (\text{Bus Freq}) / (\text{MUL} \times \text{SCL divider})$$

(a) I2C Baud Rate =  $13.98\text{MHz} / (4 \times 3840) = 910 \text{ bps}$ . Therefore,  $I2Cx_F = 10111111 = 0xBF$

(b) I2C Baud Rate =  $13.98\text{MHz} / (1 \times 20) = 699 \text{ kbps}$ . Therefore,  $I2Cx_F = 00000000 = 0x00$

---

## Master or Slave

The I2C Module inside the KL25Z device can be either the Master or the Slave. We use  $I2Cx_C1$  (I2C Control 1 register) to designate the KL25Z device as master or slave. Setting bit D5 to 1 makes the I2C of KL25Z device as Master. We also need to make bit D7 = 1 to enable the I2C module. The D6 bit is used if we want to use Interrupt. D4 = 1 for Transmit. If we want to send an acknowledgement after the byte is received, then we make D3 = 1. Therefore, we need  $I2Cx_C1 = 10111000 = 0xB8$  for Master. See Figure 9-13 and Table 9-5.

$I2Cx_C1:$	D7	D6	D5	D4	D3	D2	D1	D0	
	IICEN	IICIE	MST	TX	TXAK	TSTA	WUEN	DMAEN	0x0002

Figure 9-13:  $I2Cx_C1$  Control register

Bits	Name	Function	Description
7	I2CEN	I2C Enable	0: Disable, 1: Enable
5	MST	I2C Master Mode Select	1: Enable Master mode, 0: Slave mode
4	TX	I2C Transmit Mode Select	1: Transmit, 0: Receive
3	TXAK	Transmit Acknowledge Enable	1: Enable ACK, 0: Disable ACK

Table 9-5:  $I2Cx_C1$  register bit Description

## Slave Address

When the KL25Z device is designated as a slave, it needs to have a calling address so that it can be addressed by the master by its slave address. We use  $I2Cx_A1$  (I2C Address 1) register to hold the address as the slave device. Notice, the addresses in I2C are only 7 bits (maximum of 127 devices). In the  $I2Cx_A1$  register, the D7-D1 bits are used for the slave address and the LSB of D0 is unused and is 0 by default. (The KL25Z has the option of 10-bit address, as well. See the KL25Z reference manual for more information.)

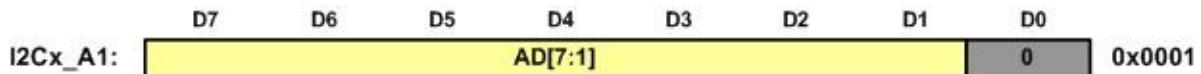


Figure 9-14: I2Cx\_A1 slave address register

## Data Register

In Master transmit mode, we place a byte of data in I2Cx\_D (I2C Data) register for transmission. This is an 8-bit register.

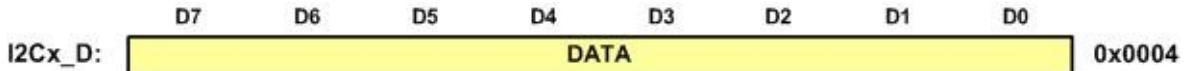


Figure 9-15: I2Cx\_D Data register

## Status Register

We use the I2Cx\_S (I2C Status) register for status. Upon reading this register, we get the status to see if a byte has been transmitted and ready for the next byte.

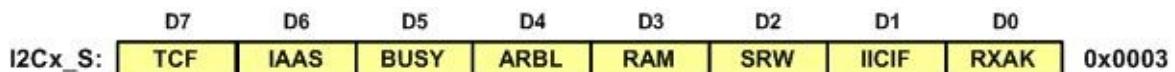


Figure 9-16: I2Cx\_S Register

Bit	Field	Description
7	TCF	Transfer Complete Flag (0: Transfer in progress, 1: Transfer complete) Note: The flag is cleared by reading the I2C_D register in receive mode and writing to the I2C_D register in transmit mode.
6	IAAS	Addressed As A Slave: The flag is set if the microcontroller is addressed by another device on the I2C bus. (0: Not addressed, 1: Addressed as a slave)
5	BUSY	Bus Busy (0: Bus is idle, 1: Bus is busy)
4	ARBL	Arbitration Lost (0: Standard bus operation, 1: Loss of arbitration) Note: The bit must be cleared by software, by writing a 1 to it.
3	RAM	Range Address Match: The flag is set if the received calling address matches the address range of the microcontroller. (0: Not addressed, 1: Addressed as a slave)
2	SRW	Slave Read/Write: When the microcontroller is called on the bus, the bit indicates the R/W bit. (0: the microcontroller must receive, 1: microcontroller must transmit.)
1	IICIF	I2C Interrupt Flag (0: No interrupt pending, 1: interrupt pending)
0	RXAK	Receive Acknowledge (0: acknowledge signal was received, 1: No acknowledge signal was received)

**Table 9-6: I2Cx\_R Register**

After a START occurs on the bus, the bus goes busy (BUSY bit 5) until a STOP is sent. Unless you are the one that sent the START and the slave address was sent without Arbitration Lost (ARBL bit 4), you should stay away from the bus because some other I2C is the bus master. If you do not own the bus and you start sending data, a bus collision will result in the transmission failure. So it is important to monitor the BUSY bit in the I2Cx\_S register before sending the START and slave address. Also, check ARBL bit after the slave address is sent before sending more data.

After writing to the data register, the transmission starts and the TCF (bit 7) of I2Cx\_S register will change to 0. It stays at 0 until the byte is transmitted. The program should check IICIF (bit 1) of I2Cx\_S register. The IICIF bit is set when the transmission is terminated whether it is complete or not. Upon the termination of the transmission, the program should check the RXAK (bit 0) of I2Cx\_S register to see whether the receiver acknowledged the reception of the data. If RXAK bit is set after a transmission, an error occurred.

## Addresses for I2C modules

Below are the addresses for some of the major registers of I2C modules:

Address	Register
4006 6000	I2C Address Register 1 (I2C0_A1)
4006 6001	I2C Frequency Divider register (I2C0_F)
4006 6002	I2C Control Register 1 (I2C0_C1)
4006 6003	I2C Status register (I2C0_S)
4006 6004	I2C Data I/O register (I2C0_D)
4006 6005	I2C Control Register 2 (I2C0_C2)
4006 7000	I2C Address Register 1 (I2C1_A1)
4006 7001	I2C Frequency Divider register (I2C1_F)
4006 7002	I2C Control Register 1 (I2C1_C1)
4006 7003	I2C Status register (I2C1_S)
4006 7004	I2C Data I/O register (I2C1_D)
4006 7005	I2C Control Register 2 (I2C1_C2)

**Table 9-7: Addresses of some I2C Registers**

## 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. See Appendix A.

Figure 9-1 in the last section showed the physical connection of the I2C buses. For the I2C Module inside the Freescale KL25Z ARM, we must enable the open-drain option for the I/O pins used by the I2C buses. The I2C pins are configured as open-drain output when the pins are assigned to an I2C module as described next.

## Configuring GPIO for I2C

In using I2C, we must configure the GPIO pins to allow the connections of the I2C SCL and SDA functions to two GPIO pins of the device. In this regard, it is same as all other peripherals. The steps are as follow:

1. Enable the clock to GPIO pins using SIM\_SCGC5.
2. Assign the I2C signals to specific pins using PORTx\_PCR register. See Table 9-8.

I2C Module	I/O Pin(ALT <sub>x</sub> )	I/O pin(ALT <sub>x</sub> )	I/O pin(ALT <sub>x</sub> ) Pin
I2C0SCL	PTE24(ALT4)	PTA3(ALT2)	PTC8(ALT2)
I2C0SDA	PTE25(ALT4)	PTA4(ALT2)	PTC9(ALT2)
I2C1SCL	PTE1(ALT6)	PTC1(ALT2)	PTC10(ALT2)
I2C1SDA	PTE0(ALT6)	PTC2(ALT2)	PTC11(ALT2)

---

Table 9-8: I2C Pin Assignment

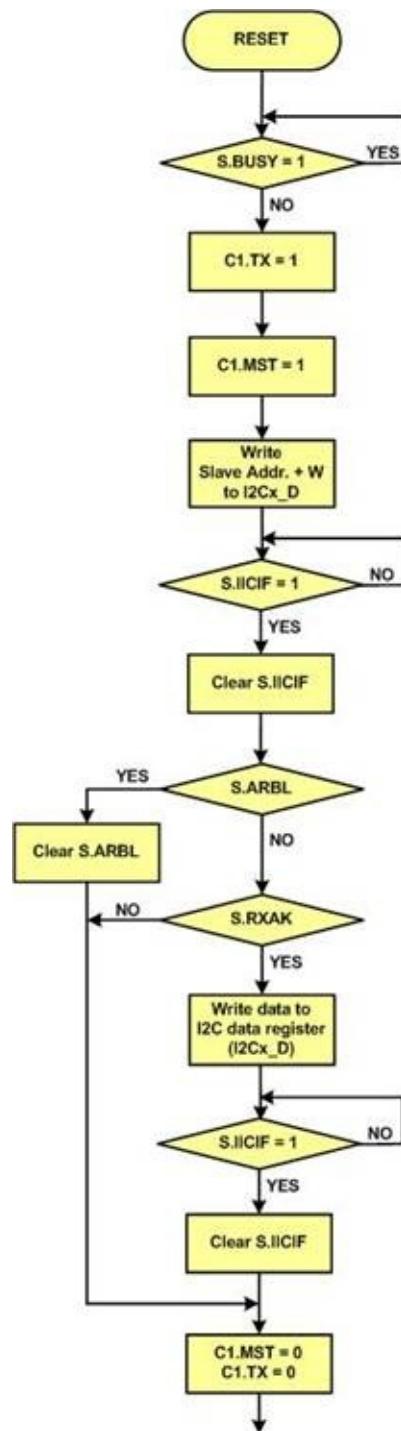
## Configuring I2C for data transmission

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 SIM\_SCGC4.
2. Disable the I2C module by writing a 0 to the I2Cx\_C1 register.
3. Set the I2C clock speed using I2Cx\_F frequency divider register.
4. Enable I2C module by setting IICEN (bit 7) of I2Cx\_C1 register.

## Send a byte of data to a slave device

1. Poll the BUSY bit of I2Cx\_S register until the bus is not busy.
2. Set TX (bit 4) of I2Cx\_C1 register for transmitting.
3. Set MST (bit 5) of I2Cx\_C1 register to put the I2C module in master mode and generate a START on the bus.
4. Write the Slave address + W to I2Cx\_D register to send the slave address.
5. Poll IICIF (bit 1) of I2Cx\_S until the transmission is terminated.
6. Clear IICIF bit by writing a 1 to it.
7. Check ARBL (bit 4) of I2Cx\_S to see whether the arbitration was lost. If yes, clear ARBL bit by writing a 1 to it and abort the transmission.
8. Check RXAK (bit 0) of I2Cx\_S to see whether the slave sent an ACK. If not, abort the transmission.
9. Write the data byte to I2Cx\_D register to send the data.
10. Poll IICIF (bit 1) of I2Cx\_S until the transmission is complete.
11. Clear IICIF bit by writing a 1 to it.
12. Check RXAK (bit 0) of I2Cx\_S to see whether the slave sent an ACK.
13. Clear MST (bit 5) and TX (bit 4) of I2Cx\_C1 register to generate a STOP on the bus. See figure below.



**Figure 9-17: Master Single Transmit**

## Review Questions

- True or false. The I2C module in Freescale ARM KL25Z supports speed up to Bus speed.
- True or false. The I2Cx\_C1 is used to enable the I2Cx module.
- True or false. There is no CS (chip select) pin in I2C.
- In Freescale ARM KL25Z, which register is used to enable the clock to I2C module?
- In Freescale ARM KL25Z, which register is used to set the I2C baud rate?



## Section 9.3: DS1337 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 system 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 DS1337 is a serial RTC with an I2C bus. In this section, we interface and program the DS1337 RTC. According to the DS1337 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 DS1337 has a built-in power-sense circuit that detects power failures and automatically switches to the battery supply." The DS1337 does not support the Daylight Savings Time option. Next, we describe the pins of the DS1337. See Figure 9-18.

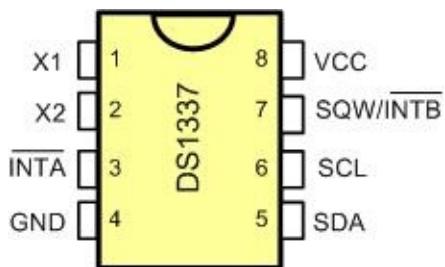


Figure 9-18: DS1337 Pins

The DS1337 is used as replacement for popular DS1307 if system voltage is 3.3V.

### X1-X2

These are input pins that allow the DS1337 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.

### VCC

Pin 8 is used as the primary voltage supply to the chip. The voltage source can be between 1.3 V to 5.5 V. When Vcc is above 1.3 V, the DS1337 starts working and keeps the time. But the I2C interface is disabled unless the Vcc is above 1.8 V.

Vcc can be connected to an external battery, thereby providing the power source to the chip when the external supply voltage is not available.

### 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 I2C bus.

### SCL (Serial Clock)

Pin 6 is the SCL pin and must be connected to the SCL line of the I2C bus.

### INTA# (Interrupt A)

The DS1337 has two Alarms: Alarm 1 and Alarm 2. If the alarm 1 is enabled, the INTA pin is asserted when the current time and date matches the values of Alarm 1 registers.

### SWQ/INTB

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. The pin can be used as the output for INTB, as well. For more information, see the DS1337 datasheet.

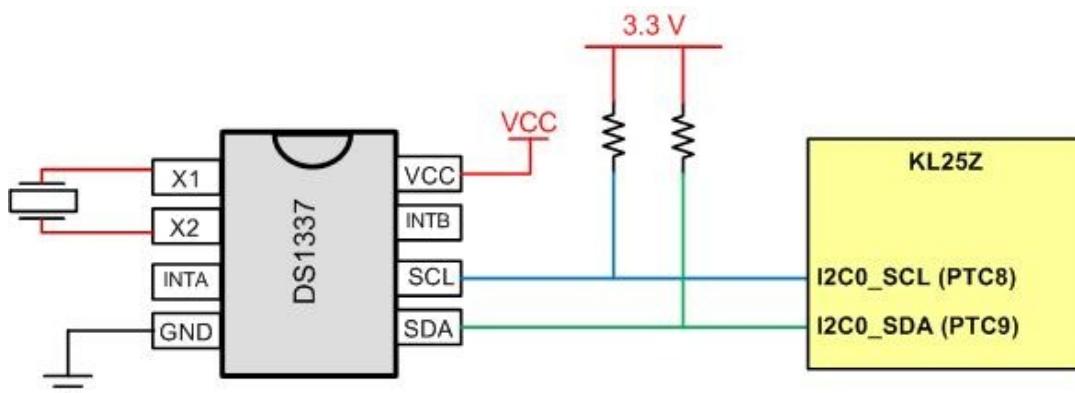


Figure 9-19: DS1337 Connections

### Address map of the DS1337

The DS1337 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. Locations 07H-0DH are set aside for Alarm 1 and Alarm 2 registers. The next bytes are used for control and status registers. Table 9-9 shows the address map of the DS1337. Next, we study the control register, and time and date access in DS1337.

Address	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Function	Range
00H	0		10 Seconds			Seconds			Seconds	00-59
01H	0		10 Minutes			Minutes			Minutes	00-59
02H	0	12/24	10 hour 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	Century	0	0	10Mnt	Month			Month Century	1-12+ Century		
06H	10 Year				Year			Year	00-99		
07H	A1M1	10 Seconds			Seconds			Alarm 1 Seconds	00-59		
08H	A1M2	10 Minutes			Minutes			Alarm 1 Minutes	00-59		
09H	A1M3	12/24	AM/PM	10 Hour	Hour			Alarm 1 Hours	1-12 00-23		
			10 Hour								
0AH	A1M4	DY/DT	10 Date	Day			Alarm 1 Day	1-7			
				Date			Alarm 1 Date	01-31			
0BH	A2M2	10 Minutes			Minutes			Alarm 2 Minutes	00-59		
0CH	A2M3	12/24	AM/PM	10 Hour	Hour			Alarm 2 Hours	1-12 00-23		
			10 Hour								
0DH	A2M4	DY/DT	10 Date	Day			Alarm 2 Day	1-7			
				Date			Alarm 2 Date	01-31			
0EH	EOSC	0	0	RS2	RS1	INTCN	A2IE	A1IE	Control		
0FH	OSF	0	0	0	0	0	A2F	A1F	Status		

Table 9-9: DS1337 Address Map

### Time and date address locations and modes

The byte addresses 0–6 are set aside for the time and date, as shown in Table 9-9. The DS1337 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.

## The DS1337 control register

As shown in Table 9-9, the control register has an address of 0EH. In the DS1337 control register, the bits control the function of the SQW/INTB and INTA pins. Figure 9-20 shows the simplified diagram for SQW/INTB pin.

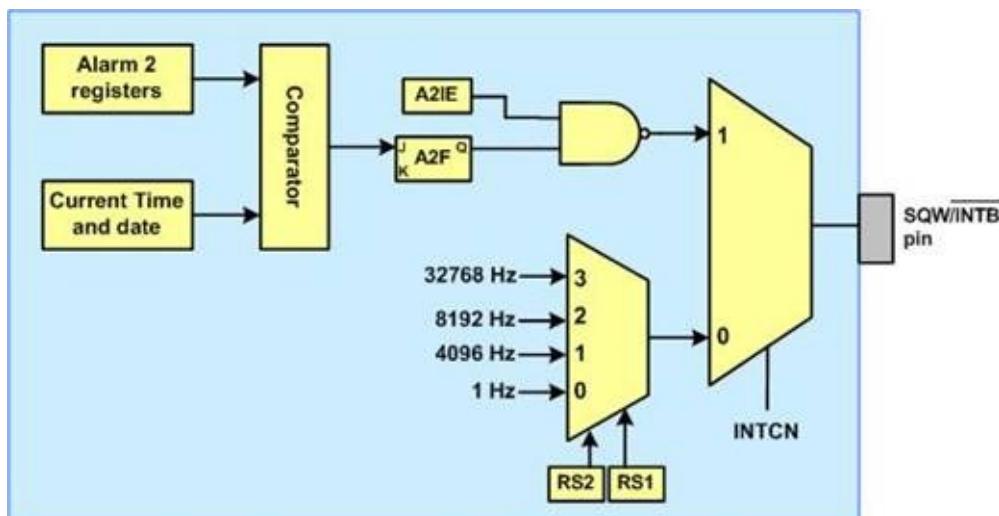


Figure 9-20: Simplified Structure of SQW/INTB Pin

The SQW/INTB pin can be used as a square wave generator or an interrupt generator. When the INTCN bit of control register is 0, the pin works as a wave generator. Using the RS2 and RS1 bits, the frequency of the generated wave is chosen. RS2-RS1 (rate select) bits select the output frequency of the generated wave according to Table 9-10.

RS2	RS1	Output Frequency
0	0	1 Hz
0	1	4.096 kHz
1	0	8.192 kHz
1	1	32.768 kHz

Table 9-10: RS bits

When INTCN = 1, the SQW/INTB works as an interrupt generator. Locations 0BH-0DH of DS1337 memory are related to Alarm 2. The contents of the Alarm 2

registers are compared with the values current time and date (locations 00H-06H). When the current date and time matches the alarm 2 values, the A2F flag of status register (location 0FH) goes high. If the A2IE (Alarm2 Interrupt Enable) bit of the control register is set, the INTB becomes 0. The pin remains 0 until the A2F flag is cleared by software. To clear the A2F flag, write 0 into it.

It can make an interrupt every minute, hour, day, or date. The bit 7 of alarm registers, are mask registers. If it is 0, the value of the register is compared with the timekeeping registers; otherwise, it is masked. Table 9-11 shows how to make interrupts every minute, hour, day, or date.

DY/DT	A2M4	A2M3	A2M2	Alarm Rate
X	1	1	1	Alarm once per minute
X	1	1	0	Alarm when minutes match
X	1	0	0	Alarm when hours and minutes match
0	0	0	0	Alarm when date, hours, and minutes match
1	0	0	0	Alarm when day, hours, and minutes match

Table 9-11: Alarm 2 Register Mask Bits

The bit 7 of the control register is EOSC (Enable Oscillator) bit. This bit is active low. If it is 0, the oscillator works.

## Register pointer

In DS1337, 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. This is useful in multi-byte read or write. When it points to location 0x0F, in the next read/write it rolls over to 0.

## Writing to DS1337

To set the value of the register pointer and write one or more bytes of data to DS1337, you can use the following steps:

1. To access the DS1337 for a write operation, after sending a START condition, you should transmit the address of DS1337 (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 DS1337 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 DS1337***

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 DS1337 you should do the following steps:

1. To access the DS1337 for a read operation, you need to set the register pointer first. After sending a START condition, you should transmit the address of DS1337 (1101 000) followed by 0 to indicate a write operation (writing the register pointer).
2. Check the acknowledge bit to be sure that DS1337 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 DS1337 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 DS1337 (1101 000) followed by 1 to indicate a read operation. Check the acknowledge bit to be sure that DS1337 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 DS1337 that the burst read is complete.
7. Transmit a STOP bit condition.

## ***Setting the Time of DS1337***

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 DS1337 using single byte write**

```

/* p9_1: I2C single byte write to DS1337

This program communicates with the DS1337 Real-time Clock via I2C.

DS1337 is a newer version of DS1307. It operates from 1.8V to 5.5V.

It has century and alarms added.

The time day (of the week), date, month and year are written using single
byte write. It sets the date to Monday Oct. 19th, 2009.

DS1337 I2C parameters:

fmax = 100 kHz, SCL PTE1, SDA PTE0
*/
#include "MKL25Z4.h"

#define SLAVE_ADDR 0x68      /* 1101 000. */
#define ERR_NONE 0
#define ERR_NO_ACK 0x01
#define ERR_ARB_LOST 0x02
#define ERR_BUS_BUSY 0x03

void I2C1_init(void);

int I2C1_byteWrite(unsigned char slaveAddr, unsigned char memAddr, unsigned
char data);

void delayUs(int n);

int main(void)
{
    unsigned char timeDateToSet[] = {0x55, 0x58, 0x16, 0x01, 0x19, 0x10, 0x09};
    int rv;
    int i;

    I2C1_init();

    for (i = 0; i < 7; i++) {
        rv = I2C1_byteWrite(SLAVE_ADDR, 0, timeDateToSet[0]);
        if (rv != ERR_NONE)
            for(;;) /* replace with error handling */
    }
}

```

```

for (;;)
{
}

/* initialize I2C1 and the port pins */
void I2C1_init(void) {

    SIM->SCGC4 |= 0x80;           /* turn on clock to I2C1 */
    SIM->SCGC5 |= 0x2000;          /* turn on clock to PortE */
    PORTE->PCR[1] = 0x0600;       /* PTE1 I2C1 SCL */
    PORTE->PCR[0] = 0x0600;       /* PTE0 I2C1 SDA */

    I2C1->C1 = 0;                /* stop I2C1 */
    I2C1->S = 2;                 /* Clear interrupt flag */
    I2C1->F = 0x1C;              /* set clock to 97.09KHz @13.981MHz bus clock */
    /* See Table 9-4. */
    I2C1->C1 = 0x80;              /* enable I2C1 */
}

/* Write a single byte to slave memory.
 * write: S-(saddr+w)-ACK-maddr-ACK-data-ACK-P
 */
int I2C1_byteWrite(unsigned char slaveAddr, unsigned char memAddr, unsigned
char data) {

    int retry = 1000;

    while (I2C1->S & 0x20) {      /* wait until bus is available */
        if (--retry <= 0)
            return ERR_BUS_BUSY;
        delayUs(100);
    }

    /* send start */
    I2C1->C1 |= 0x10;             /* Tx on */
    I2C1->C1 |= 0x20;             /* become master */
}

```

```

/* send slave address and write flag */
I2C1->D = slaveAddr << 1;

while(!(I2C1->S & 0x02)); /* wait for transfer complete */

I2C1->S |= 0x02; /* clear IF */

if (I2C1->S & 0x10) { /* arbitration lost */

I2C1->S |= 0x10; /* clear IF */

return ERR_ARB_LOST;

}

if (I2C1->S & 0x01) /* got NACK from slave */

return ERR_NO_ACK;

/* send memory address */

I2C1->D = memAddr;

while(!(I2C1->S & 0x02)); /* wait for transfer complete */

I2C1->S |= 0x02; /* clear IF */

if (I2C1->S & 0x01) /* got NACK from slave */

return ERR_NO_ACK;

/* send data */

I2C1->D = data;

while(!(I2C1->S & 0x02)); /* wait for transfer complete */

I2C1->S |= 0x02; /* clear IF */

if (I2C1->S & 0x01) /* got NACK from slave */

return ERR_NO_ACK;

/* stop */

I2C1->C1 &= ~0x30;

return ERR_NONE;
}

/* delay n microseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */

void delayUs(int n)

{
    int i; int j;

    for(i = 0 ; i < n; i++) {

```

```
|     for(j = 0; j < 7; j++) ;  
| }  
| }
```

## Setting the date of DS1337 in KL25Z

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 DS1337 using burst write

```
/* p9_2: I2C burst write to DS1337 */  
  
/* This program communicates with the DS1337 Real-time Clock via I2C. The time,  
day (of the week), date, month and year are written using burst write. It sets  
the date to Monday Oct. 19th, 2009.  
  
DS1337 I2C parameters:  
fmax = 100 kHz, SCL PTE1, SDA PTE0  
*/  
  
#include "MKL25Z4.h"  
  
#define SLAVE_ADDR 0x68      /* 1101 000. */  
#define ERR_NONE 0  
#define ERR_NO_ACK 0x01  
#define ERR_ARB_LOST 0x02  
#define ERR_BUS_BUSY 0x03  
  
void I2C1_init(void);  
int I2C1_burstWrite(unsigned char slaveAddr, unsigned char memAddr, int  
byteCount, unsigned char* data, int* cnt);  
void delayUs(int n);  
  
int main(void)
```

```

{

    unsigned char timeDateToSet[] = {0x55, 0x58, 0x16, 0x01, 0x19, 0x10, 0x09};

    int count;

    int rv;

    I2C1_init();

    /* use burst write to write day, date, month, and year */

    rv = I2C1_burstWrite(SLAVE_ADDR, 0, 7, timeDateToSet, &count);

    if (rv)

        for(;;) ; /* replace with error handling */

    for (;;) {

    }

}

/* initialize I2C1 and the port pins */

void I2C1_init(void) {

    SIM->SCGC4 |= 0x80;           /* turn on clock to I2C1 */

    SIM->SCGC5 |= 0x2000;          /* turn on clock to PortE */

    PORTE->PCR[1] = 0x0600;       /* PTE1 I2C1 SCL */

    PORTE->PCR[0] = 0x0600;       /* PTE0 I2C1 SDA */

    I2C1->C1 = 0;                /* stop I2C1 */

    I2C1->S = 2;                 /* Clear interrupt flag */

    I2C1->F = 0x1C;              /* set clock to 97.09KHz @13.981MHz bus clock */

    I2C1->C1 = 0x80;              /* enable I2C1 */

}

/* Use burst write to write multiple bytes to consecutive memory locations.

 * Burst write: S-(saddr+w)-ACK-maddr-ACK-data-ACK-data-ACK...-data-ACK-P
 */

int I2C1_burstWrite(unsigned char slaveAddr, unsigned char memAddr, int
byteCount, unsigned char* data, int* cnt) {

    int retry = 1000;

    *cnt = 0;
}

```

```

    while (I2C1->S & 0x20) { /* wait until bus is available */

    if (--retry <= 0)

    return ERR_BUS_BUSY;

    delayUs(100);

    }

    /* send start */

    I2C1->C1 |= 0x10; /* Tx on */

    I2C1->C1 |= 0x20; /* become master */

    /* send slave address and write flag */

    I2C1->D = slaveAddr << 1;

    while (!(I2C1->S & 0x02)); /* wait for transfer complete */

    I2C1->S |= 0x02; /* clear IF */

    if (I2C1->S & 0x10) { /* arbitration lost */

    I2C1->S |= 0x10; /* clear IF */

    return ERR_ARB_LOST;

    }

    if (I2C1->S & 0x01) /* got NACK from slave */

    return ERR_NO_ACK;

    /* send memory address */

    I2C1->D = memAddr;

    while (!(I2C1->S & 0x02)); /* wait for transfer complete */

    I2C1->S |= 0x02; /* clear IF */

    if (I2C1->S & 0x01) /* got NACK from slave */

    return ERR_NO_ACK;

    /* send data */

    while (byteCount- > 0) {

    I2C1->D = *data++;

    while (!(I2C1->S & 0x02)); /* wait for transfer complete */

    I2C1->S |= 0x02; /* clear IF */

    if (I2C1->S & 0x01) /* got NACK from slave */

    return ERR_NO_ACK;

    (*cnt)++;

    }

```

```

/* stop */
I2C1->C1 &= ~0x30;

return ERR_NONE;
}

/* delay n microseconds */
/* The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit(). */

void delayUs(int n)
{
    int i; int j;
    for(i = 0 ; i < n; i++) {
        for(j = 0; j < 7; j++) ;
    }
}

```

## ***Reading the date and time of DS1337 in KL25Z***

Program 9-3 shows how to read the date and time from DS1337 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 BUSBSY bit and arbitration.

### **Program 9-3:** **Reading date time of DS1337 using burst read**

```

/* p9_3: I2C burst read from DS1337 */

/*
This program communicates with the DS1337 Real-time Clock via I2C.
The time, day (of the week), date, month and year are read using burst read.

DS1337 I2C parameters:
fmax = 100 kHz, SCL PTE1, SDA PTE0 */

#include "MKL25Z4.h"
#include <stdio.h>

```

```

#include "UART.h"

#define SLAVE_ADDR 0x68      /* 1101 000. */

#define ERR_NONE 0
#define ERR_NO_ACK 0x01
#define ERR_ARB_LOST 0x02
#define ERR_BUS_BUSY 0x03

void I2C1_init(void);

int I2C1_burstRead(unsigned char slaveAddr, unsigned char memAddr, int
byteCount, unsigned char* data, int* cnt);

void delayUs(int n);

int main(void)
{
    unsigned char timeDateReadback[7];
    int count;
    int rv;

    I2C1_init();

    rv = I2C1_burstRead(SLAVE_ADDR, 0, 7, timeDateReadback, &count);
    if (rv)
        for(;;) ; /* replace with error handling */

    for (;;)
    {
    }

}

/* initialize I2C1 and the port pins */
void I2C1_init(void) {

    SIM->SCGC4 |= 0x80;          /* turn on clock to I2C1 */
    SIM->SCGC5 |= 0x2000;         /* turn on clock to PortE */
    PORTE->PCR[1] = 0x0600;       /* PTE1 I2C1 SCL */
    PORTE->PCR[0] = 0x0600;       /* PTE0 I2C1 SDA */

    I2C1->C1 = 0;                /* stop I2C1 */
    I2C1->S = 2;                 /* Clear interrupt flag */
    I2C1->F = 0x1C;              /* set clock to 97.09KHz @13.981MHz bus clock */
    I2C1->C1 = 0x80;              /* enable I2C1 */
}

```

```

}

/* Use burst read to read multiple bytes from consecutive memory locations.

Burst read: S-(saddr+w)-ACK-maddr-ACK-R-(saddr+r)-data-ACK-data-ACK-...-data-
NACK-P

*/
int I2C1_burstRead(unsigned char slaveAddr, unsigned char memAddr, int
byteCount, unsigned char* data, int* cnt) {
    int retry = 100;
    volatile unsigned char dummy;
    *cnt = 0;

    while (I2C1->S & 0x20) { /* wait until bus is available */
        if (--retry <= 0)
            return ERR_BUS_BUSY;
        delayUs(100);
    }

    /* start */
    I2C1->C1 |= 0x10;           /* Tx on */
    I2C1->C1 |= 0x20;           /* become master */

    /* send slave address and write flag */
    I2C1->D = slaveAddr << 1;
    while(!(I2C1->S & 0x02)); /* wait for transfer complete */
    I2C1->S |= 0x02;           /* clear IF */
    if (I2C1->S & 0x10)       /* arbitration lost */
        return ERR_ARB_LOST;
    if (I2C1->S & 0x01)       /* got NACK from slave */
        return ERR_NO_ACK;

    /* send address of target register in slave */
    I2C1->D = memAddr;
    while(!(I2C1->S & 0x02)); /* wait for transfer complete */
    I2C1->S |= 0x02;           /* clear IF */
    if (I2C1->S & 0x01)       /* got NACK from slave */
        return ERR_NO_ACK;

    /* restart */
    I2C1->C1 |= 0x04;          /* send Restart */

    /* send slave address and read flag */

```

```

I2C1->D = (slaveAddr << 1) | 1;
while(!(I2C1->S & 0x02)); /* wait for transfer complete */
I2C1->S |= 0x02; /* clear IF */
if (I2C1->S & 0x01) /* got NACK from slave */
    return ERR_NO_ACK;

/* change bus direction to read */
I2C1->C1 &= ~0x18; /* Tx off, prepare to give ACK */
if (byteCount == 1)
    I2C1->C1 |= 0x08; /* prepare to give NACK */
dummy = I2C1->D; /* dummy read to initiate bus read */

/* read data */
while (byteCount > 0) {
    if (byteCount == 1)
        I2C1->C1 |= 0x08; /* prepare to give NACK for last byte */
    while(!(I2C1->S & 0x02)); /* wait for transfer complete */
    I2C1->S |= 0x02; /* clear IF */
    if (byteCount == 1) {
        I2C1->C1 &= ~0x20; /* stop the bus before reading last byte */
    }
    *data++ = I2C1->D; /* read received data */
    byteCount--;
    (*cnt)++;
}

return ERR_NONE;
}

/* delay n microseconds */
/* The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit(). */

void delayUs(int n)
{
    int i; int j;
    for(i = 0 ; i < n; i++) {
        for(j = 0; j < 7; j++) ;
    }
}

```

## Review Questions

1. True or false. All of the RAM contents of the DS1337 are nonvolatile.
2. How many bytes of RAM in the DS1337 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 DS1337 are set aside for general-purpose applications?
  - (a) 7 bytes
  - (b) 8 bytes
  - (c) 56 bytes
  - (d) 64 bytes
4. True or false. The DS1337 has a single pin for data.
5. Which pin of the DS1337 is used for clock in I<sub>2</sub>C connection?
6. What is the common voltage for V<sub>bat</sub> in the DS1337?
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. Clock stretching.
5. False, the master who won the arbitration will continue.

## **Section 9-2**

1. False
2. True
3. True
4. SIM\_SCGC4
5. I2Cx\_F

## **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.

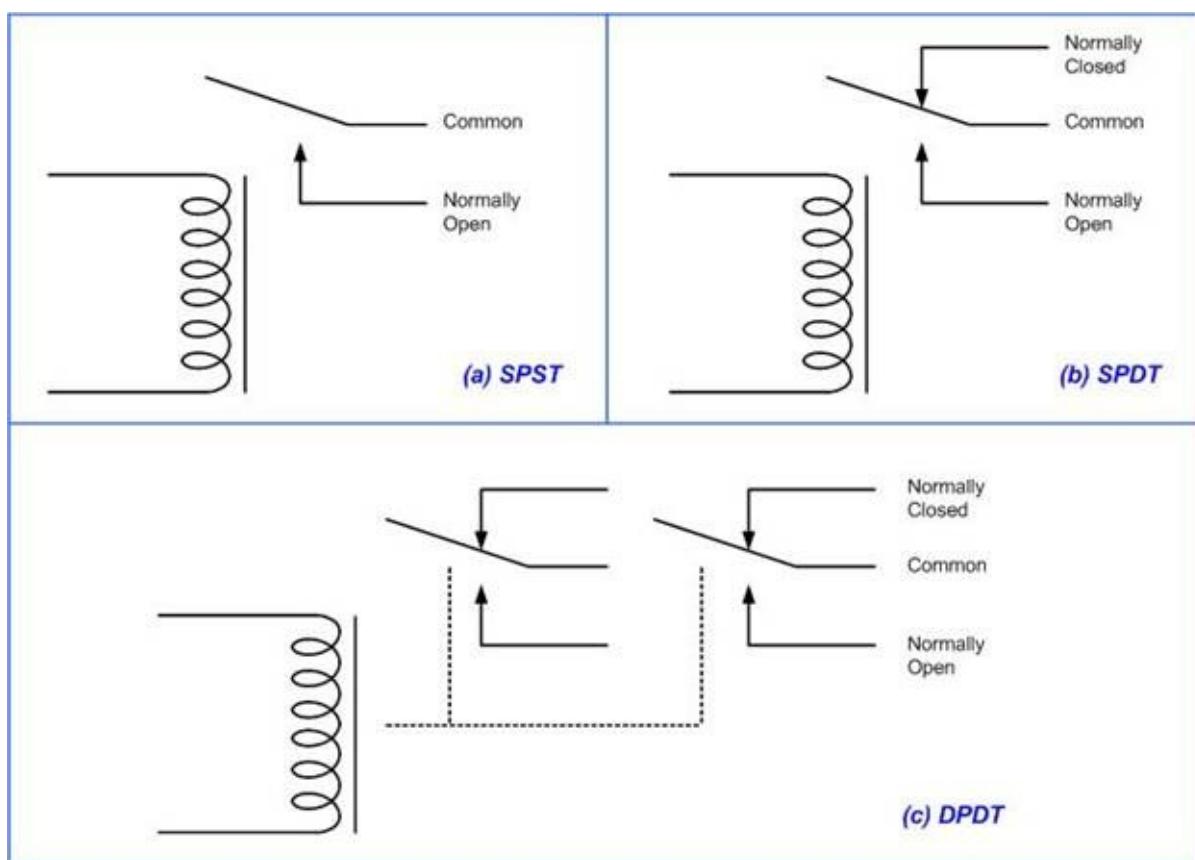


Figure 10-1: Relay Diagrams

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 PTD1.

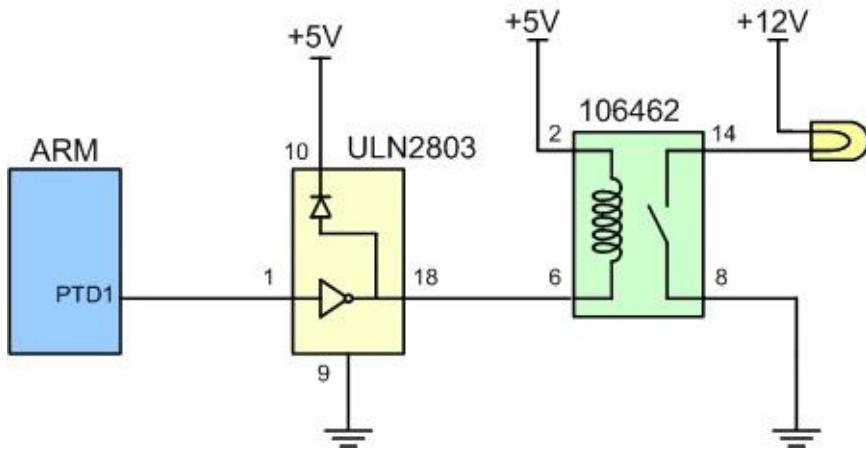


Figure 10-2: ARM Connection to Relay

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 connected to PTD1 on and off every second.
 */

#include <MKL25Z4.H>

int main (void) {
    void delayMs(int n);

    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[1] = 0x100;         /* make PTD1 pin as GPIO */
    PTD->PDDR |= 0x02;             /* make PTD1 as output pin */

    while (1) {
        PTD->PSOR = 0x02;          /* turn on PTD1 output */
        delayMs(1000);              /* wait for 1 second */
        PTD->PCOR = 0x02;           /* turn off PTD1 output */
        delayMs(1000);              /* wait for 1 second */
    }
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
```

```

int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}

```

## 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).

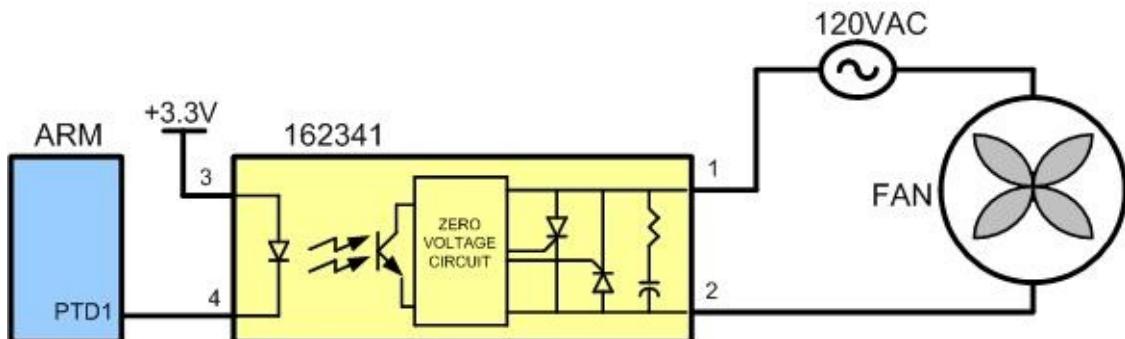


Figure 10-3: ARM Connection to a Solid-State Relay

## 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.

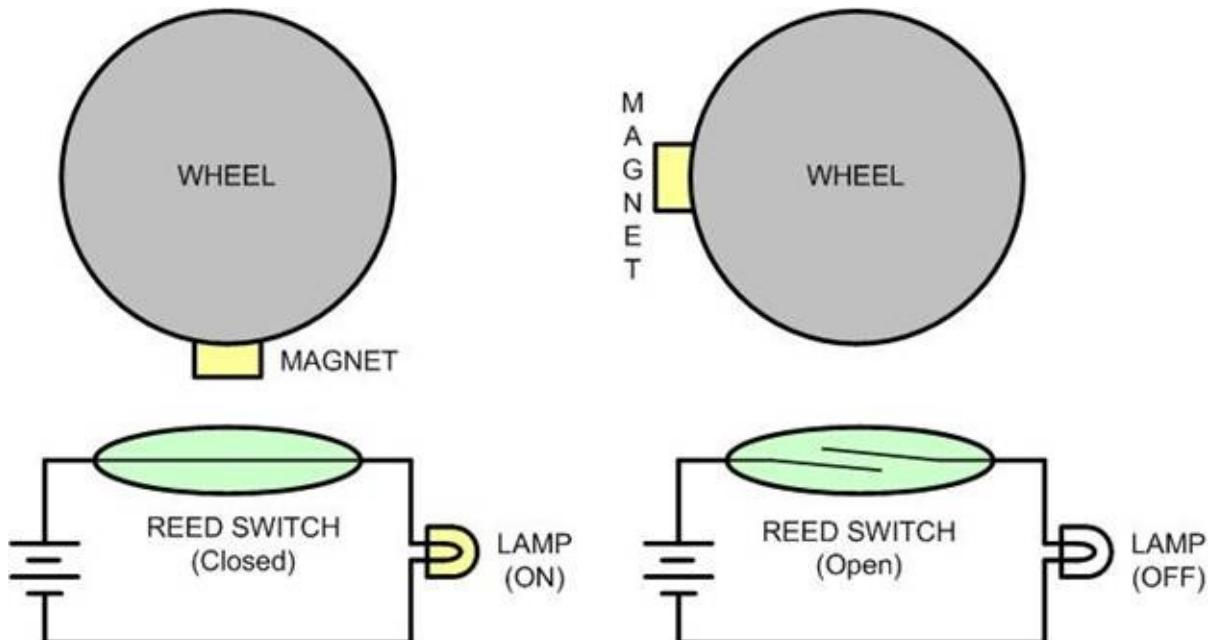


Figure 10-4: Reed Switch and Magnet Combination

## 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.

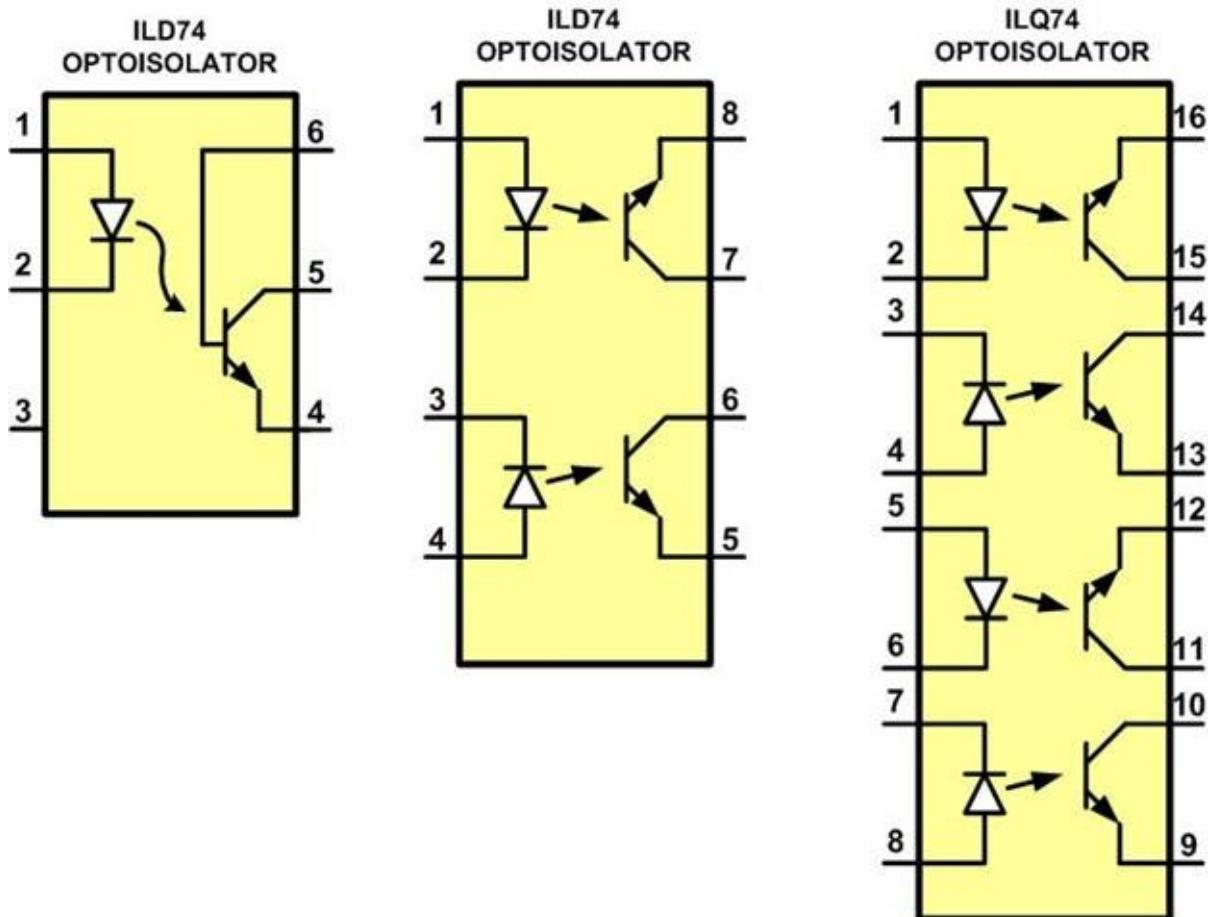


Figure 10-5: Optoisolator Package Examples

### *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.

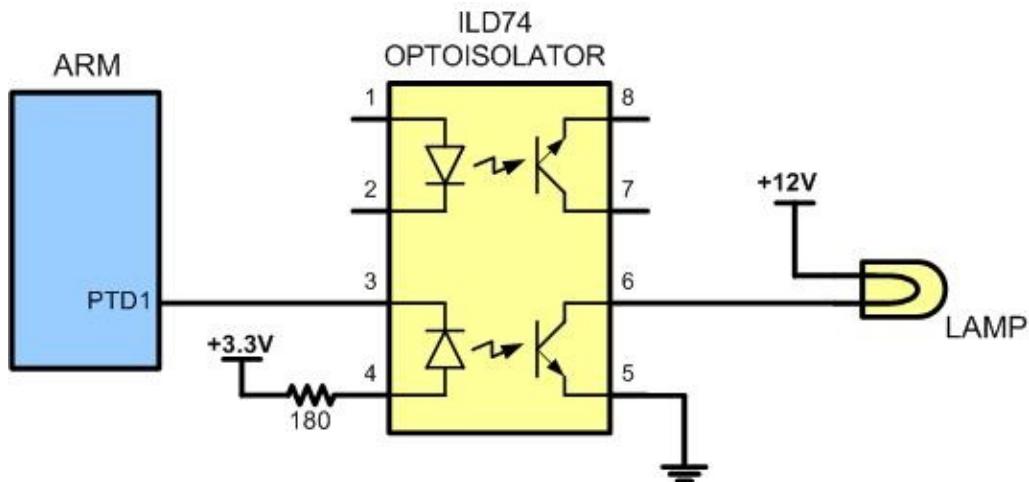


Figure 10-6: Controlling a Lamp via an Optoisolator

## 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).

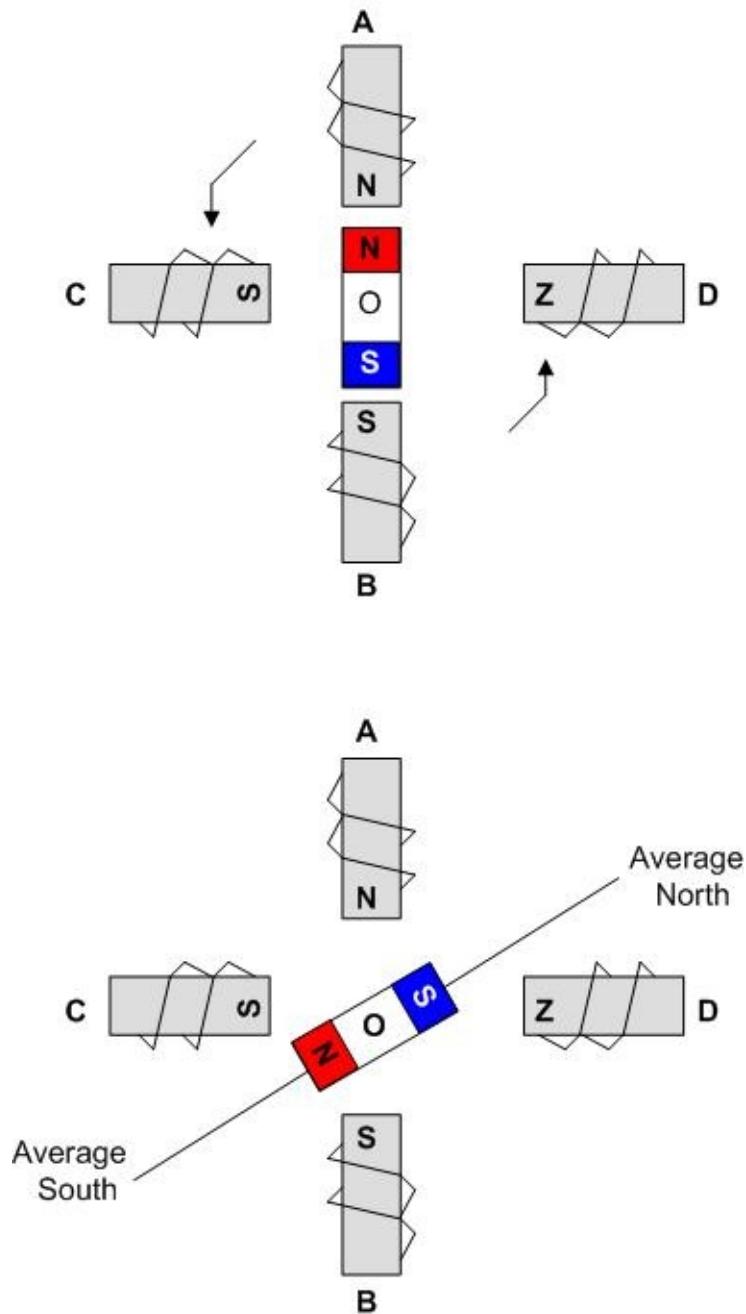


Figure 10-7: Rotor Alignment

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.

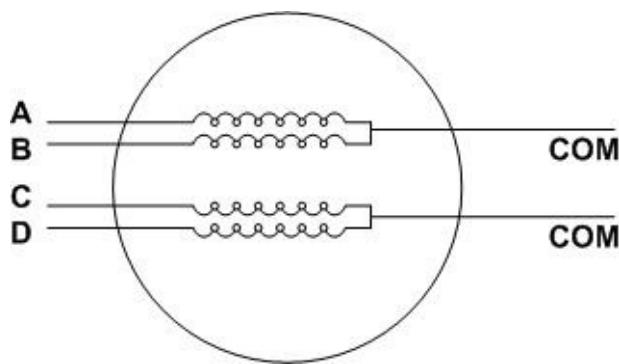


Figure 10-8: Stator Winding Configuration

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.

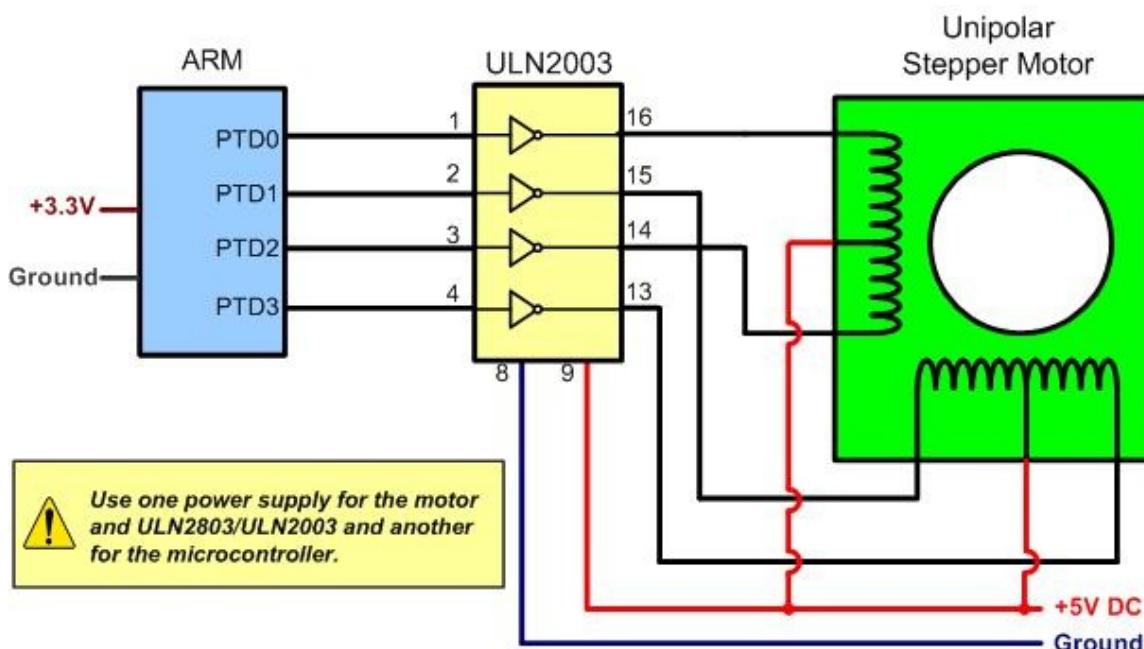


Figure 10-9: ARM Connection to Stepper Motor

### **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**

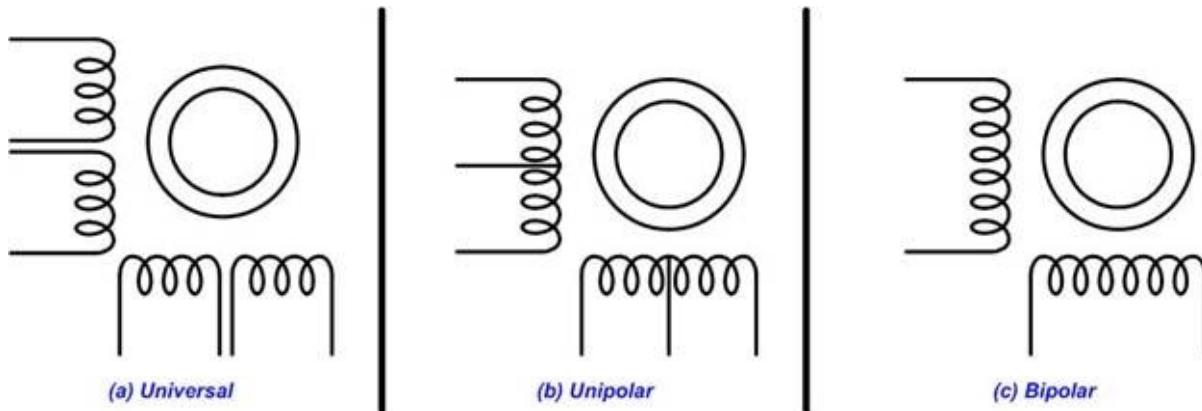
There are three common types of stepper motor interfacing: universal, unipolar, and bipolar. They can be identified by the number of connections to the motor. A universal stepper motor has eight, while the unipolar has six and the bipolar has four. The universal stepper motor can be configured for all 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

<b>151861CP</b>	7.5	unipolar	5 V	9 ohms	550 mA
<b>171601CP</b>	3.6	unipolar	7 V	20 ohms	350 mA
<b>164056CP</b>	7.5	bipolar	5 V	6 ohms	800 mA

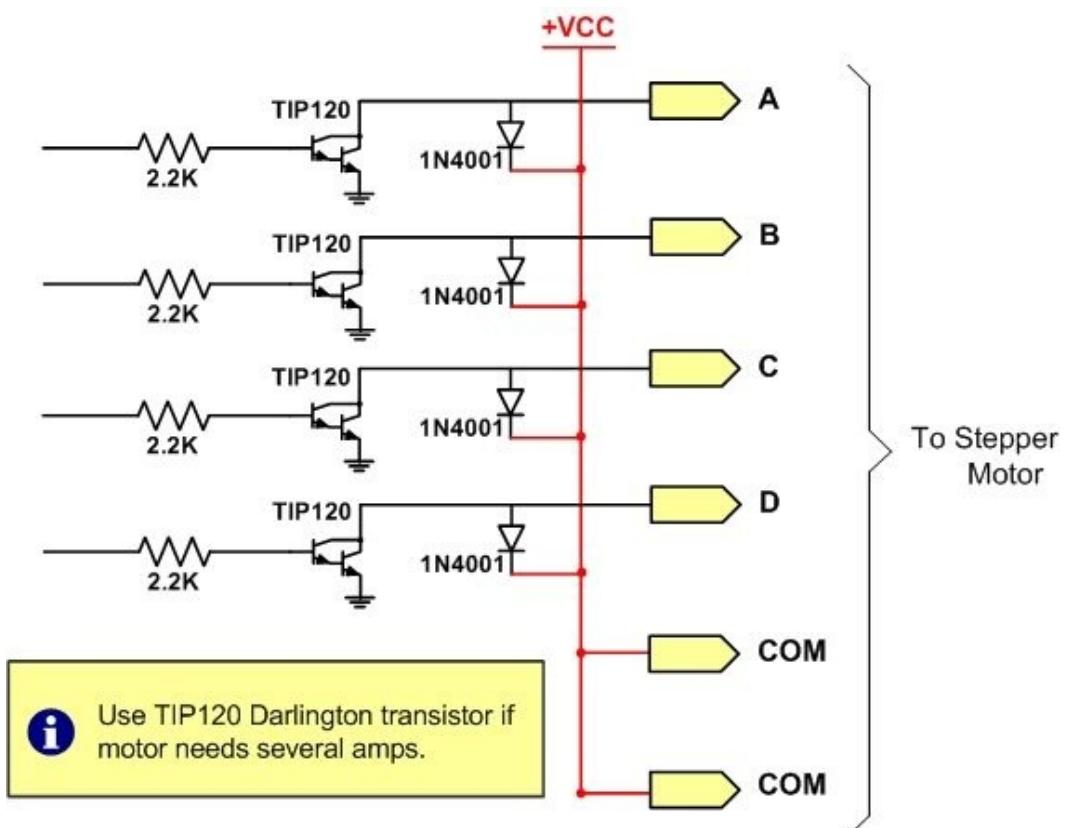
**Table 10-7: Selected Stepper Motor Characteristics ([www.Jameco.com](http://www.Jameco.com))**

Figure 10-10 shows the basic internal connections of all three types 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	$V_{CEO}$ (volts)	$I_C$ (amps)	$hfe$ (common)
TIP110	TIP115	60	2	1000
TIP111	TIP116	80	2	1000
TIP112	TIP117	100	2	1000
TIP120	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.

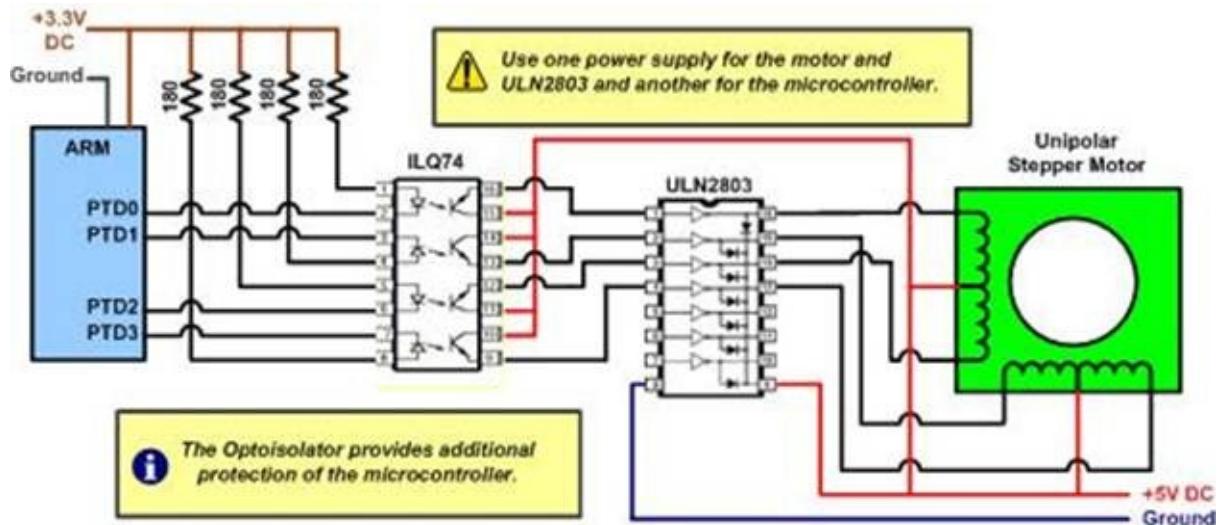


Figure 10-12: Controlling Stepper Motor via Optoisolator

See Program 10-2.

### Program 10-2: Controlling a stepper motor

```
/* p10_2.c: Stepper motor control
/* This program controls a unipolar stepper motor using PTD 3, 2, 1, 0. */

#include <MKL25Z4.H>

int delay = 100;
int direction = 0;

int main (void) {
    void delayMs(int n);

    const char steps[ ] = {0x9, 0x3, 0x6, 0xC};

    int i = 0;
    /* PTD 3, 2, 1, 0 for motor control */
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[0] = 0x100;         /* make PTD0 pin as GPIO */
    PORTD->PCR[1] = 0x100;         /* make PTD1 pin as GPIO */
    PORTD->PCR[2] = 0x100;         /* make PTD2 pin as GPIO */
    PORTD->PCR[3] = 0x100;         /* make PTD3 pin as GPIO */
    PTD->PDDR |= 0x0F;           /* make PTD3-0 as output pin */

    for (;;) {
        if (direction)
            PTD->PDOR = (steps[i++ & 3]);
        else
            PTD->PDOR = (steps[i- & 3]);
        delayMs(delay);
    }
}

/* Delay n milliseconds */
/* The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit(). */

void delayMs(int n) {
    int i;
```

```
int j;  
for(i = 0 ; i < n; i++)  
    for (j = 0; j < 7000; j++) {}  
}
```

## 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. 1100, 0110, 0011, 1001 for clockwise; and 1001, 0011, 0110, 1100 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

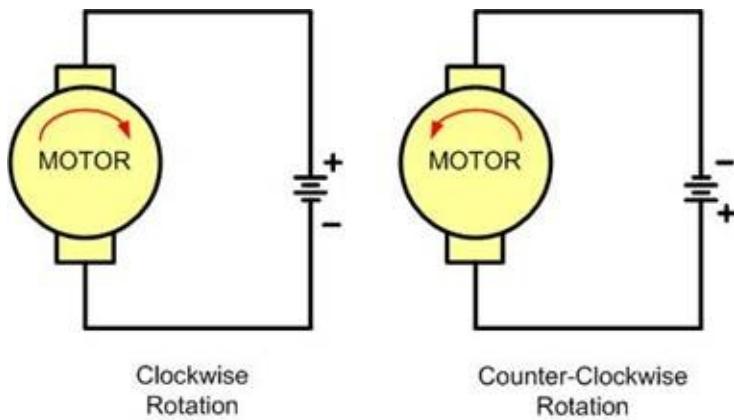
This section begins with an overview of the basic operation of the DC motors. Then we describe how to interface a DC motor to the ARM. Finally, we use C language programs to demonstrate the concept of pulse width modulation (PWM) and show how to control the speed and direction of a DC motor.

### 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)**

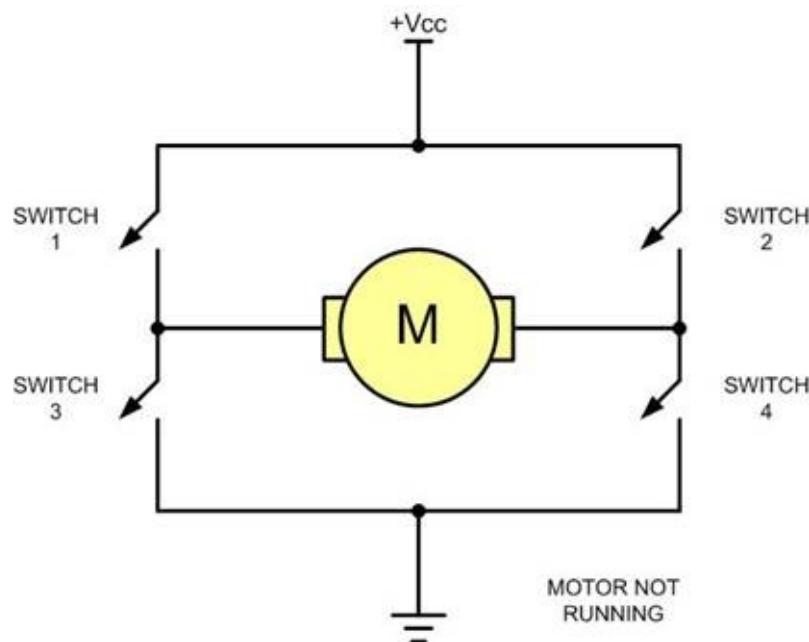
See Table 11-1 for selected DC motors.

Part No.	Nominal Volts	Volt Range	Current	RPM	Torque
<b>154915CP</b>	3 V	1.5–3 V	0.070 A	5,200	4.0 g-cm
<b>154923CP</b>	3 V	1.5–3 V	0.240 A	16,000	8.3 g-cm
<b>177498CP</b>	4.5 V	3–14 V	0.150 A	10,300	33.3 g-cm
<b>181411CP</b>	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.

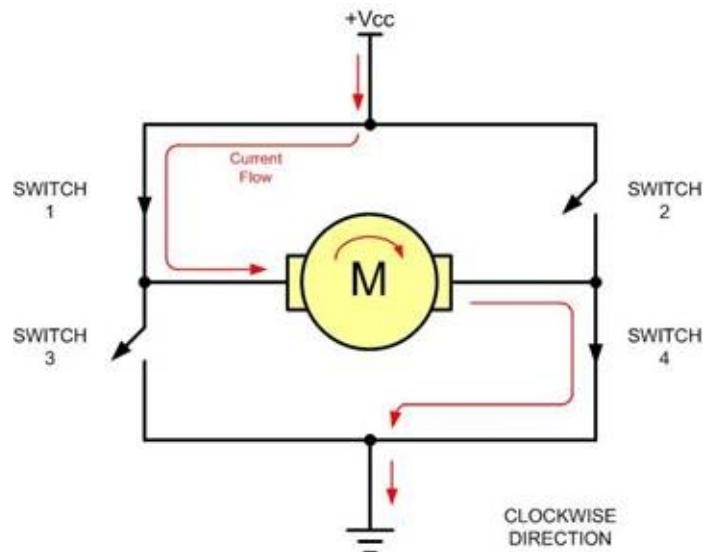


Figure 11-3: H-Bridge Motor Clockwise Configuration

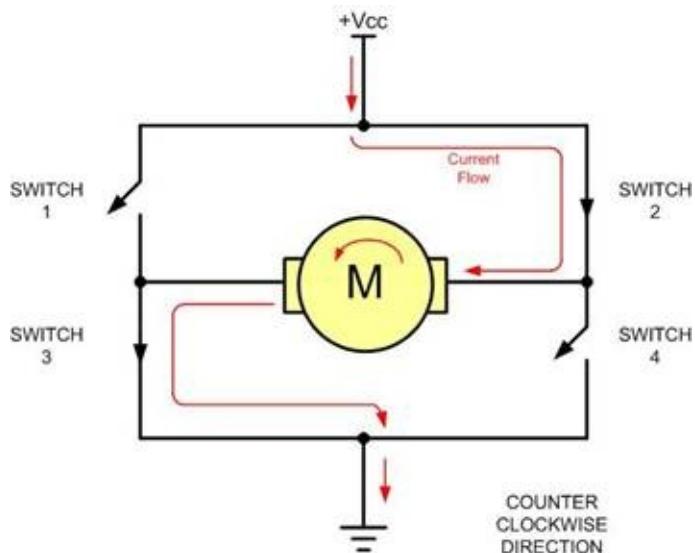
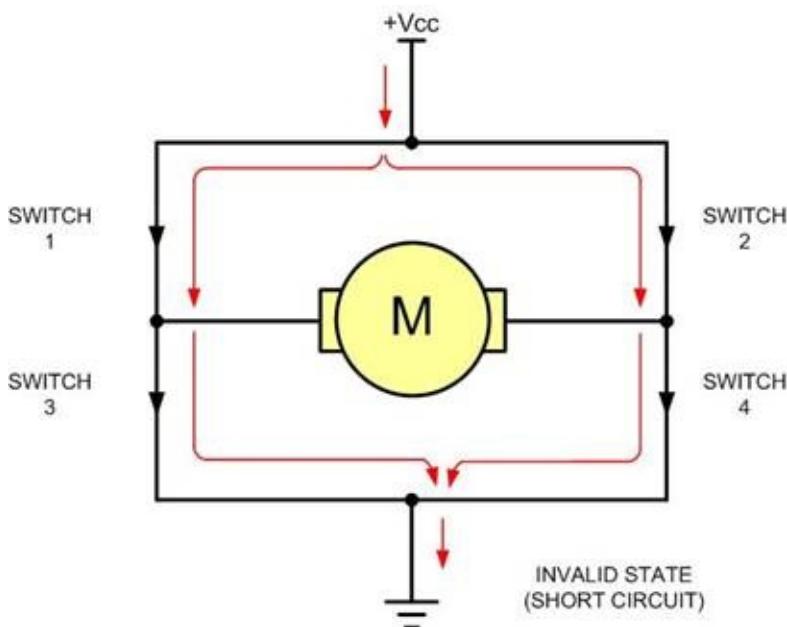


Figure 11-4: H-Bridge Motor Counterclockwise Configuration

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
<b>Off</b>	Open	Open	Open	Open
<b>Clockwise</b>	Closed	Open	Open	Closed
<b>Counterclockwise</b>	Open	Closed	Closed	Open
<b>Invalid</b>	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.

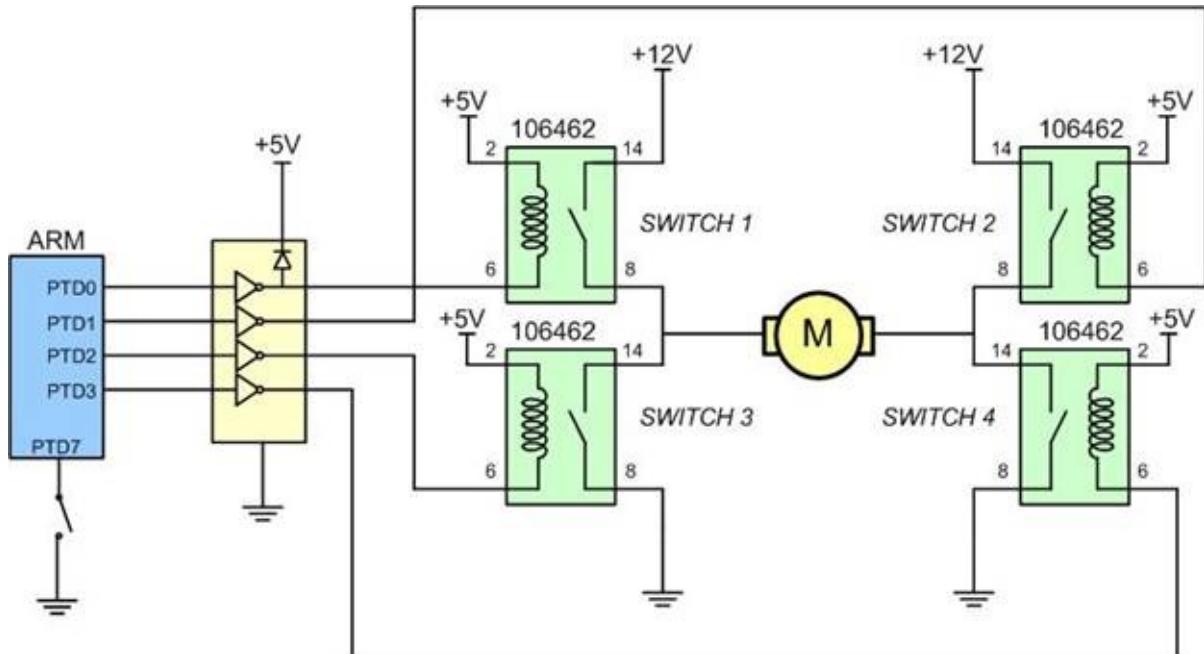
Although we do not show the relay control of an H-Bridge, Example 11-1 shows a simple program to operate a basic H-Bridge.

### Example 11-1

A switch is connected to pin PTD7. Using relays make the H-Bridge in Table 11-2 and write the proper program. We must perform the following:

- (a) If PTD7 = 0, the DC motor moves clockwise.
- (b) If PTD7 = 1, the DC motor moves counterclockwise.

### Solution 1 (Using SPST Relays):



```

int main (void) {
    void delayMs(int n);

    PORTD->PCR[0] = 0x100; /* make PTD0 pin as GPIO */
    PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
    PORTD->PCR[2] = 0x100; /* make PTD2 pin as GPIO */
    PORTD->PCR[3] = 0x100; /* make PTD3 pin as GPIO */
    PORTD->PCR[7] = 0x103; /* make PTD7 pin as GPIO and enable pullup */
    PTD->PDDR |= 0x0F;      /* make PTD0-3 as output pin */
    PTD->PDDR &= ~0x80;     /* make PTD7 as input pin */

    if((PTD->PDIR & 0x80) == 0)
    { /* PTD7 == 0 */
        PTD->PDOR &= ~0x0F; /* open all switches */
        delayMs(100);       /* wait 0.1 second */
        PTD->PDOR |= 0x09;  /* close SW1 & SW4 */

        while((PTD->PDIR & 0x80) == 0); /*PTD7 == 0 */
    }
    else
    { /* PTD7 == 1 */
        PTD->PDOR &= ~0x0F; /* open all switches */
        delayMs(100);       /* wait 0.1 second */
    }
}

```

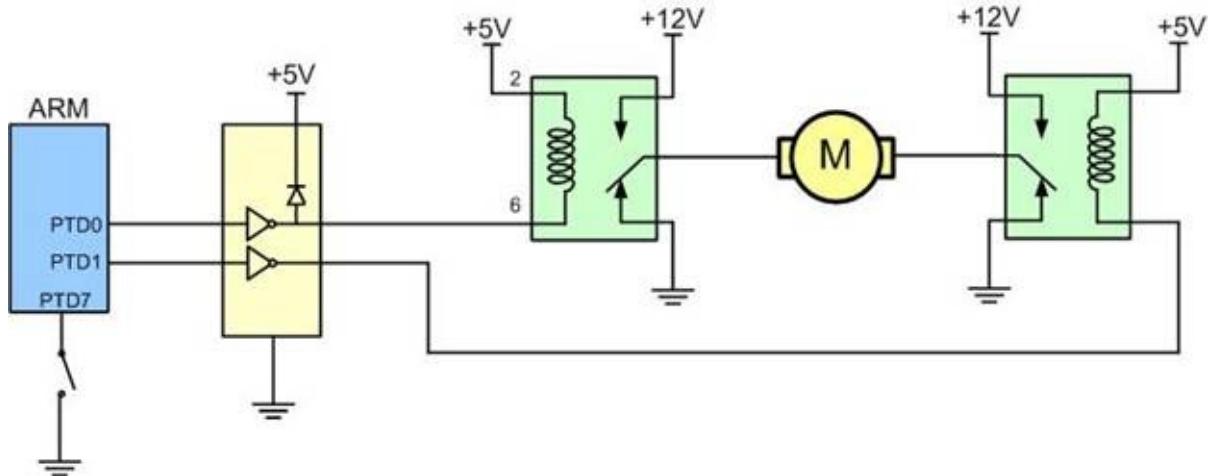
```

PTD->PDOR |= 0x06; /* close SW2 & SW3 */
while((PTD->PDIR & 0x80) != 0); /*PTD7 == 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 (void) {
    PORTD->PCR[0] = 0x100; /* make PTD0 pin as GPIO */
    PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
    PORTD->PCR[7] = 0x103; /* make PTD7 pin as GPIO and enable pullup */
    PTD->PDDR |= 0x03; /* make PTD0-1 as output pin */
    PTD->PDDR &= ~0x80; /* make PTD7 as input pin */
    if((PTD->PDIR & 0x80) == 0)
    { /* PTD7 == 0 */
        PTD->PDOR &= ~0x02; /* Relay 2 = Off */
        PTD->PDOR |= 0x01; /* Relay 1 = On */
    }
    else
    { /* PTD7 == 1 */
        PTD->PDOR &= ~0x01; /* Relay 1 = Off */
        PTD->PDOR |= 0x02; /* Relay 2 = On */
    }
}

```

}

Figure 11-6 shows the connection of the L298N to the microcontroller. Be aware that the L298N will generate heat during operation. For sustained operation of the motor, use a heat sink.

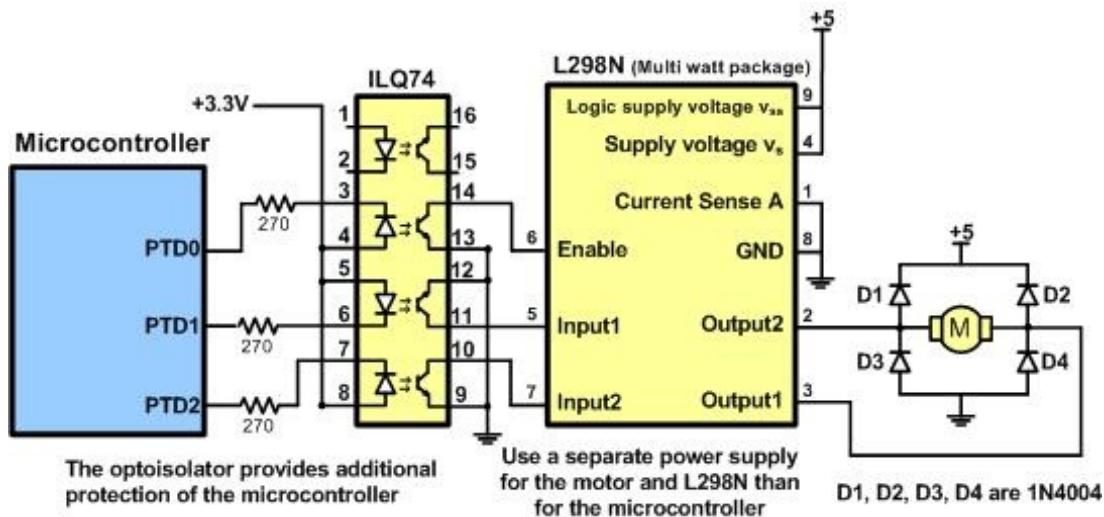


Figure 11-6: Bidirectional Motor Control Using an L298 Chip

## 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.

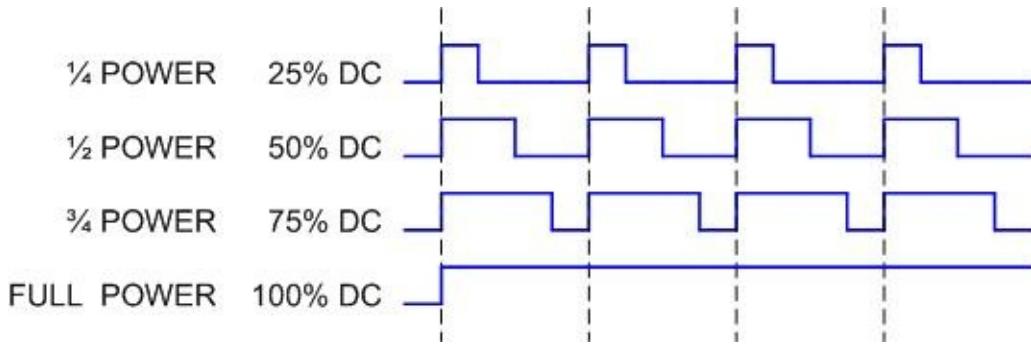


Figure 11-7: Pulse Width Modulation Comparison

## 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.

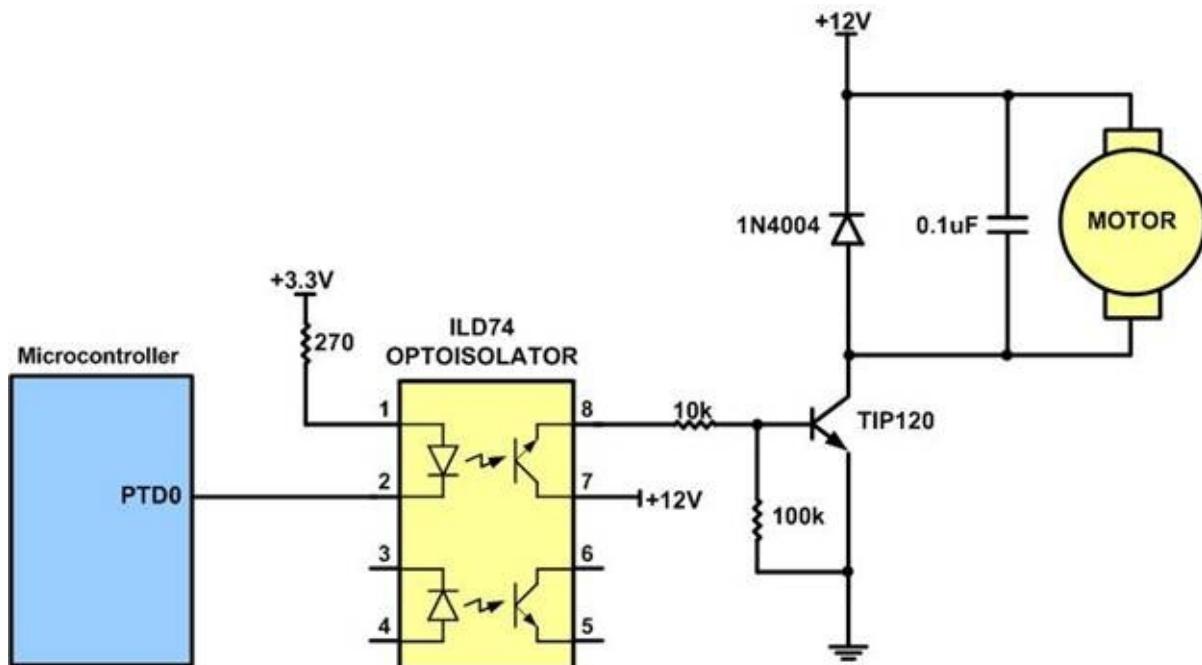
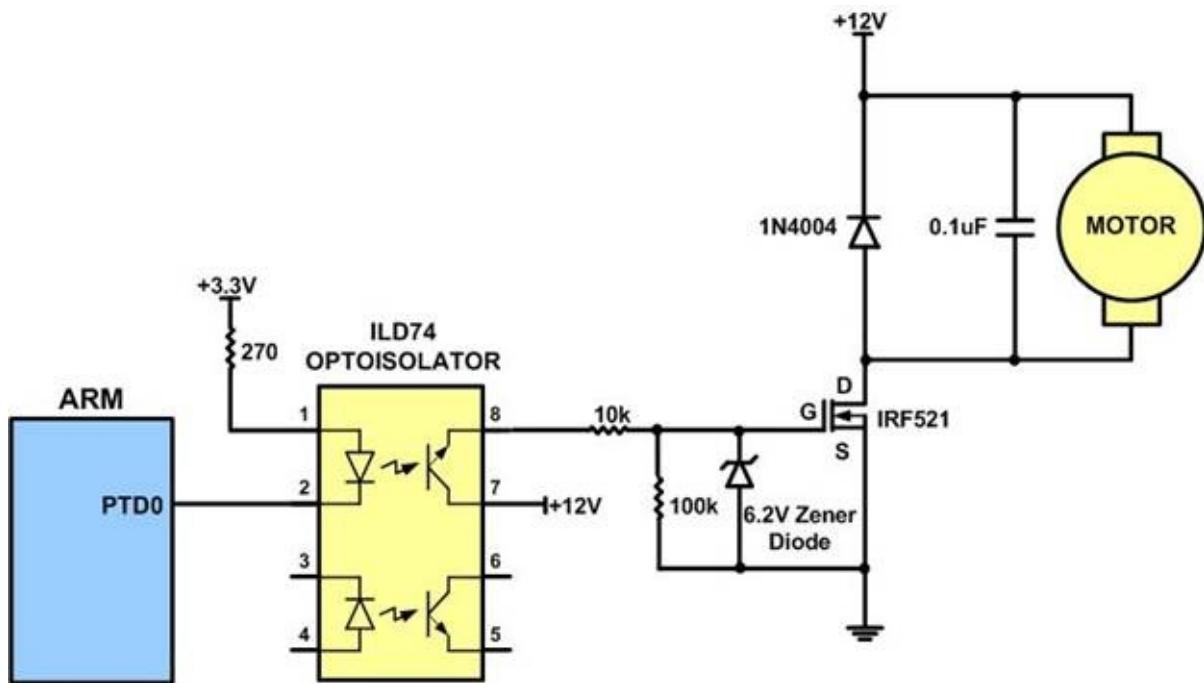


Figure 11-8: DC Motor Connection Using a Darlington Transistor

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 PTD0.



**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

1. True or false. The permanent magnet field DC motor has only two leads for + and – voltages.
2. True or false. As with a stepper motor, one can control the exact angle of a DC motor's move.
3. Why do we put a driver between the microcontroller and the DC motor?
4. How do we change a DC motor's rotation direction?
5. What is stall in a DC motor?
6. The RPM rating given for the DC motor is for \_\_\_\_\_ (no-load, loaded).



## Section 11.2: Programming PWM in Freescale ARM KL25Z

In Freescale ARM KL25Z, the PWM (Pulse Width Modulation) is incorporated into the Timer. As we saw in Chapter 5, the Timer in KL25Z is called TPM (Timer/PWM Module). To program the PWM features of the ARM KL25Z chip, we must understand the Timer topics covered in Chapter 5 since PWM is subset of the Timer. In this section, we examine the PWM features and show how to program them.

### PWM Clock source

The Clock source to the TPM module is enabled using the SIM\_SCGC6 register. See Figure 11-10.

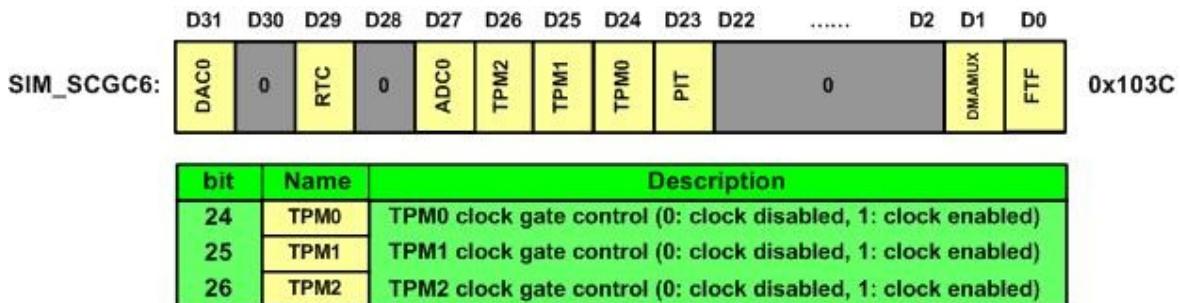


Figure 11-10: SIM\_SCGC6 Register

### CPWMS bit and the TPM counting

As discussed in Chapter 5, the TPMx\_SC register has control on the counting of the timer. See Figure 11-11, Table 11-3, and Figure 11-12.

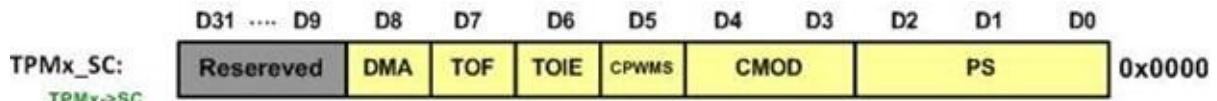


Figure 11-11: Timer Status and Control (TPMx\_SC) Register

Field	Bits	Description																		
PS	0–2	In the prescaler, the clock is divided by $2^{PS}$ . <table border="1"> <thead> <tr> <th>PS value</th> <th>000</th> <th>001</th> <th>010</th> <th>011</th> <th>100</th> <th>101</th> <th>110</th> <th>111</th> </tr> </thead> <tbody> <tr> <th>Division</th> <td>1</td> <td>2</td> <td>4</td> <td>8</td> <td>16</td> <td>32</td> <td>64</td> <td>128</td> </tr> </tbody> </table>	PS value	000	001	010	011	100	101	110	111	Division	1	2	4	8	16	32	64	128
PS value	000	001	010	011	100	101	110	111												
Division	1	2	4	8	16	32	64	128												
		Clock Mode Selection																		
		<table border="1"> <thead> <tr> <th>CMOD value</th> <th>Selected clock</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Timer stopped (No clock selected): In the mode, the TPM_CNT register receives no</td> </tr> </tbody> </table>	CMOD value	Selected clock	00	Timer stopped (No clock selected): In the mode, the TPM_CNT register receives no														
CMOD value	Selected clock																			
00	Timer stopped (No clock selected): In the mode, the TPM_CNT register receives no																			

		clock and it is stopped.
CMOD 3–4	01	Timer mode (clock selected at SIM_SOPT2): This mode can be used to generate delays, periodic interrupts, or PWM.
	10	Counter mode (clocked by LPTPM_EXTCLK pin): This mode is used to count an external event.
	11	Reserved
CPWMS 5		Center-aligned PWM select (0: Up counter mode, 1: up-down counter mode).
TOIE 6		Time Overflow Interrupt Enable (0: Disabled, 1: Enabled).
TOF 7		Timer Overflow Flag
DMA 8		DMA Enable (0: Disabled, 1: Enabled)

Table 11-3: Timer Status and Control (TPMx\_SC) Register

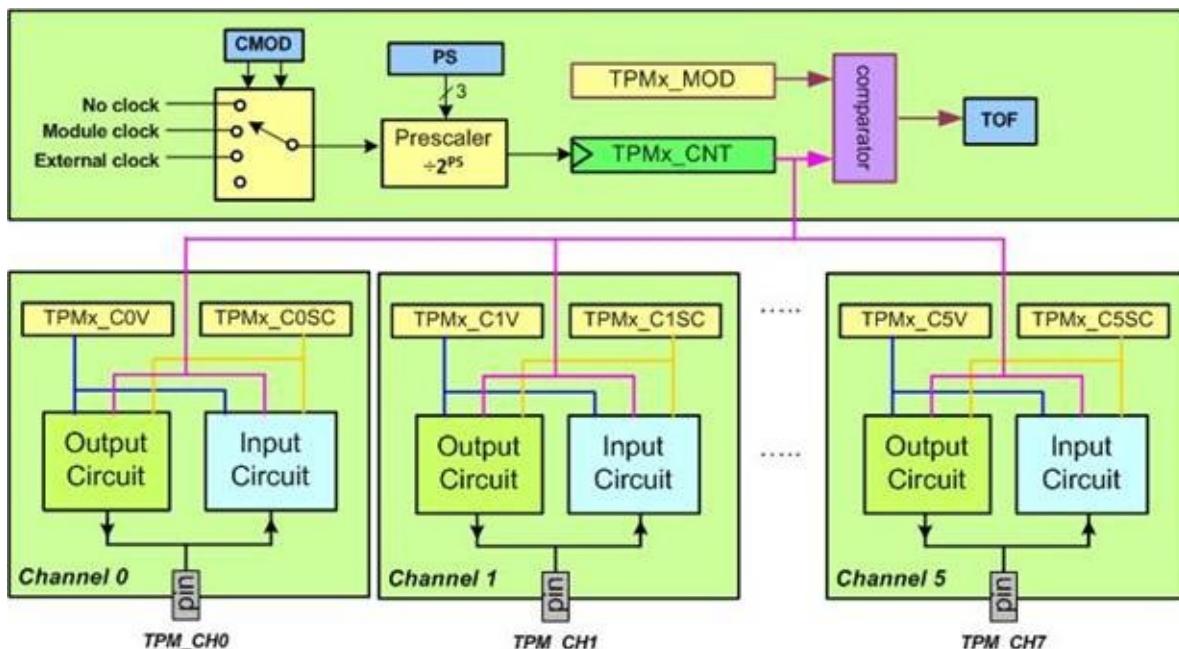


Figure 11-12: CMOD and PS (Prescaler) bits

The CPWMS bit can be set as up-counter (CPWMS=0) or up-down counter (CPWMS=1). The counter has two modes:

- 1) **Count Up:** The TPMx\_CNT counts up from the 0 value until it reaches the value of MOD register. Upon matching the MOD, the CNT is cleared to zero and count-up starts again. This is the default option for

CPWMS=0.

- 2) **Count Up-Down:** counts up from 0 until it reaches the MOD value. After reaching the MOD value, it turns around and counts down to 0. And upon reaching 0, it repeats the process. We must make CPWMS=1 to get this option. See Figure 11-13.

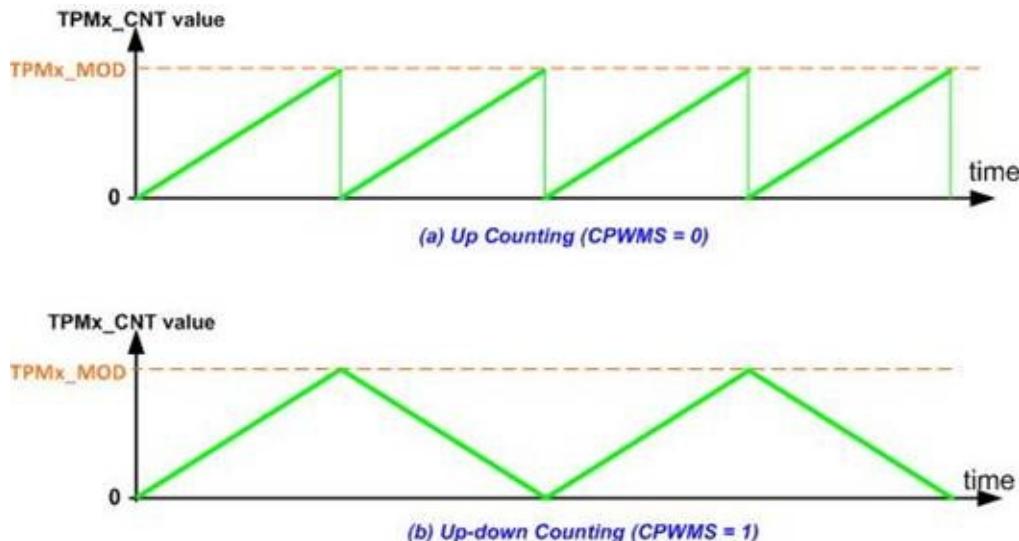


Figure 11-13: Up/Down-Counter and Up-Counter

In Chapter 5, we used TPMx\_CnSC (TPMx Channel n Status and Control) register to program the Output Control or Input Capture features of the KL25Z Timer. As it was shown, the MSnB:MSnA bits along with the ELSnB:ELSnA bits of TPMx\_CnSC register gave us the choices of Input Capture and Output Compare. We use the same bits to choose the PWM features of KL25Z. See Figure 11-14 and Table 11-4. For PWM, we can use the options of Center-aligned (up-down counting) or Edge-aligned (up counting). Each is discussed next.



Figure 11-14: TPMxCnSC (TPMx Channel Status and Control)

Field	Bit	Description
CHF	7	Channel Flag
CHIE	6	Channel interrupt enable
MSB and MSA	5-4	Channel mode select
		D5:D4 (MSB:MSA)      Output mode
		00      Channel disabled
		01      Output compare

	10	PWM
	11	Output compare
ELSB and ELSA	3-2	Edge or Level Select
DMA	0	DMA enable (0: Disabled, 1: Enabled)

Table 11-4: TPMxCnSC Register

## Edge-Aligned PWM

In the Edge-aligned PWM, the leading edge of the pulse starts at the beginning of the period. See Figure 11-15. The pulse period is set by the MOD register value (actually MOD+1) and the pulse width value is set by the CnV register. When ELSnB:ELSnA = 10, it produces high-true pulses. The output is high at the beginning of the pulse when the counter is reloaded and it goes low when the counter value matches CnV register. When ELSnB:ELSnA = x1, it produces low-true pulses. The output is low at the beginning of the pulse when the counter is reloaded and it goes high when the counter value matches CnV register. See Figure 11-15 and Table 11-5.

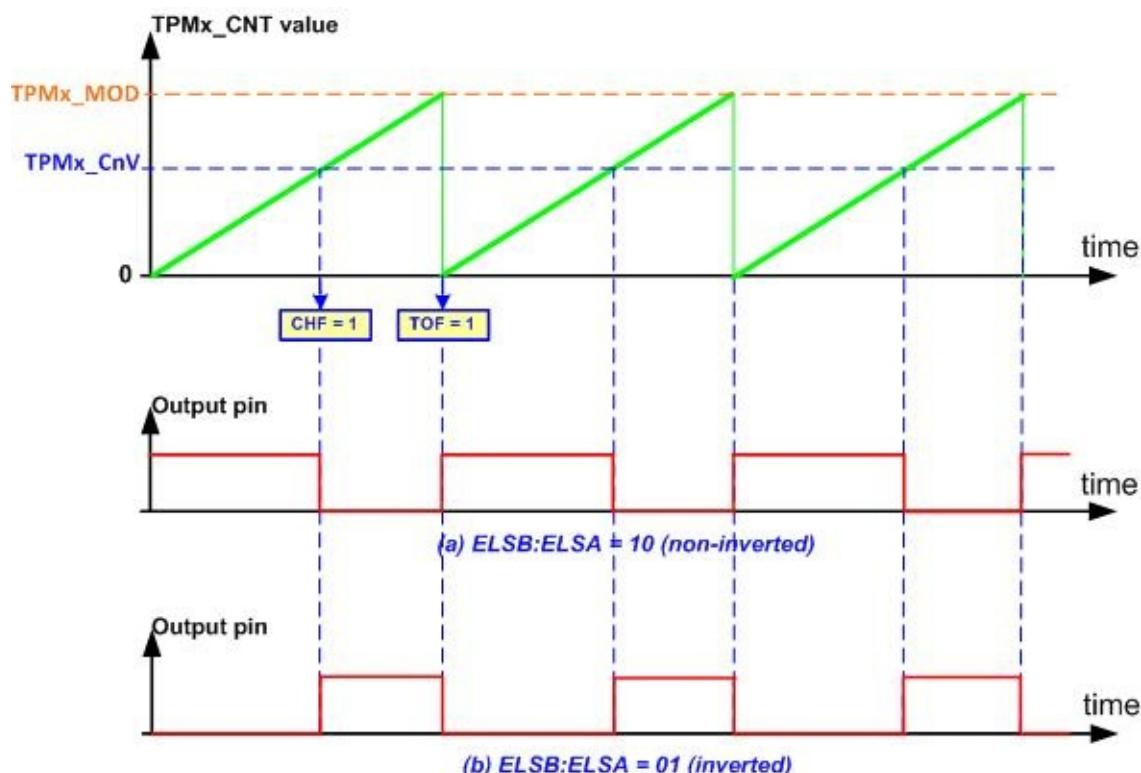


Figure 11-15: Edge-Aligned PWM

CPWMS	MSnB:MSnA	ELSnB:ELSnA	Mode	Configuration
0	10	10	Edge-Aligned PWM	Set output on reload, clear output on

			(non-inverted)	match
0	10	01 or 11	Edge-Aligned PWM (inverted)	Clear output on reload, set output on match

Table 11-5: Edge-Aligned PWM (notice bit CPWMS=0)

### The PWM output duty cycle and frequency

The pulse period is set by the MOD register. Using the CnV register we set the pulse width (duty cycle). Now, if CnV = 0, then Channel output has 0% duty cycle. The same way, if CnV greater than or equal to MOD, the duty cycle is 100% since there is never a match.

Figure 11-16 shows the output waveform when ELSB:ELSA = 10 (non-inverted), MOD = 8, and CnV = 5. The output is set on counter overflow (reload) and it is cleared on compare match. The CNT is reloaded with 0 after MOD + 1 clocks and the output is set to HIGH for CnV clocks. So, the duty cycle can be calculated using the following formula:

$$\text{Duty Cycle} = \frac{\text{CnV}}{\text{MOD} + 1} \times 100$$

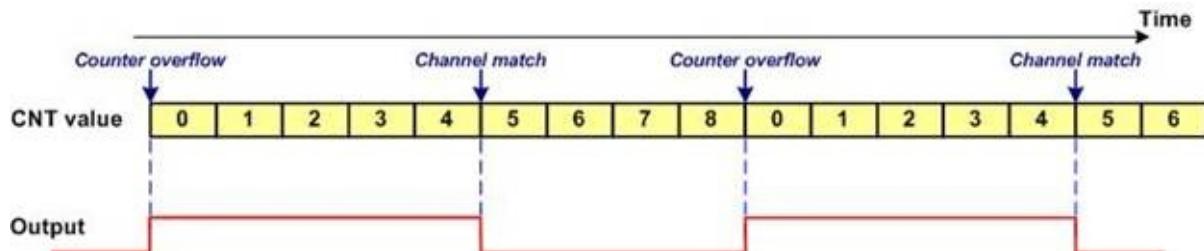


Figure 11-16: The PWM output for MOD = 8, CnV = 5, ELSB:ELSA = 10 (non-inverted)

When ELSB:ELSA = 01, the output is inverted and the duty cycle is:

$$\text{Duty Cycle} = 100 - \left( \frac{\text{CnV}}{\text{MOD} + 1} \times 100 \right)$$

In edge-aligned PWM mode, the timer counts from 0 to MOD and then rolls over. So, the frequency of the output is  $1 / (\text{MOD}+1)$  of the frequency of timer clock. The frequency of the timer clock can be selected using the prescaler. So, the frequency of the output can be calculated as follows:

$$F_{\text{generated wave}} = \frac{F_{\text{timer clock}} / \text{prescaler}}{\text{MOD}+1} = \frac{F_{\text{timer clock}}}{(\text{MOD} + 1) \times 2^{\text{PS}}}$$

See Examples 11-2 through 11-5.

### Example 11-2

Find the period (T), frequency (F) and pulse width (DC, duty cycle) of a PWM if TPMx\_MOD=999 and TMPx\_CnV=250. Assume ELSB:ELSA = 10, no prescaler,

and TPMx Module clock frequencies of (a) 8MHz, (b) 2MHz, and (c) 1MHz.

**Solution:**

(a)  $1/8\text{MHz} = 125\text{ns}$ . Now  $T = (\text{MOD}+1) \times 125\text{ns} = (999+1) \times 125\text{ns} = 125\text{ms}$ .

Frequency= $1/125\text{ms}=8000\text{Hz}$ .

The Duty Cycle is  $[\text{TPMx\_CnV}/(\text{TPMx\_MOD}+1)] \times 100 = (250/1000) \times 100 = 25\%$ .

(b)  $1/2\text{MHz} = 500\text{ns}$ . Now  $T = (\text{MOD}+1) \times 500\text{ns} = (999+1) \times 500\text{ns} = 5\text{ms}$ .

Frequency= $1/5\text{ms}=2000\text{Hz}$ .

The Duty Cycle is  $[\text{TPMx\_CnV}/(\text{TPMx\_MOD}+1)] \times 100 = (250/1000) \times 100 = 25\%$ .

(c)  $1/1\text{MHz} = 1000\text{ns}$ . Now  $T = (\text{MOD}+1) \times 1000\text{ns} = (999+1) \times 1000\text{ns} = 1\text{ms}$ .

Frequency =  $1/1\text{ms}=1000\text{Hz}$ .

The Duty Cycle is  $[\text{TPMx\_CnV}/(\text{TPMx\_MOD}+1)] \times 100 = (250/1000) \times 100 = 25\%$ .

---

---

**Example 11-3**

Assume the TPMx Module clock frequency is 8MHz. Find the value of the MOD register if we want the PWM output Frequency of (a) 5KHz, (b) 10KHz, and (c) 25KHz.

**Solution:**

The clock period for TPM Module is  $1/8\text{MHz}=0.125\mu\text{s}$  (micro second).

(a) The PWM output period is  $1/5\text{KHz} = 200\mu\text{s}$ . Now,  $\text{TPMx\_MOD} = (200\mu\text{s}/0.125\mu\text{s}) - 1 = 1600 - 1 = 1599$ .

(b) The PWM output period is  $1/10\text{KHz} = 100\mu\text{s}$ . Now,  $\text{TPWM\_MOD} = (100\mu\text{s}/0.125\mu\text{s}) - 1 = 800 - 1 = 799$ .

(c) The PWM output period is  $1/25\text{KHz} = 40\mu\text{s}$ .  $\text{TPMx\_MOD} = (40\mu\text{s}/0.125\mu\text{s}) - 1 = 320 - 1 = 319$ .

---

---

### Example 11-4

In a given PWM application, we need the PWM output frequency of 60Hz. Using the TPMx Module frequency of 41.98MHz, find out the value of the TPMx\_MOD register.

#### Solution:

$\text{TPMx\_MOD} = (41.98\text{MHz} / 60\text{Hz}) - 1 = 699,666 - 1 = 699,665$ . This is not acceptable since it is larger than 65,535, the maximum value the TPMx\_MOD register can hold. Now,  $699,666/128 - 1 = 5,465$  is acceptable if we use prescaler of 128. The lowest prescaler value we can use is 16 since  $699,666/16 - 1 = 43,728$ . Notice, the prescaler of 8 is not acceptable since  $699,666 / 8 - 1 = 87,457$ .

---

### Example 11-5

Assume the TPM0 Module clock frequency after prescaler is 16MHz and ELSB:ELSA = 10 (non-inverted). Find the value of the TPMx\_MOD and TPMx\_CnV 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 second).

- (a) The PWM output period is  $1 / 1\text{KHz} = 1\text{msec}$ . Now,  $\text{MOD} = (1\text{ms} / 62.5\text{ns}) - 1 = 16,000 - 1 = 15,999$ .

$$\text{TPM0\_CnV} = (\text{TPM0\_MOD} + 1) \times \text{Duty Cycle} / 100 = 16000 \times 25\% = 4,000$$

- (b) The PWM output period is  $1 / 5\text{KHz} = 0.2\text{msec}$ . Now,  $\text{MOD} = (2\text{ms} / 62.5\text{ns}) - 1 = 3200 - 1 = 3,199$

$$\text{TPM0\_CnV} = 3,200 \times 60 / 100 = 40\% \times 3,200 = 1920$$

- (c) The PWM output period is  $1 / 20\text{KHz} = 0.05\text{msec}$ . Now,  $\text{MOD} = (0.05\text{ms} / 62.5\text{ns}) - 1 = 800 - 1 = 799$

$$\text{TPM0\_CnV} = 800 \times 80 / 100 = 640$$

(d) The PWM output period is  $1 / 2\text{KHz} = 0.5\text{msec}$ . Now, MOD =  $(0.5\text{ms} / 62.5\text{ns}) - 1 = 8000 - 1 = 7,999$

$$\text{TPM0\_CnV} = 8000 \times 50 / 100 = 4000$$

---

## **Configuring GPIO pin for PWM**

In using PWM, we must configure the GPIO pins for TPMx output. In this regard, it is same as all other peripherals. The steps are as follow:

1. Enable the clock to GPIO pin.
2. Assign the TPMx signals to specific pins using PORTx\_PCRn register. See Appendix B.

## **Configuring PWM generator to create pulses**

After the GPIO configuration, we need to take the following steps to configure the PWM:

1. Enable clock to TPMx module in SIM\_SCGC6 register
2. Select counter clock source in SIM\_SOPT2 register
3. Disable timer while the configuration is being done.
4. Set the mode for Edge-Aligned PWM with TPMx\_SC.
5. Load the value into TPMx\_MOD register to set the desired output frequency.
6. Load the value into TPMx\_CnV register to set the desired duty cycle.
7. Enable timer.

See the next few programming examples. Program 11-1 uses TPM0, which is wired to the blue LED on the Freescale KL25Z FRDM board. When the program is running, the blue LED will light up. You do need an oscilloscope on PTD1 pin of the FRDM board to observe the waveform. The register values of Program 11-1 are from Example 11-4.

### **Program 11-1: Using TPM0 to create 60Hz with 33% duty cycle on PTD1 pin (blue LED)**

```
/* p11_1.c Generate 60Hz 33% PWM output

 * TPM0 uses MCGFLLCLK which is 41.94 MHz.
 * The prescaler is set to divide by 16.
 * The modulo register is set to 43702 and the CnV
 * register is set to 14568. See Example 11-4 for
 * the calculations of these values.
```

```

/*
#include <MKL25Z4.H>

int main (void) {
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[1] = 0x0400;       /* PTD1 used by TPM0 */

    SIM->SCGC6 |= 0x01000000;     /* enable clock to TPM0 */
    SIM->SOPT2 |= 0x01000000;     /* use MCGFLLCLK as timer counter clock */
    TPM0->SC = 0;                /* disable timer */
    TPM0->CONTROLS[1].CnSC = 0x20 | 0x08; /* edge-aligned, pulse high */
    TPM0->MOD = 43702;           /* Set up modulo register for 60 kHz */
    TPM0->CONTROLS[1].CnV = 14568; /* Set up channel value for 33% dutycycle */
    TPM0->SC = 0x0C;             /* enable TPM0 with prescaler /16 */
    while (1) {
    }
}

```

Program 11-2 is based on Program 11-1 but in the infinite loop, the value of CnV is incremented by 100 every 20 ms. The increasing CnV value lengthens the duty cycle and increase the LED intensity.

### Program 11-2: Use PWM to control LED intensity

```

/* p11_2.c Generate 60Hz with varying duty cycle PWM output

 * This program is setup identical to p11_1. But in the
 * infinite loop, the CnV register value is incremented
 * by 437 (1%) every 20ms. Because the LED is low active,
 * the longer the duty cycle results in lower light intensity.
 */

#include <MKL25Z4.H>
void delayMs(int n);

int main (void) {
    int pulseWidth = 0;
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[1] = 0x0400;       /* PTD1 used by TPM0 */

    SIM->SCGC6 |= 0x01000000;     /* enable clock to TPM0 */

```

```

SIM->SOPT2 |= 0x01000000; /* use MCGFLLCLK as timer counter clock */
TPM0->SC = 0; /* disable timer */
TPM0->CONTROLS[1].CnSC = 0x20 | 0x08; /* edge-aligned, pulse high */
TPM0->MOD = 43702; /* Set up modulo register for 60 kHz */
TPM0->CONTROLS[1].CnV = 14568; /* Set up channel value for 33% dutycycle */
TPM0->SC = 0x0C; /* enable TPM0 with prescaler /16 */
while (1) {
    pulseWidth += 437;
    if (pulseWidth > 43702)
        pulseWidth = 0;
    TPM0->CONTROLS[1].CnV = pulseWidth;
    delayMs(20);
}
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
}

```

Program 11-3 uses the slow 32kHz clock for timer counter clock. This is slow enough that the duty cycle changes can be observed with naked eyes.

**Program 11-3: Based on Program 11-2 but slow down the PWM frequency so that the duty cycle change can be observed with naked eyes**

```

/* p11_3.c Generate slow varying duty cycle PWM output

* This program is setup similar to p11_2. The slow
* 32kHz clock is used for timer counter clock so that
* the pulse width change is visible by the blue LED.
*/

#include <MKL25Z4.H>
void delayMs(int n);

int main (void) {

```

```

int pulseWidth = 0;
SIM->SCGC5 |= 0x1000;           /* enable clock to Port D */
PORTD->PCR[1] = 0x0400;        /* PTD1 used by TPM0 */

SIM->SCGC6 |= 0x01000000;     /* enable clock to TPM0 */
SIM->SOPT2 |= 0x03000000;     /* use 32KHz MCGIRCLK as timer counter clock */
TPM0->SC = 0;                 /* disable timer */
TPM0->CONTROLS[1].CnSC = 0x20 | 0x08; /* edge-aligned, pulse high */
TPM0->MOD = 20000;            /* Set up modulo register for 30Hz */
TPM0->SC = 0x08;              /* enable TPM0 */
while (1) {
    pulseWidth += 1000;
    if (pulseWidth > 20000)
        pulseWidth = 0;
    TPM0->CONTROLS[1].CnV = pulseWidth;
    delayMs(1000);
}
}

/* Delay n milliseconds
 * The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
 */
void delayMs(int n) {
    int i;
    int j;
    for(i = 0 ; i < n; i++)
        for (j = 0; j < 7000; j++) {}
}

```

## Center-Aligned PWM

If we set the CPWMS bit in TPM<sub>x</sub>\_SC register to High, then the output is Center-Aligned PWM. The counter will count up from 0 to the value in MOD register then turn around and count down to 0. That means, the period of the pulse is  $2 \times \text{MOD}$ . The same way, the pulse width =  $2 \times \text{CnV}$ . At the same time whenever the CnV=MOD, the output pin is forced High or Low depending on the ELSnB:ELSnA bits and whether the counter is counting up or down. See Figure 11-17 and Table 11-6.

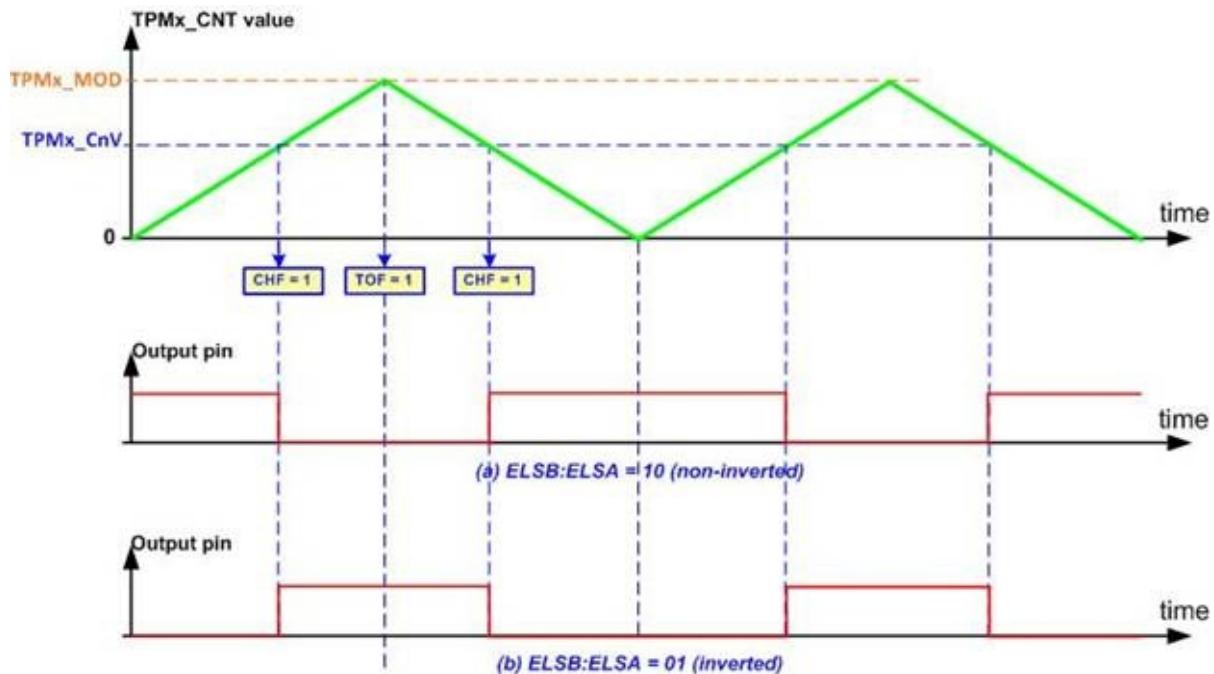


Figure 11-17: Center-Aligned PWM

CPWMS	MSnB:MSnA	ELSnB:ELSnA	Mode	Configuration
1	10	10	Center-Aligned PWM	Clear output on match-up, set output on match-down
1	10	X1	Center-Aligned PWM	Set output on match-up, clear output on match-down

Table 11-6: Center-Aligned PWM (notice CPWMMS=1)

### The PWM output duty cycle and frequency

Figure 11-18 shows the output when MOD = 7 and CnV = 4. The output is set on compare match when counting down, and is cleared on compare match when counting up. The output is HIGH for  $CnV \times 2$  clocks and each cycle takes  $MOD \times 2$  clocks. As a result, the duty cycle is:

$$\text{Duty Cycle} = \frac{CnV \times 2}{MOD \times 2} \times 100 = \frac{CnV}{MOD} \times 100$$

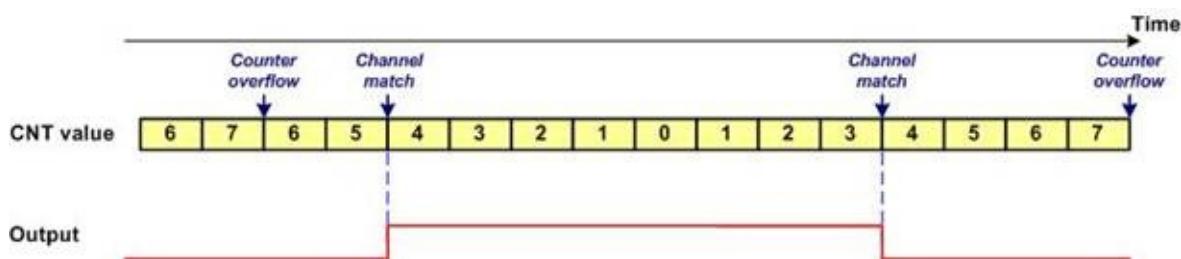


Figure 11-18: The PWM output for MOD = 7, CnV = 4, ELSB:ELSA = 10 (non-inverted)

When ELSB:ELSA = 01, the output is inverted and the duty cycle is:

$$\text{Duty Cycle} = 100 - \left( \frac{CnV}{MOD} \times 100 \right)$$

The frequency of the generated wave is:

$$F_{\text{generated wave}} = \frac{F_{\text{timer clock}} / \text{prescaler}}{\text{MOD} \times 2} = \frac{F_{\text{timer clock}}}{\text{MOD} \times 2^{PS+1}}$$

### Example 11-6

Find the period (T), frequency (F) and pulse width (DC, duty cycle) of a PWM if TPMx\_MOD=400 and TMPx\_CnV=250. Assume ELSB:ELSA = 10 (non-inverted), no prescaler, and TPWx Module clock frequencies of (a) 8MHz, (b) 2MHz, and (c) 1MHz.

#### Solution:

(a) 1/8MHz = 125ns. Now T = MOD × 2 × 125ns = 400 × 2 × 125ns = 100μs.

Frequency = 1 / 100μs = 10 kHz.

The Duty Cycle is (TPMx\_CnV / TPMx\_MOD) × 100 = (250 / 400) × 100 = 62.5%.

(b) 1/2MHz= 500ns. Now T=400 × 2 × 500ns = 400μs.

Frequency = 1 / 400μs = 2500Hz.

The Duty Cycle is (TPMx\_CnV / TPMx\_MOD) × 100 = (250/400) × 100 = 62.5%.

(c) 1/1MHz = 1μs. Now T = 400 × 2 × 1μs = 800μs.

Frequency = 1 / 800μs = 1250Hz.

The Duty Cycle is (TPMx\_CnV / TPMx\_MOD) × 100 = (250/400) × 100 = 62.5%.

---

Program 11-4 generates a waveform with duty cycle of 40% using center aligned PWM mode.

#### Program 11-4: Generate 30Hz 40% center-aligned PWM

```
/* p11_1.c Generate 30Hz 40% center-aligned PWM

 * TPM0 uses MCGFLLCLK which is 41.94 MHz.
 * The prescaler is set to divide by 16.
 * The modulo register is set to 43703 and the CnV
```

```

 * register is set to 17481. TPM0 channel 1 is
 * configured to be center-aligned pulse high.
 */

#include <MKL25Z4.H>

int main (void) {
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[1] = 0x0400;        /* PTD1 used by TPM0 */

    SIM->SCGC6 |= 0x01000000;     /* enable clock to TPM0 */
    SIM->SOPT2 |= 0x01000000;     /* use MCGFLLCLK as timer counter clock */
    TPM0->SC = 0;                /* disable timer */
    TPM0->CONTROLS[1].CnSC = 0x20 | 0x08; /* center-aligned, pulse high */
    TPM0->MOD = 43703;           /* Set up modulo register for 1 kHz */
    TPM0->CONTROLS[1].CnV = 17481; /* Set up channel value for 40% duty cycle
*/
    TPM0->SC = 0x0C | 0x20; /* enable TPM0 with prescaler /16, center-aligned
*/
    while (1) {
}
}

```

## Edge-aligned vs. center-aligned mode

See Figure 11-19. In both figures the bold vertical blue lines are repeated periodically. In the edge-aligned mode, the left edge of the pulse is always on the bold blue line while in center-aligned mode, the center of the pulse is always fixed on the bold line. In other words, in edge-aligned mode, the phase of the wave is different for different duty cycles, while it remains unchanged in the center-aligned mode. For driving motors, it is preferable to use center-aligned rather than edge-aligned.

In edge-aligned mode, the frequency of the generated wave is twice that of the center-aligned mode. Thus, edge-aligned mode is preferable when we need to generate waves with higher frequencies.

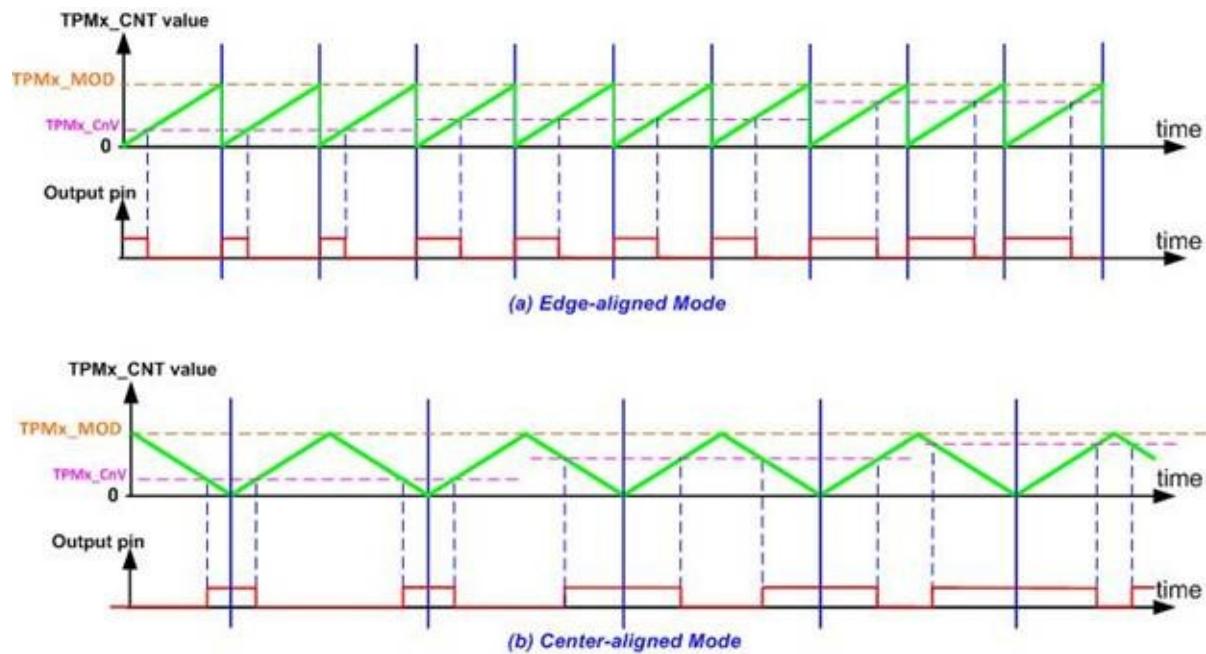


Figure 11-19: Edge-aligned vs. Center-aligned Mode

## Dead-band generation (Case Study)

One application of center-aligned PWM is to generate outputs with deadband. Review Example 11-1, when we switched the direction of the H-bridge circuit, we opened all switches and delayed for a period of time. That was deadband, a period of time when all switches are open to avoid the possibility of overlapping time when both switches on the same leg of the H-bridge are one which may cause a short circuit. The same problem exists with transistor circuits because transistors are faster to turn on and slower to turn off. If we turn one on and the other off, there will be a short time that both transistors are on.

To generate deadband, we use two center-aligned channels on the PWM module. One of the channels has positive pulse and the other negative pulse. Assuming the circuit is active high, now we have one channel centered at the time when the timer counter reaches the value in MOD register and the other channel centered at the time when the timer counter reaches 0 so they will be 180 degree out of phase. For each channel we program them to have less than 50% duty cycle therefore deadbands are created between the two channels. See Figure 11-20.

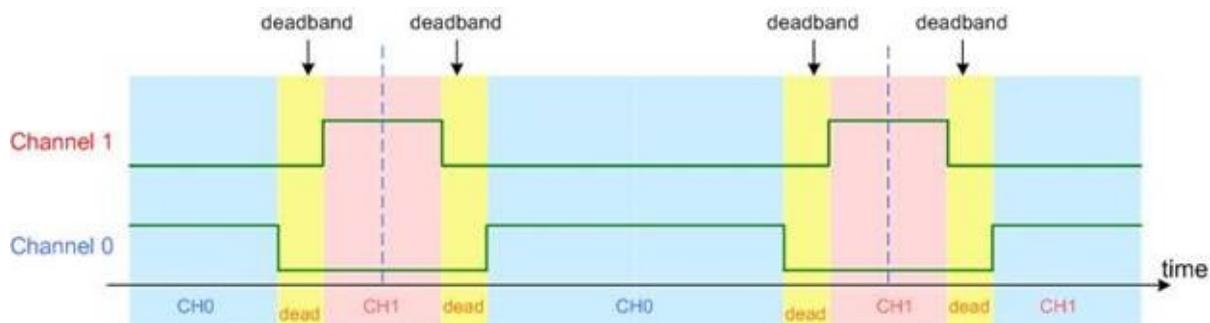


Figure 11-20: Deadband

Program 11-5 generates two 40% duty cycle outputs with 10% deadbands between them using center aligned PWM mode.

### Program 11-5: Deadband generation

```
/* p11_5.c Deadband generation with center-aligned PWM

 * TPM0 uses MCGFLLCLK which is 41.94 MHz. The prescaler
 * is set to divide by 16. The modulo register is set to 43703.
 * The timer is configured for center-aligned PWM.
 * channel 0 is configured for 60% duty cycle pulse low.
 * channel 1 is configured for 40% duty cycle pulse high.
 * This creates a 10% deadband between channel 0 high and
 * channel 1 high.

 */

#include <MKL25Z4.H>

int main (void) {
    SIM->SCGC5 |= 0x1000;          /* enable clock to Port D */
    PORTD->PCR[0] = 0x0400;        /* PTD0 used by TPM0 */
    PORTD->PCR[1] = 0x0400;        /* PTD1 used by TPM0 */

    SIM->SCGC6 |= 0x01000000;     /* enable clock to TPM0 */
    SIM->SOPT2 |= 0x01000000;     /* use MCGFLLCLK as timer counter clock */
    TPM0->SC = 0;                /* disable timer */
    TPM0->CONTROLS[0].CnSC = 0x20 | 0x04; /* center-aligned, pulse low */
    TPM0->CONTROLS[1].CnSC = 0x20 | 0x08; /* center-aligned, pulse high */
    TPM0->MOD = 43703;           /* Set up modulo register for 30Hz */
    TPM0->CONTROLS[0].CnV = 26221; /* Set up channel value for 60% duty cycle
*/
    TPM0->CONTROLS[1].CnV = 17481; /* Set up channel value for 40% duty cycle
*/
    TPM0->SC = 0x0C | 0x20; /* enable TPM0 with prescaler /16, center-aligned */
    while (1) {
    }
}
```

## Review Questions

1. To enable the clock to TPM0 modules, we use register \_\_\_\_\_.
2. If the clock to TPMx is 16MHz, what is lowest and highest clock frequency that PWM Module can use after going to prescale?
3. We use \_\_\_\_\_ register to set the PWM output Period/Frequency.
4. We use \_\_\_\_\_ register to set the PWM output pulse width.
5. True or false. In Freescale ARM KL25Z, the PWM module uses the timer registers to set the frequency and duty cycle.

# Answers to Review Questions

## Section 11.1

1. True
2. False
3. Because microcontroller/digital outputs lack sufficient current to drive the DC motor.
4. By reversing the polarity of voltages connected to the motor leads
5. The DC motor is stalled if the load is beyond what it can handle.
6. No-load

## Section 11.2

1. SIM\_SCGC6.
2.  $16\text{MHz}/128 = 125\text{ KHz}$  and 16 MHz.
3. TPMx\_MOD
4. TPMx\_CnV
5. True.





## **Chapter 12: Programming Graphic LCD**

Chapter 3 used the character LCD. In this chapter, we examine the graphic LCDs and show some programming examples, although an entire book can be dedicated to graphic LCD and its programming. Section 12.1 covers some basic concepts of graphic LCDs. In Section 12.2, we give some programming examples of graphic LCD.



## Section 12.1: Graphic LCDs

The screen of graphic LCDs is made of pixels. The pictures and the texts are created using pixels and the programmers have control over each and every individual pixel. See Figures 12-1 and 12-2.



Figure 12-1: A picture on a Mono-color LCD

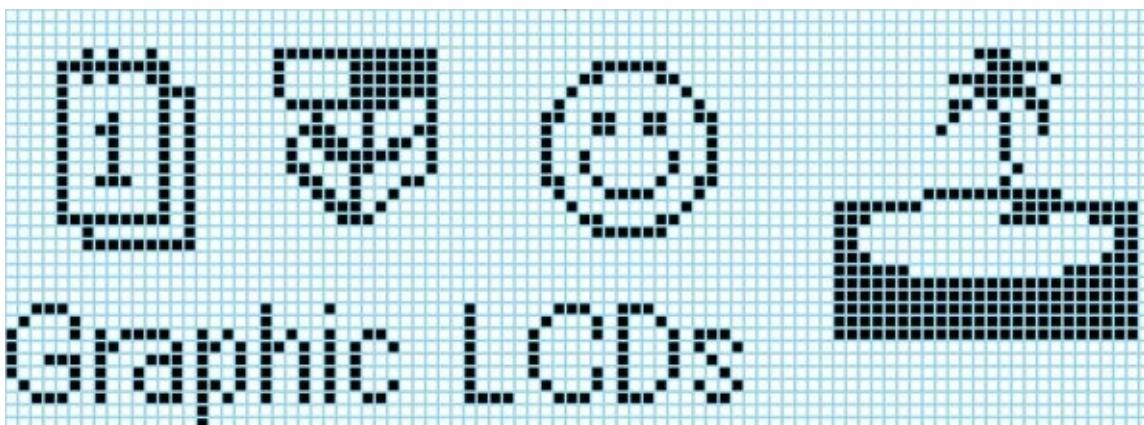


Figure 12-2: A Zoomed Picture on a Mono-color LCD

The graphic LCDs can be mono-colored (monochrome) or colored. In mono-colored LCDs each pixel can be on or off or different shades of gray; 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 primary color 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:

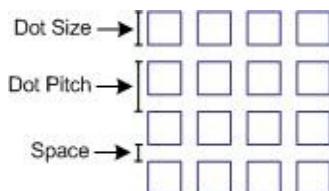
$$\text{Resolution} = \text{Pixels per line} \times \text{number of lines}$$

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 the 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 sample mono-colored LCD are 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 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:

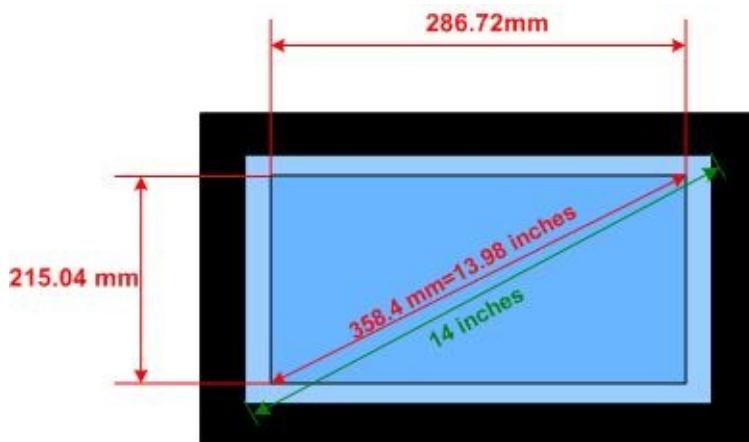
The calculation is as follows:

$$(\text{image diagonal size})^2 = (\text{number of horizontal pixels} \times \text{dot pitch})^2 + (\text{number of vertical pixels} \times \text{dot pitch})^2$$

$$(\text{diagonal size})^2 = (1024 \times 0.28 \text{ mm})^2 + (768 \times 0.28 \text{ mm})^2 = 358.4 \text{ mm}$$

$$\text{diagonal size (inches)} = 358.4 \text{ mm} \times 0.039 \text{ inch per mm} = 13.98 \text{ inches}$$

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 graphic 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.

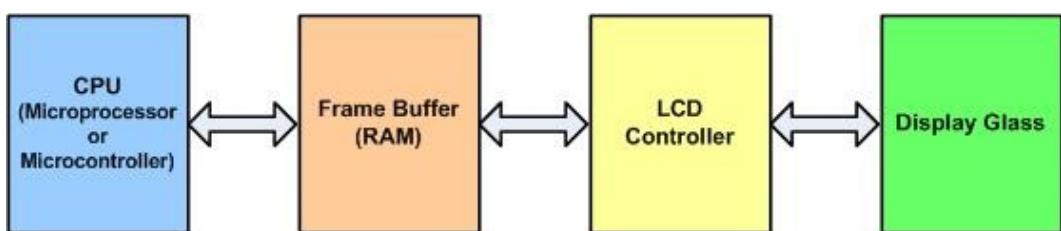


Figure 12-5: The Relationship between CPU and LCD

Graphic 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 supported colors 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 monochrome 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 primary colors of Red, Green, and Blue. The intensity (proportion) of the colors mixed can also affect 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 and 255 (0 to 0xFF) representing high intensity (255) and zero intensity. See Table 12-2. Using three primary colors and intensity, we can make many colors we want. See Figure 12-6.

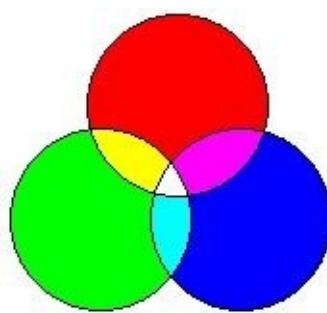


Figure 12-6: Making New Light Colors by Mixing the 3 Primary Light Colors

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 graphic 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 goes 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). You may find the definitions of these abbreviations on the Internet.

We use the following formula to calculate the minimum frame buffer memory requirement for a graphic 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.

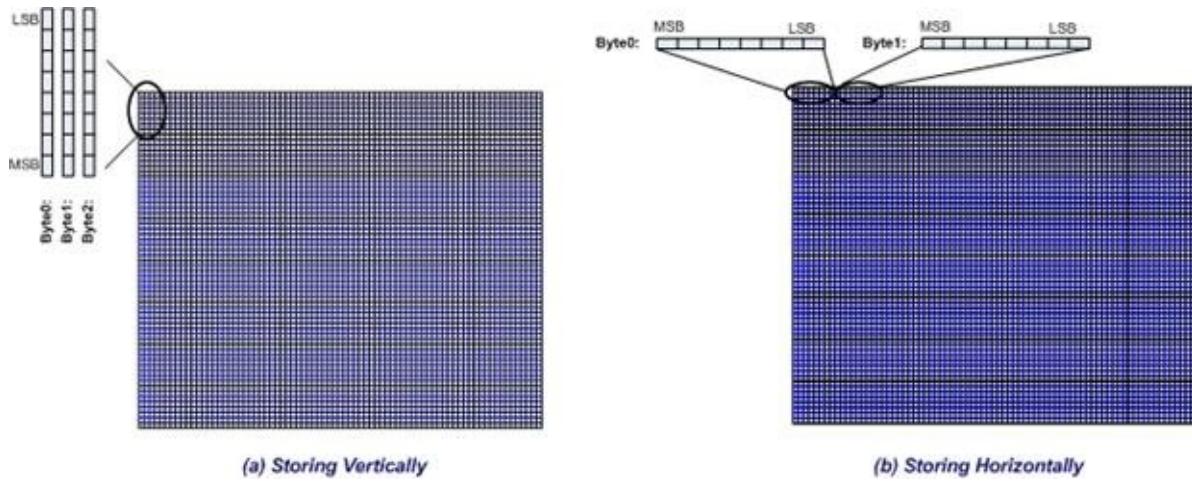


Figure 12-7: Storing Data in the LCD Memory of Mono-colored LCDs

## Review Questions

- As the number of pixels goes up, the size of display memory \_\_\_\_\_ (increases, decreases).
- If a total of 24 bits is set aside for color, how many colors are available?
- Calculate the total video memory needed for  $1024 \times 768$  resolution with 16 colors displayed at the same time.
- With BPP of 16, we get \_\_\_\_\_ colors.



## Section 12.2: Displaying Texts on Graphic LCDs

As shown in Figure 12-8 each character can be made by putting pixels next to each other.

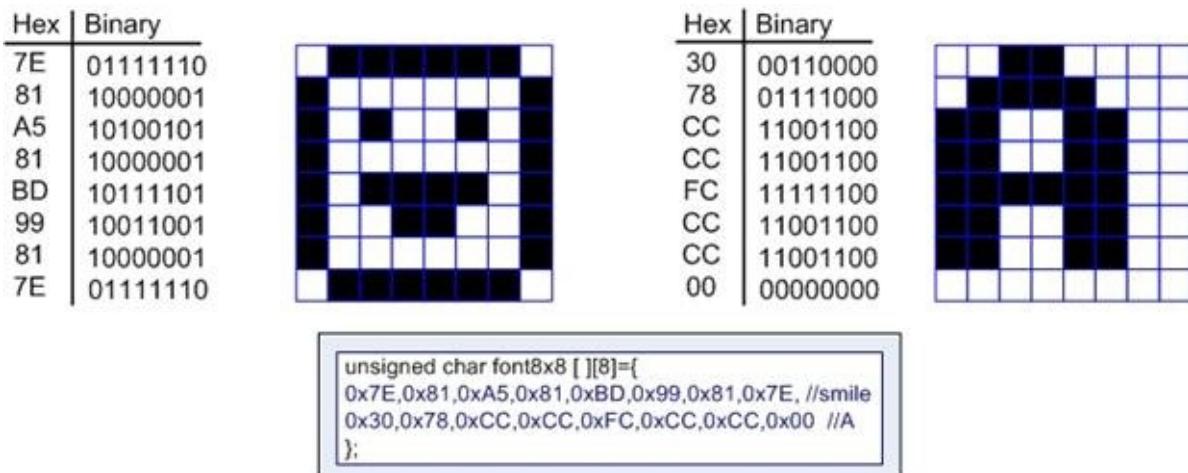


Figure 12-8: Pixel Patterns of Characters Happy Face and Letter A

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.

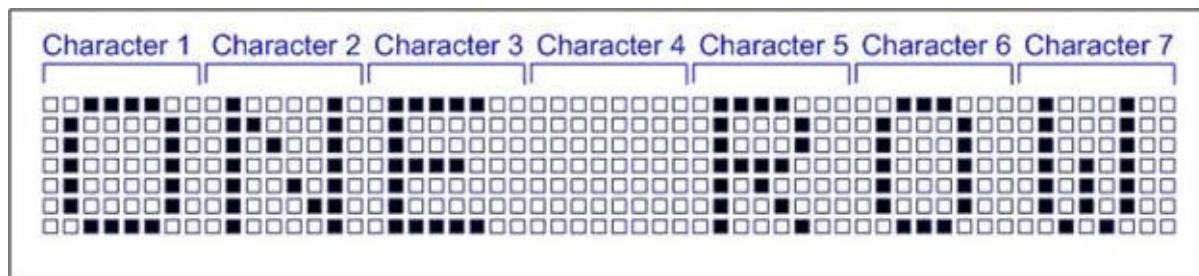
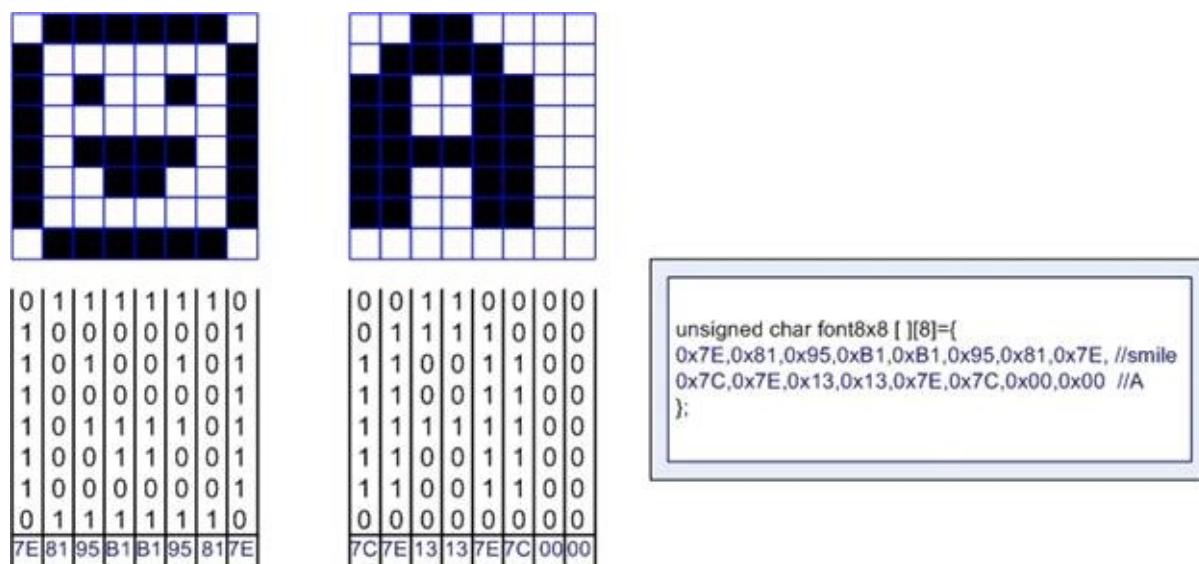


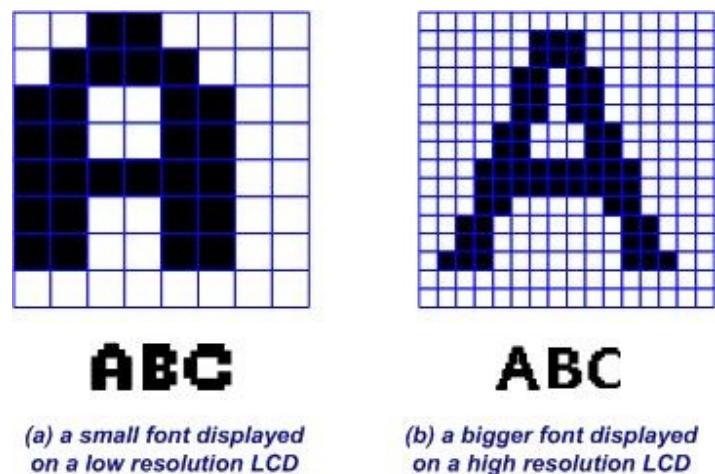
Figure 12-9: A Sample Text

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.



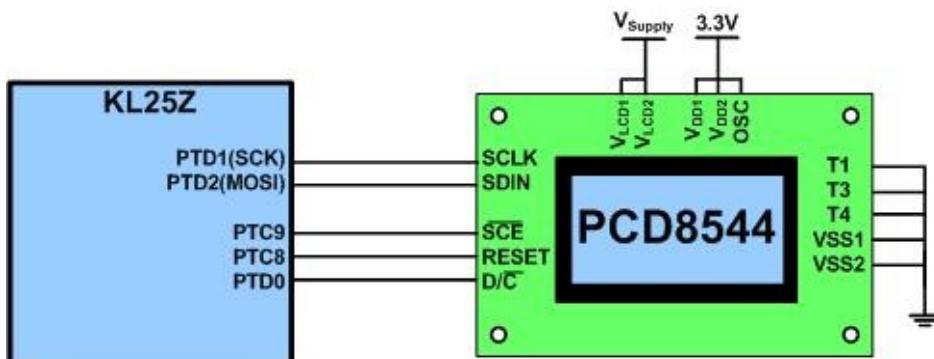
**Figure 12-10: Pixel Patterns of Happy Face and Character A and its Font for Vertical LCD**

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 is made using an array. The GLCD\_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 FRDM-KL25Z**

#### Program 12-1: Displaying a text on the PCD8544 GLCD

```
/*P12_1.c: Programming PCD8544 GLCD via SPI with FRDM-KL25Z

 * PTD1 pin as SPI SCK
 * PTD2 pin as SPI MOSI
 * PTC8 reset pin
 * PTD0 register select pin
 * PTC9 chip select
 */

#include "MKL25Z4.h"
```

```

#define RESET 0x0100      /* PTC8 reset pin */
#define DC     0x0001      /* PTDO register select pin */
#define CE     0x0200      /* PTC9 chip select */

/* define the pixel size of display */
#define GLCD_WIDTH  84
#define GLCD_HEIGHT 48

void GLCD_setCursor(unsigned char x, unsigned char y);
void GLCD_clear(void);
void GLCD_init(void);
void GLCD_data_write(unsigned char data);
void GLCD_command_write(unsigned char data);
void SPI0_init(void);
void SPI0_write(unsigned char data);
void GLCD_putchar(int c);

/* sample font table */
const 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 main(void) {
    GLCD_init();      /* initialize the GLCD controller */
    GLCD_clear();    /* clear display and home the cursor */

    GLCD_putchar(0); /* display letter A */
    GLCD_putchar(1); /* display letter B */
    GLCD_putchar(2); /* display letter C */
    while(1) { }
}

void GLCD_putchar(int c) {
    int i;
    for (i = 0; i < 6; i++)
        GLCD_data_write(font_table[c][i]);
}

void GLCD_setCursor(unsigned char x, unsigned char 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 SPI0_init(void) {
    SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
    SIM->SCGC5 |= 0x0800; /* enable clock to Port C */
    PORTD->PCR[1] = 0x200; /* make PTD1 pin as SPI SCK */
    PORTD->PCR[2] = 0x200; /* make PTD2 pin as SPI MOSI */
    PORTD->PCR[0] = 0x100; /* make PTD0 pin as DC */
    PORTC->PCR[8] = 0x100; /* make PTC8 pin as RST */
    PORTC->PCR[9] = 0x100; /* make PTC9 pin as CE */
    PTD->PDDR |= 0x01; /* make PTD0 as output pin for DC */
    PTC->PDDR |= 0x0200; /* make PTC9 as output pin for /CE */
    PTC->PSOR = CE; /* deassert /CE */
    PTC->PCOR = RESET; /* assert reset */
    SIM->SCGC4 |= 0x400000; /* enable clock to SPI0 */
    SPI0->C1 = 0x10; /* disable SPI and make SPI0 master */
    SPI0->BR = 0x60; /* set Baud rate to 1 MHz */
    SPI0->C1 |= 0x40; /* Enable SPI module */
}

/* send the initialization commands to PCD8544 GLCD controller */
void GLCD_init(void) {
    SPI0_init();

    /* hardware reset of GLCD controller */
    PTC->PSOR = RESET; /* deassert 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(unsigned char data) {
    /* select data register */
    PTD->PSOR = DC;           /* set DC */
    /* send data via SSI */
    SPI0_write(data);
}

/* write to GLCD controller command register */
void GLCD_command_write(unsigned char data) {
    /* select command register */
    PTD->PCOR = DC;           /* clear DC */
    /* send data via SSI */
    SPI0_write(data);
}

void SPI0_write(unsigned char data) {
    volatile char dummy;
    PTC->PCOR = CE;           /* assert /CE */

    while(!(SPI0->S & 0x20)) { } /* wait until tx ready */
    SPI0->D = data;             /* send register address */
    while(!(SPI0->S & 0x80)) { } /* wait until tx complete */
    dummy = SPI0->D;            /* clear SPRF */
    PTC->PSOR = CE;             /* deassert /CE */
}

```

## 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 display 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 a 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	
<b>Memory cycle time with 0 WS</b>	$2 \times 10 = 20 \text{ ns}$
<b>Memory cycle time with 1 WS</b>	$20 + 10 = 30 \text{ ns}$
<b>Memory cycle time with 2 WS</b>	$20 + 10 + 10 = 40 \text{ ns}$

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.

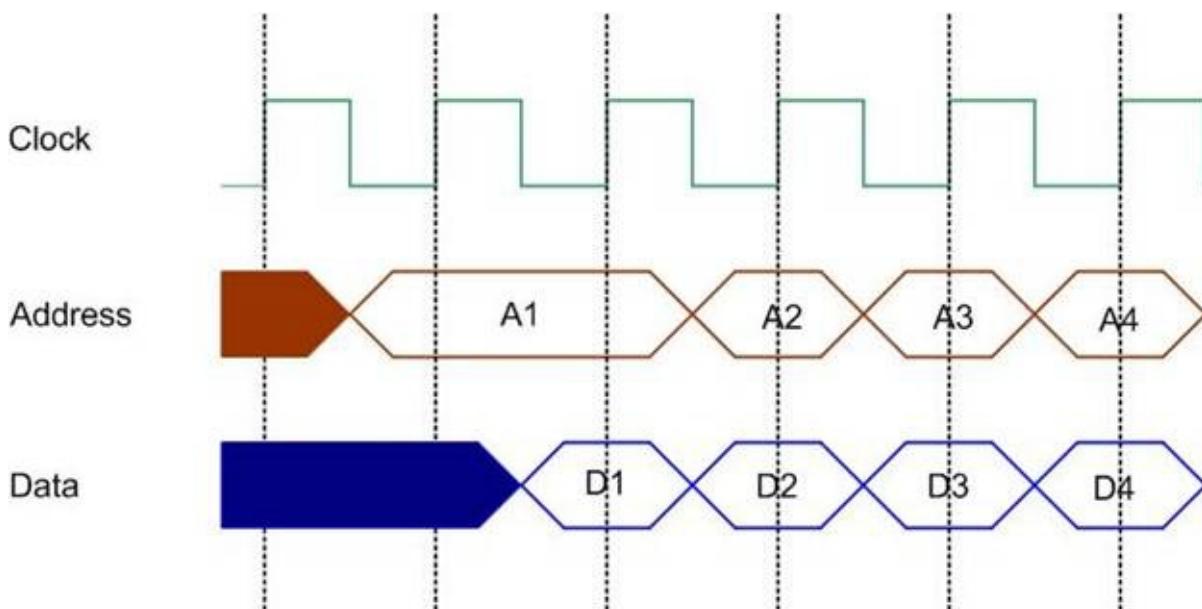


Figure 13-1: Burst Cycle Read in CPU

## 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  $1\text{M} \times 1$  organization, or 4 banks of  $256\text{M} \times 8$  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

D15 ... D12	D11 ... D8	Parity	D7 ... D4	D3 ... D0	Parity
64M x 4	64M x 4	64M x 1	64M x 4	64M x 4	64M x 1
256M x 4	256M x 4	256M x 1	256M x 4	256M x 4	256M x 1

Figure 13-3: DRAM Banks for 16-bit systems

## Memory cycle in ARM

Memory transfer cycle in ARM is one of these categories:

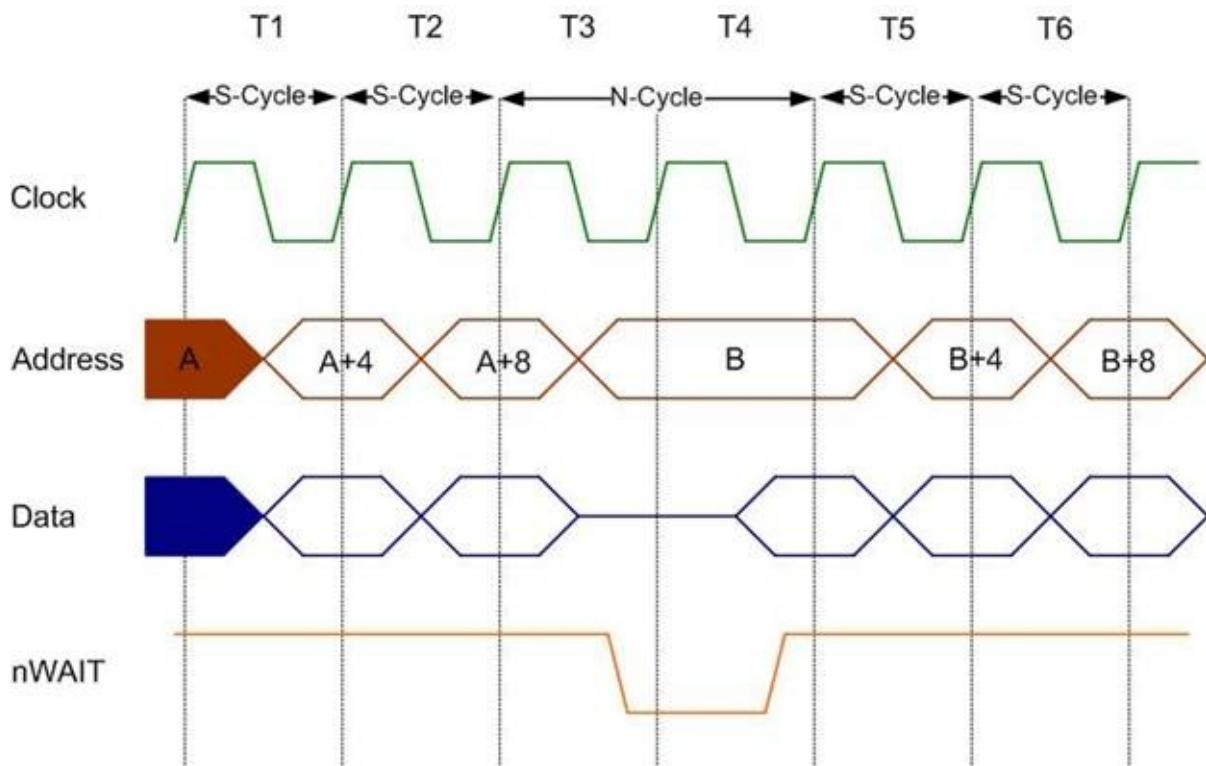
1. Non-sequential cycle: in non-sequential cycle, a location of memory is accessed which is not the same or near the last accessed memory location
2. Sequential cycle: in sequential cycle, a memory location is accessed from either the same location or the memory location after the location of preceding 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.

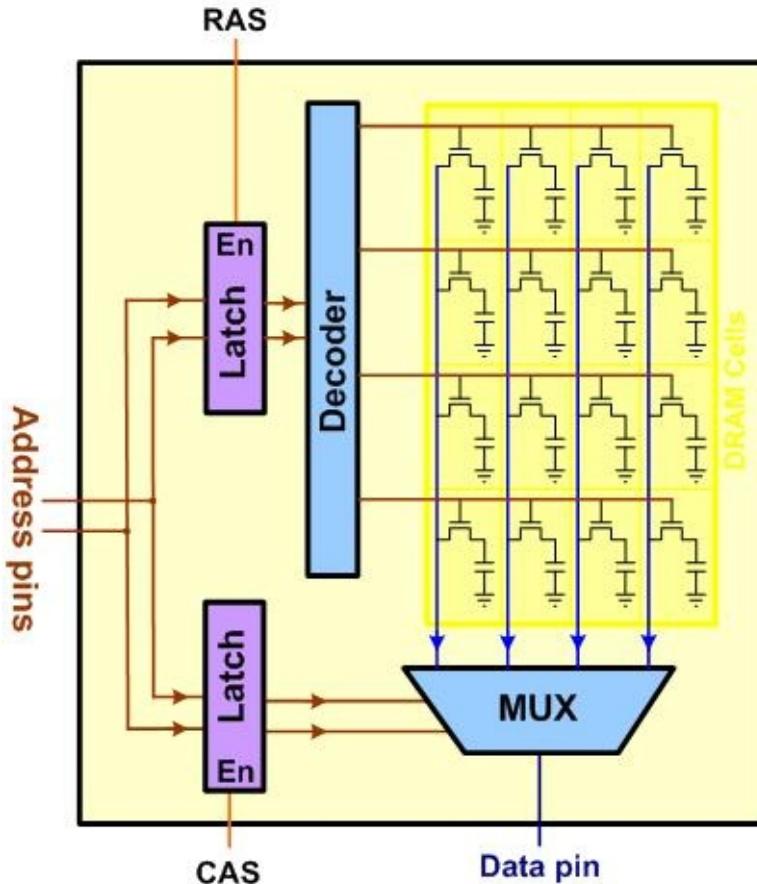


Figure 13-4: The Internal Structure of a 16×1 DRAM

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.

$$t_{RC} = t_{RAC} + t_{RP} \quad (\text{This is for standard mode})$$

$$\text{read cycle time} = \text{RAS access time} + \text{RAS precharge time}$$

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  $1M \times 1$  chip if  $t_{RC} = 85$  ns and  $t_{RAC} = 165$  ns.

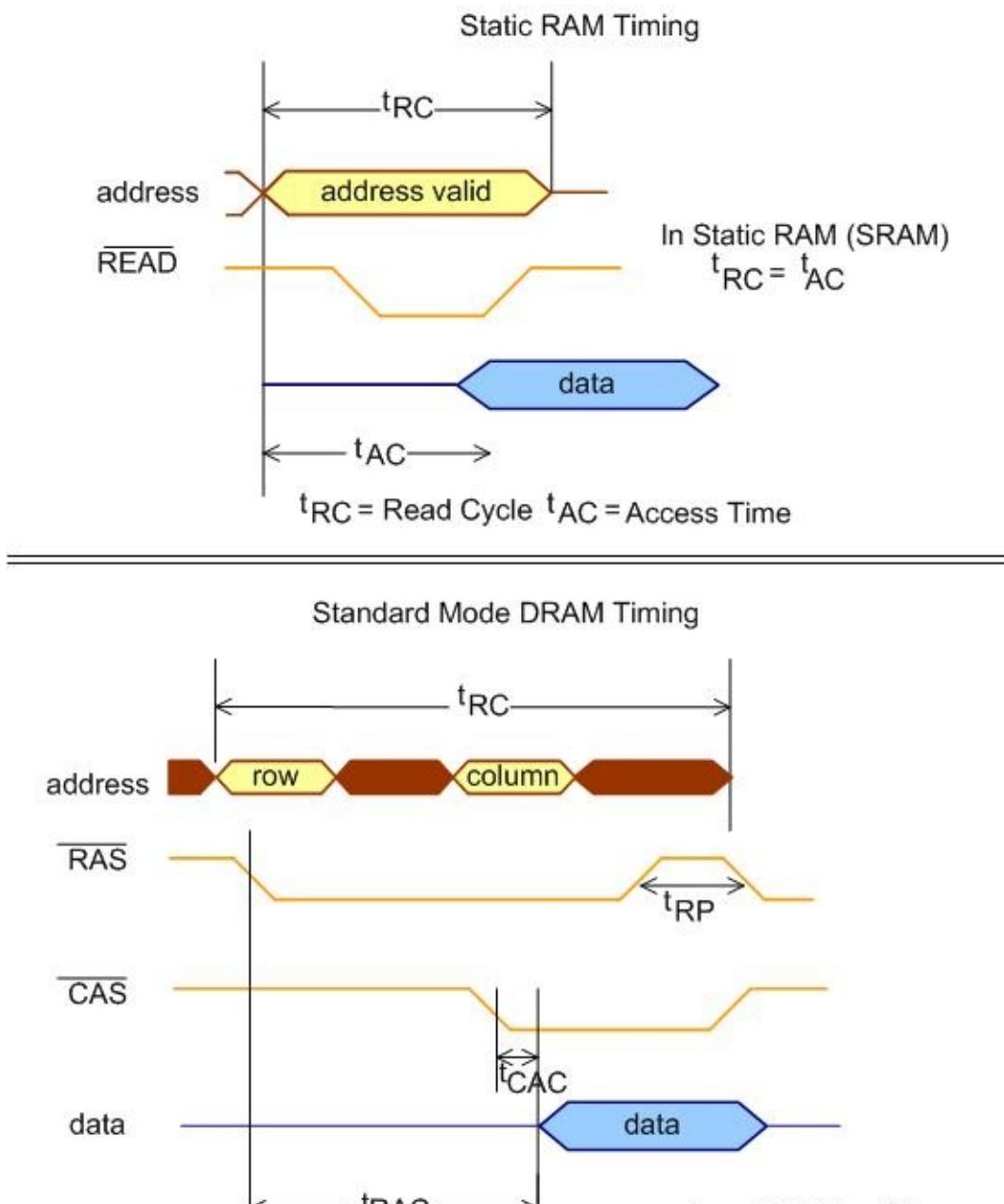
**Solution:**

For standard mode (also called random) we have the following for reading 1024 bits:

$$\text{time to read 1024 random bits} = 1024 \times t_{RC} = 1024 \times 165 \text{ ns} = 168,960 \text{ ns}$$

---

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.

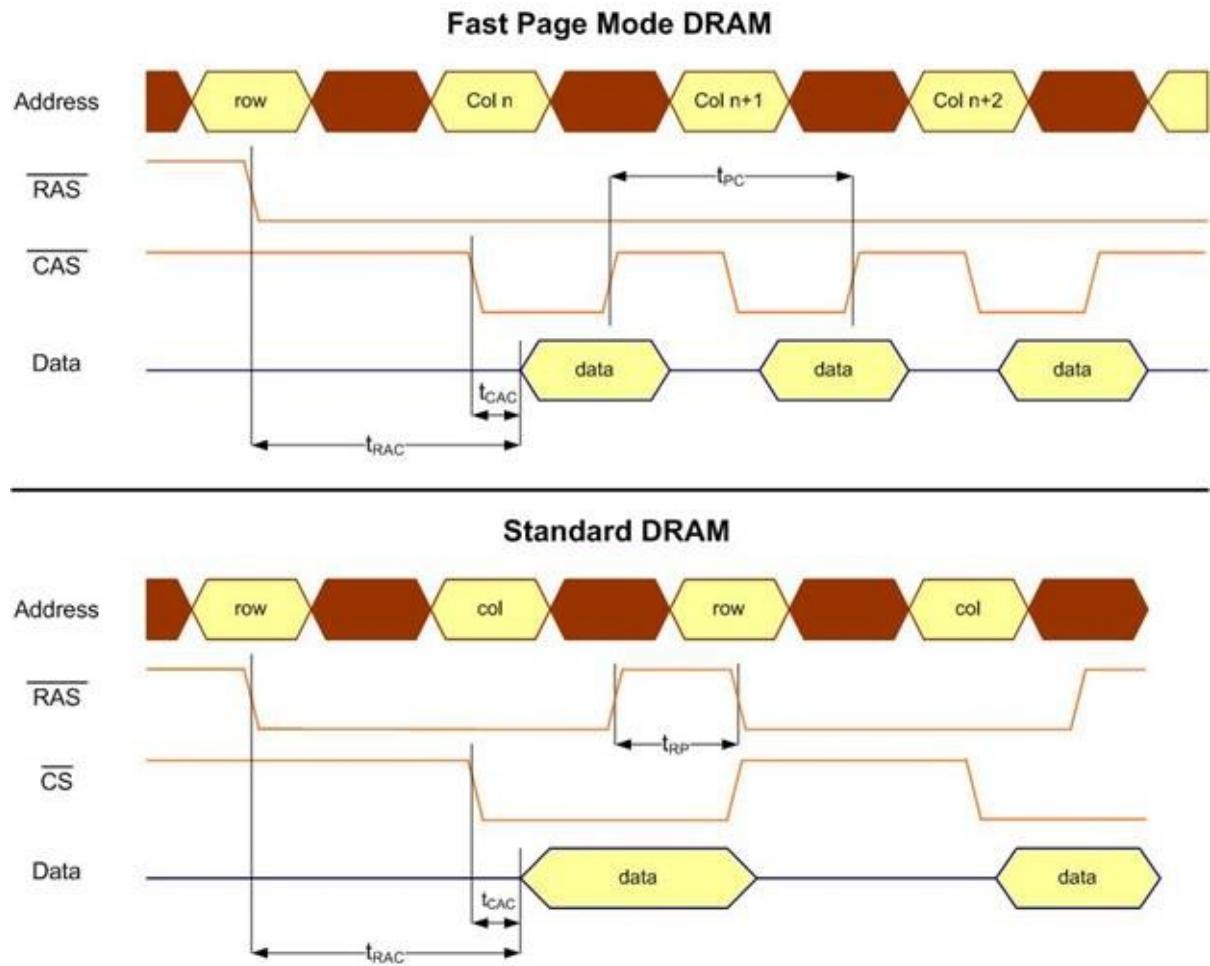
(b) 1024 pages, where each page has 1024 bits (columns).

(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:

$$\text{Time to read 1024 bits of the same page} = t_{RAC} + 1023 \times t_{PC} = 85 \text{ ns} + 1023 \times 50 \text{ ns} = 51,235 \text{ ns}$$

## 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.

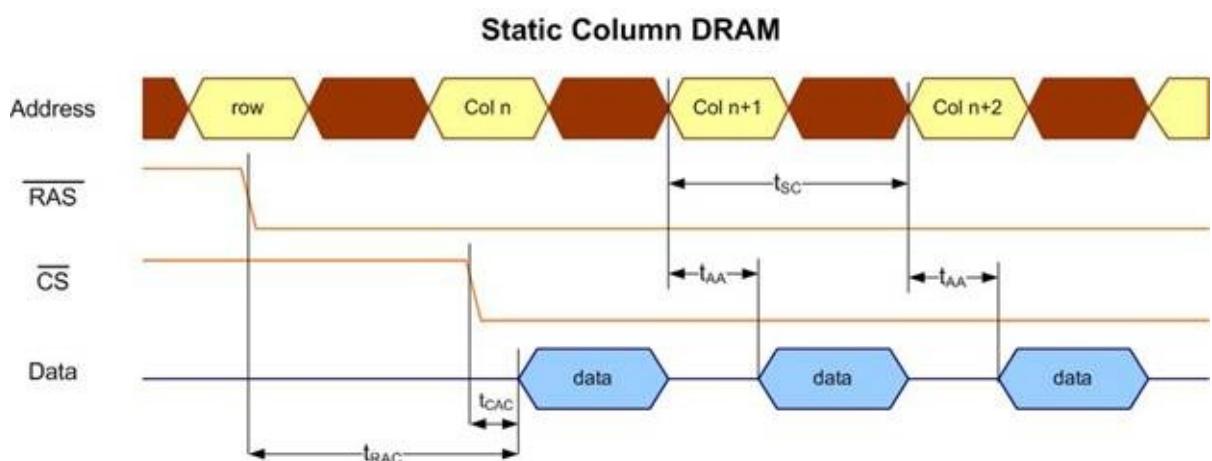


Figure 13-7: DRAM Static Column Mode Timings

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)
--------------------	-----------------------------------	-----------------------------	-------------------------------------	---------------------------

<b>MCM54102A-60</b>	60	110	30	35
<b>MCM54102A-70</b>	70	130	35	45
<b>35MCM54102A-80</b>	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 × 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
<b>Access time from row</b>	$t_{RAC} = 85\text{ns}$	$t_{RAC} = 85\text{ns}$	$t_{RAC} = 85\text{ns}$
<b>Access time from column</b>			$t_{AA} = 45\text{ns}$
<b>Cycle time</b>	$t_{RC} = 165\text{ns}$	$t_{PC} = 50\text{ns}$	$t_{SC} = 50\text{ns}$

**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
$t_{RAC}(\text{ns})$	60	60
$t_{RC}(\text{ns})$	110	110
$t_{PC}(\text{ns})$	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.

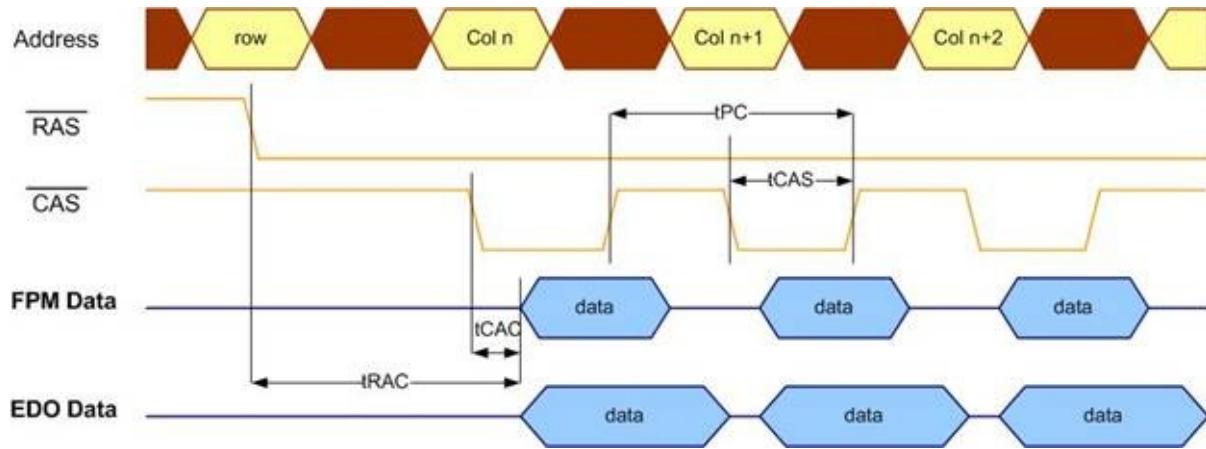


Figure 13-8: FPM and EDO Comparison

## 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.

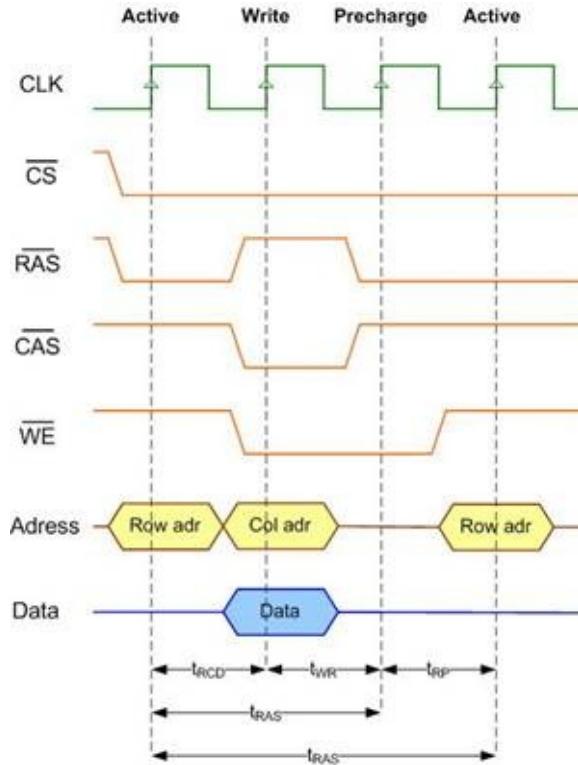


Figure 13-9: SDRAM Simple Write Command

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  $t_{RCD}$  more than 15ns. nop2 and nop3 are added to make  $t_{RAS}$  more than 37ns and nop4 and nop5 are added to make  $t_{RC}$  more than 60ns.

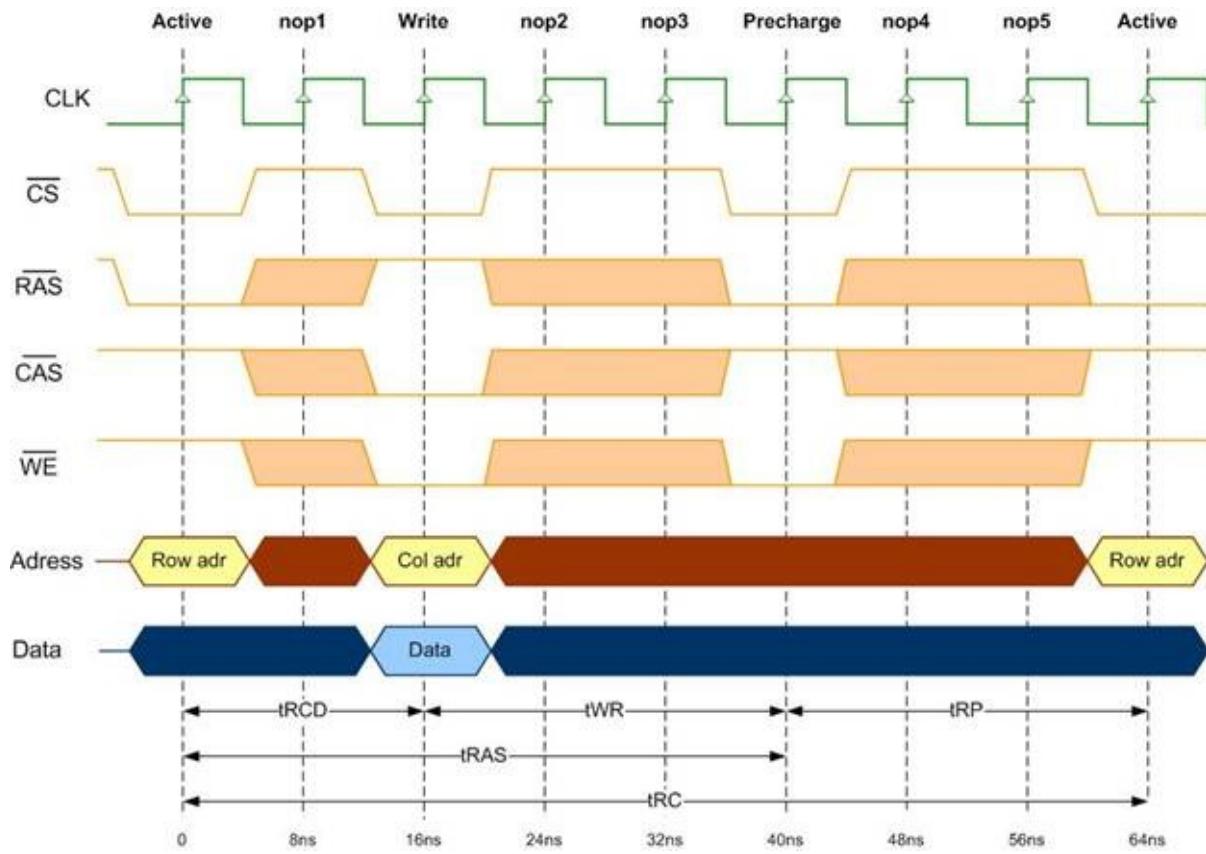


Figure 13-10: SDRAM Write Operation with NOPs

## 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.

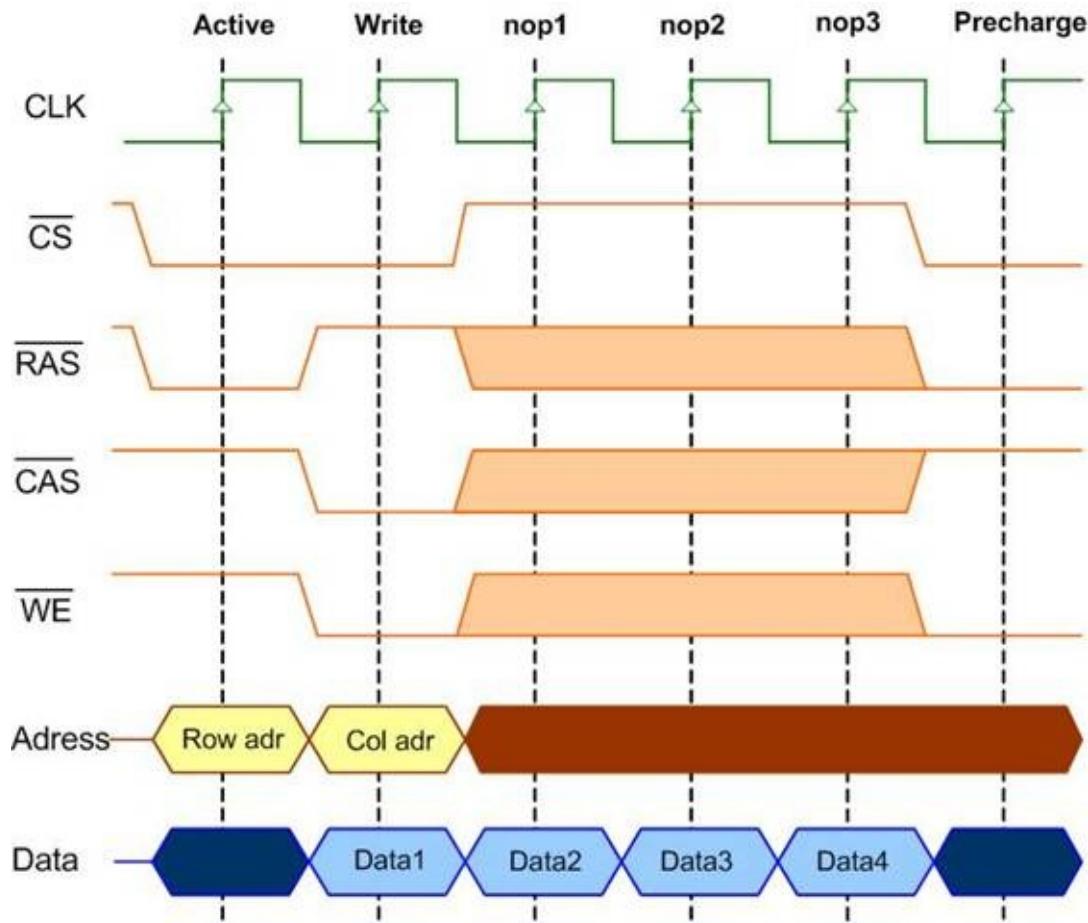


Figure 13-11: SDRAM Burst Write Operation

## 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.

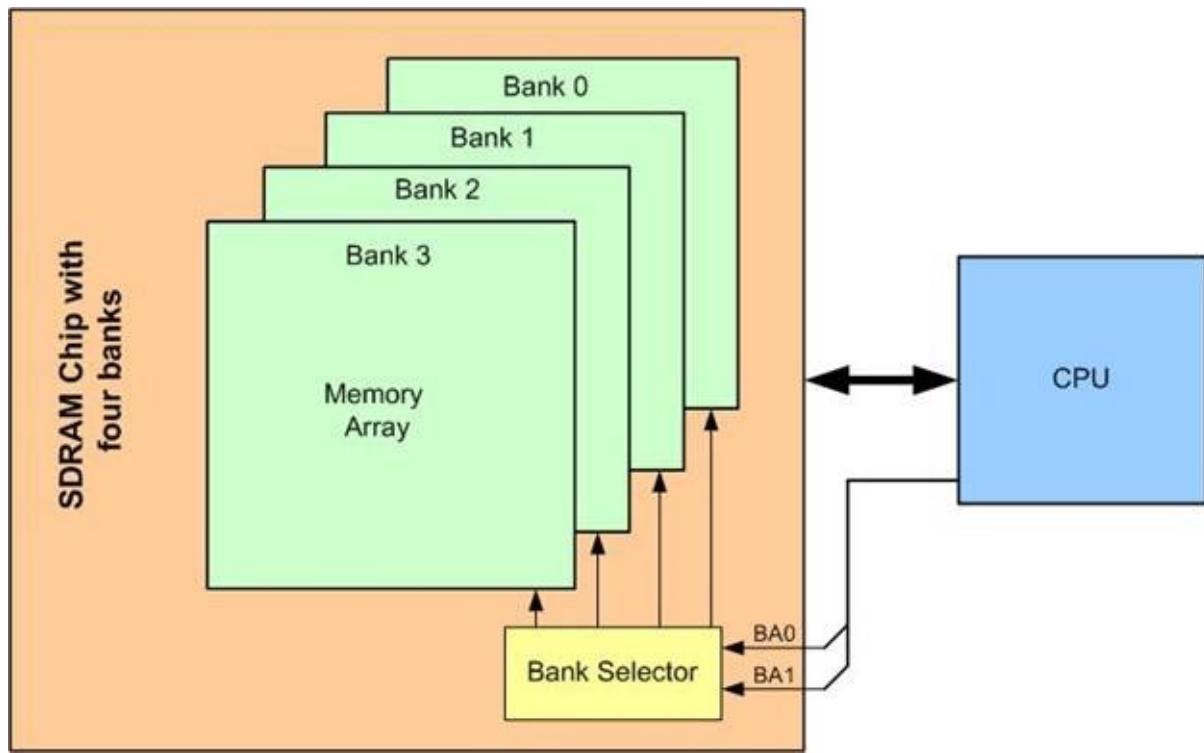


Figure 13-12: SDRAM with Four Banks

## 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

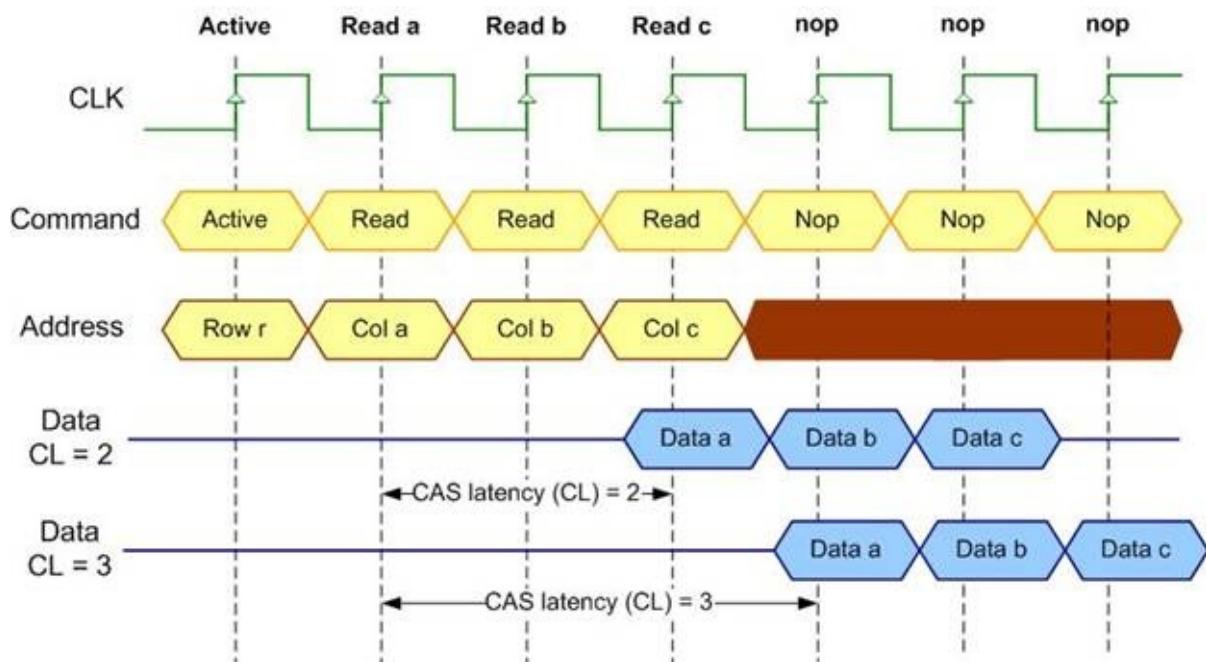


Figure 13-13: SDRAM CAS Latency

## 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.

$$\begin{array}{r} 25 \\ + 62 \\ \hline \end{array}$$

$$\begin{array}{r} + 3F \\ + 52 \\ \hline 1 18 \end{array}$$

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.

$$\begin{array}{r} 25 \\ + 62 \\ + 3F \\ + 52 \\ + \underline{E8} \\ \hline 2 00 \text{ (dropping the carry)} \end{array}$$

(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.

$$\begin{array}{r} 25 \\ + 22 \\ + 3F \\ + 52 \\ + \underline{E8} \\ \hline 1 C0 \text{ dropping the carry, we get 0xC0.} \end{array}$$

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.

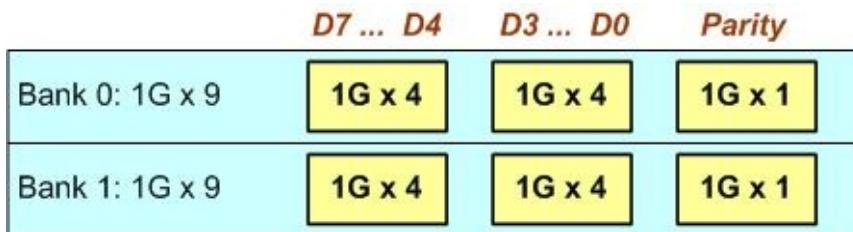


Figure 13-14: A Possible Memory Configuration for 2G DRAM

## Review Questions

1. 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.

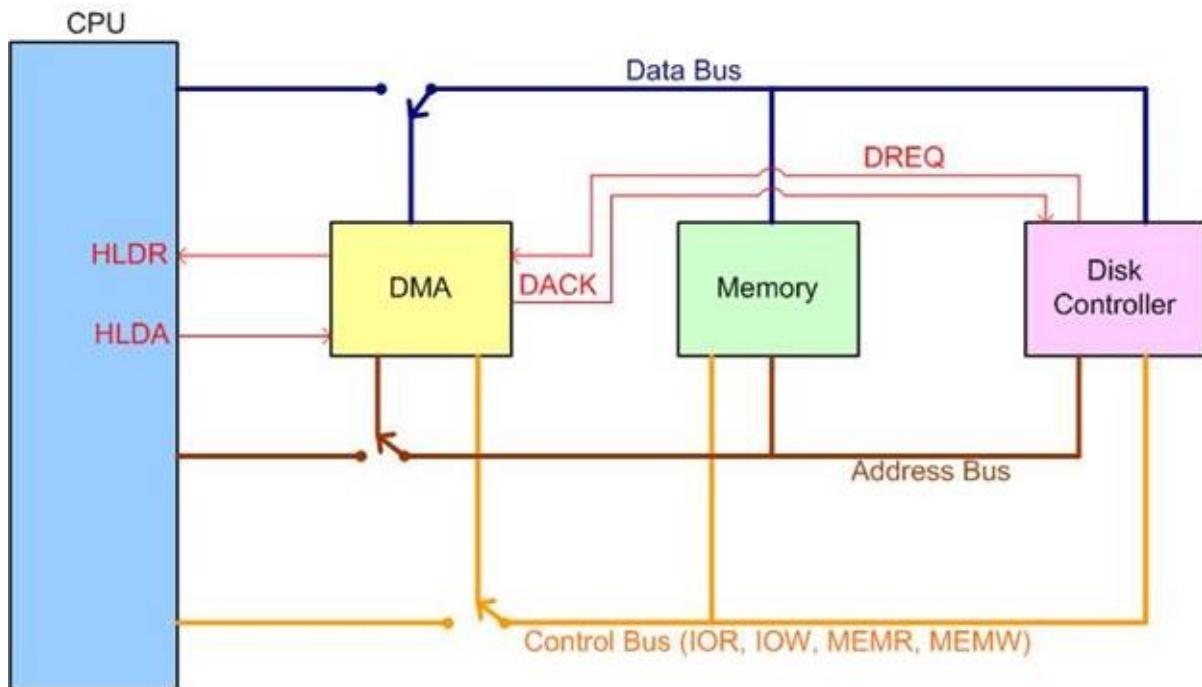


Figure 13-15: DMA Usage of System Bus

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  $256K \times 9$  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.

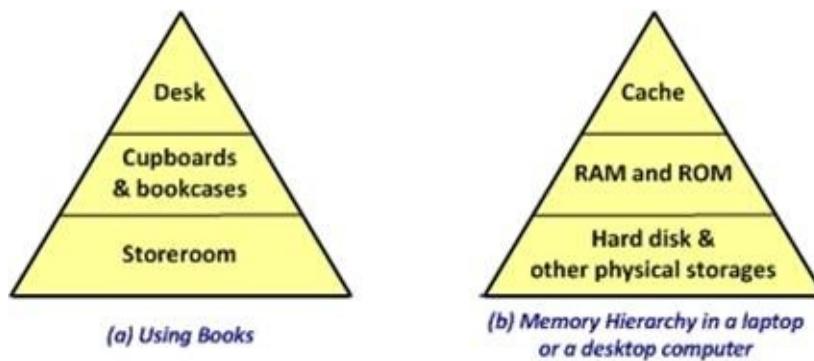


Figure 14-1: Memory Hierarchy

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.

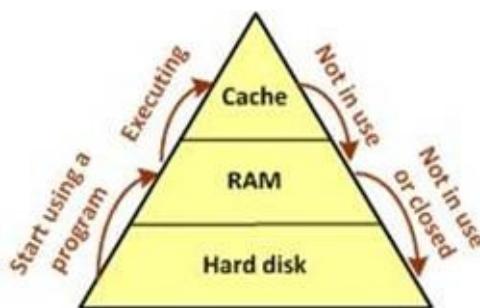


Figure 14-2: a Program Moving in the Memory Hierarchy

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.

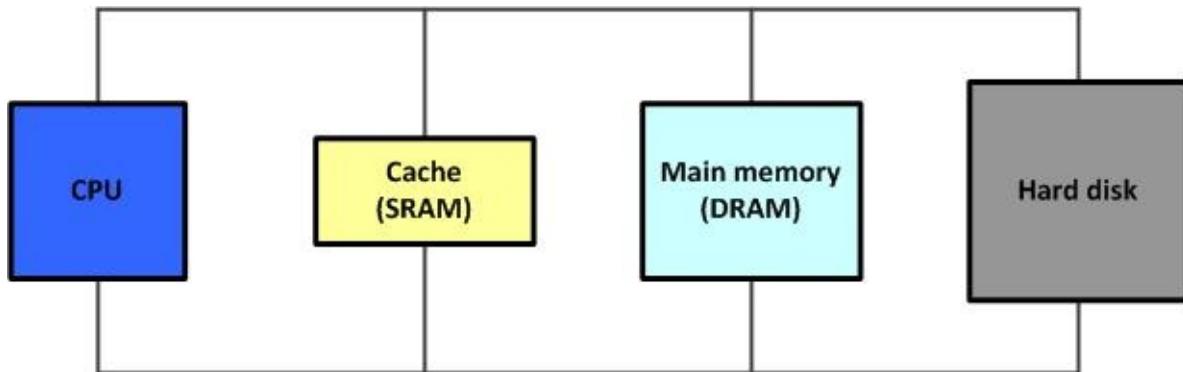


Figure 14-3: CPU and Its Relation to Various Memories

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.

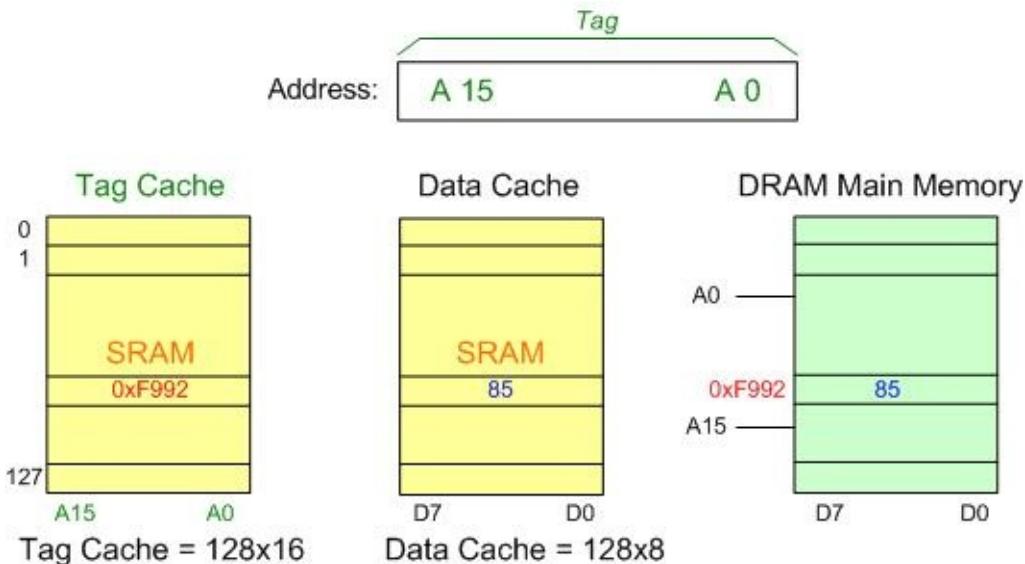


Figure 14-4: a 128-Byte Fully Associative Cache for a 16-bit System

## 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.

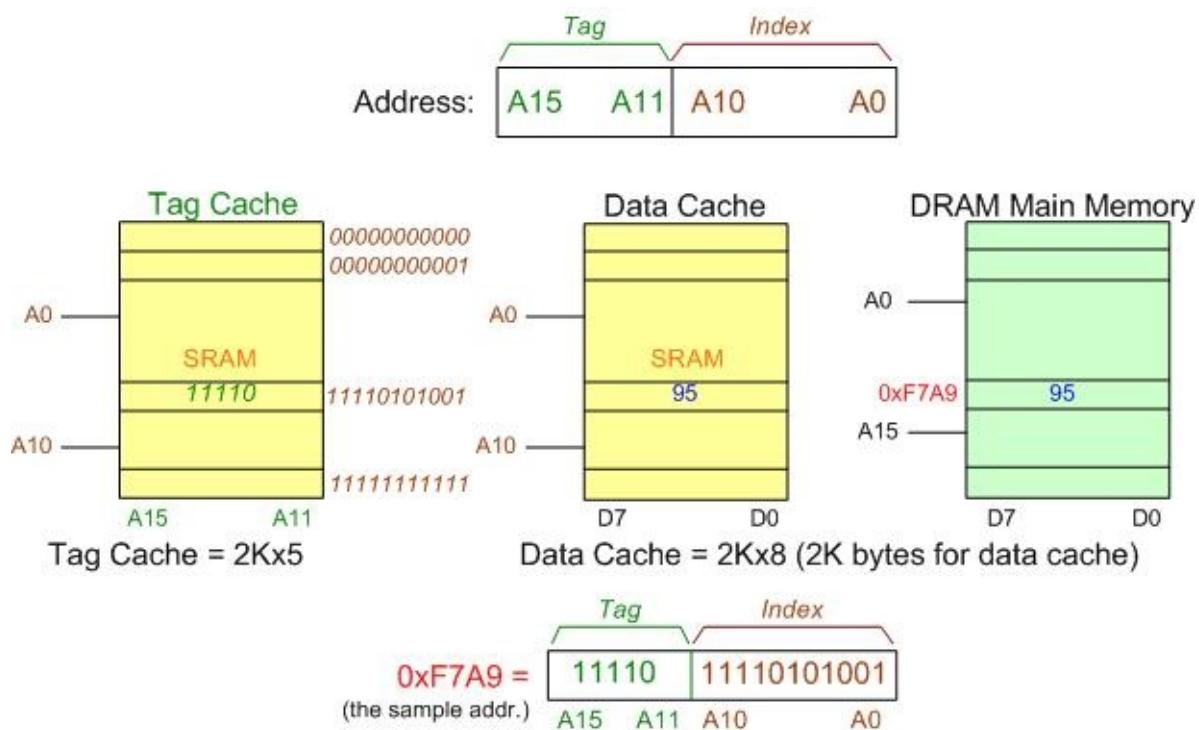
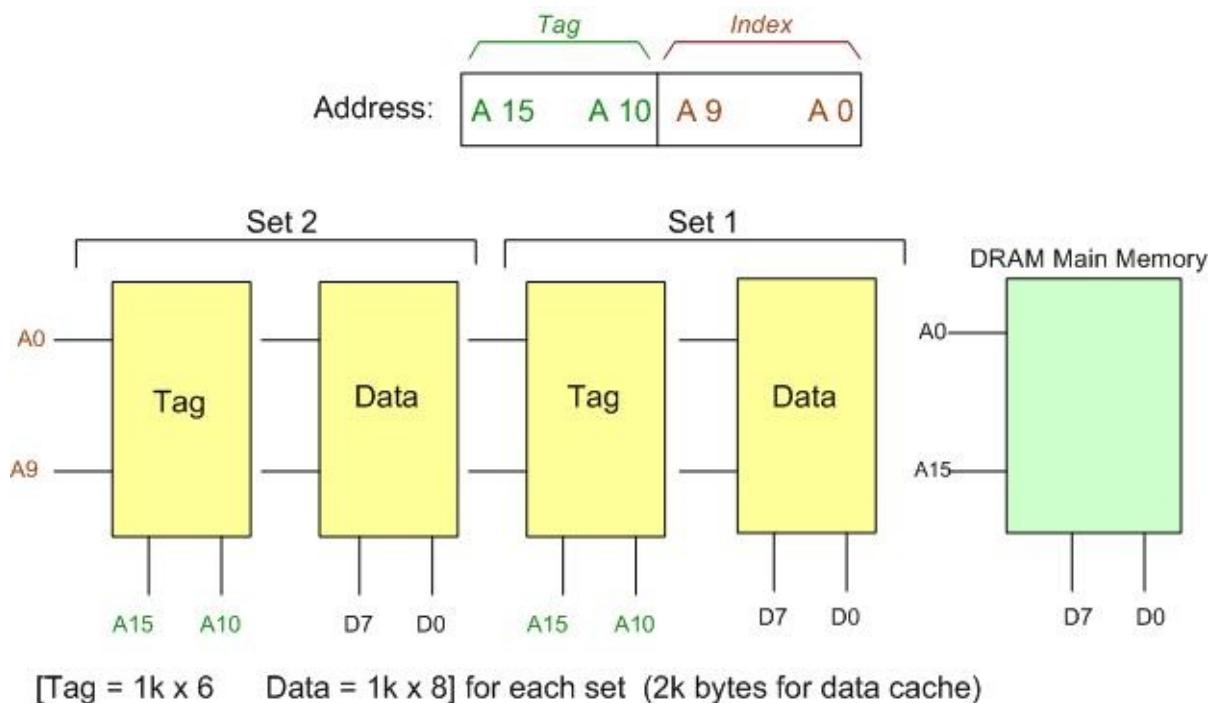


Figure 14-5: a 2KB Direct-Mapped Cache for a 16-bit System

## 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



**Figure 14-7: a 2KB Four-way Set Associative Cache for a 16-bit System**

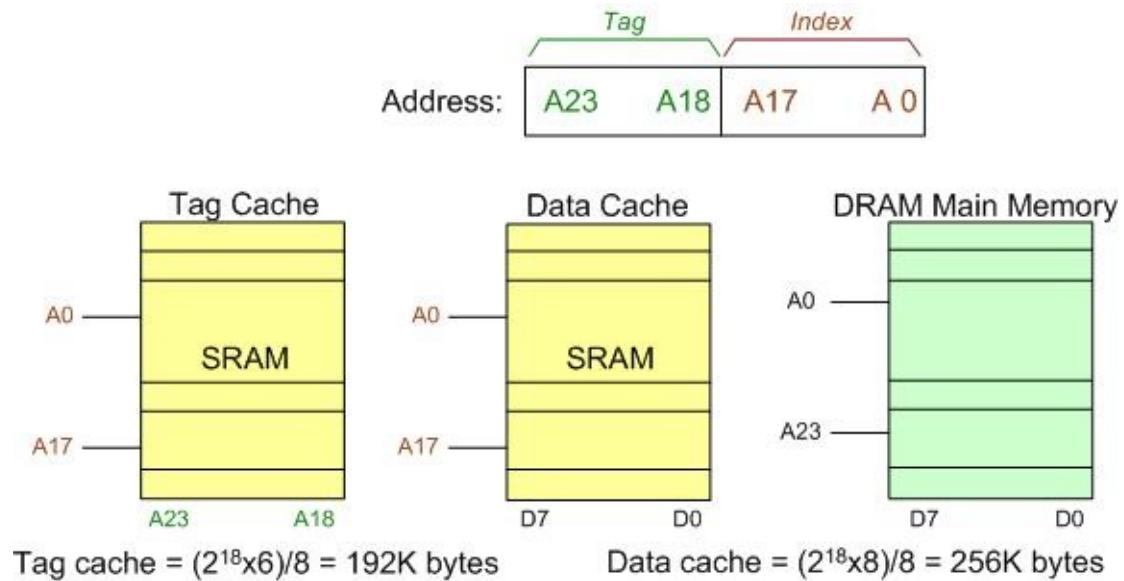
Comparing direct-mapped and 2-way set associative, one can see that with only a small amount of extra SRAM, a better hit rate can be achieved. In this organization, if the microprocessor is requesting the contents of memory location 0x41E6, there are 2 possible tags that could hold it, since cache circuitry 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  $1K \times 6$  and data of  $1K \times 8$  for each set gives a total of 14K bits [ $2 \times (1K \times 6 + 1K \times 8) = 28K$  bits]. In 4-way, there is  $512 \times 7$  for the tag and  $512 \times 8$  for data, giving a total of 32K bits [ $(512 \times 7 + 512 \times 8) \times 4 = 32K$  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.

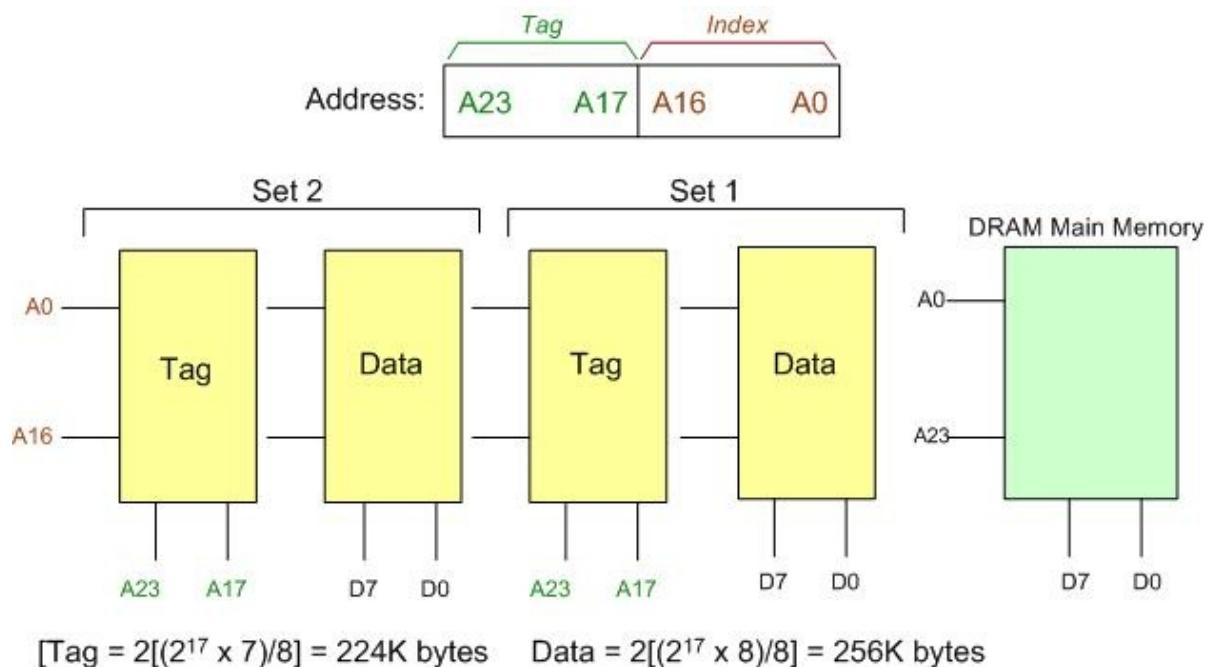
### Example 14-1

This example shows directed-mapped cache for 16M main memory.



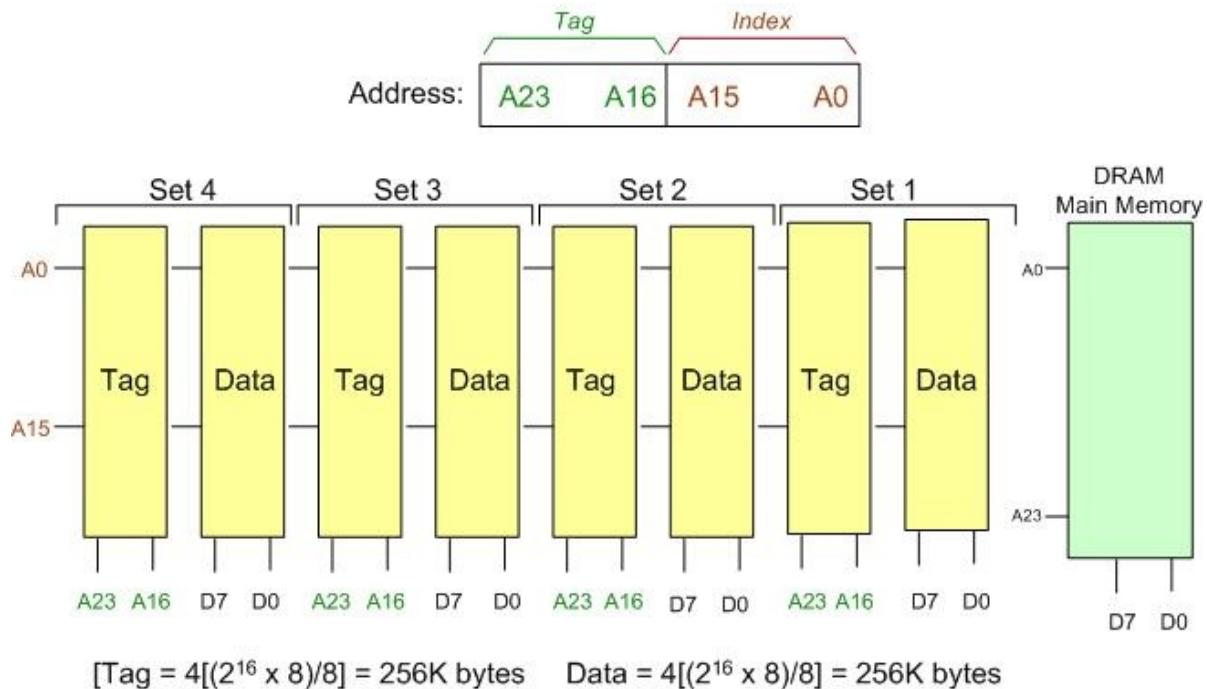
### Example 14-2

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.

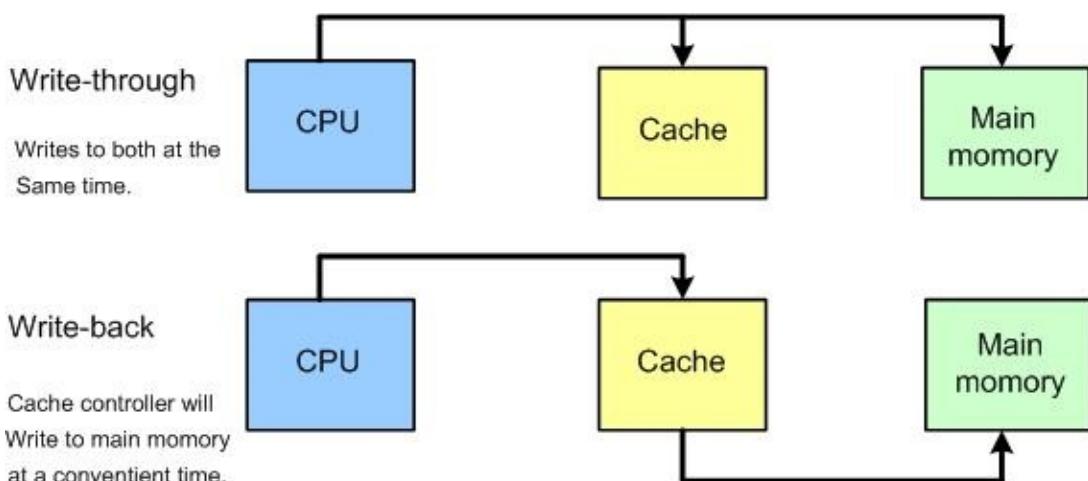


Figure 14-8: Method of Updating Main Memory

### 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.

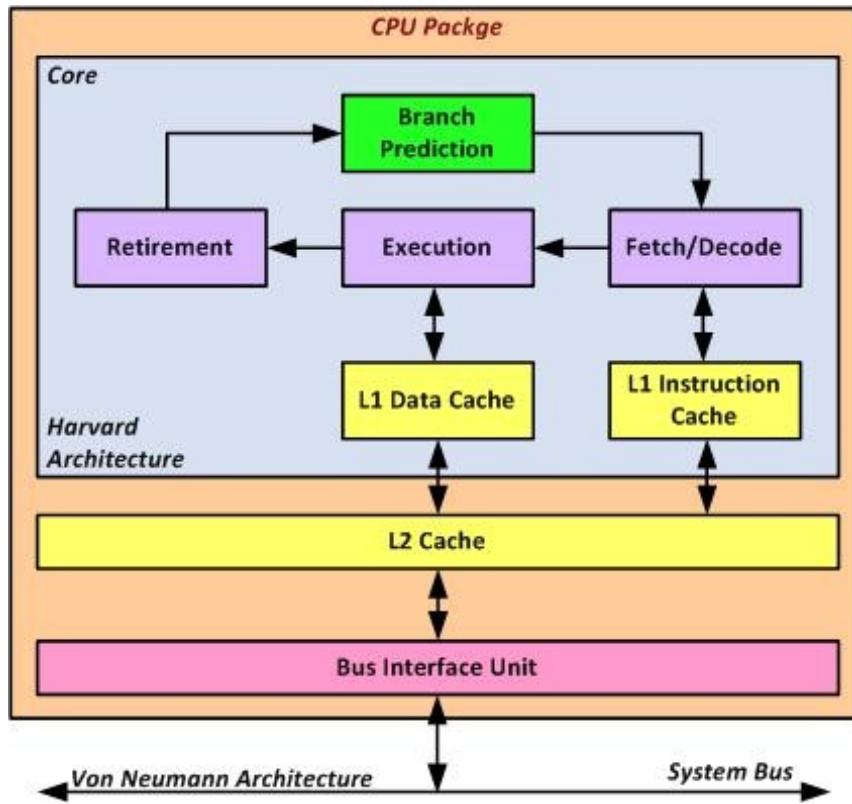


Figure 14-9: L2 Cache Feeding Code and Data to L1 Caches

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.

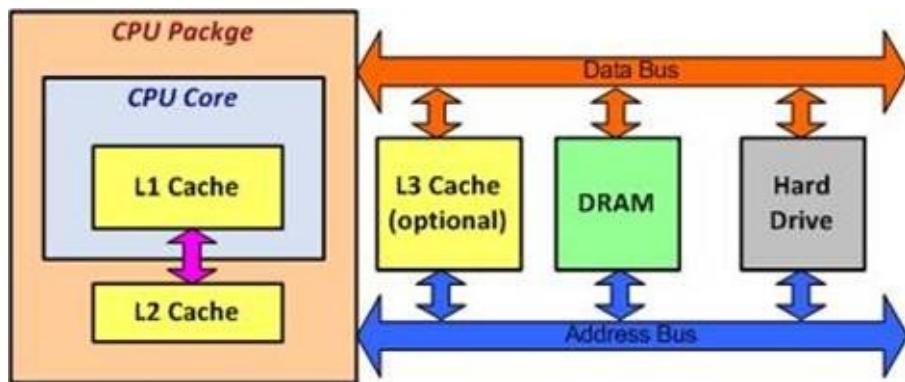


Figure 14-10: L1, L2, and L3 Cache

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:

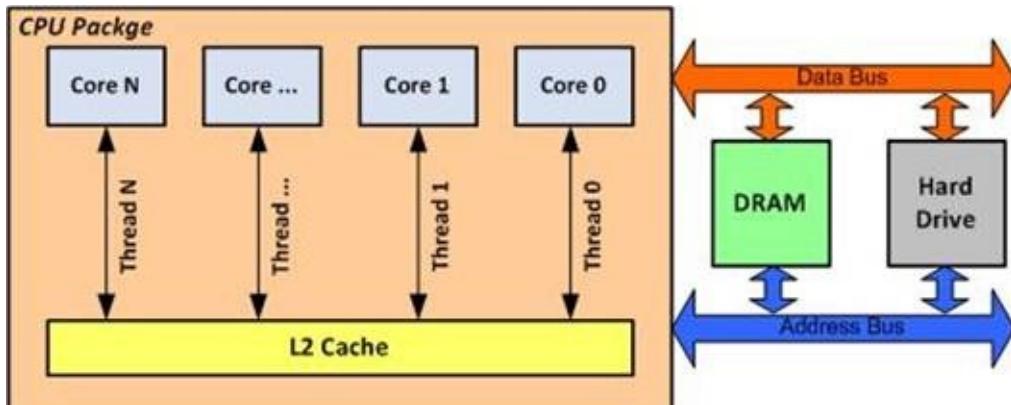
1. The speed of the L1 cache is the same as the CPU speed, since it must

feed the CPU as fast as the instructions are executed.

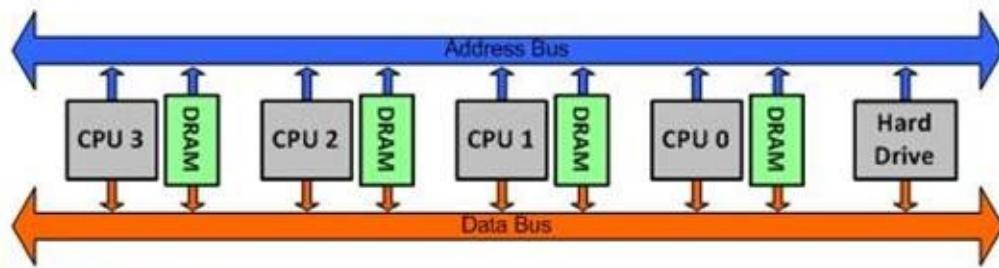
2. L2 cache works at the a fraction of the CPU since it is on the same package as the CPU.
3. L3 cache works at a fraction of the speed of the bus since it is located outside the CPU package.

## 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 called 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/paedia/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.

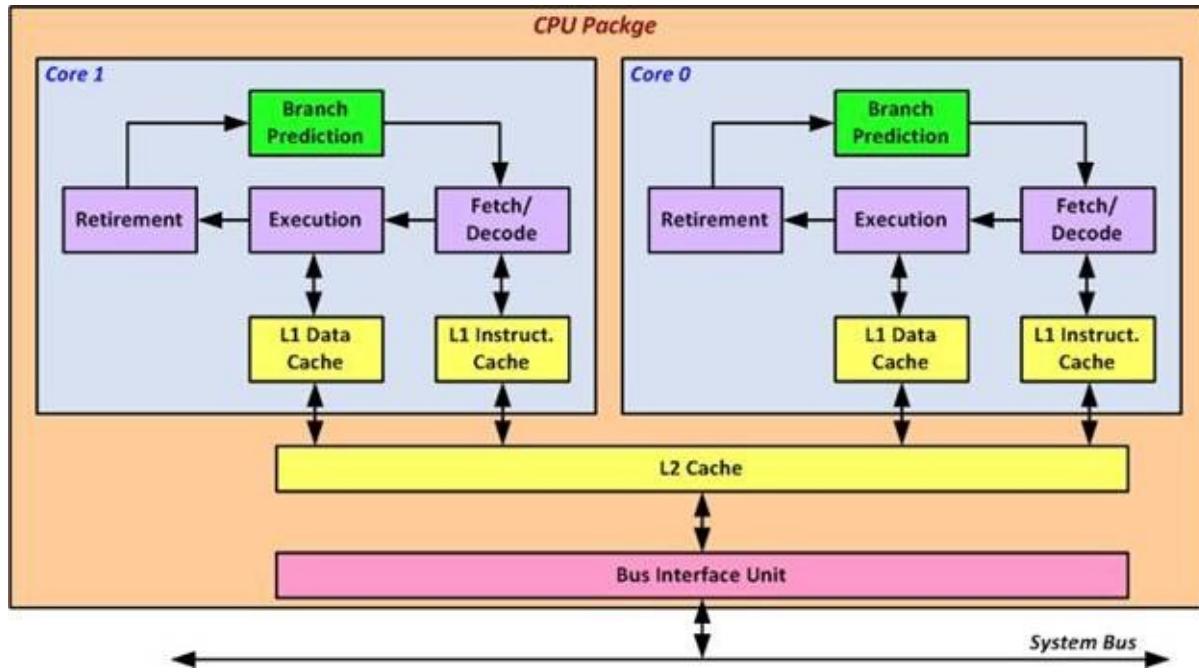


Figure 14-12: An Example of Dual-Core Processor

## 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.

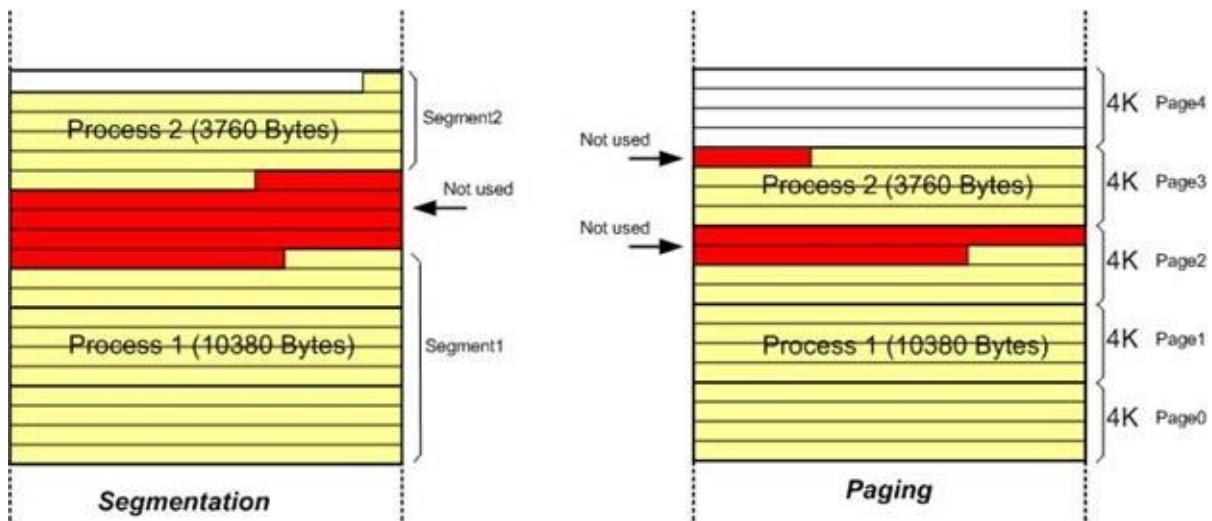
In systems without virtual memory, only one task can be active at a time and all other tasks are sitting idle (dormant). In multitasking operating systems such as Microsoft Windows and Linux each task is given a slice of the CPU's time, and many tasks can be active concurrently. For example, a word processor 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.

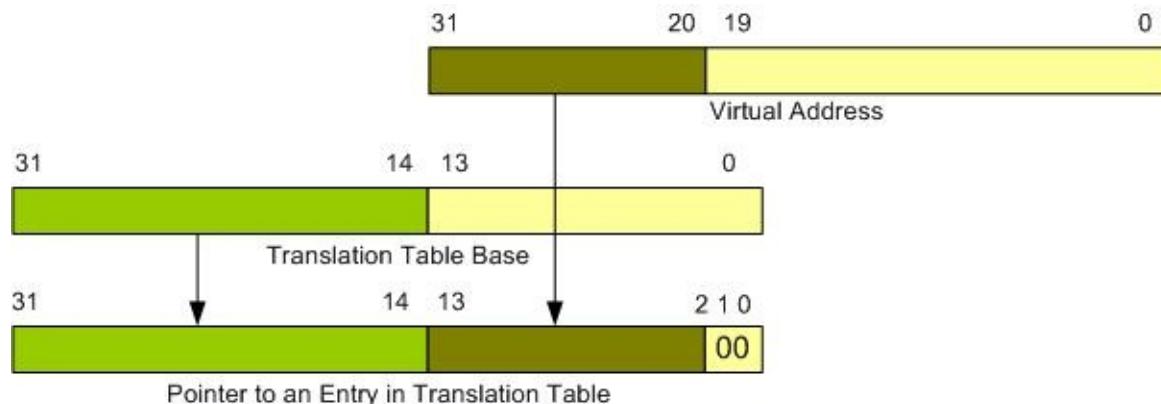


Figure 15-2: Accessing Translation Table

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.

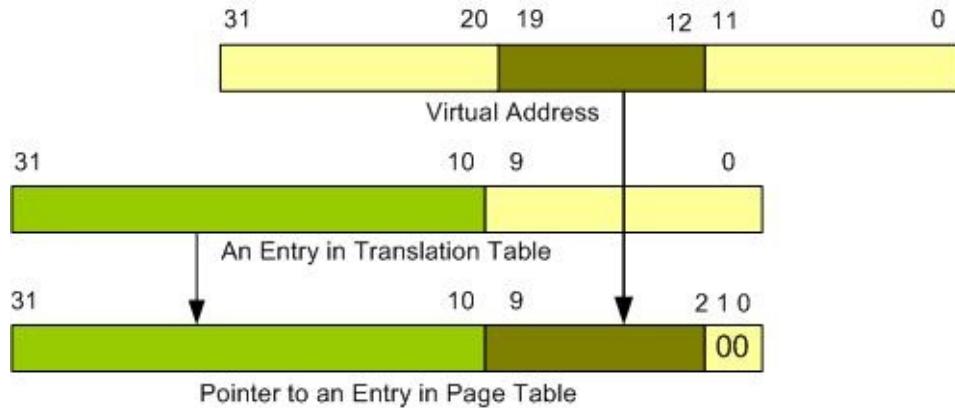


Figure 15-3: Accessing Page Table

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.

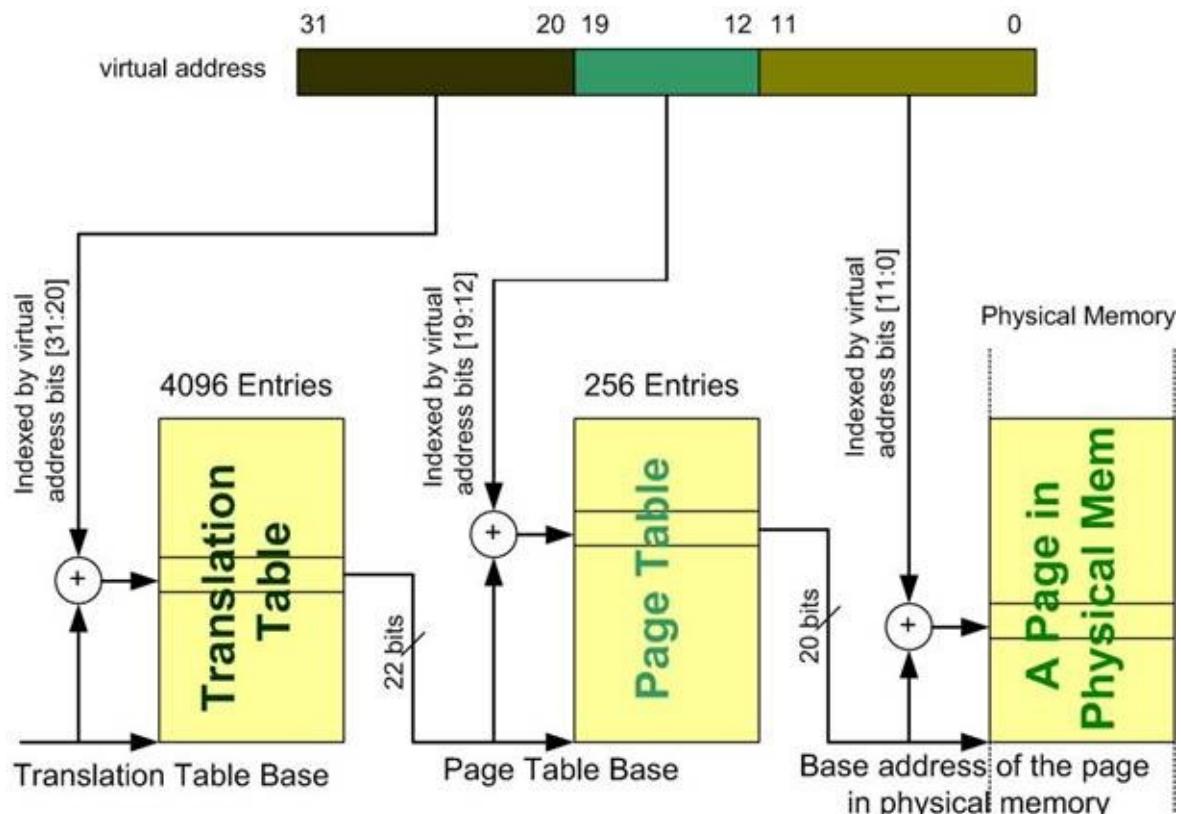


Figure 15-4: 4K Paging Mechanism in ARM

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.

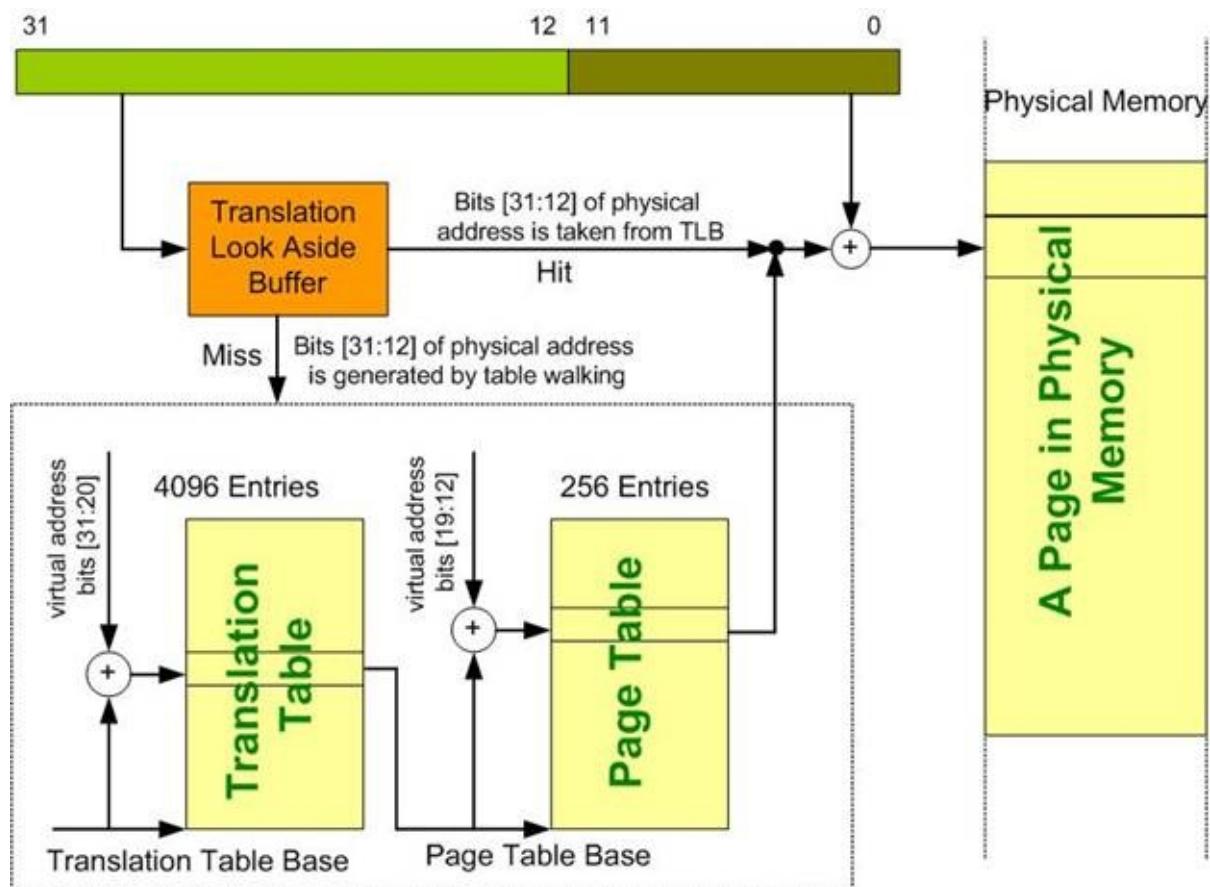


Figure 15-5: Translation Look Aside Buffer in ARM

## 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  $64 \times 32\text{-bit}$  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:

- a) 4K page
- b) 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:

- 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.
- 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.

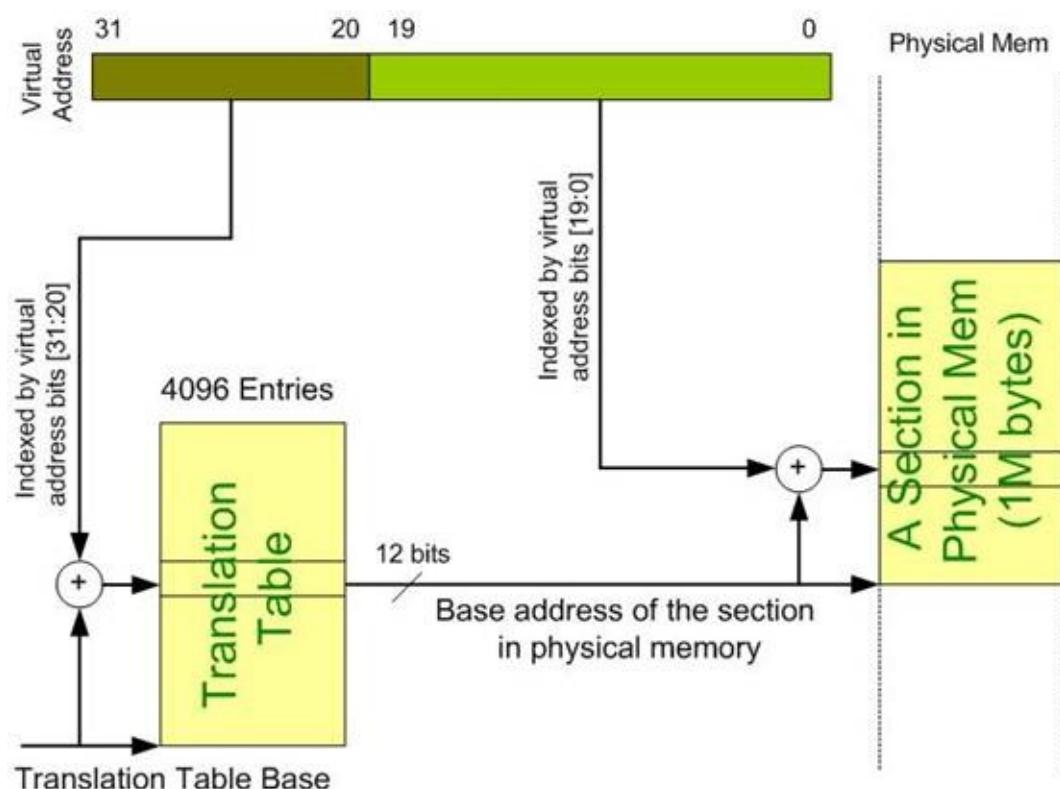


Figure 15-6: One Level Virtual-to-Physical Address Translation for 1M Page in ARM

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 ( $64 \times 1\text{M} = 64\text{M}$ ) 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 moving 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 systems 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.

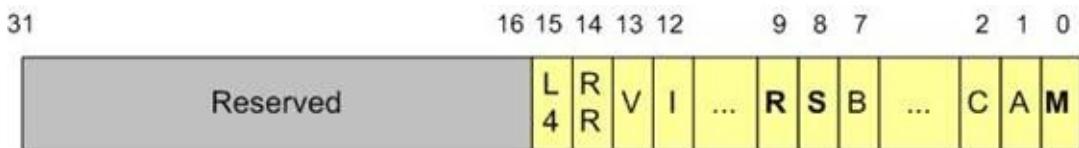


Figure 15-7: c1 Bits

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

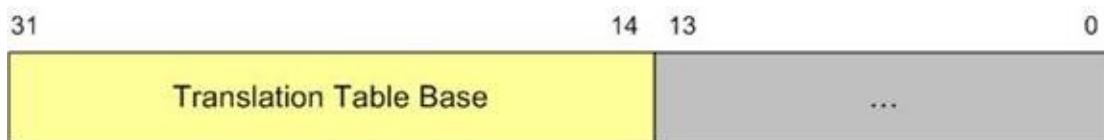
This bit allows code or data to be write-protected. For example, the core of the operating system can be write-protected to prevent from writing into it and crashing the system. In the case of old DOS, any program could alter the core of the operating system residing in main memory (DRAM), thereby crashing 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

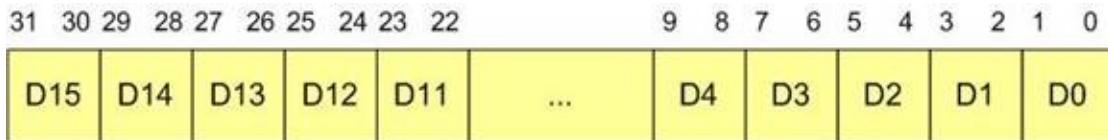


Figure 15-9: c3 Domain register

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.

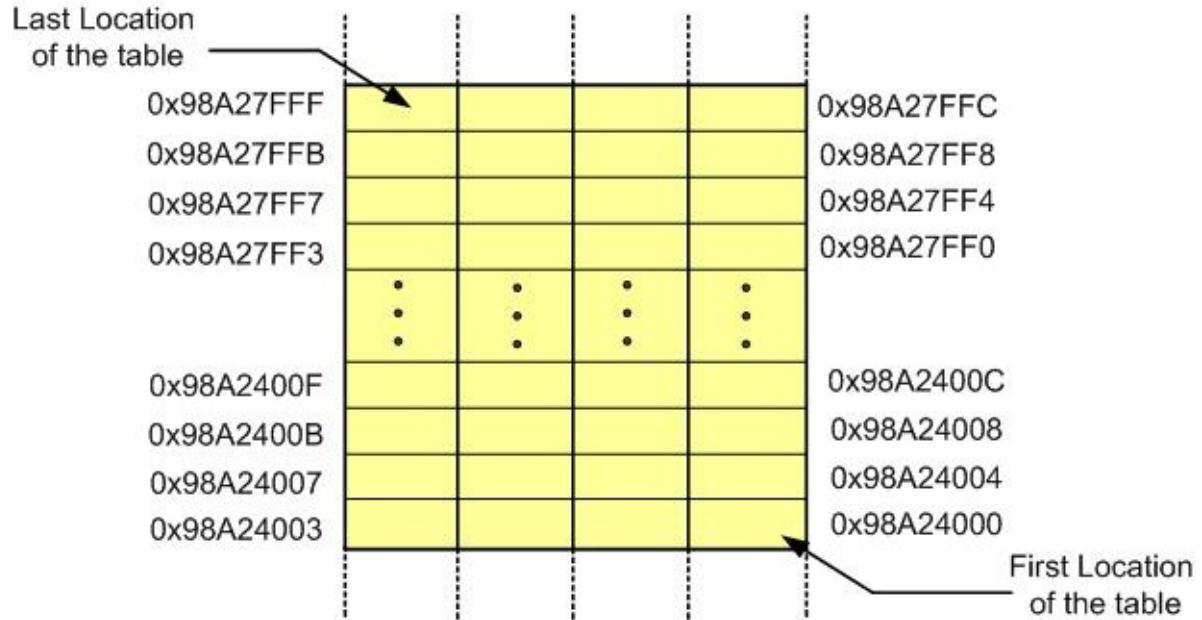
### Example 15-2

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.

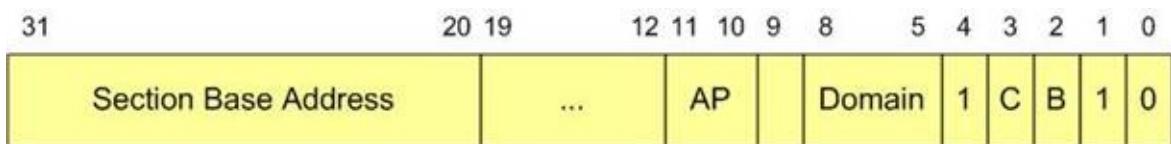


Figure 15-10: 1M page(section) descriptor

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 blocks 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 give 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:

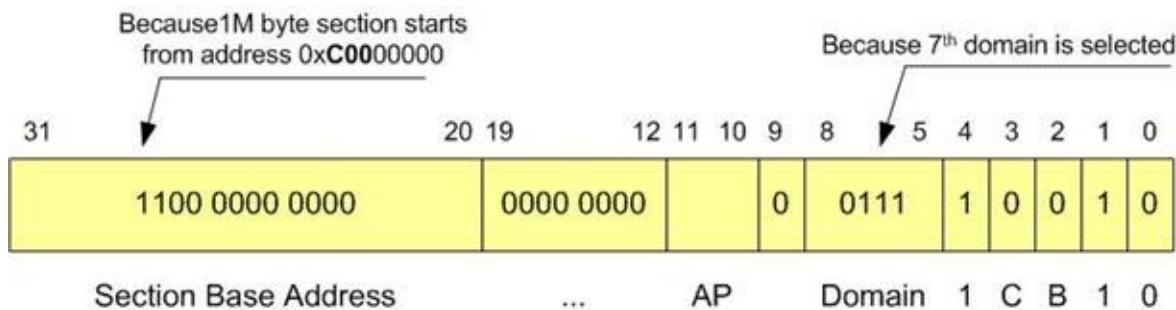
- a) Location of descriptor table if virtual address for the desired section is

0x00900400.

- b) 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.
- c) 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 0x60000000 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 0000 0XX0 1111 0010.
- c) The exact physical address of the of 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.

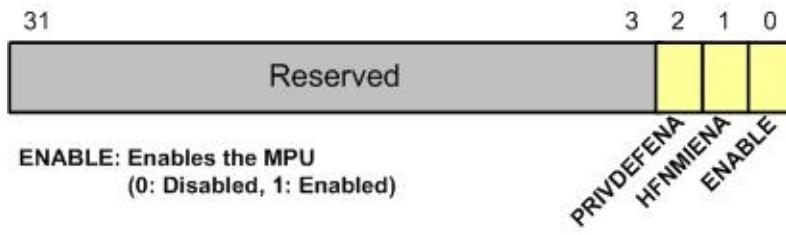


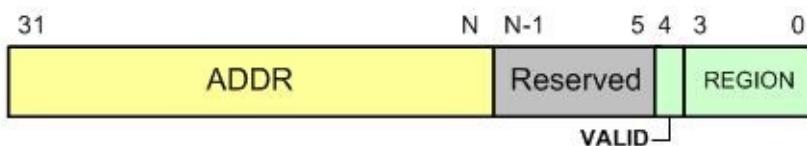
Figure 15-14: MPU\_CTRL Register

## 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 lowers 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	$2^{(N+1)}$	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

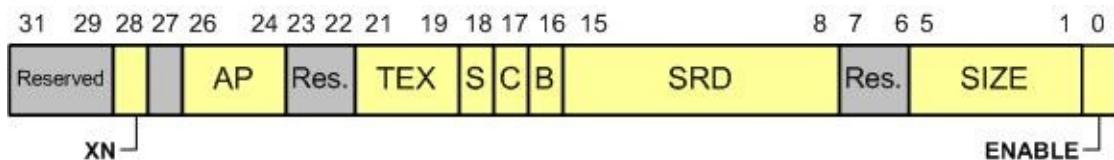
Table 15-5: Allowed region size for ARM Cortex MPU

### 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  $2^{(9+1)} = 1KB$
- (b) With N=01111 we have  $2^{(15+1)} = 64KB$
- (c) With N=11101 we have  $2^{(29+1)} = 1GB$
- (d) With N=11111 we have  $2^{(31+1)} = 4GB$



**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

<b>000</b>	No access	No access	All accesses generate a permission fault
<b>001</b>	RW	No access	Access from privileged software only
<b>010</b>	RW	RO	Writes by unprivileged software generate a permission fault
<b>011</b>	RW	RW	Full access
<b>100</b>	Unpredictable	Unpredictable	Reserved
<b>101</b>	RO	No access	Reads by privileged software only
<b>110</b>	RO	RO	Read only, by privileged or unprivileged software
<b>111</b>	RO	RO	Read only, by privileged or unprivileged software

Table 15-7: AP (Access Permission) Encoding

EX	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	Outer and inner non-cacheable
1	0	0	1	Normal	Shareable	
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	-	-	
$B_1$	$B_0$	$A_1$	$A_0$	0	Normal	Not shareable	Cached memory, $B_1$ and $B_0$ define the outer policy, while the inner policy is defined by $A_1$ and $A_0$ bits. See Table 15-9.
$B_1$	$B_0$	$A_1$	$A_0$	1	Normal	Shareable	

Table 15-8: TEX, C, B, and S encoding

$A_1$	$A_0$	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

**Caution:**  $B_1$  and  $B_0$  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:

- (a) region 0 with starting address of 0x00000000 and region size of 64KB.
- (b) 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

1. True or false. All the ARM chips come with MPU
2. True or false. Upon Power-on Reset, the MPU is enabled and ready to go.
3. 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.
4. The I/O peripherals such as GPIO (general purpose I/O) region must be assigned the \_\_\_\_\_ access permission
5. In ARM, the \_\_\_\_\_ (Privileged, Unprivileged) is assigned to Operating System.

## Answers to Review Questions

### Section 15.1

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.

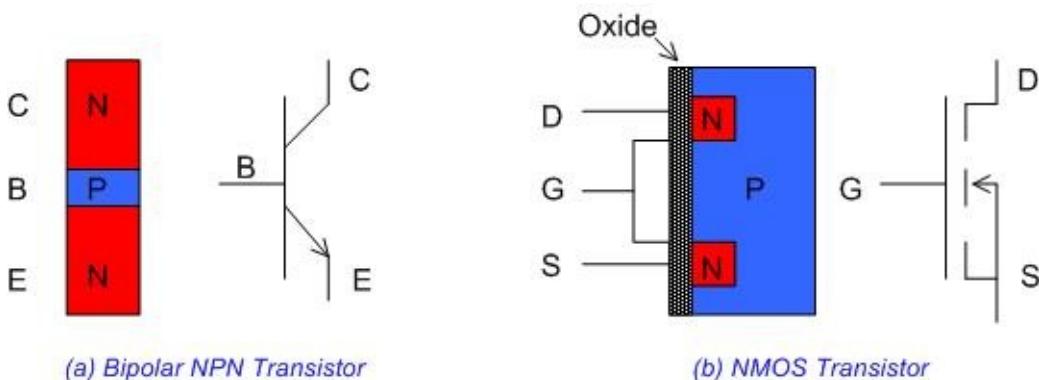


Figure A-1: Bipolar vs. MOS Transistors

In NPN-type bipolar transistors, the electron carrier leaving the emitter must

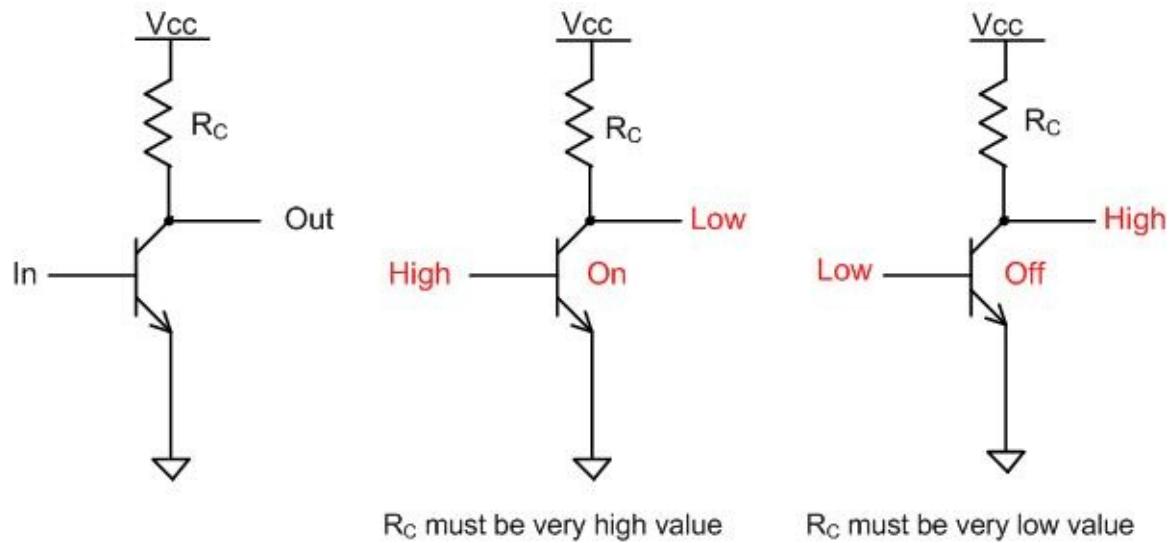
overcome two voltage barriers before it reaches the collector. One is the N-P junction of the emitter-base and the other is the P-N junction of the base-collector. The voltage barrier of the base-collector is the most difficult one for the electrons to overcome (since it is reversed biased) and it causes 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

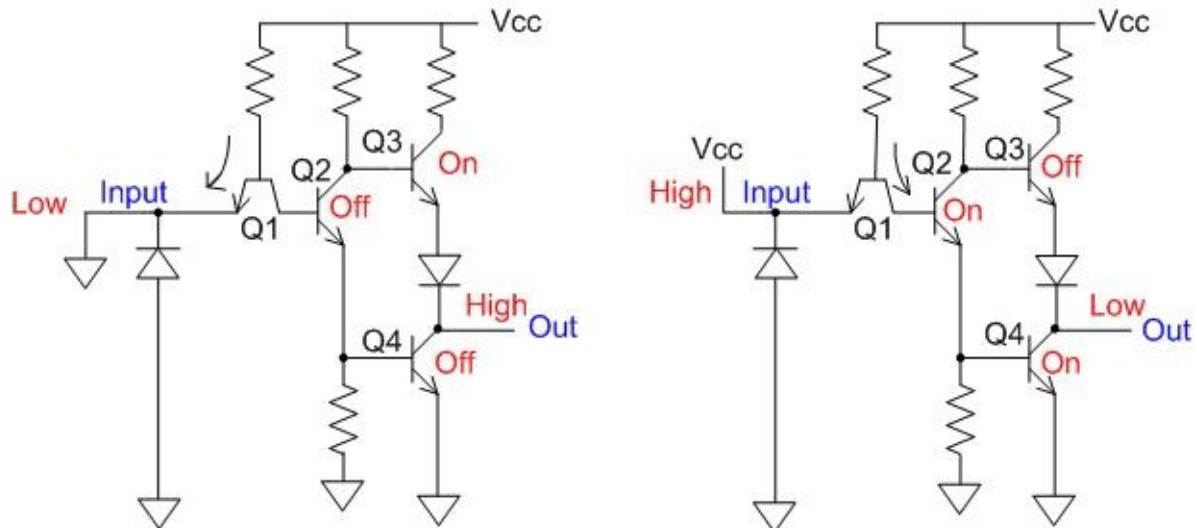
As an example of logic gates, we look at a simple inverter. In a one-transistor inverter, while the transistor plays the role of a switch, R is the pull-up resistor. See Figure A-2.



**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  $V = 5$  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 VCC. 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.

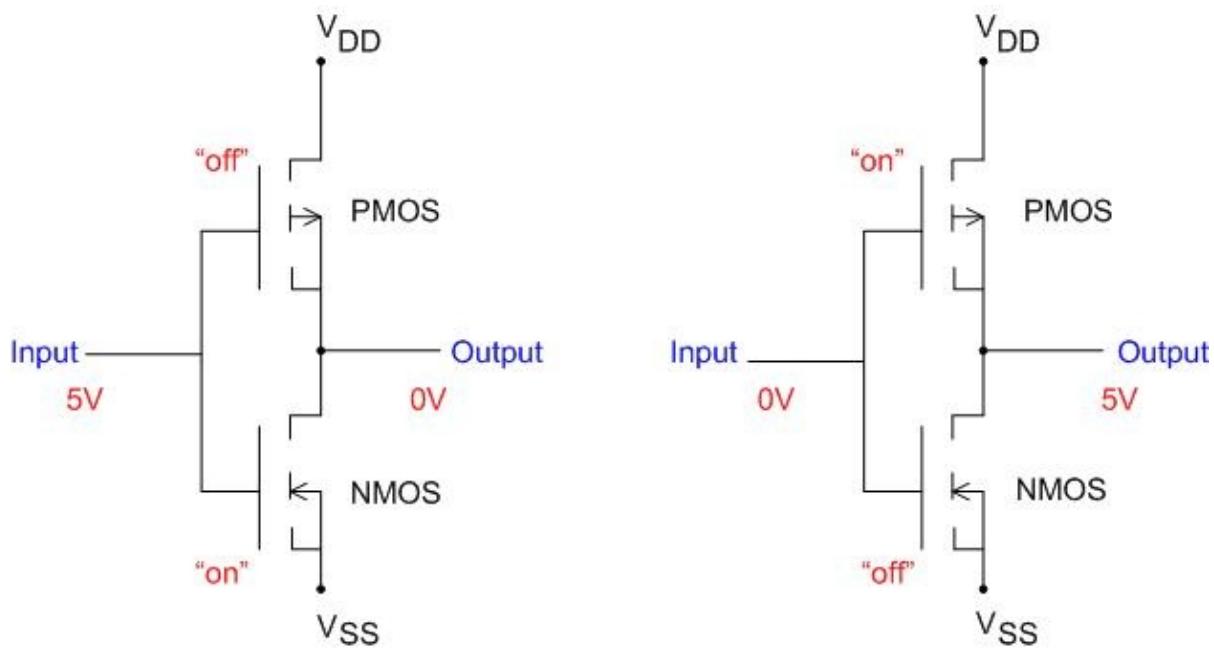
The case of a TTL inverter with totem pole output is shown in Figure A-3. In Figure A-3, Q3 plays the role of a pull-up resistor.



**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<sub>DD</sub> 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 <sub>cc</sub>	5V	5V	5V	5V
V <sub>IH</sub>	2.0V	2.0V	2.0V	3.15V
V <sub>IL</sub>	0.8V	0.8V	0.8V	1.1V
V <sub>OH</sub>	2.4V	2.7V	2.7V	3.7V
V <sub>OL</sub>	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
<b>Propagation delay</b>	10 ns	9.5 ns	4 ns	9 ns
<b>Static power dissipation (<math>f=0</math>)</b>	10 mW	2 mW	1 mW	0.0025 nW
<b>Dynamic power dissipation at <math>f = 100</math> kHz</b>	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)	Static Supply Current (mA)	High/Low Family 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

<b>AS</b>	1980	6.2	90	-15/64
<b>ALS</b>	1980	10	27	-15/64
<b>ACI/ACT</b>	1985	10	0.08	-24/24
<b>FCT</b>	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
FCTxT	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<sub>CC</sub> 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)  $\frac{f_{osc}}{f_{osc}}$

Fan-out (of high)  $\frac{f_{osc}}{f_{osc}}$

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.

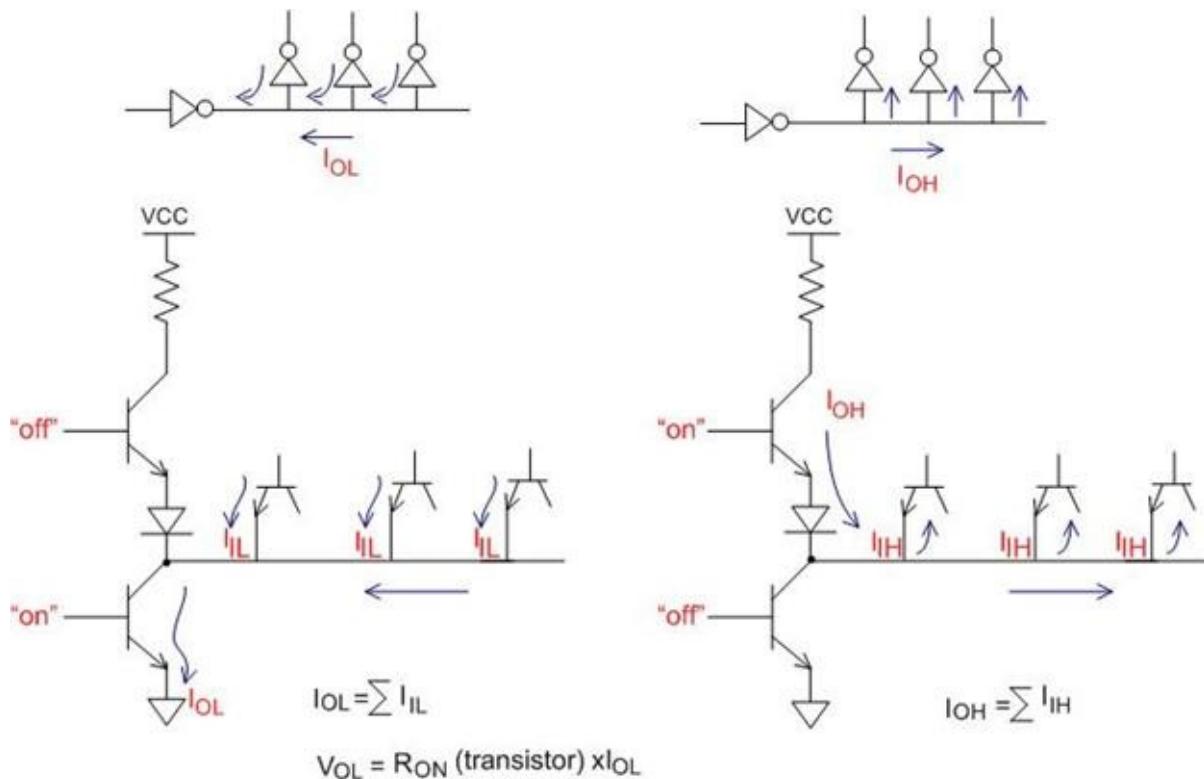


Figure A-5: Current Sinking and Sourcing in TTL

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

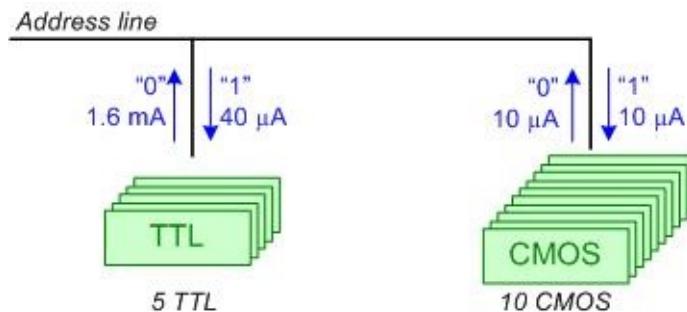
An address pin needs to drive 5 standard TTL loads in addition to 10 CMOS inputs of DRAM chips. Calculate the minimum current to drive these inputs for both logic “0” and “1”.

### Solution:

The standard load for TTL is  $I_{IH} = 40 \mu\text{A}$  and  $I_{IL} = 1.6 \text{ mA}$ , and for CMOS,  $I_{IL} = I_{IH} = 10 \mu\text{A}$ .

$$\text{minimum current for "0"} = \text{total of all } I_{IL} = 5 \times 1.6 \text{ mA} + 10 \times 10 \mu\text{A} = 8.1 \text{ mA}$$

$$\text{minimum current for "1"} = \text{total of all } I_{IH} = 5 \times 40 \mu\text{A} + 10 \times 10 \mu\text{A} = 300 \mu\text{A}$$



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  $I_{OL} = 2 \text{ mA}$ . 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)
74LS244	3	12	20	0.2
74LS254	3	12	20	0.2
<i>Note: <math>V_{OL} = 0.4 \text{ V}</math> and <math>V_{OH} = 2.4 \text{ V}</math> are assumed.</i>				

**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.

$$Q = CT \quad (\text{A-1})$$

$$Q / T = CV / T \quad (\text{A-2})$$

$$F = 1 / T \quad (\text{A-3})$$

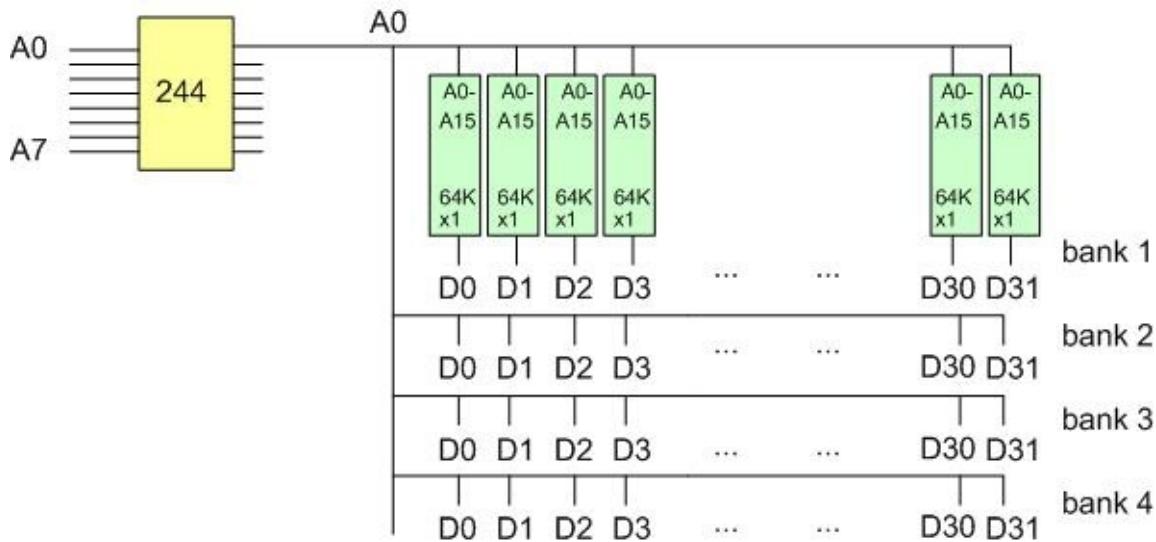
$$I = CVF \quad (\text{A-4})$$

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.

(a) delay due to capacitance derating on the address path

(b) the total address path delay for case 1

### Solution:

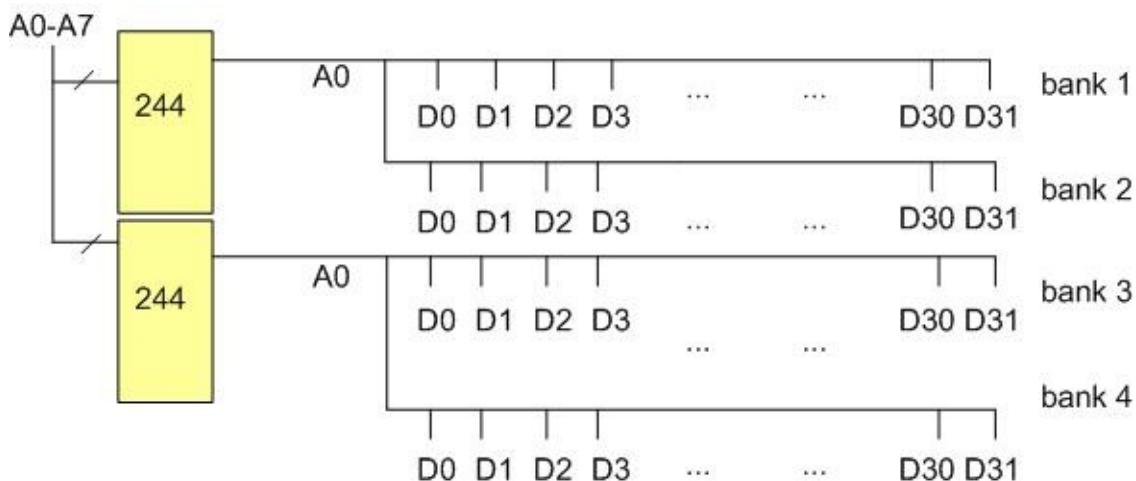
(a) Of the 320 pF capacitance seen by the 244, only 50 pF is taken care of; the rest, which is 270 ( $320 - 50 = 270$ ), causes a delay. Since there are 3 ns for each extra 100 pF, we have the following delay due to capacitance derating,  $(270/100) \times 3 \text{ ns} = 8.1 \text{ ns}$ .

(b) 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  $32 \times 5 = 160 \text{ pF}$ . In this case, we use only four 244 buffer chips, as shown in Figure A-7 and Example A-5.



*Note: the two 244 drivers for A8 - A15 are not shown*

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:

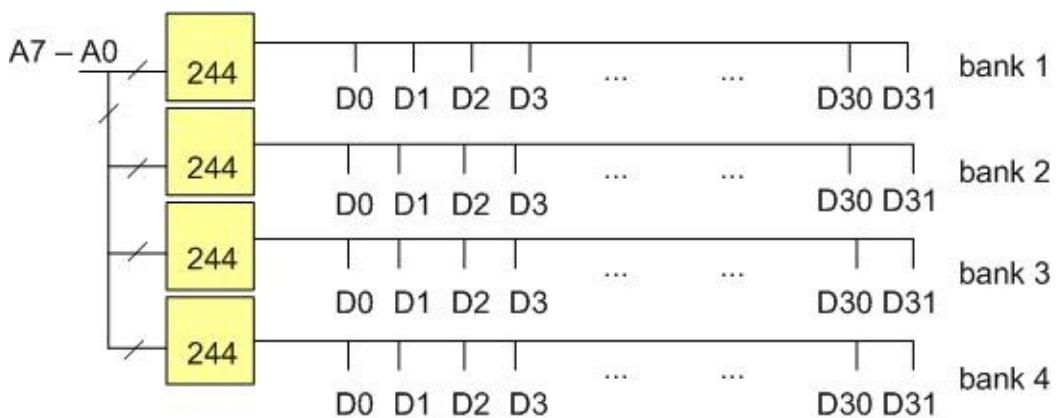
(a) Of the 160 pF capacitance seen by the 244, only 50 pF is taken care of; the rest, which is 110 pF, causes a delay. Since there are 3 ns for each extra 100 pF, we have  $(110/100) \times 3 \text{ ns} = 3.1 \text{ ns}$  delay due to capacitance derating.

(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  $4 \times 4 \times 5 = 80 \text{ pF}$ . Only 50 pF of it is taken care of by the 244, leaving 30 pF, causing a delay. See Figure A-8.



Note: A8 - A15 not shown

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:

$$P = VI = CFV^2 \quad (\text{A-5})$$

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

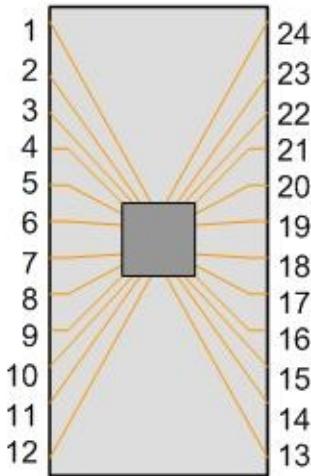


Figure A-9: Inductance and Capacitance of 24-pin DIP

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 VCC 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 small 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 VCC 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<sub>CC</sub> 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<sub>CC</sub>.

The discussion of ground bounce is also applicable to V<sub>CC</sub> when a large number of outputs changes from the low to high state and is referred to as V<sub>CC</sub> bounce. However, the effect of V<sub>CC</sub> 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.

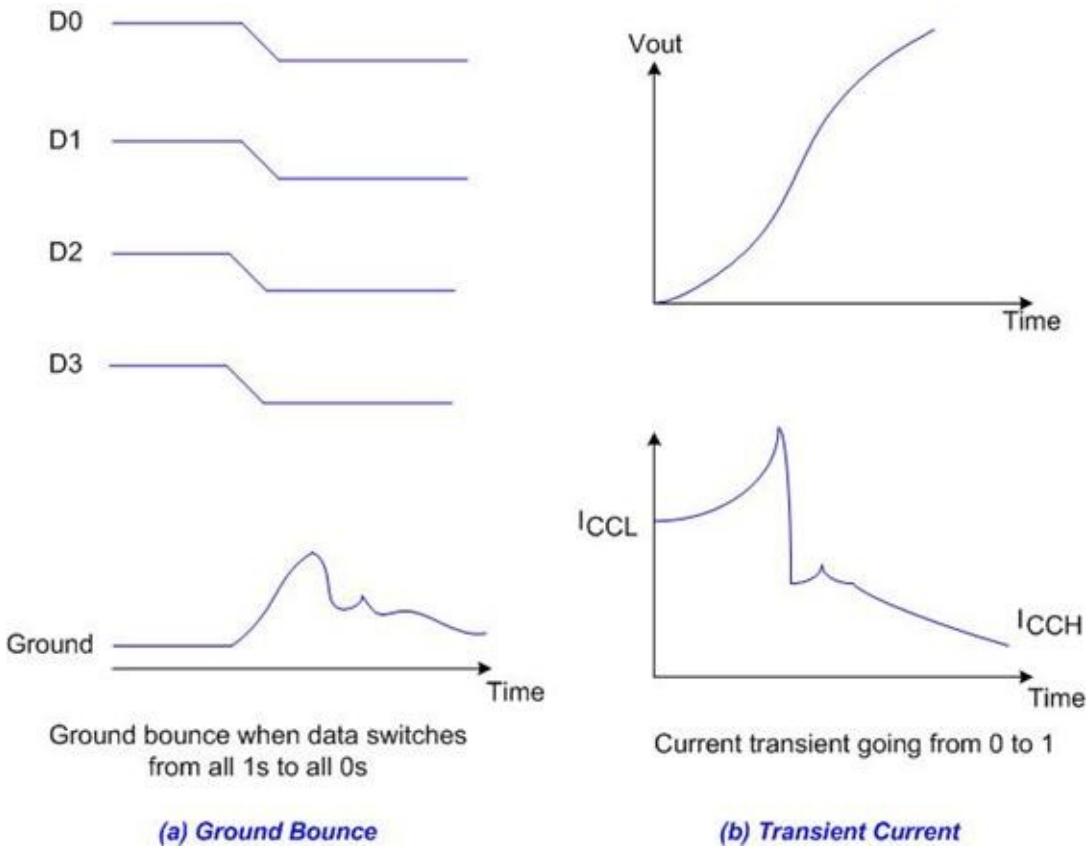


Figure A-10: (a) Ground Bounce (b) Transient Current

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.

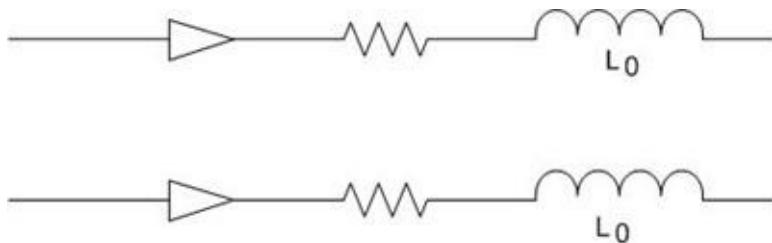


Figure A-11: Crosstalk (EMI)

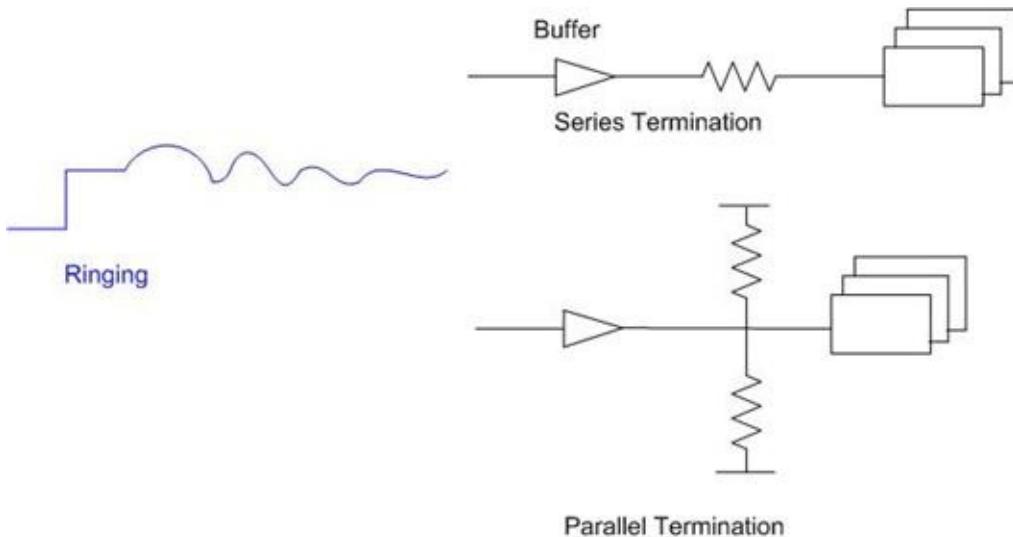
Previously, we discussed self-inductance, which is inherent in a piece of conductor. *Mutual inductance* is caused by two electric lines running parallel to each other. It is calculated as follows:

$$M = 0.002l \times \ln(2l/d) - \ln(K - 1 + d/l - d/2l)^2 \quad (\text{A-8})$$

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

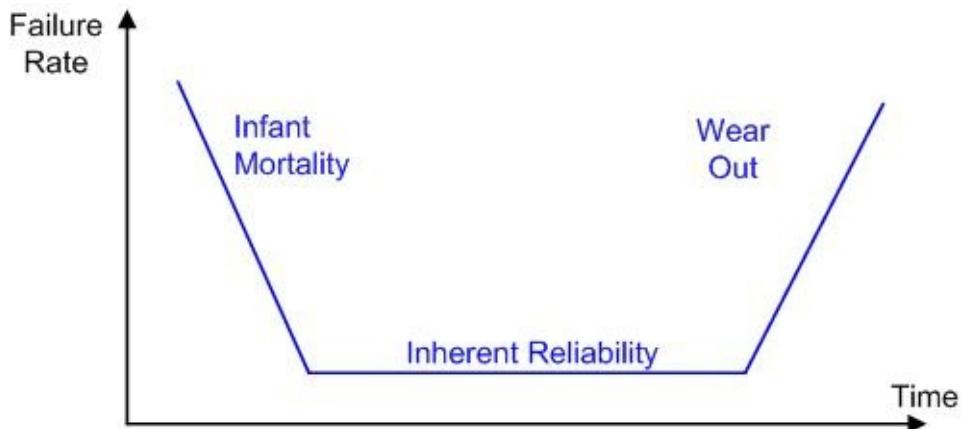
(Reprinted by permission of Integrated Device Technology, copyright IDT 1991)

**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:

$$\text{MTBF} = 1,000,000,000 \text{ hours / FIT} \quad (\text{A-9})$$

To get the MTBF rate for the system, we must divide the single-chip MTBF by the number of chips in the system.

$$\text{MTBF of system} = \text{MTBF of one chip} / \text{number of chips in system} \quad (\text{A-10})$$

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 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)

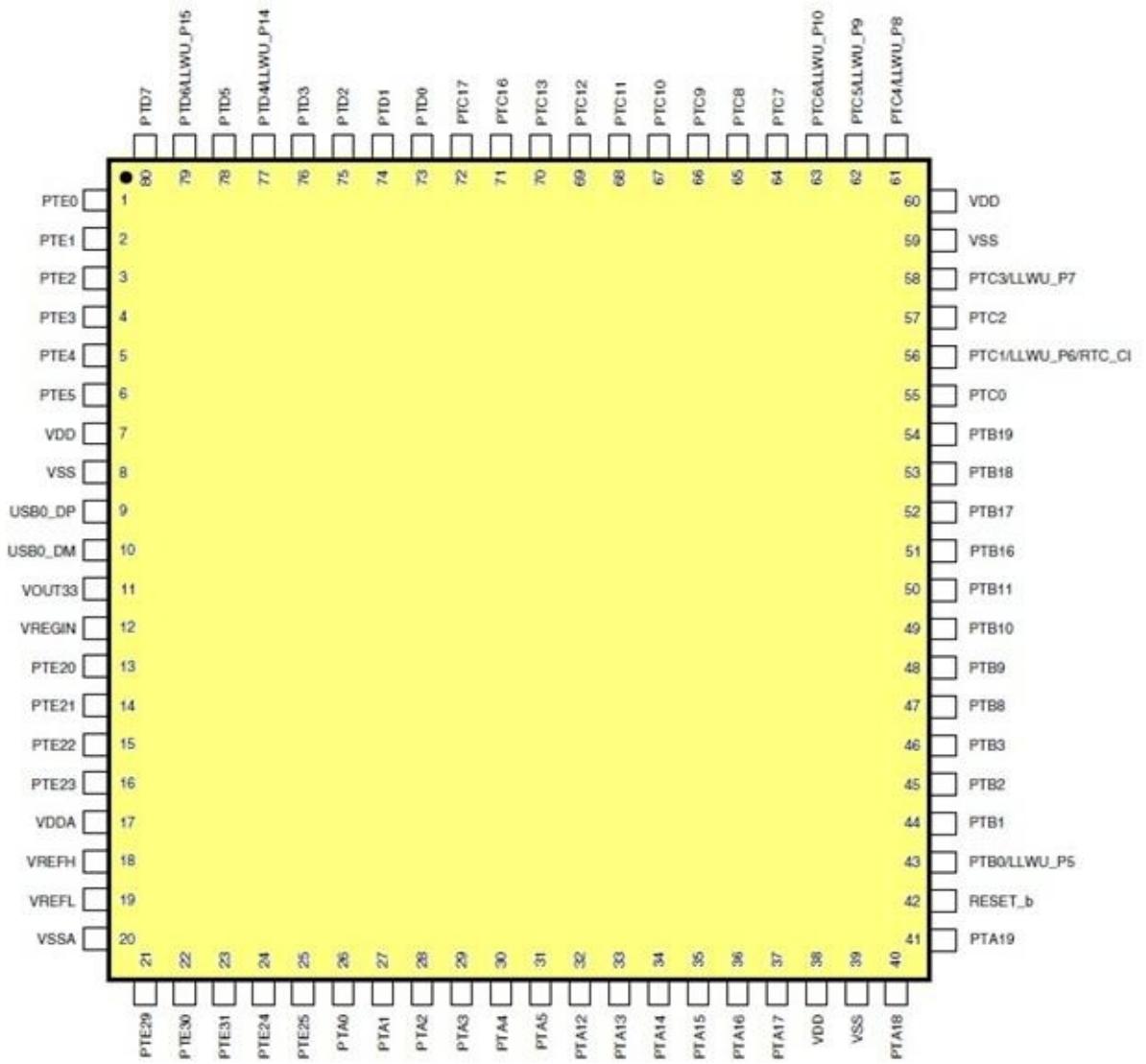


## Appendix B: KL25Z 80-pin Pinout

Pin	Pin Name	Default	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	
1	PTE0	DISABLED		PTE0		UART1_TX	RTC_CLKOUT	CMP0_OUT	I2C1_SDA	
2	PTE1	DISABLED		PTE1	SPI1_MOSI	UART1_RX		SPI1_MISO	I2C1_SCL	
3	PTE2	DISABLED		PTE2	SPI1_SCK					
4	PTE3	DISABLED		PTE3	SPI1_MISO			SPI1_MOSI		
5	PTE4	DISABLED		PTE4	SPI1_PCS0					
6	PTE5	DISABLED		PTE5						
7	VDD	VDD	VDD							
8	VSS	VSS	VSS							
9	USB0_DP	USB0_DP	USB0_DP							
10	USB0_DM	USB0_DM	USB0_DM							
11	VOUT33	VOUT33	VOUT33							
12	VREGIN	VREGIN	VREGIN							
13	PTE20	ADC0_DP0/ ADC0_SE0	ADC0_DP0/ ADC0_SE0	PTE20		TPM1_CH0	UART0_TX			
14	PTE21	ADC0_DM0/ ADC0_SE4a	ADC0_DM0/ ADC0_SE4a	PTE21		TPM1_CH1	UART0_RX			
15	PTE22	ADC0_DP3/ ADC0_SE3	ADC0_DP3/ ADC0_SE3	PTE22		TPM2_CH0	UART2_TX			
16	PTE23	ADC0_DM3/ ADC0_SE7a	ADC0_DM3/ ADC0_SE7a	PTE23		TPM2_CH1	UART2_RX			
17	VDDA	VDDA	VDDA							
18	VREFH	VREFH	VREFH							
19	VREFL	VREFL	VREFL							
20	VSSA	VSSA	VSSA							
21	PTE29	CMP0_IN5/ ADC0_SE4b				TPM0_CH2	TPM_CLKIN0			
22	PTE30	DAC0_OUT/ ADC0_SE23/ CMP0_IN4				TPM0_CH3	TPM_CLKIN1			
23	PTE31	DISABLED		PTE31		TPM0_CH4				
24	PTE24	DISABLED		PTE24		TPM0_CH0		I2C0_SCL		
25	PTE25	DISABLED		PTE25		TPM0_CH1		I2C0_SDA		
26	PTA0	SWD_CLK	TSI0_CH1	PTA0		TPM0_CH5				S
27	PTA1	DISABLED	TSI0_CH2	PTA1	UART0_RX	TPM0_CH0				

28	PTA2	DISABLED	TSI0_CH3	PTA2	UART0_TX	TPM2_CH1				
29	PTA3	SWD_DIO	TSI0_CH4	PTA3	I2C1_SCL	TPM0_CH0				S
30	PTA4	NMI_b	TSI0_CH5	PTA4	I2C1_SDA	TPM0_CH1				
31	PTA5	DISABLED		PTA5	USB_CLKIN	TPM0_CH2				
32	PTA12	DISABLED		PTA12		TPM1_CH0				
33	PTA13	DISABLED		PTA13		TPM1_CH1				
34	PTA14	DISABLED		PTA14	SPI0_PCS0	UART0_TX				
35	PTA15	DISABLED		PTA15	SPI0_SCK	UART0_RX				
36	PTA16	DISABLED		PTA16	SPI0_MOSI			SPI0_MISO		
37	PTA17	DISABLED		PTA17	SPI0_MISO			SPI0_MOSI		
38	VDD	VDD	VDD							
39	VSS	VSS	VSS							
40	PTA18	EXTAL0	EXTAL0	PTA18		UART1_RX	TPM_CLKIN0			
41	PTA19	XTAL0	XTAL0	PTA19		UART1_TX	TPM_CLKIN1		LPTMR0_ALT1	
42	RESET_b	RESET_b		PTA20						
43	PTB0/ LLWU_P5	ADC0_SE8/ TSI0_CH0	ADC0_SE8/ TSI0_CH0		I2C0_SCL	TPM1_CH0				
44	PTB1	ADC0_SE9/ TSI0_CH6	ADC0_SE9/ TSI0_CH6	PTB1	I2C0_SDA	TPM1_CH1				
45	PTB2	ADC0_SE12/ TSI0_CH7	ADC0_SE12/ TSI0_CH7	PTB2	I2C0_SCL	TPM2_CH0				
46	PTB3	ADC0_SE13/ TSI0_CH8	ADC0_SE13/ TSI0_CH8	PTB3	I2C0_SDA	TPM2_CH1				
47	PTB8	DISABLED		PTB8		EXTRG_IN				
48	PTB9	DISABLED		PTB9						
49	PTB10	DISABLED		PTB10	SPI1_PCS0					
50	PTB11	DISABLED		PTB11	SPI1_SCK					
51	PTB16	TSI0_CH9	TSI0_CH9	PTB16	SPI1_MOSI	UART0_RX	TPM_CLKIN0	SPI1_MISO		
52	PTB17	TSI0_CH10	TSI0_CH10	PTB17	SPI1_MISO	UART0_TX	TPM_CLKIN1	SPI1_MOSI		
53	PTB18	TSI0_CH11	TSI0_CH11	PTB18		TPM2_CH0				
54	PTB19	TSI0_CH12	TSI0_CH12	PTB19		TPM2_CH1				
55	PTC0	ADC0_SE14/ TSI0_CH13	ADC0_SE14/ TSI0_CH13	PTC0		EXTRG_IN		CMP0_OUT		
56	PTC1/ LLWU_P6/ RTC_CLKIN	ADC0_SE15/ TSI0_CH14	ADC0_SE15/ TSI0_CH14	PTC1/ LLWU_P6/ RTC_CLKIN	I2C1_SCL		TPM0_CH0			
		ADC0_SE11/	ADC0_SE11/							

57	PTC2	TSI0_CH15	TSI0_CH15	PTC2	I2C1_SDA		TPM0_CH1		
58	PTC3/ LLWU_P7	DISABLED		PTC3/ LLWU_P7		UART1_RX	TPM0_CH2	CLKOUT	
59	VSS	VSS	VSS						
60	VDD	VDD	VDD						
61	PTC4/ LLWU_P8	DISABLED		PTC4/ LLWU_P8	SPI0_PCS0	UART1_TX	TPM0_CH3		
62	PTC5/ LLWU_P9	DISABLED		PTC5/ LLWU_P9	SPI0_SCK	LPTMR0_ALT2			CMP0_OUT
63	PTC6/ LLWU_P10	CMP0_IN0	CMP0_IN0	PTC6/ LLWU_P10	SPI0_MOSI	EXTRG_IN		SPI0_MISO	
64	PTC7	CMP0_IN1	CMP0_IN1	PTC7	SPI0_MISO			SPI0_MOSI	
65	PTC8	CMP0_IN2	CMP0_IN2	PTC8	I2C0_SCL	TPM0_CH4			
66	PTC9	CMP0_IN3	CMP0_IN3	PTC9	I2C0_SDA	TPM0_CH5			
67	PTC10	DISABLED		PTC10	I2C1_SCL				
68	PTC11	DISABLED		PTC11	I2C1_SDA				
69	PTC12	DISABLED		PTC12			TPM_CLKN0		
70	PTC13	DISABLED		PTC13			TPM_CLKN1		
71	PTC16	DISABLED		PTC16					
72	PTC17	DISABLED		PTC17					
73	PTD0	DISABLED		PTD0	SPI0_PCS0		TPM0_CH0		
74	PTD1	ADC0_SE5b	ADC0_SE5b	PTD1	SPI0_SCK		TPM0_CH1		
75	PTD2	DISABLED		PTD2	SPI0_MOSI	UART2_RX	TPM0_CH2	SPI0_MISO	
76	PTD3	DISABLED		PTD3	SPI0_MISO	UART2_TX	TPM0_CH3	SPI0_MOSI	
77	PTD4/ LLWU_P14	DISABLED		PTD4/ LLWU_P14	SPI1_PCS0	UART2_RX	TPM0_CH4		
78	PTD5	ADC0_SE6b	ADC0_SE6b	PTD5	SPI1_SCK	UART2_TX	TPM0_CH5		
79	PTD6/ LLWU_P15	ADC0_SE7b	ADC0_SE7b	PTD6/ LLWU_P15	SPI1_MOSI	UART0_RX		SPI1_MISO	
80	PTD7	DISABLED		PTD7	SPI1_MISO	UART0_TX		SPI1_MOSI	



**Figure B-1: KL25Z 80-pin Pinout**



# Appendix C: System Clock Generation

## Multipurpose Clock Generator

The Freescale KL25Z microcontroller has a rich set of options to generate core clock, bus clock and clocks for the peripherals by the Multipurpose Clock Generator (MCG). The MCG of KL25Z supports nine different modes of operation.

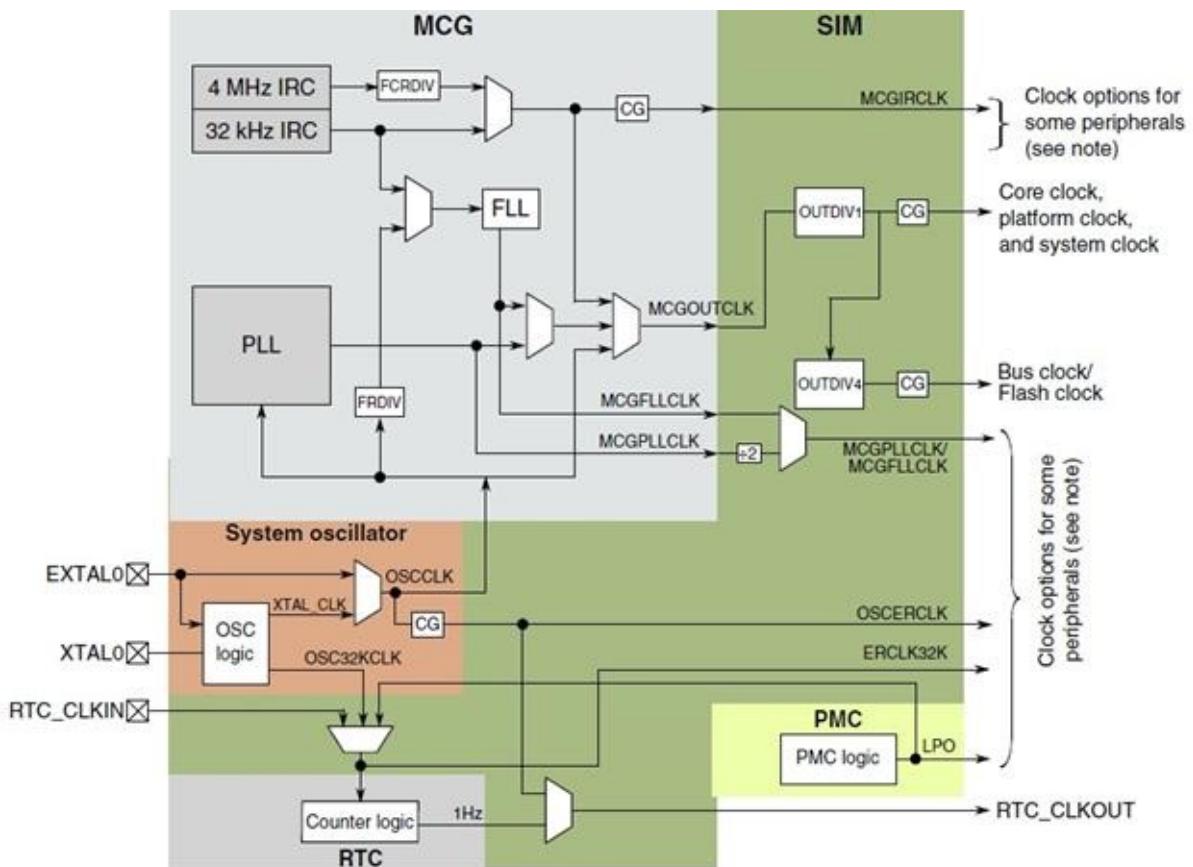


Figure C-1: Clocking Diagram (Copied from KL25Z Ref. Man.)

The MCG has two clock sources: an internal 32.768kHz clock or an external reference clock. The external reference clock could be from an external clock source or using the system oscillator in conjunction with an external crystal or a ceramic resonator. The FRDM-KL25Z comes with an 8MHz crystal connected to the system oscillator. If higher clock stability is desired, the external crystal provides better stability over the internal clock source.

The MCG also has an FLL (Frequency-Locked Loop) and a PLL (Phase-Locked Loop) to modify the frequency of the clock source. The FLL may use either internal or external clock source. The PLL can be connected to the external clock only. Although FLL and PLL provide flexibility of clock frequency, they do have some period jitter. Both FLL and PLL can be bypassed and the MCG clock output will be derived from the external reference clock.

Coming out of reset, KL25Z MCG has internal clock enabled and FLL engaged (FEI mode). The FLL factor is default to 640 so the core clock (the system clock running the ARM core including the CPU) is  $32.768\text{kHz} * 640 = 20.97\text{ MHz}$ .

In the normal run mode, the core clock may run up to 48 MHz and the bus clock 24 MHz.

## Clock Generator Programming

The MCG configuration procedures are complex and we will not attempt to cover them here.

When the Device Family Support Pack is used with Keil MDK-ARM v5.11, the startup code in system\_MKL25Z4.c file supports three MCG configurations and is default to Configuration 0.

Configuration	MCG Mode	Clock Source	Clock Modifier	Core Clock	Bus Clock
0	FEI	Internal	FLL	41.94 MHz	13.98 MHz
1	PEE	External	PLL	48.00 MHz	24.00 MHz
2	BLPE	External	bypass	8.0 MHz	8.0 MHz

Table C-1: Clock generator configurations available for FRDM-KL25Z in Keil MDK-ARM Device Family Support Pack

To select a clock configuration other than the default, you need to edit the system\_MKL25Z4.c file and change the definition of CLOCK\_SETUP.

## PLL Programming

If you need a clock frequency that is different from the available configurations, the easiest way is to modify the second configuration in PEE mode since the PLL has the most flexibility.

When in PEE mode, the external clock frequency is divided by PRDIV0 (bits 4-0 of MCG\_C5 Register) before feeding to the PLL. The VCO divider (VDIV0, bits 4-0 of MCG\_C6 Register) divides the output of VCO before feeding back to the phase detector which acts as a multiplier to the frequency. Lastly, the output of PLL is divided by OUTDIV1 (bits 31-28 of SIM\_CLKDIV1 Register) before connected to the core clock. The core clock is divided by OUTDIV4 (bits 18-16 of SIM\_CLKDIV1 Register) before it is used as the bus clock. For example, in the startup code,

- the PRDIV0 is set to 1 (divide by 2),
- the VDIV0 is set to 0 (multiply by 24),
- the OUTDIV1 is set to 1 (divide by 2), and
- the OUTDIV4 is set to 1 (divide by 2)

The external clock is running at 8 MHz so the core clock is:  $8 \text{ MHz} / 2 * 24 / 2 = 48 \text{ MHz}$ . The bus clock is  $48 \text{ MHz} / 2 = 24 \text{ MHz}$ .

For the details of the divide and multiply factors, please refer to the Reference Manual. The next example will set the core clock to 28 MHz.

- the PRDIV0 is set to 1 (divide by 2),
- the VDIV0 is set to 0x19 (multiply by 49), and
- the OUTDIV1 is set to 6 (divide by 7)

The core clock will be  $8 \text{ MHz} / 2 * 49 / 7 = 28 \text{ MHz}$ .



# References

For this book, the following references have been used:

1. [KL25 Sub-Family Reference Manual \(KL25P80M48SF0RM\)](#)
2. [FRDM-KL25Z Trainer User's Manual \(FRDMKL25ZUM\)](#)
3. [Kinetis L Series MCUs \(KLSRSRPRDSUMMAP\)](#)

The background picture of the cover is downloaded from the following website:

[http://creativity103.com/collections/Technology/slides/circuit\\_board.html](http://creativity103.com/collections/Technology/slides/circuit_board.html)