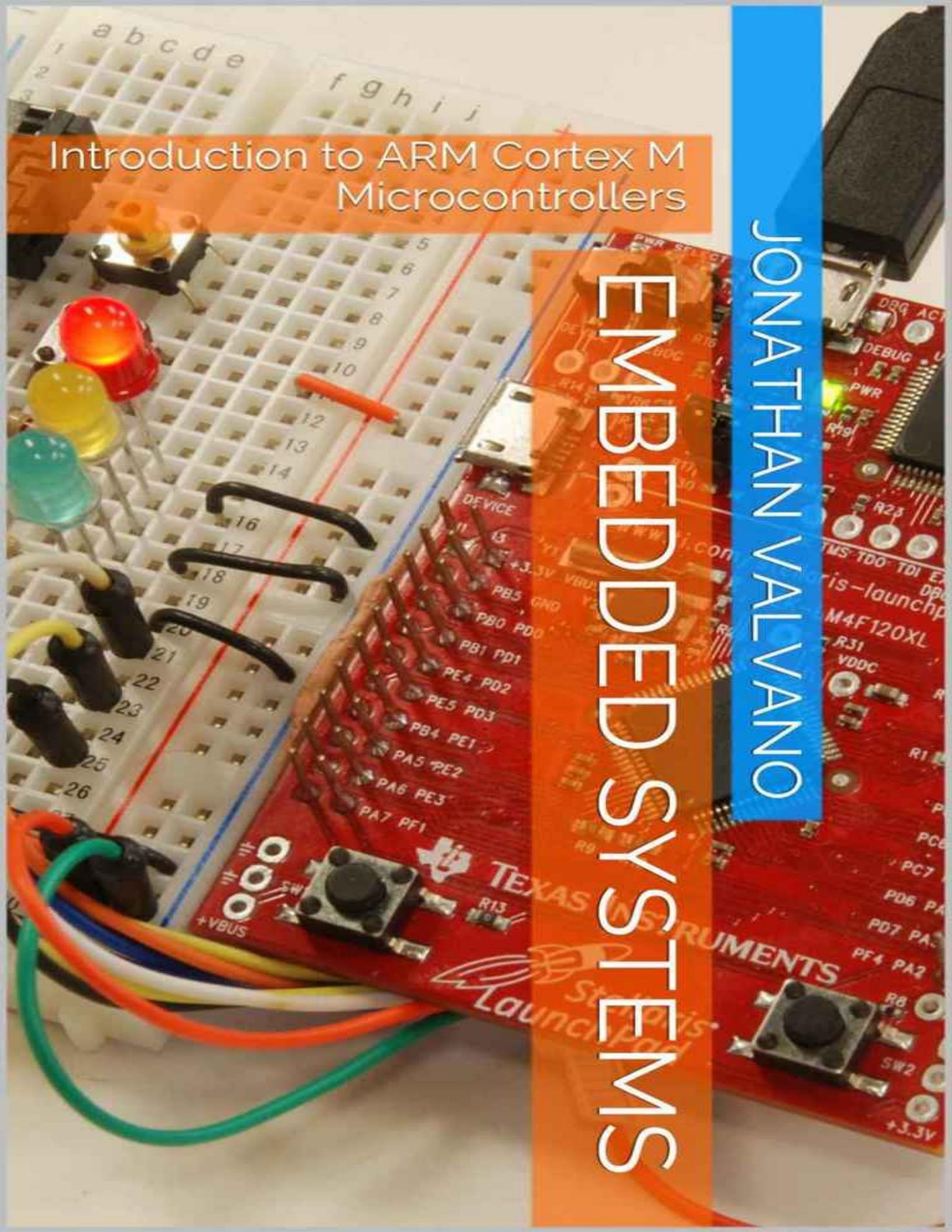


# Introduction to ARM Cortex M Microcontrollers

# EMBEDDED SYSTEMS

JONATHAN VALVANO



# EMBEDDED SYSTEMS:

---

## INTRODUCTION TO ARM®CORTEX™ -M MICROCONTROLLERS

Volume 1  
Fifth Edition  
June 2014

Jonathan W. Valvano

Fifth edition  
2nd printing  
June 2014

ARM and uVision are registered trademarks of ARM Limited.  
Cortex and Keil are trademarks of ARM Limited.  
Stellaris and Tiva are registered trademarks Texas Instruments.  
Code Composer Studio is a trademark of Texas Instruments.  
All other product or service names mentioned herein are the trademarks of their respective owners.

In order to reduce costs, this college textbook has been self-published. For more information about my classes, my research, and my books, see  
<http://users.ece.utexas.edu/~valvano/>

For corrections and comments, please contact me at: [valvano@mail.utexas.edu](mailto:valvano@mail.utexas.edu). Please cite this book as: J. W. Valvano, Embedded Systems: Introduction to ARM® Cortex™ -M Microcontrollers, Volume 1,  
<http://users.ece.utexas.edu/~valvano/>, ISBN: 978-1477508992.

Copyright © 2014 Jonathan W. Valvano

All rights reserved. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, web distribution, information networks, or information storage and retrieval, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

ISBN-13: 978-1477508992

ISBN-10: 1477508996

# **Table of Contents**

## Preface to the Fifth Edition

### Preface

### Acknowledgements

## 1. Introduction to Computers and Electronics

### 1.1. Review of Electronics

### 1.2. Binary Information Implemented with MOS transistors

### 1.3. Digital Logic

### 1.4. Digital Information stored in Memory

### 1.5. Numbers

### 1.6. Character information

### 1.7. Computer Architecture

### 1.8. Flowcharts and Structured Programming

### 1.9. Concurrent and Parallel Programming

### 1.10. Exercises

## 2. Introduction to Embedded Systems

### 2.1. Embedded Systems

### 2.2. Applications Involving Embedded Systems

### 2.3. Product Life Cycle

### 2.4. Successive Refinement

### 2.5. Quality Design

#### 2.5.1. Quantitative Performance Measurements

#### 2.5.2. Qualitative Performance Measurements

#### 2.5.3. Attitude

2.6. Debugging Theory

2.7. Switch and LED Interfaces

2.8. Introduction to C

2.9. Exercises

3. Introduction to the ARM □ Cortex □ -M Processor

3.1. Cortex □ -M Architecture

3.1.1. Registers

3.1.2. Reset

3.1.3. Memory

3.1.4. Operating Modes

3.2. The Software Development Process

3.3. ARM Cortex-M Assembly Language

3.3.1. Syntax

3.3.2. Addressing Modes and Operands

3.3.3. Memory Access Instructions

3.3.4. Logical Operations

3.3.5. Shift Operations

3.3.6. Arithmetic Operations

3.3.7. Stack

3.3.8. Functions and Control Flow

3.3.9. Assembler Directives

3.3.10. First Example Project

3.4. Simplified Machine Language Execution

3.5. CISC versus RISC

3.6. Details Not Covered in this Book

3.7. Exercises

4. Introduction to Input/Output

[4.1. Texas Instruments Microcontroller I/O pins](#)

[4.1.1. Texas Instruments LM3S1968 I/O pins](#)

[4.1.2. Texas Instruments TM4C123 LaunchPad I/O pins](#)

[4.1.3. Texas Instruments TM4C1294 Connected LaunchPad I/O pins](#)

[4.2. Basic Concepts of Input and Output Ports](#)

[4.2.1. I/O Programming and the Direction Register](#)

[4.2.2. Switch Inputs and LED Outputs](#)

[4.3. Phase-Lock-Loop](#)

[4.4. SysTick Timer](#)

[4.5. Standard I/O Driver and the printf Function](#)

[4.6. Debugging monitor using an LED](#)

[4.7. Performance Debugging](#)

[4.7.1. Instrumentation](#)

[4.7.2. Measurement of Dynamic Efficiency](#)

[4.8. Exercises](#)

[4.9. Lab Assignments](#)

[5. Modular Programming](#)

[5.1. C Keywords and Punctuation](#)

[5.2. Modular Design using Abstraction](#)

[5.2.1. Definition and Goals](#)

[5.2.2. Functions, Procedures, Methods, and Subroutines](#)

[5.2.3. Dividing a Software Task into Modules](#)

[5.2.4. How to Draw a Call Graph](#)

[5.2.5. How to Draw a Data Flow Graph](#)

[5.2.6. Top-down versus Bottom-up Design](#)

[5.3. Making Decisions](#)

## 5.3.1. Conditional Branch Instructions

## 5.3.2. Conditional if-then Statements

## 5.3.3. switch Statements

## 5.3.4. While Loops

## 5.3.5. Do-while Loops

## 5.3.6. For Loops

## 5.4. \*Assembly Macros

## 5.5. \*Recursion

## 5.6. Writing Quality Software

### 5.6.1. Style Guidelines

### 5.6.2. Comments

### 5.6.3. Inappropriate I/O and Portability

## 5.7. How Assemblers Work

## 5.8. Functional debugging

### 5.8.1. Stabilization

### 5.8.2. Single Stepping

### 5.8.3. Breakpoints with Filtering

### 5.8.4. Instrumentation: Print Statements

### 5.8.5. Desk checking

## 5.9. Exercises

## 5.10. Lab Assignments

# 6. Pointers and Data Structures

## 6.1. Indexed Addressing and Pointers

## 6.2. Arrays

## 6.3. Strings

## 6.4. Structures

## 6.5. Finite State Machines with Linked Structures

## 6.5.1. Abstraction

## 6.5.2. Moore Finite State Machines

## 6.5.3. Mealy Finite State Machines

## 6.6. \*Dynamically Allocated Data Structures

### 6.6.1. \*Fixed Block Memory Manager

### 6.6.2. \*Linked List FIFO

## 6.7. Matrices and Graphics

## 6.8. \*Tables

## 6.9. Functional Debugging

### 6.9.1. Instrumentation: Dump into Array without Filtering

### 6.9.2. Instrumentation: Dump into Array with Filtering.

## 6.10. Exercises

## 6.11. Lab Assignments

# 7. Variables, Numbers, and Parameter Passing

## 7.1. Local versus global

## 7.2. Stack rules

## 7.3. Local variables allocated on the stack

## 7.4. Stack frames

## 7.5. Parameter Passing

### 7.5.1. Parameter Passing in C

### 7.5.2. Parameter Passing in Assembly Language

### 7.5.3. C Compiler Implementation of Local and Global Variables

## 7.6. Fixed-point Numbers

## 7.7. Conversions

## 7.8. \*IEEE Floating-point numbers

## 7.9. Exercises

## 7.10. Lab Assignments

## 8. Serial and Parallel Port Interfacing

8.1. General Introduction to Interfacing

8.2. Universal Asynchronous Receiver Transmitter (UART)

**8.2.1. Asynchronous Communication**

**8.2.2. LM3S/TM4C UART Details**

**8.2.3. UART Device Driver**

8.3. Synchronous Serial Interface, SSI

8.4. Nokia 5110 Graphics LCD Interface

8.5. Scanned Keyboards

8.6. Binary actuators

**8.6.1. Interface**

**8.6.2. Electromagnetic and Solid State Relays**

**8.6.3. Solenoids**

8.7. \*Pulse-width modulation

8.8. \*Stepper motors

8.9. Exercises

8.10. Lab Assignments

## 9. Interrupt Programming and Real-time Systems

9.1. I/O Synchronization

9.2. Interrupt Concepts

9.3. Interthread Communication and Synchronization

9.4. NVIC on the ARM Cortex-M Processor

9.5. Edge-triggered Interrupts

9.6. SysTick Periodic Interrupts

9.7. Timer Periodic Interrupts

9.8. Hardware debugging tools

## 9.9. Profiling

### 9.9.1 Profiling using a software dump to study execution pattern

### 9.9.2. Profiling using an Output Port

### 9.9.3. \*Thread Profile

## 9.10. Exercises

## 9.11. Lab Assignments

## 10. Analog I/O Interfacing

### 10.1. Approximating continuous signals in the digital domain

### 10.2. Digital to Analog Conversion

### 10.3. Music Generation

### 10.4. Analog to Digital Conversion

#### 10.4.1. LM3S/TM4C ADC details

#### 10.4.2. ADC Resolution

### 10.5. Real-time data acquisition

## 10.6. Exercises

## 10.7. Lab Assignments

## 11. Communication Systems

### 11.1. Introduction

### 11.2. Reentrant Programming and Critical Sections

### 11.3. Producer-Consumer using a FIFO Queue

#### 11.3.1. Basic Principles of the FIFO Queue

#### 11.3.2. FIFO Queue Analysis

#### 11.3.3. FIFO Queue Implementation

#### 11.3.4. Double Buffer

### 11.4. Serial port interface using interrupt synchronization

### 11.5. \*Distributed Systems.

[11.6. Exercises](#)

[11.7. Lab Assignments](#)

[11.8. Best Practices](#)

[Appendix 1. Glossary](#)

[Appendix 2. Solutions to Checkpoints](#)

[Appendix 3. How to Convert Projects from Keil to CCS](#)

[Appendix 4. Assembly Reference](#)

[Index](#)

# Preface to the Fifth Edition

This fifth edition includes the new TM4C1294-based LaunchPad. Most of the code in the book is specific for the TM4C123-based LaunchPad. However, the book website includes corresponding example projects for the LM3S811, LM3S1968, LM4F120, and TM4C1294, which are ARM® Cortex™-M microcontrollers from Texas Instruments. There are now two low-cost development platforms called Tiva LaunchPad. The EK-TM4C123GXL LaunchPad retails for \$12.99, and the EK-TM4C1294XL Connected LaunchPad retails for \$19.99. The various LM3S, LM4F and TM4C microcontrollers are quite similar, so this book along with the example code on the web can be used for any of these microcontrollers. Compared to the TM4C123, the new TM4C1294 microcontroller runs faster, has more RAM, has more ROM, includes Ethernet, and has more I/O pins. This fifth edition switches the syntax from C to the industry-standard C99.



# Preface

Embedded systems are a ubiquitous component of our everyday lives. We interact with hundreds of tiny computers every day that are embedded into our houses, our cars, our toys, and our work. As our world has become more complex, so have the capabilities of the microcontrollers embedded into our devices. The ARM ® Cortex™-M family represents a new class of microcontrollers much more powerful than the devices available ten years ago. The purpose of this book is to present the design methodology to train young engineers to understand the basic building blocks that comprise devices like a cell phone, an MP3 player, a pacemaker, antilock brakes, and an engine controller.

This book is the first in a series of three books that teach the fundamentals of embedded systems as applied to the ARM ® Cortex™-M family of microcontrollers. This first book is an introduction to computers and interfacing focusing on assembly language and C programming. The second book Embedded Systems: Real-Time Interfacing to ARM Cortex-M Microcontrollers focuses on hardware/software interfacing and the design of embedded systems. The third book Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers is an advanced book focusing on operating systems, high-speed interfacing, control systems, and robotics. The third volume could also be used for professionals wishing to design or deploy a real-time operating system onto an ARM platform. This first book is an introductory book that could be used at the college level with little or no prerequisites.

An embedded system is a system that performs a specific task and has a computer embedded inside. A system is comprised of components and interfaces connected together for a common purpose. This book is an introduction to embedded systems. Specific topics include microcontrollers, fixed-point numbers, the design of software in assembly language and C, elementary data structures, programming input/output including interrupts, analog to digital conversion, digital to analog conversion.

In general, the area of embedded systems is an important and growing discipline within electrical and computer engineering. In the past, the educational market of embedded systems has been dominated by simple microcontrollers like the PIC, the 9S12, and the 8051. This is because of their market share, low cost, and historical dominance. However, as problems become more complex, so must the systems that solve them. A number of embedded system paradigms must shift in order to accommodate this growth in complexity. First, the number of calculations per second will increase from millions/sec to billions/sec. Similarly, the number of lines of software code will also increase from thousands to millions. Thirdly, systems will involve multiple microcontrollers supporting many simultaneous operations. Lastly, the need for system verification will continue to grow as these systems are deployed into safety critical applications. These changes are more than a simple growth in size and bandwidth. These systems must employ parallel programming, high-speed synchronization, real-time operating systems, fault tolerant design, priority interrupt handling, and networking. Consequently, it will be important to provide our students with these types of design experiences. The ARM platform is both low cost and provides the high-performance features required in future embedded systems. In addition, the ARM market share is large and will continue to grow. Furthermore, students trained on the ARM will be equipped to design systems across the complete spectrum from simple to complex. The purpose of writing these three books at this time is to bring engineering education into the 21<sup>st</sup> century.

This book employs many approaches to learning. It will not include an exhaustive recapitulation of the information in data sheets. First, it begins with basic fundamentals, which allows the reader to solve new problems with new technology. Second, the book presents many detailed design examples. These examples illustrate the process of design. There are multiple structural components that assist learning. Checkpoints, with answers in the back, are short easy to answer questions providing immediate feedback while reading. Simple homework, with answers to the odd questions on the web, provides more detailed learning opportunities. The book includes an index and a glossary so that information can be searched. The most important learning experiences in a class like this are of course the laboratories. Each chapter has suggested lab assignments. More detailed lab descriptions are available on the web. Specifically for this volume, look at the lab assignments for EE319K. For Volume 2, refer to the EE445L labs, and for Volume 3, look at the lab assignments for EE445M/EE380L.6.

There is a web site accompanying this book <http://users.ece.utexas.edu/~valvano/arm>. Posted here are ARM Keil™ uVision® and Texas Instruments Code Composer Studio™ projects for each of the example programs in the book. You will also find data sheets and Excel spreadsheets relevant to the material in this book.

The book will cover embedded systems for ARM ® Cortex™-M microcontrollers with specific details on the TM4C123 and TM4C1294. The web site includes corresponding examples for the LM3S811, LM3S1968, and LM3S8962. In these books the terms **LM3S** and **TM4C** will refer to families of microcontrollers with the Texas Instruments Stellaris ® and Tiva ® lines. Although the solutions are specific for the **LM3S** and **TM4C** families, it will be possible to use these books for other ARM derivatives.

# Acknowledgements

I owe a wonderful debt of gratitude to Daniel Valvano. He wrote and tested most of the software examples found in these books. Secondly, he created and maintains the example web site, <http://users.ece.utexas.edu/~valvano/arm>. Lastly, he meticulously proofread this manuscript.

Many shared experiences contributed to the development of this book. First I would like to acknowledge the many excellent teaching assistants I have had the pleasure of working with. Some of these hard-working, underpaid warriors include Pankaj Bishnoi, Rajeev Sethia, Adson da Rocha, Bao Hua, Raj Randeri, Santosh Jodh, Naresh Bhavaraju, Ashutosh Kulkarni, Bryan Stiles, V. Krishnamurthy, Paul Johnson, Craig Kochis, Sean Askew, George Panayi, Jeehyun Kim, Vikram Godbole, Andres Zambrano, Ann Meyer, Hyunjin Shin, Anand Rajan, Anil Kottam, Chia-ling Wei, Jignesh Shah, Icaro Santos, David Altman, Nachiket Kharalkar, Robin Tsang, Byung Geun Jun, John Porterfield, Daniel Fernandez, Deepak Panwar, Jacob Egner, Sandy Hermawan, Usman Tariq, Sterling Wei, Seil Oh, Antonius Keddis, Lev Shuhatovich, Glen Rhodes, Geoffrey Luke, Karthik Sankar, Tim Van Ruitenbeek, Raffaele Cetrulo, Harshad Desai, Justin Capogna, Arindam Goswami, Jungho Jo, Mehmet Basoglu, Kathryn Loeffler, Evgeni Krimer, Nachiappan Valliappan, Razik Ahmed, Sundeep Korrapati, Peter Garatoni, Manan Kathuria, Jae Hong Min, Pratyusha Nidamaluri, Dayo Lawal, Aditya Srikanth, Kurt Fellows, James Beecham, Austin Blackstone, Brandon Carson, Kin Hong Mok, Omar Baca, Sam Oyetunji, Zack Lalanne, Nathan Quang Minh Thai, Paul Fagen, Zhuoran Zhao, Sparsh Singhai, Saugata Bhattacharyya, Chinmaya Dattathri, Emily Ledbetter, Kevin Gilbert, Siavash Kamali, Yen-Kai Huang, Michael Xing, Katherine Olin, Mitchell Crooks, Prachi Gupta, Mark Meserve, Sourabh Shirhatti, Dylan Zika, Kelsey Ball, Greg Cerna, Sabine Francis, Ahmad El Youssef, and Wooseok Lee. These teaching assistants have contributed greatly to the contents of this book and particularly to its laboratory assignments. Since 1981, I estimate I have taught embedded systems to over 5000 students. Spring 2014, Professor Yerraballi and I taught a massive open online class (MOOC) based on this book. We had over 40,000 students register, and over 5,000 students finished at least one lab on the hardware platform. We plan to rerun this MOOC Spring 2015. My students have recharged my energy each semester with their enthusiasm, dedication, and quest for knowledge. I have decided not to acknowledge them all individually. However, they know I feel privileged to have had this opportunity.

Next, I appreciate the patience and expertise of my fellow faculty members here at the University of Texas at Austin. From a personal perspective Dr. John Pearce provided much needed encouragement and support throughout my career. In addition, Drs. John Cogdell, John Pearce, and Francis Bostick helped me with analog circuit design. The book and accompanying software include many finite state machines derived from the digital logic examples explained to me by Dr. Charles Roth. The educational content presented in this book is result of the combined efforts of the entire teaching staff of EE319K: Drs. Ramesh Yerraballi, Mattan Erez, Andreas Gerstlauer, Nina Telang, William Bard and I. This team has created an educationally rich lab course that is both engaging and achievable for the freshman engineer. Each time we teach EE319K, we create a capstone design experience centered on a class competition. You can see descriptions and photos of our class design competitions at <http://users.ece.utexas.edu/~valvano/>.

I have a special appreciation for all those who reviewed the first edition. Joe Bungo passed chapters around to fellows at ARM. In particular, the suggestions and corrections from Chris Shore and Drew Barbier were totally awesome. Cathy Wicks and Larissa Swanland from Texas Instruments supported these books with money, development kits, and chips. Bill Bard and Ramesh Yerraballi gave valuable feedback on how to make this book an effective teaching tool for freshmen. Austin Blackstone has been a constant contributor to these books. In particular, he created and debugged the Code Composer Studio™ versions of the example programs posted on the web.

Sincerely, I appreciate the valuable lessons of character and commitment taught to me by my parents and grandparents. I recall how hard my parents and grandparents worked to make the world a better place for the next generation. Most significantly, I acknowledge the love, patience and support of my wife, Barbara, and my children, Ben Dan and Liz. In particular, Dan designed and tested most of the LM3S and TM4C software presented in this book.

By the grace of God, I am truly the happiest man on the planet, because I am surrounded by these fine people.

Jonathan W. Valvano

The true engineering experience occurs not with your eyes and ears, but rather with your fingers and elbows. In other words, engineering education does not happen by listening in class or reading a book; rather it happens by designing under the watchful eyes of a patient mentor. So, go build something today, then show it to someone you respect!

Good luck

# 1. Introduction to Computers and Electronics

## Chapter 1 objectives are to:

- Present brief reviews of electronics and Ohm's Law
- Present a brief review of computer fundamentals
- Review digital logic and information
- Introduce software design using flowcharts

The overall objective of this book is to teach the fundamentals of embedded systems. It is an effective approach to learn new techniques by doing them. But, the dilemma in learning a laboratory-based topic like embedded systems is that there is a tremendous volume of details that first must be learned before hardware and software systems can be designed. The approach taken in this book is to learn by doing. One of the advantages of a bottom-up approach to learning is that the student begins by mastering simple concepts. Once the student truly understands simple concepts, he or she can then embark on the creative process of design, which involves putting the pieces together to create a more complex system. True creativity is needed to solve complex problems using effective combinations of simple components. Embedded systems afford an effective platform to teach new engineers how to program for three reasons. First, there is no operating system. Thus, in a bottom-up fashion the student can see, write, and understand all software running on a system that actually does something. Second, embedded systems involve input/output that is easy for the student to touch, hear, and see. Third, embedded systems are employed in many every-day products, motivating students by showing them how electrical and computer engineering processes can be applied in the real world. Rather than introduce the voluminous details in an encyclopedic fashion, the book is organized by basic concepts, and the details are introduced as they are needed. We will start with simple systems and progressively add complexity. The overriding themes for Chapters 1 and 2 will be to present the organizational framework with which embedded systems will be designed. Chapter 3 explains how the computer works. Chapter 4 is an introduction to I/O. Chapters 5, 6, and 7 present the details of software development on an embedded system. Interfacing I/O devices to build embedded systems is presented in Chapters 8, 9, 10, and 11.

# 1.1. Review of Electronics

Most readers of this book will have had some prior training in electronics. However, this brief section will provide an overview of the electronics needed to understand electric circuits in this book. **Current (I)** is defined as the movement of electrons. In particular, 1 ampere (A) of current is  $6.241 \times 10^{18}$  electrons per second, or one coulomb per second. Current is directional and measured at one point as the number of electrons travelling per second. Current has an amplitude and a direction. Because electrons are negatively charged, if the electrons are moving to the left, we define current as flowing to the right. **Voltage (V)** is an electrical term representing the potential difference between two points. The units of voltage are volts (V), and it is always measured as a difference. Voltage is the electromotive force or potential to produce current. We will see two types of conducting media: a **wire** and a **resistor**. Wires, made from copper, will allow current to freely flow, but forcing current to flow through a resistor will require energy. The electrical property of a resistor is resistance in ohms ( $\Omega$ ). Ideally, a wire is simply a resistor with a resistance of  $0 \Omega$ . The basic relation between voltage, current, and resistance for a resistor is known as **Ohm's Law**, which can be written three ways:

$$V = I * R$$

$$I = V / R$$

$$R = V / I$$

$$\text{Voltage} = \text{Current} * \text{Resistance}$$

$$\text{Current} = \text{Voltage} / \text{Resistance}$$

$$\text{Resistance} = \text{Voltage} / \text{Current}$$

The left side of Figure 1.1 shows a circuit element representation of a resistor, of resistance R. Whenever we define voltage, we must clearly specify the two points across which the potential is defined. Typically we label voltages with + and -, defining the voltage V as the potential to produce current from the + down to the -. When defining current we draw an arrow signifying the direction of the current. If the voltage V is positive, then the current I will be positive meaning the current is down in this figure. However, because electrons have negative charge, the electrons are actually flowing up. According to the passive sign convention, we define positive current as the direction of the flow of positive charge (or the opposite direction of the flow of negative charge). The middle of Figure 1.1 shows a circuit with a  $1 \text{ k}\Omega$  resistor placed across a 3.7 V battery. According to Ohm's Law, 3.7 mA of current will flow down across the resistor. In this circuit, current flows clockwise from the + terminal of the battery, down across the resistor, and then back to the - terminal of the battery.

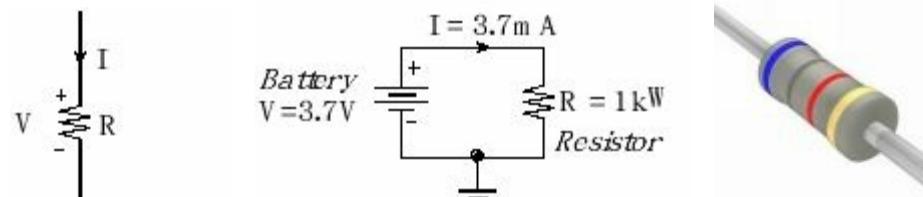


Figure 1.1. The voltage and current definitions; a circuit with a battery; and a drawing of a resistor.

**Checkpoint 1.1:** There is 1 V across a resistor, and 2 mA is flowing. What is the resistance?

**Checkpoint 1.2:** There is 5V across a  $100\ \Omega$  resistor. How much current is flowing?

**Checkpoint 1.3:** What happens if you place a wire directly from + terminal to the –terminal of a battery?

There are two analogous physical scenarios that might help you understand the concept of voltage, current, and resistance. The first analogy is flowing water through a pipe. We place a large reservoir of water in a tower, connect the water through a pipe, and attach a faucet at the bottom of the pipe, see Figure 1.2. In this case pressure is analogous to voltage, water flow is analogous to current, and fluid resistance of the faucet is analogous to electrical resistance. Notice that water pressure is defined as the potential to cause water to flow, and it is measured between two places. Pressure has a polarity, and water flow has a direction. If the faucet is turned all the way off, its resistance is infinite, and no water flows. If the faucet is turned all the way on, its resistance is not zero, but some finite amount. As we turn the faucet we are varying the fluid resistance. The fluid resistance will determine the amount of flow:

$$\text{Flow} = \text{Pressure}/\text{Resistance}$$

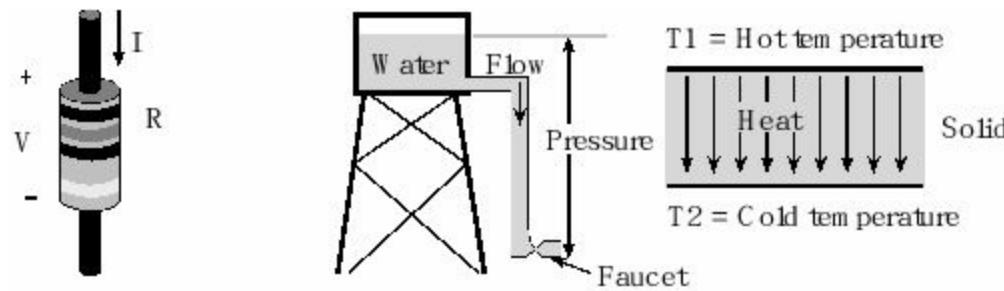


Figure 1.2. Three analogous physical systems demonstrating Ohm's Law.

**Checkpoint 1.4:** If pressure is measured in Newtons/m<sup>2</sup> (Pascal) and flow measured in m<sup>3</sup>/sec, what are the units of fluid resistance?

A second analogy is heat flow across a solid. If we generate a temperature gradient across a solid, heat will flow from the hot side to the cold side (right side of Figure 1.2). This solid could be a glass window on a house or the wall of your coffee cup. In this case temperature gradient is analogous to voltage, heat flow is analogous to current, and thermal resistance of the solid is analogous to electrical resistance. Notice that potential is defined as the temperature difference between two places. Heat flow also has a direction. If the coffee cup is made from metal, its thermal resistance is low, lots of heat will flow, and the coffee cools off quickly. If the coffee cup is made of Styrofoam, its resistance is high, little heat will flow, and the coffee remains hot for a long time. The thermal resistance of the material the amount of flow:

$$\text{Flow} = (T_1 - T_2)/\text{Resistance}$$

**Checkpoint 1.5:** If heat flow is measured in watts (Joules/sec) and temperature measured in °C, what are the units of thermal resistance?

The R-value of insulation put in the walls and ceiling of a house is usually given in units per square area, e.g.,  $\text{m}^2 \cdot ^\circ\text{C}/\text{W}$ . The amount of heat flow across a wall is:

$$\text{Flow} = \text{Area} * (\text{T}_1 - \text{T}_2)/\text{R-value}$$

Another important parameter occurring when current flows through a resistor is **power**. The power (P in watts) dissipated in a resistor can be calculated from voltage (V in volts), current (I in amps), and resistance (R in ohms). Interestingly, although voltage has a polarity (+ and -) and current has a direction, power has neither a polarity nor a direction.

$$P = V * I$$

$$\text{Power} = \text{Voltage} * \text{Current}$$

$$P = V^2 / R$$

$$\text{Power} = \text{Voltage}^2 / \text{Resistance}$$

$$P = I^2 * R$$

$$\text{Power} = \text{Current}^2 * \text{Resistance}$$

**Checkpoint 1.6:** There is 1 V across a resistor, and 2 mA is flowing. How much power is being dissipated?

**Checkpoint 1.7:** There is 5V across a  $100 \Omega$  resistor. How much power is being dissipated?

The **energy** (E in joules) stored in a battery can be calculated from voltage (V in volts), current (I in amps), and time (t in seconds). Energy has neither polarity nor direction.

$$E = V * I * t$$

$$\text{Energy} = \text{Voltage} * \text{Current} * \text{time}$$

$$E = P * t$$

$$\text{Energy} = \text{Power} * \text{time}$$

A switch is an element used to modify the behavior of the circuit (Figure 1.3). If the switch is pressed, its resistance is 0, and current can flow across the switch. If the switch is not pressed, its resistance is infinite, and no current will flow. In reality, the ON-resistance of a switch is less than  $0.1\Omega$ , but this is so close to zero, we can assume the ideal value of 0 in most cases. Similarly, the OFF-resistance is actually greater than  $100\text{M}\Omega$ , but this is so close to infinity that we can again assume the ideal value of infinity. The classic electrical circuit involves a battery, a light bulb (modeled in this circuit as a  $100\Omega$  resistor), and a switch.

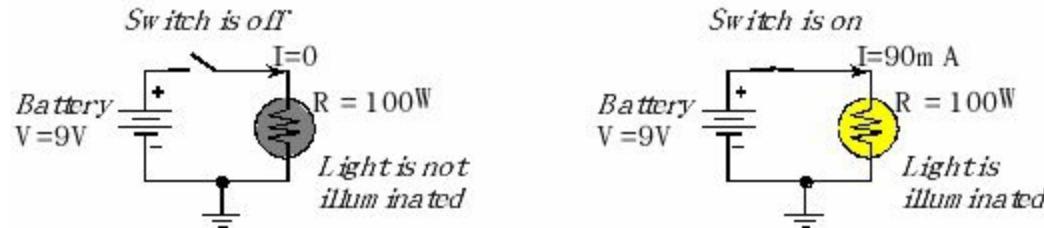


Figure 1.3. When the switch is open, no current can flow, and the bulb does not emit light. When the switch is closed, 90 mA of current will flow, and the bulb emits light.

**Checkpoint 1.8:** If the switch is on, how much power is being dissipated in the bulb?

There are a few basic rules that allow us to solve for voltages and currents within a circuit comprised with batteries, switches, and resistors.

Current always flows in a loop. In Figure 1.3 when the switch is pressed, current flows out of the + side of battery, across the switch, through the light and back to the – side of the battery. When there is no loop, no current can flow. In Figure 1.3 when the switch is off, the loop is broken, and no current will flow.

Kirchhoff's Voltage Law (KVL). The sum of the voltages around the loop is zero. For a battery, we label the + and – sides exactly the way the battery is labeled. For a resistor, we label the current arrow and the voltage + – like the left side of Figure 1.1. The important step is the direction of the current arrow must match the polarity of the corresponding voltage. It is common practice to draw arrows in the direction the currents actually flow, so the voltages will be positive. However, sometimes we don't know which way the current will flow, so we can just guess. If we happen to guess wrong, both the current and voltage will calculate to be negative and the correct behavior will still be obtained. We can think of the switch as a resistor of either 0 or infinity resistance, so it too can be labeled with a current arrow and a voltage polarity. Figure 1.4 shows the light circuit redrawn to show voltages and currents. As we are going around a circle and pass from + to –, we add that voltage. However, if we pass across an element from – to + we subtract that voltage.

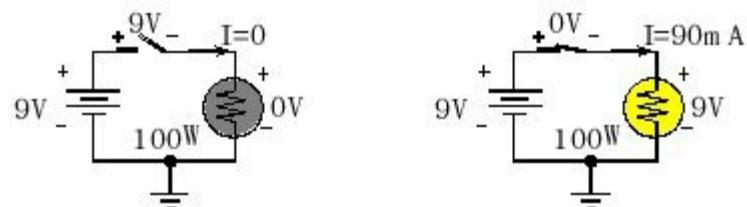


Figure 1.4. The voltages around a loop will sum to zero (KVL).

Kirchhoff's Current Law (KCL). The sum of the currents into a node equal the sum of the currents leaving a node as shown in Figure 1.5. To solve circuits using KCL and KVL, the current arrow across a resistor goes from the + voltage to the – voltage. Conversely, the current arrow across a battery goes from the – voltage to the + voltage. This is the same thing as saying current comes out of the battery's + terminal and into the battery's – terminal. At Node A, there is one incoming current and one outgoing current. This is a simple but important fact that  $I_1 = I_2$ . At Node B, there is one incoming current and two outgoing currents. Therefore,  $I_3 = I_4 + I_5$ . There are two currents into Node C and two currents out of Node C; thus,  $I_6 + I_7 = I_8 + I_9$ .

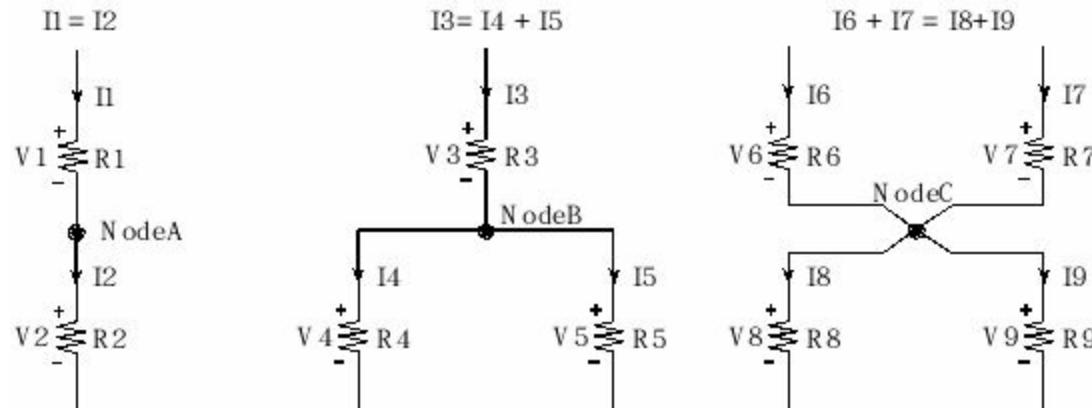


Figure 1.5. The sum of the currents into a node will equal the sum of the currents leaving (KCL).

**Observation:** If at all possible, draw the circuit so current flows down across the resistors and switches. As a secondary rule have currents go left to right across resistors and switches.

Series resistance. If resistor R<sub>1</sub> is in series with resistor R<sub>2</sub>, this combination behaves like one resistor with a value equal to R<sub>1</sub>+R<sub>2</sub>. See Figure 1.6. This means if replace the two series resistors in a circuit with one resistor at R= R<sub>1</sub>+R<sub>2</sub>, the behavior will be the same. The V equals V<sub>1</sub>+V<sub>2</sub>. By KCL, the currents through the two resistors are the same. These two facts can be used to derive the **voltage divider rule**

$$V_2 = I \cdot R_2 = (V/R) \cdot R_2 = V \cdot R_2 / (R_1 + R_2)$$

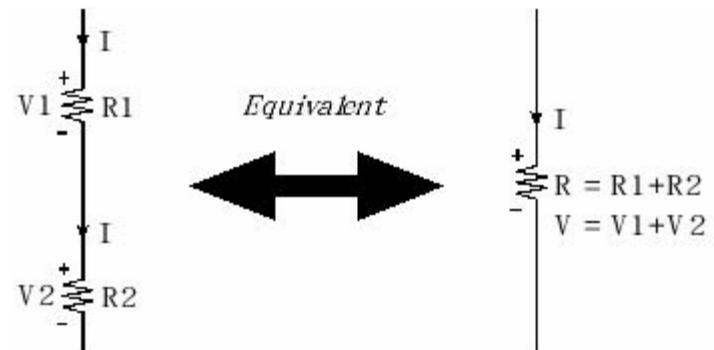


Figure 1.6. The series combination of two resistors, R<sub>1</sub> R<sub>2</sub>, is equivalent to one resistor at R<sub>1</sub>+R<sub>2</sub>.

**Checkpoint 1.9:** Using Figure 1.6, assume I is 1mA, R<sub>1</sub> is 1k Ω and R<sub>2</sub> is 2k Ω , what is V?

**Checkpoint 1.10:** Using Figure 1.6, assume V is 6V, R<sub>1</sub> is 1k Ω and R<sub>2</sub> is 2k Ω , what is V<sub>2</sub>?

Parallel resistance. If resistor R<sub>1</sub> is in parallel with resistor R<sub>2</sub>, this combination behaves like one resistor with a value equal to

$$R = \frac{R_1 \cdot R_2}{R_1 + R_2} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}$$

See Figure 1.7. This means we can replace the two parallel resistors in a circuit with one resistor at R= R<sub>1</sub>\*R<sub>2</sub>/(R<sub>1</sub>+R<sub>2</sub>). The voltages across R<sub>1</sub> and R<sub>2</sub> will be the same because of KVL. Due to KCL, I=I<sub>1</sub>+I<sub>2</sub>. These facts can be used to derive the **current divider rule**

$$I_1 = V/R_1 = (I \cdot R)/R_1 = I \cdot (R_1 \cdot R_2 / (R_1 + R_2)) / R_1 = I \cdot R_2 / (R_1 + R_2)$$

$$I_2 = V/R_2 = (I \cdot R)/R_2 = I \cdot (R_1 \cdot R_2 / (R_1 + R_2)) / R_2 = I \cdot R_1 / (R_1 + R_2)$$

$$I = I_1 + I_2$$

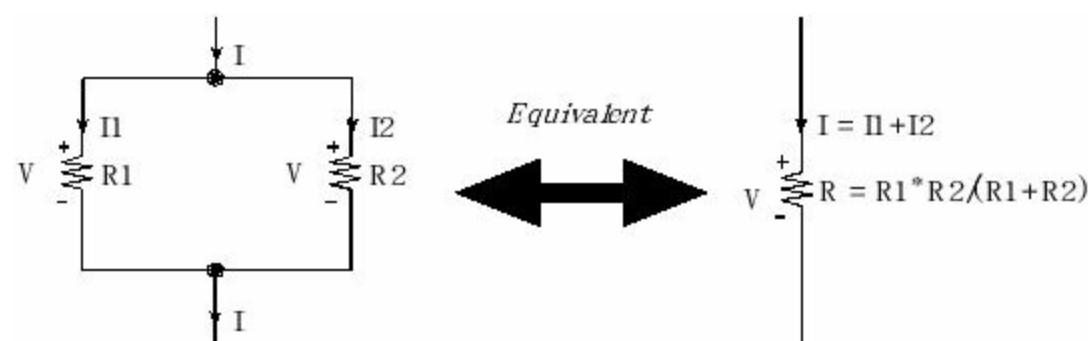


Figure 1.7. The parallel combination of two resistors, R<sub>1</sub> R<sub>2</sub>, is equivalent to one resistor at R<sub>1</sub>\*R<sub>2</sub>/(R<sub>1</sub>+R<sub>2</sub>).

**Checkpoint 1.11:** Using Figure 1.7, assume I is 1mA, R<sub>1</sub> is 2k Ω and R<sub>2</sub> is 3k Ω , what is V?

**Checkpoint 1.12:** Using Figure 1.7, assume V is 6V, R1 is 2k  $\Omega$  and R2 is 3k  $\Omega$  , what is I2?

# 1.2. Binary Information Implemented with MOS transistors

Information is stored on the computer in binary form. A binary **bit** can exist in one of two possible states. In **positive logic**, the presence of a voltage is called the ‘1’, true, asserted, or high state. The absence of a voltage is called the ‘0’, false, not asserted, or low state. Figure 1.8 shows the output of a typical complementary metal oxide semiconductor (CMOS) circuit. The left side shows the condition with a true bit at the output, and the right side shows a false at the output. The output of each digital circuit consists of a p-type transistor “on top of” an n-type transistor. In digital circuits, each transistor is essentially on or off. If the transistor is **on**, it is equivalent to a short circuit between its two output pins. Conversely, if the transistor is **off**, it is equivalent to an open circuit between its outputs pins.

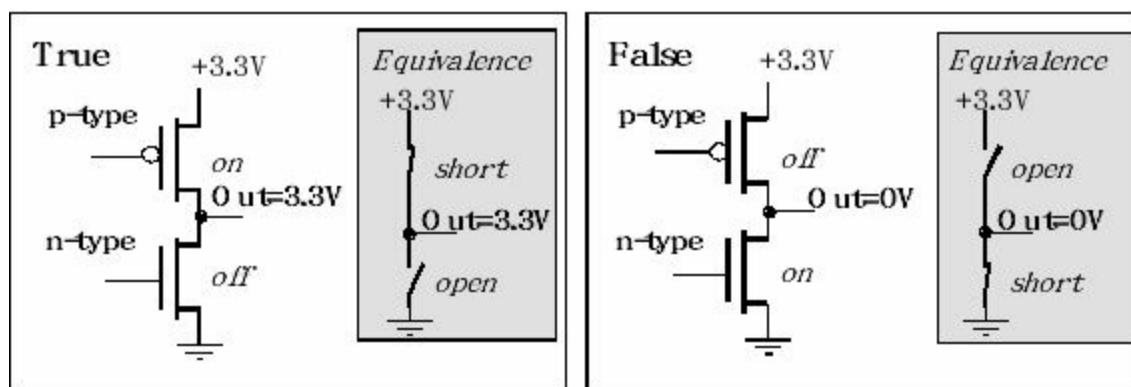


Figure 1.8. A binary bit at the output is true if a voltage is present and false if the voltage is 0.

Every family of digital logic is a little different, but on the Cortex-M microcontrollers from TI powered with 3.3 V supply, a voltage between 2 and 5 V is considered high, and a voltage between 0 and 1.3 V is considered low, as drawn in Figure 1.9. Separating the two regions by 0.7 V allows digital logic to operate reliably at very high speeds. The design of transistor-level digital circuits is beyond the scope of this book. However, it is important to know that digital data exist as binary bits and encoded as high and low voltages.

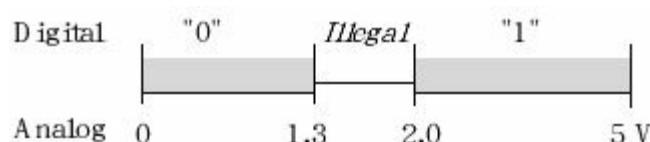


Figure 1.9. Mapping between analog voltage and the corresponding digital meaning on the LM3S/TM4C.

If the information we wish to store exists in more than two states, we use multiple bits. A collection of 2 bits has 4 possible states (00, 01, 10, and 11). A collection of 3 bits has 8 possible states (000, 001, 010, 011, 100, 101, 110, and 111). In general, a collection of  $n$  bits has  $2^n$  states. For example, a

**byte** contains eight bits, and is built by grouping eight binary bits into one object, as shown in Figure 1.10. Another name for a collection of eight bits is **octet** (octo is Latin and Greek meaning 8.) Information can take many forms, e.g., numbers, logical states, text, instructions, sounds, or images. What the bits mean depends on how the information is organized and more importantly how it is used. This figure shows one byte in the state representing the binary number 01100111. Again, the output voltage 3.3V means true or 1, and the output voltage of 0V means false or 0.

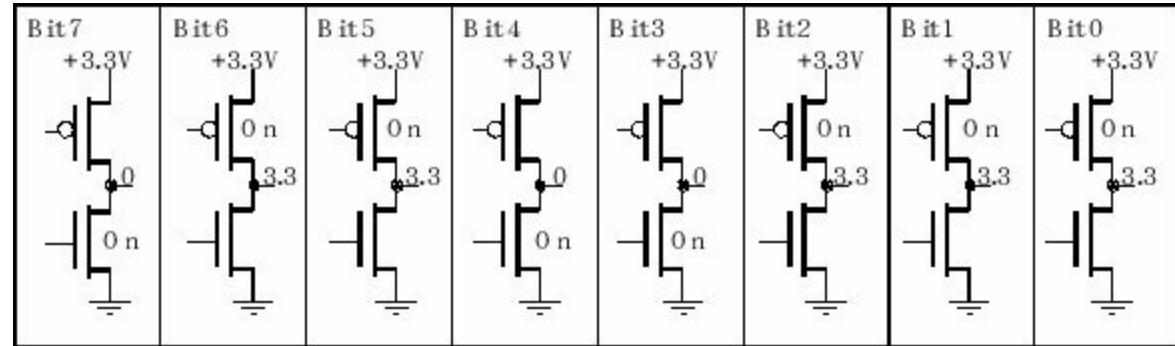


Figure 1.10. A byte is comprised of 8 bits, in this case representing the binary number 01100111.

# 1.3. Digital Logic

In order to understand how the computer works, we will need some understanding of digital logic. Transistors made with metal oxide semiconductors are called MOS. In the digital world MOS transistors can be thought of as voltage controlled switches. Circuits made with p-type and n-type MOS transistors are called complementary metal oxide semiconductors or CMOS. The 74HC04 is a high-speed CMOS NOT gate, as shown in Figure 1.11. There are just a few rules one needs to know for understanding how CMOS transistor-level circuits work. Each transistor acts like a switch between its source and drain pins. In general, current can flow from source to drain across an active p-type transistor, and no current will flow if the switch is open. From a first approximation, we can assume no current flows into or out of the gate. For a p-type transistor, the switch will be closed (transistor active) if its gate is low. A p-type transistor will be off (its switch is open) if its gate is high.

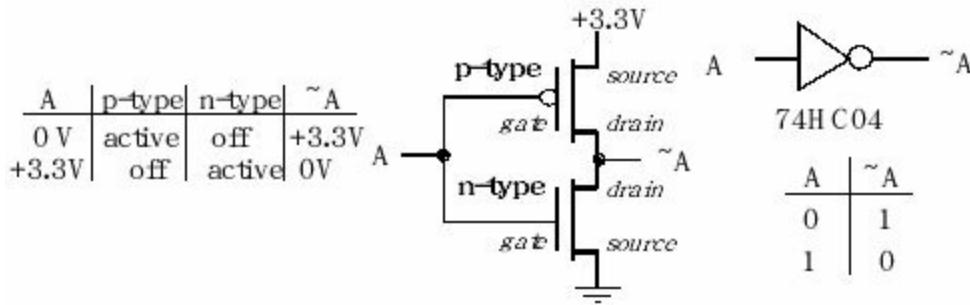


Figure 1.11. CMOS implementation of a NOT gate.

The gate on the n-type works in a complementary fashion, hence the name complementary metal oxide semiconductor. For an n-type transistor, the switch will be closed (transistor active) if its gate is high. An n-type transistor will be off (its switch is open) if its gate is low. Therefore, consider the two possibilities for the circuit in Figure 1.11. If the input A is high (+3.3V), then the p-type transistor is off and the n-type transistor is active. The closed switch across the source-drain of the n-type transistor will make the output low (0V). Conversely, if A is low (0V), then p-type transistor is active and the n-type transistor is off. The closed switch across the source-drain of the p-type transistor will make the output high (+3.3V).

The AND, OR, EOR digital logic takes two inputs and produces one output; see Figure 1.12 and Table 1.1. We can understand the operation of the AND gate by observing the behavior of its six transistors. If both inputs A and B are high, both T3 and T4 will be active. Furthermore, if A and B are both high, T1 and T2 will be off. In this case, the signal labeled  $\sim(A \& B)$  will be low because the T3–T4 switch combination will short this signal to ground. If A is low, T1 will be active and T3 off. Similarly, if B is low, T2 will be active and T4 off. Therefore if either A is low or if B is low, the signal labeled  $\sim(A \& B)$  will be high because one or both of the T1, T2 switches will short this signal to +3.3V. Transistors T5 and T6 create a logical complement, converting the signal  $\sim(A \& B)$  into the desired result of A&B. We can use the **and** operation to extract, or **mask**, individual bits from a value.

A	B	AND	NAND	OR	NOR	EOR	Ex NOR
---	---	-----	------	----	-----	-----	--------

0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1
Symbol	A&B	$\sim$ (A&B)	A B	$\sim(A B)$	$A^B$	$\sim(A^B)$	

**Table 1.1. Two-input one-output logical operations.**

We can understand the operation of the OR gate by observing the behavior of its six transistors. If both inputs A and B are low, both T1 and T2 will be active. Furthermore, if A and B are both low, T3 and T4 will be off. In this case, the signal labeled  $\sim(A|B)$  will be high because the T1–T2 switch combination will short this signal to +3.3V. If A is high, T3 will be active and T1 off. Similarly, if B is high, T4 will be active and T2 off. Therefore if either A is high or if B is high, the signal labeled  $\sim(A|B)$  will be low because one or both of the T3, T4 switches will short this signal to ground. Transistors T5 and T6 create a logical complement, converting the signal  $\sim(A|B)$  into the desired result of A|B. We use the or operation to set individual bits.

When writing software we will have two kinds of logic operations. When operating on numbers (collection of bits) we will perform **logic** operations bit by bit. In other words, the operation is applied independently on each bit. In C, the logic operator for AND is **&**. For example, if number A is 01100111 and number B is 11110000 then

$$\begin{array}{ll} A = & 01100111 \\ B = & \underline{11110000} \\ A \& B & 01100000 \end{array}$$

The other type of logic operation occurs when operating on **Boolean** values. In C, the condition false is represented by the value 0, and true is any nonzero value. In this case, if the Boolean A is 01100111 and B is 11110000 then both A and B are true. The standard value for true is the value 1. In C, the Boolean operator for AND is **&&**. Performing Boolean operation yields

$$\begin{array}{ll} A = & 01100111 \\ B = & \underline{11110000} \\ A \&\& B & 1 \end{array}$$

In C, the logic operator for OR is **|**. The operation is applied independently on each bit . E.g.,

$$\begin{array}{ll} A = & 01100111 \\ B = & \underline{11110000} \\ A | B & 11110111 \end{array}$$

In C, the Boolean operator for OR is `||`. Performing Boolean operation of **true** OR **true** yields **true**. Although 1 is the standard value for a true, any nonzero value is considered as true.

$$\begin{array}{ll} A = & 01100111 \\ B = & \underline{11110000} \\ A \parallel B & 1 \end{array}$$

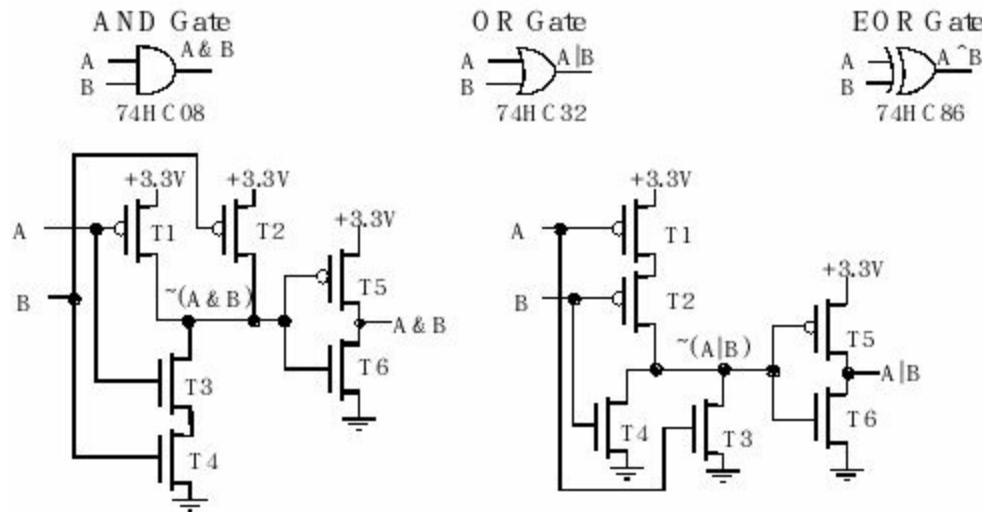


Figure 1.12. Logical operations can be implemented with discrete transistors or digital gates.

**Checkpoint 1.13:** Using just the 74HC gates shown in Figure 1.12, design a three-input one-output logic function (called a three-input AND) such that the output is high if and only if all inputs are high. In other words, the output is low if one or more inputs are low.

**Checkpoint 1.14:** Using just the 74HC gates shown in Figure 1.12, design a three-input one-output logic function (called a three-input OR) such that the output is high if one or more inputs are high. In other words, the output is low if and only if all inputs are low.

**Observation:** In C, `(5&2)` will be zero because  $(0101\&0010)=0000$ . However, `(5&&2)` will be true, because 5 means true, 2 means true, and true&&true is true..

Other convenient logical operators are shown as digital gates in Figure 1.13. The **NAND** operation is defined by an AND followed by a NOT. If you compare the transistor-level circuits in Figures 1.12 and 1.13, it would be more precise to say AND is defined as a NAND followed by a NOT. Similarly, the OR operation is a **NOR** followed by a NOT. The **exclusive NOR** operation implements the bit-wise equals operation.

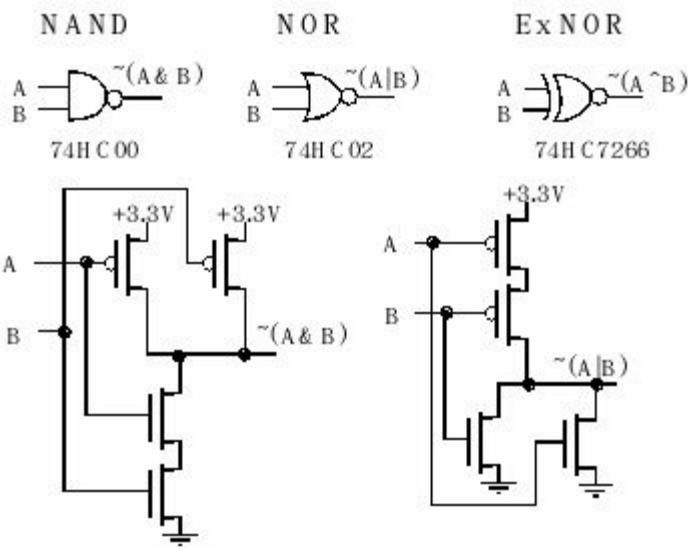


Figure 1.13. Other logical operations can also be implemented with MOS transistors.

**Boolean Algebra** is the mathematical framework for designing digital logic. Some fundamental laws of Boolean Algebra are listed below.

$$A \& B = B \& A$$

Commutative Law

$$A | B = B | A$$

Commutative Law

$$(A \& B) \& C = A \& (B \& C)$$

Associative Law

$$(A | B) | C = A | (B | C)$$

Associative Law

$$(A | B) \& C = (A \& C) | (B \& C)$$

Distributive Law

$$(A \& B) | C = (A | C) \& (B | C)$$

Distributive Law

$$A \& 0 = 0$$

Identity of 0

$$A | 0 = A$$

Identity of 0

$$A \& 1 = A$$

Identity of 1

$$A | 1 = 1$$

Identity of 1

$$A | A = A$$

Property of OR

$$A | (\sim A) = 1$$

Property of OR

$$A \& A = A$$

Property of AND

$$A \& (\sim A) = 0$$

Property of AND

$$\sim(\sim A) = A$$

Inverse

$$\sim(A | B) = (\sim A) \& (\sim B)$$

De Morgan's Theorem

$$\sim(A \& B) = (\sim A) | (\sim B)$$

De Morgan's Theorem

When multiple operations occur in a single expression, **precedence** is used to determine the order of operation. Usually NOT is evaluated first, then AND, and then OR. This order can be altered using parentheses.

There are multiple ways to symbolically represent the digital logic functions. For example,  $\sim A$ ,  $A'$ ,  $\overline{A}$ ,  $\neg A$  and  $\bar{A}$  are five ways to represent NOT(A). One can use the pipe symbol ( $\mid$ ) or the plus sign to represent logical OR:  $A \mid B$ ,  $A+B$ . In this book we will not use the plus sign to represent OR to avoid confusion with arithmetic addition. One can use the ampersand symbol ( $\&$ ) or a multiplication sign ( $*$ ,  $\times$ ) to represent logical AND:  $A \& B$ ,  $A \cdot B$ . In this book we will not use the multiplication sign to represent AND to avoid confusion with arithmetic multiplication. Another symbolic rule is adding a special character ( $* n \backslash$ ) to a name to signify the signal is negative logic (0 means true and 1 means false). These symbols do not signify an operation, but rather are part of the name used to clarify its meaning. E.g., Enable\* is a signal than means enable when the signal is zero.

Digital storage devices are used to make registers and memory. The simplest storage device is the **set-reset latch**. One way to build a set-reset latch is shown on the left side of Figure 1.14. If the inputs are  $S^*=0$  and  $R^*=1$ , then the  $Q$  output will be 1. Conversely, if the inputs are  $S^*=1$  and  $R^*=0$ , then the  $Q$  output will be 0. Normally, we leave both the  $S^*$  and  $R^*$  inputs high. We make the signal  $S^*$  go low, then back high to set the latch, making  $Q=1$ . Conversely, we make the signal  $R^*$  go low, then back high to reset the latch, making  $Q=0$ . If both  $S^*$  and  $R^*$  are 1, the value on  $Q$  will be remembered or stored. This latch enters an unpredictable mode when  $S^*$  and  $R^*$  are simultaneously low.

The **gated D latch** is also shown in Figure 1.14. The front-end circuits take a data input,  $D$ , and a control signal,  $W$ , and produce the  $S^*$  and  $R^*$  commands for the set-reset latch. For example, if  $W=0$ , then the latch is in its quiescent state, remembering the value on  $Q$  that was previously written. However, if  $W=1$ , then the data input is stored into the latch. In particular, if  $D=1$  and  $W=1$ , then  $S^*=0$  and  $R^*=1$ , making  $Q=1$ . Furthermore, if  $D=0$  and  $W=1$ , then  $S^*=1$  and  $R^*=0$ , making  $Q=0$ . So, to use the gated latch, we first put the data on the  $D$  input, next we make  $W$  go high, and then we make  $W$  go low. This causes the data value to be stored at  $Q$ . After  $W$  goes low, the data does not need to exist at the  $D$  input anymore. If the  $D$  input changes while  $W$  is high, then the  $Q$  output will change correspondingly. However, the last value on the  $D$  input is remembered or latched when the  $W$  falls, as shown in Table 1.2.

The **D flip-flop**, shown on the right of Figure 1.14, can also be used to store information. D flip-flops are the basic building block of RAM and registers on the computer. To save information, we first place the digital value we wish to remember on the  $D$  input, and then give a rising edge to the **clock** input. After the rising edge of the **clock**, the value is available at the  $Q$  output, and the  $D$  input is free to change. The operation of the clocked D flip-flop is defined on the right side of Table 1.2. The 74HC374 is an 8-bit D flip-flop, such that all 8 bits are stored on the rising edge of a single clock. The 74HC374 is similar in structure and operation to a register, which is high-speed memory inside the processor. If the gate (G) input on the 74HC374 is high, its outputs will be HiZ (floating), and if the gate is low, the outputs will be high or low depending on the stored values on the flip-flop. The D flip-flops are edge-triggered, meaning that changes in the output occur at the rising edge of the input clock.

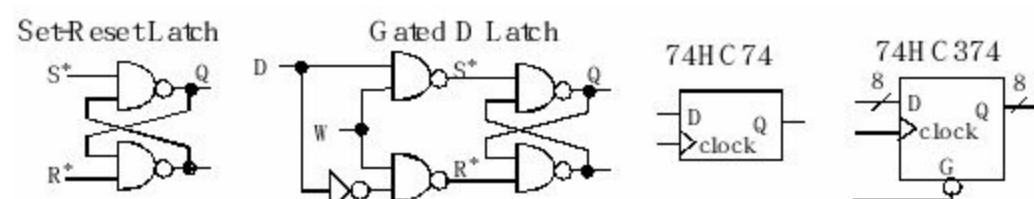


Figure 1.14. Digital storage elements.

D	W	Q		D	clock	Q
0	0	$Q_{old}$		0	0	$Q_{old}$
1	0	$Q_{old}$		0	1	$Q_{old}$
0	1	0		1	0	$Q_{old}$
1	1	1		1	1	$Q_{old}$
0	↓	0		0	↑	0
1	↓	1		1	↑	1

**Table 1.2. D flip-flop operation.**  $Q_{old}$  is the value of the D input at the time of fall of W or rise of clock.

The **tristate driver**, shown in Figure 1.15, can be used dynamically control signals within the computer. It is called tristate because there are three possible outputs: high, low, and HiZ. The tristate driver is an essential component from which computers are built. To activate the driver, we make its gate ( $G^*$ ) low. When the driver is active, its output (Y) equals its input (A). To deactivate the driver, we make its  $G^*$  high. When the driver is not active, its output Y floats independent of A. We will also see this floating state with the open collector logic, and it is also called HiZ or high impedance. The HiZ output means the output is neither driven high nor low. The operation of a tristate driver is defined in Table 1.3. The 74HC244 is an 8-bit tristate driver, such that all 8 bits are active or not active controlled by a single gate. The 74HC374 8-bit D flip-flop includes tristate drivers on its outputs. Normally, we can't connect two digital outputs together. The tristate driver provides a way to connect multiple outputs to the same signal, as long as at most one of the gates is active at a time.

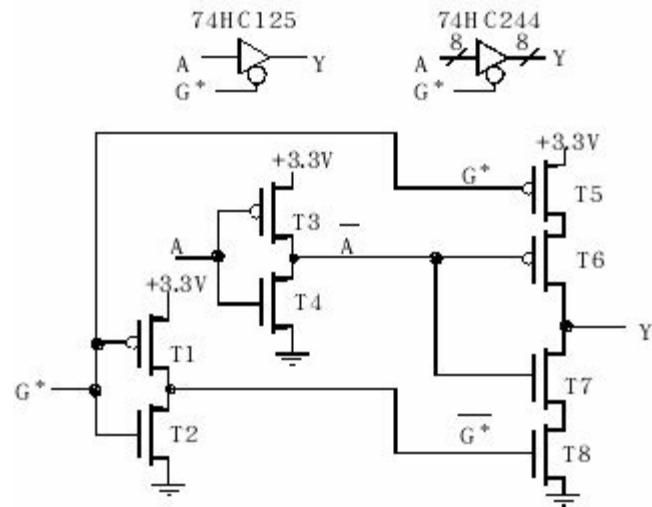


Figure 1.15. A 1-bit tristate driver and an 8-bit tristate driver ( $G^*$  is in negative logic).

Table 1.3 describes how the tristate driver in Figure 1.15 works. Transistors T1 and T2 create the logical complement of  $G^*$ . Similarly, transistors T3 and T4 create the complement of A. An input of  $G^*=0$  causes the driver to be active. In this case, both T5 and T8 will be on. With T5 and T8 on, the circuit behaves like a cascade of two NOT gates, so the output Y equals the input A. However, if the input  $G^*=1$ , both T5 and T8 will be off. Since T5 is in series with the +3.3V, and T8 in series with the ground, the output Y will be neither high nor low. I.e., it will float.

A	G*	T1	T2	T3	T4	T5	T6	T7	T8	Y
0	0	on	off	on	off	on	off	on	on	0
1	0	on	off	off	on	on	on	off	on	1
0	1	off	on	on	off	off	off	on	off	HiZ
1	1	off	on	off	on	off	on	off	off	HiZ

**Table 1.3. Tristate driver operation. HiZ is the floating state, such that the output is not high or low.**

The output of an open collector gate, drawn with the ‘×’, has two states low (0V) and HiZ (floating) as shown in Figure 1.16. Consider the operation of the transistor-level circuit for the 74HC05. If A is high (+3.3V), the transistor is active, and the output is low (0V). If A is low (0V), the transistor is off, and the output is neither high nor low. In general, we can use an **open collector NOT** gate to switch current on and off to a device, such as a relay, a light emitting diode (LED), a solenoid, or a small motor. The 74HC05, the 74LS05, the 7405, and the 7406 are all open collector NOT gates. 74HC04 is high-speed CMOS and can only sink up to 4 mA when its output is low. Since the 7405 and 7406 are transistor-transistor-logic (TTL) they can sink more current. In particular, the 7405 has a maximum output low current ( $I_{OL}$ ) of 16 mA, whereas the 7406 has a maximum  $I_{OL}$  of 40 mA.

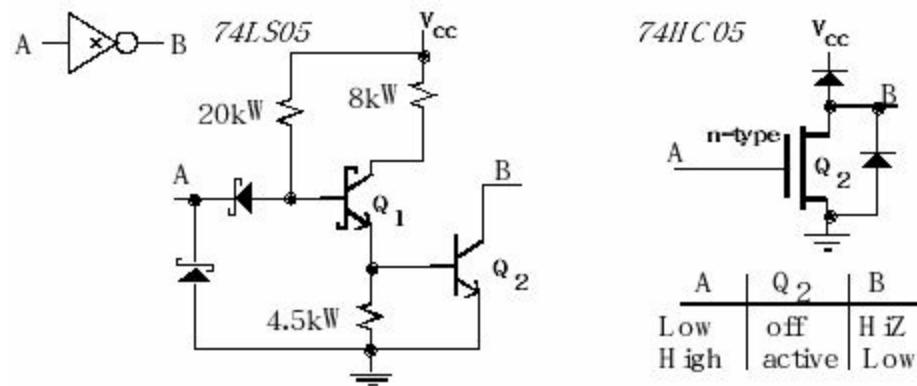


Figure 1.16. Two transistor implementations of an open collector NOT gate.

In the computer, we can build an 8-bit storage element, shown logically as Figure 1.10, by combining 8 flip-flops. This basic storage element is called a **register**, as shown in Figure 1.17. A **bus** is a collection of wires used to pass data from one place to another. In this circuit, the signals D7–D0 represent the data bus. Registers on the Tiva ® microcontrollers are 32-bits wide, but in this example we show an 8-bit register. We call it storage because as long the circuit remains powered, the digital information represented by the eight voltages Q7–Q0 will be remembered. There are two operations one performs on a register: write and read. To perform a write, one first puts the desired information on the 8 data bus wires (D7–D0). As you can see from Figure 1.17, these data bus signals are present on the D inputs of the 8 flip-flops. Next, the system pulses the **Write** signal high then low. This **Write** pulse will latch or store the desired data into the 8 flip-flops. The read operation will place a copy of the register information onto the data bus. Notice the gate signals of the tristate drivers are negative logic. This means if the **Read\*** signal is high, the tristate drivers are off, and this register does not affect signals on the bus. However, the read operation occurs by setting the **Read\*** signal low, which will place the register data onto the bus.

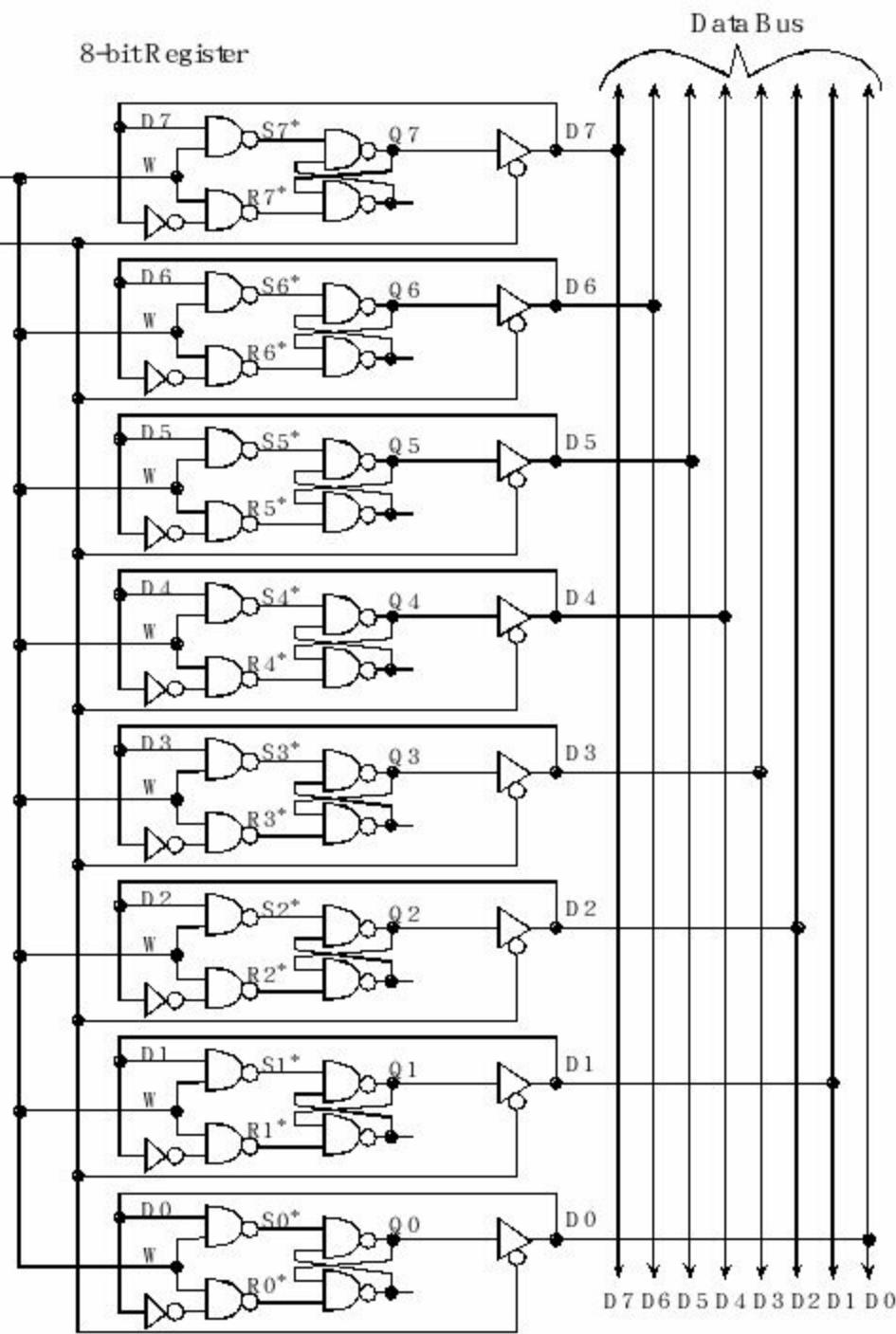


Figure 1.17. Digital logic implementation of a register.

# 1.4. Digital Information stored in Memory

**Memory** is a collection of hardware elements in a computer into which we store information, as shown in Figure 1.18. For most computers in today's market, each memory cell contains one byte of information, and each byte has a unique and sequential address. The memory is called **byte-addressable** because each byte has a separate address. The address of a memory cell specifies its physical location, and its content is the data. When we **write** to memory, we specify an address and 8, 16, or 32 bits of data, causing that information to be stored into the memory. Typically data flows from processor into memory during a write cycle. When we **read** from memory we specify an address, causing 8, 16, or 32 bits of data to be retrieved from the memory. Typically data flows from memory into the processor during a read cycle. **Read Only Memory**, or ROM, is a type of memory where the information is programmed or burned into the device, and during normal operation it only allows read accesses. **Random Access Memory** (RAM) is used to store temporary information, and during normal operation we can read from or write data into RAM. The information in the ROM is **nonvolatile**, meaning the contents are not lost when power is removed. In contrast, the information in the RAM is **volatile**, meaning the contents are lost when power is removed. The system can quickly and conveniently read data from a ROM. It takes a comparatively long time to program or burn data into a ROM. Writing to Flash ROM is a two-step process. First, the ROM is erased, causing all the bits to become 1. Second, the system writes zeroes into the ROM as needed. Each of these two steps requires around 1 ms to complete. In contrast, it is fast and easy to both read data from and write data into a RAM. Writing to RAM is about 100,000 times faster (on the order of 10 ns). ROM on the other hand is much denser than RAM. This means we can pack more ROM bits into a chip than we can pack RAM bits. Most microcontrollers have much more ROM than RAM. The TM4C123 has 32,768 bytes of RAM and 262,144 bytes of ROM.

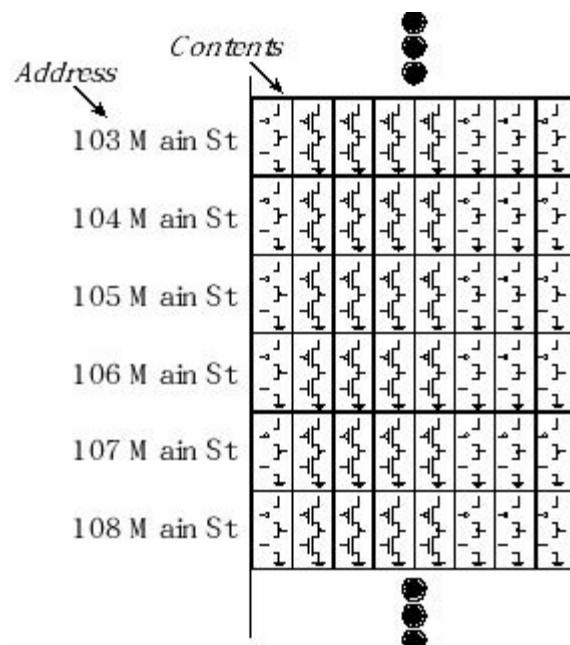


Figure 1.18. Memory is a sequential collection of data storage elements.

A great deal of confusion exists over the abbreviations we use for large numbers. In 1998 the International Electrotechnical Commission (IEC) defined a new set of abbreviations for the powers of 2, as shown in Table 1.4. These new terms are endorsed by the Institute of Electrical and Electronics Engineers (IEEE) and International Committee for Weights and Measures (CIPM) in situations where the use of a binary prefix is appropriate. The confusion arises over the fact that the mainstream computer industry, such as Microsoft, Apple, and Dell, continues to use the old terminology. According to the companies that market to consumers, a 1 GHz is 1,000,000,000 Hz but 1 Gbyte of memory is 1,073,741,824 bytes. The correct terminology is to use the SI-decimal abbreviations to represent powers of 10, and the IEC-binary abbreviations to represent powers of 2. The scientific meaning of 2 kilovolts is 2000 volts, but 2 kibibytes is the proper way to specify 2048 bytes. The term **kibibyte** is a contraction of kilo binary byte and is a unit of information or computer storage, abbreviated KiB.

$$1 \text{ KiB} = 2^{10} \text{ bytes} = 1024 \text{ bytes}$$

$$1 \text{ MiB} = 2^{20} \text{ bytes} = 1,048,576 \text{ bytes}$$

$$1 \text{ GiB} = 2^{30} \text{ bytes} = 1,073,741,824 \text{ bytes}$$

These abbreviations can also be used to specify the number of binary bits. The term **kibibit** is a contraction of kilo binary bit, and is a unit of information or computer storage, abbreviated Kibit.

A **mebibyte** (1 MiB is 1,048,576 bytes) is approximately equal to a megabyte (1 MB is 1,000,000 bytes), but mistaking the two has nonetheless led to confusion and even legal disputes. In the engineering community, it is appropriate to use terms that have a clear and unambiguous meaning.

Value	SI Decimal	SI Decimal		Value	IEC Binary	IEC Binary
$1000^1$	k	kilo-		$1024^1$	Ki	kibi-
$1000^2$	M	mega-		$1024^2$	Mi	mebi-
$1000^3$	G	giga-		$1024^3$	Gi	gibi-
$1000^4$	T	tera-		$1024^4$	Ti	tebi-
$1000^5$	P	peta-		$1024^5$	Pi	pebi-
$1000^6$	E	exa-		$1024^6$	Ei	exbi-
$1000^7$	Z	zetta-		$1024^7$	Zi	zebi-
$1000^8$	Y	yotta-		$1024^8$	Yi	yobi-

**Table 1.4. Common abbreviations for large numbers.**

# 1.5. Numbers

To solve problems using a computer we need to understand numbers and what they mean. Each digit in a **decimal** number has a place and a value. The **place** is a power of 10 and the **value** is selected from the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. A decimal number is simply a combination of its digits multiplied by powers of 10. For example

$$1984 = 1 \cdot 10^3 + 9 \cdot 10^2 + 8 \cdot 10^1 + 4 \cdot 10^0$$

Fractional values can be represented by using the negative powers of 10. For example,

$$273.15 = 2 \cdot 10^2 + 7 \cdot 10^1 + 3 \cdot 10^0 + 1 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

In a similar manner, each digit in a **binary** number has a place and a value. In binary numbers, the place is a power of 2, and the value is selected from the set {0, 1}. A binary number is simply a combination of its digits multiplied by powers of 2. To eliminate confusion between decimal numbers and binary numbers, we will put a subscript 2 after the number to mean binary. Because of the way the microcontroller operates, most of the binary numbers in this book will have 8, 16, or 32 bits. An 8-bit number is called a **byte**, and a 16-bit number is called a **halfword**. For example, the 8-bit binary number for 106 is

$$01101010_2 = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 64 + 32 + 8 + 2$$

**Checkpoint 1.15:** What is the numerical value of the 8-bit binary number  $11111111_2$ ?

Binary is the natural language of computers but a big nuisance for us humans. To simplify working with binary numbers, humans use a related number system called **hexadecimal**, which uses base 16. Just like decimal and binary, each hexadecimal digit has a place and a value. In this case, the place is a power of 16 and the value is selected from the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. As you can see, hexadecimal numbers have more possibilities for their digits than are available in the decimal format; so, we add the letters A through F, as shown in Table 1.5. A hexadecimal number is a combination of its digits multiplied by powers of 16. To eliminate confusion between various formats, we will put a 0x or a \$ before the number to mean hexadecimal. Hexadecimal representation is a convenient mechanism for us humans to define binary information, because it is extremely simple for humans to convert back and forth between binary and hexadecimal. Hexadecimal number system is often abbreviated as “hex”. A **nibble** is defined as 4 binary bits, or one hexadecimal digit. Each value of the 4-bit nibble is mapped into a unique hex digit, as shown in Table 1.5.

Hex Digit	Decimal Value	Binary Value
0	0	0000
1	1	0001
2	2	0010

3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A or a	10	1010
B or b	11	1011
C or c	12	1100
D or d	13	1101
E or e	14	1110
F or f	15	1111

**Table 1.5. Definition of hexadecimal representation.**

For example, the hexadecimal number for the 16-bit binary 0001 0010 1010 1101 is

$$0x12AD = 1 \cdot 16^3 + 2 \cdot 16^2 + 10 \cdot 16^1 + 13 \cdot 16^0 = 4096 + 512 + 160 + 13 = 4781$$

**Observation:** In order to maintain consistency between assembly and C programs, we will use the 0x format when writing hexadecimal numbers in this book.

**Checkpoint 1.16:** What is the numerical value of the 8-bit hexadecimal number 0xFE?

As illustrated in Figure 1.19, to convert from binary to hexadecimal we can:

- 1) Divide the binary number into right justified nibbles,
- 2) Convert each nibble into its corresponding hexadecimal digit.

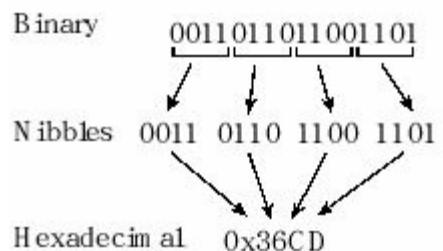


Figure 1.19. Example conversion from binary to hexadecimal.

**Checkpoint 1.17:** Convert the binary number  $01000111_2$  to hexadecimal.

**Checkpoint 1.18:** Convert the binary number  $110110101011_2$  to hexadecimal.

As illustrated in Figure 1.20, to convert from hexadecimal to binary we can:

- 1) Convert each hexadecimal digit into its corresponding 4-bit binary nibble,
- 2) Combine the nibbles into a single binary number.

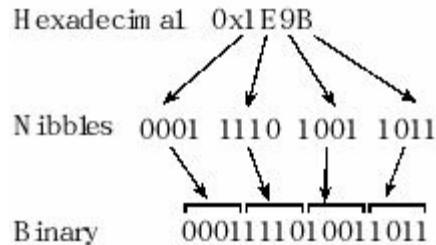


Figure 1.20. Example conversion from hexadecimal to binary.

**Checkpoint 1.19:** Convert the hex number 0x49 to binary.

**Checkpoint 1.20:** Convert the hex number 0xBEEF to binary.

**Checkpoint 1.21:** How many binary bits does it take to represent 0x12345?

Computer programming environments use a wide variety of symbolic notations to specify the numbers in hexadecimal. As an example, assume we wish to represent the binary number  $01111010_2$ . Some assembly languages use \$7A. Some assembly languages use 7AH. The C language uses 0x7A. Patt's LC-3 simulator uses x7A. We will use the 0x7A format.

**Precision** is the number of distinct or different values. We express precision in alternatives, decimal digits, bytes, or binary bits. **Alternatives** are defined as the total number of possibilities. For example, an 8-bit number format can represent 256 different numbers. An 8-bit **digital to analog converter** (DAC) can generate 256 different analog outputs. An 8-bit **analog to digital converter** (ADC) can measure 256 different analog inputs. Table 1.6 illustrates the relationship between precision in binary bits and precision in alternatives. The operation  $[[ x ]]$  is defined as the greatest integer of  $x$ . E.g.,  $[[2.1]]$   $[[2.9]]$  and  $[[3.0]]$  are all equal to 3. The **Bytes** column in Table 1.6 specifies how many bytes of memory it would take to store a number with that precision assuming the data were not packed or compressed in any way.

Binary bits	Bytes	Alternatives
8	1	256
10	2	1024
12	2	4096
16	2	65536
20	3	1,048,576
24	3	16,777,216
30	4	1,073,741,824
32	4	4,294,967,296
n	$[[n/8]]$	$2^n$

Table 1.6. Relationship between bits, bytes and alternatives as units of precision.

**Checkpoint 1.22:** How many bytes of memory would it take to store a 60-bit number?

**Decimal digits** are used to specify precision of measurement systems that display results as numerical values, as defined in Table 1.7. A full decimal digit can be any value 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. A digit that can be either 0 or 1 is defined as a  $\frac{1}{2}$  decimal digit. The terminology of a  $\frac{1}{2}$  decimal digit did not arise from a mathematical perspective of precision, but rather it arose from the physical width of the light emitting diode (LED) or liquid crystal display (LCD) module used to display a blank or '1' as compared to the width of a full digit. Notice in Figure 1.21 that the 7-segment modules capable of displaying 0 to 9 are about 1 cm wide; however, the corresponding 2-segment modules capable of being blank or displaying a 1 are about half as wide. Similarly, we define a digit that can be + or - also as a half decimal digit, because it has 2 choices. A digit that can be 0,1,2,3 is defined as a  $\frac{3}{4}$  decimal digit, because it is wider than a  $\frac{1}{2}$  digit but narrower than a full digit. We also define a digit that can be -1, -0, +0, or +1 as a  $\frac{3}{4}$  decimal digit, because it also has 4 choices. We use the expression  $4\frac{1}{2}$  decimal digits to mean 20,000 alternatives and the expression  $4\frac{3}{4}$  decimal digits to mean 40,000 alternatives. The use of a  $\frac{1}{2}$  decimal digit to mean twice the number of alternatives or one additional binary bit is widely accepted. On the other hand, the use of  $\frac{3}{4}$  decimal digit to mean four times the number of alternatives or two additional binary bits is not as commonly accepted. For example, consider the two ohmmeters shown in Figure 1.21. As illustrated in the figure, both are set to the 0 to 200 k $\Omega$  range. The  $3\frac{1}{2}$  digit ohmmeter has a resolution of 0.1 k $\Omega$  with measurements ranging from 0.0 to 199.9 k $\Omega$ . On the other hand, the  $4\frac{1}{2}$  digit ohmmeter has a resolution of 0.01 k $\Omega$  with measurements ranging from 0.00 to 199.99 k $\Omega$ .

**Observation:** A good rule of thumb to remember is  $2^{10 \cdot n} \approx 10^{3 \cdot n}$ .

Decimal digits	Alternatives
3	1000
$3\frac{1}{2}$	2000
$3\frac{3}{4}$	4000
4	10000
$4\frac{1}{2}$	20000
$4\frac{3}{4}$	40000
n	$10^n$
$n\frac{1}{2}$	$2 \cdot 10^n$
$n\frac{3}{4}$	$4 \cdot 10^n$

**Table 1.7. Definition of decimal digits as a unit of precision.**



Figure 1.21. Two ohmmeters: the one on the left has 3½ decimal digits and the one on the right has 4½.

**Checkpoint 1.23:** How many binary bits is equivalent to 3½ decimal digits?

**Checkpoint 1.24:** About how many decimal digits is 64 binary bits? You can answer this without a calculator, just using the “rule of thumb”.

A **byte** contains 8 bits as shown in Figure 1.22, where each bit  $b_7, \dots, b_0$  is binary and has the value 1 or 0. We specify  $b_7$  as the **most significant bit** or MSB, and  $b_0$  as the least significant bit or LSB. In C, the specifier **char** means 8 bits or 1 byte. In C99, the specifiers **uint8\_t** and **int8\_t** mean 8 bits or 1 byte. In this book, we will use **char** to store ASCII characters and we will use **uint8\_t** and **int8\_t** to store 8-bit numbers.

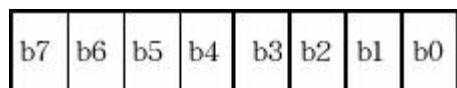


Figure 1.22. 8-bit binary format.

If a byte is used to represent an unsigned number, then the value of the number is

$$N = 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0$$

Notice that the significance of bit  $n$  is  $2^n$ . There are 256 different unsigned 8-bit numbers. The smallest unsigned 8-bit number is 0 and the largest is 255. For example,  $00001010_2$  is  $8+2$  or 10. Other examples are shown in Table 1.8. The least significant bit can tell us if the number is even or odd. Furthermore, if the bottom  $n$  bits are 0, the number is divisible by  $2^n$ .

binary	hex	Calculation	decimal
$00000000_2$	0x00		0
$01000001_2$	0x41	$64+1$	65
$00010110_2$	0x16	$16+4+2$	22
$10000111_2$	0x87	$128+4+2+1$	135
$11111111_2$	0xFF	$128+64+32+16+8+4+2+1$	255

**Table 1.8. Example conversions from unsigned 8-bit binary to hexadecimal and to decimal.**

**Checkpoint 1.25:** Convert the binary number  $01101011_2$  to unsigned decimal.

**Checkpoint 1.26:** Convert the hex number 0x46 to unsigned decimal.

The **basis** of a number system is a subset from which linear combinations of the basis elements can be used to construct the entire set. The basis represents the “places” in a “place-value” system. For positive integers, the basis is the infinite set  $\{1, 10, 100, \dots\}$ , and the “values” can range from 0 to 9. Each positive integer has a unique set of values such that the dot-product of the value vector times the basis vector yields that number. For example, 2345 is the dot-product  $(\dots, 2, 3, 4, 5) \cdot (\dots, 1000, 100, 10, 1)$ , which is  $2*1000+3*100+4*10+5$ . For the unsigned 8-bit number system, the basis elements are

$\{1, 2, 4, 8, 16, 32, 64, 128\}$

The values of a binary number system can only be 0 or 1. Even so, each 8-bit unsigned integer has a unique set of values such that the dot-product of the values times the basis yields that number. For example,  $69=0x45$  is  $(0,1,0,0,0,1,0,1) \square (128,64,32,16,8,4,2,1)$ , which equals  $0*128+1*64+0*32+0*16+0*8+1*4+0*2+1*1$ . Conveniently, there is no other set of 0's and 1's, such that set of values multiplied by the basis is 69. In other words, each 8-bit unsigned binary representation of the values 0 to 255 is unique.

One way for us to convert a decimal number into binary is to use the basis elements. The overall approach is to start with the largest basis element and work towards the smallest. More precisely, we start with the most significant bit and work towards the least significant bit. One by one, we ask ourselves whether or not we need that basis element to create our number. If we do, then we set the corresponding bit in our binary result and subtract the basis element from our number. If we do not need it, then we clear the corresponding bit in our binary result. We will work through the algorithm with the example of converting 100 to 8-bit binary, see Table 1.9. We start with the largest basis element (in this case 128) and ask whether or not we need to include it to make 100? Since our number is less than 128, we do not need it, so bit 7 is zero. We go the next largest basis element, 64 and ask, "do we need it?" We do need 64 to generate our 100, so bit 6 is one and we subtract 100 minus 64 to get 36. Next, we go the next basis element, 32 and ask, "do we need it?" Again, we do need 32 to generate our 36, so bit 5 is one and we subtract 36 minus 32 to get 4. Continuing along, we do not need basis elements 16 or 8, but we do need basis element 4. Once we subtract the 4, our working result is zero, so basis elements 2 and 1 are not needed. Putting it together, we get  $01100100_2$ ,  $100 = 64+32+4$ .

**Checkpoint 1.27:** In this conversion algorithm, how can we tell if a basis element is needed?

**Observation:** If the least significant binary bit is zero, then the number is even.

**Observation:** If the right-most n bits (least sign.) are zero, then the number is divisible by  $2^n$ .

**Observation:** Bit 7 of an 8-bit number determines whether it is greater than or equal to 128.

Number	Basis	Need it?	bit	Operation
100	128	no	bit 7=0	none
100	64	yes	bit 6=1	subtract 100-64
36	32	yes	bit 5=1	subtract 36-32
4	16	no	bit 4=0	none
4	8	no	bit 3=0	none
4	4	yes	bit 2=1	subtract 4-4
0	2	no	bit 1=0	none
0	1	no	bit 0=0	none

**Table 1.9. Example conversion from decimal to unsigned 8-bit binary to hexadecimal.**

**Checkpoint 1.28:** Give the representations of the decimal 45 in 8-bit binary and hexadecimal.

**Checkpoint 1.29:** Give the representations of the decimal 200 in 8-bit binary and hexadecimal.

One of the first schemes to represent signed numbers was called **one's complement**. It was called one's complement because to negate a number, we complement (logical not) each bit. For example, if 25 equals  $00011001_2$  in binary, then  $-25$  is  $11100110_2$ . An 8-bit one's complement number can vary from -127 to +127. The most significant bit is a sign bit, which is 1 if and only if the number is negative. The difficulty with this format is that there are two zeros +0 is  $00000000_2$ , and  $-0$  is  $11111111_2$ . Another problem is that one's complement numbers do not have basis elements. These limitations led to the use of two's complement.

The **two's complement** number system is the most common approach used to define signed numbers. It is called two's complement because to negate a number, we complement each bit (like one's complement), and then add 1. For example, if 25 equals  $00011001_2$  in binary, then  $-25$  is  $11100111_2$ . If a byte is used to represent a signed two's complement number, then the value of the number is

$$N = -128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0$$

**Observation:** One usually means two's complement when one refers to signed integers.

There are 256 different signed 8-bit numbers. The smallest signed 8-bit number is -128 and the largest is 127. For example,  $10000010_2$  equals  $-128+2$  or -126. Other examples are shown in Table 1.10.

binary	Hex	Calculation	decimal
$00000000_2$	0x00		0
$01000001_2$	0x41	$64+1$	65
$00010110_2$	0x16	$16+4+2$	22
$10000111_2$	0x87	$-128+4+2+1$	-121
$11111111_2$	0xFF	$-128+64+32+16+8+4+2+1$	-1

**Table 1.10. Example conversions from signed 8-bit binary to hexadecimal and to decimal.**

**Checkpoint 1.30:** Convert the signed binary number  $11101010_2$  to signed decimal.

**Checkpoint 1.31:** Are the signed and unsigned decimal representations of the 8-bit hex number 0x45 the same or different?

For the signed 8-bit number system the basis elements are

$$\{1, 2, 4, 8, 16, 32, 64, -128\}$$

**Observation:** The most significant bit in a two's complement signed number will specify the sign.

Notice that the same binary pattern of  $1111111_2$  could represent either 255 or -1. It is very important for the software developer to keep track of the number format. The computer cannot determine whether a number is signed or unsigned. You, as the programmer, will determine whether the number is signed or unsigned by the specific assembly instructions you select to operate on the number. Some operations like addition, subtraction, multiplication, and shift left (multiply by 2) use the same hardware (instructions) for both unsigned and signed operations. On the other hand, division, and shift right (divide by 2) require separate hardware (instructions) for unsigned and signed operations.

Similar to the unsigned algorithm, we can use the basis to convert a decimal number into signed binary. We will work through the algorithm with the example of converting -100 to 8-bit binary, as shown in Table 1.11. We start with the most significant bit (in this case -128) and decide do we need to include it to make -100? Yes (without -128, we would be unable to add the other basis elements together to get any negative result), so we set bit 7 and subtract the basis element from our value. Our new value equals -100 minus -128, which is 28. We go the next largest basis element, 64 and ask, "do we need it?" We do not need 64 to generate our 28, so bit 6 is zero. Next we go the next basis element, 32 and ask, "do we need it?" We do not need 32 to generate our 28, so bit 5 is zero. Now we need the basis element 16, so we set bit 4, and subtract 16 from our number 28 ( $28-16=12$ ). Continuing along, we need basis elements 8 and 4 but not 2,1. Putting it together we get  $10011100_2$  (which means  $-128+16+8+4$ ).

Number	Basis	Need it	bit	Operation
-100	-128	yes	bit 7=1	subtract -100 - -128
28	64	no	bit 6=0	none
28	32	no	bit 5=0	none
28	16	yes	bit 4=1	subtract 28-16
12	8	yes	bit 3=1	subtract 12-8
4	4	yes	bit 2=1	subtract 4-4
0	2	no	bit 1=0	none
0	1	no	bit 0=0	none

**Table 1.11. Example conversion from decimal to signed 8-bit binary.**

**Observation:** To take the negative of a two's complement signed number we first complement (flip) all the bits, then add 1.

A second way to convert negative numbers into binary is to first convert them into unsigned binary, then do a two's complement negate. For example, we earlier found that +100 is  $01100100_2$ . The two's complement negate is a two-step process. First we do a logic complement (flip all bits) to get  $10011011_2$ . Then add one to the result to get  $10011100_2$ .

A third way to convert negative numbers into binary is to first add 256 to the number, then convert the unsigned result to binary using the unsigned method. For example, to find  $-100$ , we add 256 plus  $-100$  to get 156. Then we convert 156 to binary resulting in  $10011100_2$ . This method works because in 8-bit binary math adding 256 to a number does not change the value. E.g.,  $256-100$  has the same 8-bit binary value as  $-100$ .

**Checkpoint 1.32:** Give the representations of  $-45$  in 8-bit binary and hexadecimal.

**Checkpoint 1.33:** Why can't you represent the number  $200$  using 8-bit signed binary?

**Sign-magnitude** representation dedicates one bit as the sign leaving the remaining bits to specify the magnitude of the number. If  $b_7$  is 1 then the number is negative, otherwise the number is positive.

$$N = (-1)^{b_7} \cdot (64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0)$$

Unfortunately, there is no basis set for the sign-magnitude number system. For example,  $10000010_2$  equals  $-1 \cdot 2$  or  $-2$ . Other examples are shown in Table 1.12.

binary	hex	Calculation	decimal
$00000000_2$	0x00		0
$01000001_2$	0x41	$64+1$	65
$00010110_2$	0x16	$16+4+2$	22
$10000111_2$	0x87	$-1 \cdot (4+2+1)$	-7
$11111111_2$	0xFF	$-1 \cdot (64+32+16+8+4+2+1)$	-127

**Table 1.12. Example conversions from sign-magnitude 8-bit binary to hexadecimal and to decimal.**

Another problem with sign-magnitude is that there are two representations of the number 0: “ $00000000$ ” and “ $10000000$ ”. But, the biggest advantage of two’s complement signed numbers over sign-magnitude is that the same addition and subtraction hardware can be used for both signed and unsigned numbers. We also can use the same hardware for shift left. Although the hardware for these three operations works for both signed and unsigned numbers, the overflow (error) conditions are distinct. The C bit in the condition code register signifies unsigned overflow, and the V bit means a signed overflow has occurred. Unfortunately, we must use separate signed and unsigned operations for divide, and shift right.

**Common Error:** An error will occur if you use signed operations on unsigned numbers, or use unsigned operations on signed numbers.

**Maintenance Tip:** To improve the clarity of our software, always specify the format of your data (signed versus unsigned) when defining or accessing the data.

When communicating with humans (input or output), computers need to store information in an easy-to-read decimal format. One such format is **binary coded decimal** or BCD. The 8-bit BCD format contains two decimal digits, and each decimal digit is encoded in four-bit binary. For example, the number 72 is stored as 0x72 or  $01110010_2$ . We can represent numbers from 0 to 99 using 8-bit BCD.

**Checkpoint 1.34:** What binary values are used to store the number 25 in 8-bit BCD format?

A **halfword** or **double byte** contains 16 bits, where each bit  $b_{15}, \dots, b_0$  is binary and has the value 1 or 0, as shown in Figure 1.23. In C, the specifier **short** means 16 bits or 2 bytes. In C99, the specifiers **uint16\_t** and **int16\_t** mean 16 bits or 2 bytes.

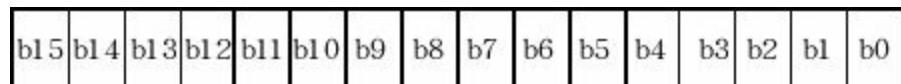


Figure 1.23. 16-bit binary format.

If a halfword is used to represent an unsigned number, then the value of the number is

$$\begin{aligned} N = & 32768 \cdot b_{15} + 16384 \cdot b_{14} + 8192 \cdot b_{13} + 4096 \cdot b_{12} \\ & + 2048 \cdot b_{11} + 1024 \cdot b_{10} + 512 \cdot b_9 + 256 \cdot b_8 \\ & + 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0 \end{aligned}$$

There are 65536 different unsigned 16-bit numbers. The smallest unsigned 16-bit number is 0 and the largest is 65535. For example,  $0010000110000100_2$ , or 0x2184 is  $8192+256+128+4$  or 8580. Other examples are shown in Table 1.13.

binary	hex	Calculation	decimal
$0000000000000000_2$	0x0000		0
$0000010000000001_2$	0x0401	$1024+1$	1025
$0000110010100000_2$	0x0CA0	$2048+1024+128+32$	3232
$1000111000000010_2$	0x8E02	$32768+2048+1024+512+2$	36354
$1111111111111111_2$	0xFFFF	$32768+16384+8192+4096+2048+1024+512+256+128+64+32+16+8+4+2+1$	65535

Table 1.13. Example conversions from unsigned 16-bit binary to hexadecimal and to decimal.

**Checkpoint 1.35:** Convert the 16-bit binary number  $0010000001101010_2$  to unsigned decimal.

**Checkpoint 1.36:** Convert the 16-bit hex number 0x1234 to unsigned decimal.

For the unsigned 16-bit number system the basis elements are

{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768}

**Checkpoint 1.37:** Convert the unsigned decimal number 1234 to 16-bit hexadecimal.

**Checkpoint 1.38:** Convert the unsigned decimal number 10000 to 16-bit binary.

There are also 65536 different signed 16-bit numbers. The smallest two's complement signed 16-bit number is  $-32768$  and the largest is  $32767$ . For example,  $1101000000000100_2$  or  $0xD004$  is  $-32768+16384+4096+4$  or  $-12284$ . Other examples are shown in Table 1.14.

binary	hex	Calculation	decimal
$0000000000000000_2$	0x0000		0
$0000010000000001_2$	0x0401	$1024+1$	1025
$0000110010100000_2$	0x0CA0	$2048+1024+128+32$	3232
$1000010000000010_2$	0x8402	$-32768+1024+2$	-31742
$1111111111111111_2$	0xFFFF	$-32768+16384+8192+4096+2048+1024+512+256+128+64+32+16+8+4+2+1$	-1

**Table 1.14. Example conversions from signed 16-bit binary to hexadecimal and to decimal.**

If a halfword is used to represent a signed two's complement number, then the value of the number is

$$\begin{aligned} N = & -32768 \cdot b_{15} + 16384 \cdot b_{14} + 8192 \cdot b_{13} + 4096 \cdot b_{12} + 2048 \cdot b_{11} + 1024 \cdot b_{10} + \\ & 512 \cdot b_9 + 256 \cdot b_8 + 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0 \end{aligned}$$

**Checkpoint 1.39:** Convert the 16-bit hex number 0x1234 to signed decimal.

**Checkpoint 1.40:** Convert the 16-bit hex number 0xABCD to signed decimal.

For the signed 16-bit number system the basis elements are

$$\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, -32768\}$$

**Common Error:** An error will occur if you use 16-bit operations on 8-bit numbers, or use 8-bit operations on 16-bit numbers.

**Maintenance Tip:** To improve the clarity of your software, always specify the precision of your data when defining or accessing the data.

**Checkpoint 1.41:** Convert the signed decimal number 1234 to 16-bit hexadecimal.

**Checkpoint 1.42:** Convert the signed decimal number  $-10000$  to 16-bit binary.

On the ARM, a **word** is 32 bits wide. In C, the specifier **long** means 32 bits. In C99, the specifiers **uint32\_t** and **int32\_t** mean 32 bits. Consider an unsigned number with 32 bits, where each bit  $b_{31}, \dots, b_0$  is binary and has the value 1 or 0. If a 32-bit number is used to represent an unsigned integer, then the value of the number is

$$N = 2^{31} \cdot b_{31} + 2^{30} \cdot b_{30} + \dots + 2 \cdot b_1 + b_0 = \sum_{i=0}^{31} 2^i * b_i$$

There are  $2^{32}$  different unsigned n-bit numbers. The smallest unsigned 32-bit number is 0 and the largest is  $2^{32}-1$ . This range is 0 to about 4 billion. For the unsigned 32-bit number system, the basis elements are

$$\{1, 2, 4, \dots, 2^{29}, 2^{30}, 2^{31}\}$$

If a 32-bit binary number is used to represent a signed two's complement number, then the value of the number is

$$N = -2^{31} \cdot b_{31} + 2^{30} \cdot b_{30} + \dots + 2 \cdot b_1 + b_0 = -2^{31} \cdot b_{31} + \sum_{i=0}^{30} 2^i * b_i$$

There are also  $2^{32}$  different signed n-bit numbers. The smallest signed n-bit number is  $-2^{31}$  and the largest is  $2^{31}-1$ . This range is about -2 billion to +2 billion. For the signed 32-bit number system, the basis elements are

$$\{1, 2, 4, \dots, 2^{29}, 2^{30}, -2^{31}\}$$

**Maintenance Tip:** When programming in C, we will use data types **char** **short** and **long** when we wish to explicitly specify the precision as 8-bit, 16-bit or 32-bit. Whereas, we will use the **int** data type only when we don't care about precision, and we wish the compiler to choose the most efficient way to perform the operation. On most compilers for the ARM, the **int** data type will be 32 bits.

**Observation:** When programming in assembly, we will always explicitly specify the precision of our numbers and calculations.

The C99 programming standard eliminates the confusion, defining these types:

<b>int8_t</b>	signed 8-bit
<b>int16_t</b>	signed 16-bit
<b>int32_t</b>	signed 32-bit
<b>int64_t</b>	signed 64-bit
<b>char</b>	8-bit ASCII characters

<b>uint8_t</b>	unsigned 8-bit
<b>uint16_t</b>	unsigned 16-bit
<b>uint32_t</b>	unsigned 32-bit
<b>uint64_t</b>	unsigned 64-bit

We will use **fixed-point** numbers when we wish to express values in our computer that have noninteger values. A fixed-point number contains two parts. The first part is a variable integer, called I. The variable integer will be stored on the computer. The second part of a fixed-point number is a fixed constant, called the **resolution**  $\Delta$ . The fixed constant will NOT be stored on the computer. The fixed constant is something we keep track of while designing the software operations. The value of the number is the product of the variable integer times the fixed constant. The integer may be signed or unsigned. An unsigned fixed-point number is one that has an unsigned variable integer. A signed fixed-point number is one that has a signed variable integer. The **precision** of a number system is the total number of distinguishable values that can be represented. The precision of a fixed-point number is determined by the number of bits used to store the variable integer. On most microcontrollers, we can use 8, 16, or 32 bits for the integer. With **binary fixed point** the fixed constant is a power of 2. An example is shown in Figure 1.24.

Binary fixed-point value =  $I \cdot 2^n$  for some constant integer n

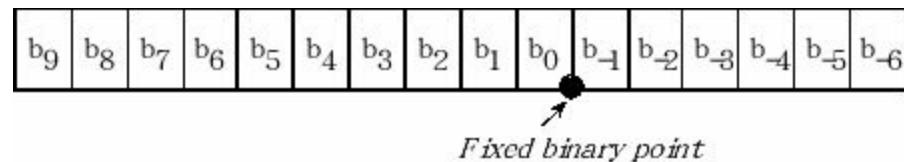


Figure 1.24. 16-bit binary fixed-point format with  $\Delta=2^{-6}$ .

# 1.6. Character information

We can use bytes to represent characters with the American Standard Code for Information Interchange (ASCII) code. Standard ASCII is actually only 7 bits, but is stored using 8-bit bytes with the most significant bit equal to 0. For example, the capital ‘V’ is defined by the 8-bit binary pattern  $01010110_2$ . Table 1.15 shows the ASCII code for some of the commonly-used nonprinting characters. In C we will use the **char** data type to represent characters.

Abbr.	ASCII character	Binary	Hexadecimal	Decimal
BS	Delete or Backspace	$00001000_2$	0x08	8
HT	Tab	$00001001_2$	0x09	9
CR	Enter or Return	$00001101_2$	0x0D	13
LF	Line feed	$00001010_2$	0x0A	10
SP	Space	$00100000_2$	0x20	32

**Table 1.15. Common special characters and their ASCII representations.**

The 7-bit ASCII code definitions are given in the Table 1.16. For example, the letter ‘V’ is in the 0x50 column and the 6 row. Putting the two together yields hexadecimal 0x56.

## BITS 4 to 6

		0	1	2	3	4	5	6	7
	0	NUL	DLE	SP	0	@	P	`	p
B	1	SOH	DC1/XON	!	1	A	Q	a	q
I	2	STX	DC2	"	2	B	R	b	r
T	3	ETX	DC3/XOFF	#	3	C	S	c	s
S	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
O	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
T	8	BS	CAN	(	8	H	X	h	x
O	9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z	
3	B	VT	ESC	+	;	K	[	k	{
	FF	FS	,	<	L	\	I		

C								
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

**Table 1.16. Standard 7-bit ASCII.**

**Checkpoint 1.43:** How is the character ‘0’ represented in ASCII?

**Checkpoint 1.44:** Assume variable n contains an ASCII code 0 to 9. Write a formula that converts the ASCII code in n into the corresponding decimal number.

Standard ASCII code uses only 7 bits and thus can only represent 128 different characters. The ISO/IEC 8859 standard uses the 8th bit of the byte to define additional characters such as graphics and letters in other alphabets. This standard is jointly published by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). Unfortunately, there can be one character with multiple numerical encodings or one numerical value that could represent different characters. This ambiguity has led to more complex encoding schemes using multiple bytes to represent character data such as the Unicode Standard, see <http://www.unicode.org/>. Unicode is an active and ongoing consortium with a goal to provide a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. ISO/IEC 10646 is the corresponding international standard synchronized with the Unicode Standard. As embedded systems are asked to communicate with other computers across the world, these standards will be a critical component for guaranteeing unambiguous communication.

One way to encode a character string is to use null-termination. In this way, the characters of the string are stored one right after the other, and the end of the string is signified by the **NUL** character (0x00). For example, the string “Valvano” is encoded as these 8 bytes 0x56, 0x61, 0x6C, 0x76, 0x61, 0x6E, 0x6F, 0x00. Typically we use a pointer to the first byte to identify the string, as shown in Figure 1.25.

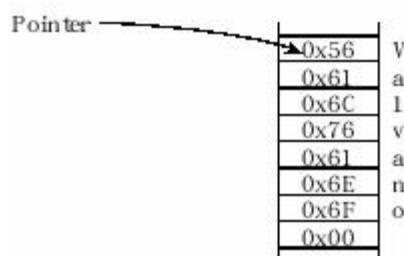


Figure 1.25. Strings are stored as a sequence of ASCII characters, followed by a null.

**Checkpoint 1.45:** How is “Hello World” encoded as a null-terminated ASCII string?

**Observation:** When outputting to some devices we send just a 13 (CR, 0x0D) to go to the next line, while for other devices we need to send both a 13 and a 10 (LF, 0x0A).

# 1.7. Computer Architecture

A **computer** combines a processor, random access memory (RAM), read only memory (ROM), and input/output (I/O) ports. The common bus in Figure 1.26 defines the von Neumann architecture. Computers are not intelligent. Rather, you are the true genius. Computers are electronic idiots. They can store a lot of data, but they will only do exactly what we tell them to do. Fortunately, however, they can execute our programs quite quickly, and they don't get bored doing the same tasks over and over again. **Software** is an ordered sequence of very specific instructions that are stored in memory, defining exactly what and when certain tasks are to be performed. It is a set of instructions, stored in memory, that are executed in a complicated but well-defined manner. The **processor** executes the software by retrieving and interpreting these instructions one at a time. A **microprocessor** is a small processor, where small refers to size (i.e., it fits in your hand) and not computational ability. For example, Intel Xeon, AMD FX, and Sun SPARC are microprocessors. An ARM ® Cortex™-M microcontroller includes a processor together with the bus and some peripherals.

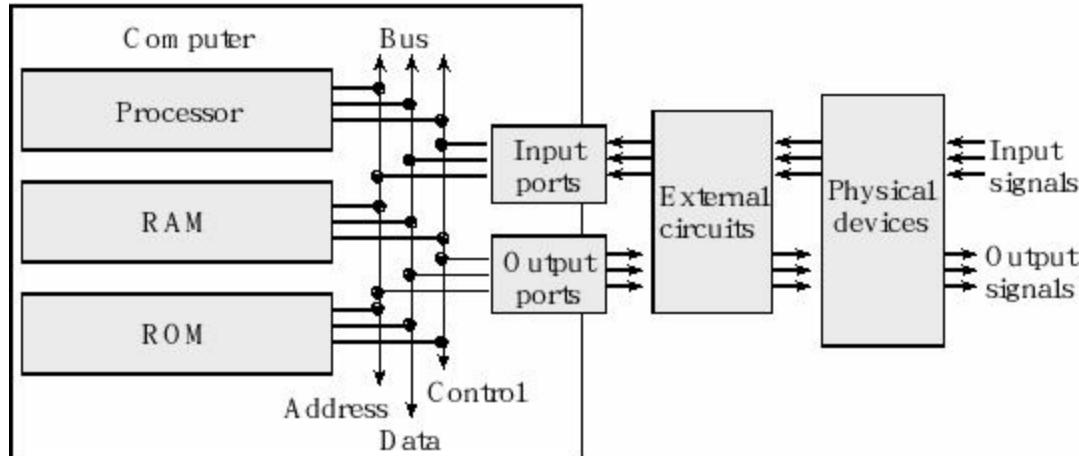


Figure 1.26. The basic components of a von Neumann computer include processor, memory and I/O.

A **microcomputer** is a small computer, where again small refers to size (i.e., you can carry it) and not computational ability. For example, a desktop PC is a microcomputer. Small in this context describes its size not its computing power. Consequently, there can be great confusion over the term microcomputer, because it can refer to a very wide range of devices from a PIC12C508, which is an 8-pin chip with 512 words of ROM and 25 bytes RAM, to the most powerful I7-based personal computer.

A **port** is a physical connection between the computer and its outside world. Ports allow information to enter and exit the system. Information enters via the input ports and exits via the output ports. Other names used to describe ports are I/O ports, I/O devices, interfaces, or sometimes just devices. A **bus** is a collection of wires used to pass information between modules.

A very small microcomputer, called a **microcontroller**, contains all the components of a computer (processor, memory, I/O) on a single chip. As shown in Figure 1.27, the Atmel ATTiny, the Texas Instruments MSP430, and the Texas Instruments TM4C123 are examples of microcontrollers. Because a microcomputer is a small computer, this term can be confusing because it is used to describe a wide range of systems from a 6-pin ATTiny4 running at 1 MHz with 512 bytes of program memory to a personal computer with state-of-the-art 64-bit multi-core processor running at multi-GHz speeds having terabytes of storage.

The computer can store information in **RAM** by writing to it, or it can retrieve previously stored data by reading from it. RAMs are **volatile**; meaning if power is interrupted and restored the information in the RAM is lost. Most microcontrollers have **static RAM (SRAM)** using six metal-oxide-semiconductor field-effect transistors (MOS or MOSFET) to create each memory bit. Four transistors are used to create two cross-coupled inverters that store the binary information, and the other two are used to read and write the bit.

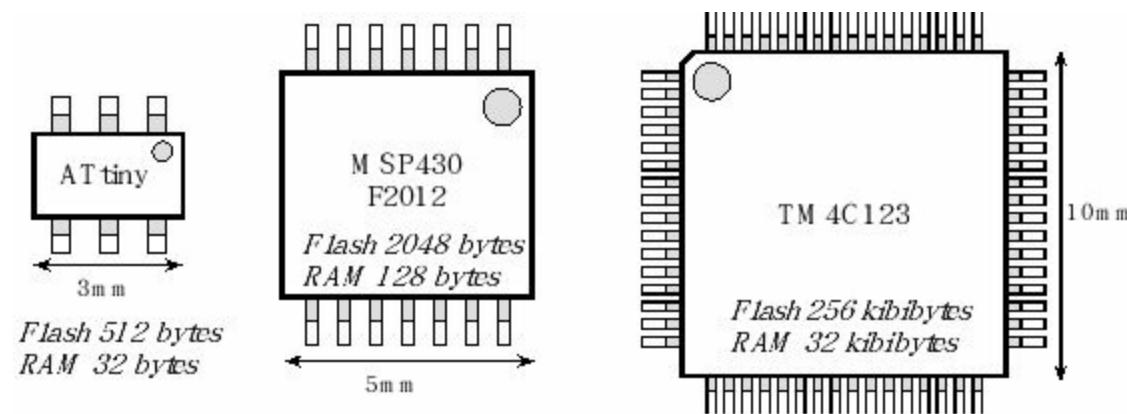


Figure 1.27. A microcontroller is a complete computer on a single chip. The TM4C123 has 43 I/O pins. The TM4C1294 has 1024 kibibytes of Flash ROM, 256 kibibytes of RAM, and 90 I/O pins.

Information is programmed into **ROM** using techniques more complicated than writing to RAM. From a programming viewpoint, retrieving data from a ROM is identical to retrieving data from RAM. ROMs are nonvolatile; meaning if power is interrupted and restored the information in the ROM is retained. Some ROMs are programmed at the factory and can never be changed. A Programmable ROM (PROM) can be erased and reprogrammed by the user, but the erase/program sequence is typically 10000 times slower than the time to write data into a RAM. Some PROMs are erased with ultraviolet light and programmed with voltages, while electrically erasable PROM (EEPROM) are both erased and programmed with voltages. We cannot program ones into the ROM. We first erase the ROM, which puts ones into the entire memory, and then we program the zeros as needed. **Flash ROM** is a popular type of EEPROM. Each flash bit requires only two MOSFET transistors. The input (gate) of one transistor is electrically isolated, so if we trap charge on this input, it will remain there for years. The other transistor is used to read the bit by sensing whether or not the other transistor has trapped charge. In regular EEPROM, you can erase and program individual bytes. Flash ROM must be erased in large blocks. On many of Stellaris/Tiva family of microcontrollers, we can erase the entire ROM or just a 1024-byte block. Because flash is smaller than regular EEPROM, most microcontrollers have a large flash into which we store the software. For all the systems in this book, we will store instructions and constants in flash ROM and place variables and temporary data in static RAM.

**Checkpoint 1.46:** What are the differences between a microcomputer, a microprocessor, and a microcontroller?

**Checkpoint 1.47:** Which has a higher information density on the chip in bits per mm<sup>2</sup>: static RAM or flash ROM? Assume all MOSFETs are approximately the same size in mm<sup>2</sup>.

The external devices attached to the microcontroller provide functionality for the system. An **input port** is hardware on the microcontroller that allows information about the external world to be entered into the computer. The microcontroller also has hardware called an **output port** to send information out to the external world. Most of the pins shown in Figure 1.27 are input/output ports.

An **interface** is defined as the collection of the I/O port, external electronics, physical devices, and the software, which combine to allow the computer to communicate with the external world. An example of an input interface is a switch, where the operator toggles the switch, and the software can recognize the switch position. An example of an output interface is a light-emitting diode (LED), where the software can turn the light on and off, and the operator can see whether or not the light is shining. There is a wide range of possible inputs and outputs, which can exist in either digital or analog form. In general, we can classify I/O interfaces into four categories

**Parallel** - binary data are available simultaneously on a group of lines

**Serial** - binary data are available one bit at a time on a single line

**Analog** - data are encoded as an electrical voltage, current, or power

**Time** - data are encoded as a period, frequency, pulse width, or phase shift

# 1.8. Flowcharts and Structured Programming

The remainder of this chapter will discuss the art and science of designing embedded systems from a general perspective. If you need to write a paper, you decide on a theme, and then begin with an outline. In the same manner, if you design an embedded system, you define its specification (what it does) and begin with an organizational plan. In this chapter, we will present three graphical tools to describe the organization of an embedded system: flowcharts, data flow graphs, and call graphs. You should draw all three for every system you design. In this section, we introduce the flowchart syntax that will be used throughout the book. Programs themselves are written in a linear or one-dimensional fashion. In other words, we type one line of software after another in a sequential fashion. Writing programs this way is a natural process, because the computer itself usually executes the program in a top-to-bottom sequential fashion. This one-dimensional format is fine for simple programs, but conditional branching and function calls may create complex behaviors that are not easily observed in a linear fashion. Flowcharts are one way to describe software in a two-dimensional format, specifically providing convenient mechanisms to visualize conditional branching and function calls. Flowcharts are very useful in the initial design stage of a software system to define complex algorithms. Furthermore, flowcharts can be used in the final documentation stage of a project, once the system is operational, in order to assist in its use or modification.

Figures throughout this section illustrate the syntax used to draw flowcharts (Figure 1.28). The oval shapes define entry and exit points. The main **entry point** is the starting point of the software. Each function, or subroutine, also has an entry point. The **exit point** returns the flow of control back to the place from which the function was called. When the software runs continuously, as is typically the case in an embedded system, there will be no main exit point. We use rectangles to specify **process** blocks. In a high-level flowchart, a process block might involve many operations, but in a low-level flowchart, the exact operation is defined in the rectangle. The parallelogram will be used to define an **input/output** operation. Some flowchart artists use rectangles for both processes and input/output. Since input/output operations are an important part of embedded systems, we will use the parallelogram format, which will make it easier to identify input/output in our flowcharts. The diamond-shaped objects define a branch point or **conditional** block. Inside the diamond we can define what is being tested. Each arrow out of a condition block must be labeled with the condition causing flow to go in that direction. There must be at least two arrows out of a condition block, but there could be more than two. However, the condition for each arrow must be mutually exclusive (you can't say "if I'm happy go left and if I'm tall go right" because it is unclear what you want the software to do if I'm happy and tall). Furthermore, the complete set of conditions must define all possibilities (you can't say "if temperature is less than 20 go right and if the temperature is above 40 go left" because you have not defined what to do if the temperature is between 20 and 40). The rectangle with double lines on the side specifies a call to a **predefined function**. In this book, functions, subroutines, and procedures are terms that all refer to a well-defined section of code that performs a specific operation. Functions usually return a result parameter, while procedures usually do not. Functions and procedures are terms used when describing a high-level language, while subroutines are often used when describing assembly language. When a function (or subroutine or procedure) is called, the software execution path jumps to the function, the specific operation is

performed, and the execution path returns to the point immediately after the function call. Circles are used as **connectors**. A connector with an arrow pointing out of the circle defines a label or a spot in the algorithm. There should be one label connector for each number. Connectors with an arrow pointing into the circle are jumps or goto commands. When the flow reaches a goto connector, the execution path jumps to the position specified by the corresponding label connector. It is bad style to use a lot of connectors.

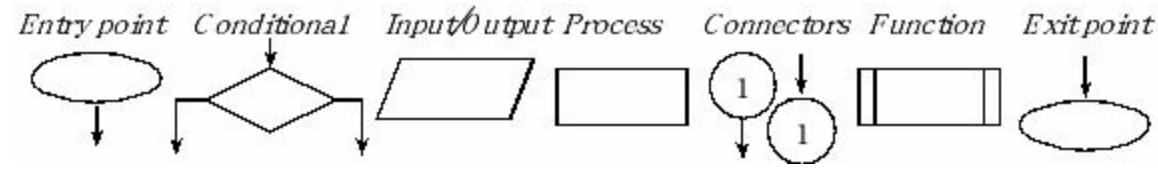


Figure 1.28. Flowchart symbols.

There are a seemingly unlimited number of tasks one can perform on a computer, and the key to developing great products is to select the correct ones. Just like hiking through the woods, we need to develop guidelines (like maps and trails) to keep us from getting lost. One of the fundamentals when developing software, regardless whether it is a microcontroller with 1000 lines of assembly code or a large computer system with billions of lines of code, is to maintain a consistent structure. One such framework is called **structured programming**. A good high-level language will force the programmer to write structured programs. Structured programs are built from three basic building blocks: the **sequence**, the **conditional**, and the **while-loop**. At the lowest level, the process block contains simple and well-defined commands. I/O functions are also low-level building blocks. Structured programming involves combining existing blocks into more complex structures, as shown in Figure 1.29.

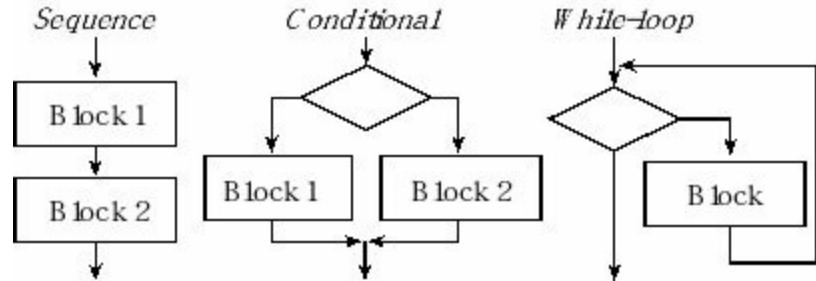


Figure 1.29. Flowchart showing the basic building blocks of structured programming.

---

**Example 1.1:** Using a flowchart describe the control algorithm that a toaster might use to cook toast. There will be a start button the user pushes to activate the machine. There is other input that measures toast temperature. The desired temperature is preprogrammed into the machine. The output is a heater, which can be on or off. The toast is automatically lowered into the oven when heat is applied and is ejected when the heat is turned off.

**Solution:** This example illustrates a common trait of an embedded system, that is, they perform the same set of tasks over and over forever. The program starts at **main** when power is applied, and the system behaves like a toaster until it is unplugged. Figure 1.30 shows a flowchart for one possible toaster algorithm. The system initially waits for the operator to push the start button. If the switch is not pressed, the system loops back reading and checking the switch over and over. After the start button is pressed, heat is turned on. When the toast temperature reaches the desired value, heat is turned off, and the process is repeated.

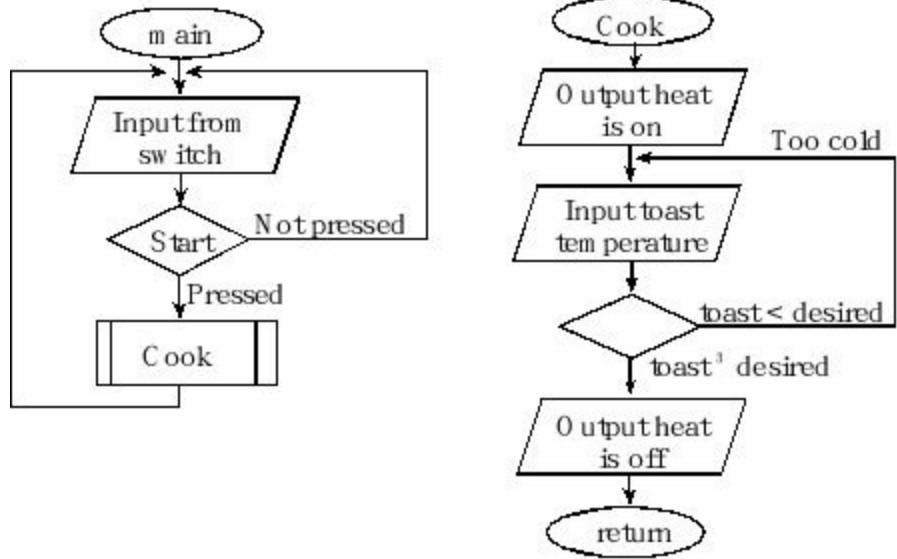


Figure 1.30. Flowchart illustrating the process of making toast.

**Safety tip:** When dealing with the potential for fire, you may want to add some safety features such as a time out or an independent check for temperature overflow.

**Observation:** The predefined functions in this chapter do not communicate any data between the calling routine and function. Data passed into a function are called input parameters, and data passed from the function back to the calling routine are called output parameters.

**Observation:** Notice in Figure 1.30 we defined a function **Cook** even though it was called from only one place. You might be tempted to think it would have been better to paste the code for the function into the one place it was called. There are many reasons it would be better to define the function as a separate software object: it will be easier to debug because there is a clear beginning and end of the function, it will make the overall system simpler to understand, and in the future we may wish to reuse this function for another purpose.

---

**Example 1.2.** The system has one input and one output. An event should be recognized when the input goes from 0 to 1 and back to 0 again. The output is initially 0, but should go 1 after four events are detected. After this point, the output should remain 1. Design a flowchart to solve this problem.

**Solution:** This example also illustrates the concept of a subroutine. We break a complex system into smaller components so that the system is easier to understand and easier to test. In particular, once we know how to detect an event, we will encapsulate that process into a subroutine, called **Event**. In this example, the **main** program first sets the output to zero, calls the function **Event** four times, then it sets the output to one. To detect the 0 to 1 to 0 edges in the input, it first waits for 1, and then it waits for 0 again. See Figure 1.31.

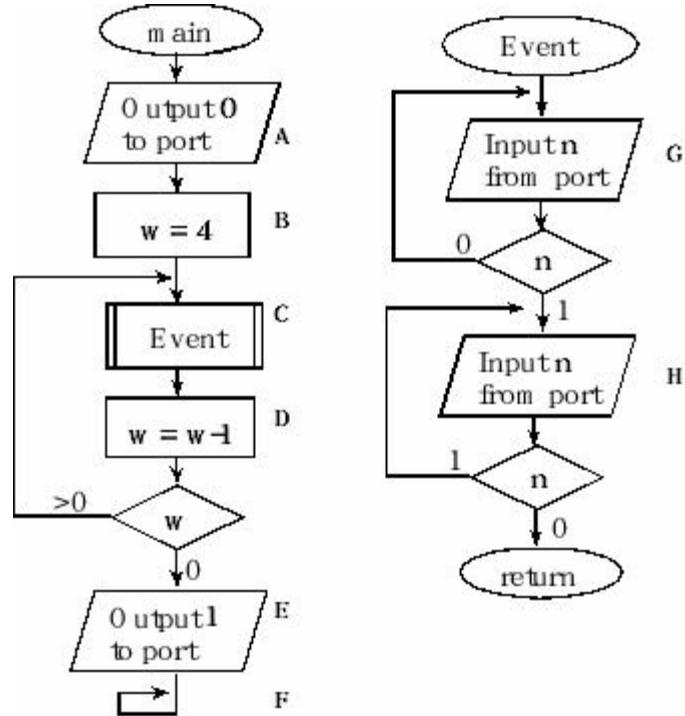
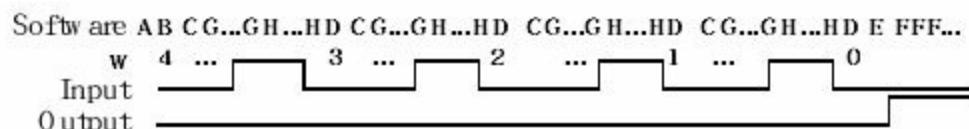


Figure 1.31. Flowchart illustrating the process waiting for four events.

The letters **A** through **H** in Figure 1.31 specify the software activities in this simple example. In this example, execution is sequential and predictable.



# 1.9. Concurrent and Parallel Programming

Many problems cannot be implemented using the single-threaded execution pattern described in the previous section. **Parallel programming** allows the computer to execute multiple threads at the same time. A computer with a multi-core processor can simultaneously execute a separate program in each of its cores. Fork and join are the fundamental building blocks of parallel programming. After a **fork**, two or more software threads will be run in parallel, i.e., the threads will run simultaneously on separate processors. Two or more simultaneous software threads can be combined into one using a **join** (Figure 1.32). Software execution after the join will wait until all threads above the join are complete. As an analogy, if I want to dig a big hole in my back yard, I will invite three friends over and give everyone a shovel. The fork operation changes the situation from me working alone to four of us ready to dig. The four digging tasks are run in parallel. The four diggers do not have to be performing the exact same task, but they operate simultaneously and they cooperate towards a common goal. When the overall task is complete, the join operation causes the friends to go away, and I am working alone again. A complex system may employ multiple microcontrollers, each running its own software. We classify this configuration as parallel or **distributed programming**.

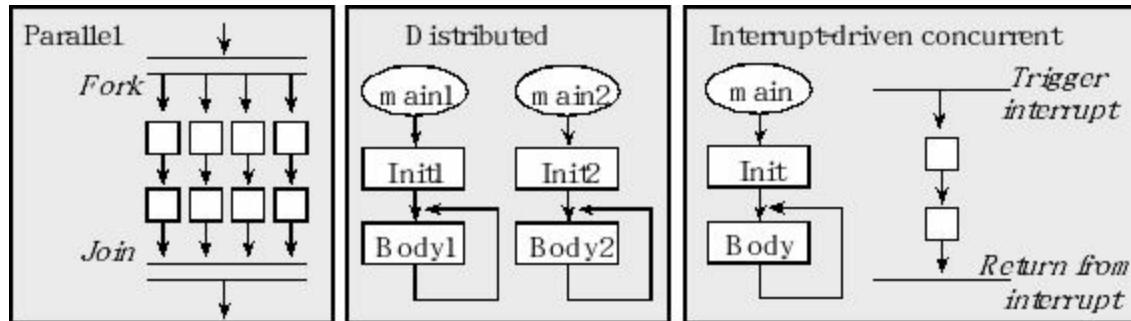


Figure 1.32. Flowchart symbols to describe parallel, distributed, and concurrent programming.

**Concurrent programming** allows the computer to execute multiple threads, but only one at a time. Interrupts are one mechanism to implement concurrency on real-time systems. Interrupts have a hardware trigger and a software action. An interrupt is a parameter-less subroutine call, triggered by a hardware event. The flowchart symbols for interrupts are also shown in Figure 1.32. The trigger is a hardware event signaling it is time to do something. Examples of interrupt triggers we will see in this book include new input data has arrived, output device is idle, and periodic event. The second component of an interrupt-driven system is the software action called an **interrupt service routine** (ISR). The **foreground** thread is defined as the execution of the main program, and the **background** threads are executions of the ISRs. Consider the analogy of sitting in a comfy chair reading a book. Reading a book is like executing the main program in the foreground. You start reading at the beginning of the book and basically read one page at time in a sequential fashion. You might jump to the back and look something up in the glossary, then jump back to where you were, which is analogous to a function call. Similarly, you might read the same page a few times, which is analogous to a program loop. Even though you skip around a little, the order of pages you read follows a logical and well-defined sequence. Conversely, if the telephone rings, you place a bookmark in the book, and answer the phone. When you are finished with the phone conversation, you hang up the phone and

continue reading in the book where you left off. The ringing phone is analogous to hardware trigger and the phone conversation is like executing the ISR.

---

**Example 1.3.** Design a flowchart for a system that performs two independent tasks. The first task is to output a 20 kHz square wave on **PORTA** in real time (period is 50  $\mu$ s). The second task is to read a value from **PORTB**, divide the value by 4, add 12, and output the result on **PORTD**. This second task is repeated over and over.

**Solution:** In this example, there are two threads: foreground and background, see Figure 1.33. Real time means the **PORTA** must change every 25  $\mu$ s. Therefore, we will use a periodic interrupt to guarantee this real-time requirement. In particular, the timer system will be configured so that a hardware trigger will occur every 25  $\mu$ s, and the software action will toggle **PORTA**. Toggle means, if it is 1 make it 0, and if it is 0 make it one. On the computer we will toggle by reading the port, performing an exclusive or with 1, and then writing it back to the port. The background thread creates the square wave in real time because the 0-to-1 and 1-to-0 edges occur (almost) exactly every 25  $\mu$ s. Tasks that are not time-critical can be performed in the foreground by the main program. In this example, the foreground thread inputs from **PORTB**, performs a calculation, and outputs to **PORTD**. Because both threads are active at the same time, we say the system is multi-threaded and the threads are running concurrently. Even though we will learn C later in the book, a simplified C program is given.

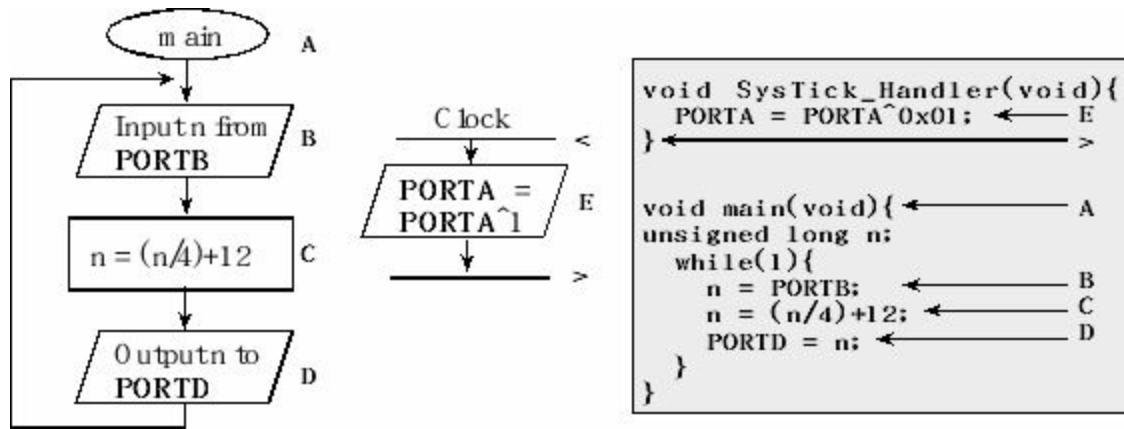
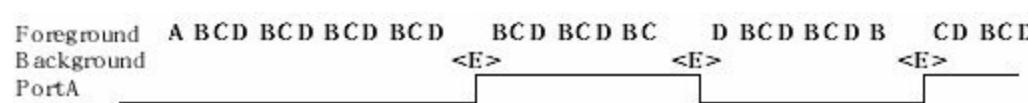


Figure 1.33. Flowchart for a multi-threaded solution of a system performing two tasks.

The letters ( **A - E** ) in Figure 1.33 specify the software activities in this multi-threaded example. In particular, **main** is executed in the foreground. In the foreground, execution is sequential and predictable (if **C** is to occur, it will come after **B** and before **D**.) On the other hand, with interrupts, the hardware trigger causes the interrupt service routine to execute. The symbol < signifies the hardware halting the main program and launching the ISR. The symbol > signifies the ISR software executing a **return from interrupt** instruction, which resumes execution in the main program. The execution of the ISR is predictable too; in this case it is executed every 25  $\mu$ s, but ISR execution does not depend on execution in the foreground. The execution sequence of this two-threaded system might be something like the following



In a single processor system like the ARM Cortex M processor, the interrupt must suspend foreground execution, execute the interrupt service routine in the background, and then resume execution of the foreground. The main program executes the sequence **BCD** over and over as it searches for prime numbers. In this example, the periodic timer causes the execution of **E** every 25 µs. Even though **C** will come after **B** and before **D**, interrupts may or may not inject a **<E>** between any two instructions of the foreground thread. Being able to inject an **E** exactly every 25 µs is how the real-time constraint is satisfied.

To illustrate the concept of parallel programming, consider the problem of finding the maximum value in a buffer, as implemented in Figure 1.34. Finding the maximum value in the first half of the buffer can be executed in parallel with finding the maximum value in the second half of the buffer. Although the ARM Cortex M microcontroller cannot execute software tasks in parallel, most experts believe the market share for multicore processors in the embedded field will increase over time. State-of-the-art microprocessors found in desktop computers have two or more cores, which do support parallel program execution. It is important to distinguish parallel programming like Figure 1.34, from multi-threading like Figure 1.33. Multi-threading, as we will be developing in this book, switches among multiple software tasks executing one task at a time.

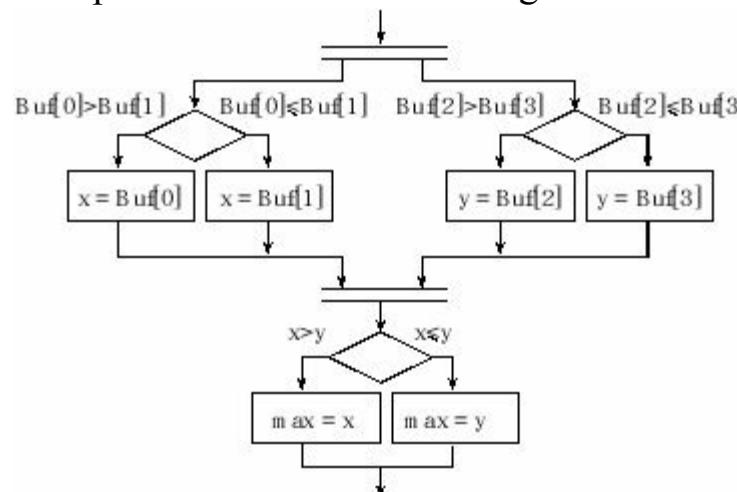


Figure 1.34. Parallel programming solution for finding the maximum value in a buffer.

# 1.10. Exercises

**1.1** In 16 words or less give definitions of the following terms

- |            |            |          |
|------------|------------|----------|
| a) Voltage | b) Current | c) Power |
| d) Energy  | e) KCL     | f) KVL   |

**1.2** In 16 words or less give definitions of the following terms

- |                      |                      |                 |
|----------------------|----------------------|-----------------|
| a) Positive logic    | b) CMOS              | c) Byte         |
| d) p-type transistor | e) n-type transistor | f) Exclusive or |

**1.3** Give two examples of these laws applied to Boolean Logic (one for AND one for OR)

- |                    |                  |                        |
|--------------------|------------------|------------------------|
| a) Commutative Law | b) Associate Law | c) Distributive Law    |
| d) Identity of 0   | e) Identity of 1 | f) De Morgan's Theorem |

**1.4** In 16 words or less give definitions of the following terms

- |              |             |                   |
|--------------|-------------|-------------------|
| a) Flip-flop | b) HiZ      | c) Open collector |
| d) Tristate  | e) Register | f) Bus            |

**1.5** In 16 words or less give definitions of the following terms

- |                |                 |         |
|----------------|-----------------|---------|
| a) Memory read | b) Memory write | c) ROM  |
| d) Volatile    | e) RAM          | f) Port |

**1.6** In 16 words or less give definitions of the following terms

- |                          |             |                         |
|--------------------------|-------------|-------------------------|
| a) Interrupt programming | b) Function | c) Structured           |
| d) Join                  | e) Fork     | f) Parallel programming |

**1.7** Fill in this table with the equivalent resistance (all values are in ohms)

R1	R2	R1 in series with R2	R1 in parallel with R2
1000	1000		
1000	10000		
1000		4000	
1000			800
	5000		1200

**1.8** Fill in this table with the equivalent resistance (all values are in ohms)

R1	R2	R1 in series with R2	R1 in parallel with R2
100	200		

100	0
100	700
100	75
200	50

**1.9** 10 V is applied across the series combination of a  $1000\Omega$  and a  $2000\Omega$  resistor. What is the voltage across the  $2000\Omega$  resistor? What is the current through the  $2000\Omega$  resistor?

**1.10** 10 V is applied across the parallel combination of a  $1000\Omega$  and a  $2000\Omega$  resistor. What is the voltage across the  $2000\Omega$  resistor? What is the current through the  $2000\Omega$  resistor?

**1.11** What is the fewest number of bits than can represent all the numbers from 0 to 100?

**1.12** Assuming we have a 4-bit number system, what numbers can we represent (starting is 0)?

**1.13** How many binary bits does it take to represent 10,000,000? How many bytes? Using the fact that  $2^{10}$  is about  $10^3$ , it is possible to solve this question without a calculator.

**1.14** How many binary bits does it take to represent 100,000,000,000? How many bytes? Using the fact that  $2^{10}$  is about  $10^3$ , it is possible to solve this question without a calculator.

**1.15** In C99, an **int8\_t** is 8 bits, an **int16\_t** is 16-bits, and an **int32\_t** is 32 bits. Assuming each is signed, give the range of each type of number.

**1.16** In C99, an **uint8\_t** is 8 bits, an **uint16\_t** is 16-bits, and an **uint32\_t** is 32 bits. Assuming each is unsigned, give the range of each type of number.

**1.17** How many binary bits is  $2^{3/4}$  decimal digits?

**1.18** About how many decimal digits is 20 binary bits?

**1.19** Each row of the following table is to contain an equal value expressed in binary, hexadecimal, and decimal. Complete the missing values. Assume the decimal values are unsigned. The first row illustrates the process.

binary	hexadecimal	decimal
$01101001_2$	0x69	105
	0x46	
		47
$10001110_2$		
	0xE5	
		95
$111001001110_2$		
	0x02B9	
		10000

**1.20** Each row of the following table is to contain an equal value expressed in binary, hexadecimal, and decimal. Complete the missing values. Assume the decimal values are unsigned. The first row illustrates the process.

binary	hexadecimal	decimal
$10101101_2$	0xAD	173
	0x58	
		143
$11011_2$		
	0x1554	
		26
$100010010111101_2$		
	0x24A6	
		14321

**1.21** You know that the 8-bit hexadecimal representation for -1 is 0xFF. Use this fact and count backwards to quickly find the hexadecimal representations of -2, -3, and -4.

**1.22** You know that the 16-bit hexadecimal representation for -1 is 0xFFFF. Use this fact and count backwards to quickly find the hexadecimal representations of -2, -3, and -4.

**1.23** You know that the 32-bit hexadecimal representation for -1 is 0xFFFFFFFF. Use this fact and count backwards to quickly find the hexadecimal representations of -2, -3, and -4.

**1.24** Each row of the following table is to contain an equal value expressed in binary, hexadecimal, and decimal. Complete the missing values. Assume each value is 8 bits and the decimal numbers are signed. The first row illustrates the process.

binary	hexadecimal	decimal
$01011110_2$	0x5E	94
	0xD2	
		-67
$11001011_2$		
	0xE1	
		79
$00101010_2$		
	0xC7	
		-101

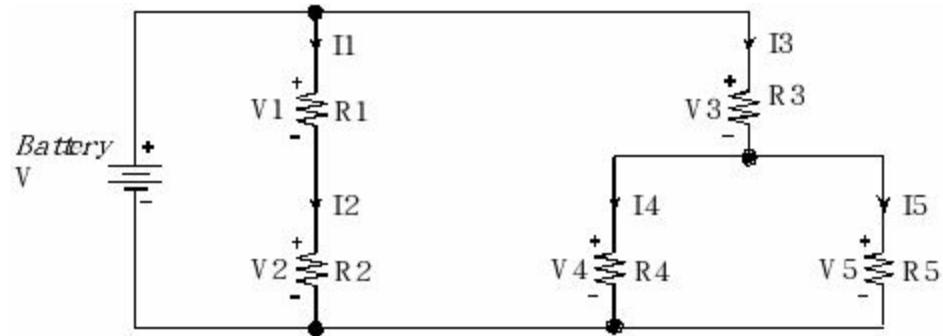
**1.25** Each row of the following table is to contain an equal value expressed in binary, hexadecimal, and decimal. Complete the missing values. Assume each value is 8 bits and the decimal numbers are signed. The first row illustrates the process.

binary	hexadecimal	decimal

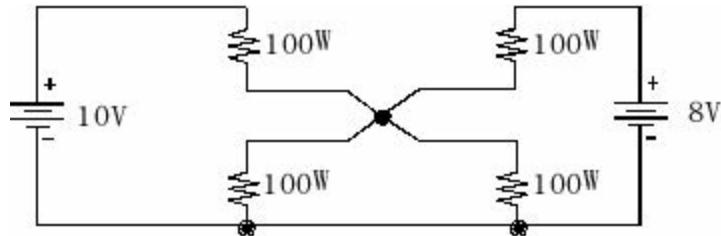
$11111110_2$	0xFE	-2
	0xFD	-82
$00110011_2$	0xA4	51
$11000000_2$	0x22	-121

**D1.26** Find the currents ( $I_1, I_2, I_3, I_4$ , and  $I_5$ ) and voltages ( $V_1, V_2, V_3, V_4$  and  $V_5$ ). You can give answer to three significant figures (e.g., 0.0123A, 12.3mA, 1.23V, 1230mV)

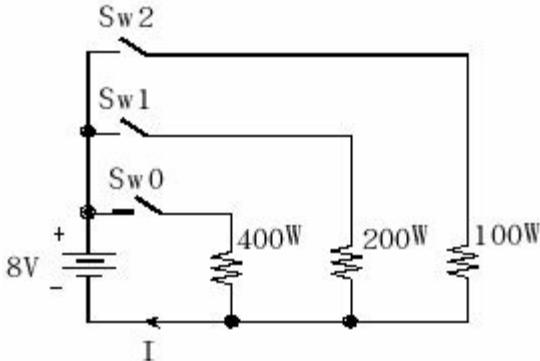
- a) Battery voltage is 10 V and all resistors are  $1 \text{ k}\Omega$ .
- b) Battery voltage is 4 V and  $R_1=1 \text{ k}\Omega$   $R_2=4 \text{ k}\Omega$    $R_3=1 \text{ k}\Omega$    $R_4=2 \text{ k}\Omega$   and  $R_5=2 \text{ k}\Omega$ .
- c) Battery voltage is 2 V and  $R_1=1 \text{ k}\Omega$   $R_2=4 \text{ k}\Omega$    $R_3=1 \text{ k}\Omega$    $R_4=2 \text{ k}\Omega$   and  $R_5=2 \text{ k}\Omega$ .
- d) Battery voltage is 5 V and  $R_1=1 \text{ k}\Omega$   $R_2=9 \text{ k}\Omega$    $R_3=0 \text{ k}\Omega$    $R_4=2 \text{ k}\Omega$   and  $R_5=\infty \text{ k}\Omega$ .



**D1.27** Use Ohm's Law and KCL to solve for the voltage in the middle of the resistors



**D1.28** Consider this 3-bit digital to analog converter. We define the logic state of each switch as 0 or 1, where 0 means not pushed and 1 means pushed. Define a 3-bit number  $n$  (0 to 7) which specifies the three switch positions.  $n = 0$  means none are pushed.  $n = 1$  means  $\text{Sw}_0$  is pushed.  $n = 2$  means  $\text{Sw}_1$  is pushed.  $n = 3$  means  $\text{Sw}_1$  and  $\text{Sw}_0$  are pushed.  $n = 4$  means  $\text{Sw}_2$  is pushed.  $n = 5$  means  $\text{Sw}_2$  and  $\text{Sw}_0$  are pushed.  $n = 6$  means  $\text{Sw}_2$  and  $\text{Sw}_1$  are pushed.  $n = 7$  means all are pushed. Derive a relationship between the current  $I$  and the number  $n$ .



**D1.29** Build a 2-bit decoder, which has two inputs A, B and four outputs C0, C1, C2, C3. This decoder has exactly one output which is true. That true output is specified by the binary value represented by the two inputs. Build it with AND OR and NOT gates as shown in Section 1.2.

A	B	C0	C1	C2	C3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

**D1.30** Build a 3-bit decoder, which has three inputs A, B, C and eight outputs C0–C7. This decoder has exactly one output which is true. That true output is specified by the binary value represented by the three inputs. Build it with AND OR and NOT gates as shown in Section 1.2.

A	B	C	C0	C1	C2	C3	C4	C5	C6	C7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

**D1.31** Build a 1-bit selector or multiplexer, which has three inputs A, B, S and one output O. If the S signal is 0, then the output O equals A. If the S signal is 1, then the output O equals B. Build it with AND OR and NOT gates as shown in Section 1.2.

S	A	B	O
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

**D1.32** In order to reduce power, some microcomputers run on 2.5 V instead of 3.3 V. Redraw Figure 1.1 using 2.5 V power, and define what logic high and logic low would be for this system. Assume the resistance path from the +3.3V supply to ground for +3.3V logic is approximately equal to the resistance from the +2.5V supply to ground for +2.5V logic. What is the percentage reduction in power occurring by switching from +3.3V to +2.5V?

**D1.33** Redraw the circuit in D1.28 using just  $200\ \Omega$  resistors (use as many as you need.)

**D1.34** Redesign the circuit in D1.28 using 4 switches. This digital to analog circuit will convert a 4-bit number  $n$  into an output current  $I$ .

**D1.35** Modify the transistor level circuit in Figure 1.13 to implement a 3-input NAND. The output will be low if and only if all three inputs are high. I.e., design it with p-type and n-type MOS transistors.

**D1.36** Modify the transistor level circuit in Figure 1.13 to implement a 3-input NOR. The output will be high if and only if all three inputs are low. I.e., design it with p-type and n-type MOS transistors.

**D1.37** Design a two-input AND function using just 2-input NOR gates. You can use as many NOR gates as you need.

**D1.38** Design a two-input OR function using just 2-input NAND gates. You can use as many NAND gates as you need.

**D1.39** Design a transistor level implementation of an exclusive OR gate.

**D1.40** Using a flowchart, describe the control algorithm that a thermostat uses to maintain constant temperature. Assume the inputs are current temperature in F, the desired temperature in F, and an AC/off/heat three-way switch. The outputs are AC (on/off) and heat (on/off).

**D1.41** Using a flowchart, describe the cruise control algorithm that a car uses to maintain constant speed. Assume the inputs are current speed in mph, brake (on/off), and a cruise on/off momentary button. The output is accelerator position (0 to 100%). The desired current is the current speed at the time the cruise control is activated. Touching the brake turns off the system.

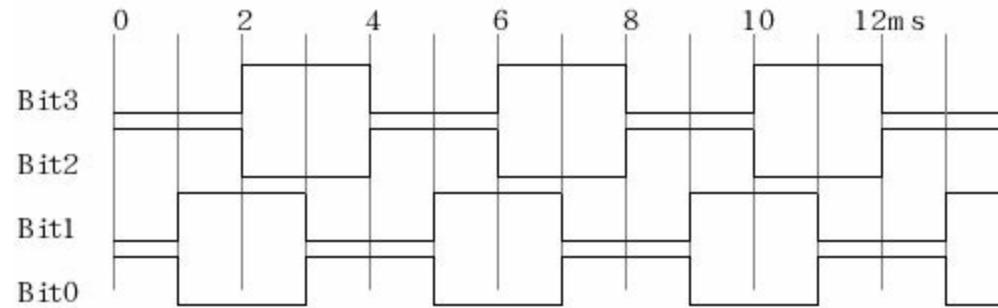
**D1.42** Draw a flowchart for an incremental motor controller. There is an input that specifies the motor speed in RPM (range is 0 to 255 RPM). The desired speed is 100 RPM. There is an output that the software can use to adjust power to the motor. This output must be a number between 0 and 100. Do not attempt to output numbers less than 0 or more than 100. An incremental controller is a simple algorithm. If the speed is too slow and the output is less than 100, then increase the output by one. If the speed is too fast and the output is greater than 0, then decrease the output by one.

**D1.43** Draw a flowchart for a stepper motor controller. There is one binary input that is true if the operator wishes to spin the motor and false if the operator wishes the motor to stop. There is a 4-bit output connected to the motor. To spin the motor, the sequence 5, 6, 10, 9 is output over and over. To stop the motor, simply cease to perform outputs (leave the output at which ever number you left off at: 5, 6, 10, or 9).

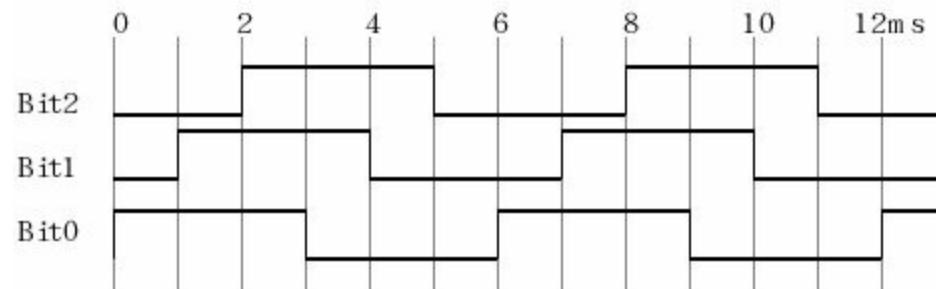
**D1.44** Draw a flowchart for an algorithm to determine which rising edge occurs first. There are two binary inputs: In1 and In2. Each input can be 0 or 1. There are two binary outputs: Out1 and Out2, one corresponding to each input. Initially both outputs are 0. If In1 goes from 0 to 1 before In2, then set Out1 to 1. If In2 goes from 0 to 1 before In1, then set Out2 to 1. Your answer should operate properly as long as both signals do not rise at the same time or nearly the same time. No matter how complicated you draw the flowchart it will have a minor flaw. Trying to solve this kind of problem in software creates a **race condition**. A race occurs when both signals rise very close to each other, it is possible in a tiny fraction of the cases for Out2 to be set even when In1 rises first. Put A B C ... labels on your flowchart and give one scenario resulting in the improper output.

**D1.45** Draw two flowcharts that count the number of rising edges (0 to 1) on an input port. At the beginning of the algorithm, set the counter to 0. At each rising edge, increment the counter. In the first flowchart, do not use interrupts (wait for zero, wait for 1). In the second flowchart, assume there is an input-triggered interrupt that occurs on the rising edge of the input.

**D1.46** Draw a flowchart that implements the following 4-bit output sequence. The sequence can spin a stepper motor. The output changes every 1ms. You may assume there is a periodic interrupt that triggers every 1 ms. This interrupt rate will determine the speed of the motor, which will be fixed in this example. Considering the 4-bit output as one number the sequence is 5, 6, 10, 9, 5, 6, 10, 9 ... There are two good solutions. The simple solution uses three decision blocks. The best solution uses a software variable and the exclusive OR operation. The main program sets the output to 5, and initializes any variables you wish to use.



**D1.47** Draw a flowchart that implements the following 3-bit output sequence. This sequence can spin a **brushless DC motor**. In this simplified system, the output changes every 1ms (in a real brushless DC motor the interrupt times will depend on measurements of shaft position). You may assume there is a periodic interrupt that triggers every 1 ms. Considering the 3-bit output as one number the output sequence is 1, 3, 7, 6, 4, 0 repeated over and over. There are a number of good solutions. The simple solution uses five decision blocks. The main program sets the output to 1, and initializes any variables you wish to use.



# **2. Introduction to Embedded Systems**

## **Chapter 2 objectives are to:**

- Present an introduction to embedded systems
- Outline the basic steps in design
- Define data flow and call graphs as a design tool
- Present a short introduction to C programming

The objective of this chapter is to present the framework or fundamentals needed for system design in general, and embedded system design in particular. A system is a collection of components that are combined together to perform a single complicated task in a coordinated manner. Component is used here in a very broad sense including software, computer hardware, digital hardware, analog circuits, mechanical hardware, power supply and distribution, sensors, and actuators. A system is comprised from components and interfaces. In a recursive manner, we can design a component by connecting simpler components together with interfaces. Using this model, one can envision top-down design as the process beginning with one large component, and then subdividing each component into simpler subcomponents connected together with interfaces. This process completes when each component is so simple it can be built or purchased. Testing can then proceed in a bottom-up fashion by connecting actual components together with actual interfaces. Systems have structure defined by the components and interconnections. Systems have behavior. For an embedded system, behavior is embodied by the responses of its outputs to changes in its inputs. Both time and state are important factors in describing the behavior of an embedded system. Systems have interconnectivity. Examples of these interconnections include mechanical force (bolts and nuts), energy (power flowing out of a battery), information flow (communication channel), synchronization (if this happens, make that happen), and integration of information (yaw, pitch, and roll combine to define tilt angle). Systems have rules or assumptions of expected usage. For example, the designer of an automobile expects you to drive it on the road. This expected usage is often found as a list of constraints. If you learn two things from reading this chapter, let them be these: every project needs a well-written requirements document, and testing is something we must do at every stage of a project (beginning, middle, and end.)

## 2.1. Embedded Systems

To better understand the expression **embedded microcomputer system**, consider each word separately. In this context, the word “embedded” means hidden inside so one can’t see it. The term “micro” means small, and a “computer” contains a processor, memory, and a means to exchange data with the external world. The word “system” means multiple components interfaced together for a common purpose. Systems have structure, behavior, and interconnectivity operating in a framework bound by rules and regulations. Another name for embedded systems is **Cyber-Physical Systems**, introduced in 2006 by Helen Gill of the National Science Foundation, because these systems combine the intelligence of a computer with the physical objects of our world. In an embedded system, we use ROM for storing the software and fixed constant data and RAM for storing temporary information. Many microcomputers employed in embedded systems use Flash EEPROM, which is an electrically-erasable programmable ROM, because the information can easily be erased and reprogrammed. The functionality of a digital watch is defined by the software programmed into its ROM. When you remove the batteries from a watch and insert new batteries, it still behaves like a watch because the ROM is nonvolatile storage. As shown in Figure 2.1, the term embedded microcomputer system refers to a device that contains one or more microcomputers inside. **Microcontrollers**, which are microcomputers incorporating the processor, RAM, ROM and I/O ports into a single package, are often employed in an embedded system because of their low cost, small size, and low power requirements. Microcontrollers like the Texas Instruments TM4C family are available with a large number and wide variety of I/O devices, such as parallel ports, serial ports, timers, digital to analog converters (DAC), and analog to digital converters (ADC). The I/O devices are a crucial part of an embedded system, because they provide necessary functionality. The software together with the I/O ports and associated interface circuits give an embedded computer system its distinctive characteristics. Managing time, both as an input and an output, is a critical task. It is not only important to get the correct output, but to get the correct output at the proper time. The microcontrollers often must communicate with each other. How the system interacts with humans is often called the **human-computer interface (HCI)** or **man-machine interface (MMI)**.

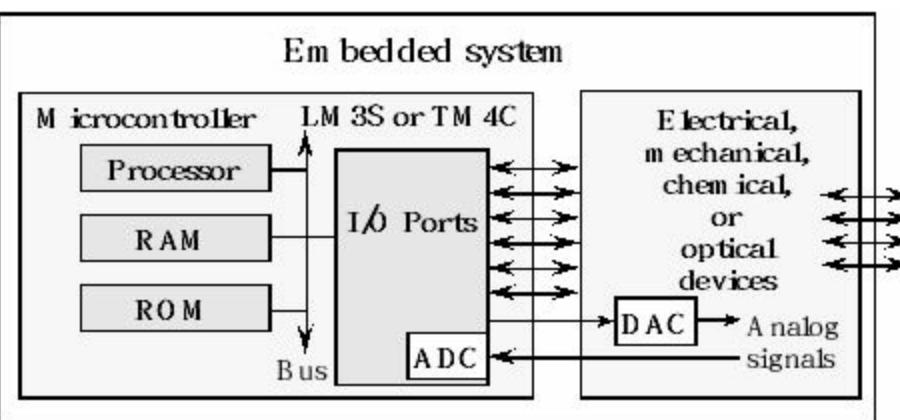


Figure 2.1. An embedded system includes a microcomputer interfaced to external physical devices.

**Checkpoint 2.1:** What is an embedded system?

A digital multimeter, as shown in Figure 2.2, is a typical embedded system. This embedded system has two inputs: the mode selection dial on the front and the red/black test probes. The output is a liquid crystal display (LCD) showing measured parameters. The large black chip inside the box is a microcontroller. The software that defines its very specific purpose is programmed into the ROM of the microcontroller. As you can see, there is not much else inside this box other than the microcontroller, a fuse, a rotary dial to select the mode, a few interfacing resistors, and a battery.

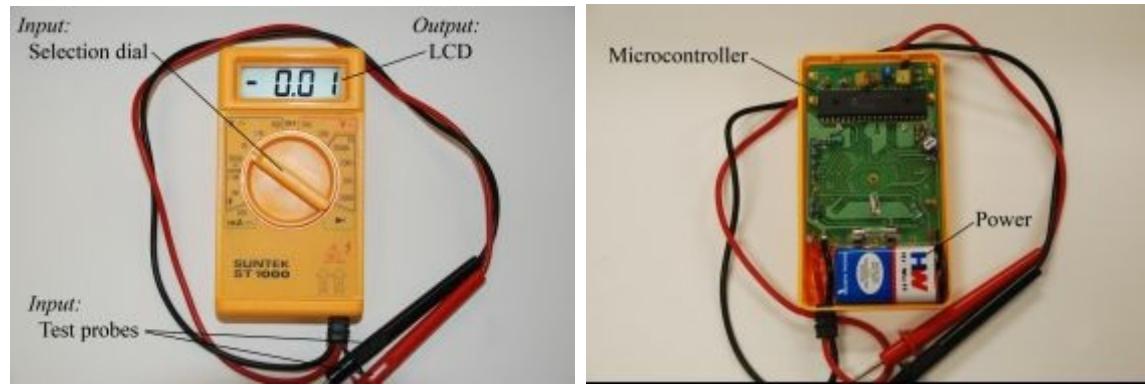


Figure 2.2. A digital multimeter contains a microcontroller programmed to measure voltage, current and resistance.

As defined previously, a microcomputer is a small computer. One typically restricts the term **embedded** to refer to systems that do not look and behave like a typical computer. Most embedded systems do not have a keyboard, a graphics display, or secondary storage (disk). There are two ways to develop embedded systems. The first technique uses a microcontroller, like the ARM Cortex M-series. In general, there is no operating system, so the entire software system is developed. These devices are suitable for low-cost, low-performance systems. Volume 3 will describe how to design a real-time operating system for the Cortex M family of microcontrollers. On the other hand, one can develop a high-performance embedded system around a more powerful microcontroller such as the ARM Cortex A-series. These systems typically employ an operating system and are first designed on a development platform, and then the software and hardware are migrated to a stand-alone embedded platform.

### **Checkpoint 2.2:** What is a microcomputer?

The external devices attached to the microcontroller allow the system to interact with its environment. An **interface** is defined as the hardware and software that combine to allow the computer to communicate with the external hardware. We must also learn how to interface a wide range of inputs and outputs that can exist in either digital or analog form. This first volume provides an introduction to microcomputer programming, hardware interfacing, and the design of embedded systems. Volume 2 of this series will focus on the details of hardware interfacing and system design. Volume 3 describes real-time operating systems and applies embedded system design to real-time data acquisition, digital signal processing, high-speed networks, and digital control systems. In general, we can classify I/O interfaces into parallel, serial, analog or time. Because of low cost, low power, and high performance, there has been and will continue to be an advantage of using time-encoded inputs and outputs.

A **device driver** is a set of software functions that facilitate the use of an I/O port. One of the simplest I/O ports on the Texas Instruments TM4C family is a parallel port or General Purpose Input/Output (GPIO). One such parallel port is Port A. The software will refer to this port using the name **GPIO\_PORTA\_DATA\_R**. Ports are a collection of pins, usually 8, which can be used for either input or output. If Port A is an input port, then when the software reads from **GPIO\_PORTA\_DATA\_R**, it gets eight bits (each bit is 1 or 0), representing the digital levels (high or low) that exist at the time of the read. If Port A is an output port, then when the software writes to **GPIO\_PORTA\_DATA\_R**, it sets the outputs on the eight pins high (1) or low (0), depending on the data value the software has written.

The other general concept involved in most embedded systems is they run in **real time**. In a real-time computer system, we can put an upper bound on the time required to perform the input-calculation-output sequence. A real-time system can guarantee a worst case upper bound on the response time between when the new input information becomes available and when that information is processed. This response time is called interface **latency**. Another real-time requirement that exists in many embedded systems is the execution of periodic tasks. A periodic task is one that must be performed at equal-time intervals. A real-time system can put a small and bounded limit on the time error between when a task should be run and when it is actually run. Because of the real-time nature of these systems, microcontrollers have a rich set of features to handle many aspects of time.

**Checkpoint 2.3:** An input device allows information to be entered into the computer. List some of the input devices available on a general purpose computer.

**Checkpoint 2.4:** An output device allows information to exit the computer. List some of the output devices available on a general purpose computer.

The embedded computer systems in these three volumes will contain an ARM Cortex M microcontroller, which will be programmed to perform a specific dedicated application. Software for embedded systems typically solves only a limited range of problems. The microcomputer is embedded or hidden inside the device. In an embedded system, the software is usually programmed into ROM and therefore fixed. Even so, **software maintenance** (e.g., verification of proper operation, updates, fixing bugs, adding features, extending to new applications, end user configurations) is still extremely important. In fact, because microcomputers are employed in many safety-critical devices, injury or death may result if there are hardware and/or software faults. Consequently, testing must be considered in the original design, during development of intermediate components, and in the final product. The role of simulation is becoming increasingly important in today's market place as we race to build better and better machines with shorter and shorter design cycles. An effective approach to building embedded systems is to first design the system using a hardware/software simulator, then download and test the system on an actual microcontroller.

## 2.2. Applications Involving Embedded Systems

An **embedded computer system** includes a microcomputer with mechanical, chemical, and electrical devices attached to it, programmed for a specific dedicated purpose, and packaged up as a complete system. Any electrical, mechanical, or chemical system that involves inputs, decisions, calculations, analyses, and outputs is a candidate for implementation as an embedded system. Electrical, mechanical, and chemical sensors collect information. Electronic interfaces convert the sensor signals into a form acceptable for the microcomputer. For example, a **tachometer** is a sensor that measures the revolutions per second of a rotating shaft. Microcomputer software performs the necessary decisions, calculations, and analyses. Additional interface electronics convert the microcomputer outputs into the necessary form. **Actuators** can be used to create mechanical or chemical outputs. For example, an electrical motor converts electrical power into mechanical power.

**Checkpoint 2.5:** There is a microcomputer embedded in a digital watch. List three operations the software must perform.

In contrast, a general-purpose computer system typically has a keyboard, disk, and graphics display and can be programmed for a wide variety of purposes. Typical general-purpose applications include word processing, electronic mail, business accounting, scientific computing, and data base systems. The user of a general-purpose computer does have access to the software that controls the machine. In other words, the user decides which operating system to run and which applications to launch. Because the general-purpose computer has a removable disk or network interface, new programs can easily be added to the system. The most common type of general-purpose computer is the **personal computer**, e.g., the Apple MacBook, costing less than \$3,000. Computers more powerful than the personal computer can be grouped in the **workstation** category, ranging from \$3,000 to \$50,000 range. **Supercomputers** cost above \$50,000. These computers often employ multiple processors and have much more memory than the typical personal computer. The workstations and supercomputers are used for handling large amounts of information (business applications) or performing large calculations (scientific research.) This book will not specifically cover the general-purpose computer, although many of the basic principles of embedded computers do apply to all types of computer systems.

**Observation:** Since the advent of the personal computer in 1983, there have always been three classes of computer systems: personal, workstation and supercomputer. While the computational power of each class has grown by orders of magnitude, the price range of each has remained relatively constant.

A cell phone has five or more microcontrollers. Automobiles employ dozens of microcontrollers. In fact, upscale homes contain hundreds of microcontrollers, and the average consumer now interacts with microcontrollers thousands of times each day. Embedded microcomputers impact virtually all aspects of daily life (Table 2.1):

- Consumer Electronics
- Home

- Communications
- Automotive
- Military
- Industrial
- Business
- Shipping
- Medical
- Computer components

Functions performed by the microcontroller

### **Consumer/Home:**

Washing machine energy	Controls the water and spin cycles, saving water and energy
Exercise equipment rate	Measures speed, distance, acceleration, calories, heart rate
Remote controls interact with user	Accepts key touches, sends infrared pulses, learns how to interact
Clocks and watches	Maintains the time, alarm, and display
Games and toys	Entertains the user, joystick input, video output
Audio/video with sounds and pictures	Interacts with the operator, enhances performance
Set-back thermostats	Adjusts day/night thresholds saving energy

### **Communication:**

Answering machines	Plays outgoing messages and saves incoming messages
Telephone system	Switches signals and retrieves information
Cellular phones	Interacts with key pad, microphone, and speaker
Satellites	Sends and receives messages

### **Automotive:**

Automatic braking	Optimizes stopping on slippery surfaces
Noise cancellation	Improves sound quality, removing noise
Theft deterrent devices	Allows keyless entry, controls alarm
Electronic ignition	Controls sparks and fuel injectors
Windows and seats	Remembers preferred settings for each driver
Instrumentation	Collects and provides necessary information

### **Military:**

Smart weapons	Recognizes friendly targets
Missile guidance	Directs ordnance at the desired target

Global positioning	Determines where you are on the planet, suggests paths,
coordinates troops	
Surveillance	Collects information about enemy activities
<b>Industrial/Business/Shipping:</b>	
Point-of-sale systems	Accepts inputs and manages money, keeps credit
information secure	
Temperature control	Adjusts heating and cooling to maintain temperature
Robot systems	Inputs from sensors, controls the motors improving
productivity	
Inventory systems	Reads and prints labels, maximizing profit, minimizing
shipping delay	
Automatic sprinklers	Controls the wetness of the soil maximizing plant
growth	
<b>Medical:</b>	
Infant apnea monitors	Detects breathing, alarms if stopped
Cardiac monitors	Measures heart function, alarms if problem
Cancer treatments	Controls doses of radiation, drugs, or heat
Prosthetic devices	Increases mobility for the handicapped
Medical records	Collect, organize, and present medical information
<b>Computer Components:</b>	
Mouse	Translates hand movements into commands for the
main computer	
USB flash drive	Facilitates the storage and retrieval of information
Keyboard	Accepts key strokes, decodes them, and transmits to the
main computer	

**Table 2.1. Products involving embedded systems.**

## 2.3. Product Life Cycle

In this section, we will introduce the product development process in general. The basic approach is introduced here, and the details of these concepts will be presented throughout the remaining chapters of the book. As we learn software/hardware development tools and techniques, we can place them into the framework presented in this section. As illustrated in Figure 2.3, the development of a product follows an analysis-design-implementation-testing-deployment cycle. For complex systems with long life-spans, we transverse multiple times around the life cycle. For simple systems, a one-time pass may suffice.

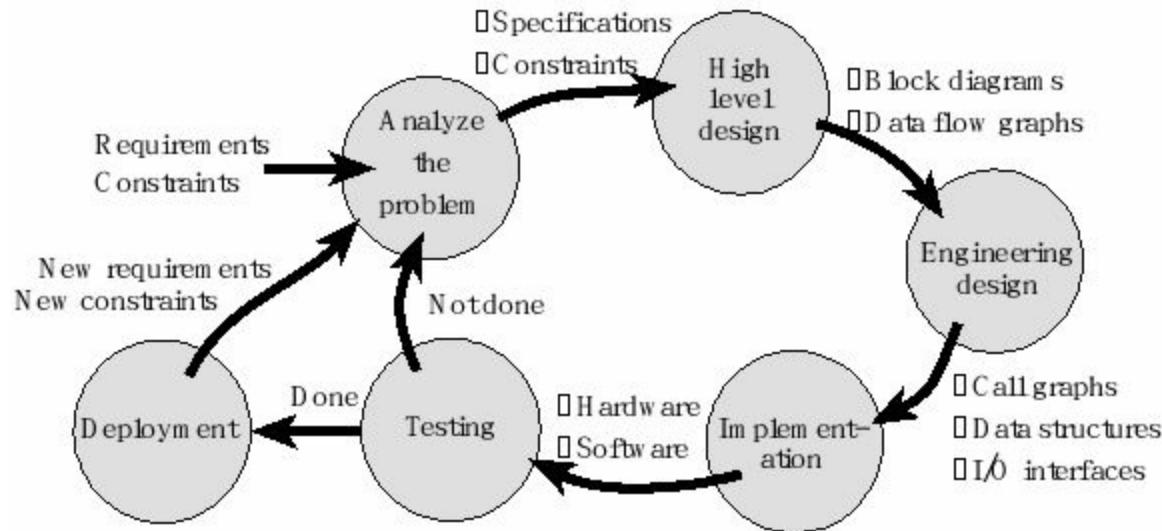


Figure 2.3. Product life cycle.

During the **analysis phase**, we discover the requirements and constraints for our proposed system. We can hire consultants and interview potential customers in order to gather this critical information. A **requirement** is a specific parameter that the system must satisfy. We begin by rewriting the system requirements, which are usually written in general form, into a list of detailed **specifications**. In general, specifications are detailed parameters describing how the system should work. For example, a requirement may state that the system should fit into a pocket, whereas a specification would give the exact size and weight of the device. For example, suppose we wish to build a motor controller. During the analysis phase, we would determine obvious specifications such as range, stability, accuracy, and response time. There may be less obvious requirements to satisfy, such as weight, size, battery life, product life, ease of operation, display readability, and reliability. Often, improving the performance on one parameter can be achieved only by decreasing the performance of another. This art of compromise defines the tradeoffs an engineer must make when designing a product. A **constraint** is a limitation, within which the system must operate. The system may be constrained to such factors as cost, safety, compatibility with other products, use of specific electronic and mechanical parts as other devices, interfaces with other instruments and test equipment, and development schedule. The following measures are often considered during the analysis phase of a project:

**Safety:** The risk to humans or the environment

**Accuracy:** The difference between the expected truth and the actual parameter

**Precision:** The number of distinguishable measurements

**Resolution:** The smallest change that can be reliably detected

**Response time:** The time between a triggering event and the resulting action

**Bandwidth:** The amount of information processed per time

**Maintainability:** The flexibility with which the device can be modified

**Testability:** The ease with which proper operation of the device can be verified

**Compatibility:** The conformance of the device to existing standards

**Mean time between failure:** The reliability of the device, the life of a product

**Size and weight:** The physical space required by the system

**Power:** The amount of energy it takes to operate the system

**Nonrecurring engineering cost (NRE cost):** The one-time cost to design and test

**Unit cost:** The cost required to manufacture one additional product

**Time-to-prototype:** The time required to design, build, and test an example system

**Time-to-market:** The time required to deliver the product to the customer

**Human factors:** The degree to which our customers like/appreciate the product

### **Checkpoint 2.6:** What's the difference between a requirement and a specification?

The following is one possible outline of a **Requirements Document**. IEEE publishes a number of templates that can be used to define a project (IEEE STD 830-1998). A requirements document states what the system will do. It does not state how the system will do it. The main purpose of a requirements document is to serve as an agreement between you and your clients describing what the system will do. This agreement can become a legally binding contract. Write the document so that it is easy to read and understand by others. It should be unambiguous, complete, verifiable, and modifiable.

## 1. Overview

- 1.1. Objectives: Why are we doing this project? What is the purpose?
- 1.2. Process: How will the project be developed?
- 1.3. Roles and Responsibilities: Who will do what? Who are the clients?
- 1.4. Interactions with Existing Systems: How will it fit in?
- 1.5. Terminology: Define terms used in the document.
- 1.6. Security: How will intellectual property be managed?

## 2. Function Description

- 2.1. Functionality: What will the system do precisely?
- 2.2. Scope: List the phases and what will be delivered in each phase.
- 2.3. Prototypes: How will intermediate progress be demonstrated?
- 2.4. Performance: Define the measures and describe how they will be determined.

2.5. Usability: Describe the interfaces. Be quantitative if possible.

2.6. Safety: Explain any safety requirements and how they will be measured.

### 3. Deliverables

3.1. Reports: How will the system be described?

3.2. Audits: How will the clients evaluate progress?

3.3. Outcomes: What are the deliverables? How do we know when it is done?

**Observation:** To build a system without a requirements document means you are never wrong, but never done.

During the **high-level design** phase, we build a conceptual model of the hardware/software system. It is in this model that we exploit as much abstraction as appropriate. The project is broken into modules or subcomponents. Modular design will be presented in Chapter 5. During this phase, we estimate the cost, schedule, and expected performance of the system. At this point we can decide if the project has a high enough potential for profit. A **data flow graph** is a block diagram of the system, showing the flow of information. Arrows point from source to destination. The rectangles represent hardware components, and the ovals are software modules. We use data flow graphs in the high-level design, because they describe the overall operation of the system while hiding the details of how it works. Issues such as safety (e.g., Isaac Asimov's first Law of Robotics "A robot may not harm a human being, or, through inaction, allow a human being to come to harm") and testing (e.g., we need to verify our system is operational) should be addressed during the high-level design. A data flow graph for a simple position measurement system is shown in Figure 2.4. The sensor converts position in an electrical resistance. The analog circuit converts resistance into the 0 to +3.3V voltage range required by the ADC. The 12-bit ADC converts analog voltage into a digital sample. The **ADC driver**, using the ADC and timer hardware, collects samples and calculates voltages. The software converts voltage to position. Voltage and position data are represented as fixed-point numbers within the computer. The position data is passed to the **LCD driver** creating ASCII strings, which will be sent to the **liquid crystal display** (LCD) module.

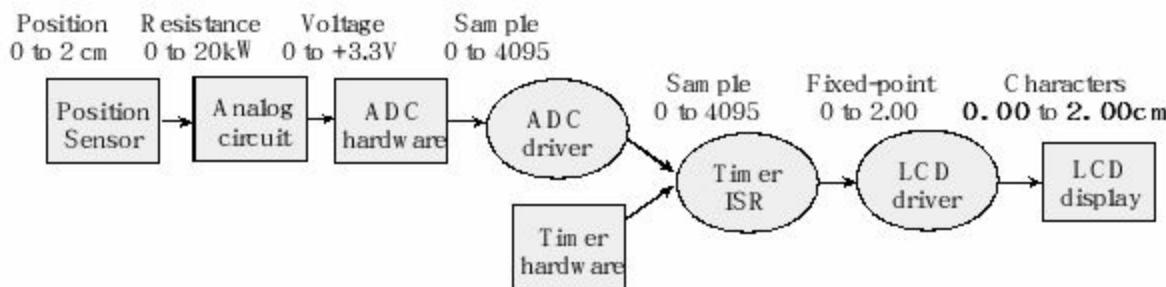


Figure 2.4. A data flow graph showing how the position signal passes through the system.

The next phase is **engineering design**. We begin by constructing a preliminary design. This system includes the overall top-down hierarchical structure, the basic I/O signals, shared data structures, and overall software scheme. At this stage there should be a simple and direct correlation between the hardware/software systems and the conceptual model developed in the high-level design. Next, we finish the top-down hierarchical structure and build mock-ups of the mechanical parts (connectors, chassis, cables etc.) and user software interface. Sophisticated 3-D CAD systems can create realistic

images of our system. Detailed hardware designs must include mechanical drawings. It is a good idea to have a second source, which is an alternative supplier that can sell our parts if the first source can't deliver on time. **Call graphs** are a graphical way to define how the software/hardware modules interconnect. **Data structures**, which will be presented throughout the book, include both the organization of information and mechanisms to access the data. Again safety and testing should be addressed during this low-level design.

A call graph for a simple position measurement system is shown in Figure 2.5. Again, rectangles represent hardware components, and ovals show software modules. An arrow points from the calling routine to the module it calls. The I/O ports are organized into groups and placed at the bottom of the graph. A high-level call graph, like the one shown in Figure 2.5, shows only the high-level hardware/software modules. A detailed call graph would include each software function and I/O port. Normally, hardware is passive and the software initiates hardware/software communication, but as we will learn in this book, it is possible for the hardware to interrupt the software and cause certain software modules to be run. In this system, the timer hardware will cause the **ADC** software to collect a sample. The timer interrupt service routine (ISR) gets the next sample from the **ADC** software, converts it to position, and displays the result by calling the **LCD** interface software. The double-headed arrow between the ISR and the hardware means the hardware triggers the interrupt and the software accesses the hardware.

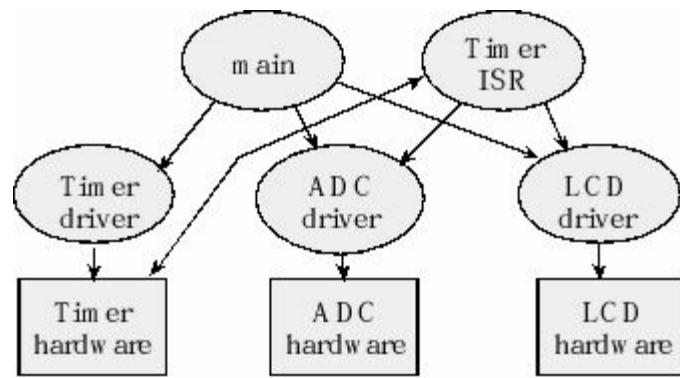


Figure 2.5. A call graph for a simple position measurement system.

**Observation:** If module A calls module B, and B returns data, then a data flow graph will show an arrow from B to A, but a call graph will show an arrow from A to B.

The next phase is **implementation**. We can also call this stage **development** because we will develop a possible solution. An advantage of a top-down design is that implementation of subcomponents can occur simultaneously. During the initial iterations of the life cycle, it is quite efficient to implement the hardware/software using simulation. One major advantage of simulation is that it is usually quicker to implement an initial product on a simulator versus constructing a physical device out of actual components. Rapid prototyping is important in the early stages of product development. This allows for more loops around the analysis-design-implementation-testing-deployment cycle, which in turn leads to a more sophisticated product.

Recent software and hardware technological developments have made significant impacts on the software development for embedded microcomputers. The simplest approach is to use a cross-assembler or cross-compiler to convert source code into the machine code for the target system. The machine code can then be loaded into the target machine. Debugging embedded systems with this simple approach is very difficult for two reasons. First, the embedded system lacks the usual keyboard and display that assist us when we debug regular software. Second, the nature of embedded systems involves the complex and real-time interaction between the hardware and software. These real-time interactions make it impossible to test software with the usual single-stepping and print statements.

The next technological advancement that has greatly affected the manner in which embedded systems are developed is simulation. Because of the high cost and long times required to create hardware prototypes, many preliminary feasibility designs are now performed using hardware/software simulations. A simulator is a software application that models the behavior of the hardware/software system. If both the external hardware and software program are simulated together, even though the simulated time is slower than the clock on the wall, the real-time hardware/software interactions can be studied.

During the **testing** phase, we evaluate the performance of our system. First, we debug the system and validate basic functions. Next, we use careful measurements to optimize performance such as static efficiency (memory requirements), dynamic efficiency (execution speed), accuracy (difference between expected truth and measured), and stability (consistent operation.) Debugging techniques will be presented at the end of most chapters.

**Maintenance** is the process of correcting mistakes, adding new features, optimizing for execution speed or program size, porting to new computers or operating systems, and reconfiguring the system to solve a similar problem. No system is static. Customers may change or add requirements or constraints. To be profitable, we probably will wish to tailor each system to the individual needs of each customer. Maintenance is not really a separate phase, but rather involves additional loops around the life cycle.

Figure 2.3 describes top-down design as a cyclic process, beginning with a problem statement and ending up with a solution. With a **bottom-up** design we begin with solutions and build up to a problem statement. Many innovations begin with an idea, “what if...?” In a bottom-up design, one begins with designing, building, and testing low-level components. The low-level designs can be developed in parallel. Bottom-up design may be inefficient because some subsystems may be designed, built, and tested, but never used. As the design progresses the components are fit together to make the system more and more complex. Only after the system is completely built and tested does one define the overall system specifications. The bottom-up design process allows creative ideas to drive the products a company develops. It also allows one to quickly test the feasibility of an idea. If one fully understands a problem area and the scope of potential solutions, then a top-down design will arrive at an effective solution most quickly. On the other hand, if one doesn't really understand the problem or the scope of its solutions, a bottom-up approach allows one to start off by learning about the problem.

## 2.4. Successive Refinement

Throughout the book in general, we discuss how to solve problems on the computer. In this section, we discuss the process of converting a problem statement into an algorithm. Later in the book, we will show how to map algorithms into assembly language. We begin with a set of general specifications, and then create a list of requirements and constraints. The general specifications describe the problem statement in an overview fashion, requirements define the specific things the system must do, and constraints are the specific things the system must not do. These requirements and constraints will guide us as we develop and test our system.

**Observation:** Sometimes the specifications are ambiguous, conflicting, or incomplete.

There are two approaches to the situation of ambiguous, conflicting, or incomplete specifications. The best approach is to resolve the issue with your supervisor or customer. The second approach is to make a decision and document the decision.

**Performance Tip:** If you feel a system specification is wrong, discuss it with your supervisor. We can save a lot of time and money by solving the correct problem in the first place.

**Successive refinement**, **stepwise refinement**, and **systematic decomposition** are three equivalent terms for a technique to convert a problem statement into a software algorithm. We start with a task and decompose the task into a set of simpler subtasks. Then, the subtasks are decomposed into even simpler sub-subtasks. We make progress as long as each subtask is simpler than the task itself. During the task decomposition we must make design decisions as the details of exactly how the task will be performed are put into place. Eventually, a subtask is so simple that it can be converted to software code. We can decompose a task in four ways, as shown in Figure 2.6. The **sequence**, **conditional**, and **iteration** are the three building blocks of structured programming. Because embedded systems often have real-time requirements, they employ a fourth building block called interrupts. We will implement time-critical tasks using interrupts, which are hardware-triggered software functions. Interrupts will be discussed in more detail in Chapters 9, 10, and 11. When we solve problems on the computer, we need to answer these questions:

- What does being in a state mean?
- What is the starting state of the system?
- What information do we need to collect?
- What information do we need to generate?
- How do we move from one state to another?
- What is the desired ending state?

- List the parameters of the state
- Define the initial state
- List the input data
- List the output data
- Specify actions we could perform
- Define the ultimate goal

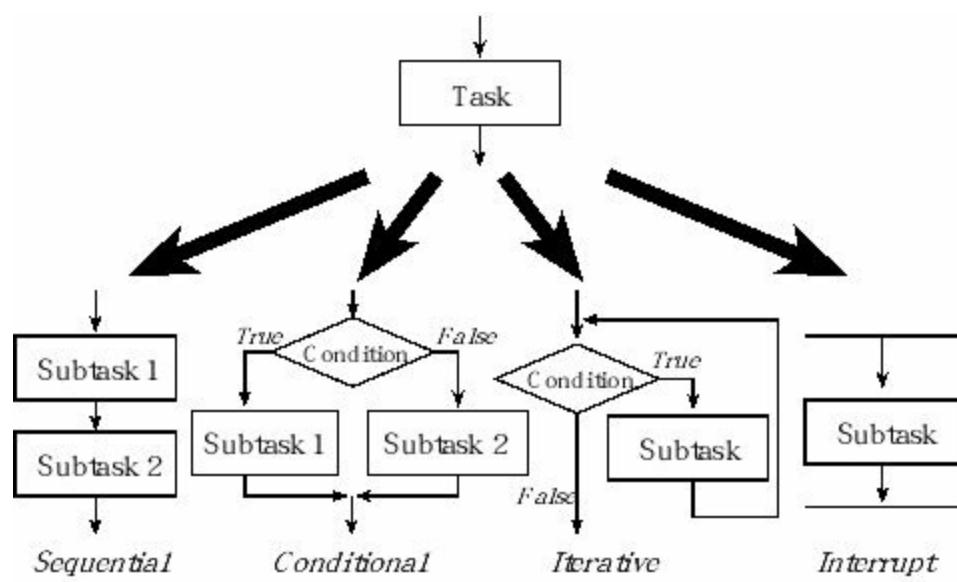


Figure 2.6. We can decompose a task using the building blocks of structured programming.

We need to recognize these phrases that translate to four basic building blocks:

“do A then do B”	→ sequential
“do A and B in either order”	→ sequential
“if A, then do B”	→ conditional
“for each A, do B”	→ iterative
“do A until B”	→ iterative
“repeat A over and over forever”	→ iterative (condition always true)
“on external event do B”	→ interrupt
“every t msec do B”	→ interrupt

---

### Example 2.1. Build a digital door lock using seven switches.

**Solution:** The system has seven binary inputs from the switches and one binary output to the door lock. The **state** of this system is defined as “door locked” and “door unlocked”. Initially, we want the door to be locked, which we can make happen by turning a solenoid off (make binary output low). If the 7-bit binary pattern on the switches matches a pre-defined keycode, then we want to unlock the door (make binary output high). Because the switches might bounce (flicker on and off) when changed, we will make sure the switches match the pre-defined keycode for at least 1 ms before unlocking the door. We can change states by writing to the output port for the solenoid. Like most embedded systems, there is no ending state. Once the switches no longer match the keycode the door will lock again. The first step in successive refinement is to divide the tasks into those performed once (Initialization), and those tasks repeated over and over (Execute lock), as shown as the left flowchart in Figure 2.7.

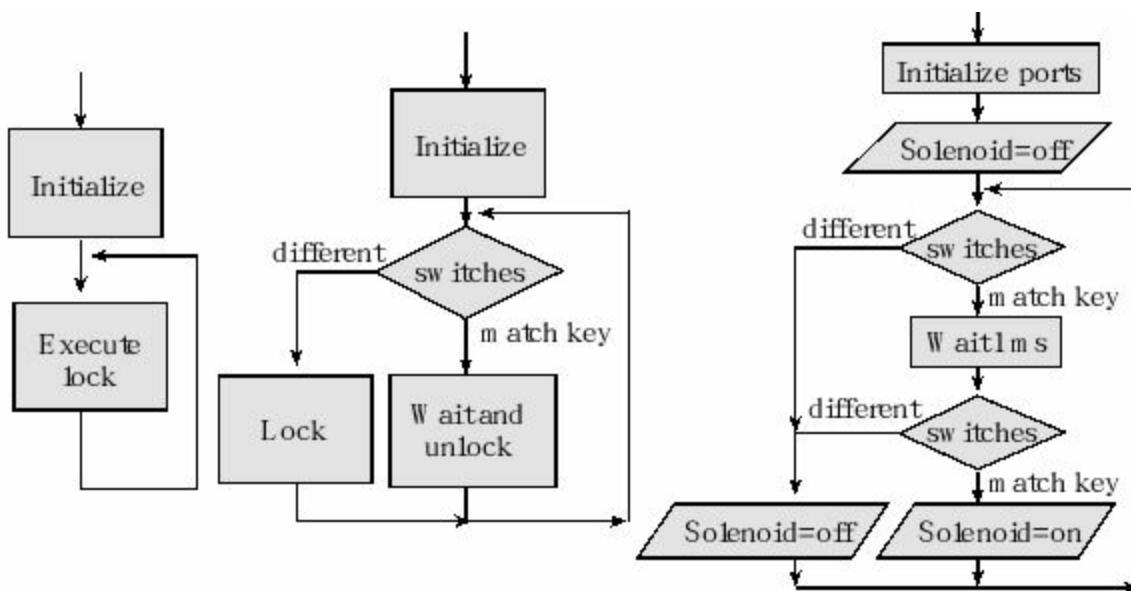


Figure 2.7. We can decompose a task using the building blocks of structured programming.

As shown in the middle flow chart, we implement if the switches match the key, then unlock. If the switches do not match we will lock the door. To verify the user entered the proper keycode the switches must match, then match again after 1 ms. There are two considerations when designing a system: security and safety. Notice that the system will lock the door if power is removed, because power applied to the solenoid will unlock the door. For safety reasons, there should be a mechanical way to unlock the door from the inside in case of emergency.

---

## 2.5. Quality Design

Embedded system development is similar to other engineering tasks. We can choose to follow well-defined procedures during the development and evaluation phases, or we can meander in a haphazard way and produce code that is hard to test and harder to change. The ultimate goal of the system is to satisfy the stated objectives such as accuracy, stability, and input/output relationships. Nevertheless it is appropriate to separately evaluate the individual components of the system. Therefore in this section, we will evaluate the quality of our software. There are two categories of performance criteria with which we evaluate the “goodness” of our software. Quantitative criteria include dynamic efficiency (speed of execution), static efficiency (memory requirements), and accuracy of the results. Qualitative criteria center on ease of software maintenance. Another qualitative way to evaluate software is ease of understanding. If your software is easy to understand then it will be:

- Easy to debug (fix mistakes)
- Easy to verify (prove correctness)
- Easy to maintain (add features)

**Common Error:** Programmers who sacrifice clarity in favor of execution speed often develop software that runs fast, but is error-prone and difficult to change.

Golden Rule of Software Development

Write software for others as you wish they would write for you.

### 2.5.1. Quantitative Performance Measurements

In order to evaluate our software quality, we need performance measures. The simplest approaches to this issue are quantitative measurements. **Dynamic efficiency** is a measure of how fast the program executes. It is measured in seconds or processor bus cycles. **Static efficiency** is the number of memory bytes required. Since most embedded computer systems have both RAM and ROM, we specify memory requirement in global variables, stack space, fixed constants and program. The global variables plus the stack must fit into the available RAM. Similarly, the fixed constants plus the program must fit into the available ROM. We can also judge our embedded system according to whether or not it satisfies given requirements and constraints, like accuracy, cost, power, size, reliability, and time-table.

### 2.5.2. Qualitative Performance Measurements

Qualitative performance measurements include those parameters to which we cannot assign a direct numerical value. Often in life the most important questions are the easiest to ask, but the hardest to answer. Such is the case with software quality. So therefore we ask the following qualitative questions. Can we prove our software works? Is our software easy to understand? Is our software easy to change? Since there is no single approach to writing the best software, we can only hope to present some techniques that you may wish to integrate into your own software style. In fact, this book devotes considerable effort to the important issue of developing quality software. In particular, we will study self-documented code, abstraction, modularity, and layered software. These issues indeed play a profound effect on the bottom-line financial success of our projects. Although quite real, because there is often not an immediate and direct relationship between a software's quality and profit, we may be mistakenly tempted to dismiss the importance of quality.

To get a benchmark on how good a programmer you are, take the following two challenges. In the first challenge, find a major piece of software that you have written over 12 months ago, and then see if you can still understand it enough to make minor changes in its behavior. The second challenge is to exchange with a peer a major piece of software that you have both recently written (but not written together), then in the same manner, see if you can make minor changes to each other's software.

**Observation:** You can tell if you are a good programmer if 1) you can understand your own code 12 months later, and 2) others can make changes to your code.

### 2.5.3. Attitude

Good engineers employ well-defined design processes when developing complex systems. When we work within a structured framework, it is easier to prove our system works (verification) and to modify our system in the future (maintenance.) As our software systems become more complex, it becomes increasingly important to employ well-defined software design processes. Throughout this book, a very detailed set of software development rules will be presented. This book focuses on real-time embedded systems written in assembly language and C, but most of the design processes should apply to other languages as well. At first, it may seem radical to force such a rigid structure to software. We might wonder if creativity will be sacrificed in the process. True creativity is more about good solutions to important problems and not about being sloppy and inconsistent. Because software maintenance is a critical task, the time spent organizing, documenting, and testing during the initial development stages will reap huge dividends throughout the life of the software project.

**Observation:** The easiest way to debug is to write software without any bugs.

We define **clients** as programmers who will use our software. A client develops software that will call our functions. We define **coworkers** as programmers who will debug and upgrade our software. A coworker, possibly ourselves, develops, tests, and modifies our software.

Writing quality software has a lot to do with attitude. We should be embarrassed to ask our coworkers to make changes to our poorly written software. Since so much software development effort involves maintenance, we should create software modules that are easy to change. In other words, we should expect each piece of our code will be read by another engineer in the future, whose job it will be to make changes to our code. We might be tempted to quit a software project once the system is running, but this short time we might save by not organizing, documenting, and testing will be lost many times over in the future when it is time to update the code.

As project managers, we must reward good behavior and punish bad behavior. A company, in an effort to improve the quality of their software products, implemented the following policies.

The employees in the customer relations department receive a bonus for every software bug that they can identify. These bugs are reported to the software developers, who in turn receive a bonus for every bug they fix.

### **Checkpoint 2.7:** Why did the above policy fail horribly?

We should demand of ourselves that we deliver bug-free software to our clients. Again, we should be embarrassed when our clients report bugs in our code. We should be mortified when other programmers find bugs in our code. There are a few steps we can take to facilitate this important aspect of software design.

Test it now. When we find a bug, fix it immediately. The longer we put off fixing a mistake the more complicated the system becomes, making it harder to find. Remember that bugs do not go away on their own, but we can make the system so complex that the bugs will manifest themselves in mysterious and obscure ways. For the same reason, we should completely test each module individually, before combining them into a larger system. We should not add new features before we are convinced the existing system is bug-free. In this way, we start with a working system, add features, and then debug this system until it is working again. This incremental approach makes it easier to track progress. It allows us to undo bad decisions, because we can always revert back to a previously working system. Adding new features before the old ones are debugged is very risky. With this sloppy approach, we could easily reach the project deadline with 100% of the features implemented, but have a system that doesn't run. In addition, once a bug is introduced, the longer we wait to remove it, the harder it will be to correct. This is particularly true when the bugs interact with each other. Conversely, with the incremental approach, when the project schedule slips, we can deliver a working system at the deadline that supports some of the features.

### **Maintenance Tip:** Go from working system to working system.

Plan for testing. How to test each module should be considered at the start of a project. In particular, testing should be included as part of the design of both hardware and software components. Our testing and the client's usage go hand in hand. In particular, how we test the module will help the client understand the context and limitations of how our component is to be used. On the other hand, a clear understanding of how the client wishes to use our hardware/software component is critical for both its design and its testing.

### **Maintenance Tip:** It is better to have some parts of the system that run with 100% reliability than to have the entire system with bugs.

Get help. Use whatever features are available for organization and debugging. Pay attention to warnings, because they often point to misunderstandings about data or functions. Misunderstanding of assumptions that can cause bugs when the software is upgraded, or reused in a different context than originally conceived. Remember that computer time is a lot cheaper than programmer time.

**Maintenance Tip:** It is better to have a system that runs slowly than to have one that doesn't run at all.

Deal with the complexity. In the early days of microcomputer systems, software size could be measured in 100's of lines of source code using 1000's of bytes of memory. These early systems, due to their small size, were inherently simple. The explosion of hardware technology (both in speed and size) has led to a similar increase in the size of software systems. Some people forecast that by the next decade, automobiles will have 10 million lines of code in their embedded systems. The only hope for success in a large software system will be to break it into simple modules. In most cases, the complexity of the problem itself cannot be avoided. E.g., there is just no simple way to get to the moon. Nevertheless, a complex system can be created out of simple components. A real creative effort is required to orchestrate simple building blocks into larger modules, which themselves are grouped to create even larger systems. Use your creativity to break a complex problem into simple components, rather than developing complex solutions to simple problems.

**Observation:** There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other way is make it so complicated that there are no obvious deficiencies. C.A.R. Hoare, "The Emperor's Old Clothes," CACM Feb. 1981.

## 2.6. Debugging Theory

The last section of every chapter in this book will address debugging techniques. Every programmer is faced with the need to debug and verify the correctness of his or her software. A **debugging instrument** is hardware or software used for the purpose of debugging. In this book, we will study hardware-level probes like the logic analyzer, oscilloscope, and Joint Test Action Group (JTAG standardized as the IEEE 1149.1); software-level tools like simulators, monitors, and profilers; and manual tools like inspection and print statements. **Nonintrusiveness** is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in program performance by the debugging instrument itself. For example, a print statement added to your source code is very intrusive because it significantly affects the real-time interaction of the hardware and software. It is important to quantify the intrusiveness of an instrument. Let  $t$  be the average time it takes to run the software code comprising instrument. This time  $t$  is how much less time the system has to perform its regular duties. Let  $\Delta t$  be the average time between executions of the instrument. A quantitative measure of intrusiveness is  $t/\Delta t$ , which is the fraction of the time consumed by the process of debugging itself. A debugging instrument is classified as **minimally intrusive** if it has a negligible effect on the system being debugged. In other words, if  $t/\Delta t$  so small that the debugging processes have a finite but inconsequential effect on the system behavior, we classify it as minimally intrusive. In a real microcomputer system, breakpoints and single-stepping are intrusive, because the real hardware continues to change while the software has stopped. When a program interacts with real-time events, the performance can be significantly altered when using intrusive debugging tools. On the other hand, we will learn later in this book that dumps, dumps with filter, and monitors are much less intrusive. Examples of monitors are LEDs or organic LED (OLED) displays. A logic analyzer that passively monitors the activity of the software is completely nonintrusive. Interestingly, breakpoints and single-stepping on a mixed hardware/software simulator are often nonintrusive, because the simulated hardware and the software are affected together.

**Checkpoint 2.8:** What does it mean for a debugging instrument to be minimally intrusive? Give both a general answer and a specific criterion.

Research in the area of program monitoring and debugging mirrors the rapid pace of developments in other areas of computer architecture and software systems. Because of the complexity explosion in computer systems, effective debugging tools are essential. The critical aspect of debugging an embedded system is the ability to see what the software is doing, where it is executing, and when it did do it, without the debugger itself modifying system behavior. Terms such as program testing, diagnostics, performance debugging, functional debugging, tracing, profiling, instrumentation, visualization, optimization, verification, performance measurement, and execution measurement have specialized meanings, but they are also used interchangeably, and they often describe overlapping functions. For example, the terms profiling, tracing, performance measurement, or execution measurement may be used to describe the process of examining a program from a time viewpoint. But, tracing is also a term that may be used to describe the process of monitoring a program state or history for functional errors, or to describe the process of stepping through a program with a

debugger. Usage of these terms among researchers and users vary.

Furthermore, the meaning and scope of the term debugging itself is not clear. In this book the goal of debugging is to maintain and improve software, and the role of a debugger is to support this endeavor. The debugging process is defined as testing, stabilizing, localizing, and correcting errors. Although testing, stabilizing, and localizing errors are important and essential to debugging, they are auxiliary processes: the primary goal of debugging is to remedy faults or to correct errors in a program.

**Stabilization** is process of fixing the inputs so that the system can be run over and over again yielding repeatable outputs.

Although, a wide variety of program monitoring and debugging tools are available today, in practice it is found that an overwhelming majority of users either still prefer or rely mainly upon “rough and ready” manual methods for locating and correcting program errors. These methods include desk-checking, dumps, and print statements, with print statements being one of the most popular manual methods. Manual methods are useful because they are readily available, and they are relatively simple to use. But, the usefulness of manual methods is limited: they tend to be highly intrusive, and they do not provide adequate control over repeatability, event selection, or event isolation. A real-time system, where software execution timing is critical, usually cannot be debugged with simple print statements, because the print statement itself will require too much time to execute.

**Black-box testing** is simply observing the inputs and outputs without looking inside. Black-box testing has an important place in debugging a module for its functionality. On the other hand, **white-box testing** allows you to control and observe the internal workings of a system. A common mistake made by new engineers is to just perform black box testing. Effective debugging uses both. One must always start with black-box testing by subjecting a hardware or software module to appropriate test-cases. Once we document the failed test-cases, we can use them to aid us in effectively performing the task of white-box testing.

A debugging instrument is defined as hardware or software that is added to the system for the purpose of debugging. A print statement is a common example of an instrument. Using the editor, one adds print statements to the code that either verify proper operation or illustrate the programming errors. If we test a system, then remove the instruments, the system may actually stop working, because of the importance of timing in embedded systems. If we leave debugging instruments in the final product, we can use the instruments to test systems on the production line, or test systems returned for repair. On the other hand, sometimes we wish to provide for a mechanism to reliably and efficiently remove all instruments when the debugging is done. Consider the following mechanisms as you develop your own unique debugging style.

- Place all instruments in a unique column, so you can easily distinguish instruments from regular programs.
- Define all debugging instruments as functions that all have a specific pattern in their names. In this way, the find/replace mechanism of the editor can be used to find all the calls to the instruments.
- Define the instruments so that they test a run time global flag. When this flag is turned off, the instruments perform no function. Notice that this method leaves a permanent copy of the debugging code in the final system, causing it to suffer a

runtime overhead, but the debugging code can be activated dynamically without recompiling. Many commercial software applications utilize this method because it simplifies “on-site” customer support.

- Use conditional compilation (or conditional assembly) to turn on and off the instruments when the software is compiled. When the assembler or compiler supports this feature, it can provide both performance and effectiveness.

The emergence of concurrent languages and the increasing use of embedded real-time systems place further demands on debuggers. The complexities introduced by the interaction of multiple events or time dependent processes are much more difficult to debug than errors associated with sequential programs. The behavior of non-real-time sequential programs is reproducible: for a given set of inputs their outputs remain the same. In the case of concurrent or real-time programs this does not hold true. Control over repeatability, event selection, and event isolation is even more important for concurrent or real-time environments.

**Checkpoint 2.9:** Consider the difference between a runtime flag that activates a debugging command versus an assembly/compile-time flag. In both cases it is easy to activate/deactivate the debugging statements. For each method, list one factor for which that method is superior to the other.

**Checkpoint 2.10:** What is the advantage of leaving debugging instruments in a final delivered product?

**Observation:** There are two important components of debugging: having control over events and being able to see what is happening. Remember: **control** and **observability**!

**Common Error:** The most common debugging mistake new programmers make is to simply observe the overall inputs and outputs system without looking inside the device. Then they go to their professor and say, “My program gives incorrect output. Do you know why?”

## 2.7. Switch and LED Interfaces

This section is completely out of place. It really belongs back in Chapters 4 or 8 with the other input/output devices. However, I couldn't wait to show you how much fun it is to make the microcontroller interact with real physical devices. So in this section I will take a short side step from the business of concepts and theories to teach you how to connect switches and LEDs to the microcontroller. We will use switches to input data and use LEDs to output results.

Input/output devices are critical components of an embedded system. The first input device we will study is the **switch**. It allows the human to input binary information into the computer. Typically we define the asserted state, or logic true, when the switch is pressed. Contact switches can also be used in machines to detect mechanical contact (e.g., two parts touching, paper present in the printer, or wheels on the ground etc.) A single pole single throw (SPST) switch has two connections. The switches are shown as the device between the little open circles in Figure 2.8. In a normally-open switch (NO), the resistance between the connections is infinite (over  $100\text{ M}\Omega$  on the B3F tactile switch) if the switch is not pressed and zero (under  $0.1\text{ }\Omega$  on the B3F tactile switch) if the switch is pressed. There are two ways to interface the switch to the microcontroller: positive and negative logic.

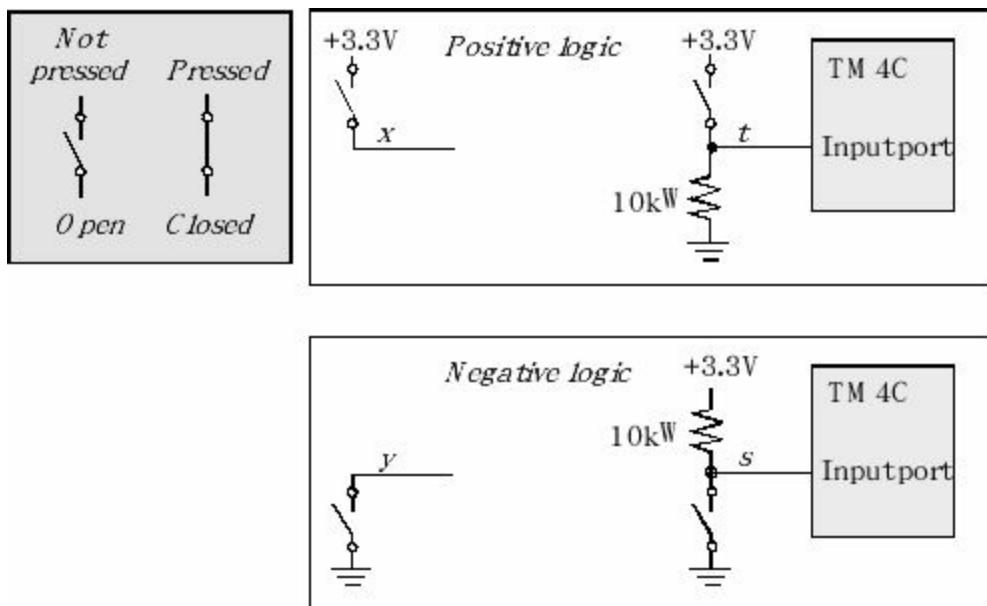


Figure 2.8. Single Pole Single Throw (SPST) Switch interface. The photo is a B3F tactile switch.

With **positive logic** the asserted state has a larger voltage than the unasserted state. In other words, when the switch is pressed, the digital signal will be a logic one ( $t = 3.3\text{V}$ ), and when the switch is not pressed the digital signal will be a logic zero ( $t = 0\text{V}$ ). Because we wish to have a  $3.3\text{V}$  signal when we press the switch, we connect one side of the switch to  $+3.3\text{V}$ . Notice in Figure 2.8 the signal  $x$  will be  $3.3\text{V}$  when the switch is pressed. However,  $x$  will float, which means  $x$  has no voltage, when the switch is not pressed. If we add a pull-down resistor to ground the signal  $t$  will be  $0\text{V}$  when the switch is not pressed and  $3.3\text{V}$  if the switch is pressed, as desired. Notice that  $10\text{ k}\Omega$  is 100,000 times larger than the on-resistance of the switch and 10,000 times smaller than its off-resistance.

Another way to choose the pull-down resistor is to consider the input current of the microcontroller input pin. The current into the microcontroller will be less than  $2\mu\text{A}$  (defined as  $I_{IL}$  and  $I_{IH}$  in the data sheet). So, if the current into microcontroller is  $2\mu\text{A}$ , then the voltage drop across the  $10\text{ k}\Omega$  resistor will be  $0.02\text{ V}$ , which is negligibly small.

With **negative logic** the asserted state has a smaller voltage than the unasserted state. In other words, when the switch is not pressed, the digital signal will be a logic one ( $s = 3.3\text{V}$ ), and when the switch is pressed the digital signal will be a logic zero ( $s = 0\text{V}$ ). With the negative logic interface we want the pressed state to create a  $0\text{V}$ . So, to convert the resistance of the switch into a digital signal, we can connect one side of the switch to  $0\text{V}$ . Notice in Figure 2.8 the signal  $y$  will be  $0\text{V}$  when the switch is pressed, but will float when the switch is not pressed. If we add a pull-up resistor to  $+3.3\text{V}$  the signal  $s$  will be  $0\text{V}$  when the switch is pressed and  $3.3\text{V}$  if the switch is not pressed, as desired.

One of the complicating issues with mechanical switches is they can bounce (oscillate on and off) when touched and when released. The contact bounce varies from switch to switch and from time to time, but usually bouncing is a transient event lasting less than  $5\text{ ms}$ . We can eliminate the effect of bounce if we design software that waits at least  $10\text{ ms}$  between times we read the switch values.

A **light emitting diode** (LED) emits light when an electric current passes through it. LEDs have polarity, meaning current must pass from anode to cathode to activate. The anode is labeled **a** or **+**, and cathode is labeled **k** or **-**. The cathode is the short lead and there may be a slight flat spot on the body of round LEDs. Thus, the anode is the longer lead. The brightness of an LED depends on the applied electrical power ( $P=I^*V$ ). Since the LED voltage is approximately constant in the active region (see left side of Figure 2.9), we can establish the desired brightness by setting the current.

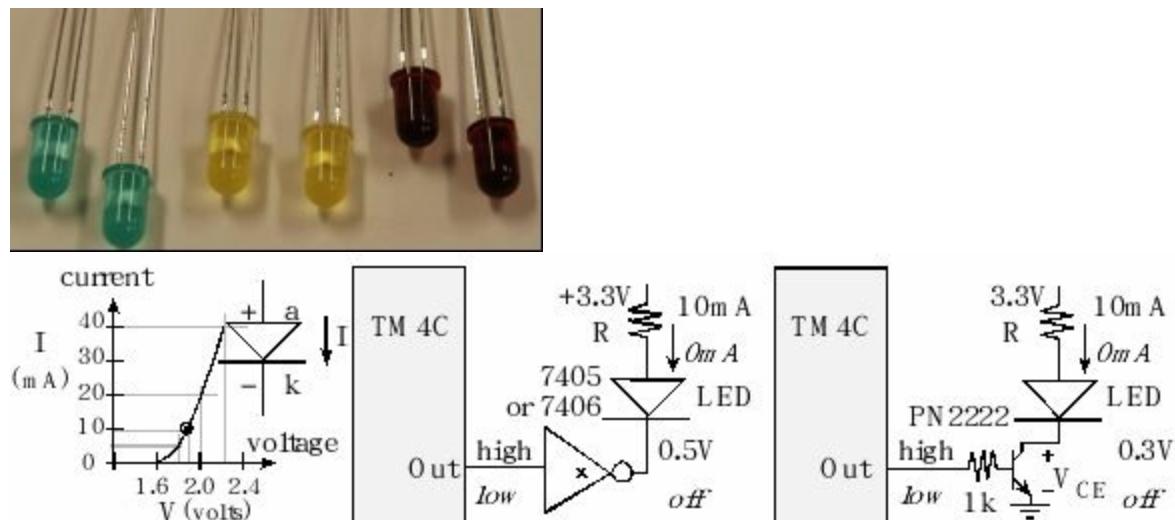


Figure 2.9. Positive logic LED interface (Lite-On LTL-10223W).

**Checkpoint 2.11:** What resistor value in Figure 2.9 is needed if the desired LED operating point is  $1.7\text{V}$  and  $12\text{ mA}$ ?

If the LED current is above 8 mA, we cannot connect it directly to the microcontroller because the high currents may damage the chip. Figure 2.9 shows two possible interface circuits we could use. In both circuits if the software makes its output high the LED will be on. If the software makes its output low the LED will be off (shown in Figure 2.9 with italics). When the software writes a logic 1 to the output port, the input to the 7405/PN2222 becomes high, output from the 7405/PN2222 becomes low, 10 mA travels through the LED, and the LED is on. When the software writes a logic 0 to the output port, the input to the 7405/PN2222 becomes low, output from the 7405/PN2222 floats (neither high nor low), no current travels through the LED, and the LED is dark. The value of the resistor is selected to establish the proper LED current. When active, the LED voltage will be about 2 V, and the power delivered to the LED will be controlled by its current. If the desired brightness requires an operating point of 1.9 V at 10 mA, then the resistor value should be

$$R = \frac{3.3 - V_d - V_{OL}}{I_d} = \frac{3.3 - 1.9 - 0.5}{0.01} = 90\text{W}$$

where  $V_d$ ,  $I_d$  is the desired LED operating point, and  $V_{OL}$  is the output low voltage of the LED driver. If we use a standard resistor value of  $100\Omega$  in place of the  $90\Omega$ , then the current will be  $(3.3 - 1.9 - 0.5\text{V})/100\Omega$ , which is about 9 mA. This slightly lower current is usually acceptable.

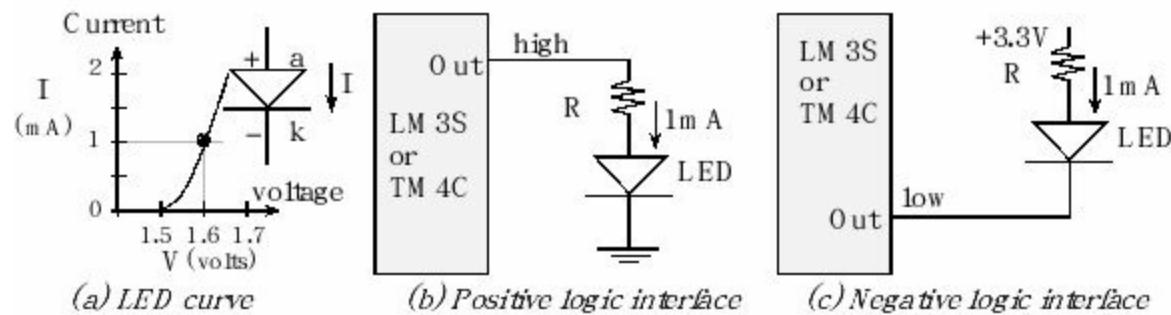


Figure 2.10. Low current LED interface (Agilent HLMP-D150).

When the LED current is less than 8 mA, we can interface it directly to an output pin without using a driver. The LED shown in Figure 2.10 has an operating point of 1.7 V and 1 mA. For the positive logic interface (Figure 2.10b) we calculate the resistor value based on the desired LED voltage and current

$$R = \frac{V_{OH} - V_d}{I_d} = \frac{2.4 - 1.6}{0.001} = 800\text{W}$$

where  $V_{OH}$  is the output high voltage of the microcontroller output pin. Since  $V_{OH}$  can vary from 2.4 to 3.3 V, it makes sense to choose a resistor from a measured value of  $V_{OH}$ , rather than the minimum value of 2.4 V. Negative logic means the LED is activated when the software outputs a zero. For the negative logic interface (Figure 2.10c) we use a similar equation to determine the resistor value

$$R = \frac{3.3 - V_d - V_{OL}}{I_d} = \frac{3.3 - 1.6 - 0.4}{0.001} = 1.3\text{kW}$$

where  $V_{OL}$  is the output low voltage of the microcontroller output pin.

If we use a 1.2 kΩ in place of the 1.3 kΩ, then the current will be  $(3.3-1.6-0.4V)/1.2k\Omega$ , which is about 1.08 mA. This slightly higher current is usually acceptable. If we use a standard resistor value of 1.5 kΩ in place of the 1.3 kΩ, then the current will be  $(3.3-1.6-0.4V)/1.5k\Omega$ , which is about 0.87 mA. This slightly lower current is usually acceptable.

**Design for tolerance** means making it work for a range of possibilities. Assume the resistor value in Figure 2.10c is 1.3kΩ, and the diode voltage remains at 1.6V. The  $V_{OL}$  could range from 0 to 0.4 V. At  $V_{OL}=0V$ ,  $I_d=(3.3-1.6-0.0V)/1.3k\Omega$ , which is about 1.3 mA. At  $V_{OL}=0.4V$ ,  $I_d=(3.3-1.6-0.4V)/1.3k\Omega$ , which is about 1.0 mA. So the uncertainty in  $V_{OL}$  causes a 1.0 to 1.3 mA uncertainty in  $I_d$ . This is usually acceptable. However, it makes sense to measure each of these voltages and currents in the actual circuit to verify its proper operation.

**Checkpoint 2.12:** What resistor value in of Figure 2.10 is needed if the desired LED operating point is 1.7V and 2 mA? Use the negative logic interface.

**Observation:** Using standard resistor values will make our product less expensive and easier to obtain parts.

## 2.8. Introduction to C

This section is a brief introduction to C. C is a general-purpose programming language initially developed by Dennis Ritchie between 1969 and 1973 while at AT&T Bell Labs. At the time, there were programming languages called A and another named B, so Ritchie decided to name his language C. Dennis Ritchie and Brian Kernighan wrote the first book on C, *The C Programming Language*. Ritchie was also one of the developers of the UNIX operating system. As C became more popular, many derivative languages were introduced. C++ was developed by Bjarne Stroustrup 1979-1983 also at Bell Labs. C++ is a language originally called “C plus classes”. In 1999, a professional standard version of C, called C99, was defined. When you download Tivaware (<http://www.ti.com/tool/sw-tm4c>) from Texas Instruments, you will notice TI’s example code for the TM4C123 has been written in C99. In this book, we will adhere to the C99 standard, because it is prevalent in industry.

A **compiler** is system software that converts a high-level language program (human readable format) into object code (machine readable format). It produces software that is fast but to change the software we need to edit the source code and recompile.

C code ( `z = x+y;` ) → Assembly code ( `ADD R2,R1,R0` ) → Machine code ( `0xEB010200` )

An **assembler** is system software that converts an assembly language program (human readable format) into object code (machine readable format). An **interpreter** executes directly the high level language. It is interactive but runs slower than compiled code. Many languages can be compiled or interpreted. The original BASIC (Beginner's All-purpose Symbolic Instruction Code) was interpreted. This means the user typed software to the computer, and the interpreter executed the commands as they were typed. In this book, an example of the interpreter will be the command window while running the debugger. For more information on this interpreter, run Keil uVision and execute **Help->uVisionHelp**. Next, you need to click the Contents tab, open the **uVisionIDEUsersGuide**, and then click **DebugCommands**. It will show you a list of debugger commands you can type into the command window.

A **linker** builds software system by connecting (linking) software components. In Keil uVision, the build command (**Project->BuildTarget**) performs both a compilation and a linking. The example code in Program 2.1 has three software components that are linked together. These components are **startup.s** **uart.c** and **main.c**.

In an embedded system, the **loader** will program object code into flash ROM. We place object code in ROM because ROM is retains its information if power is removed and restored. In Keil uVision, the download command (**Flash->Download**) performs a load operation.

A **debugger** is a set of hardware and software tools we use to verify system is operating correctly. The two important aspects of a good debugger are control and observability.

Before we write software, we need to develop a plan. Software development is an iterative process. Even though we list steps the development process in a 1,2,3,4 order, in reality we cycle through these steps over and over. I like to begin with step 4), deciding how I will test it even before I decide what it does.

- 1) We begin with a list of the inputs and outputs. This usually defines what the overall system will do. We specify the range of values and their significance.
- 2) Next, we make a list of the required data. We must decide how the data is structured, what does it mean, how it is collected, and how it can be changed.
- 3) Next we develop the software algorithm, which is a sequence of operations we wish to execute. There are many approaches to describing the plan. Experienced programmers can develop the algorithm directly in C language. On the other hand, most of us need an abstractive method to document the desired sequence of actions. Flowcharts and pseudo code are two common descriptive formats. There are no formal rules regarding pseudo code, rather it is a shorthand for describing what to do and when to do it. We can place our pseudo code as documentation into the comment fields of our program. Next we write software to implement the algorithm as define in the flowchart and pseudo code.
- 4) The last stage is debugging. Learning debugging skills will greatly improve the quality of your software and the efficiency at which you can develop code.

Every C program has a **main** , and execution will begin at the top of this main program. There are four sections of a C program as shown in Program 2.1. The first section is the **documentation** section, which includes the purpose of the software, the authors, the date, and any copyright information. When the software involves external hardware we will add information about how the external hardware is connected. The second section is the **preprocessor directives**. We will use the preprocessor directive **#include** to connect this software with other modules. We use diamond braces to include system libraries, like the standard I/O, and we use quotes to link up with other user code within the project. In this case, the **uart**module is software we wrote to perform I/O with the universal asynchronous receiver/transmitter (UART). The details of the UART will be presented later in Section 8.2. For now, we just need to know when we call **printf** , information is transmitted out of the microcontroller and displayed for the operator to see. The third section is **global declarations** section. This section will include global variables and function prototypes for functions defined in this module. The last section will be the functions themselves. In this book we will use the terms subroutine, procedure, function, and program interchangeably. Every software system in C has exactly one **main** program, which defines where it begins execution.

Line 1 defines the start of main, and in this book all our C code will have this line exactly. Lines 2-5 are the body of the main program. Line 2 calls a function in the UART driver to initialize the serial port. Line 3 will output the message “Hello world” to the serial port. Line 4 will cause line 5 to be executed over and over. Line 5 doesn’t do anything, so the program will output the message once and sit there and do nothing forever. Line 6 is the end of the main, but this line will never be reached.C has a lot of paired symbols. For every ( there is a matching ) . For every { there is a matching } . For every " there is a matching ". The <> symbols will be matched only in **#include** statements. Other symbols that will be matched are [] and " .

## /\* \* \* \* 0. Documentation Section

```

// This program demonstrates the sections of a C program
// Author: Ramesh Yerraballi & Jon Valvano
// Date: 5/28/2014
// 1. Pre-processor Directives Section
#include <stdio.h> // Diamond braces for sys lib: Standard I/O
#include <stdint.h> // C99 definitions
#include "uart.h" // Quotes for user lib: UART lib
// 2. Global Declarations section
// 3. Subroutines Section
// MAIN: Mandatory routine for a C program to be executable
int main(void){      // line 1
    UART_Init();      // line 2
    printf("Hello world"); // line 3
    while(1){          // line 4
    }                  // line 5
}                  // line 6

```

Program 2.1. Simple program illustrating the sections of a C program.

If you wish to run a similar program, you can download and run the **Printf\_LF120** example from the web site. Preprocessor lines begin with # in the first column, like the **#include** lines in the above example. All preprocessor lines are invoked first (one pass through the software), and then all the other lines will be compiled as regular C (second pass).

In assembly language, symbols placed at the beginning of each line have special meaning. On the contrary, C is a **free field language**. Except for preprocessor lines, spaces, tabs and line breaks have the same meaning. This means we can place more than one statement on a single line, or place a single statement across multiple lines. For example a function could have been written without any line breaks

```
uint32_t Random(void){M=1664525*M+1013904223;return(M);}
```

Since we rarely make hardcopy printouts of our software, it is not necessary to minimize the number of line breaks. Furthermore, we could have added extra line breaks. I prefer the style of the program on the right because each line contains one complete thought or action. As you get more experienced, you will develop a programming style that is easy to understand. Although spaces, tabs, and line breaks are syntactically equivalent, their proper usage will have a profound impact on the readability of your software.

<b>uint32_t</b> <b>Random(void){</b> <b>M =</b> <b>1664525*M</b> <b>+1013904223;</b> <b>return(M);</b> <b>}</b>	<b>uint32_t Random(void){</b> <b>M = 1664525*M+1013904223;</b> <b>return(M);</b> <b>}</b>
---	--

The variable **M**, the function **Random**, the operation **\***, and the keyword **uint32\_t** are tokens in C. Each token must be contained on a single line. We see in the above example that tokens can be separated by white spaces, which include space, tab, line break, or by special characters. Special characters include punctuation marks (Table 2.2) and operations (Table 2.3).

Punctuation	Meaning
<b>;</b>	End of statement
<b>:</b>	Defines a label
<b>,</b>	Separates elements of a list
<b>( )</b>	Start and end of a parameter list
<b>{ } </b>	Start and stop of a compound statement
<b>[ ]</b>	Start and stop of a array index
<b>" "</b>	Start and stop of a string
<b>' '</b>	Start and stop of a character constant

**Table 2.2. Special characters can be punctuation marks.**

Operation	Meaning	Operation	Meaning
<b>=</b>	Assignment statement	<b>==</b>	Equal to comparison
<b>?</b>	Selection	<b>&lt;=</b>	Less than or equal to
<b>&lt;</b>	Less than	<b>&gt;=</b>	Greater than or equal to
<b>&gt;</b>	Greater than	<b>!=</b>	Not equal to
<b>!</b>	Logical not (true to false, false to true)	<b>&lt;&lt;</b>	Shift left
<b>~</b>	1's complement	<b>&gt;&gt;</b>	Shift right
<b>+</b>	Addition	<b>++</b>	Increment
<b>-</b>	Subtraction	<b>--</b>	Decrement
<b>*</b>	Multiply or pointer reference	<b>&amp;&amp;</b>	Boolean and
<b>/</b>	Divide	<b>  </b>	Boolean or
<b>%</b>	Modulo, division remainder	<b>+=</b>	Add value to
<b> </b>	Logical or	<b>-=</b>	Subtract value to
<b>&amp;</b>	Logical and, or address of	<b>*=</b>	Multiply value to
<b>^</b>	Logical exclusive or	<b>/=</b>	Divide value to
<b>.</b>		<b> =</b>	Or value to

Used to access parts of a structure			
	<b>&amp;=</b>	And value to	
	<b>^=</b>	Exclusive or value to	
	<b>&lt;&lt;=</b>	Shift value left	
	<b>&gt;&gt;=</b>	Shift value right	
	<b>%=</b>	Modulo divide value to	
	<b>-&gt;</b>	Pointer to a structure	

**Table 2.3. Special characters can be operators; operators can be made from 1, 2, or 3 characters.**

Punctuation marks (semicolons, colons, commas, apostrophes, quotation marks, braces, brackets, and parentheses) are very important in C. It is one of the most frequent sources of errors for both the beginning and experienced programmers. Another situation where spaces, tabs and line breaks matter is string constants. We cannot type tabs or line breaks within a string constant. The characters between the first " and second " define the string constant. A string is a set of ASCII characters terminated with a 0. For example, the following C code will output my name (see Section 4.5):

```
printf("Jonathan Valvano");
```

Program 2.2 illustrates the assignment operator. Notice that in the line **x=1;** the **x** is on the left hand side of the **=**. This specifies the address of **x** is the destination of assignment. On the other hand, in the line **z=x;** the **x** is on the right hand side of the **=**. This specifies the value of **x** will be assigned into the variable **z**. Also remember that the line **z=x;** creates two copies of the data. The original value remains in **x**, while **z** also contains this value.

```
int32_t x,y,z; // Three 32-bit signed variables
int main(void){
    x = 1; // set the value of x to 1
    y = 2; // set the value of y to 2
    z = x; // set the value of z to the value of x (both are 1)
    x = y = z = 0; // all all three to zero
    while(1){
    }
}
```

**Program 2.2. Simple program illustrating C arithmetic operators.**

Program 2.3 illustrates the arithmetic operations of addition, subtraction, multiplication and division. In the operation **x+4\*y**, multiplication has precedence over addition.

```
int32_t x,y,z; // Three 32-bit signed variables
int main(void){
```

```

x=1; y=2; // set the values of x and y
z = x+4*y; // arithmetic operation
x++; // same as x=x+1;
y--; // same as y=y-1;
x = y<<2; // left shift same as x=4*y;
z = y>>2; // right shift same as x=y/4;
y += 2; // same as y=y+2;
while(1){
}
}

```

Program 2.3. Simple program illustrating C arithmetic operators.

Program 2.4 defines a function that we can call from elsewhere in the software. We will introduce a simple conditional control structure. Assume all of Port A is configured as an output port, and Port B bit 2 as an input port. The goal is to make Port A equal to 0 if Port B bit 2 is zero, and make Port A equal to 100 if Port B bit 2 is set. The expression **GPIO\_PORTB\_DATA\_R&0x04** will return 0 if Port B bit 2 is 0 and will return a 4 if Port B bit 2 is 1. The expression **(GPIO\_PORTB\_DATA\_R&0x04)==0** will return true if Port B bit 2 is 0 and will return a false if Port B bit 2 is 1. The statement immediately following the **if** will be executed if the condition is true. The statement immediately following the **else** will be executed if the condition is false.

```

void CheckPB2(void){
    if((GPIO_PORTB_DATA_R&0x04)==0){ // test bit 2 of Port B
        GPIO_PORTA_DATA_R = 0; // if PB2 is 0, then Port A=0
    }else{
        GPIO_PORTA_DATA_R = 100; // if PB2 is not 0, then Port A=100
    }
}

```

Program 2.4. Simple program illustrating the C if else control structure.

Program 2.5 also defines a function. Assume that Port A bit 3 is configured as an output pin. The goal is to make the output pin toggle 100 times. Like the **if** statement, the **while** statement has a conditional test (i.e., returns a true/false). The statement immediately following the **while** will be executed over and over until the conditional test becomes false.

```

void TogglePA3(void){ uint32_t count;
    count = 0;
    while(count < 100){
        GPIO_PORTA_DATA_R = GPIO_PORTA_DATA_R^0x08; // toggle bit 3
        count++; // increment counter until counter becomes 100,
    }
}

```

Program 2.5. Simple program illustrating the C while control structure.

The **for** control structure has three parts and a body.

```
for(part1;part2;part3){body;}
```

In Program 2.5, the first part **count=0** is executed once at the beginning. Before the body is executed, the end-condition part 2 is executed. If the condition is true, **count<100** then the body **GPIO\_PORTA\_DATA\_R ^= 0x08;** is executed. After the body is executed, the third part is executed, **count++**. The second part is always a conditional that results in a true or a false. The body and third part are repeated until the conditional is false.

```
void TogglePA3(void){ uint32_t count;
for(count = 0; count < 100; count++){
    GPIO_PORTA_DATA_R ^= 0x08; // toggle bit 3
}
}
```

Program 2.6. A rewrite of Program 2.5 illustrating the C for-loop control structure.

As with all programming languages the order of the tokens is important. There are two issues to consider when evaluating complex statements. The **precedence** of the operator determines which operations are performed first. In expression **z=x+4\*y**, the **4\*y** is performed first because **\*** has higher precedence than **+** and **=**. The addition is performed second because **+** has higher precedence than **=**. The assignment **=** is performed last. Sometimes we use parentheses to clarify the meaning of the expression, even when they are not needed. Therefore, the line **z=x+4\*y;** could have been written as **z=(x+4\*y);** **z=(x+4\*y);** or **z=(x+(4\*y));**

The second issue is the **associativity**. Associativity determines the left to right or right to left order of evaluation when multiple operations of equal precedence are combined. For example **+** and **-** have the same precedence, so how do we evaluate the following?

**z = y-2+x;**

We know that **+** and **-** associate the left to right, this function is the same as **z=(y-2)+x;**. Meaning the subtraction is performed first because it is more to the left than the addition. Most operations associate left to right, but the following table illustrates that some operators associate right to left.

Precedence	Operators	Associativity
Highest	<b>0 [] . -&gt; ++ (postfix) -- (postfix)</b>	Left to right
	<b>++ (prefix) -- (prefix) ! ~ sizeof (type)</b>	Right to left
	<b>+ (unary)</b>	
	<b>- (unary) &amp; (address) * (dereference)</b>	
	<b>* / %</b>	Left to right
	<b>+</b> <b>-</b>	Left to right
	<b>&lt;&lt; &gt;&gt;</b>	Left to right
	<b>&lt; &lt;= &gt; &gt;=</b>	Left to right
	<b>== !=</b>	Left to right
	<b>&amp;</b>	Left to right

<code>^</code>	Left to right
<code> </code>	Left to right
<code>&amp;&amp;</code>	Left to right
<code>  </code>	Left to right
<code>? :</code>	Right to left
<code>= += -= *= /= %= &lt;=&gt;=  = &amp;=</code>	Right to left
<code>^=</code>	
<code>Lowest ,</code>	Left to right

**Table 2.4. Precedence and associativity determine the order of operation.**

**Observation:** When confused about precedence (and aren't we all) add parentheses to clarify the expression.

There are two types of **comments**. The first type explains how to use the software. These comments are usually placed at the top of the file, within the header file, or at the start of a function. The reader of these comments will be writing software that uses or calls these routines. The second type of comments assists a future programmer (ourselves included) in changing, debugging or extending these routines. We usually place these comments within the body of the functions. The comments on the right of each line are examples of the second type.

As mentioned earlier, the preprocessor directives begin with # in the first column. As the name implies preprocessor commands are processed first. I.e., the compiler passes through the program handling the preprocessor directives. Although there are many possibilities (assembly language, conditional compilation, interrupt service routines), I thought I'd mention the two most important ones early in the book. We create a macro using `#define` to define constants.

### `#define SIZE 10`

Basically, wherever **SIZE** is found as a token, it is replaced with the number **10**. A second important directive is the `#include`, which allows you to include another entire file at that position within the program. The `#include` directive will include the file named **tm4c123ge6pm.h** at this point in the program. This file will define all the I/O port names for the TM4C123. There are similar header files for each of the microcontrollers.

### `#include "tm4c123ge6pm.h"`

It is important for the C programmer to distinguish the two terms declaration and definition. A function **declaration** specifies its name, its input parameters and its output parameter. Another name for a function declaration is **prototype**. A data structure declaration specifies its type and format. On the other hand, a function definition specifies the exact sequence of operations to execute when it is called. A function **definition** will generate object code, which are machine instructions to be loaded into memory that perform the intended operations. A data structure definition will reserve space in memory for it. The confusing part is that the definition will repeat the declaration specifications. The C compiler performs just one pass through the code, and we must declare data/functions before we can access/invoke them. To run, of course, all data and functions must be defined. For example the declaration for the function **Random** could be written as

```
uint32_t Random(void);
```

We can see that the declaration shows us how to use the function, not how the function works. Because the C compilation is a one-pass process, an object must be declared or defined before it can be used in a statement. Actually the preprocessor performs the first pass through the program that handles the preprocessor directives. A top-down approach is to first declare a function, use the function, and lastly define the function as illustrated in Program 2.7. You see the high-level operations first.

```
uint32_t Random(void);
void main(void){uint32_t n;
    while(1){
        n = (Random()>>24)%52; // n varies from 0 to 51
    }
}
uint32_t M=1;
uint32_t Random(void){
    M = 1664525*M+1013904223;
    return(M);
}
```

Program 2.7. A main program that calls a function. In this case the declaration occurs first.

A bottom-up approach is to first define a function and then use the function as illustrated in Program 2.8. In the bottom-up approach, the definition both declares its structure and defines what it does. In bottom-up, you see the low-level operations first.

```
uint32_t M=1;
uint32_t Random(void){
    M = 1664525*M+1013904223;
    return(M);
}
void main(void){uint32_t n;
    while(1){
        n = (Random()>>24)%52; // n varies from 0 to 51
    }
}
```

Program 2.8. A main program that calls a function. In this case the definition occurs before its use.

A **function** is a sequence of operations that can be invoked from other places within the software. We can pass zero or more parameters into a function. A function can have zero or one output parameter. The add function in Program 2.9 has two 32-bit signed input parameters, and one 32-bit signed output parameter. The numbers in the comments are added to simplify our discussion.

```
int32_t add(int32_t x, int32_t y){ int32_t z; // 1
```

```

z = x+y;          // 2
if((x>0)&&(y>0)&&(z<0))z= 2147483647; // 3
if((x<0)&&(y<0)&&(z>0))z=-2147483648; // 4
return(z);        // 5
}
void main(void){ int32_t a,b;      // 6
a = add(2000,2000);           // 7
b = 0;                      // 8
while(1){                   // 9
    b = add(b,1);           // 10
}
}

```

Program 2.9. A function with two inputs and one output.

The interesting part is that after the operations within the function are performed, control returns to the place right after where the function was called. In C, execution begins with the main program. The execution sequence is shown below (numbers on right are line numbers):

```

1      void main(void){ int32_t a,b;      // 6
2          a = add(2000,2000);           // 7
3          int32_t add(long x, long y){ long z;      // 1
4              z = x+y;                // 2
5              if((x>0)&&(y>0)&&(z<0))z= 2147483647; // 3
6              if((x<0)&&(y<0)&&(z>0))z=-2147483648; // 4
7              return(z);              // 5
8          b = 0;                    // 8
9          while(1){               // 9
10             b = add(b,1);         // 10
11             int32_t add(int32_t x, int32_t y){ int32_t z; // 1
12                 z = x+y;          // 2
13                 if((x>0)&&(y>0)&&(z<0))z= 2147483647; // 3
14                 if((x<0)&&(y<0)&&(z>0))z=-2147483648; // 4
15                 return(z);        // 5
16             }                   // 11
17             while(1){            // 9
18                 b = add(b,1);       // 10
19                 int32_t add(int32_t x, int32_t y){ int32_t z; // 1
20                     z = x+y;          // 2
21                     if((x>0)&&(y>0)&&(z<0))z= 2147483647; // 3
22                     if((x<0)&&(y<0)&&(z>0))z=-2147483648; // 4
23                     return(z);        // 5
}

```

Notice that the return from the first call goes to line 8, while all the other returns go to line 11. The execution sequence repeats lines 9,10,1,2,3,4,5,11 indefinitely. The purpose of line 3 is to detect overflow, which occurs when two large positive numbers are added such that the sum is too big and does not fit into the 32-bit result. The purpose of line 4 is to detect underflow, which occurs when two large negative numbers are added such that the sum is too small and does not fit into the 32-bit result.

The functions in Programs 2.4, 2.5, and 2.6 had neither an input or output parameter. To specify the absence of a parameter we use the expression **void**.

The body of a function consists of a statement that performs the work. Normally the body is a compound statement between a {} pair. If the function has a return parameter, then all exit points must specify what to return.

Although C is a free field language, notice how the indenting has been added to programs in this book. The purpose of this indenting is to make the program easier to read. On the other hand since C is a free field language, the following two statements are quite different

Example A                   **if(n1>100) n2=100; n3=0;**  
Example B                   **if(n1>100) {n2=100; n3=0;}**

In both cases **n2=100;** is executed only if  $n1 > 100$ . In Example A, the statement **n3=0;** is always executed. In Example B, **n3=0;** is executed only if  $n1 > 100$ .

Variables declared outside of a function, like M in Program 2.8, are properly called **external variables** because they are defined outside of any function. While this is the standard term for these variables, it is confusing because there is another class of external variable, one that exists in a separately compiled source file. In this document we will refer to variables in the present source file as **globals**, and we will refer to variables defined in another file as **externals**. There are two reasons to employ **global variables**. The first reason is data permanence. The other reason is information sharing. Normally we pass information from one module to another explicitly using input and output parameters, but there are applications like interrupt programming where this method is unavailable. For these situations, one module can store data into a global while another module can view it.

**Local variables** are very important in C programming. They contain temporary information that is accessible only within a narrow scope. We can define local variables at the start of a compound statement. We call these local variables since they are known only to the block in which they appear, and to subordinate blocks. The variables a, b, and z in Program 2.9 are local. In C, local variable must be declared immediately after a brace { that begins a compound statement. Unlike globals, which are said to be static, locals are created dynamically when their block is entered, and they cease to exist when control leaves the block. Furthermore, local names supersede the names of globals and other locals declared at higher levels of nesting. Therefore, locals may be used freely without regard to the names of other variables. Although two global variables cannot use the same name, a local variable of one block can use the same name as a local variable in another block. Programming errors and confusion can be avoided by understanding these conventions.

# 2.9. Exercises

**2.1** In 16 words or less give definitions of the following terms

- |                              |                             |                                 |
|------------------------------|-----------------------------|---------------------------------|
| <b>a)</b> Embedded           | <b>b)</b> Device driver     | <b>c)</b> Top down              |
| <b>d)</b> Call graph         | <b>e)</b> Data flow graph   | <b>f)</b> Successive refinement |
| <b>g)</b> Dynamic efficiency | <b>h)</b> Static efficiency | <b>i)</b> Real time             |

**2.2** Define the following acronyms

- |                |                |               |
|----------------|----------------|---------------|
| <b>a)</b> HCI  | <b>b)</b> JTAG | <b>c)</b> NRE |
| <b>d)</b> GPIO | <b>e)</b> LED  | <b>f)</b> MMI |
| <b>g)</b> LCD  | <b>h)</b> ADC  | <b>i)</b> DAC |

**2.3** Explain the differences between specification, requirement, and constraint.

**2.4** In 16 words or less give definitions of the following debugging terms

- |                      |                               |                         |
|----------------------|-------------------------------|-------------------------|
| <b>a)</b> Instrument | <b>b)</b> Intrusiveness       | <b>c)</b> Black-box     |
| <b>d)</b> White-box  | <b>e)</b> Minimally intrusive | <b>f)</b> Stabilization |

**2.5** There is a microcomputer embedded in a vending machine. List three operations the software must perform.

**2.6** What is the difference between a microcomputer and a microcontroller?

**2.7** Consider a system with four modules named A, B, C, and D. Draw the call graph if A calls B, C calls A and D, and B calls C. Why can't the four individual modules in this system be tested one at a time?

**2.8** List 3 factors that we can use to evaluate the “goodness” of a program.

**D2.9** Draw a data flow graph of the thermostat algorithm developed in Exercise 1.27.

**D2.10** Draw a data flow graph of the cruise control algorithm developed in Exercise 1.28.

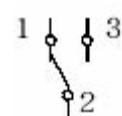
**D2.11** Draw a data flow graph of the incremental controller developed in Exercise 1.42.

**D2.12** Draw a data flow graph of the stepper motor controller developed in Exercise 1.43.

**D2.13** Consider an MP3 player, which has a memory storage device, an MP3 decoder (converts compressed data into raw form), buttons, an LCD, a USB link to a PC, dual DACs for stereo output, and stereo headphones. Draw a possible data flow graph for system.

**D2.14** You are given a double-pole switch that has three pins, labeled 1, 2, and 3. If the switch is not pressed, pins 1 and 2 are connected ( $0\Omega$  resistance) and pins 2 and 3 are not connected (infinite resistance). If the switch is pressed, pins 2 and 3 are connected ( $0\Omega$  resistance) and pins 1 and 2 are not connected (infinite resistance). Pins 1 and 3 are never connected (it is a break-before-make switch). Interface this switch to the microcontroller, such that input pin is high (3.3V) if the switch is pressed and input pin is low (0V) if the switch is not pressed. You do not need to debounce the

switch. Label all chip numbers and resistor values. No software is required. The best solution will not require any resistors.



**D2.15** Interface an LED that requires 1 mA at 2.5 V. A digital output high on the microcontroller turns on the LED. Assume  $V_{OH} = 3.2V$ . I.e., this interface is positive logic.

**D2.16** Interface an LED that requires 2 mA at 2.0 V. A digital output low on the microcontroller turns on the LED. I.e., this interface is negative logic. Because of the direct connection to the microcontroller, you should use 3.3V to power the LED (and not 5V). Assume  $V_{OL} = 0.3V$ .

**D2.17** Interface an LED that requires 15 mA at 2.5 V. Use a 7405 driver and a current limiting resistor. A digital output high on the microcontroller turns on the LED. I.e., this interface is positive logic. The 7405 output voltage  $V_{OL}$  is 0.5V. At this current you can safely use either 3.3V or 5V to power the LED.

**D2.18** Interface an LED that requires 30 mA at 1.5 V. Use a 7406 driver and a current limiting resistor. A digital output high the microcontroller turns on the LED. I.e., this interface is positive logic. The 7406 output voltage  $V_{OL}$  is 0.5V. At this large current, it is a good idea to replace the 3.3V connection in Figure 1.9 with 5V. This way the current comes directly from the USB cable (5V) and does not pass through the 3.3V regulator on the evaluation board.

**D2.19** Interface an LED that requires 100 mA at 1.5 V. Use a PN2222 transistor driver and a current limiting resistor. A digital output high the microcontroller turns on the LED. I.e., this interface is positive logic. The PN2222 output voltage  $V_{CE}$  is 0.3V. At this large current, it is a good idea to replace the 3.3V connection in Figure 1.9 with 5V. This way the current comes directly from the USB cable (5V) and does not pass through the 3.3V regulator on the evaluation board.

# 3. Introduction to the ARM Cortex -M Processor

## Chapter 3 objectives are to:

- Introduce Cortex -M processor architecture
- Present a subset of the Cortex -M core assembly language
- Define the memory-mapped I/O structure of the LM3S/TM4C family
- Describe addressing modes
- Present pseudo-operations
- Describe how software is developed

In this chapter we present a general description of the ARM Cortex -M processor. Rather than reproducing the voluminous details that can be found in the data sheets, we will present general concepts and give specific examples illustrating these concepts. In particular, we will define a subset of the instructions, with which the simple systems in this introductory book can be designed. The idea behind this subset is to provide functional completeness without concern for minimizing code size or execution speed. After reading this chapter, you should be able to look up and understand detailed specifics in the Cortex -M Technical Reference Manual. Data sheets can be found on the web sites of either ARM or the companies that make the microcontrollers, like Texas Instruments. Some of these data sheets are also posted on the web site accompanying this book. This web site can be found at <http://users.ece.utexas.edu/~valvano/arm>.

There are two reasons we must learn the assembly language of the computer which we are using. Sometimes, but not often, we wish to optimize our application for maximum execution speed or minimum memory size, and writing pieces of our code in assembly language is one approach to such optimizations. The most important reason, however, is that by observing the assembly code generated by the compiler for our C code we can truly understand what our software is doing. Based on this understanding, we can evaluate, debug, and optimize our system.

# 3.1. Cortex -M Architecture

Figure 3.1 shows a simplified block diagram of a microcontroller based on the ARM ® Cortex™-M processor. It is a **Harvard architecture** because it has separate data and instruction buses. The Cortex -M instruction set combines the high performance typical of a 32-bit processor with high code density typical of 8-bit and 16-bit microcontrollers. Instructions are fetched from flash ROM using the ICode bus. Data are exchanged with memory and I/O via the system bus interface. On the Cortex -M4 there is a second I/O bus for high-speed devices like USB. There are many sophisticated debugging features utilizing the DCode bus. The nested vectored interrupt controller (NVIC) manages **interrupts**, which are hardware-triggered software functions. Some internal peripherals, like the NVIC communicate directly with the processor via the private peripheral bus (PPB). The tight integration of the processor and interrupt controller provides fast execution of interrupt service routines (ISRs), dramatically reducing the interrupt latency.

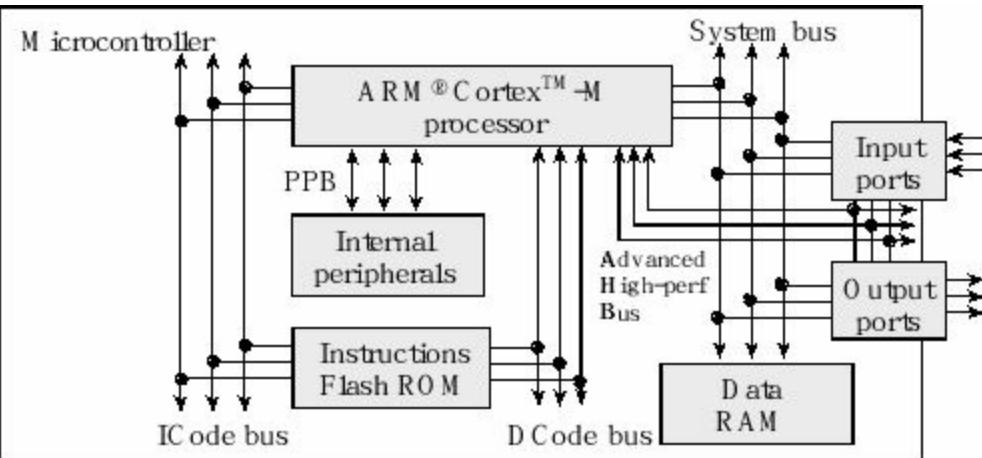


Figure 3.1. Harvard architecture of an ARM® Cortex -M-based microcontroller.

Even though data and instructions are fetched 32-bits at a time, each 8-bit byte has a unique address. This means memory and I/O ports are byte addressable. The processor can read or write 8-bit, 16-bit, or 32-bit data. Exactly how many bits are affected depends on the instruction, which we will see later in this chapter.

## 3.1.1. Registers

Registers are high-speed storage inside the processor. The registers are depicted in Figure 3.2. R0 to R12 are general purpose registers and contain either data or addresses. Register R13 (also called the stack pointer, SP) points to the top element of the stack. Register R14 (also called the link register, LR) is used to store the return location for functions. The LR is also used in a special way during exceptions, such as interrupts. Interrupts are covered in Chapter 9. Register R15 (also called the program counter, PC) points to the next instruction to be fetched from memory. The processor fetches an instruction using the PC and then increments the PC.

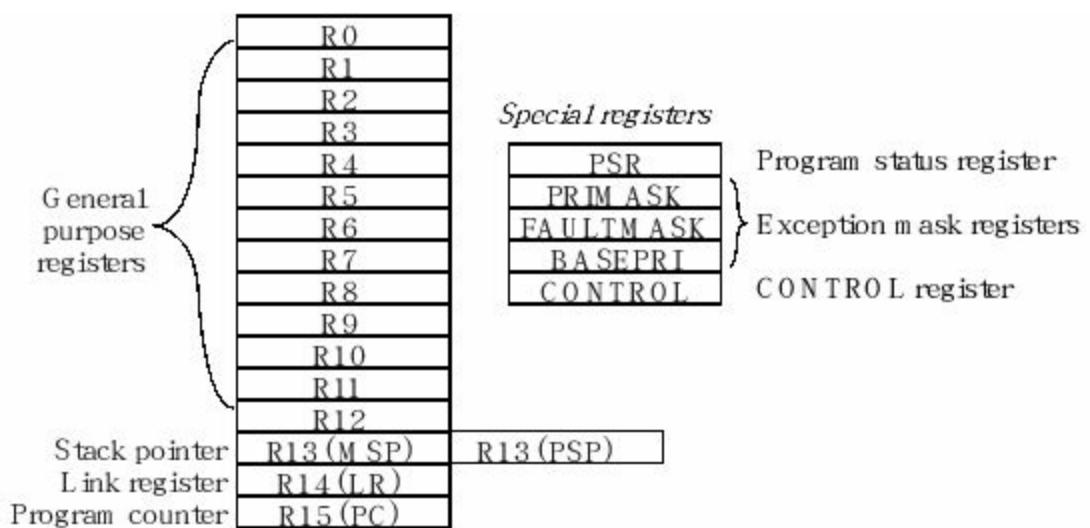


Figure 3.2. Registers on the ARM® Cortex™ -M processor.

The ARM Architecture **Procedure Call Standard**, AAPCS, part of the **ARM Application Binary Interface** (ABI), uses registers R0, R1, R2, and R3 to pass input parameters into a C function. Functions must preserve the values of registers R4–R11. Also according to AAPCS we place the return parameter in Register R0. AAPCS requires we push and pop an even number of registers to maintain an 8-byte alignment on the stack. In this book, the SP will always be the main stack pointer (MSP), not the Process Stack Pointer (PSP).

There are three status registers named Application Program Status Register (APSRR), the Interrupt Program Status Register (IPSR), and the Execution Program Status Register (EPSR) as shown in Figure 3.3. These registers can be accessed individually or in combination as the **Program Status Register** (PSR). The N, Z, V, C, and Q bits give information about the result of a previous ALU operation. In general, the **N bit** is set after an arithmetical or logical operation signifying whether or not the result is negative. Similarly, the **Z bit** is set if the result is zero. The **C bit** means carry and is set on an unsigned overflow, and the **V bit** signifies signed overflow. The **Q bit** indicates that “saturation” has occurred – while you might want to look it up, saturated arithmetic is beyond the scope of this book.

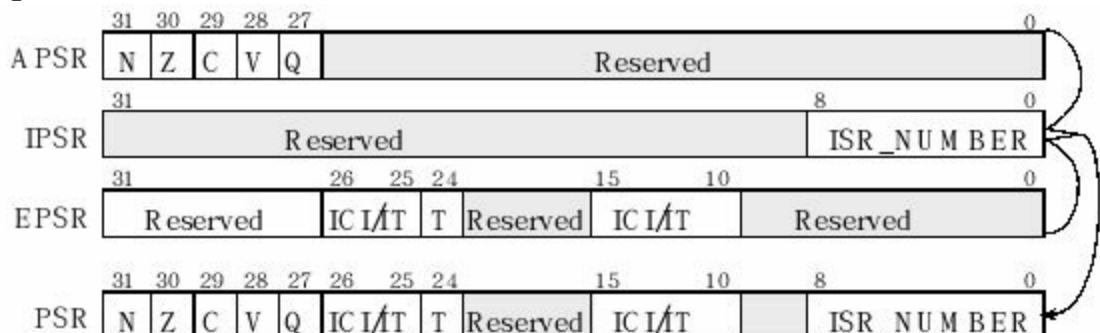


Figure 3.3. The program status register of the ARM® Cortex™ -M processor.

The **T** bit will always be 1, indicating the ARM® Cortex™-M processor is executing Thumb® instructions. The ISR\_NUMBER indicates which interrupt if any the processor is handling. Bit 0 of the special register **PRIMASK** is the interrupt mask bit. If this bit is 1, most interrupts and exceptions are not allowed. If the bit is 0, then interrupts are allowed. Bit 0 of the special register **FAULTMASK** is the fault mask bit. If this bit is 1, all interrupts and faults are not allowed. If the bit is 0, then interrupts and faults are allowed. The nonmaskable interrupt (NMI) is not affected by these mask bits. The **BASEPRI** register defines the priority of the executing software. It prevents interrupts with lower or equal priority but allows higher priority interrupts. For example if **BASEPRI** equals 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. A lower number means a higher priority interrupt. The details of interrupt processing will be presented in subsequent chapters.

### 3.1.2. Reset

A reset occurs immediately after power is applied and when the reset signal is asserted – this can usually be triggered by pushing the reset button available on most boards. After a reset, the processor is in thread mode, running at a privileged level, and using the MSP stack pointer. The 32-bit value at flash ROM location 0 is loaded into the SP. All stack accesses are word aligned. Thus, the least significant two bits of SP must be 0. A reset also loads the 32-bit value at location 4 into the PC. This value is called the reset vector. All instructions are halfword aligned. Thus, the least significant bit of PC must be 0. However, the assembler (or linker) will set the least significant bit in the reset vector, so the processor will properly initialize the Thumb bit (T) in the PSR. On the ARM® Cortex™-M processor, the T bit should always be set to 1. On reset, the processor initializes the LR to 0xFFFFFFFF.

### 3.1.3. Memory

Microcontrollers within the same family differ by the amount of memory and by the types of I/O modules. All LM3S and TM4C microcontrollers have a Cortex®-M processor. There are hundreds of members in this family; some of them are listed in Table 3.1.

Part number	RAM	Flash	I/O	I/O modules
LM3S811	8	64	32	PWM
LM3S1968	64	256	52	PWM
LM3S6965	64	256	42	PWM, Ethernet
LM3S8962	64	256	42	PWM, CAN, Ethernet, IEEE1588
TM4C1231C3PM	32	12	43	floating point, CAN, DMA
TM4C1233H6PM*	32	256	43	floating point, CAN, DMA, USB
TM4C123GH6PM	32	256	43	

				floating point, CAN, DMA, USB, PWM
TM4C123GH6ZRB	32	256	120	floating point, CAN, DMA, USB, PWM
TM4C1294NCPDT	256	1024	90	floating point, CAN, DMA, USB, PWM, Ethernet
	KiB	KiB	pins	

**Table 3.1. Memory and I/O modules (all have SysTick, RTC, timers, UART, I<sup>2</sup>C, SSI, and ADC). \*The TM4C1233H6PM is identical to the LM4F120H5QR, which is on the EK-LM4F120XL LaunchPad.**

The memory map of TM4C123 is illustrated in Figure 3.4. Although specific for the TM4C123, all ARM® Cortex™-M microcontrollers have similar memory maps. In general, Flash ROM begins at address 0x0000.0000, RAM begins at 0x2000.0000, the peripheral I/O space is from 0x4000.0000 to 0x5FFFF.FFFF, and I/O modules on the private peripheral bus (PPB) exist from 0xE000.0000 to 0xE00F.FFFF. In particular, the only differences in the memory map for the various 180 members of the LM3S/TM4C family are the ending addresses of the flash and RAM. The M4 has an advanced high-performance bus (AHB). Having multiple buses means the processor can perform multiple tasks in parallel. The following is some of the tasks that can occur in parallel

ICode bus	Fetch opcodes from ROM
DCode bus	Read constant data from ROM
System bus	Read/write data from RAM or I/O, fetch opcode from RAM
PPB	Read/write data from internal peripherals like the NVIC
AHB	Read/write data from high-speed I/O and parallel ports (M4 only)

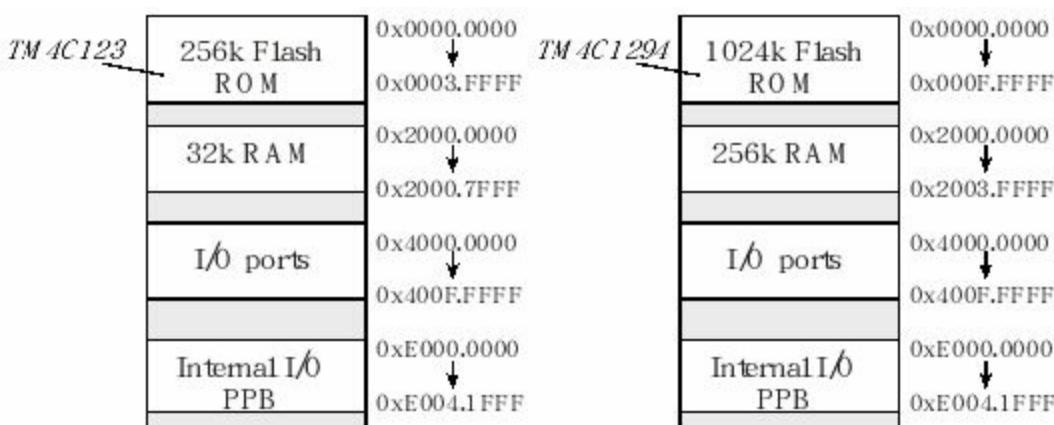


Figure 3.4. Memory maps of the TM4C123 and the TM4C1294.

When we store 16-bit data into memory it requires two bytes. Since the memory systems on most computers are byte addressable (a unique address for each byte), there are two possible ways to store in memory the two bytes that constitute the 16-bit data. Freescale microcomputers implement the **big endian** approach that stores the most significant byte at the lower address. Intel microcomputers implement the **little endian** approach that stores the least significant byte at the lower address. Cortex M microcontrollers use the little endian format. Many ARM processors are **biendian**, because they can be configured to efficiently handle both big and little endian data. Instruction fetches on the ARM are always little endian. Figure 3.5 shows two ways to store the 16-bit number 1000 (0x03E8) at locations 0x2000.0850 and 0x2000.0851. Computers must choose to use either the big or little endian approach when storing 32-bit numbers into memory that is byte (8-bit) addressable. Figure 3.6 shows the big and little endian formats that could be used to store the 32-bit number 0x12345678 at locations 0x2000.0850 through 0x2000.0853. Again the Cortex M uses little endian for 32-bit numbers.

Address	Data	Address	Data
0x2000.0850	0x03	0x2000.0850	0xE8
0x2000.0851	0xE8	0x2000.0851	0x03

Big Endian                      Little Endian

Figure 3.5. Example of big and little endian formats of a 16-bit number.

Address	Data	Address	Data
0x2000.0850	0x12	0x2000.0850	0x78
0x2000.0851	0x34	0x2000.0851	0x56
0x2000.0852	0x56	0x2000.0852	0x34
0x2000.0853	0x78	0x2000.0853	0x12

Big Endian                      Little Endian

Figure 3.6. Example of big and little endian formats of a 32-bit number.

In the previous two examples, we normally would not pick out individual bytes (e.g., the 0x12), but rather capture the entire multiple byte data as one nondivisible piece of information. On the other hand, if each byte in a multiple byte data structure is individually addressable, then both the big and little endian schemes store the data in first to last sequence. For example, if we wish to store the four ASCII characters ‘LM3S’, which is 0x4C4D3353 at locations 0x2000.0850 through 0x2000.0853, then the ASCII ‘L’=0x4C comes first in both big and little endian schemes, as illustrated in Figure 3.7.

Address	Data
0x2000.0850	0x4C
0x2000.0851	0x4D
0x2000.0852	0x33
0x2000.0853	0x53

Big Endian and Little Endian

Figure 3.7. Character strings are stored in the same for both big and little endian formats.

The terms “big and little endian” come from Jonathan Swift’s satire Gulliver’s Travels. In Swift’s book, a Big Endian refers to a person who cracks their egg on the big end. The Lilliputians were Little Endians because they insisted that the only proper way is to break an egg on the little end. The Lilliputians considered the Big Endians as inferiors. The Big and Little Endians fought a long and senseless war over the best way to crack an egg.

**Common Error:** An error will occur when data is stored in Big Endian by one computer and read in Little Endian format on another.

### 3.1.4. Operating Modes

The processor knows whether it is running in the foreground (i.e., the main program) or in the background (i.e., an interrupt service routine). ARM processors define the foreground as **thread mode**, and the background as **handler mode**. Switching between thread and handler modes occurs automatically. The processor begins in thread mode, signified by ISR\_NUMBER=0. Whenever it is servicing an interrupt it switches to handler mode, signified by setting ISR\_NUMBER to specify which interrupt is being processed. All interrupt service routines run using the MSP. For simplicity all software in this book will use the main stack pointer (MSP).

## 3.2. The Software Development Process

In this book we will begin with assembly language, and then introduce C. However, the process described in this section applies to both assembly and C. Either the ARM Keil™ uVision® or the Texas Instruments **Code Composer Studio™ (CCStudio)** integrated development environment (IDE) can be used to develop software for the Cortex M microcontrollers. Both include an editor, assembler, compiler, and simulator. Furthermore, both can be used to download and debug software on a real microcontroller. Either way, the entire development process is contained in one application, as shown in Figure 3.8.

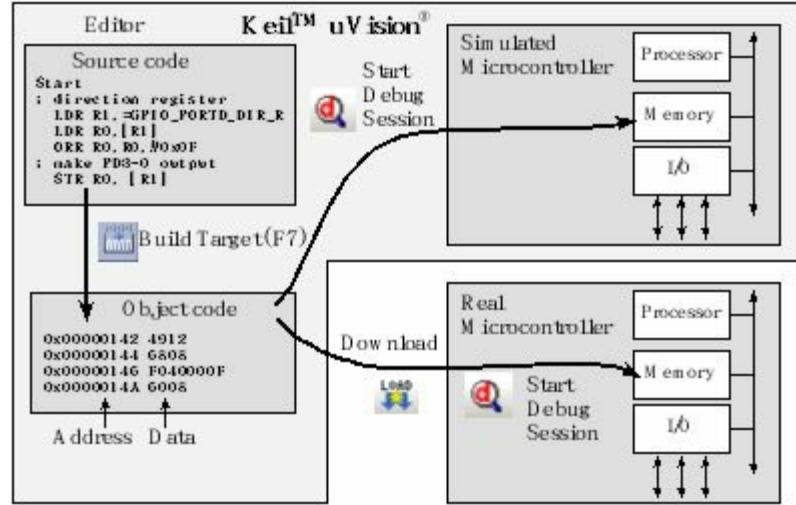


Figure 3.8. Assembly language or C development process.

To develop software, we first use an **editor** to create our **source code**. Source code contains specific set of sequential commands in human-readable-form. Next, we use an **assembler** or **compiler** to translate our source code into object code. On ARM Keil™uVision® we compile/assemble by executing the command **Project->Build Target** (short cut F7). Object code or machine instructions contains these same commands in machine-readable-form. Most assembly source code is one-to-one with the object code that is executed by the computer. For example, when programming in a high level language like C or Java, one line of a program can translate into several machine instructions. In contrast, one line of assembly code usually translates to exactly one machine instruction. The assembler/compiler may also produce a **listing file**, which is a human-readable output showing the addresses and object code that correspond to each line of the source program. The **target** specifies the platform on which we will be running the object code. When testing software with the simulator, we choose the **Simulator** as the target. When simulating, there is no need to download, we simply launch the simulator by executing the **Debug->Start Debug Session** command. The simulator is an easy and inexpensive way to get started on a project. However, its usefulness will diminish as the I/O becomes more complex.

In a real system, we choose the real microcontroller via its JTAG debugger as the target. In this way the object code is downloaded into the EEPROM of the microcontroller. Most microcontrollers contain built-in features that assist in programming their EEPROM. In particular, we will use the JTAG debugger connected via a USB cable to download and debug programs. The JTAG is both a

**loader** and a **debugger**. We program the EEPROM by executing the **Flash->Download** command. After downloading we can start the system by hitting the reset button on the board or we can debug it by executing **Debug->Start Debug Session** command in the uVision® IDE.

In contrast, the loader on a general purpose computer typically reads the object code from a file on a hard drive or CD and stores the code in RAM. When the program is run, instructions are fetched from RAM. Since RAM is volatile, the programs on a general purpose computer must be loaded each time the system is powered up.

For embedded systems, we typically perform initial testing on a simulator. The process for developing applications on real hardware is identical except the target is switched from a simulated microcontroller to the real microcontroller. It is best to have a programming reference manual handy when writing assembly language. These three reference manuals for the Cortex M3/M4 processor are available as pdf files and are posted on the book web site.

CortexM\_InstructionSet.pdf Cortex M -M3/M4 Instruction Set Technical User's Manual

CortexM4\_TRM\_r0p1.pdf Cortex M -M3/M4 Technical Reference Manual  
QuickReferenceCard.pdf ARM® and Thumb-2 Instruction Set Quick Reference Card

A description of each instruction can also be found by searching the Contents page of the help engine included with the ARM Keil™ uVision® or TI CCStudio applications. There are a lot of settings required to create a software project from scratch. I strongly suggest those new to the process first run lots of existing projects. Next, pick an existing project most like your intended solution, and then make a copy of that project. Finally, make modifications to the copy a little bit at a time as you morph the existing project into your solution. After each modification verify that it still runs. If you take a project that runs, make hundreds of changes to it, and then notice that it no longer runs, you will not know which of the many changes caused the failure.

The objective of software development is to translate a desired set of actions into an explicit set of commands that the computer executes. It is simple when the desired actions occur as a sequence of commands. Most of the time software asks the machine to execute one command after another. We write software as an ordered list of instructions one after another from the top to the bottom of the page, and the machine executes them in this order. The complexity occurs when we need to make decisions (branch), execute subtasks (functions), and perform multiple tasks concurrently or in parallel (interrupts and distributed systems). To handle these complexities we use instructions that are exceptions to this “one after another” rule (Figure 3.9). As we will learn in Chapter 9, interrupt are triggered by hardware events, causing the software interrupt service routine (shown as Clock in Figure 3.9) to be executed. A **bus fault** occurs if the software accesses an unimplemented memory location or accesses an I/O that is not properly configured. Bus faults are usually caused by software bugs.

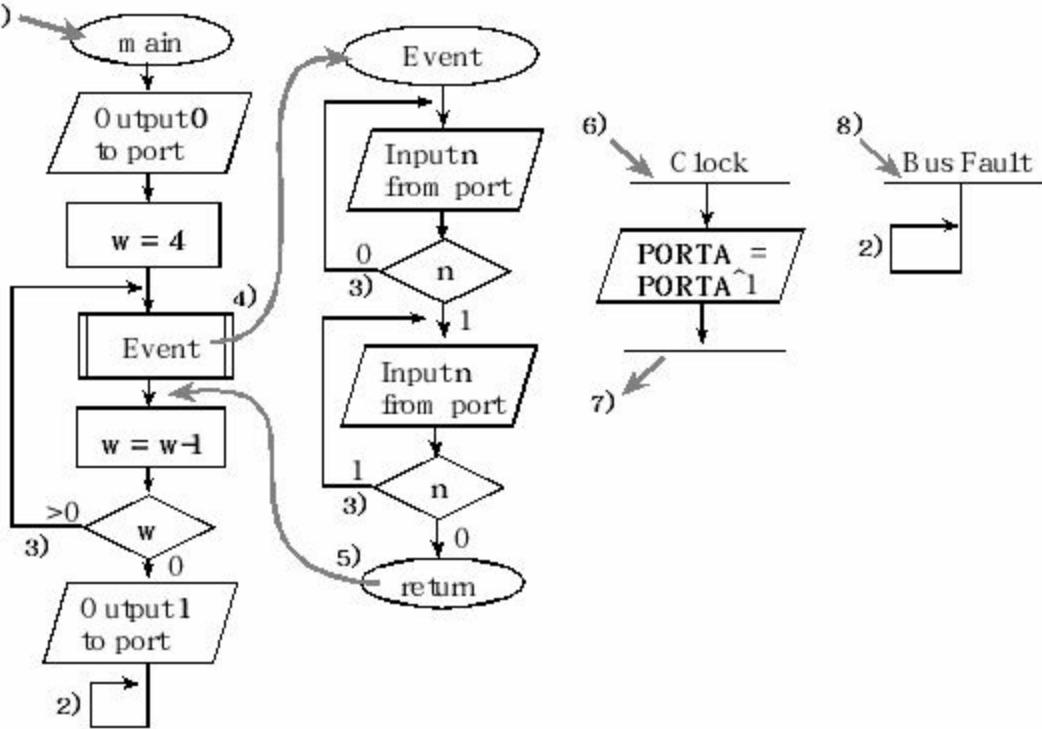


Figure 3.9. Flowchart showing examples of execution that breaks the “one after another rule”.

These exceptions, as numbered in Figure 3.9, include

- 1) The computer uses the reset vector to decide where to start (reset or power on).
- 2) There can be an unconditional branch causing the software to always go to a spot.
- 3) A conditional branch will go to a spot if a certain condition is true.
- 4) A function call will cause the software to go execute the code for that function.
- 5) A return from function will return to the place that called the function.
- 6) An interrupt will suspend execution and begin an interrupt service routine.
- 7) A return from interrupt will return to the place where it was before the interrupt.
- 8) A hardware or software mistake will cause a bus fault and stop execution.

# 3.3. ARM Cortex-M Assembly Language

This section focuses on the ARM® Cortex™-M assembly language. There are many ARM® processors, and this book focuses on Cortex® -M microcontrollers, which executes Thumb® instructions extended with Thumb-2 technology. This book will not describe in detail all the Thumb instructions. Rather, we focus on only a subset of the Thumb® instructions. This subset will be functionally complete without regard to minimizing code size or optimizing for execution speed. Furthermore, we will show general forms of instructions, but in many cases there are specific restrictions on which registers can be used and the sizes of the constants. For further details, please refer to the ARM® Cortex™-M Technical Reference Manual.

## 3.3.1. Syntax

Assembly language instructions have four fields separated by spaces or tabs. The **label field** is optional and starts in the first column and is used to identify the position in memory of the current instruction. You must choose a unique name for each label. The **opcode field** specifies the processor command to execute. The **operand field** specifies where to find the data to execute the instruction. Thumb instructions have 0, 1, 2, 3, or 4 operands, separated by commas. The **comment field** is also optional and is ignored by the assembler, but it allows you to describe the software making it easier to understand. You can add optional spaces between operands in the operand field. However, a semicolon must separate the operand and comment fields. Good programmers add comments to explain the software.

Label	Opcode	Operands	Comment
Func	MOV	R0, #100	; this sets R0 to 100
BX LR		; this is a function return	

**Observation:** A good comment explains why an operation is being performed, how it is used, how it can be changed, or how it was debugged. A bad comment explains what the operation does. The comments in the above two assembly lines are examples of bad comments.

When describing assembly instructions we will use the following list of symbols

**Ra Rd Rm Rn Rt** and **Rt2** represent registers

{Rd,} represents an optional destination register

#imm12 represents a 12-bit constant, 0 to 4095

#imm16 represents a 16-bit constant, 0 to 65535

**operand2** represents the flexible second operand as described in Section 3.4.2

{cond} represents an optional logical condition as listed in Table 3.2

{type} encloses an optional data type as listed in Table 3.3

{S} is an optional specification that this instruction sets the condition code bits

**Rm {, shift}** specifies an optional shift on **Rm** as described in Section 3.4.2

**Rn {, #offset}** specifies an optional offset to **Rn** as described in Section 3.4.2

For example, the general description of the addition instruction

**ADD{cond} {Rd,} Rn, #imm12**

could refer to either of the following examples.

**ADD R0,#1 ; R0=R0+1**

**ADD R0,R1,#10 ; R0=R1+10**

Table 3.2 shows the conditions **{cond}** that we will use for conditional branching. **{cond}** used on other instructions must be part of an if-then (IT) block, explained in Section 3.5.

Suffix	Flags	Meaning
<b>EQ</b>	Z = 1	Equal
<b>NE</b>	Z = 0	Not equal
<b>CS or HS</b>	C = 1	Higher or same, unsigned $\geq$
<b>CC or LO</b>	C = 0	Lower, unsigned <
<b>MI</b>	N = 1	Negative
<b>PL</b>	N = 0	Positive or zero
<b>VS</b>	V = 1	Overflow
<b>VC</b>	V = 0	No overflow
<b>HI</b>	C = 1 and Z = 0	Higher, unsigned >
<b>LS</b>	C = 0 or Z = 1	Lower or same, unsigned $\leq$
<b>GE</b>	N = V	Greater than or equal, signed $\geq$
<b>LT</b>	N $\neq$ V	Less than, signed <
<b>GT</b>	Z = 0 and N = V	Greater than, signed >
<b>LE</b>	Z = 1 or N $\neq$ V	Less than or equal, signed $\leq$
<b>AL</b>	Can have any value	Always. This is the default when no suffix specified

**Table 3.2. Condition code suffixes used to optionally execution instruction.**

It is much better to add comments to explain how or even better why we do the action. Good comments also describe how the code was tested and identify limitations. But for now we are learning what the instruction is doing, so in this chapter comments will describe what the instruction does. The assembly **source code** is a text file (with Windows file extension **.s**) containing a list of instructions. If register R0 is an input parameter, the following is a function that will return in register R0 the value (100\*input+10).

**Func MOV R1,#100 ; R1=100**

**MUL R0,R0,R1 ; R0=100\*input**

**ADD R0,#10 ; R0=100\*input+10**

**BX LR ; return 100\*input+10**

The **assembler** translates assembly source code into **object code**, which are the machine instructions executed by the processor. All object code is halfword-aligned. This means instructions can be 16 or 32 bits wide, and the program counter bit 0 will always be 0. The **listing** is a text file containing a mixture of the object code generated by the assembler together with our original source code.

Address	Object code	Label	Opcode	Operand	comment
0x000005E2	F04F0164	Func	MOV	R1,#0x64 ; R1=100	
0x000005E6	FB00F001		MUL	R0,R0,R1 ; R0=100*input	
0x000005EA	F100000A		ADD	R0,R0,#0x0A ; R0=100*input+10	
0x000005EE	4770		BX	LR ; return 100*input+10	

When we **build** a project all files are assembled or compiled then linked together. The address values shown in the listing are relative to the particular file being assembled. When the entire project is built, the files are linked together, and the **linker** decides exactly where in memory everything will be. After building the project, it can be downloaded, which programs the object code into flash ROM. You are allowed to load and execute software out of RAM. But for an embedded system, we typically place executable instructions into nonvolatile flash ROM. The listing you see in the debugger will specify the absolute address showing you exactly where in memory your variables and instructions exist.

### 3.3.2. Addressing Modes and Operands

A fundamental issue in program development is the differentiation between data and address. When we put the number 1000 into Register R0, whether this is data or address depends on how the 1000 is used. To run efficiently, we try to keep frequently accessed information in registers. However, we need to access memory to fetch parameters or save results. The **addressing mode** is the format the instruction uses to specify the memory location to read or write data. The addressing mode is associated more specifically with the operands, and a single instruction could exercise multiple addressing modes for each of the operands. When the import is obvious though, we will use the expression “the addressing mode of the instruction”, rather than “the addressing mode of an operand in an instruction”. All instructions begin by fetching the machine instruction (op code and operand) pointed to by the PC. When extended with Thumb-2 technology, some machine instructions are 16 bits wide, while others are 32 bits. Some instructions operate completely within the processor and require no memory data fetches. For example, the **ADD R1,R2** instruction performs R1+R2 and stores the sum back into R1. If the data is found in the instruction itself, like **MOV R0,#1**, the instruction uses **immediate addressing** mode. A register that contains the address or the location of the data is called a **pointer** or **index** register. **Indexed addressing** mode uses a register pointer to access memory. The addressing mode that uses the PC as the pointer is called **PC-relative addressing** mode. It is used for branching, for calling functions, and accessing constant data stored in ROM. The addressing mode is called PC relative because the machine code contains the address difference between where the program is now and the address to which the program will access. The **MOV** instruction will move data within the processor without accessing memory. The **LDR** instruction will read a 32-bit word from memory and place the data in a register. With PC-relative addressing, the assembler automatically calculates the correct PC offset.

Register. Most instructions operate on the registers. In general, data flows towards the op code (right to left). In other words, the register closest to the op code gets the result of the operation. In each of these instructions, the result goes into R2.

```
MOV R2,#100 ; R2=100, immediate addressing  
LDR R2,[R1] ; R2= value pointed to by R1  
ADD R2,R0 ; R2= R2+R0  
ADD R2,R0,R1 ; R2= R0+R1
```

Register list. The stack push and stack pop instructions can operate on one register or on a list of registers. SP is the same as R13, LR is the same as R14, and PC is the same as R15.

```
PUSH {LR} ; save LR on stack  
POP {LR} ; remove from stack and place in LR  
PUSH {R1-R3,LR} ; save R1,R2,R3 and link register  
POP {R1-R3,PC} ; restore R1,R2,R3 and PC
```

Immediate addressing. With immediate addressing mode, the data itself is contained in the instruction. Once the instruction is fetched no additional memory access cycles are required to get the data. Notice the number 100 (0x64) is embedded in the machine code of the instruction shown in Figure 3.10. Immediate addressing is only used to get, load, or read data. It will never be used with an instruction that stores to memory.

```
MOV R0,#100 ; R0=100, immediate addressing
```

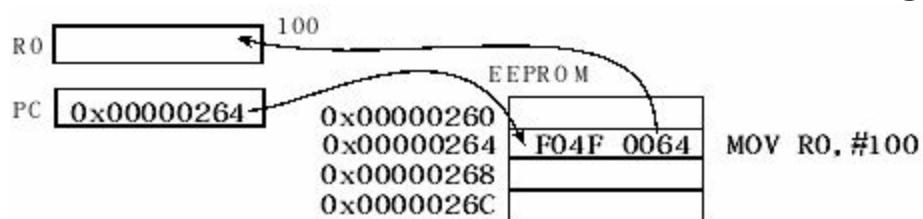


Figure 3.10. An example of immediate addressing mode, data is in the instruction.

Indexed addressing. With indexed addressing mode, the data is in memory and a register will contain a pointer to the data. Once the instruction is fetched, one or more additional memory access cycles are required to read or write the data. In these examples, R1 points to RAM. In this book, we will focus on just the first two forms of indexed addressing.

```
LDR R0,[R1] ; R0= value pointed to by R1  
LDR R0,[R1,#4] ; R0= word pointed to by R1+4  
LDR R0,[R1,#4]! ; first R1=R1+4, then R0= word pointed to by R1  
LDR R0,[R1],#4 ; R0= word pointed to by R1, then R1=R1+4  
LDR R0,[R1,R2] ; R0= word pointed to by R1+R2  
LDR R0,[R1,R2, LSL #2] ; R0= word pointed to by R1+4*R2
```

In Figure 3.11, R1 points to RAM, the instruction **LDR R0,[R1]** will read the 32-bit value pointed to by R1 and place it in R0. R1 could be pointing to any valid object in the memory map (i.e., RAM, ROM, or I/O), and R1 is not modified by this instruction.

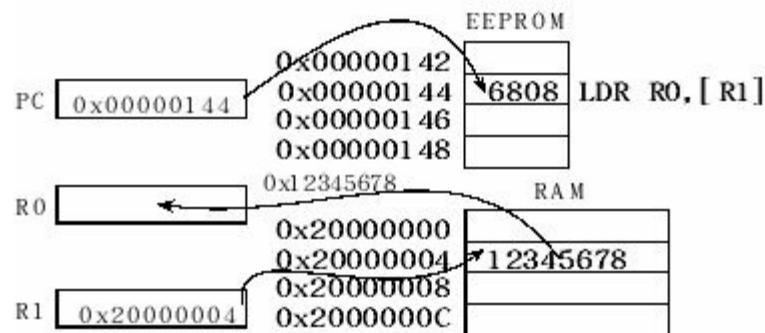


Figure 3.11. An example of indexed addressing mode, data is in memory.

In Figure 3.12, R1 points to RAM, the instruction **LDR R0,[R1,#4]** will read the 32-bit value pointed to by R1+4 and place it in R0. Even though the memory address is calculated as R1+4, the Register R1 itself is not modified by this instruction.

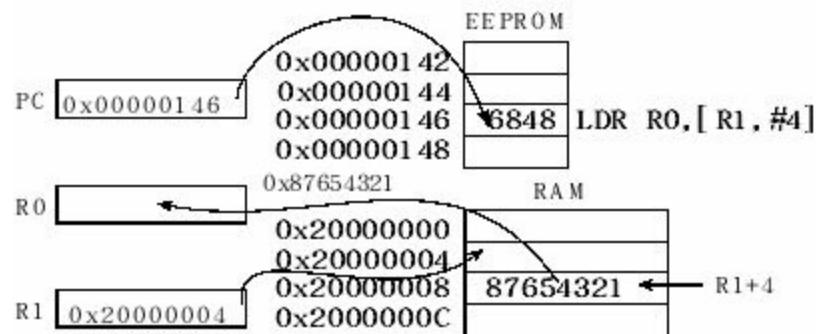


Figure 3.12. An example of indexed addressing mode with offset, data is in memory.

**PC-relative addressing.** PC-relative addressing is indexed addressing mode using the PC as the pointer. The PC always points to the instruction that will be fetched next, so changing the PC will cause the program to branch. A simple example of PC-relative addressing is the unconditional branch. In assembly language, we simply specify the label to which we wish to jump, and the assembler encodes the instruction with the appropriate PC-relative offset.

## B Location ; jump to Location, using PC-relative addressing

The same addressing mode is used for a function call. Upon executing the **BL** instruction, the return address is saved in the link register (LR). In assembly language, we simply specify the label defining the start of the function, and the assembler creates the appropriate PC-relative offset.

## BL Subroutine ; call Subroutine, using PC-relative addressing

Typically, it takes two instructions to access data in RAM or I/O. The first instruction uses PC-relative addressing to create a pointer to the object, and the second instruction accesses the memory using the pointer. We can use the =Something operand for any symbol defined by our program. In this case **Count** is the label defining a 32-bit variable in RAM.

**LDR R1,=Count ; R1 points to variable Count, using PC-relative**  
**LDR R0,[R1] ; R0= value pointed to by R1**

The operation caused by the above two **LDR** instructions is illustrated in Figure 3.13. Assume a 32-bit variable **Count** is located in the data space at RAM address 0x2000.0000. First, **LDR R1,=Count** makes R1 equal to 0x2000.0000. I.e., R1 points to **Count**. The assembler places a constant 0x2000.0000 in code space and translates the =**Count** into the correct PC-relative access to the constant (e.g., **LDR R1,[PC,#28]**). In this case, the constant 0x2000.0000, the address of **Count**, will be located at PC+28. Second, the **LDR R0,[R1]** instruction will dereference this pointer, bringing the 32-bit contents at location 0x2000.0000 into R0. Since **Count** is located at 0x2000.0000, these two instructions will read the value of **Count** into R0.

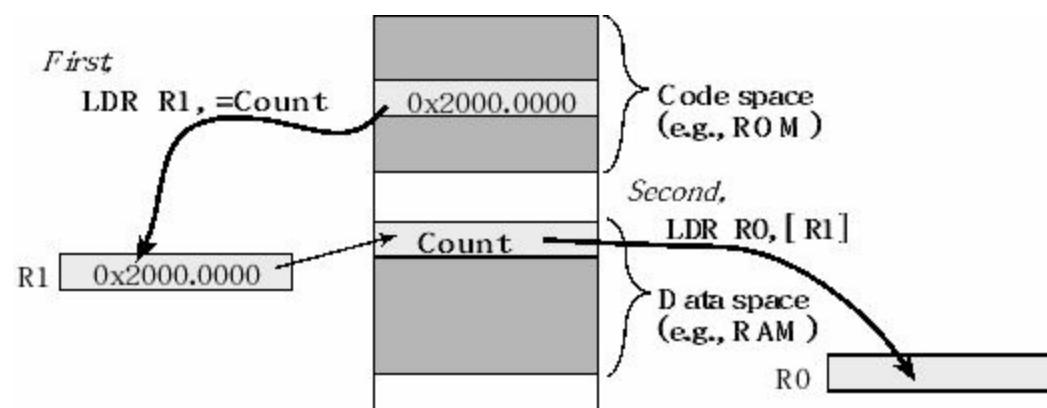


Figure 3.13. Indexed addressing using R1 as a register pointer to access memory. Data is moved into R0. Code space is where we place programs and data space is where we place variables.

Flexible second operand <op2>. Many instructions have a flexible second operand, shown as <op2> in the descriptions of the instruction. <op2> can be a constant or a register with optional shift. The flexible second operand can be a constant in the form #constant

**ADD Rd, Rn, #constant ;Rd = Rn+constant**

where **constant** is calculated as one of these four, X and Y are hexadecimal digits:

- Constant produced by shifting an unsigned 8-bit value left by any number of bits
- Constant of the form **0x00XY00XY**
- Constant of the form **0xXY00XY00**
- Constant of the form **0xXYXYXYXY**

We can also specify a flexible second operand in the form **Rm {,shift}**. If **Rd** is missing, **Rn** is also the destination. For example:

**ADD Rd, Rn, Rm {,shift} ;Rd = Rn+Rm**

**ADD Rn, Rm {,shift} ;Rn = Rn+Rm**

where **Rm** is the register holding the data for the second operand, and **shift** is an optional shift to be applied to **Rm**. The optional **shift** can be one of these five formats:

<b>ASR #n</b>	Arithmetic (signed) shift right <b>n</b> bits, $1 \leq n \leq 32$ .
<b>LSL #n</b>	Logical (unsigned) shift left <b>n</b> bits, $1 \leq n \leq 31$ .
<b>LSR #n</b>	Logical (unsigned) shift right <b>n</b> bits, $1 \leq n \leq 32$ .
<b>ROR #n</b>	Rotate right <b>n</b> bits, $1 \leq n \leq 31$ .
<b>RRX</b>	Rotate right one bit, with extend.

If we omit the shift, or specify **LSL #0**, the value of the flexible second operand is **Rm**. If we specify a shift, the shift is applied to the value in **Rm**, and the resulting 32-bit value is used by the instruction. However, the contents in the register **Rm** remain unchanged. For example,

**ADD R0,R1,LSL #4 ; R0 = R0 + R1\*16 (R1 unchanged)**

**ADD R0,R1,R2,ASR #4 ; signed R0 = R1 + R2/16 (R2 unchanged)**

An **aligned access** is an operation where a word-aligned address is used for a word, dual word, or multiple word access, or where a halfword-aligned address is used for a halfword access. Byte accesses are always aligned. The address of an aligned word access will have its bottom two bits equal to zero. An **unaligned** word access means we are accessing a 32-bit object (4 bytes) but the address is not evenly divisible by 4. The address of an aligned halfword access will have its bottom bit equal to zero. An unaligned halfword access means we are accessing a 16-bit object (2 bytes) but the address is not evenly divisible by 2. The Cortex -M processor supports unaligned access only for the following instructions:

<b>LDR</b>	Load 32-bit word
<b>LDRH</b>	Load 16-bit unsigned halfword
<b>LDRSH</b>	Load 16-bit signed halfword (sign extend bit 15 to bits 31-16)
<b>STR</b>	Store 32-bit word
<b>STRH</b>	Store 16-bit halfword

Transfers of one byte are allowed for the following instructions:

<b>LDRB</b>	Load 8-bit unsigned byte
<b>LDRSB</b>	Load 8-bit signed byte (sign extend bit 7 to bits 31-8)
<b>STRB</b>	Store 8-bit byte

When loading a 32-bit register with an 8- or 16-bit value, it is important to use the proper load, depending on whether the number being loaded is signed or unsigned. This determines what is loaded into the most significant bits of the register to ensure that the number keeps the same value when it is promoted to 32 bits. When loading an 8-bit unsigned number, the top 24 bits of the register will become zero. When loading an 8-bit signed number, the top 24 bits of the register will match bit 7 of the memory data (signed extend). Note that there is no such thing as a signed or unsigned store. For example, there is no **STRSH**; there is only **STRH**. This is because 8, 16, or all 32 bits of the register are stored to an 8-, 16-, or 32-bit location, respectively. No promotion occurs. This means that the value stored to memory can be different from the value located in the register if there is overflow. When using **STRB** to store an 8-bit number, be sure that the number in the register is 8 bits or less.

All other read and write memory operations generate a usage fault exception if they perform an unaligned access, and therefore their accesses must be address aligned. Also, unaligned accesses are usually slower than aligned accesses, and some areas of memory do not support unaligned accesses. But unaligned accesses may allow programs to use memory more efficiently at the cost of performance. The tradeoff between speed and size is a common motif.

**Observation:** This book adds a dot in the middle of 32-bit hexadecimal numbers (e.g., 0x2000.0000). This dot helps the reader visualize the number. However, this dot should not be used when writing actual software.

**Common Error:** Since not every instruction supports every addressing mode, it would be a mistake to use an addressing mode not available for that instruction.

**Checkpoint 3.1:** What is the addressing mode used for?

**Checkpoint 3.2:** Assume R3 equals 0x2000.0000 at the time **LDR R2,[R3,#8]** is executed. What address will be accessed? If R3 is changed, to what value will R3 become?

**Checkpoint 3.3:** Assume R3 equals 0x2000.0000 at the time **LDR R2,[R3],#8** is executed. What address will be accessed? If R3 is changed, to what value will R3 become?

### 3.3.3. Memory Access Instructions

This section presents mechanisms to read from and write to memory. As illustrated in Figure 3.13, to access memory we first establish a pointer to the object, then use indexed addressing. Usually code space is in ROM, but it is possible to assign code space to RAM. Data space is where we place variables. There are four types of memory objects, and typically we use a specific register to access them.

<u>Memory object type</u>	<u>Register</u>	<u>Example operand</u>
Constants in code space	PC	=Constant [PC,#28]
Local variables on the stack	SP	[SP,#0x04]
Global variables in RAM	R0 – R12	[R0]
I/O ports	R0 – R12	[R0]

The **ADR** instruction uses PC-relative addressing and is a handy way to generate a pointer to a constant in code space or an address within the program. The general form for **ADR** is

**ADR{cond} Rd, label**

where **{cond}** is an optional condition (see Table 3.2), **Rd** is the destination register, and **label** is a label within the code space within the range of -4095 to +4095 from the address in the PC. In reality, the assembler will generate an **ADD** or **SUB** instruction to calculate the desired address using an offset to the PC. **DCD** is an assembler directive that defines a 32-bit constant. We use it to create constants in code space (ROM). In the following example, after executing the **ADR** instruction, R5 points to **Pi**, and after executing the **LDR** instruction, R6 contains the data at **Pi**.

**Access ADR R5,Pi ;R5 points to Pi**

**LDR R6,[R5] ;R6 = 314159**

...

**BX LR**

**Pi DCD 314159**

We use the **LDR** instruction to load data from memory into a register. There is a special form of **LDR** which instructs the assembler to load a constant or address into a register. This is a “pseudo-instruction” and the assembler will output suitable instructions to generate the specified value in the register. This form for **LDR** is

**LDR{cond} Rd, =number**

**LDR{cond} Rd, =label**

where **{cond}** is an optional condition (see Table 3.2), **Rd** is the destination register, and **label** is a label anywhere in memory. Figure 3.13 illustrates how to create a pointer to a variable in RAM. A similar approach can be used to access I/O ports. On the TM4C family, Port A exists at address 0x4000.43FC. After executing the first **LDR** instruction, R5 equals 0x4000.43FC, which is a pointer to Port A, and after executing the second **LDR** instruction, R6 contains the value at Port A at the time it was read.

**Input LDR R5,=0x400043FC ;R5=0x400043FC, R5 points to PortA**

**LDR R6,[R5] ;Input from PortA into R6**

; ...

**BX LR**

The assembler translated the above assembly into this equivalent (the #16 represents the number of bytes between the **LDR R6,[R5]** instruction and the **DCD** definition).

**Input LDR R5,[PC,#16] ;PC+16 is the address of the DCD**

**LDR R6,[R5]**

; ...

**BX LR**

**DCD 0x400043FC**

We use the **LDR** instruction to load data from RAM to a register and the **STR** instruction to store data from a register to RAM. In real life, when we move a box to the basement, push a broom across the floor, load bullets into a gun, store spoons in a drawer, pop a candy into our mouth, or transfer employees to a new location, there is a physical object and the action changes the location of that object. Assembly language uses these same verbs, but the action will be different. In most cases, it creates a copy of the data and places the copy at the new location. In other words, since the original data still exists in the previous location, there are now two copies of the information. The exception to this memory-access-creates-two-copies-rule is a stack pop. When we pop data from the stack, it no longer exists on the stack leaving us just one copy. For example in Figure 3.13, the instruction **LDR R0,[R1]** loads the contents of the variable **Count** into R0. At this point, there are two copies of the data, the original in RAM and the copy in R0. If we next add 1 to R0, the two copies have different values. When we learn about interrupts in Chapter 9, we will take special care to handle shared information stored in global RAM, making sure we access the proper copy.

When accessing memory data, the type of data can be 8, 16, 32, or 64 bits wide. For 8-bit and 16-bit accesses the type can also be signed or unsigned. To specify the data type we add an optional modifier, as listed in Table 3.3. When we load an 8-bit or 16-bit unsigned value into a register, the most significant bits are filled with 0, called **zero pad**.

When we load an 8-bit or 16-bit signed value into a register, the sign bit of the value is filled into the most significant bits, called **sign extension**. This way, if we load an 8-bit -10 (0xF6) into a 32-bit register, we get the 32-bit -10 (0xFFFF.FFF6). When we store an 8-bit or 16-bit value, only the least significant bits are used.

{type}	Data type	Meaning
	32-bit word	0 to 4,294,967,295 or -2,147,483,648 to +2,147,483,647
<b>B</b>	Unsigned 8-bit byte	0 to 255, Zero pad to 32 bits on load
<b>SB</b>	Signed 8-bit byte	-128 to +127, Sign extend to 32 bits on load
<b>H</b>	Unsigned 16-bit halfword	0 to 65535, Zero pad to 32 bits on load
<b>SH</b>	Signed 16-bit halfword	-32768 to +32767, Sign extend to 32 bits on load
<b>D</b>	64-bit data	Uses two registers

**Table 3.3. Optional modifier to specify data type when accessing memory.**

Most of the addressing modes listed in the previous section can be used with load and store. The following lists the general form for some of the load and store instructions

**LDR{type}{cond} Rd, [Rn] ; load memory at [Rn] to Rd**

**STR{type}{cond} Rt, [Rn] ; store Rt to memory at [Rn]**

**LDR{type}{cond} Rd, [Rn, #n] ; load memory at [Rn+n] to Rd**

**STR{type}{cond} Rt, [Rn, #n] ; store Rt to memory [Rn+n]**

**LDR{type}{cond} Rd, [Rn,Rm,LSL #n] ; load memory at [Rn+Rm<<n] to Rd**

**STR{type}{cond} Rt, [Rn,Rm,LSL #n] ; store Rt to memory [Rn+Rm<<n]**

The move instructions get their data from the machine instruction or from within the processor and do not require additional memory access instructions.

**MOV{S}{cond} Rd, <op2> ; set Rd equal to the value specified by op2**

**MOV{cond} Rd, #im16 ; set Rd equal to im16, im16 is 0 to 65535**

**MVN{S}{cond} Rd, <op2> ; set Rd equal to the -value specified by op2**

### 3.3.4. Logical Operations

Software uses logical and shift operations to combine information, to extract information and to test information. A **unary operation** produces its result given a single input parameter. Examples of unary operations include negate, complement, increment, and decrement. In discrete digital logic, the **complement** operation is called a NOT gate; previously shown in Figure 1.11. The complement function is defined in Table 3.4.

A	$\sim A$
0	1
1	0

**Table 3.4. Logical complement.**

When designing digital logic we use gates, such as NOT AND OR, to convert individual input signals into individual output signals. However, when writing software using logic functions, we take two 32-bit numbers and perform 32 logic operations at the same time in a bit-wise fashion yielding one 32-bit result.

Boolean Logic has two states: true and false. As mentioned in Chapter 1, the false is 0, and the true state is any nonzero value.

A **binary operation** produces a single result given two inputs. The **logical and (&)** operation yields a true result if both input parameters are true. The **logical or (|)** operation yields a true result if either input parameter is true. The **exclusive or (^)** operation yields a true result if exactly one input parameter is true. The logical operators are summarized in Table 3.5 and shown as digital gates in Figure 1.12. The logical instructions on the ARM Cortex-M processor take two inputs, one from a register and the other from the flexible second operand. These operations are performed in a bit-wise fashion on two 32-bit input parameters yielding one 32-bit output result. The result is stored into the destination register. For example, the calculation  $r = m \& n$  means each bit is calculated separately,  $r_{31} = m_{31} \& n_{31}$ ,  $r_{30} = m_{30} \& n_{30}$ , ...,  $r_0 = m_0 \& n_0$ .

In C, when we write **r=m&n;** **r=m|n;** **r=m^n;** the logical operation occurs in a bit-wise fashion as described by Table 3.5. However, in C we define the Boolean functions as **r=m&&n;** **r=m||n;** For Booleans, the operation occurs in a word-wise fashion. For example, **r=m&&n;** means **r** will become zero if either **m** is zero or **n** is zero. Conversely, **r** will become 1 if both **m** is nonzero and **n** is nonzero.

A <b>Rn</b>	B <b>Operand2</b>	<b>A&amp;B AND</b>	<b>A B ORR</b>	<b>A^B EOR</b>	<b>A&amp;(~B) BIC</b>	<b>A (~B) ORN</b>
0	0	0	0	0	0	1
0	1	0	1	1	0	0
1	0	0	1	1	1	1
1	1	1	1	0	0	1

**Table 3.5. Logical operations performed by the Cortex -M processor.**

All instructions place the result into the destination register **Rd**. If **Rd** is omitted, the result is placed into **Rn**, which is the register holding the first operand. If the optional **S** suffix is specified, the N and Z condition code bits are updated on the result of the operation. In the comments next to the instructions below, we use **op2** to represent the 32-bit value generated by the flexible second operand, **<op2>**. Some flexible second operands may affect the C bit. These logical instructions will leave the V bit unchanged.

<b>AND{S}{cond} {Rd,} Rn, &lt;op2&gt;</b>	<b>;Rd=Rn&amp;op2</b>
<b>ORR{S}{cond} {Rd,} Rn, &lt;op2&gt;</b>	<b>;Rd=Rn op2</b>
<b>EOR{S}{cond} {Rd,} Rn, &lt;op2&gt;</b>	<b>;Rd=Rn^op2</b>
<b>BIC{S}{cond} {Rd,} Rn, &lt;op2&gt;</b>	<b>;Rd=Rn&amp;(~op2)</b>
<b>ORN{S}{cond} {Rd,} Rn, &lt;op2&gt;</b>	<b>;Rd=Rn (~op2)</b>

For example, assume R1 is 0x12345678 and R2 is 0x87654321. The **ORR R0,R1,R2** will perform this operation, placing the 0x97755779 result in R0.

<b>R1</b>	0001 0010 0011 0100 0101 0110 0111 1000
<b>R2</b>	<u>1000 0111 0110 0101 0100 0011 0010 0001</u>
<b>ORR</b>	1001 0111 0111 0101 0101 0111 0111 1001

---

**Example 3.1:** Write code to set bit 0 in a 32-bit variable called **N**.

**Solution:** First, we perform a 32-bit read, bringing **N** into Register R0. Second we perform a logical OR setting bit 0, and lastly we store the result back into **N**.

<b>LDR R1, =N ; R1 = &amp;N (R1 points to // C implementation N)</b>	<b>N = N   0x00000001;</b>
<b>LDR R0, [R1] ; R0 = N</b>	
<b>ORR R0, R0, #1 ; R0 = N 1</b>	
<b>STR R0, [R1] ; N = N 1</b>	

Program 3.1. Example code showing a logical OR.

**Checkpoint 3.4:** If R1 is 0x12345678 and R2 is 0x87654321, what would R0 be after the instruction **AND R0,R1,R2** is executed? What would R0 be after **EORR0,R1,R2** ?

**Checkpoint 3.5:** Using just the 74HC gates shown in Figures 1.11, 1.12, and 1.13, design one-bit BIC and ORN circuits as defined in Table 3.5.

**Observation:** We use the logical OR to make bits become one, and we use the logical AND to make bits become zero.

### 3.3.5. Shift Operations

Like programming in C, the assembly shift instructions take two input parameters and yield one output result. In C, the left shift operator is `<<` and the right shift operator is `>>`. E.g., to left shift the value in M by N bits and store the result in R we execute: `R = M<<N`. Similarly, to right shift the value in M by N bits and store the result in R we execute: `R = M>>N`.

The **logical shift right** (LSR) is similar to an unsigned divide by  $2^n$ , where n is the number of bits shifted as shown in Figure 3.14. A zero is shifted into the most significant position, and the carry flag will hold the last bit shifted out. The right shift operations do not round. For example, a right shift by 3 bits is similar to divide by 8. However, 15 right-shifted three times ( $15>>3$ ) is 1, while  $15/8$  is much closer to 2. In general, the LSR discards bits shifted out, and the UDIV truncates towards 0. Thus, when using UDIV to divide unsigned numbers by a power of 2, UDIV and LSR yield identical results.

The **arithmetic shift right** (ASR) is similar to a signed divide by  $2^n$ . Notice that the sign bit is preserved, and the carry flag will hold the last bit shifted out. This right shift operation also does not round. Again, a right shift by 3 bits is similar to divide by 8. However, -9 right-shifted three times ( $-9>>3$ ) is -2, while implementing -9 divided by 8 using the SDIV instruction yields -1. In general, the ASR discards bits shifted out, and the SDIV truncates towards 0.

The **logical shift left** (LSL) operation works for both unsigned and signed multiply by  $2^n$ . A zero is shifted into the least significant position, and the carry bit will contain the last bit that was shifted out.

The two **rotate** operations can be used to create multiple-word shift functions. There is no rotate left instruction, because a rotate left 10 bits is the same as rotate right 22 bits.

All shift instructions place the result into the destination register **Rd** . **Rm** is the register holding the value to be shifted. The number of bits to shift is either in register **Rs** , or specified as a constant **n** . If the optional **S** suffix is specified, the **N** and **Z** condition code bits are updated on the result of the operation. The **C** bit is the carry out after the shift as shown in Figure 3.14. These shift instructions will leave the **V** bit unchanged.

**Observation:** Use logic shift for unsigned numbers and arithmetic shifts for signed numbers.

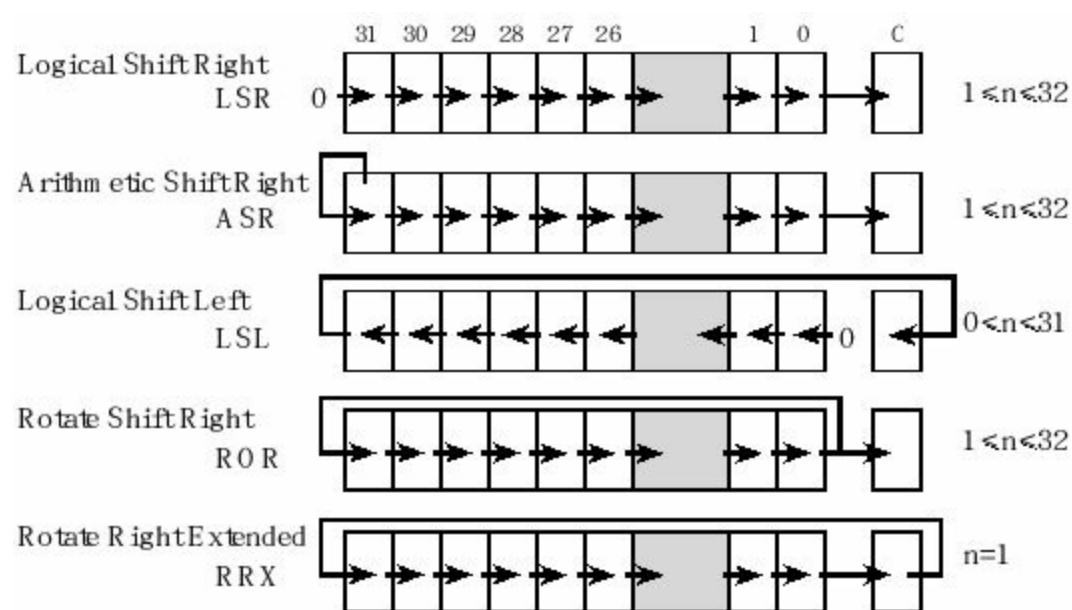


Figure 3.14. Shift operations.

**LSR{S}{cond} Rd, Rm, Rs ; logical shift right Rd=Rm>>Rs (unsigned)**

**LSR{S}{cond} Rd, Rm, #n ; logical shift right Rd=Rm>>n (unsigned)**

**ASR{S}{cond} Rd, Rm, Rs ; arithmetic shift right Rd=Rm>>Rs (signed)**

**ASR{S}{cond} Rd, Rm, #n ; arithmetic shift right Rd=Rm>>n (signed)**

**LSL{S}{cond} Rd, Rm, Rs ; shift left Rd=Rm<<Rs (signed, unsigned)**

**LSL{S}{cond} Rd, Rm, #n ; shift left Rd=Rm<<n (signed, unsigned)**

**ROR{S}{cond} Rd, Rm, Rs ; rotate right**

**ROR{S}{cond} Rd, Rm, #n ; rotate right**

**RXX{S}{cond} Rd, Rm ; rotate right 1 bit with extension**

**Example 3.2:** Write code that reads from variable **N**, shifts right twice, and stores the result in variable **M**. Both variables are 32-bit unsigned.

**Solution:** First, we perform a 32-bit read, bringing **N** into Register R1. Second we divide by 4 using a shift right operation, and lastly we store the result into **M**. Since the value gets smaller, no overflow can occur. If the variables were signed, then the **LSR** instruction should be replaced with an **ASR** instruction. In C, the shift right operator is `>>`

<pre>LDR R3, =N      ; R3 = &amp;N (R3 points to N)</pre>	<pre>// C implementation M = N&gt;&gt;2;</pre>
<pre>LDR R1, [R3]    ; R1 = N</pre>	
<pre>LSR R0, R1, #2  ; R0 = N&gt;&gt;2</pre>	
<pre>LDR R2, =M      ; R2 = &amp;M (R2 points to M)</pre>	
<pre>STR R0, [R2]    ; M = N&gt;&gt;2</pre>	

Program 3.2. Example code showing a right shift.

**Checkpoint 3.6:** Let **N** and **M** be 16-bit signed locations. Write code to implement  $M=4*N$ .

**Example 3.3:** Assume we have three 8-bit variables named **High** , **Low** , and **Result** . **High** and **Low** have 4 bits of data; each is a number from 0 to 15. Take these two 4-bit nibbles and combine them into one 8-bit value, storing the combination in **Result** .

**Solution:** The solution uses the **shift** operation to move the bits into position, then it uses the logical **OR** operation to combine the two parts into one number. We will assume both **High** and **Low** are bounded within the range of 0 to 15. The expression **High<<4** will perform four logical shift lefts. Registers R2, R3, and R4 point to (contain the address of) variables.

<b>LDR R2, =High ; R2 = &amp;High</b>	<b>// C implementation</b>
<b>LDR R3, =Low ; R3 = &amp;Low</b>	<b>Result = (High&lt;&lt;4) Low;</b>
<b>LDR R4, =Result ; R4 = &amp;Result</b>	
<b>LDRB R1, [R2] ; R1 = High</b>	
<b>LSL R0, R1, #4 ; R0 = R1&lt;&lt;4 =</b>	
<b>High&lt;&lt;4</b>	
<b>LDRB R1, [R3] ; R1 = Low</b>	
<b>ORR R0, R0, R1 ; R0 =</b>	
<b>(High&lt;&lt;4) Low</b>	
<b>STRB R0, [R4] ; Result =</b>	
<b>(High&lt;&lt;4) Low</b>	

Program 3.3. Example code showing a left shift.

To illustrate how the above program works, let  $0\ 0\ 0\ 0\ h_3\ h_2\ h_1\ h_0$  be the value of **High** , and let  $0\ 0\ 0\ 0\ l_3\ l_2\ l_1\ l_0$  be the value of **Low** . The **LDRB R1,[R2]** instruction brings the 8-bit **High** into Register R1. The **LSL R0,R1,#4** instruction moves the **High** into bit positions 4-7 of Register R0. The **LDRB R1, [R3]** instruction brings the 8-bit **Low** into Register R1. Finally, the **ORR R0,R0,R1** instruction combines **High** and **Low** , and the **STRB R0,[R4]** instruction stores the combination into **Result** .

$0\ 0\ 0\ 0\ h_3\ h_2\ h_1\ h_0$  value of **High**

$h_3\ h_2\ h_1\ h_0\ 0\ 0\ 0\ 0$  after four **LSL** s

$0\ 0\ 0\ 0\ l_3\ l_2\ l_1\ l_0$  value of **Low**

$h_3\ h_2\ h_1\ h_0\ l_3\ l_2\ l_1\ l_0$  result of the **ORR** instruction

### 3.3.6. Arithmetic Operations

When software executes arithmetic instructions, the operations are performed by digital hardware inside the processor. Even though the design of such logic is complex, we will present a brief introduction, in order to provide a little insight as to how the computer performs arithmetic. It is important to remember that arithmetic operations (addition, subtraction, multiplication, and division) have constraints when performed with finite precision on a processor. An overflow error occurs when the result of an arithmetic operation cannot fit into the finite precision of the register into which the result is to be stored.

For example, consider an 8-bit unsigned number system, where the numbers can range from 0 to 255. If we add two numbers together the result can range from 0 to 510, which is a 9-bit unsigned number. These numbers are similar to the numbers 1–12 on a clock, as drawn in Figure 3.15. If it is 11 o'clock and we wait 3 hours, it becomes 2 o'clock. Shown in the middle of Figure 3.15, if we add 64 to 224, the result becomes 32. In most cases, we would consider this an error. An unsigned overflow occurs during addition when we cross the 255-0 barrier (carry set on overflow). If we subtract two 8-bit unsigned numbers the result can range from -255 to +255, which is a 9-bit signed number. Subtraction moves in a counter-clockwise direction on the number wheel. As shown on the right side of Figure 3.15, if we subtract 64 from 32 (32-64), we get the incorrect result of 224. An unsigned overflow occurs during subtraction if we cross the 255-0 barrier in the other direction (carry clear on overflow).

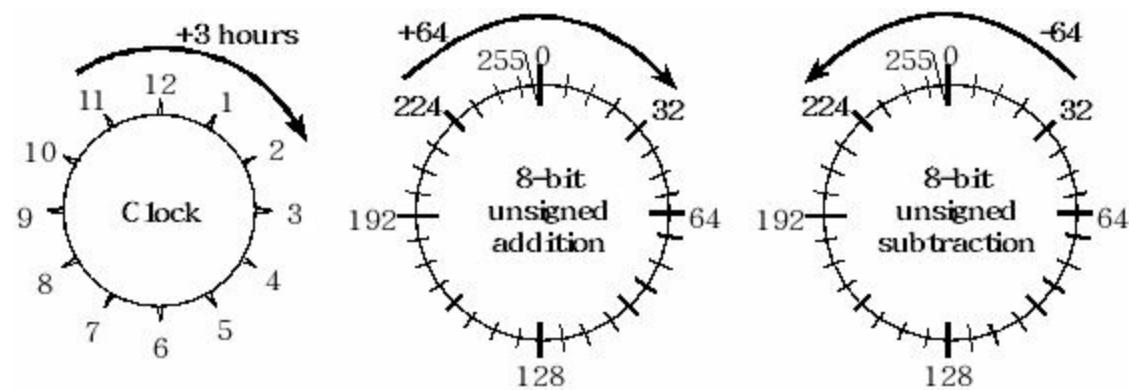


Figure 3.15. The carry bit is set on addition when crossing the 255-0 boundary. The carry bit is cleared on subtraction when crossing the 255-0 boundary.

Similarly, when two 32-bit numbers are added or subtracted, the result may not fit back into a 32-bit register. The same addition and subtraction hardware (instructions) can be used to operate on either unsigned or signed numbers. Although we use the same instructions, we must use separate overflow detection for signed and unsigned operations.

**Checkpoint 3.7:** How many bits does it take to store the result of two unsigned 32-bit numbers added together?

**Checkpoint 3.8:** How many bits does it take to store the result of two signed 32-bit numbers added together?

**Checkpoint 3.9:** Where is the barrier (discontinuity) on an unsigned 32-bit number wheel?

We begin the design of an adder circuit with a simple subcircuit called a binary full adder, as shown in Figure 3.16. There are two binary data inputs A, B, and a carry input,  $C_{in}$ . There is one data output,  $S_{out}$ , and one carry output,  $C_{out}$ . As shown in Table 3.6,  $C_{in}$ , A, and B are three independent binary inputs each of which could be 0 or 1. These three inputs are added together (the sum could be 0, 1, 2, or 3), and the result is encoded in the two-bit binary result with  $C_{out}$  as the most significant bit and  $S_{out}$  as the least significant bit.  $C_{out}$  is true if the sum is 2 or 3, and  $S_{out}$  is true if the sum is 1 or 3.

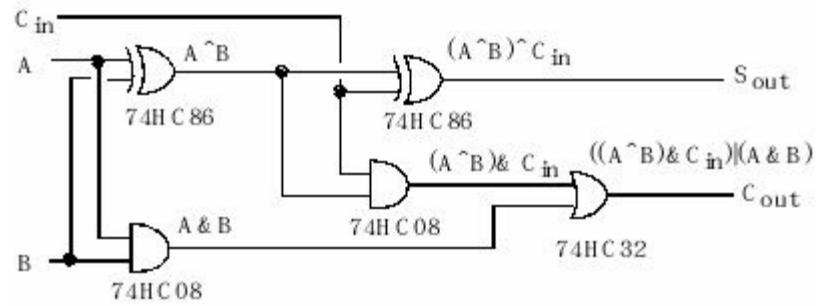


Figure 3.16. A binary full adder.

A	B	$C_{in}$	$A+B+C_{in}$	$C_{out}$	$S_{out}$
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

Table 3.6. Input/output response of a binary full adder.

Figure 3.17 shows an 8-bit adder formed by cascading eight binary full adders. Similarly, we build a 32-bit adder by cascading 32 binary full adders together. The carry into the 32-bit adder is zero, and the carry out will be saved in the carry bit.

For an 8-bit unsigned number, there are only 256 possible values, which are 0 to 255. We can think of the numbers as positions along a circle, like a clock. There is a discontinuity at the 0|255 interface; everywhere else adjacent numbers differ by  $\pm 1$  value. If we add two unsigned numbers, we start at the position of the first number and move in a clockwise direction the number of steps equal to the second number. As shown in Figure 3.18, if 96+64 is performed in 8-bit unsigned precision, the correct result of 160 is obtained. In this case, the carry bit will be 0 signifying the answer is correct. On the other hand, if 224+64 is performed in 8-bit unsigned precision, the incorrect result of 32 is obtained. In this case, the carry bit will be 1, signifying the answer is wrong.

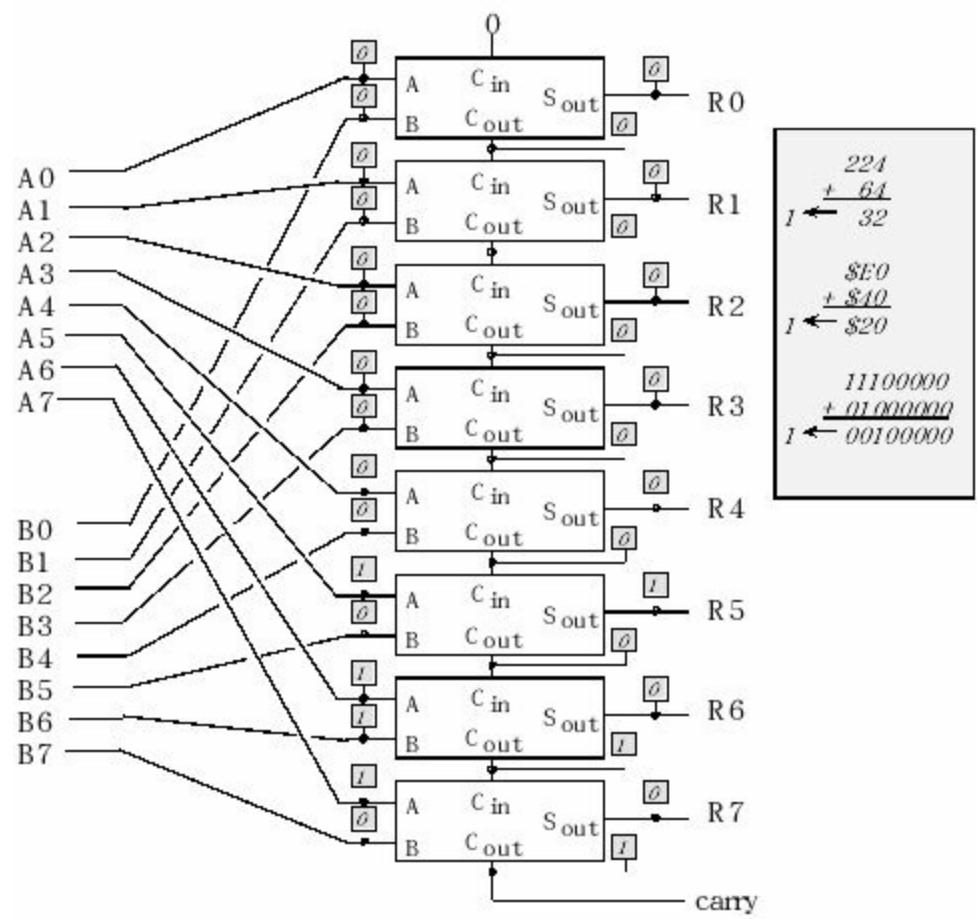


Figure 3.17. We make an 8-bit adder cascading eight binary full adders.

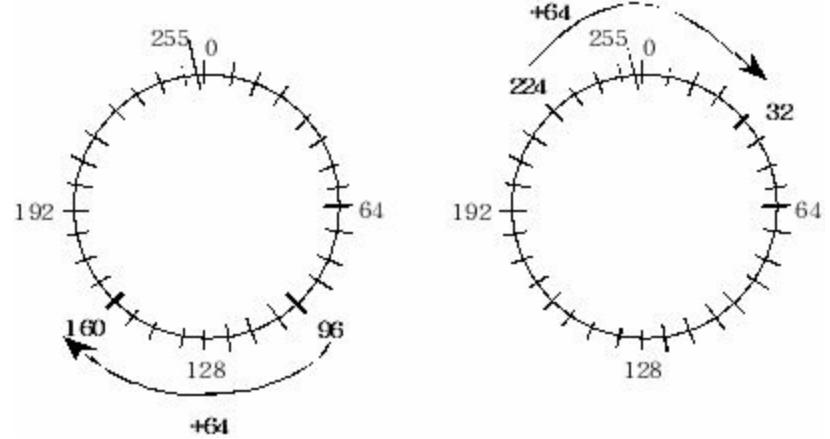


Figure 3.18. Number wheel showing  $96+64$  and  $224+64$ .

To calculate the negative of a two's complement number, we complement all of the bits and add 1. For example, the 8-bit binary representation for -100 is 10011100. The complement of this binary value is 01100011. When we add 1 to 01100011, we get the binary 01100100, which is the proper representation for 100. Using this fact, we can build an 8-bit subtractor ( $R=A-B$ ) by first negating B, then using eight binary full adders to add A plus -B, as shown in Figure 3.19. The carry into the 8-bit adder is one, and the carry out is saved in the carry bit. After a subtraction on the Cortex<sup>TM</sup>-M processor the carry is clear if an error occurred, and the carry is set if no error occurred and the answer is correct.

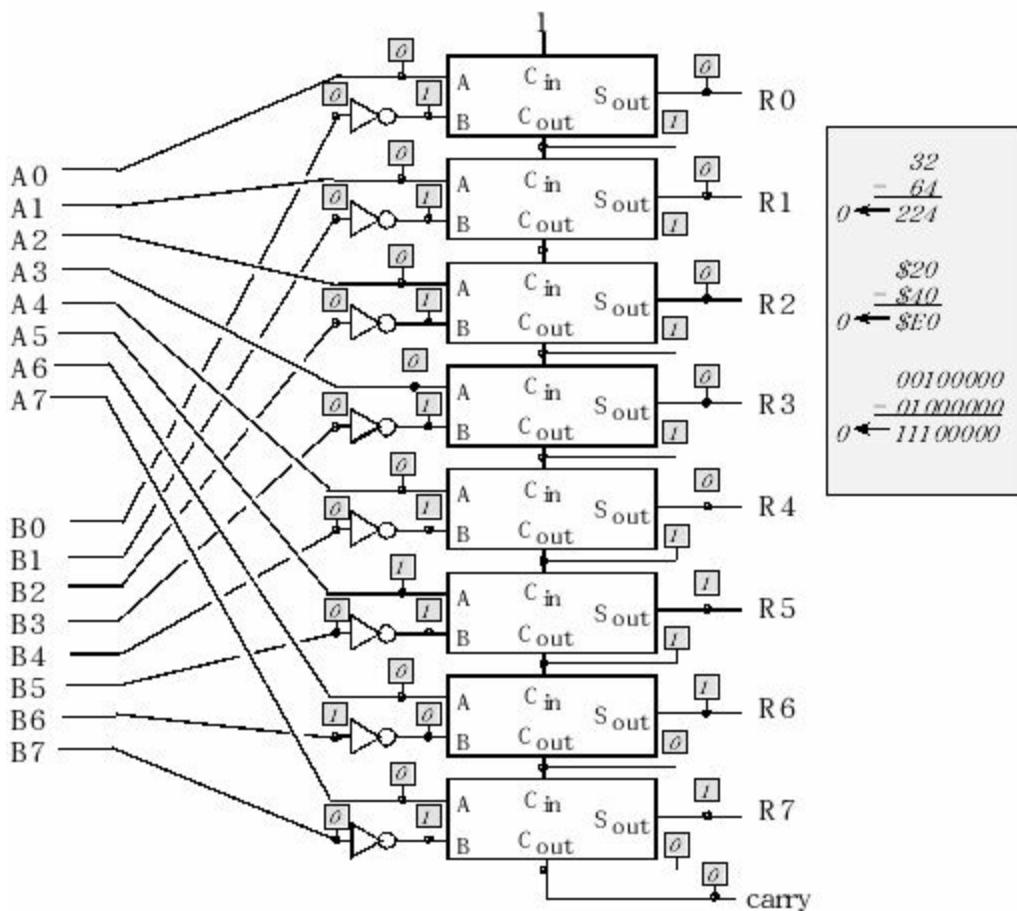


Figure 3.19. We make an 8-bit subtractor using eight binary full adders. Carry out is 0 in this case because the unsigned result is incorrect.

For subtraction, we start at the position of the first number a move in a counterclockwise direction the number of steps equal to the second number. As shown in Figure 3.20, if 160-64 is performed in 8-bit unsigned precision, the correct result of 96 is obtained (carry bit will be 1.) On the other hand, if 32-64 is performed in 8-bit unsigned precision, the incorrect result of 224 is obtained (carry bit will be 0.)

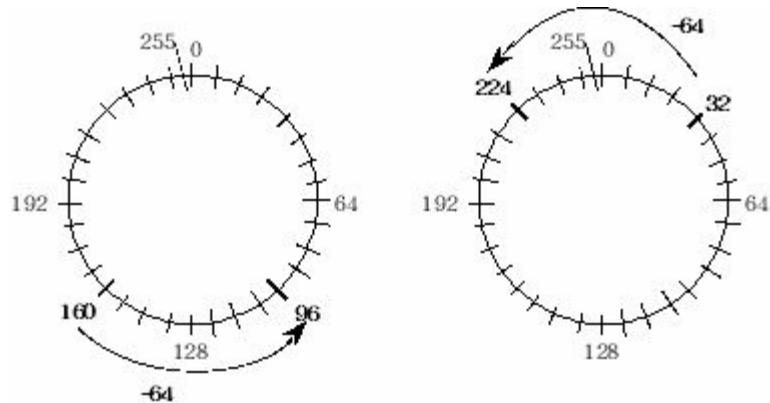


Figure 3.20. Number wheel showing 160-64 and 32-64.

In general, we see that the carry bit is set when we cross over from 255 to 0 while adding. The carry bit is cleared when we cross over from 0 to 255 while subtracting.

**Observation:** The carry bit, C, is set after an unsigned addition when the result is incorrect. The carry bit, C, is cleared after an unsigned subtraction when the result is incorrect.

For an 8-bit signed number, the possible values range from -128 to 127. Again there is a discontinuity, but this time it exists at the -128|127 interface, everywhere else adjacent numbers differ by  $\pm 1$ . The meanings of the numbers with bit 7=1 are different from unsigned, but we add and subtract signed numbers on the number wheel in a similar way (e.g., addition of a positive number moves clockwise.) Therefore, we can use the same hardware (Figures 3.17 and 3.19) to add and subtract two's complement signed numbers. The only difference is the carry out generated by the circuits do not represent an error when adding or subtracting two's complement signed numbers. Instead a new bit, called overflow or **V**, will be calculated to signify errors when operating on signed numbers. Adding a negative number is the same as subtracting a positive number hence this operation would cause a counterclockwise motion. As shown in Figure 3.21, if  $-32+64$  is performed, the correct result of 32 is obtained. In this case, the overflow bit will be 0 signifying the answer is correct. On the other hand, if  $96+64$  is performed, the incorrect result of -96 is obtained. In this case, the overflow bit will be 1 signifying the answer is wrong.

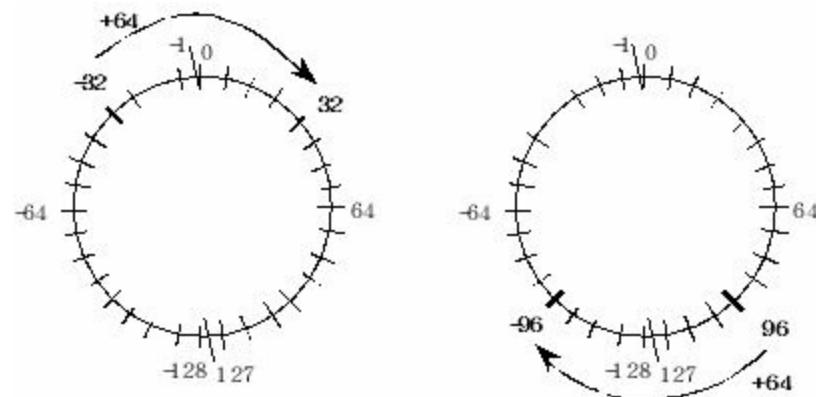


Figure 3.21. Number wheel showing  $-32+64$  and  $96+64$ .

For subtracting signed numbers, we again move in a counterclockwise direction. Subtracting a negative number is the same as adding a positive number; hence this operation would cause a clockwise motion. As shown in Figure 3.22, if  $32-64$  is performed, the correct result of -32 is obtained (overflow bit will be 0.) On the other hand, if  $-96-64$  is performed, the incorrect result of 96 is obtained (overflow bit will be 1.)

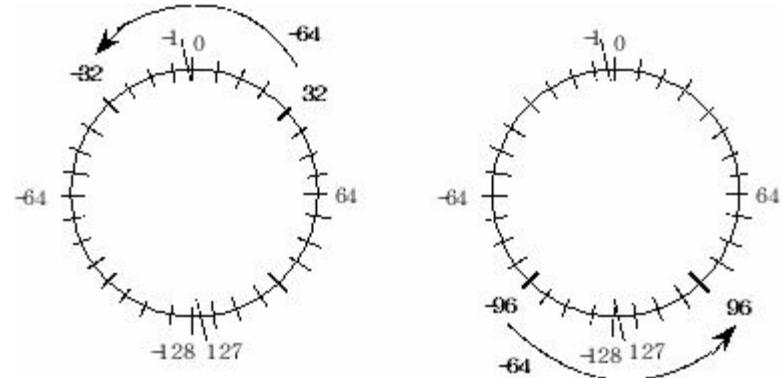


Figure 3.22. Number wheel showing  $32-64$  and  $-96-64$ .

In general, we see that the overflow bit, **V**, is set when we cross over from 127 to -128 while adding or cross over from -128 to 127 while subtracting.

**Observation:** The overflow bit, **V**, is set after a signed addition or subtraction when the result is incorrect.

In the arithmetic operations below, the 32-bit value can be specified by the #im12 constant or generated by the flexible second operand, <op2>. When Rd is absent, the result is placed back in Rn .

<b>ADD{S}{cond} {Rd,} Rn, &lt;op2&gt;</b>	<b>;Rd = Rn + op2</b>
<b>ADD{S}{cond} {Rd,} Rn, #im12</b>	<b>;Rd = Rn + im12</b>
<b>SUB{S}{cond} {Rd,} Rn, &lt;op2&gt;</b>	<b>;Rd = Rn - op2</b>
<b>SUB{S}{cond} {Rd,} Rn, #im12</b>	<b>;Rd = Rn - im12</b>
<b>RSB{S}{cond} {Rd,} Rn, &lt;op2&gt;</b>	<b>;Rd = op2 - Rn</b>
<b>RSB{S}{cond} {Rd,} Rn, #im12</b>	<b>;Rd = im12 - Rn</b>
<b>CMP{cond} Rn, &lt;op2&gt;</b>	<b>;Rn - op2</b>
<b>CMN{cond} Rn, &lt;op2&gt;</b>	<b>;Rn - (-op2)</b>

The compare instructions **CMP** and **CMN** do not save the result of the subtraction or addition but always set the condition code. The compare instructions are used to create conditional execution, such as if-then, for loops, and while loops. The compiler may use **RSB** or **CMN** to optimize execution speed.

If the optional **S** suffix is present, addition and subtraction set the condition code bits as shown in Table 3.7. The addition and subtraction instructions work for both signed and unsigned values. As designers, we must know in advance whether we have signed or unsigned numbers. The computer cannot tell from the binary which type it is, so it sets both C and V. Our job as programmers is to look at the C bit if the values are unsigned and look at the V bit if the values are signed.

Bit	Name	Meaning after addition or subtraction
N	Negative	Result is negative
Z	Zero	Result is zero
V	Overflow	Signed overflow
C	Carry	Unsigned overflow

**Table 3.7. Condition code bits contain the status of the previous arithmetic operation.**

If the two inputs to an addition operation are considered as unsigned, then the C bit (carry) will be set if the result does not fit. In other words, after an unsigned addition, the C bit is set if the answer is wrong. If the two inputs to a subtraction operation are considered as unsigned, then the C bit (carry) will be clear if the result does not fit. If the two inputs to an addition or subtraction operation are considered as signed, then the V bit (overflow) will be set if the result does not fit. In other words, after a signed addition, the V bit is set if the answer is wrong. If the result is unsigned, the N=1 means the result is greater than or equal to  $2^{31}$ . Conversely, if the result is signed, the N=1 means the result is negative.

Assuming the optional **S** suffix is present, condition code bits are set after the **addition**  $R=X+M$ , where X is initial register value, M is the flexible second operand or the #im12 constant, and R is the final register value. The N bit is set if the unsigned result is above 2147483647 ( $2^{31}-1$ ) or if the signed result is negative. The Z bit is set if the result is zero. The Z bit will be clear if any of the result bits are set.

N: result is negative

$$N = R_{31}$$

Z: result is zero

$$Z = \overline{R_{31}} \& \overline{R_{30}} \& L \& \overline{R_0}$$

If the V bit is set after a signed addition, then the result is incorrect because a signed overflow occurred. The first term of the following equation is true if you add two negative numbers together and get a positive result. The second term is true if you add two positive numbers together and get a negative result.

V: signed overflow

$$V = X_{31} \& M_{31} \& \overline{R_{31}} \mid \overline{X_{31}} \& \overline{M_{31}} \& R_{31}$$

If the C bit is set after an unsigned addition ( $R=X+M$ ), then the result is incorrect because an unsigned overflow occurred. The first term of the following equation is true if you add two numbers both above 2147483647. The second term is true if the M input is above 2147483647, but the result is less than or equal to 2147483647. The third term is true if the X input is above 2147483647, but the result is less than or equal to 2147483647.

C: unsigned overflow

$$C = X_{31} \& M_{31} \mid M_{31} \& \overline{R_{31}} \mid \overline{R_{31}} \& X_{31}$$

**Checkpoint 3.10:** Assume Register R0 is initially 0x7000.0000 and R1 is initially 0x2000.0000. After executing **adds R0,R0,R1** what is the value in Register R0, and the NZVC bits?

**Checkpoint 3.11:** Assume Register R0 is initially 1 and R1 is initially 0xFFFF.FFFF. After executing **adds R0,R0,R1** what is the value in Register R0, and the NZVC bits?

If the optional **S** suffix is present, condition code bits are set after the **subtraction**  $R=X-M$ , where X is initial register value, M is the flexible second operand or the #im12 constant, and R is the final register value.

N: result is negative

$$N = R_{31}$$

Z: result is zero

$$Z = \overline{R_{31}} \& \overline{R_{30}} \& L \& \overline{R_0}$$

If the V bit is set after a signed subtraction ( $R=X-M$ ), then the result is incorrect because a signed overflow occurred. The first term of the following equation is true if you subtract a negative number minus a positive number and get a positive result (a negative number minus a positive number should still be negative). The second term is true if you subtract a positive number minus a negative number and get a negative result (a positive number minus a negative number should still be positive).

V: signed overflow

$$V = X_{31} \& \overline{M_{31}} \& \overline{R_{31}} \mid \overline{X_{31}} \& M_{31} \& R_{31}$$

If the C bit is clear after an unsigned subtraction ( $R=X-M$ ), then the result is incorrect because an unsigned overflow occurred. The first term of the following equation is true if you subtracted a big number ( $M > 2147483647$ ) from a little number ( $X < 2147483647$ ). The second term is true if the M input is above 2147483647, but the result is greater than 2147483647. The third term is true if the X input is less than or equal to 2147483647, but the result is greater than 2147483647.

C: unsigned overflow

$$C = \overline{X_{31}} \& \overline{M_{31}} \mid M_{31} \& R_{31} \mid R_{31} \& \overline{X_{31}}$$

**Checkpoint 3.12:** Assume Register R0 is initially 100 and R1 is initially 200. After executing the instruction **subs R0,R0,R1** what is the value in Register R0, and the NZVC bits?

**Checkpoint 3.13:** Assume Register R0 is initially 100 and R1 is initially -200. After executing the instruction **subs R0,R0,R1** what is the value in Register R0, and the NZVC bits?

**Common Error:** Ignoring overflow (signed or unsigned) can result in significant errors.

**Observation:** Microcomputers have two sets of conditional branch instructions (if statements) that make program decisions based on either the C or V bit.

---

**Example 3.4:** Write code that reads from variable **N** adds 10 and stores the result in variable **M**. Both variables are 32-bit.

**Solution:** First, we perform a 32-bit read, bringing **N** into Register R1. Second we add 10, and lastly we store the result into **M**. Since the value gets larger, no overflow could occur. This solution ignores the overflow error.

```
LDR R3, =N ; R3 = &N (R3 points to // C implementation  
N) M = N+10;  
LDR R1, [R3] ; R1 = N  
ADD R0, R1, #10 ; R0 = N+10  
LDR R2, =M ; R2 = &M (R2 points  
to M)  
STR R0, [R2] ; M = N+10
```

Program 3.4. Example code showing a 32-bit add.

---

---

**Example 3.5:** Write code that reads from variable **N**, adds 10, and stores the result in variable **M**. Both variables are 8-bit unsigned.

**Solution:** First, we perform an 8-bit read, bringing **N** into Register R1. Since the register is 32 bits and the data 8 bits, the **LDRB** instruction will pad bits 32-8 to be zeros. Second we add 10, and lastly we store the result into **M**. Overflow cannot occur on the addition. However since the value gets larger, an error could occur when the 9-bit result is written back to an 8-bit variable. This solution ignores this error.

```
LDR R3, =N ; R3 = &N (R3 points to // C implementation  
N) M = N+10;  
LDRB R1, [R3] ; R1 = N  
ADD R0, R1, #10 ; R0 = N+10  
LDR R2, =M ; R2 = &M (R2 points  
to M)
```

## Program 3.5. Example code showing an addition of 8-bit variables.

**Checkpoint 3.14:** Modify Program 3.5 so it operates on unsigned 16-bit variables.

**Checkpoint 3.15:** Modify Program 3.5 so it operates on signed 8-bit variables.

**Observation:** Notice the C compiler uses the same + operator for 8-bit, 16-bit or 32-bit variables. Similarly, it will implement the appropriate operations for signed or unsigned numbers.

There are some instructions that operate only on signed numbers and others that work only for unsigned numbers. An error will occur if you use unsigned instructions after operating on signed numbers, and vice-versa. There are some applications where arithmetic errors are not possible. For example if we had two 8-bit unsigned numbers that we knew were in the range of 0 to 100, then no overflow is possible when they are added together.

Typically the numbers we are processing are either signed or unsigned (but not both), so we need only consider the corresponding C or V bit (but not both the C and V bits at the same time.) In other words, if the two numbers are unsigned, then we look at the C bit and ignore the V bit. Conversely, if the two numbers are signed, then we look at the V bit and ignore the C bit. There are two appropriate mechanisms to deal with the potential for arithmetic errors when adding and subtracting. The first mechanism, used by most compilers, is called **promotion**. Promotion involves increasing the precision of the input numbers, and performing the operation at that higher precision. An error can still occur if the result is stored back into the smaller precision. Fortunately, the program has the ability to test the intermediate result to see if it will fit into the smaller precision. To promote an unsigned number we add zeros to the left side. In a previous example, we added the unsigned 8-bit 224 to 64, and got the wrong result of 32. With promotion we first convert the two 8-bit numbers to 32 bits, then add

Decimal 8-bit	32-bit
224	1110,0000 0000,0000,0000,0000,0000,1110,0000
<u>+ 64</u>	<u>+0100,0000</u> <u>+0000,0000,0000,0000,0000,0100,0000</u>
288	0010,0000 0000,0000,0000,0000,0001,0010,0000

We can check the 32-bit intermediate result (e.g., 288) to see if the answer will fit back into the 8-bit result. In Figure 3.23, A and B are 8-bit unsigned inputs,  $A_{32}$ ,  $B_{32}$ , and  $R_{32}$  are 32-bit intermediate values, and R is an 8-bit unsigned output.

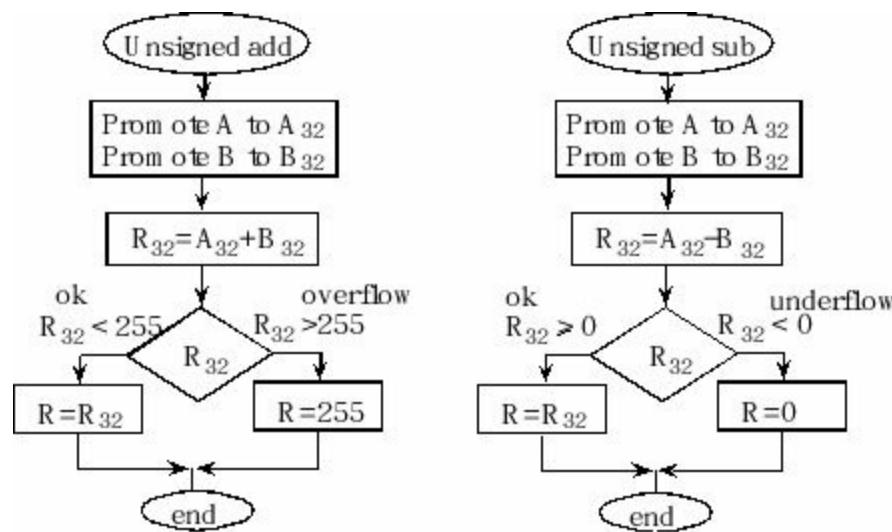


Figure 3.23. Flowcharts showing how to use promotion to detect and correct unsigned arithmetic errors.

In C, if we load a lower precision number into a higher precision variable, it will automatically promote. Unsigned promotion occurs moving an `uint8_t` value into an `uint_32` register or variable, and signed promotion occurs when moving from `int8_t` to `int_32`. No error occurs on promotion. However, if we load a higher precision number into a lower precision variable, it will automatically demote. For example writing a 32-bit `uint32_t` value into an 8-bit `uint_8` variable will discard the top 24 bits. Error can occur on demotion if the result does not fit. The C code in Program 3.6 adds and subtracts two 8-bit values, using promotion to detect for errors.

```

uint8_t A,B,R;
void add(void){ uint32_t result;
    result = A+B; // promote and perform 32-bit addition
    if(result>255){ // check for overflow
        result = 255; // yes, overflow occurred
    }
    R = result; // demote back to 8 bits
}
void sub(void){ int32_t result;
    result = A-B; // promote and perform 32-bit subtraction
    if(result<0){ // check for underflow
        result = 0; // yes, underflow occurred
    }
    R = result; // demote back to 8 bits
}

```

Program 3.6. Using promotion to detect and compensate for unsigned overflow errors.

**Observation:** When performing calculations on 8-bit or 16-bit numbers, most C compilers for the Cortex-M processor will first promote to 32 bits, perform the operations using 32-bit operations, and then demote the result back to the original precision.

**Common Error:** Even though most C compilers automatically promote to a higher precision during the intermediate calculations, they do not check for overflow when demoting the result back to the original format.

To promote a signed number, we duplicate the sign bit (called sign extension) as we add binary digits to the left side. Earlier, we performed the 8-bit signed operation -96-64 and got a signed overflow. With promotion we first convert the two numbers to 32 bits, then subtract

Decimal 8-bit	32-bit
-96 1010,0000	1111,1111,1111,1111,1111,1111,1010,0000
<u>-64 -0100,0000</u>	<u>-0000,0000,0000,0000,0000,0000,0100,0000</u>
-160 0110,0000	1111,1111,1111,1111,1111,1111,0110,0000

We can check the 32-bit intermediate result (e.g., -160) to see if the answer will fit back into the 8-bit result. In Figure 3.24, A and B are 8-bit signed inputs,  $A_{32}$ ,  $B_{32}$ , and  $R_{32}$  are 32-bit signed intermediate values, and R is an 8-bit signed output.

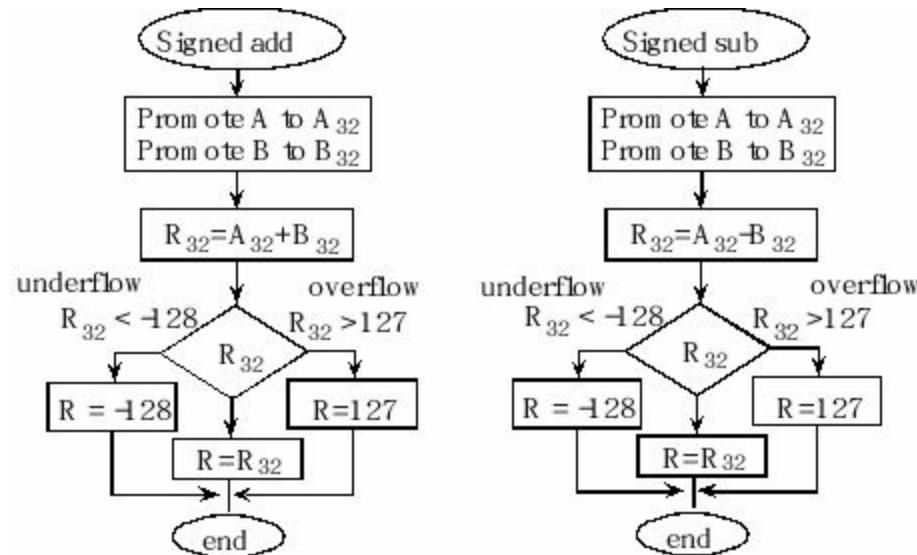


Figure 3.24. Flowcharts showing how to use promotion to detect and correct signed arithmetic errors.

The C code in Program 3.7 adds and subtracts two 8-bit signed numbers. The compiler will automatically promote A and B to signed 32-bit values before the addition.

```

int8_t A,B,R;
void add(void){ int32_t result;
    result = A+B; // promote and perform 32-bit addition
    if(result>127){ // check for overflow
        result = 127; // yes, overflow occurred, set to ceiling
    }
    if(result<-128){ // check for underflow
        result = -128; // yes, underflow occurred, set to floor
    }
    R = result; // demote back to 8 bits
}

```

```

}
void sub(void){ int32_t result;
  result = A-B; // promote and perform 32-bit subtraction
  if(result>127){ // check for overflow
    result = 127; // yes, overflow occurred, set to ceiling
  }
  if(result<-128){ // check for underflow
    result = -128; // yes, underflow occurred, set to floor
  }
  R = result; // demote back to 8 bits
}

```

Program 3.7. Using promotion to detect and compensate for signed overflow errors.

Notice in Program 3.7, when the result was too big the software set the result to maximum, and when the answer was too small the software set the result to minimum. We call this error handling technique **ceiling and floor**. ARM calls this mechanism **saturation** and is implemented with the **SSAT** and **USAT** instructions. Rather than discuss these two instructions, this section will focus on fundamental principles. It is analogous to movements inside a room. If we try to move up (add a positive number or subtract a negative number) the ceiling will prevent us from exceeding the bounds of the room. Similarly, if we try to move down (subtract a positive number or add a negative number) the floor will prevent us from going too low. The ceiling and floor prevent us from leaving the room. For our 32-bit addition and subtraction, we will prevent the 0 to 4294967295 and 4294967295 to 0 crossovers for unsigned operations and -2147483648 to +2147483647 and +2147483647 to -2147483648 crossovers for signed operations. These operations are described by the flowcharts in Figure 3.25. If the carry bit is set after an unsigned addition the result is adjusted to the largest possible unsigned number (ceiling). If the carry bit is clear after an unsigned subtraction, the result is adjusted to the smallest possible unsigned number (floor.).

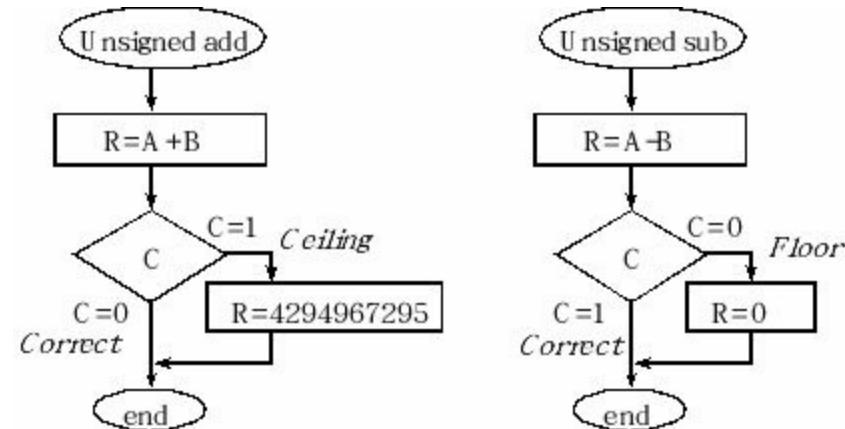


Figure 3.25. Flowcharts showing how to use overflow bits to detect and correct unsigned arithmetic errors.

There are no mechanisms in C to access the condition code bits of the processor. So, implementation of this approach must be performed in assembly language. Assume **A**, **B**, and **R** are three 32-bit (4-byte) global variables defined in RAM. The following assembly language adds two unsigned 8-bit numbers, using the algorithm presented in Figure 3.25.

```

LDR R2, =A      ; R2 = &A (pointer to A)
LDR R3, =B      ; R3 = &B (pointer to B)
LDR R4, =R      ; R4 = &R (pointer to R)
LDR R0, [R2]    ; R0 = A
LDR R1, [R3]    ; R1 = B
ADDS R0, R0, R1 ; R0 = A+B
BCC ok1          ; if C==0, no error
err1 MOV R0, #0xFFFFFFFF ; ceiling, R0 = 4,294,967,295
ok1 STR R0, [R4] ; R = A+B

```

The following assembly language subtracts two unsigned 32-bit numbers.

```

LDR R2, =A      ; R2 = &A (pointer to A)
LDR R3, =B      ; R3 = &B (pointer to B)
LDR R4, =R      ; R4 = &R (pointer to R)
LDR R0, [R2]    ; R0 = A
LDR R1, [R3]    ; R1 = B
SUBS R0, R0, R1 ; R0 = A-B
BCS ok2          ; if C==1, no error
err2 MOV R0, #0   ; floor, R0 = 0
ok2 STR R0, [R4] ; R = A-B

```

Signed addition and subtraction are described by the flowcharts in Figure 3.26. If the overflow bit is set after a signed operation the result is adjusted to the largest (ceiling) or smallest (floor) possible signed number depending on whether it was a -2147483648 to 2147483647 cross over (N=0) or 2147483647 to -2147483648 cross over (N=1). Notice that after a signed overflow, the sign bit of the result is always wrong because there was a cross over.

The following assembly language adds two signed 32-bit numbers, using the ceiling and floor algorithm presented in Figure 3.26.

```

LDR R2, =A      ; R2 = &A (pointer to A)
LDR R3, =B      ; R3 = &B (pointer to B)
LDR R4, =R      ; R4 = &R (pointer to R)
LDR R0, [R2]    ; R0 = A
LDR R1, [R3]    ; R1 = B
ADDS R0, R0, R1 ; R0 = A+B
BVC ok3          ; if V==0, skip to the end
err3 MOV R0, #0x7FFFFFFF ; R0 = 2,147,483,647
BMI ok3          ; if N==1, it was overflow
under3 MOV R0, #0x80000000 ; R0 = -2,147,483,648

```

ok3 STR R0, [R4]

; R = A+B

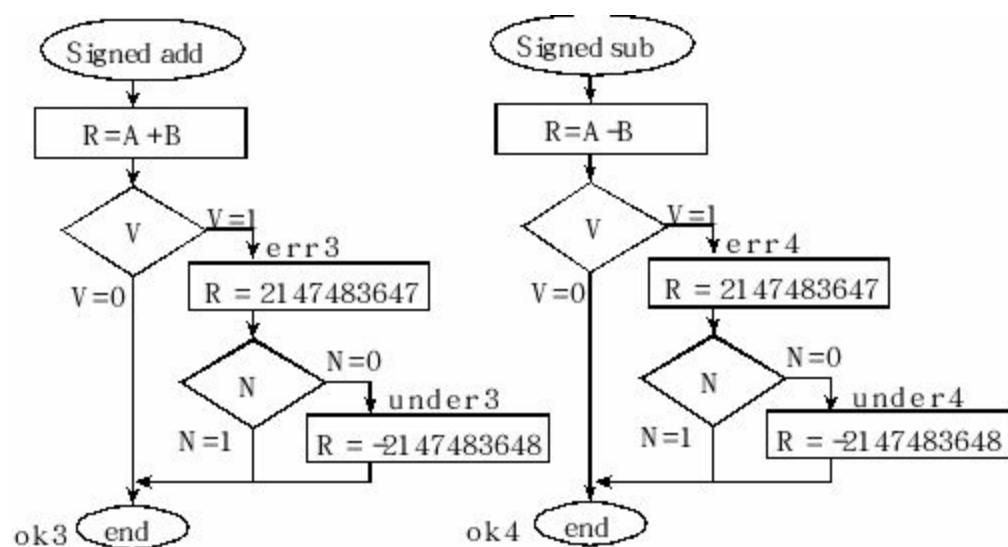


Figure 3.26. Flowcharts showing how to use overflow bits to detect and correct signed arithmetic errors.

**Checkpoint 3.16:** Write assembly to implement 32-bit subtraction with ceiling and floor.

Multiplication and division occurring in computers utilize a variety of complex algorithms to reduce power and minimize execution time. These algorithms are beyond the scope of this book. However, to illustrate binary multiplication we will present a very simple 8-bit unsigned algorithm, which uses a combination of shift and addition operations. Let A and B be two unsigned 8-bit numbers. The goal is to make  $R=A \cdot B$ . Simple calculations of  $0 \cdot 0=0$  and  $255 \cdot 255=65025$  illustrate the fact that the multiplication of two 8-bit numbers will fit into a 16-bit product. In general, an n-bit number multiplied by an m-bit number yields an  $(n+m)$ -bit product. First, we define one of the multiplicands in terms of its basis representation.

$$B = 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0$$

Next, we distribute multiplication over addition

$$R = A \cdot 128 \cdot b_7 + A \cdot 64 \cdot b_6 + A \cdot 32 \cdot b_5 + A \cdot 16 \cdot b_4 + A \cdot 8 \cdot b_3 + A \cdot 4 \cdot b_2 + A \cdot 2 \cdot b_1 + A \cdot b_0$$

We can simplify the equation leaving only one-bit shifts

$$R = 2 \cdot (2 \cdot (2 \cdot (2 \cdot (2 \cdot (A \cdot b_7 + A \cdot b_6) + A \cdot b_5) + A \cdot b_4) + A \cdot b_3) + A \cdot b_2) + A \cdot b_1) + A \cdot b_0$$

The multiplication by a power of 2 is a logical shift left, and the multiplication by a binary bit (0 or 1) is an add or no-add conditional. For an 8-bit multiply, we will use 16-bit shifts and additions, yielding a 16-bit product. Since the product, R, is a 16-bit unsigned number, there can be no overflow error in this 8 by 8 into 16-bit multiply.

**Checkpoint 3.17:** How many bits does it take to store the result of two unsigned 32-bit numbers multiplied together?

**Checkpoint 3.18:** How many bits does it take to store the result of two signed 32-bit numbers multiplied together?

Multiply( **MUL** ), multiply with accumulate( **MLA** ), and multiply with subtract( **MLS** ) use 32-bit operands and produce a 32-bit result. These three multiply instructions only save the bottom 32 bits of the result. They can be used for either signed or unsigned numbers, but no overflow flags are generated. If the **Rd** register is omitted, the **Rn** register is the destination. If the **S** suffix is added to **MUL** , then the Z and N bits are set according to the result. The division instructions do not set condition code flags and will round towards zero if the division does not evenly divide into an integer quotient.

<b>MUL{S}{cond} {Rd,} Rn, Rm</b>	<b>;Rd = Rn * Rm</b>
<b>MLA{cond} Rd, Rn, Rm, Ra</b>	<b>;Rd = Ra + Rn*Rm</b>
<b>MLS{cond} Rd, Rn, Rm, Ra</b>	<b>;Rd = Ra - Rn*Rm</b>
<b>UDIV{cond} {Rd,} Rn, Rm</b>	<b>;Rd = Rn/Rm unsigned</b>
<b>SDIV{cond} {Rd,} Rn, Rm</b>	<b>;Rd = Rn/Rm signed</b>

The following four multiply instructions use 32-bit operands and produce a 64-bit result. The two registers **RdLo** and **RdHi** contain the least significant and most significant parts respectively of the 64-bit result, signified as **Rd** . These multiply instructions do not set condition code flags.

<b>UMULL{cond} RdLo, RdHi, Rn, Rm</b>	<b>;Rd = Rn * Rm</b>
<b>SMULL{cond} RdLo, RdHi, Rn, Rm</b>	<b>;Rd = Rn * Rm</b>
<b>UMLAL{cond} RdLo, RdHi, Rn, Rm</b>	<b>;Rd = Rd + Rn*Rm</b>
<b>SMLAL{cond} RdLo, RdHi, Rn, Rm</b>	<b>;Rd = Rd + Rn*Rm</b>

**Checkpoint 3.19:** Can the 32 by 32 bit multiply instructions UMULL or SMULL overflow?

**Observation:** There is no overflow detection available for 32-bit multiplication.

---

**Example 3.6:** Write code that reads from variable **N** multiplies by 5, adds 25, and stores the result in variable **M**. Both variables are 32-bit.

**Solution:** First, we perform a 32-bit read, bringing **N** into Register R1. Second we multiply by 5 and add 10, and lastly we store the result into **M**. Since the value gets larger, overflow could occur. This solution ignores the overflow error.

<b>LDR R3, =N ; R3 = &amp;N (R3 points to N)</b>	<b>// C implementation</b>
<b>LDR R1, [R3] ; R1 = N</b>	<b>M = 5*N+25;</b>
<b>MOV R0, #5 ; R0 = 5</b>	
<b>MUL R1, R0, R1 ; R1 = 5*N</b>	
<b>MOV R0, #25 ; R0 = 25</b>	
<b>ADD R0, R0, R1 ; R0 = 25+5*N</b>	
<b>LDR R2, =M ; R2 = &amp;M (R2 points to M)</b>	
<b>STR R0, [R2] ; M = 25+5*N</b>	

## Program 3.8. Example code showing a 32-bit multiply and addition.

---

**Example 3.7:** Write code to convert a variable **N** ranging from 0 to 1023 into a variable **M**, which ranges from 0 to 3000. Essentially compute  $M = 2.93255 * N$ . Both variables are 32-bit.

**Solution:** First, we perform a 32-bit read, bringing **N** into Register R1. Second we multiply by 3000 and divide by 1023, and lastly we store the result into **M**. Since the input range is bounded ( $3000 * 1023 < 2^{32}$ ) no overflow error can occur.

```
LDR R3, =N ; R3 = &N (R3 points to // C implementation  
N) M = (3000*N)/1023;  
LDR R1, [R3] ; R1 = N  
MOV R0, #3000 ; R0 = 3000  
MUL R1, R0, R1 ; 3000*N (0 to  
3069000)  
MOV R0, #1023 ; R0 = 1023  
UDIV R0, R1, R0 ; R0 = R1/R0 =  
3000*N/1023  
LDR R2, =M ; R2 = &M (R2 points  
to M)  
STR R0, [R2] ; M = (3000*N)/1023
```

When dividing by an unsigned number, we can implement rounding by adding half the divisor prior to the division.

```
LDR R3, =N ; R3 = &N (R3 points to // C implementation  
N) M =  
LDR R1, [R3] ; R1 = N (3000*N+512)/1023;  
MOV R0, #3000 ; R0 = 3000  
MUL R1, R0, R1 ; 3000*N (0 to  
3069000)  
ADD R1, R1, #512; 3000*N + 512  
MOV R0, #1023 ; R0 = 1023  
UDIV R0, R1, R0 ; R0 = R1/R0 =  
3000*N/1023  
LDR R2, =M ; R2 = &M (R2 points  
to M)  
STR R0, [R2] ; M =  
(3000*N+512)/1023
```

To improve execution speed we can replace the division with a right shift. Notice that  $3003/1024$  is approximately equal to  $3000/1023$ . For example if  $N$  is 1023, the  $M$  will be  $(1023*3003+512)>>10$ , which equals exactly 3000. Again we add half the effective divisor to implement rounding.

<b>LDR R3, =N ; R3 = &amp;N (R3 points to N)</b>	<b>// C implementation</b>
<b>LDR R1, [R3] ; R1 = N</b>	<b>M =</b>
<b>MOV R0, #3003 ; R0 = 3003</b>	<b>(3003*N+512)&gt;&gt;10;</b>
<b>MUL R1, R0, R1 ; 3003*N</b>	
<b>ADD R1, R1, #512; 3003*N + 512</b>	
<b>LSR R0, R1, #10 ; (3003*N + 512)&gt;&gt;10</b>	
<b>LDR R2, =M ; R2 = &amp;M (R2 points to M)</b>	
<b>STR R0, [R2] ; M = (3003*N+512)&gt;&gt;10</b>	

**Checkpoint 3.20:** Give a single mathematical equation relating the dividend, divisor, quotient, and remainder. This equation gives a unique solution as long as you assume the remainder is strictly less than the divisor. Assume the sign of the remainder matches the sign of the dividend.

### 3.3.7. Stack

The **stack** is a last-in-first-out temporary storage. To create a stack, a block of RAM is allocated for this temporary storage. On the ARM ® Cortex™-M processor, the stack always operates on 32-bit data. The stack pointer (SP) points to the 32-bit data on the top of the stack. The stack grows downwards in memory as we push data on to it so, although we refer to the most recent item as the “top of the stack” it is actually the item stored at the lowest address! To push data on the stack, the stack pointer is first decremented by 4, and then the 32-bit information is stored at the address specified by SP. To pop data from the stack, the 32-bit information pointed to by SP is first retrieved, and then the stack pointer is incremented by 4. SP points to the last item pushed, which will also be the next item to be popped. The boxes in Figure 3.27 represent 32-bit storage elements in RAM. The grey boxes in the figure refer to actual data stored on the stack, and the white boxes refer to locations in memory that do not contain stack data. This figure illustrates how the stack is used to push the contents of Registers R0, R1, and R2 in that order. Assume Register R0 initially contains the value 1, R1 contains 2, and R2 contains 3. The drawing on the left shows the initial stack. The software executes these six instructions in this order:

PUSH {R0}  
PUSH {R1}  
PUSH {R2}  
POP {R3}  
POP {R4}  
POP {R5}

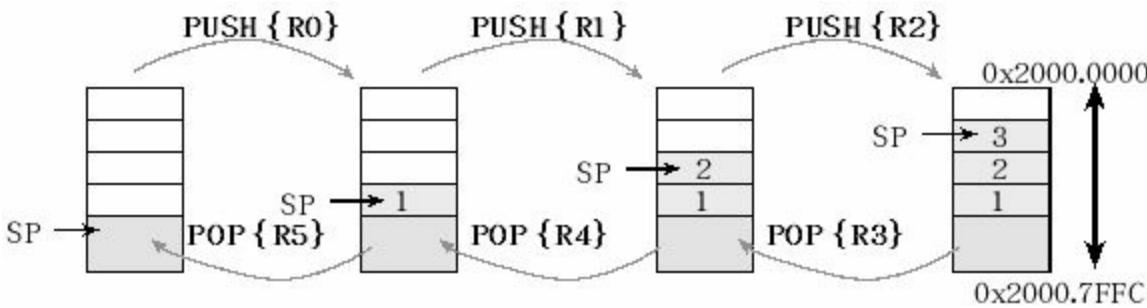


Figure 3.27. Stack picture showing three numbers first being pushed, then three numbers being popped. You are allowed to draw stack pictures so that the lowest address is on the top (like this one) or so that lowest address is on the bottom. The important matter is to be clear, accurate, and consistent.

The instruction **PUSH {R0}** saves the value of R0 on the stack. It first decrements SP by 4, and then it stores the contents of R0 into the memory location pointed to by SP. The right-most drawing shows the stack after the push occurs three times. The stack contains the numbers 1, 2, and 3, with 3 on top. The instruction **POP{R3}** retrieves data from the stack. It first moves the value from memory pointed to by SP into R3, and then it increments SP by 4. After the pop occurs three times the stack reverts to its original state and registers R3, R4, and R5 contain 3 2 1 respectively. We define the 32-bit word pointed to by SP as the **top** entry of the stack. If it exists, we define the 32-bit data immediately below the top, at  $SP+4$ , as **next** to top. Here are the rules one has to follow when using the stack:

1. Functions should have an equal number of pushes and pops
2. Stack accesses (push or pop) should not be performed outside the allocated area
3. Stack reads and writes should not be performed within the free area
4. Stack push should first decrement SP, then store the data
5. Stack pop should first read the data, and then increment SP

Functions that violate rule number 1 will probably crash when incorrect data are popped off at a later time. Violations of rule number 2 can be caused by a stack underflow or overflow. Overflow occurs when the number of elements becomes larger than the allocated space. Stack underflow is caused when there are more pops than pushes, and it is always the result of a software bug. A stack overflow can be caused by two reasons. If the software mistakenly pushes more than it pops, then the stack pointer will eventually overflow its bounds. Even when there is exactly one pop for each push, a stack overflow can occur if the stack is not allocated large enough. The processor will generate a **bus fault** when the software tries read from or write to an address that doesn't exist. If valid RAM exists below the stack then further stack operations will corrupt data in this memory.

Executing an interrupt service routine will automatically push information on the stack. Since interrupts are triggered by hardware events, exactly when they occur is not under software control. Therefore, violations of rules 3, 4, and 5 will cause erratic behavior when operating with interrupts. Rules 4 and 5 are followed automatically by the **PUSH** and **POP** instructions.

First, we will consider the situation where the allocated stack area is placed at the beginning of RAM. For example, assume we allocate 4096 bytes for the stack from 0x2000.0000 to 0x2000.0FFF, see the left side of Figure 3.28. The SP is initialized to 0x2000.1000, and the stack is considered empty. If the SP becomes less than 0x2000.0000 a **stack overflow** has occurred. The stack overflow will cause a bus fault because there is nothing at address 0x1FFF.FFFC. If the software tries to read from or write to any location greater than or equal to 0x2000.1000 then a **stack underflow** has occurred. At this point the stack and global variables may exist at overlapping addresses. Stack underflow is a very difficult bug to recognize, because the first consequence will be unexplained changes to data stored in global variables.

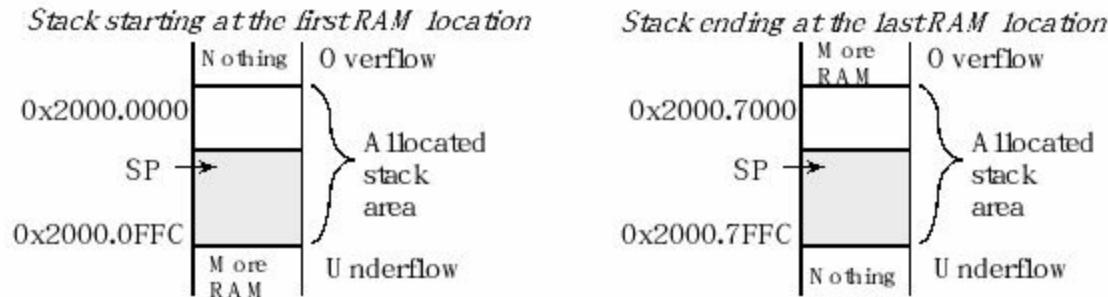


Figure 3.28. Drawings showing two possible ways to allocate the stack area in RAM.

Next, we will consider the situation where the allocated stack area is placed at the end of RAM. The TM4C123 has 32 KiB of RAM from 0x2000.0000 to 0x2000.7FFF. So in this case we allocate the 4096 bytes for the stack from 0x2000.7000 to 0x2000.7FFF, shown on the right side of Figure 3.28. The SP is initialized to 0x2000.8000, and the stack is considered empty. If the SP becomes less than 0x2000.7000 a stack overflow has occurred. The stack overflow will not cause a bus fault because there is memory at address 0x2000.6FFC. Stack overflow in this case is a very difficult bug to recognize, because the first consequence will be unexplained changes to data stored below the stack region. If the software tries to read from or write to any location greater than or equal to 0x2001.0000 then a stack underflow has occurred. In this case, stack underflow will cause a bus fault. We will also use the stack to save program state during interrupt processing.

### 3.3.8. Functions and Control Flow

Normally the computer executes one instruction after another in a linear fashion. In particular, the next instruction to execute is found immediately following the current instruction. Figure 3.9 illustrates exceptions to the one after another rule. More specifically, we use branch instructions to deviate from this straight line path. Table 3.2 lists the conditional execution available on the ARM ® Cortex™-M processor. In this section, we will use the conditional branch instruction to implement if-then, while-loop, and for-loop control structures.

<b>B{cond} label</b>	<b>;branch to label</b>
<b>BX{cond} Rm</b>	<b>;branch indirect to location specified by Rm</b>
<b>BL{cond} label</b>	<b>;branch to subroutine at label</b>
<b>BLX{cond} Rm</b>	<b>;branch to subroutine indirect specified by Rm</b>

**Subroutines**, **procedures**, and **functions** are code sequences that can be called to perform specific tasks. They are important conceptual tools because they allow us to develop modular software. The programming languages Pascal, FORTRAN, and Ada distinguish between functions, which return values, and procedures, which do not. On the other hand, the programming languages C, C++, Java, and Lisp do not make this distinction and treat functions and procedures as synonymous. Object-oriented programming languages use the term **method** to describe subprograms that are part of objects; it is also used in conjunction with type classes. In assembly language, we use the term subroutine for all subprograms whether or not they return a value. Modular programming allows us to build complex systems using simple components. In this section we present a short introduction on the syntax for defining subroutines. We define a subroutine by giving it a name in the label field, followed by instructions, which when executed, perform the desired effect. The last instruction in a subroutine will be **BX LR**, which we use to return from the subroutine. In Program 3.9, we define the subroutine named **Change**, which adds 25 to the variable **Num**. The flowchart for this example is drawn in Figure 3.29. In assembly language, we will use the **BL** instruction to call this subroutine. At run time, the **BL** instruction will save the return address in the **LR** register. The return address is the location of the instruction immediately after the **BL** instruction. At the end of the subroutine, the **BX LR** instruction will retrieve the return address from the **LR** register, returning the program to the place from which the subroutine was called. More precisely, it returns to the instruction immediately after the instruction that performed the subroutine call. The comments specify the order of execution. The while-loop causes instructions 4–10 to be repeated over and over.

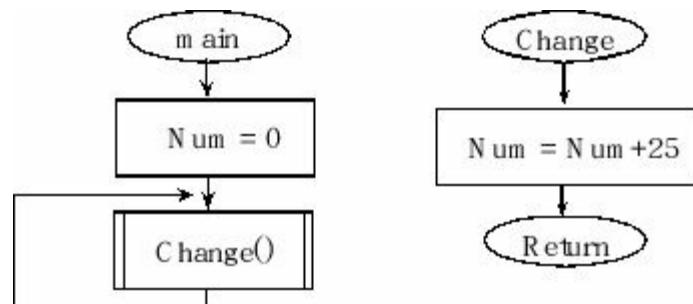


Figure 3.29. A flowchart of a simple function that generates a pseudo random number.

In C, input parameters, if any, are passed in R0–R3. If there are more than 4 input parameters, they are pushed on the stack. The output parameter, if needed, is returned in R0.

<b>Change</b> LDR R1,=Num ; 5) R1 = &Num <b>LDR</b> R0,[R1] ; 6) R0 = Num <b>ADD</b> R0,R0,#25 ; 7) R0 = Num+25 <b>STR</b> R0,[R1] ; 8) Num = Num+25 <b>BX LR</b> ; 9) return <b>main</b> LDR R1,=Num ; 1) R1 = &Num <b>MOV</b> R0,#0 ; 2) R0 = 0 <b>STR</b> R0,[R1] ; 3) Num = 0 <b>loop</b> BL Change ; 4) function call
--

<b>uint32_t</b> Num; <b>void</b> Change( <b>void</b> ) { Num = Num+25; } <b>void</b> main( <b>void</b> ) { Num = 0; <b>while</b> (1){ Change(); } }
--

## B loop ; 10) repeat

Program 3.9. Assembly and C versions that define a simple function. The 1-2...-10 show the execution sequence.

Recall that all object code is halfword aligned, meaning bit 0 of the PC is always clear. When the **BL** instruction is executed, bits 31–1 of register LR are loaded with the address of the instruction after the **BL**, and bit 0 is set to one. When the **BX LR** instruction is executed, bits 31–1 of register LR are put back into the PC, and bit 0 of LR goes into the T bit. On the ARM® Cortex™-M processor, the T bit should always be 1, meaning the processor is always in the Thumb state. Normally, the proper value of bit 0 of the LR is assigned automatically by the **BL** instruction.

Decision making is an important aspect of software programming. Two values are compared and certain blocks of program are executed or skipped depending on the results of the comparison. In assembly language it is important to know the precision (e.g., 16-bit, 32-bit) and the format of the two values (e.g., unsigned, signed). It takes three steps to perform a comparison. We begin by reading the first value into a register. The second step is to compare the first value with the second value. We can use either a subtract instruction (**SUBS**) or a compare instruction (**CMP**). These instructions set the condition code bits. The last step is a conditional branch. The available conditions are listed in Table 3.2. The branch will occur if the condition is true.

Program 3.10 illustrates an if-then structure involving testing for unsigned greater than or equal to. It will increment **Num** if it is less than 25600. Since the variable is unsigned, we use an unsigned conditional. Furthermore, we want to execute the increment if **Num** is less than 25600, so we perform the opposite conditional branch (greater than or equal to) to skip over.

<b>Change LDR R1,=Num ; R1 = &amp;Num</b>	<b>uint32_t Num;</b>
<b>LDR R0,[R1] ; R0 = Num</b>	<b>void Change(void){</b>
<b>CMP R0,#25600</b>	<b>if(Num &lt; 25600){</b>
<b>BHS skip</b>	<b>    Num = Num+1;</b>
<b>ADD R0,R0,#1 ; R0 = Num+1</b>	<b>}</b>
<b>STR R0,[R1] ; Num = Num+1</b>	<b>}</b>
<b>skip BX LR ; return</b>	

Program 3.10. Software showing an if-then control structure (BHS used because it is unsigned).

Program 3.11 illustrates an if-then-else structure involving signed numbers. It will increment **Num** if it is less than 100, otherwise it will set it to -100. Since the variable is signed, we use asigned conditional. Again, we want to execute the increment if **Num** is less than 100, so we perform the opposite conditional branch (greater than or equal to) to skip over.

<b>Change LDR R1,=Num ; R1 = &amp;Num</b>	<b>int32_t Num;</b>
<b>LDR R0,[R1] ; R0 = Num</b>	<b>void Change(void){</b>
<b>CMP R0,#100</b>	<b>if(Num &lt; 100){</b>
<b>BGE else</b>	<b>    Num = Num+1;</b>
<b>ADD R0,R0,#1 ; R0 = Num+1</b>	<b>}</b>
<b>B skip</b>	<b>else{</b>

```

else MOV R0,#-100 ; -100           Num = -100;
skip STR R0,[R1] ; update Num      }
BX LR    ; return                 }

```

Program 3.11. Software showing an if-then-else control structure (BGE used because it is signed).

### 3.3.9. Assembler Directives

We use assembler directives to assist and control the assembly process. Directives or pseudo-ops are not part of the instruction set. These directives change the way the code is assembled. The first batch defines where and how the objects (code and variables) are placed in memory. **CODE** is the place for machine instructions, typically ROM. **DATA** is the place for global variables, typically RAM. **STACK** is the place for the stack, also in RAM. The **ALIGN=n** modifier starts the area aligned to  $2^n$  bytes. **.text** is used for code sections produced by the C compiler, or for code sections otherwise associated with the C library. Using **.text** makes this assembly code callable from C. Normally RAM areas are initialized to zero, but **NOINIT** defines a RAM area that is not initialized. The fact that ROM begins at 0x0000.0000 and RAM begins at 0x2000.0000 is specified in the **Project->Options**, Linker tab.

<b>AREA RESET,CODE,READONLY</b>	<b>;reset vectors in flash ROM</b>
<b>AREA DATA</b>	<b>;places objects in data memory (RAM)</b>
<b>AREA .text,CODE,READONLY,ALIGN=2</b>	<b>;code in flash ROM</b>
<b>AREA STACK,NOINIT,READWRITE,ALIGN=3</b>	<b>;stack area</b>

The next two directives are used to link between files. Normally labels in one file are not accessible in another file. If we have a global object (function or variable) we add an **EXPORT** directive in the file where the object is defined and an **IMPORT** directive in the file wishing to access the object. We can EXPORT a function in an assembly file, and call that function from a C file. Similarly, we can define a function in a C file, and IMPORT the function into an assembly file. **GLOBAL** is a synonym for **EXPORT**.

<b>IMPORT name</b>	<b>;imports function “name” from other file</b>
<b>EXPORT name</b>	<b>;exports public function “name” for use elsewhere</b>

The **ALIGN** directive is used to ensure the next object is aligned properly. For example, machine instructions must be half-word aligned, 32-bit data accessed with **LDR STR** must be word-aligned. Nice programmers place an **ALIGN** at the end of each file so the start of every file is automatically aligned.

<b>ALIGN</b>	<b>;skips 0 to 3 bytes to make next word aligned</b>
<b>ALIGN 2</b>	<b>;skips 0 or 1 byte to make next halfword aligned</b>
<b>ALIGN 4</b>	<b>;skips 0 to 3 bytes to make next word aligned</b>

The **THUMB** directive is placed at the top of the file to specify code is generated with Thumb instructions. We place an **END** directive at the end of each file.

**THUMB**  
**END**

The following directives can add variables and constants.

<b>DCB expr{,expr}</b>	<b>;places 8-bit byte(s) into memory</b>
<b>DCW expr{,expr}</b>	<b>;places 16-bit halfword(s) into memory</b>
<b>DCD expr{,expr}</b>	<b>;places 32-bit word(s) into memory</b>
<b>SPACE size</b>	<b>;reserves size bytes, uninitialized</b>

The **EQU** directive gives a symbolic name to a numeric constant, a register-relative value or a program-relative value. We will use **EQU** to define I/O port addresses. For example,

```
GPIO_PORTD_DATA_R EQU 0x400073FC
GPIO_PORTD_DIR_R EQU 0x40007400
GPIO_PORTD_DEN_R EQU 0x4000751C
```

### 3.3.10. First Example Project

We begin the example by writing a **Requirements document**.

#### 1. Overview

1.1. Objectives: Why are we doing this project? What is the purpose?

The objectives of this project are to design, build and test a random number generator.

1.2. Process: How will the project be developed?

The project will be developed using a Tiva board, and written in Keil uVision.

1.3. Roles and Responsibilities: Who will do what? Who are the clients?

Valvano will write the code, and the readers of this book are the clients.

1.4. Interactions with Existing Systems: How will it fit in?

The random number generator will be callable from other existing Cortex M systems.

1.5. Terminology: Define terms used in the document.

Pseudo random number, linear congruential generator, seed

1.6. Security: How will intellectual property be managed?

The solution will be posted on the internet, and free for others to use.

#### 2. Function Description

2.1. Functionality: What will the system do precisely?

The system will have two operations. 1) The generator can be initialized by setting the seed to any 32-bit number. 2) When the user calls **Random** a 32-bit random number will be returned.

2.2. Scope: List the phases and what will be delivered in each phase.

Phase 1 is the design; phase 2 is the implementation; and phase 3 is the testing.

### 2.3. Prototypes: How will intermediate progress be demonstrated?

A prototype system can be run on the simulator or on a real LaunchPad board.

### 2.4. Performance: Define the measures and describe how they will be determined.

It generates a pseudo random sequence of numbers. This means the generator will return a unique 32-bit number for the first  $2^{32}$  calls. After that it will loop through that sequence over and over. Let N be any constant less than 256, and we create  $n = (\text{random}() \gg 24) \% N$ , which will be a random number from 0 to N-1, like Program 2.8. The random number generator is fair: for every calculation of n, the chances it is 0 to N-1 are equally likely. The random number generator is random: if I know prior determinations of n, the chances of the next generator of n is still equally likely to be 0 to N-1.

### 2.5. Usability: Describe the interfaces. Be quantitative if possible.

To initialize the system, we write a number into the seed variable, **M**. To generate a new random number we call **Random**, and the number is returned in R0.

**Solution:** Every ARM Keil™uVision® project will have a file named **Startup.s**. CCStudio projects will have a similar file. There will be small differences between the file we use for assembly and the one we use for C, but basically this file will contain the following components:

Defines the size of the stack

Defines the size of the heap (which we will not use)

Defines the reset vector and all the interrupt vectors

The reset handler that jumps to your code

Default interrupt service routines that do nothing

Defines some functions for enabling and disabling interrupts

The reset handler is code that is run when the reset button is pushed or on power up. The reset handler initializes global variables by branching to the **\_main** function, which in turn, calls your **main()** function. If you are writing C code, you will not need to edit the **Startup.s** file. For assembly, you will have to replace the branch to **\_main** with a branch to your application code. The assembly examples with this book all use **Start** as the user code to be run on reset. This means if you name your program **Start**, you can use one of the **Startup.s** files from the book examples without editing it.

Program 3.12 implements a pseudo random number generator using a linear congruential generator (LCG).  $M_0$  is the seed which is first number in the sequence, 1. a is the multiplier, 1664525. c is the increment, 1013904223 and  $m = 2^{32}$ . The pseudo random sequence is generated using this formula, which calculates the next number from the previous one:

$$M_{n+1} = (a * M_n + c) \bmod m$$

The **AREA DATA** directive specifies the following lines are placed in data space (typically RAM). The **MSPACE 4** allocates four uninitialized bytes, which is one 32-bit word. The **AREA CODE** directive specifies the following lines are placed in code space (typically ROM). The **|.text|** connects this program to the C code generated by the compiler, which we need if linking assembly code to C code. **ALIGN=2** will force the machine code to be word-aligned as required. In assembly, the registers are used for storing temporary information. The **LDR R4,=M** instruction establishes R4 as a pointer to the variable **M**. When the function is called the return address is saved in LR. Register R0 has the return parameter, which is a new random number. The **BX LR** is the return from subroutine instruction (moves LR back to PC) making the PC point to the **B loop** instruction.

<pre> <b>THUMB</b> <b>AREA DATA, ALIGN=2</b> <b>EXPORT M [DATA,SIZE=4]</b> <b>M SPACE 4</b> <b>AREA</b> <b> .text ,CODE,READONLY,ALIGN=2</b> <b>EXPORT Start</b> <b>Start LDR R2,=M ; R2 = &amp;M</b> <b>MOV R0,#1 ; Initial seed</b> <b>STR R0,[R2] ; M=1</b> <b>loop BL Random</b> <b>    B loop</b> <b>; Return R0= random number generator</b> <b>; Linear congruent generator</b> <b>Random LDR R2,=M ; R2 = &amp;M</b> <b>    LDR R0,[R2] ; R0 = M</b> <b>    LDR R1,=1664525</b> <b>    MUL R0,R0,R1 ; R0 = 1664525*M</b> <b>    LDR R1,=1013904223</b> <b>    ADD R0,R0,R1 ;</b> <b>    1664525*M+1013904223</b> <b>    STR R0,[R2] ; store M</b> <b>    BX LR</b> <b>    ALIGN</b> <b>END</b> </pre>	<pre> <b>// C implementation</b> <b>uint32_t M;</b>  <b>// Random number</b> <b>generator</b> <b>// from Numerical</b> <b>Recipes</b> <b>// by Press et al.</b> <b>uint32_t Random(void){</b> <b>    M = 1664525*M</b> <b>    +1013904223;</b> <b>    return(M);</b> <b>}</b>  <b>void main(void){</b> <b>    uint32_t n;</b> <b>    M = 1; // seed</b> <b>    while(1){</b> <b>        n = Random();</b> <b>    }</b> <b>}</b> </pre>
---	--

Program 3.12. Assembly and C projects that implement a random number generator.

If we are in the main program, we can use any register we need. However, by AAPCS convention functions only freely modify registers R0–R3 and R12. If a function needs to use R4 through R11, it is best to push the current register value onto the stack, use the register, and then pop the old value off the stack before returning. In order to preserve an 8-byte stack alignment, AAPCS requires us to push and pop an even number of registers. If function calls are nested (one subroutine calls another subroutine) then the LR must be saved on the stack.

Sometimes we wish to write functions in assembly and call them from C. In Program 3.13, the **Random** function is written in assembly, and it is called from C. By the AAPCS convention, the first parameter is passed in Register R0. We export the function name in the assembly file, and create a prototype for the function in the C file. For C to access an assembly variable, we export it in the assembly file, and add an extern statement it in the C file

<pre> <b>THUMB</b> <b>AREA DATA, ALIGN=2</b> <b>M SPACE 4</b> <b>AREA</b> <b>.text CODE,READONLY,ALIGN=2</b> <b>EXPORT Random</b> <b>EXPORT M</b> ; Return R0= random number generator ; Linear congruent generator <b>Random LDR R2,=M ; R2 = &amp;M</b> <b>LDR R0,[R2] ; R0 = M</b> <b>LDR R1,=1664525</b> <b>MUL R0,R0,R1 ; R0 = 1664525*M</b> <b>LDR R1,=1013904223</b> <b>ADD R0,R0,R1 ;</b> <b>1664525*M+1013904223</b> <b>STR R0,[R2] ; store M</b> <b>BX LR</b> <b>ALIGN</b> <b>END</b> </pre>	<pre> // C calls assembly <b>uint32_t Random(void);</b> <b>extern uint32_t M;</b>  <b>void main(void){</b>     <b>uint32_t n;</b>     <b>M = 1; // seed</b>     <b>while(1){</b>         <b>n = Random();</b>     <b>}</b> <b>}</b> </pre>
--	--

Program 3.13. Assembly function and assembly variable is accessed from C.

Conversely, sometimes we wish to write functions in C and access them from assembly. Again, the first parameter is passed in Register R0. We import the function name and import the variable in the assembly file. Notice that C functions and variables have explicitly defined types, whereas the type for assembly functions and variables is specified in how it is used.

<pre> <b>THUMB</b> <b>IMPORT Random</b> <b>IMPORT M</b> <b>AREA</b> <b>.text CODE,READONLY,ALIGN=2</b> <b>EXPORT Start</b> <b>Start LDR R2,=M ; R2 = &amp;M</b> <b>MOV R0,#1 ; Initial seed</b> <b>STR R0,[R2] ; M=1</b> <b>loop BL Random</b> <b>B loop</b> </pre>	<pre> <b>uint32_t M;</b>  // Random number generator // from Numerical Recipes // by Press et al.  <b>uint32_t Random(void){</b>     <b>M = 1664525*M</b>     <b>+1013904223;</b>     <b>return(M);</b> </pre>
---	--

```
ALIGN  
END
```

```
}
```

### Program 3.14. C function and variable accessed from assembly.

The problem with LCG functions is the least significant bits go through very short cycles. For example, bit 0 of M has a cycle length of 2, repeating the pattern 0,1,... Similarly, bit i has a cycle length of only  $2^{i+1}$ . E.g., bit 1 repeats 0,0,1,1,... For this reason, we use the top 8 bits when generating a random number from 0 to N-1,  $n = (\text{random}() \gg 24) \% N$  ( $N \leq 256$ .)

## 3.4. Simplified Machine Language Execution

In this section, we present a cycle-by-cycle analysis for a simple processor. The purpose of considering a simplified version is to understand in general how a computer executes instructions without being burdened with the extreme complexities that exist in today's high-speed processors. The major differences between the real ARM® Cortex™-M processor and the simplified processor are shown in Table 3.8.

Actual ARM® Cortex™-M processor	Simplified processor
Sometimes 8-, 16-, 32-bit access	All opcode accesses are aligned 16-bit
Special case for unaligned access	All data accesses are aligned 32-bit
Instruction queue enhances speed	Simple fetch-execute sequence
Fetches op codes for later execution	Fetches op codes for immediate execution
Fetches op codes that are never executed	Fetched op codes are always executed
Five buses with simultaneous accessing	One shared bus
Harvard architecture	Von Neumann architecture

**Table 3.8. Differences between a real and simplified bus cycles.**

This simple processor has four major components, as illustrated in Figure 3.30. The **control unit** (CU) orchestrates the sequence of operations in the processor. The CU issues commands to the other three components. The **instruction register** (IR) contains the op code for the current instruction. Most op codes on the Cortex™-M processor are 32 bits wide, but some are 16 bits. On this simple processor op codes are 16 or 32 bits. The **arithmetic logic unit** (ALU) performs arithmetic operations such as addition, subtraction, multiplication and division. The ALU also performs logical operations such as and, or, and shift. The program counter (PC) points to the memory containing the instruction to execute next. The **bus interface unit** (BIU) reads data from the bus during a read cycle, and writes data onto the bus during a write cycle. The **effective address register** (EAR) contains the data address corresponding to an operand in the current instruction. This address could be a source or destination address depending on whether the operation is a read or write respectively.

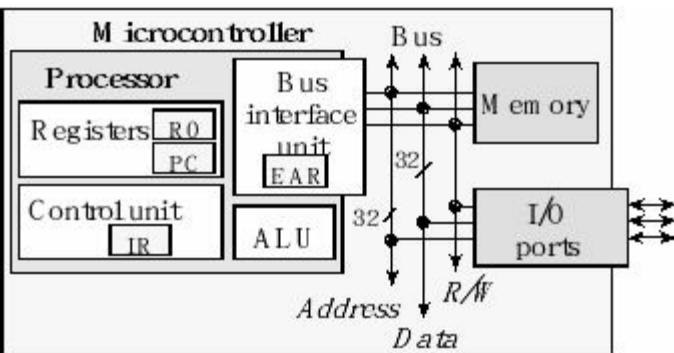


Figure 3.30. Block diagram of a simplified microcontroller.

This simplified bus contains 32 address lines, 32 data lines, and an R/W signal. There are two types of bus cycles that the processor uses to communicate with memory. For both types of cycles, the processor drives the address bus and the R/W signal, see Figure 3.30. The 32-bit address bus selects which memory location (or I/O device) to access. The R/W signal specifies read or write. During a read cycle ( $R/W=1$ ), the memory at the specified address puts the information on the data bus, and the processor transfers the information (32 bits in this simplified simulation) into the appropriate place within the processor. The processor has three types of read cycles:

**Instruction fetch.** The address is the PC and the 32-bit data is loaded into the instruction register, IR. The PC is incremented by 2 or by 4.

**Data fetch.** The address is the EAR, and the 32-bit data is loaded into a register or sent to the ALU.

**Stack pop.** First, the 32-bit data is read from memory pointed to by SP and stored in a register, then the stack pointer is incremented  $SP=SP+4$ .

During a write cycle ( $R/W=0$ ), the processor puts the 32-bit information on the data bus, and the memory transfers the information into the specified location. The write cycles can be grouped into two types:

**Data write.** The 32-bit data from a register is stored in memory at the address specified by the EAR.

**Stack push.** First, the stack pointer is decremented  $SP=SP-4$ , then the 32-bit data from a register is stored in memory at the address specified by the SP.

The terms read and write cycle are used from the perspective of the processor. During a read cycle (or memory LOAD) data flows from memory or input device into the processor. Assume Register R0 equals 0x2000.0000 and memory location 0x2000.0000 contains a 0x12345678. When the processor executes **LDR R1,[R0]**, the instruction is first fetched, then there is a memory read cycle that copies the data from memory into Register R1 (Figure 3.31).

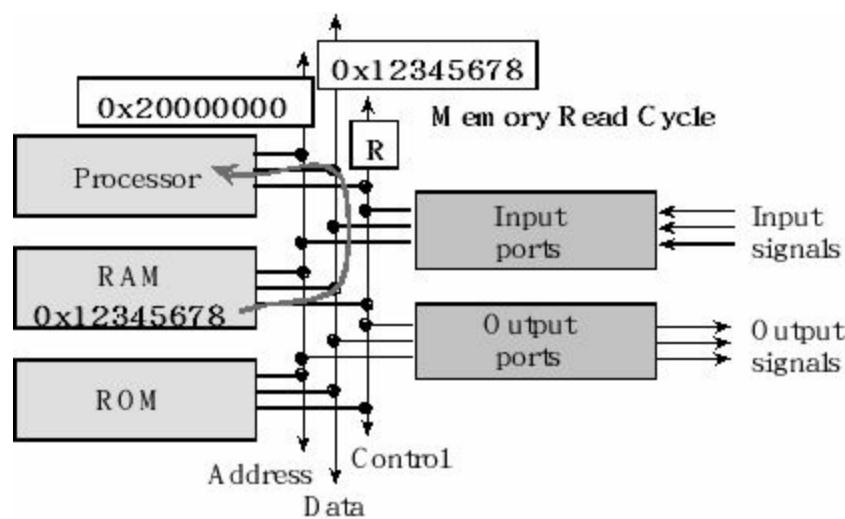


Figure 3.31. A read cycle copies data from RAM, ROM or input device into the processor.

During a write cycle (or memory STORE) data flows from the processor into memory or output device. Assume R2 is 0x2000.0004 and R3 is 0x11223344. When the processor executes **STR R3, [R2]**, the processor will first fetch the instruction, but there is a memory write cycle that copies the data from the Register R3 into memory (Figure 3.32).

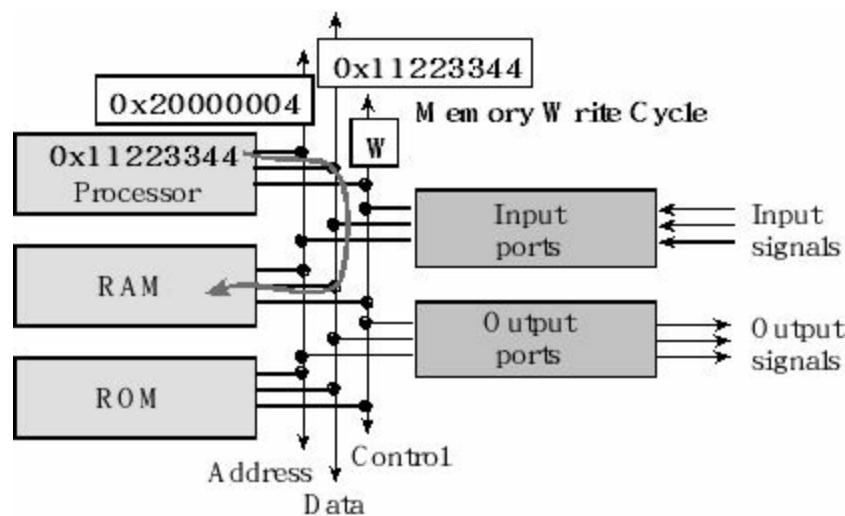


Figure 3.32. A write cycle copies data from the processor into RAM, or output device.

In general, the execution of an instruction goes through many phases. First, the computer fetches the machine code for the instruction by reading the value in memory pointed to by the program counter (PC). After each word of the instruction is fetched, the PC is incremented by 2 or 4 (depending on the size of the instruction). During phases 2 and 3, the instruction is decoded, and the effective address is determined (EAR). The load and pop instructions require additional data, and during phase 4 the data is retrieved from memory at the effective address. During phase 5, the actual function for this instruction is performed. Sometimes the computer bus is idle at this time, because no additional data is required. The store and push instructions require data to be written back to memory. A simple instruction like **ADD R0,R1,R2** takes only one cycle to complete: one cycle to fetch the op code, and no bus cycles are required to perform the addition because arithmetic and logic operations occur inside the processor. A load or store instruction like **LDR R0,[R1]** takes two cycles to complete: one cycle to fetch the op code, and a second cycle to access the memory contents. The simplified execution has six phases, but in this discussion we will focus only on those phases that generate bus cycles (shown in **bold**):

<u>Phase</u>	<u>Function</u>	<u>R/W</u>	<u>Address</u>	<u>Comment</u>
<b>1</b>	<b>Op code fetch</b>	<b>read</b>	<b>PC</b>	<b>Put op code into IR</b>
2	Decode instruction	none		Increment PC by 2 or by 4
3	Evaluation address	none		Determine EAR
<b>4</b>	<b>Data read</b>	<b>read</b>	<b>SP,EAR</b>	<b>Data passes through ALU,</b> ALU operations
5	Free cycle			
<b>6</b>	<b>Data store</b>	<b>write</b>	<b>SP,EAR</b>	<b>Results stored in memory</b>

The Cortex™-M processor does not have both a memory read and a memory write cycle in the same instruction. All instructions have a phase 1, but the other phases may or may not occur for any specific instruction. Phase 1 will fetch the entire machine code for the instruction. The op code is placed in the IR, and the operand either contains data itself or is used to determine the memory address of the data. The subsequent phases may require 0 or 1 bus cycle to complete. Each bus cycle reads or writes one piece of data. On the real Cortex™-M processor, read and write cycles can transfer 8-bit, 16-bit or 32-bit data, but for this simplified analysis all cycles are 32-bit.

Phase 1. **Opcode fetch.** The execution begins with fetching the entire op code and putting it in the IR. Additional phase 1 cycles will occur until the entire machine code (op code and operand) are fetched. The PC is incremented by 2 or 4 after fetching the op code.

Phase 2. **Decode instruction.** The op code will tell the control unit exactly what steps need to be performed to execute the instruction. This phase happens so quickly that bus cycles are not needed.

Phase 3. **Evaluate address.** During this phase, the processor will set the EAR to point to the address where memory is to be accessed. Usually, this phase does not require any bus cycles.

Phase 4. **Data read.** The load and pop instructions require data from memory. These instructions will use the EAR to read data from memory as needed. It takes a bus cycle to read data from memory, but since registers are inside the processor, no bus cycles occur as data is saved into a register. Remember registers do not have addresses, and see in Figure 3.30 that registers are not attached to the bus.

Phase 5. **Free cycles.** Any ALU functions occur next. On a real microcontroller the ALU requires time to execute, but the simplified cycle-by-cycle simulation does not account for these do-nothing cycles.

Phase 6. **Data write.** The store and push instructions require writing data to memory. The address of these writes is determined by the EAR.

The goal of this simplified execution is to visualize bus activity during execution. The right side of Program 3.15 shows the assembly source code, and the left side shows the assembly listing created by the assembler. We will begin execution at **Start**. After each opcode fetch, the PC is incremented by 2 or 4 depending on whether the instruction was 2 or 4 bytes. The BL instruction will call the subroutine by saving the return address in LR. Notice bit 0 is set; this signifies LR points to Thumb code. When the return from subroutine instruction BX LR is executed, the least significant bit of LR is moved to the T-bit (Thumb mode).

One of the interesting observations of this simple analysis is that the processor uses different buses to fetch op codes than it uses to fetch data from memory and I/O. Because they are different buses, there is the opportunity to perform these fetches in parallel. Unfortunately, sometimes parallel operation can occur and sometimes it cannot. For this reason, it is almost impossible to predetermine the time it will take to execute an instruction. In other words, counting bus cycles to determine how long it will take a program to execute will only be approximate. For more information on counting bus cycles see pages 32-39 of the technical reference manual CortexM4\_TRM\_r0p1.pdf

		AREA DATA, ALIGN=2
		Num SPACE 4
		AREA
		CODE,READONLY,ALIGN=2
		THUMB
		EXPORT Start
0x000001BA 4906	LDR r1,	Change LDR R1,=Num ; 5)
[pc,#24]		LDR R0,[R1] ; 6)
0x000001BC 6808	LDR r0,	ADD R0,R0,#25 ; 7)
[r1,#0x00]		STR R0,[R1] ; 8)
0x000001BE F1000019	ADD	BX LR ; 9)
r0,r0,#0x19		Start LDR R1,=Num ; 1)
0x000001C2 6008	STR r0,	MOV R0,#0 ; 2)
[r1,#0x00]		STR R0,[R1] ; 3)
0x000001C4 4770	BX lr	loop BL Change ; 4)
0x000001C6 4903	LDR r1,	B loop ; 10)
[pc,#12]		
0x000001C8 F04F0000	MOV	
r0,#0x00		
0x000001CC 6008	STR r0,	
[r1,#0x00]		
0x000001CE F7FFFF4	BL.W	
0x000001BA		

**0x000001D2 E7FC B**  
**0x000001CE**  
**0x000001D4 20000000**

Program 3.15. Assembly version of Program 3.9. The 1-2-...-10 show the execution sequence.

<u>R/W</u>	<u>Address</u>	<u>Data</u>	<u>Operations</u>	<u>Instruction</u>
Read	0x000001C6	0x4903	1) PC=0x000001C8	LDR R1,=Num
Read	0x000001D4	0x20000000	R1=0x20000000	
Read	0x000001C8	0xF04F0000	2) PC=0x000001CC,R0=0	MOV R0,#0
Read	0x000001CC	0x6008	3) PC=0x000001CE	STR R0,[R1]
Write	0x20000000	0x00000000	Num=0	
Read	0x000001CE	0xF7FFFFF4	4) LR=0x000001D3,PC=0x000001BA BL Change	
Read	0x000001BA	0x4906	5) PC=0x000001BC	LDR R1,=Num
Read	0x000001D4	0x20000000	R1=0x20000000	
Read	0x000001BC	0x6808	6) PC=0x000001BE	LDR R0,[R1]
Read	0x20000000	0x00000000	R0=0	
Read	0x000001BE	0xF1000019	7) PC=0x000001C2,R0=25	ADD R0,R0,#25
Read	0x000001C2	0x6008	8) PC=0x000001C4	STR R0,[R1]
Write	0x20000000	0x00000019	Num=25	
Read	0x000001C4	0x4770	9) LR=0x000001D2	BX LR
Read	0x000001D2	0xE7FC	10) PC=0x000001CE	B Loop

## 3.5. CISC versus RISC

There are two classifications of processors: complex instruction set computer (CISC) and reduced instruction set computer (RISC). Table 3.9 lists general observations when deciding whether to call a computer CISC or RISC. In reality, there are a wide range of architectures and these architectures exist in the spectrum ranging from completely CISC to completely RISC. Examples of CISC include Intel x86 and Freescale 9S12. Examples of RISC include LC3, MIPS, AVR (Atmel), PowerPC (IBM), SPARC (Sun), MSP430 (TI), and ARM. The ARM company name originally began as Acorn RISC Machine, changed to A RISC Machine, and now the ARM company name is not an acronym at all, it just is ARM.

CISC	RISC
Many instructions	Few instructions
Instructions have varying lengths	Instructions have fixed lengths
Instructions execute in varying times	Instructions execute in 1 or 2 bus cycles
Many instructions can access memory	Few instructions (e.g., load and store) can access memory
In one instruction, the processor can both read and write memory	No one instruction can both read and write memory in the same instruction
Fewer and more specialized registers. E.g., some registers contain data, others contain addresses	Many identical general purpose registers
Many different types of addressing modes	Limited number of addressing modes. E.g., Thumb has register, immediate, and indexed.

**Table 3.9. General characteristics of CISC and RISC architectures.**

In a CISC computer, the complexity is embedded in the processor. In a RISC computer, the complexity exists in the assembly code generated by the programmer or the compiler. RISC computers can be designed for low power because of the simplicity of the architecture (e.g., MSP430). Which architecture is best is beyond the scope of this book, but it is important to recognize the terminology. It is very difficult to compare the execution speed of two computers, especially between a CISC and a RISC. One way to compare is to run a benchmark program on both, and measure the time it takes to execute.

Time to execute benchmark = Instructions/program \* Average cycles/instruction \* Seconds/cycle

For example, the 80 MHz ARM Cortex M has one bus cycle every 12.5 ns. On average it may require 1.5 cycles per instruction. If the benchmark program executes 10,000,000 assembly instructions, then the time to execute the benchmark will be 0.1875 seconds.

## 3.6. Details Not Covered in this Book

A first time reader can skip this section. Actually, there are two stack pointers: the main stack pointer (MSP) and the process stack pointer (PSP). Only one stack pointer is active at a time. In a high-reliability operating system, we could activate the PSP for user software and the MSP for operating system software. This way the user program could crash without disturbing the operating system. Because of the simple and dedicated nature of the embedded systems developed in this book, we will exclusively use the main stack pointer.

The ARM ® Cortex™-M processor has two privilege levels called privileged and unprivileged. Bit 0 of the **CONTROL** register is the **thread mode privilege level** (TPL). If TPL is 1 the processor level is privileged. If the bit is 0, then processor level is unprivileged. Running at the unprivileged level prevents access to various features, including the system timer and the interrupt controller. Bit 1 of the CONTROL register is the active stack pointer selection (ASPSEL). If ASPSEL is 1, the processor uses the PSP for its stack pointer. If ASPSEL is 0, the MSP is used. When designing a high-reliability operating system, we will run the user code at an unprivileged level using the PSP and the OS code at the privileged level using the MSP.

In this book we will not consider the **Q bit**, which is the sticky saturation flag and is set by the **SSAT** and **USAT** instructions. The ICI/IT bits are used by interrupts and by the IF-THEN instructions.

The ARM ® Cortex™-M processor uses **bit-banding** to allow read/write access to individual bits in RAM and some bits in the I/O space. For more information on bit-banding, refer to Chapter 2 of Volume 2. We will see an alternative way to access GPIO pins on the Texas Instruments microcontrollers in the next chapter called **bit-specific addressing**.

If-then-else control structures are commonly found in computer software. If the **BHI** in Program 3.10 or the **BGE** in Program 3.11 were to branch, the instruction pipeline would have to be flushed and refilled. In order to optimize execution speed for short if-then and if-then-else control structures, the ARM ® Cortex™-M processor employs conditional execution. There can be between one and four conditionally executed instructions following an **IT** instruction. The syntax is

**IT{x{y{z}}}** **cond**

where **x** **y** and **z** specify the existence of the optional second, third, or fourth conditional instruction respectively. For example **IT** has one conditional instruction, **ITT** has two conditional instructions, **ITTT** has three conditional instructions, and **ITTTT** has four conditional instructions. We can specify **x** **y** and **z** as **T** for execute if true or **E** for else. The **cond** field choices are listed in Table 3.2. The conditional suffixes for the instructions must match the conditional field of the **IT** instruction. In particular, the conditional for the true instructions exactly match the conditional for the **IT** instruction. Furthermore, the else instructions must have the logical complement conditional. If the condition is true, the instruction is executed. If the condition is false, the instruction is fetched but not executed. For example, Program 3.10 could have been written as follows. The two **T**'s in **ITT** means there are two true instructions.

### **Change LDR R1,=Num ; R1 = &Num (R1 points to Num)**

```
LDR R0,[R1] ; R0 = Num  
CMP R0,#25600  
ITT LO  
ADDLO R0,R0,#1 ; if(R0<25600) R0 = Num+1  
STRLO R0,[R1] ; if(R0<25600) Num = Num+1  
BX LR ; return
```

Program 3.11 could have been written as follows. The one T and one E in **ITE** means there is one true and one else instruction.

### **Change LDR R1,=Num ; R1 = &Num (R1 points to Num)**

```
LDR R0,[R1] ; R0 = Num  
CMP R0,#100  
ITE LT  
ADDLT R0,R0,#1 ; if(R0< 100) R0 = Num+1  
MOVGE R0,#-100 ; if(R0>=100) R0 = -100  
STR R0,[R1] ; update Num  
BX LR ; return
```

The following assembly converts one hex digit (0–15) in register R0 to ASCII in register R1. The one T and one E in **ITE** means there is one true and one else instruction.

```
CMP R0,#9 ; Convert R0 (0 to 15) into ASCII  
ITE GT ; Next 2 are conditional  
ADDGT R1,R0,#55 ; Convert 0xA -> 'A'  
ADDLE R1,R0,#48 ; Convert 0x0 -> '0'
```

By themselves, the conditional branch instructions do not require a preceding **IT** instruction. However, a conditional branch can be used as the last instruction of an **IT** block. There are a lot of restrictions on IT. For more details, refer to the programming reference manual.

# 3.7. Exercises

- 3.1** What is special about Register 13? Register 14? Register 15?
- 3.2** In 20 words or less describe the differences between von Neumann and Harvard architectures.
- 3.3** What happens when you load a value into Register 15 with bit 0 set?
- 3.4** How much RAM and ROM are in TM4C123? What are the specific address ranges of these memory components?
- 3.5** How much RAM and ROM are in TM4C1294? What are the specific address ranges of these memory components?
- 3.6** What are the bits in the Program Status Register (PSR) of Cortex<sup>TM</sup>-M processor?
- 3.7** What does the effective address register contain?
- 3.8** What is the purpose of the following registers PSR SP PC IR EAR?
- 3.9** What happens if you execute these four assembly instructions?  
**PUSH {R1}**  
**PUSH {R2}**  
**POP {R1}**  
**POP {R2}**
- 3.10** Write assembly code that pushes registers R1 R3 and R5 onto the stack.
- 3.11** How do you initialize the stack?
- 3.12** How do you specify where to begin execution after a reset?
- 3.13** What does word-aligned mean?
- 3.14** When does the LR have to be pushed on the stack?
- 3.15** What is the difference between big and little endian?
- 3.16** What are the differences between the following three instructions?  
**LDR R0,[R1]**      **LDRH R0,[R1]**      **LDRB R0,[R1]**
- 3.17** What are the differences between the following pairs of instructions?  
**LDRH R0,[R1]**      and      **LDRSH R0,[R1]**  
**LDR R0,[R1]**      and      **STR R0,[R1]**
- 3.18** What are the addressing modes used in each of the following instructions?  
**LDR R0,[R1]**  
**PUSH {R0}**  
**MOV R0,#1**

**BL Function**

**LDR R0,=1234567**

**3.19** Explain how does the return from subroutine work in these two functions?

<b>Function PUSH {R4,LR}</b>	<b>Function2</b>
<b>;stuff</b>	<b>;stuff</b>
<b>POP {R4,PC}</b>	<b>BX LR</b>

**3.20** Does the associative principle hold for signed integer multiply and divide? Assume **Out1 Out2** **A B C** are all the same precision (e.g., 32 bits). In particular do these two C calculations always achieve identical outputs? If not, give an example.

**Out1 = (A\*B)/C;**

**Out2 = A\*(B/C);**

**3.21** Does the associative principle hold for signed integer addition and subtraction? Assume **Out3 Out4 A B C** are all the same precision (e.g., 32 bits). In particular do these two C calculations always achieve identical outputs? If not, give an example.

**Out3 = (A+B)-C;**

**Out4 = A+(B-C);**

**3.22** Consider the addition of two signed 32-bit numbers. If a positive number is added to a negative number, under what conditions will a signed overflow occur?

**3.23** Consider the subtraction of two signed 32-bit numbers. If a positive number is subtracted from another positive number, under what conditions will a signed overflow occur?

**3.24** Let A and B be two 8-bit inputs to an 8-bit binary adder. Fill in the table showing R=A+B and the four PSR bits after each addition. The first row illustrates the process.

A	B	R	NZVC
10	100	110	0000
0x40	0xA2		
0xC3	0x6F		
100	-100		
110	146		
50			0101

**3.25** Let A and B be two 8-bit inputs to an 8-bit binary subtractor. Fill in the table showing R=A-B and the four PSR bits after each subtraction. The first row illustrates the process. Calculate the carry bit in a similar way as the Cortex M does.

A	B	R	NZVC
100	10	90	0001
0x51	0x93		
0xDF	0x9F		
-70	-70		

**3.26** Let A be an 8-bit input. Fill in the table showing the promotion to 16-bit unsigned and 16-bit signed. Give all answers in 16-bit hexadecimal. The first row illustrates the process.

A	Unsigned 16-bit	Signed 16- bit
0x80	0x0080	0xFF80
0x51		
0xED		
0x00		
0xBF		

**3.27** Assume the PC is 0x00000134. The variable M is located in RAM at address 0x2000.0000. The first three lines of Program 3.12 are

```
Start LDR R2,=M ; R4 = &M
    MOV R0,#1 ; Initial seed
    STR R0,[R2] ; M=1
```

When assembled and linked it creates this object code

```
0x00000134 4A07 LDR r2,[pc,#28] ; @0x00000154
0x00000136 F04F0001 MOV r0,#0x01
0x0000013A 6010 STR r0,[r2,#0x00]
```

with this code at the end

```
0x00000154 0000
0x00000156 2000
```

Show the simplified bus cycles occurring when the three instructions are executed. Specify which registers get modified during each cycle, and the corresponding new values. Just show the three instructions.

**3.28** Assume the PC is 0x0000013C and the SP is 0x20000408. The fourth line of Program 3.12 is a function call

```
loop BL Random
    B loop
```

**Random** LDR R2,=M ; R2 = &M

When assembled and linked it creates this object code

```
0x0000013C F000F801 BL.W 0x00000142
0x00000140 E7FC B 0x0000013C
0x00000142 4A04 LDR r2,[pc,#16] ; @0x00000154
```

Show the simplified bus cycles occurring when the BL instruction is executed. Specify which registers get modified during each cycle, and the corresponding new values. Just show the one instruction.

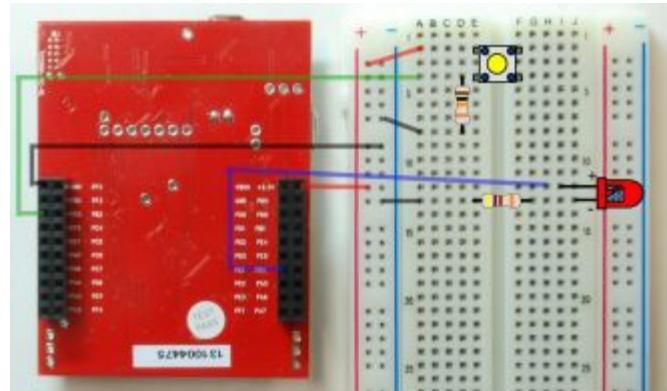
# 4. Introduction to Input/Output

## Chapter 4 objectives are to:

- Describe the parallel ports on the LM3S/TM4C family
- Present the SysTick timer
- Describe the system clocks
- Write software to input from switches and output to LEDs
- Present some practical tips for debugging

Our first input/output interfaces will use the parallel ports or GPIO, allowing us to exchange digital information with the external world. Specifically, we will learn how to connect switches and LEDs to the microcontroller. The second technique we will learn is to control time. We can select the execution speed of the microcontroller using the phase-lock-loop, and we can perform time delays using the SysTick timer.

Even though we will design systems based specifically on the LM3S/TM4C family, these solutions can, with little effort, be implemented on other versions of the Cortex™-M family. We present examples with binary inputs and binary outputs. The liquid crystal display is a low-cost, low-power, and simple interface used in many embedded applications.



From the very beginning of a project, we must consider how the system will be tested. In this chapter we present some debugging techniques that will be very useful for verifying proper operation of our system. Effective debugging tools are designed into the system becoming part of the system, rather than attached onto the system after it is built.

## 4.1. Texas Instruments Microcontroller I/O pins

Table 3.1 listed the memory configuration for some of the LM3S/TM4C microcontrollers. In this section, we present the I/O pin configurations for the LM3S1968, TM4C123 and TM4C1294 microcontrollers. The regular function of a pin is to perform parallel I/O, described later in Section 4.4. Most pins, however, have one or more alternative functions. For example, port pins PA1 and PA0 can be either regular parallel port pins or an asynchronous serial port called universal asynchronous receiver/transmitter (UART). The ability to manage time, as an input measurement and an output parameter, has made a significant impact on the market share growth of microcontrollers. Joint Test Action Group (**JTAG**), standardized as the IEEE 1149.1, is a standard test access port used to program and debug the microcontroller board. Each microcontroller uses Port C pins 3,2,1,0 for the JTAG interface.

**Common Error:** Even though it is possible to use the four JTAG pins as general I/O, debugging most microcontroller boards will be more stable if these four pins are left dedicated to the JTAG debugger.

I/O pins on Cortex-M microcontrollers have a wide range of alternative functions:

- **UART**  
**Universal asynchronous receiver/transmitter**
  - **SSI**  
**Synchronous serial interface**
  - **I<sup>2</sup>C**  
**Inter-integrated circuit**
  - **Timer compare**  
**Periodic interrupts, input capture, and output**
  - **PWM**  
**Pulse width modulation**
  - **ADC signals**  
**Analog to digital converter, measure analog**
  - **Analog Comparator**  
**Compare two analog signals**
  - **QEI**  
**Quadrature encoder interface**
  - **USB**  
**Universal serial bus**
  - **Ethernet**  
**High-speed network**
  - **CAN**  
**Controller area network**

The **UART** can be used for serial communication between computers. It is asynchronous and allows for simultaneous communication in both directions. The **SSI** is alternately called serial peripheral interface (SPI). It is used to interface medium-speed I/O devices. In this book, we will use it to interface a graphics display. In Volume 2 we use SSI to interface a digital to analog converter (DAC). In Volume 3 we use it with a secure digital card (SDC). **I<sup>2</sup>C** is a simple I/O bus that we will use to interface low speed peripheral devices. Input capture and output compare will be used to create periodic interrupts and measure period, pulse width, phase, and frequency. **PWM** outputs will be used to apply variable power to motor interfaces. In a typical motor controller, input capture measures rotational speed, and PWM controls power. A PWM output can also be used to create a DAC. The **ADC** will be used to measure the amplitude of analog signals and will be important in data acquisition systems. The analog comparator takes two analog inputs and produces a digital output depending on which analog input is greater. The **QEI** can be used to interface a brushless DC motor. **USB** is a high-speed serial communication channel. The **Ethernet** port can be used to bridge the microcontroller to the Internet or a local area network. The **CAN** creates a high-speed communication channel between microcontrollers and is commonly found in automotive and other distributed control applications. The advanced topics of USB Ethernet and CAN are covered in Volume 3.

**Observation:** The expression **mixed-signal** refers to a system with both analog and digital components. Notice how many I/O ports perform this analog↔digital bridge: ADC, DAC, analog comparator, PWM, QEI, input capture, and output compare.

## 4.1.1. Texas Instruments LM3S1968 I/O pins

Figure 4.1 draws the I/O port structure for the LM3S1968 microcontroller. Most pins have two names: the port pin (PA0) and the alternate function name (U0Rx). However, pins PF5, PF7, PG3, PG5, and PH2 have no alternate function. Because the I/O ports are connected to the system bus interface, the microcontroller can perform I/O bus cycles simultaneous with instruction fetches from flash ROM. The LM3S1968 has 3 UART ports, 2 SSI ports, 2 I<sup>2</sup>C ports, a 10-bit ADC, 6 PWM outputs, 4 timer input capture/output compare pins, 2 quadrature encoder interfaces, and three analog comparators. The ADC can sample up to 1 million times per second. There are 52 digital I/O lines and 8 ADC inputs. Table 4.1 lists the regular and alternate names of the port pins.

Regular	Alternate Pin Name	Alternate Function
PA0 – PA1	U0RX, U0TX	Universal Asynchronous Receiver/Transmit, UART0
PA2 – PA5	S0CLK, S0FS, S0RX, S0TX	Synchronous Serial Interface, SSI0
PA6 – PA7	SCL1, SDA1	Inter-Integrated Circuit, I <sup>2</sup> C1
PB0	CCP0	Timer 0A Capture/Compare
PB1	CCP2	Timer 1A Capture/Compare
PB2 – PB3	SCL0, SDA0	Inter-Integrated Circuit, I <sup>2</sup> C0
PB4, PB6, PF4	C0-, C0+, C0o	Analog Comparator 0

PB5, PC5	C1-, C1+	Analog Comparator 1
PB7, PC0 – PC3	TRST, TCLK, TMS, TDI, TDO	JTAG Debugger
PC4, PF0, PD0	PHA0, PHB0, IDX0	Quadrature Encoder Interface, QEI0
PC6, PC7	C2+, C2-	Analog Comparator 2
PD2 – PD3	U1RX, U1TX	Universal Asynchronous Receiver/Transmit, UART1
PE0 – PE3	S1CLK, S1FS, S1RX, S1TX	Synchronous Serial Interface, SSI1
PF2, PF3	PWM4, PWM5	Pulse Width Modulator 2
PF6	CCP1	Timer 0B Capture/Compare
PG0 – PG1	U2RX, U2TX	Universal Asynchronous Receiver/Transmit, UART2
PG2, PD1	PWM0, PWM1	Pulse Width Modulator 0
PG4	CCP3	Timer 1B Capture/Compare
PG6, PG7, PF1	PHA1, PHB1, IDX1	Quadrature Encoder Interface, QEI1
PH0, PH1	PWM2, PWM3	Pulse Width Modulator 1
PH3	Fault	Hold all PWM outputs in safe state

**Table 4.1. LM3S1968 I/O pins that have alternate functions.**

Figure 4.2 shows a Texas Instruments evaluation kit for the LM3S1968. There are five switches and one LED on the board, see Figure 4.3. The part numbers for these kits are EKK-LM3S1968, EKI-LM3S1968, EKC-LM3S1968, EKT-LM3S1968, and EKS-LM3S1968. The different versions specify which compiler is included on the CD in the kit.

**Observation:** To use the switches on the LM3S1968 board you need to activate the internal pull-up resistors for the port, set bits 3 – 7 in `GPIO_PORTG_PUR_R`.

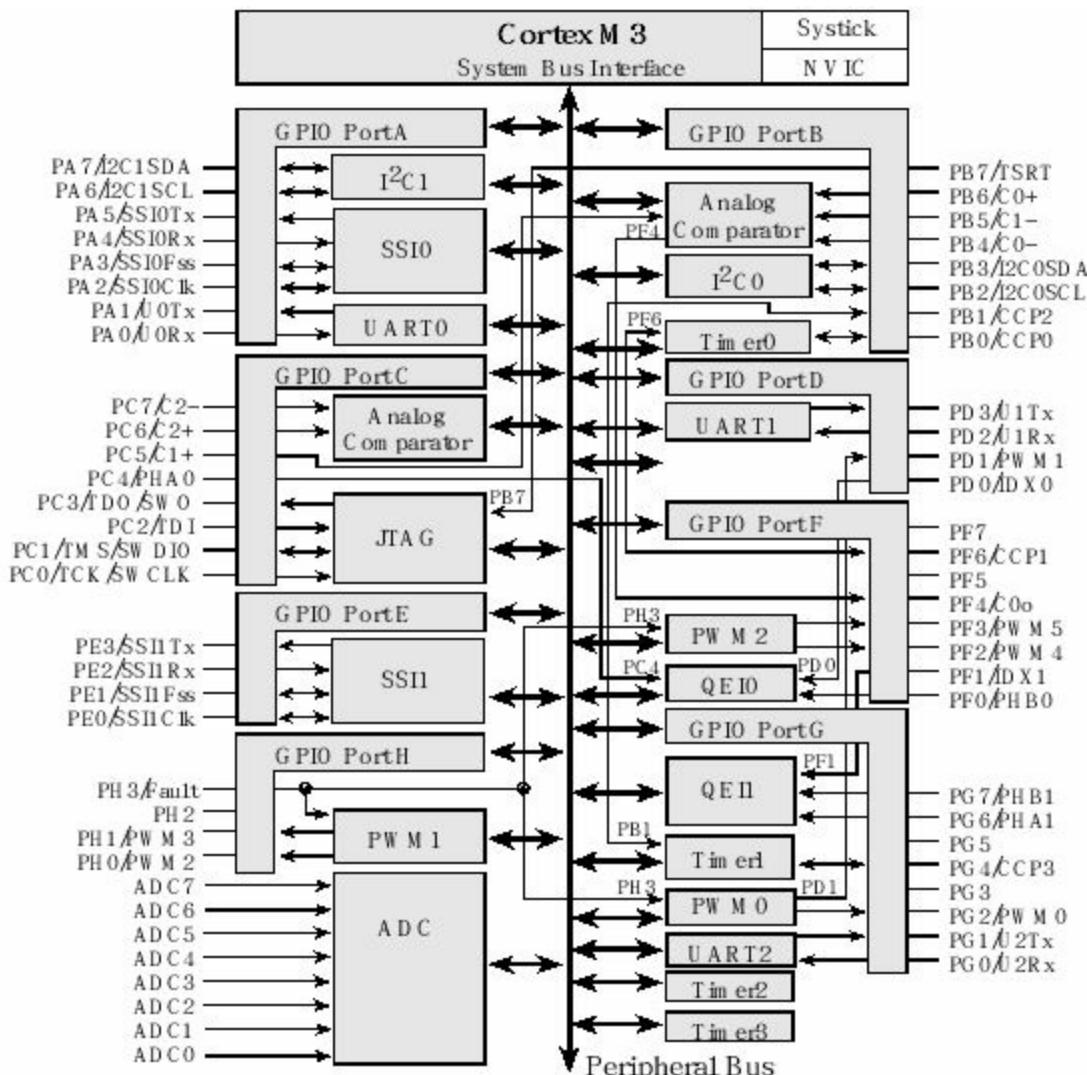


Figure 4.1. I/O port pins for the LM3S1968 microcontroller.

**Observation:** The switches on the LM3S1968 board are negative logic.

There are a number of possibilities for designing prototype systems using evaluation kits. One option is to solder individual wires to pins as needed. This approach is simple and reliable. It is appropriate if the kit is being used for one application and the choice of pins is unlikely to change. The disadvantage is changing pins requires unsoldering and resoldering.

A second approach is to solder a female socket onto the evaluation kit. To connect a pin to your external circuit, you place a solid wire into the socket. This method is convenient if you plan to move wires as the design changes. After a long period, the female socket can wear out or the ends of wires may break off inside the socket. Changing the socket is very difficult.

A third approach is illustrated in Figure 4.2. The breadboard interface was built using Samtec TSW-133-09-L-S-RE and TSW-133-08-L-S-RA connectors. Right-angle male-male headers are soldered to the board in such a way that the male pins can be inserted into a standard solderless breadboard. This approach is convenient if you are prototyping on a solderless breadboard. This configuration is extremely robust and can withstand multiple insertions and extractions. Push straight down to insert the board into the breadboard. To remove the board, use two small screwdrivers and wedge between the board and the breadboard on each side a little at a time. To assemble this interface, it may be helpful to separately insert each unsoldered header into the breadboard to hold it in place while it is being soldered. If the spacing between the headers and the development board is not correct, then it

will not fit into the breadboard. Notice how the development board fits into the slit in the middle of the breadboard. See details at

<http://users.ece.utexas.edu/~valvano/EE345L/Labs/Fall2011/LM3S1968soldering.pdf>

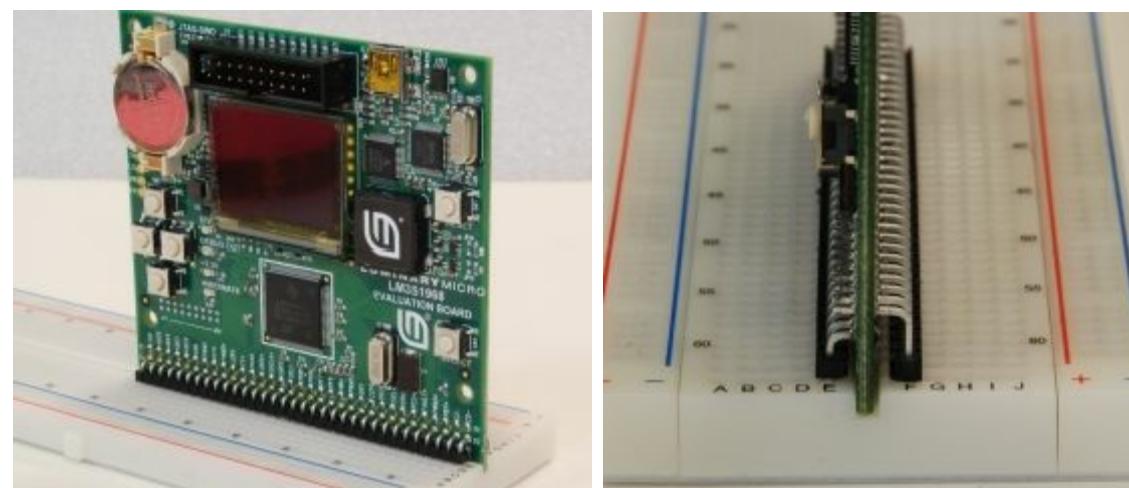


Figure 4.2. Evaluation kit for the LM3S1968 microcontroller.

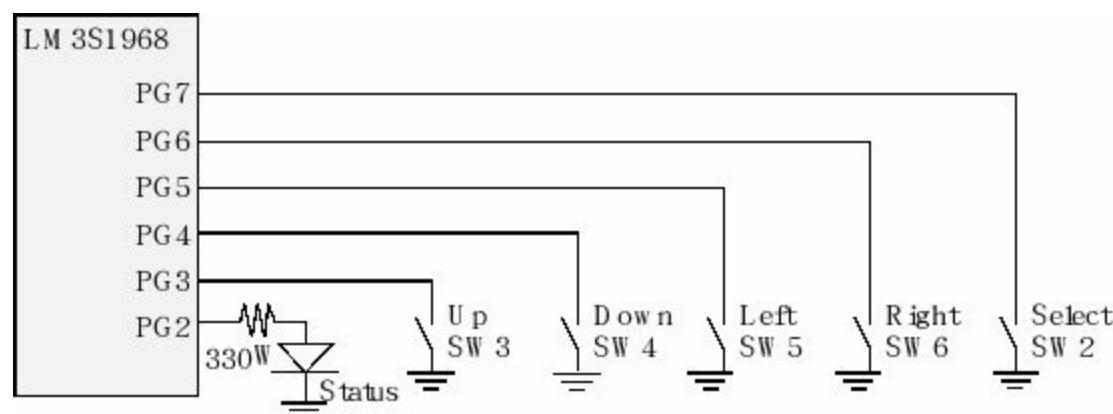


Figure 4.3. Switch and LED interfaces on the LM3S1968 evaluation board.

A fourth approach is to solder male headers onto the evaluation kit. To connect a pin to your external circuit, you use female to male single-wire jumper cables. This method is also convenient if you plan to move wires as the design changes. After a long period, the female end of the cable will wear out and need to be replaced. One option for building this female to male jumper cable is to crimp a 24-gauge solid wire into a Molex 16-02-0103 female socket.

Table 4.2 lists the physical devices attached to pins on the kit. These connections can be broken by removing a jumper on the board. By removing the jumper the pin is available for your circuits. You must enable internal pull-ups to use the switches on the board.

Pin	Function	To Isolate, Remove...
PG3	SW3 Up Momentary Negative Logic Push Button	JP1
PG2/PWM0	User LED	JP2
PH3/Fault	OLED Display Power Enable	JP3
PA0/U0RX	Virtual COM Port Receive	JP4
PA3/S0FS	OLED Display Chip Select	JP5

PG6/PHA1	SW6 Right Momentary Negative Logic Push Button	JP6
PG7/PHB1	Select Momentary Negative Logic Push Button	JP7
PG5	SW5 Left Momentary Negative Logic Push Button	JP8
PG4	SW4 Down Momentary Negative Logic Push Button	JP9
PA5/S0TX	OLED Display Data In	JP10
PA2/S0CLK	OLED Display Clock	JP11
PH2	OLED Display Data/Control Select	JP12
PA1/U0TX	Virtual COM Port Transmit	JP13
PH0/PWM2	Sound +	JP14
PH1/PWM3	Sound -	JP15
PC0/TCK/SWCLK	JTAG Debugger Clock	Do Not Use
PC1/TMS/SWDIO	JTAG Debugger Mode Select	Do Not Use
PC2/TDI	JTAG Debugger Data In	Do Not Use
PC3/TDO/SWO	JTAG Debugger Data Out	Do Not Use
PB7/TRST	JTAG Debugger Test Reset	Do Not Use

**Table 4.2. Port pins connected to physical devices on the LM3S1968 evaluation kit.**

## 4.1.2. Texas Instruments TM4C123 LaunchPad I/O pins

Figure 4.4 draws the I/O port structure for the LM4F120H5QR and TM4C123GH6PM. These microcontrollers are used on the EK-LM4F120XL and EK-TM4C123GXL LaunchPads. Pins on the LM3S family have two possibilities: digital I/O or an alternative function. However, pins on the TM4C family can be assigned to as many as eight different I/O functions. Pins can be configured for digital I/O, analog input, timer I/O, or serial I/O. For example PA0 can be digital I/O, serial input, or CAN. There are two buses used for I/O. The digital I/O ports are connected to both the advanced peripheral bus (like the LM3S family) and the advanced high-performance bus (runs faster). Because of the multiple buses, the microcontroller can perform I/O bus cycles simultaneous with instruction fetches from flash ROM. The LM4F120H5QR has eight UART ports, four SSI ports, four I2C ports, two 12-bit ADCs, twelve timers, a CAN port, and a USB interface. The TM4C123GH6PM adds up to 16 PWM outputs. There are 43 I/O lines. There are twelve ADC inputs; each ADC can convert up to 1 million samples per second. Table 4.3 lists the regular and alternate names of the port pins.

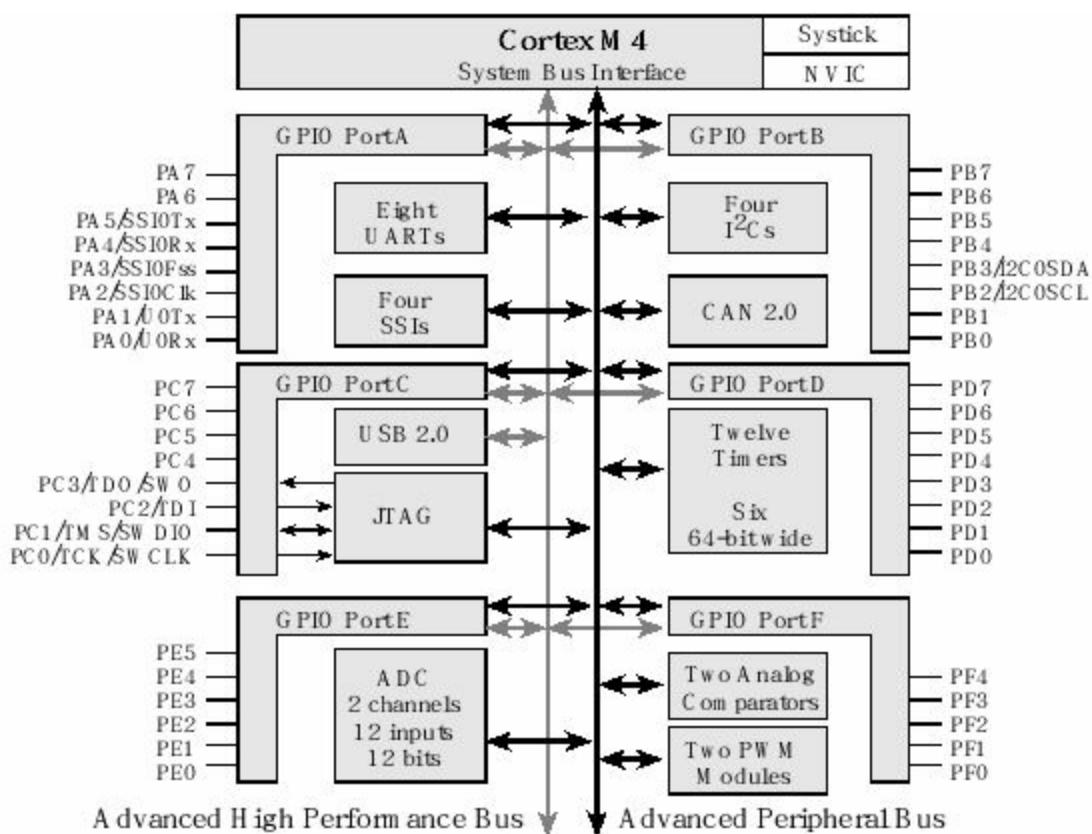


Figure 4.4. I/O port pins for the LM4F120H5QR / TM4C123GH6PM microcontrollers.

Each pin has one configuration bit in the **AMSEL** register. We set this bit to connect the port pin to the ADC or analog comparator. For digital functions, each pin also has four bits in the **PCTL** register, which we set to specify the alternative function for that pin (0 means regular I/O port). Table 4.3 shows the 4-bit **PCTL** configuration used to connect each pin to its alternate function. For example, column “5” means set 4-bit field in **PCTL** to 0101.

Pins PC3 – PC0 were left off Table 4.3 because these four pins are reserved for the JTAG debugger and should not be used for regular I/O. Notice, some alternate function modules (e.g., U0Rx) only exist on one pin (PA0). While other functions could be mapped to two or three pins (e.g., CAN0Rx could be mapped to one of the following: PB4, PE4, or PF0.)

For example, if we wished to use UART7 on pins PE0 and PE1, we would set bits 1,0 in the **DEN** register (enable digital), clear bits 1,0 in the **AMSEL** register (disable analog), write a 0001,0001 to bits 7–0 in the **PCTL** register (enable UART7 functionality), and set bits 1,0 in the **AFSEL** register (enable alternate function). If we wished to sample an analog signal on PD0, we would set bit 0 in the alternate function select register **AFSEL**, clear bit 0 in the digital enable register **DEN** (disable digital), set bit 0 in the analog mode select register **AMSEL** (enable analog), and activate one of the ADCs to sample channel 7. Additional examples will be presented throughout the book.

PA6	Port	I <sub>2</sub> C1SCL	M1PWM2			
PA7	Port	I <sub>2</sub> C1SDA	M1PWM3			
PB0	Port U1Rx			T2CCP0		
PB1	Port U1Tx			T2CCP1		
PB2	Port	I <sub>2</sub> C0SCL		T3CCP0		
PB3	Port	I <sub>2</sub> C0SDA		T3CCP1		
PB4	Ain10	Port SSI2Clk	M0PWM2	T1CCP0	CAN0Rx	
PB5	Ain11	Port SSI2Fss	M0PWM3	T1CCP1	CAN0Tx	
PB6	Port SSI2Rx	M0PWM0		T0CCP0		
PB7	Port SSI2Tx	M0PWM1		T0CCP1		
PC4	C1-	Port U4Rx	U1Rx	M0PWM6	IDX1	WT0CCP0 U1RTS
PC5	C1+	Port U4Tx	U1Tx	M0PWM7	PhA1	WT0CCP1 U1CTS
PC6	C0+	Port U3Rx			PhB1	WT1CCP0 USB0epen
PC7	C0-	Port U3Tx				WT1CCP1 USB0pflt
PD0	Ain7	Port SSI3Clk SSI1Clk	I <sub>2</sub> C3SCL	M0PWM6 M1PWM0		WT2CCP0
PD1	Ain6	Port SSI3Fss SSI1Fss	I <sub>2</sub> C3SDA	M0PWM7 M1PWM1		WT2CCP1
PD2	Ain5	Port SSI3Rx SSI1Rx		M0Fault0		WT3CCP0 USB0epen
PD3	Ain4	Port SSI3Tx SSI1Tx			IDX0	WT3CCP1 USB0pflt
PD4	USB0DM	Port U6Rx				WT4CCP0
PD5	USB0DP	Port U6Tx				WT4CCP1
PD6		Port U2Rx		M0Fault0	PhA0	WT5CCP0
PD7		Port U2Tx			PhB0	WT5CCP1 NMI
PE0	Ain3	Port U7Rx				
PE1	Ain2	Port U7Tx				
PE2	Ain1	Port				
PE3	Ain0	Port				
PE4	Ain9	Port U5Rx	I <sub>2</sub> C2SCL	M0PWM4 M1PWM2		CAN0Rx
PE5	Ain8	Port U5Tx	I <sub>2</sub> C2SDA	M0PWM5 M1PWM3		CAN0Tx
PF0		Port U1RTS	SSI1Rx	CAN0Rx	M1PWM4	PhA0 T0CCP0
PF1		Port U1CTS	SSI1Tx		M1PWM5	PhB0 T0CCP1
PF2		Port	SSI1Clk	M0Fault0	M1PWM6	T1CCP0
PF3		Port	SSI1Fss	CAN0Tx	M1PWM7	T1CCP1
PF4		Port			M1Fault0	IDX0 T2CCP0 USB0epen

**Table 4.3. PMCx bits in the GPIOPTCL register on the LM4F/TM4C specify alternate functions. PD4 and PD5 are hardwired to the USB device. PA0 and PA1 are hardwired to the serial port. PWM not on LM4F120.**

The Tiva ® LaunchPad evaluation board (Figure 4.5) is a low-cost development board available as part number EK-LM4F120XL and EK-TM4C123GXL from [www.ti.com](http://www.ti.com) and from regular electronic distributors like Digikey, Mouser, Newark, and Avnet. The kit provides an integrated In-Circuit Debug Interface (ICDI), which allows programming and debugging of the onboard LM4F microcontroller. One USB cable is used by the debugger (ICDI), and the other USB allows the user to develop USB applications (device or host). The user can select board power to come from either the debugger (ICDI) or the USB device (device) by setting the Power selection switch.

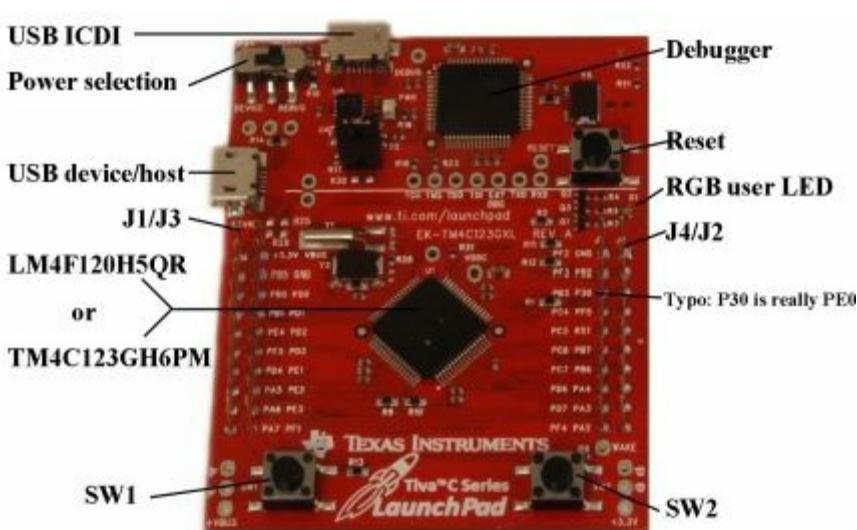


Figure 4.5. Tiva® LaunchPad based on the LM4F120H5QR or TM4C123GH6PM.

Pins PA1 – PA0 create a serial port, which is linked through the debugger cable to the PC. The serial link is a physical UART as seen by the LM4F/TM4C and mapped to a virtual COM port on the PC. The USB device interface uses PD4 and PD5. The JTAG debugger requires pins PC3 – PC0. The LaunchPad connects PB6 to PD0, and PB7 to PD1. If you wish to use both PB6 and PD0 you will need to remove the R9 resistor. Similarly, to use both PB7 and PD1 remove the R10 resistor.

The Tiva® LaunchPad evaluation board has two switches and one 3-color LED. See Figure 4.6. The switches are negative logic and will require activation of the internal pull-up resistors. In particular, you will set bits 0 and 4 in **GPIO\_PORTF\_PUR\_R** register. The LED interfaces on PF3 – PF1 are positive logic. To use the LED, make the PF3 – PF1 pins an output. To activate the red color, output a one to PF1. The blue color is on PF2, and the green color is controlled by PF3. The 0- $\Omega$  resistors (R1, R2, R11, R12, R13, R25, and R29) can be removed to disconnect the corresponding pin from the external hardware.

The LaunchPad has four 10-pin connectors, labeled as J1 J2 J3 J4 in Figures 4.5 and 4.7, to which you can attach your external signals. The top side of these connectors has male pins, and the bottom side has female sockets. The intent is to stack boards together to make a layered system, see Figure 4.7. Texas Instruments also supplies Booster Packs, which are pre-made external devices that will plug into this 40-pin connector. The Booster Packs for the MSP430 LaunchPad are compatible (one simply plugs these 20-pin connectors into the outer two rows) with this board. The inner 10-pin headers (connectors J3 and J4) are not intended to be compatible with other TI LaunchPads. J3 and J4 apply only to Stellaris/Tiva Booster Packs.

There are two methods to connect external circuits to the LaunchPad. One method uses male to female jumper cable (e.g., item number 826 at [www.adafruit.com](http://www.adafruit.com)) or solder a solid wire into a female socket (e.g., Hirose DF11-2428SCA) creating a male to female jumper wire. The second method uses male-male wires and connect to the bottom of the LaunchPad.

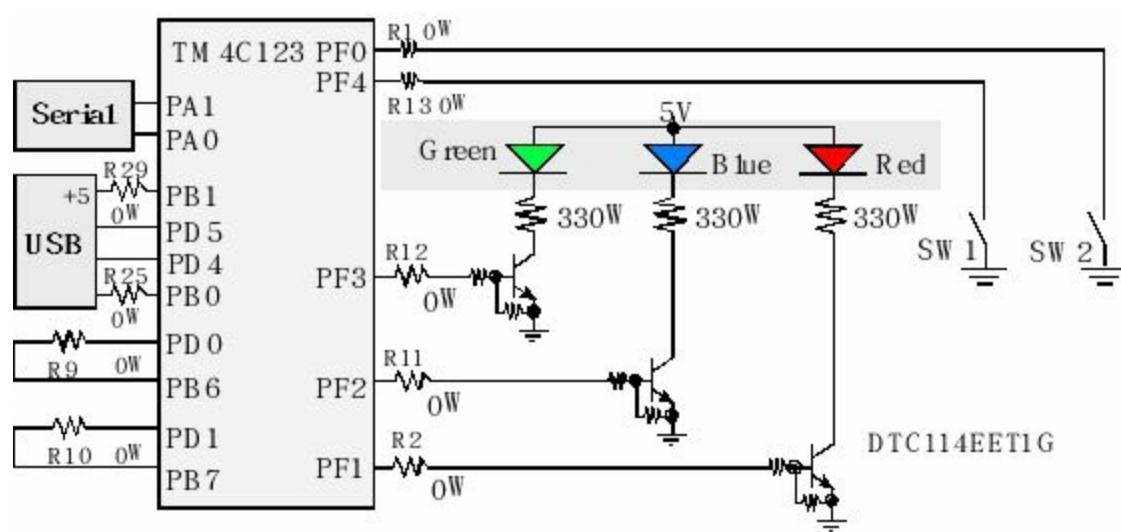


Figure 4.6. Switch and LED interfaces on the LaunchPad Evaluation Board. The zero ohm resistors can be removed so the corresponding pin can be used for its regular purpose.

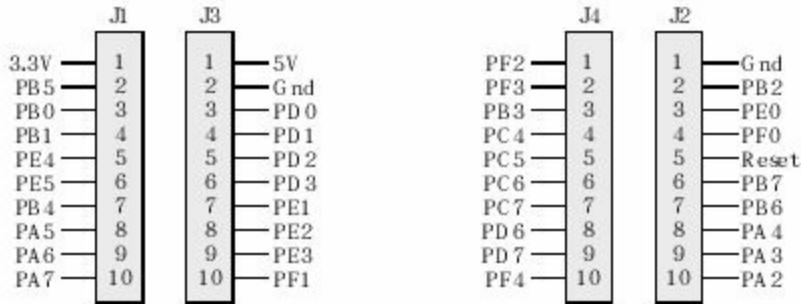


Figure 4.7. Interface connectors on the Tiva® LM4F120/TM4C123 LaunchPad Evaluation Board.

### 4.1.3. Texas Instruments TM4C1294 Connected LaunchPad I/O pins

Figure 4.8 shows the 90 I/O pins available on the TM4C1294NCPDT, which is the microcontroller used on the Connected LaunchPad. Pins on the TM4C family can be assigned to as many as seven different I/O functions, see Table 4.4. Pins can be configured for digital I/O, analog input, timer I/O, or serial I/O. For example PA0 can be digital I/O, serial input, I2C clock, Timer I/O, or CAN receiver. There are two buses used for I/O. Unlike the TM4C123, the digital I/O ports are only connected to the advanced high-performance bus. The microcontroller can perform I/O bus cycles simultaneous with instruction fetches from flash ROM. The TM4C1294NCPDT has eight UART ports, four SSI ports, ten I2C ports, two 12-bit ADCs, eight timers, two CAN ports, a USB interface, 8 PWM outputs, and an Ethernet port. Of the 90 I/O lines, twenty pins can be used for analog inputs to the ADC. The ADC can convert up to 1M samples per second. Table 4.4 lists the regular and alternate functions of the port pins.

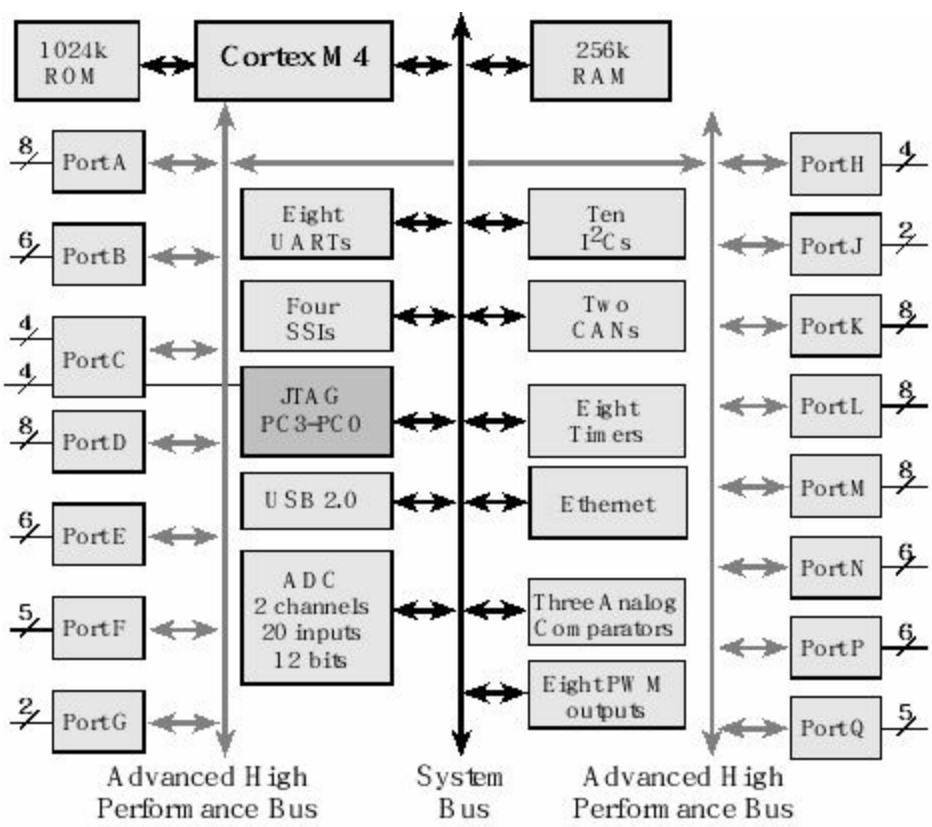


Figure 4.8. I/O port pins for the TM4C1294NCPDT microcontroller.

Figure 4.9 shows the pin locations of the two Booster Pack connectors. There are three methods to connect external circuits to the Connected LaunchPad. One method uses male to female jumper cable (e.g., item number 826 at [www.adafruit.com](http://www.adafruit.com)) or solder a solid wire into a female socket (e.g., Hirose DF11-2428SCA) creating a male-to-female jumper wire. In this method, you connect the female socket to the top of the LaunchPad and the male pin into a solderless breadboard. The second method uses male-to-male wires interfacing to the bottom of the LaunchPad. The third method uses two 49-pin right-angle headers so the entire LaunchPad can be plugged into a breadboard. You will need one each of Samtec parts TSW-149-09-L-S-RE and TSW-149-08-L-S-RA. This configuration is shown in Figure 4.9, and directions can be found at

<http://users.ece.utexas.edu/~valvano/arm/TM4C1294soldering.pdf>

The Connected LaunchPad has two switches and four LEDs. Switch SW1 is connected to pin PJ0, and SW2 is connected to PJ1. These two switches are negative logic and require enabling the internal pull up (**PUR**). A reset switch will reset the microcontroller and your software will start when you release the switch. Positive logic LEDs D1, D2, D3, and D4 are connected to PN1, PN0, PF4, and PF0 respectively. A power LED indicates that 3.3 volt power is present on the board. R19 is a 0 Ω resistor connecting PA3 and PQ2. Similarly, R20 is a 0 Ω resistor connecting PA2 and PQ3. You need to remove R19 if you plan to use both PA3 and PQ2. You need to remove R20 if you plan to use both PA2 and PQ3. See Figure 4.10.

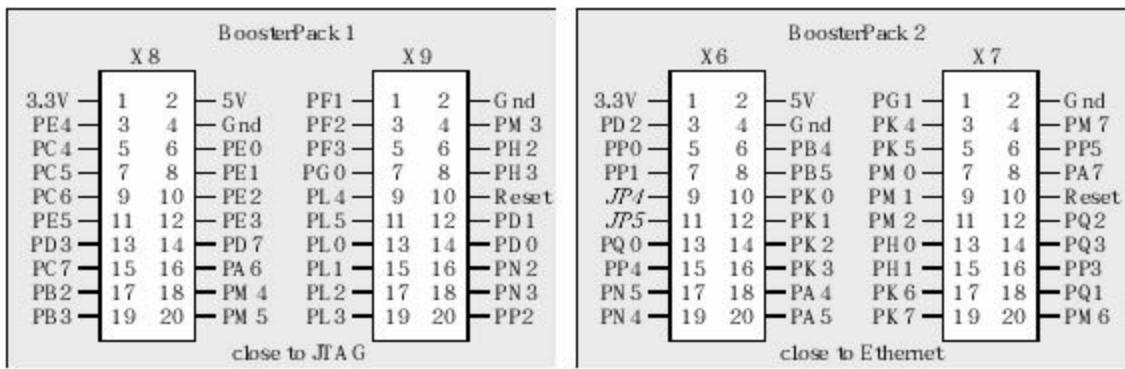


Figure 4.8. Interface connectors on the EK-TM4C1294-XL LaunchPad Evaluation Board.

Jumper JP1 has six pins creating three rows of two. Exactly one jumper should be connected in the JP1 block, which selects the power source. The top position is for BoosterPack power. The middle position draws power from the USB connector, labeled OTG, on the left side of the board near the Ethernet jack. We recommend placing the JP1 jump in the bottom position so power is drawn from the ICDI (Debug) USB connection. Under normal conditions, you should place jumpers in both J2 and J3. Jumpers J2 and J3 facilitate measuring current to the microcontroller. We recommend you place JP4 and JP5 in the “UART” position so PA1 and PA0 are connected to the PC as a virtual COM port. Your code runs on the 128-pin TM4C1294 microcontroller. There is a second TM4C microcontroller on the board, which acts as the JTAG debugger for your TM4C1294. You connect the Debug USB to a PC in order to download and debug software on the board. The other USB is for user applications.

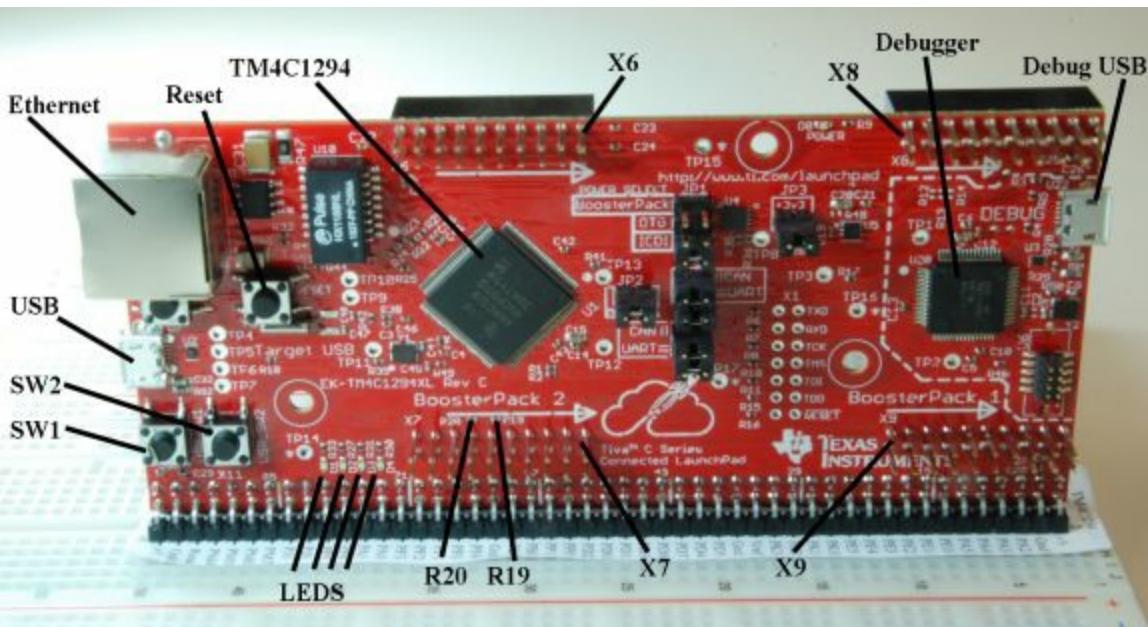


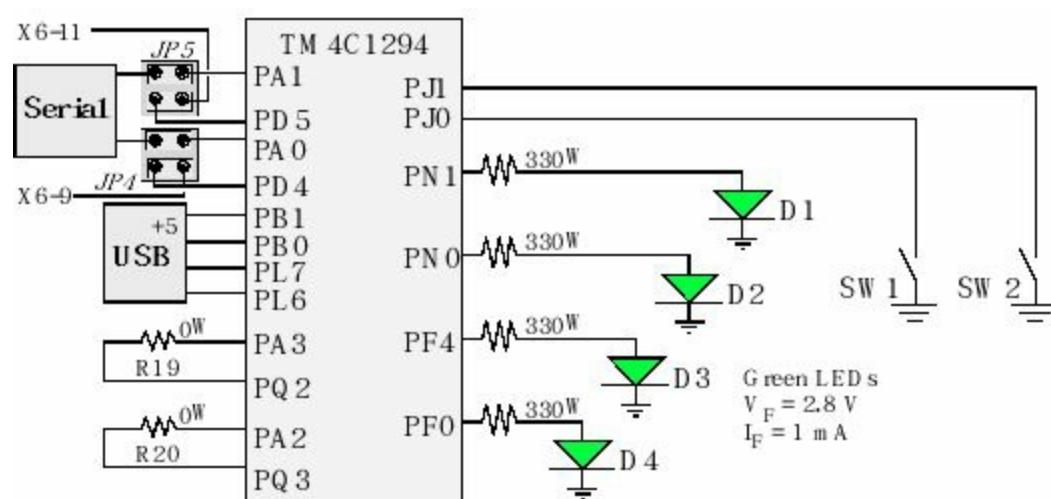
Figure 4.9. EK-TM4C1294-XL Connected LaunchPad.

Pin	Analog	1	2	3	5	6	7	11	13	14	15
PA0 -		U0Rx	I2C9SCL	T0CCP0	-	-	CAN0Rx	-	-	-	-
PA1 -		U0Tx	I2C9SDA	T0CCP1	-	-	CAN0Tx	-	-	-	-
PA2 -		U4Rx	I2C8SCL	T1CCP0	-	-	-	-	-	-	SSI0Clk
PA3 -		U4Tx	I2C8SDA	T1CCP1	-	-	-	-	-	-	SSI0Fss
PA4 -		U3Rx	I2C7SCL	T2CCP0	-	-	-	-	-	-	SSI0XDA
PA5 -		U3Tx	I2C7SDA	T2CCP1	-	-	-	-	-	-	SSI0XDA
PA6 -		U2Rx	I2C6SCL	T3CCP0	USB0EPEN	-	-	-	SSI0XDAT2	-	EPI0S8
PA7 -		U2Tx	I2C6SDA	T3CCP1	USB0PFLT	-	-	USB0EPEN	SSI0XDAT3	-	EPI0S9

PB0	USB0ID	U1Rx	I2C5SCL	T4CCP0	-	-	CAN1Rx	-	-	-	-
PB1	USB0VBUS	U1Tx	I2C5SDA	T4CCP1	-	-	CAN1Tx	-	-	-	-
PB2	-	-	I2C0SCL	T5CCP0	-	-	-	-	-	USB0STP	EPI0S27
PB3	-	-	I2C0SDA	T5CCP1	-	-	-	-	-	USB0CLK	EPI0S28
PB4	AIN10	U0CTS	I2C5SCL	-	-	-	-	-	-	-	SSI1Fss
PB5	AIN11	U0RTS	I2C5SDA	-	-	-	-	-	-	-	SSI1Clk
PC4	C1-	U7Rx	-	-	-	-	-	-	-	-	EPI0S7
PC5	C1+	U7Tx	-	-	-	-	RTCCLK	-	-	-	EPI0S6
PC6	C0+	U5Rx	-	-	-	-	-	-	-	-	EPI0S5
PC7	C0-	U5Tx	-	-	-	-	-	-	-	-	EPI0S4
PD0	AIN15	-	I2C7SCL	T0CCP0	C0o	-	-	-	-	-	SSI2XDA <sup>A</sup>
PD1	AIN14	-	I2C7SDA	T0CCP1	C1o	-	-	-	-	-	SSI2XDA <sup>A</sup>
PD2	AIN13	-	I2C8SCL	T1CCP0	C2o	-	-	-	-	-	SSI2Fss
PD3	AIN12	-	I2C8SDA	T1CCP1	-	-	-	-	-	-	SSI2Clk
PD4	AIN7	U2Rx	-	T3CCP0	-	-	-	-	-	-	SSI1XDA <sup>A</sup>
PD5	AIN6	U2Tx	-	T3CCP1	-	-	-	-	-	-	SSI1XDA <sup>A</sup>
PD6	AIN5	U2RTS	-	T4CCP0	USB0EPEN	-	-	-	-	-	SSI2XDA <sup>A</sup>
PD7	AIN4	U2CTS	-	T4CCP1	USB0PFLT	-	-	-	-	-	SSI2XDA <sup>A</sup>
PE0	AIN3	U1RTS	-	-	-	-	-	-	-	-	-
PE1	AIN2	U1DSR	-	-	-	-	-	-	-	-	-
PE2	AIN1	U1DCD	-	-	-	-	-	-	-	-	-
PE3	AIN0	U1DTR	-	-	-	-	-	-	-	-	-
PE4	AIN9	U1RI	-	-	-	-	-	-	-	-	SSI1XDA <sup>A</sup>
PE5	AIN8	-	-	-	-	-	-	-	-	-	SSI1XDA <sup>A</sup>
PF0	-	-	-	EN0LED0	M0PWM0	-	-	-	-	SSI3XDAT1	TRD2
PF1	-	-	-	EN0LED2	M0PWM1	-	-	-	-	SSI3XDAT0	TRD1
PF2	-	-	-	-	M0PWM2	-	-	-	-	SSI3Fss	TRD0
PF3	-	-	-	-	M0PWM3	-	-	-	-	SSI3Clk	TRCLK
PF4	-	-	-	EN0LED1	M0FAULT0	-	-	-	-	SSI3XDAT2	TRD3
PG0	-	I2C1SCL	-	EN0PPS	M0PWM4	-	-	-	-	-	EPI0S11
PG1	-	I2C1SDA	-	-	M0PWM5	-	-	-	-	-	EPI0S10
PH0	-	U0RTS	-	-	-	-	-	-	-	-	EPI0S0
PH1	-	U0CTS	-	-	-	-	-	-	-	-	EPI0S1
PH2	-	U0DCD	-	-	-	-	-	-	-	-	EPI0S2
PH3	-	U0DSR	-	-	-	-	-	-	-	-	EPI0S3
PJ0	-	U3Rx	-	EN0PPS	-	-	-	-	-	-	-
PJ1	-	U3Tx	-	-	-	-	-	-	-	-	-
PK0	AIN16	U4Rx	-	-	-	-	-	-	-	-	EPI0S0
PK1	AIN17	U4Tx	-	-	-	-	-	-	-	-	EPI0S1
PK2	AIN18	U4RTS	-	-	-	-	-	-	-	-	EPI0S2
PK3	AIN19	U4CTS	-	-	-	-	-	-	-	-	EPI0S3
PK4	-	I2C3SCL	-	EN0LED0	M0PWM6	-	-	-	-	-	EPI0S32
PK5	-	I2C3SDA	-	EN0LED2	M0PWM7	-	-	-	-	-	EPI0S31
PK6	-	I2C4SCL	-	EN0LED1	M0FAULT1	-	-	-	-	-	EPI0S25
PK7	-	U0RI	I2C4SDA	-	RTCCLK	M0FAULT2	-	-	-	-	EPI0S24
PL0	-	-	I2C2SDA	-	-	M0FAULT3	-	-	-	USB0D0	EPI0S16
PL1	-	-	I2C2SCL	-	-	PhA0	-	-	-	USB0D1	EPI0S17
PL2	-	-	-	C0o	PhB0	-	-	-	-	USB0D2	EPI0S18
PL3	-	-	-	C1o	IDX0	-	-	-	-	USB0D3	EPI0S19
PL4	-	-	T0CCP0	-	-	-	-	-	-	USB0D4	EPI0S26
Pin	Analog	1	2	3	5	6	7	11	13	14	15
PL5	-	-	T0CCP1	-	-	-	-	-	-	USB0D5	EPI0S33

PL6 USB0DP	-	-	T1CCP0	-	-	-	-	-
PL7 USB0DM	-	-	T1CCP1	-	-	-	-	-
PM0 -	-	-	T2CCP0	-	-	-	-	EPI0S15
PM1 -	-	-	T2CCP1	-	-	-	-	EPI0S14
PM2 -	-	-	T3CCP0	-	-	-	-	EPI0S13
PM3 -	-	-	T3CCP1	-	-	-	-	EPI0S12
PM4 TMPR3	U0CTS	-	T4CCP0	-	-	-	-	-
PM5 TMPR2	U0DCD	-	T4CCP1	-	-	-	-	-
PM6 TMPR1	U0DSR	-	T5CCP0	-	-	-	-	-
PM7 TMPR0	U0RI	-	T5CCP1	-	-	-	-	-
PN0 -	U1RTS	-	-	-	-	-	-	-
PN1 -	U1CTS	-	-	-	-	-	-	-
PN2 -	U1DCD U2RTS	-	-	-	-	-	-	EPI0S29
PN3 -	U1DSR U2CTS	-	-	-	-	-	-	EPI0S30
PN4 -	U1DTR U3RTS	I2C2SDA	-	-	-	-	-	EPI0S34
PN5 -	U1RI U3CTS	I2C2SCL	-	-	-	-	-	EPI0S35
PP0 C2+	U6Rx	-	-	-	-	-	-	SSI3XDA
PP1 C2-	U6Tx	-	-	-	-	-	-	SSI3XDA
PP2 -	U0DTR	-	-	-	-	-	USB0NXT	EPI0S29
PP3 -	U1CTS U0DCD	-	-	RTCCLK	-	-	USB0DIR	EPI0S30
PP4 -	U3RTS U0DSR	-	-	-	-	-	USB0D7	-
PP5 -	U3CTS I2C2SCL	-	-	-	-	-	USB0D6	-
PQ0 -	-	-	-	-	-	-	SSI3Clk	EPI0S20
PQ1 -	-	-	-	-	-	-	SSI3Fss	EPI0S21
PQ2 -	-	-	-	-	-	-	SSI3XDAT0	EPI0S22
PQ3 -	-	-	-	-	-	-	SSI3XDAT1	EPI0S23
PQ4 -	U1Rx	-	-	DIVSCLK	-	-	-	-

**Table 4.4. PMCx bits in the GPIO PCTL register on the TM4C1294 specify alternate functions.**  
**PD7 can be NMI by setting PCTL to 8. PL6 and PL7 are hardwired to the USB device.**



**Figure 4.10. Switch and LED interfaces on the Connected LaunchPad Evaluation Board.** The zero ohm resistors can be removed so all the pins can be used.

Each pin has one configuration bit in the **AMSEL** register. We set this bit to connect the port pin to the ADC or analog comparator. For digital functions, each pin also has four bits in the **PCTL** register, which we set to specify the alternative function for that pin (0 means regular I/O port). Table 4.4 shows the 4-bit **PCTL** configuration used to connect each pin to its alternate function. For example, column “3” means set 4-bit field in **PCTL** to 0011.

Pins PC3 – PC0 were left off Table 4.4 because these four pins are reserved for the JTAG debugger and should not be used for regular I/O. Notice, some alternate function modules (e.g., U0Rx) only exist on one pin (PA0). While other functions could be mapped to two or three pins. For example, T0CCP0 could be mapped to one of the following: PA0, PD0, or PL4.

The PCTL bits in Table 4.4 can be tricky to understand. For example, if we wished to use UART6 on pins PP0 and PP1, we would set bits 1,0 in the **DEN** register (enable digital), clear bits 1,0 in the **AMSEL** register (disable analog), write a 0001,0001 to bits 7–0 in the **PCTL** register (enable UART6 functionality), and set bits 1,0 in the **AFSEL** register (enable alternate function). If we wished to sample an analog signal on PD0, we would set bit 0 in the alternate function select register **AFSEL**, clear bit 0 in the digital enable register **DEN** (disable digital), set bit 0 in the analog mode select register **AMSEL** (enable analog), and activate one of the ADCs to sample channel 15.

Additional examples will be presented throughout the book.

Jumpers JP4 and JP5 select whether the serial port on UART0 (PA1 – PA0) or on UART2 (PD5 – 4) is linked through the debugger cable to the PC. The serial link is a physical UART as seen by the TM4C1294 and is mapped to a virtual COM port on the PC. The USB device interface uses PL6 and PL7. The JTAG debugger requires pins PC3 – PC0.

To use the negative logic switches, make the pins digital inputs, and activate the internal pull-up resistors. In particular, you will activate the Port J clock, clear bits 0 and

1 in **GPIO\_PORTJ\_DIR\_R** register, set bits 0 and 1 in **GPIO\_PORTJ\_DEN\_R** register, and set bits 0 and 1 in **GPIO\_PORTJ\_PUR\_R** register. The LED interfaces are positive logic. To use the LEDs, make the PN1, PN0, PF4, and PF0 pins an output. You will activate the Port N clock, set bits 0 and 1 in **GPIO\_PORTN\_DIR\_R** register, and set bits 0 and 1 in **GPIO\_PORTN\_DEN\_R** register. You will activate the Port F clock, set bits 0 and 4 in **GPIO\_PORTF\_DIR\_R** register, and set bits 0 and 4 in **GPIO\_PORTF\_DEN\_R** register.

## 4.2. Basic Concepts of Input and Output Ports

The simplest I/O port on a microcontroller is the parallel port. A parallel I/O port is a simple mechanism that allows the software to interact with external devices. It is called parallel because multiple signals can be accessed all at once. An **input port**, which is read only, allows the software to read external digital signals. That means a read cycle access from the port address returns the values existing on the inputs at that time. In particular, the tristate driver (triangle shaped circuit in Figure 4.11) will drive the input signals onto the data bus during a read cycle from the port address. A write cycle access to an input port usually produces no effect. The digital values existing on the input pins are copied into the microcontroller when the software executes a read from the port address. There are no digital input-only ports on the TM4C family of microcontrollers. TM4C microcontrollers have 5V-tolerant digital inputs, meaning an input voltage from 2.0 to 5.0 V will be considered high, and a voltage from 0 to 1.3 V will be considered as low. Check to see the pins on your microcontroller are 5-V tolerant.

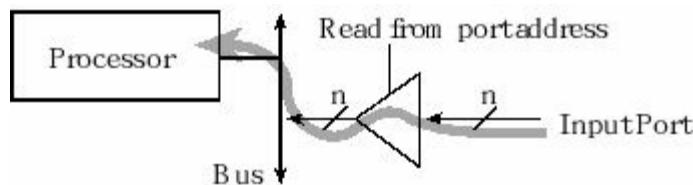


Figure 4.11. A read only input port allows the software to sense external digital signals.

**Checkpoint 4.1:** What happens if the software writes to an input port like Figure 4.11?

While an input device usually just involves the software reading the port, an output port can participate in both the read and write cycles very much like a regular memory. Figure 4.12 describes a **readable output port**. A write cycle to the port address will affect the values on the output pins. In particular, the microcontroller places information on the data bus and that information is clocked into the D flip-flops. Since it is a readable output, a read cycle access from the port address returns the current values existing on the port pins. There are no output-only ports on the TM4C family of microcontrollers.

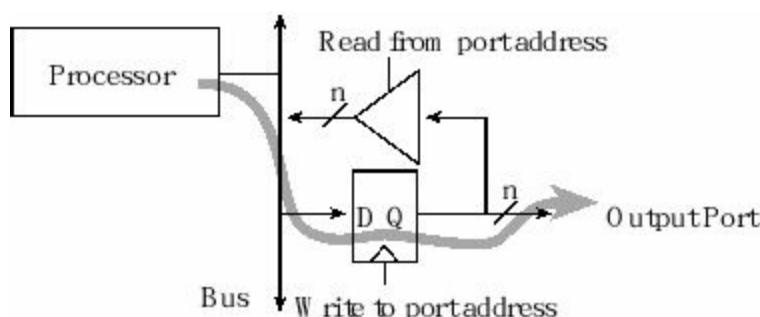


Figure 4.12. A readable output port allows the software to generate external digital signals.

**Checkpoint 4.2:** What happens if the software reads from an output port like Figure 4.11?

To make the microcontroller more marketable, the ports on TM4C microcontrollers can be software-specified to be either inputs or outputs. Microcontrollers use the concept of a **direction register** to determine whether a pin is an input (direction register bit is 0) or an output (direction register bit is 1), as shown in Figure 4.13. We define an initialization **ritual** as a program executed once during start up that initializes hardware and software. If the ritual makes the direction bit zero, the port pin behaves like a simple input, and if it makes the direction bit one, the port pin becomes a readable output. Each digital port pin has its own direction bit. This means some pins on a port may be inputs while others are outputs. The digital port pins on most microcontrollers are bidirectional, operating similar to Figure 4.13.

**Common Error:** Many program errors can be traced to confusion between I/O ports and regular memory. For example, you should not write to an input port, and sometimes we cannot read from an output port.

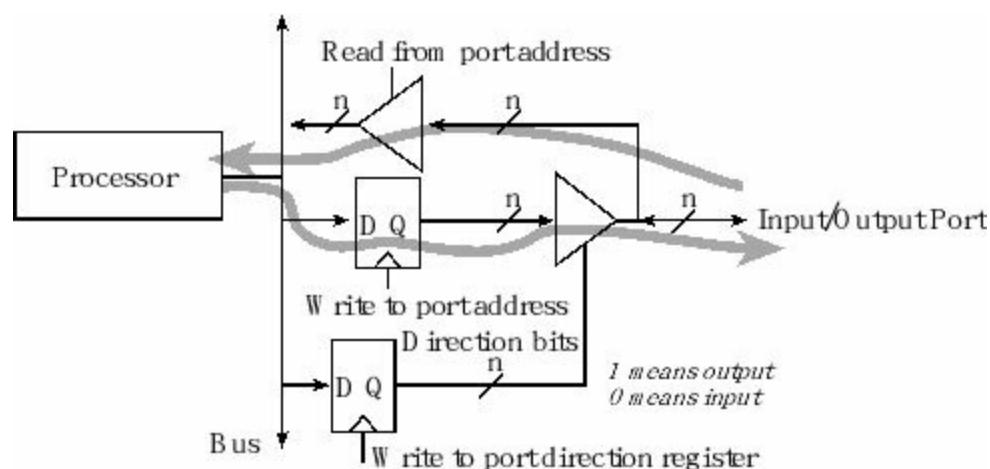


Figure 4.13. A bidirectional port can be configured as a read-only input port or a readable output port.

## 4.2.1. I/O Programming and the Direction Register

On most embedded microcontrollers, the I/O ports are memory mapped. This means the software can access an input/output port simply by reading from or writing to the appropriate address. It is important to realize that even though I/O operations “look” like reads and writes to memory variables, the I/O ports often DO NOT act like memory. For example, some bits are read-only, some are write-only, some can only be cleared, others can only be set, and some bits cannot be modified. To make our software easier to understand we include symbolic definitions for the I/O ports. We set the direction register(e.g., **GPIO\_PORTF\_DIR\_R**) to specify which pins are input and which are output. Individual port pins can be general purpose I/O (GPIO) or have an alternate function. We will set bits in the alternate function register (e.g., **GPIO\_PORTF\_AFSEL\_R**) when we wish to activate the alternate functions listed in Tables 4.1, 4.3, and 4.4. To use a pin as a digital input or output, we must set the corresponding bit in the digital enable register(e.g., **GPIO\_PORTF\_DEN\_R**). To use a pin as an analog input we must set the corresponding bit in the analog mode select register (e.g., **GPIO\_PORTF\_AMSEL\_R**). Typically, we write to the direction and alternate function registers once during the initialization phase. We use the data register(e.g., **GPIO\_PORTF\_DATA\_R**) to perform the actual input/output on the port. Table 4.5 shows some of the parallel port registers for the TM4C123. Each of the ports has a clock, which can be separately enabled by writing to the **SYSCTL\_RCGCGPIO\_R** register.

The only differences among the TM4C family are the number of ports and available pins in each port. For example, the TM4C1294 has fifteen digital I/O ports A (8 bits), B (6 bits), C (8 bits), D (8 bits), E (6 bits), F (5 bits), G (2 bits), H (4 bits), J (2 bits), K (8 bits), L (8 bits), M (8 bits), N(6 bits), P (6 bits), and Q (5 bits). Furthermore, the TM4C1294 has different addresses for ports. Refer to the file **tm4c1294ncpdt.h** or to the data sheet for more the specific addresses of its I/O ports.

**Common Error:** You will get a bus fault if you access a port without enabling its clock.

Address	7	6	5	4	3	2	1	0	Name
\$400F.E608	-	-	GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_RCGCGPIO_R
\$400F.EA08	-	-	GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_PGPIO_R
\$4000.43FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTA_DATA_R
\$4000.4400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTA_DIR_R
\$4000.4420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTA_AFSEL_R
\$4000.4510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTA_PUR_R
\$4000.451C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTA_DEN_R
\$4000.4524	1	1	1	1	1	1	1	1	GPIO_PORTA_CR_R
\$4000.4528	0	0	0	0	0	0	0	0	GPIO_PORTA_AMSEL_R
\$4000.53FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTB_DATA_R
\$4000.5400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTB_DIR_R
\$4000.5420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTB_AFSEL_R
\$4000.5510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTB_PUR_R
\$4000.551C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTB_DEN_R
\$4000.5524	1	1	1	1	1	1	1	1	GPIO_PORTB_CR_R
\$4000.5528	0	0	AMSEL	AMSEL	0	0	0	0	GPIO_PORTB_AMSEL_R
\$4000.63FC	DATA	DATA	DATA	DATA	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_DATA_R
\$4000.6400	DIR	DIR	DIR	DIR	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_DIR_R
\$4000.6420	SEL	SEL	SEL	SEL	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_AFSEL_R
\$4000.6510	PUE	PUE	PUE	PUE	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_PUR_R
\$4000.651C	DEN	DEN	DEN	DEN	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_DEN_R
\$4000.6524	1	1	1	1	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_CR_R

\$4000.6528	AMSEL	AMSEL	AMSEL	AMSEL	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_AMSEL_R
\$4000.73FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTD_DATA_R
\$4000.7400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTD_DIR_R
\$4000.7420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTD_AFSEL_R
\$4000.7510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTD_PUR_R
\$4000.751C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTD_DEN_R
\$4000.7524	CR	1	1	1	1	1	1	1	GPIO_PORTD_CR_R
\$4000.7528	0	0	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTD_AMSEL_R
\$4002.43FC	-	-	DATA	DATA	DATA	DATA	DATA	DATA	GPIO PORTE DATA R
\$4002.4400	-	-	DIR	DIR	DIR	DIR	DIR	DIR	GPIO PORTE DIR R
\$4002.4420	-	-	SEL	SEL	SEL	SEL	SEL	SEL	GPIO PORTE AFSEL R
\$4002.4510	-	-	PUE	PUE	PUE	PUE	PUE	PUE	GPIO PORTE PUR R
\$4002.451C	-	-	DEN	DEN	DEN	DEN	DEN	DEN	GPIO PORTE DEN R
\$4002.4524	-	-	1	1	1	1	1	1	GPIO PORTE CR R
\$4002.4528	-	-	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO PORTE AMSEL R
\$4002.53FC	-	-	-	DATA	DATA	DATA	DATA	DATA	GPIO PORTF DATA R
\$4002.5400	-	-	-	DIR	DIR	DIR	DIR	DIR	GPIO PORTF DIR R
\$4002.5420	-	-	-	SEL	SEL	SEL	SEL	SEL	GPIO PORTF AFSEL R
\$4002.5510	-	-	-	PUE	PUE	PUE	PUE	PUE	GPIO PORTF PUR R
\$4002.551C	-	-	-	DEN	DEN	DEN	DEN	DEN	GPIO PORTF DEN R
\$4002.5524	-	-	-	1	1	1	1	CR	GPIO PORTF CR R
\$4002.5528	-	-	-	0	0	0	0	0	GPIO PORTF AMSEL R
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0	
\$4000.452C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO PORTA_PCTL_R
\$4000.552C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO PORTB_PCTL_R
\$4000.652C	PMC7	PMC6	PMC5	PMC4	0x1	0x1	0x1	0x1	GPIO PORTC_PCTL_R
\$4000.752C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO PORTD_PCTL_R
\$4002.452C	----	----	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO PORTE_PCTL_R
\$4002.552C	----	----	----	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO PORTF_PCTL_R
\$4000.6520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO PORTC_LOCK_R
\$4000.7520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO PORTD_LOCK_R
\$4002.5520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO PORTF_LOCK_R

**Table 4.5. Some TM4C123 parallel ports. Each register is 32 bits wide. For PMC bits, see Table 4.3, 4.4.**

To initialize an I/O port for general use we perform seven steps. We will skip steps three four and six in this chapter because the default state after a reset is to disable analog function and disable alternate function. First, we activate the clock for the port by setting the corresponding bit in **RCGCGPIO** register. Because it takes time for the clock to stabilize, we next will wait for its status bit in the **PRGPIO** to be true. Second, we unlock the port; unlocking is needed only for pins PD7, and PF0 on the TM4C123. The only pin needing unlocking on the TM4C1294 is PD7. Third, we disable the analog function of the pin, because we will be using the pin for digital I/O. Fourth, we clear bits in the **PCTL** (Tables 4.3, 4.4) to select regular digital function. Fifth, we set its direction register. The direction register specifies bit for bit whether the corresponding pins are input or output. A bit in **DIR** set to 0 means input and 1 means output. Sixth, we clear bits in the alternate function register, and lastly, we enable the digital port. Turning on the clock must be first but the other steps can occur in any order.

In this first example we will make PF4 and PF0 input, and we will make PF3 PF2 and PF1 output, as shown in Programs 4.1a and 4.1b. To use Port F we first must activate its clock (bit 5) in the **SYSCTL\_RCGCGPIO\_R** register. The second step is to unlock the port (TM4C123 only), by writing a special value to the **LOCK** register, followed by setting bits in the **CR** register. The fifth step is to specify whether the pin is an input or an output by clearing or setting bits in the **DIR** register. The last step is to enable the corresponding I/O pins by writing ones to the **DEN** register. To run this example on the LaunchPad, we also set bits in the **PUR** register for the two switch inputs (Figure 4.6) to have an internal pull-up resistor.

When the software reads from location 0x400253FC, the bottom 8 bits are returned with the values currently on Port F. The top 24 bits are returned zero. As shown in Figure 4.13, when reading an I/O port, the input pins report the high/low state currently on the input, and the output pins show the value last written to the port. The function **PortF\_Input** will read from all five Port F pins, and return a value depending on the status of the input pins at the time of the read. As shown in Figure 4.13, when writing to an I/O port, the input pins are not affected, and the output pins are changed to the value written to the port. That value remains until written again. The function **PortF\_Output** will write new values to the three output pins. The **#include** will define symbolic names for all the I/O ports for that microcontroller. The header files are in the **inc** folder. Use the one for your microcontroller.

```
#include "inc/tm4c123gh6pm.h"
void PortF_Init(void){
    SYSCTL_RCGCGPIO_R |= 0x00000020; // 1) activate clock for Port F
    while((SYSCTL_PGPIO_R&0x00000020) == 0){};// ready?
    GPIO_PORTF_LOCK_R = 0x4C4F434B; // 2) unlock GPIO Port F
    GPIO_PORTF_CR_R = 0x1F; // allow changes to PF4-0
    GPIO_PORTF_DIR_R = 0x0E; // 5) PF4,PF0 in, PF3-1 out
    GPIO_PORTF_PUR_R = 0x11; // enable pull-up on PF0 and PF4
    GPIO_PORTF_DEN_R = 0x1F; // 7) enable digital I/O on PF4-0
}
uint32_t PortF_Input(void){
    return (GPIO_PORTF_DATA_R&0x11); // read PF4,PF0 inputs
}
void PortF_Output(uint32_t data){ // write PF3-PF1 outputs
    GPIO_PORTF_DATA_R = data;
}
```

Program 4.1a. A set of functions using PF4,PF0 as inputs and PF3-1 as outputs (InputOutput\_xxx.zip).

Only PD7, and PF0 on the TM4C123 and PD7 on the TM4C1294 need to be unlocked. All the other bits on the two microcontrollers are always unlocked.

**Observation:** Programs in Chapter 4 will skip writing zeros to AMSEL AFSEL and PCTL because these three registers are initialized to zero by the hardware reset. The versions on the web site will write zeros to the AMSEL AFSEL and PCTL registers.

**PortF\_Init**

```

LDR R1, =SYSCTL_RCGCGPIO_R ; 1) activate clock for Port F
LDR R0, [R1]
ORR R0, R0, #0x20      ; set bit 5 to turn on clock
STR R0, [R1]
NOP
NOP          ; allow time for clock to finish
LDR R1, =GPIO_PORTF_LOCK_R ; 2) unlock the lock register
LDR R0, =0x4C4F434B      ; unlock GPIO Port F Commit Register
STR R0, [R1]
LDR R1, =GPIO_PORTF_CR_R ; enable commit for Port F
MOV R0, #0xFF           ; 1 means allow access
STR R0, [R1]
LDR R1, =GPIO_PORTF_DIR_R ; 5) set direction register
MOV R0, #0x0E            ; PF0 and PF7-4 input, PF3-1 output
STR R0, [R1]
LDR R1, =GPIO_PORTF_PUR_R ; pull-up resistors for PF4,PF0
MOV R0, #0x11            ; enable weak pull-up on PF0 and PF4
STR R0, [R1]
LDR R1, =GPIO_PORTF_DEN_R ; 7) enable Port F digital port
MOV R0, #0xFF            ; 1 means enable digital I/O
STR R0, [R1]
BX LR

```

### PortF\_Input

```

LDR R1, =GPIO_PORTF_DATA_R ; pointer to Port F data
LDR R0, [R1]      ; read all of Port F
AND R0,R0,#0x11    ; just the input pins, bits 4,0
BX LR             ; return R0 with inputs

```

### PortF\_Output

```

LDR R1, =GPIO_PORTF_DATA_R ; pointer to Port F data
STR R0, [R1]      ; write to PF3-1
BX LR

```

Program 4.1b. A set of functions using PF4,PF0 as inputs and PF3-1 as outputs (InputOutput\_xxxasm.zip).

**Checkpoint 4.3:** Does the entire port need to be defined as input or output, or can some pins be input while others are output?

**Checkpoint 4.4:** How do we change Program 4.1 to run using Port B?

In Program 4.1 the assumption was the software module had access to all of Port F. In other words, this software owned all pins of Port F. The TM4C123 Port F has only 5 pins, and we used them all. In most cases, a software module needs access to only some of the port pins. If two or more software modules access the same port, a conflict will occur if one module changes modes or output values set

by another module. It is good software design to write **friendly** software, which only affects the individual pins as needed. Friendly software does not change the other bits in a shared register. Conversely, **unfriendly** software modifies more bits of a register than it needs to. The difficulty of unfriendly code is each module will run properly when tested by itself, but weird bugs result when two or more modules are combined.

Consider the problem that a software module needs to output to just Port D bit 7. After enabling the clock for Port D, we use read-modify-write software to initialize just pin 7. The following initialization does not modify the configurations for the other 7 bits in Port D.

```
SYSCTL_RCGCGPIO_R |= 0x08;      // 1) activate clock for Port D
while((SYSCTL_PRGPIO_R&0x08) == 0){};// ready?
GPIO_PORTD_LOCK_R = 0x4C4F434B; // 2) unlock GPIO Port D7
GPIO_PORTD_CR_R |= 0x80;        // allow changes to PD7
GPIO_PORTD_DIR_R |= 0x80;       // 5) PD7 out
GPIO_PORTD_DEN_R |= 0x80;       // 7) enable digital I/O on PD7
```

There is no conflict if multiple modules enable the clock for Port D. There are two friendly ways on TM4C microcontrollers to access individual port bits. The first method is to use read-modify-write software to change just the pins of interest, leaving the other pins unchanged. A read-or-write sequence can be used to set bits. For example, if we wished to set PD7

LDR R1, =GPIO_PORTD_DATA_R LDR R0, [R1] ; previous ORR R0, R0, #0x80 ; set bit 7 STR R0, [R1]	// make PD7 high GPIO_PORTD_DATA_R  = 0x80;
---	---

A read-and-write sequence can be used to clear one or more bits. If we wished to clear PD7

LDR R1, =GPIO_PORTD_DATA_R LDR R0, [R1] ; previous BIC R0, R0, #0x80 ; clear bit 7 STR R0, [R1]	// make PD7 low GPIO_PORTD_DATA_R &= ~0x80;
---	---

The second method uses the **bit-specific addressing**. The Texas Instruments microcontrollers implement a more flexible way to access port pins than the bit-banding described in Volume 2. This bit-specific addressing doesn't work for all the I/O registers, just the parallel port data registers. The Texas Instruments mechanism allows collective access to 0 to 8 bits in a data port. We define eight address offset constants in Table 4.6.

If we wish to access bit	Constant
7	0x0200
6	0x0100
5	0x0080

4	0x0040
3	0x0020
2	0x0010
1	0x0008
0	0x0004

**Table 4.6. Address offsets used to specify individual data port bits.**

Basically, if we are interested in bit b, the constant is  $4 * 2^b$ . There 256 possible bit combinations we might be interested in accessing, from all of them to none of them. Each possible bit combination has a separate address for accessing that combination. For each bit we are interested in, we add up the corresponding constants from Table 4.6 and then add that sum to the base address for the port. The base addresses for the data ports for each microcontroller can be found in its data sheet; open the data sheet for your microcontroller, go to the **GPIO** chapter, **Register Descriptions** section, and search for **GPIO DATA**. Figure 4.14 shows a snapshot of the TM4C123 data sheet, illustrating the base address for Port A is 0x4000.4000. For example, assume we are interested in Port A bits 1, 2, and 3 on the TM4C123. We look up the constants for bits 1,2,3 in Table 4.8, which are 0x0008, 0x0010, and 0x0020. The sum of

$$0x4000.4000 + 0x0008 + 0x0010 + 0x0020$$

is the address 0x4000.4038. If we read from 0x4000.4038 only bits 1, 2, and 3 will be returned. If we write to this address only bits 1, 2, and 3 will be modified.

The screenshot shows a PDF document window for the TM4C123 data sheet. On the left, there is a navigation pane with a tree view of register descriptions. The node '10.5. Register Descriptions' is expanded, and 'Register 1: GPIO Data (GPIO DATA), offset 0x000' is selected and highlighted with a blue background. The main content area is titled 'Register 1: GPIO Data (GPIO DATA)'. It contains several paragraphs of text explaining the register's function and how it interacts with other GPIO registers. At the bottom of this section, there is a table with two columns: 'GPIO Data (GPIO DATA)' and 'Base address'. An arrow points from the text 'GPIO Port A (APB) base: 0x4000.4000' to the column header 'Base address'.

GPIO Data (GPIO DATA)	Base address
GPIO Port A (APB) base: 0x4000.4000	Base address
GPIO Port A (AHB) base: 0x4005.8000	
GPIO Port B (APB) base: 0x4000.5000	
GPIO Port B (AHB) base: 0x4005.9000	
GPIO Port C (APB) base: 0x4000.6000	
GPIO Port C (AHB) base: 0x4005.A000	
GPIO Port D (APB) base: 0x4000.7000	
GPIO Port D (AHB) base: 0x4005.B000	

**Figure 4.14. Snapshot of the TM4C123 data sheet, showing how to look up**

## GPIO base addresses.

The TM4C1294 uses the advanced high-performance bus (AHB) to access the GPIO registers. The base addresses for the TM4C1294 ports are listed in Table 4.7.

GPIO Port A 0x4006.0000	0x4005.8000	GPIO Port J
GPIO Port B 0x4006.1000	0x4005.9000	GPIO Port K
GPIO Port C 0x4006.2000	0x4005.A000	GPIO Port L
GPIO Port D 0x4006.3000	0x4005.B000	GPIO Port M
GPIO Port E 0x4006.4000	0x4005.C000	GPIO Port N
GPIO Port F 0x4006.5000	0x4005.D000	GPIO Port P
GPIO Port G 0x4006.6000	0x4005.E000	GPIO Port Q
GPIO Port H	0x4005.F000	

**Table 4.7. Base addresses for the TM4C1294 GPIO ports.**

If we want to read and write all 8 bits of Port A, the constants will add up to 0x03FC. Notice that the sum of the base address (0x4000.4000) and all the constants yields the 0x4000.43FC address used in Table 4.5 and Program 4.1. In other words, read and write operations to **GPIO\_PORTA\_DATA\_R** will access all 8 bits of Port A. If we are interested in just bit 5 of Port A, we add 0x0080 to 0x4000.4000, and we can define this in C and in assembly as

```
#define PA5 ((volatile uint32_t *)0x40004080))  
PA5 EQU 0x40004080
```

Now, a simple write operation can be used to set **PA5**. The following code is friendly because it does not modify the other 7 bits of Port A. This code sets Port A bit 5.

```
PA5 = 0x20; // make PA5 high
```

A simple write sequence will clear **PA5**. The following code is also friendly.

```
PA5 = 0x00; // make PA5 low
```

A read from **PA5** will return 0x20 or 0x00 depending on whether the pin is high or low, respectively. If **PA5** is an output, the following code is also friendly.

```
PA5 = PA5^0x20; // toggle PA5
```

**Checkpoint 4.5:** What happens if we write to location 0x4000.4000?

**Checkpoint 4.6:** Specify a #define that allows us to access bits 7 and 1 of Port A. Use this #define to make both bits 7 and 1 of Port A high.

**Checkpoint 4.7:** Specify a #define that allows us to access bits 6, 1, 0 of Port B. Use this #define to make bits 6, 1 and 0 of Port B high.

To understand the port definitions in C, we remember **#define** is simply a copy paste. E.g.,

```
data = PA5;
```

becomes

```
data = (*((volatile uint32_t *)0x40004080));
```

To understand why we define ports this way, let's break this port definition into pieces. First, 0x40004080 is the address of Port Abit 5. If we write just **#define PA5 0x40004080** it will create

```
data = 0x40004080;
```

which does not read the contents of PA5 as desired. This means we need to dereference the address. If we write **#define PA5 (\*0x40004080)** it will create

```
data = (*0x40004080);
```

This will attempt to read the contents at 0x40004080, but doesn't know whether to read 8, 16, or 32 bits. So the compiler gives a syntax error because the type of data does not match the type of (\*0x40004080). To solve a type mismatch in C we **typecast**, placing a (new type) in front of the object we wish to convert. We wish force the type conversion to unsigned 32 bits, so we modify the definition to include the typecast,

```
#define PA5 (*((volatile uint32_t *)0x40004080))
```

The **volatile** is added because the value of a port can change beyond the direct action of the software. It forces the C compiler to read a new value each time through a loop and not rely on the previous value.

## 4.2.2. Switch Inputs and LED Outputs

There are four ways to interface a switch to the microcontroller as shown in Figure 4.15.

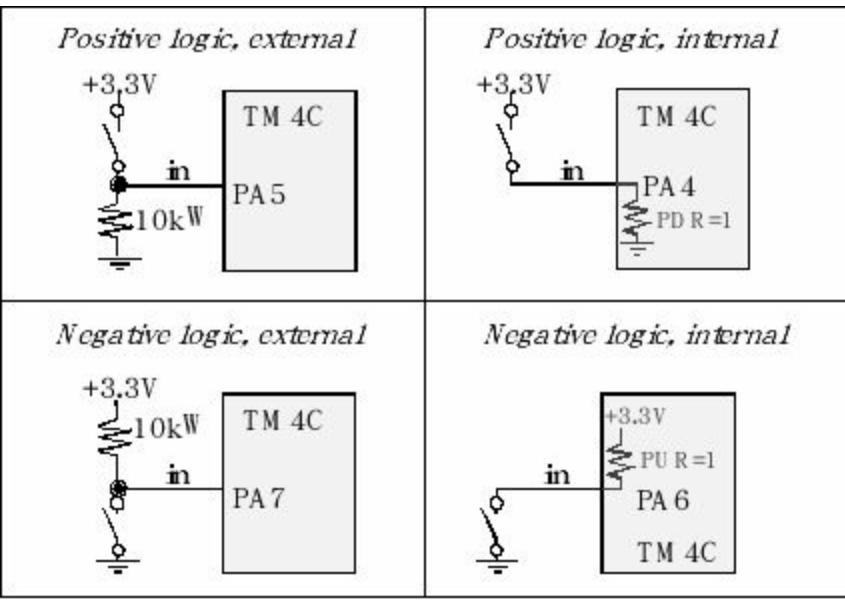


Figure 4.15. Interface of a switch to a microcomputer input.

We can use either positive or negative logic, and we can use an external resistor or select an internal resistor. Notice the positive logic circuit with external resistor is essentially the same as the positive logic circuit with internal resistance; the difference lies with whether the pull-down resistor is connected externally as a  $10\text{ k}\Omega$  resistor or internally by setting the corresponding PDR bit during software initialization.

In all cases we will initialize the pin as an input. The initialization function will enable the clock, clear the direction register bit to specify input, and enable the pin. In Program 4.2, we will interface PA5 to a switch using an external resistor and positive logic. Notice the software is friendly because it just affects PA5 without affecting the other bits in Port A. The input function reads Port A and returns a true (0x20) if the switch is pressed and returns a false (0) if the switch is not pressed. The first function uses the bit-specific address to get just PA5, while the second reads the entire port and selects bit 5 using a logical AND.

```
#define PA5 (*((volatile uint32_t *)0x40004080))

PA5 EQU 0x40004080
Switch_Init
    LDR R1,
    =SYSCTL_RCGCGPIO_R
    LDR R0, [R1]
    ORR R0, R0, #0x01
    STR R0, [R1] ; Port A clock
    NOP ; time for clock to finish
    NOP
    LDR R1,
    =GPIO_PORTA_DIR_R
    LDR R0, [R1]
    BIC R0, #0x20 ; PA5 input
    STR R0, [R1]
    LDR R1,
```

```
void Switch_Init(void){
    SYSCTL_RCGCGPIO_R |= 0x01;
    // 1) activate clock for Port A

    while((SYSCTL_PGPIO_R&0x01)
        == 0){};// ready?
    // 2) no need to unlock GPIO Port A
    GPIO_PORTA_DIR_R &= ~0x20;
    // 5) direction PA5 input

    GPIO_PORTA_DEN_R |= 0x20;
    // 7) enable PA5 digital port
}
```

```

=GPIO_PORTA_DEN_R
    LDR R0, [R1]
    ORR R0, #0x20 ; 7) digital
    STR R0, [R1] ; on PA5
    BX LR

Switch_Input
    LDR R1, =PA5 ; 0x40004080
    LDR R0, [R1] ; read just PA5
    BX LR ; 0x20 or 0x00

Switch_Input2
    LDR R1, =
GPIO_PORTA_DATA_R
    LDR R0, [R1] ; read port
    AND R0, #0x20 ; just bit 5
    BX LR ; 0x20 or 0x00

    // return 0x20(pressed)
    // or 0(not pressed)
    uint32_t Switch_Input(void){
        return PA5;
    }

    // return 0x20(pressed)
    // or 0(not pressed)
    uint32_t Switch_Input2(void){
        return
(GPIO_PORTA_DATA_R&0x20);
    }

```

Program 4.2. Software interface for a switch on PA5 (Switch\_xxx.zip).

**Maintenance Tip:** When interacting with just some of the bits of an I/O register it is better to modify just the bits of interest, leaving the other bits unchanged. In this way, the action of one piece of software does not undo the action of another piece.

To interface an LED we connect it to a pin, see Figure 4.15, and we initialize the pin as an output. On the TM4C123, the maximum output current is 8 mA. The TM4C1294 has a maximum output current of 12 mA. If our microcontroller can output the current needed by the LED, we can use the two circuits on the left of Figure 4.16. For the positive-logic low-current configuration, assume the output voltage on PF2 is 3.2V, and the LED parameters are 1.7V 2mA (HLMP-4700). Using Ohm's Law we know the voltage drop across the resistor is the current times the resistance. The voltage drop across the resistor should be 3.2-1.7V and the current should be 2mA. So,  $(3.2-1.7)/0.002=R$ . Using this relationship we calculate the resistance  $R=(3.2-1.7)/0.002$ , which is  $750\Omega$ .

For the negative-logic low-current configuration, assume the output voltage on PF2 is 0.1V, and the LED parameters are 1.7V 2mA (HLMP-4700). Again, we know the voltage drop across the resistor is the current times the resistance,  $(3.3-1.7-0.1)/0.002=R$ . Using this relationship we calculate the resistance  $R=(3.3-1.7-0.1)/0.002$ , which is again  $750\Omega$ .

If the LED current is more than can be supplied by our microcontroller, we can use the two circuits on the right of Figure 4.16. If the input to a 7405 or 7406 is high its output will be low (0.5V). If the input to a 7405 or 7406 is low, its output will float (not driven, hiZ). If the input to a 7407 is low its output will be low (0.5V). If the input to a 7407 is high, its output will float (not driven, hiZ). If the output low voltage of the 7405/7406/7407 is 0.5V, the LED parameters are 2V 10mA (LiteOn LTL-10233-W), we can choose  $R=(5.0-2-0.5)/0.01$  which is about  $220\Omega$ . The 7406 and 7407 drivers can sink up to 40 mA.

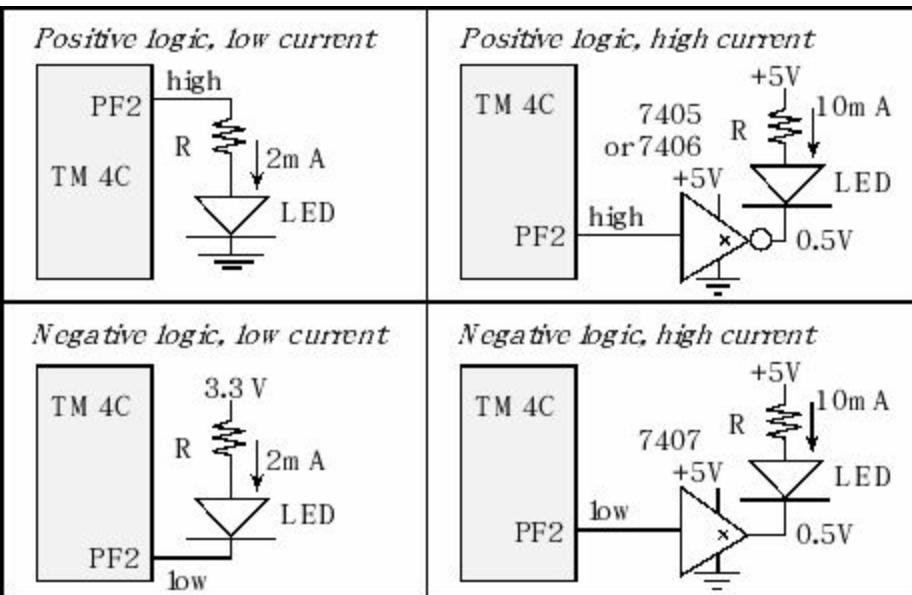


Figure 4.16. Four LED interface circuits to a microcontroller output.

The initialization in Program 4.3 will enable the clock, set the direction register bit to specify output, and enable the pin. On the TM4C123, the default setting for an output pin is 2mA maximum current. To activate 8-mA mode, we set bits in the **GPIO\_PORTF\_DR8R\_R** register. Activating 8-mA mode does not set the current at 8 mA; rather it activates the output pin so it could have a maximum of 8 mA output. Notice the software is friendly because it just affects PF2 without affecting the other bits in Port F. The output function uses the bit-specific address to output. The toggle function reads Port F, flips bit 0, and stores the result back.

```
#define PF2 (*((volatile uint32_t *)0x40025010))

PF2 EQU 0x40025010
LED_Init
    LDR R1,
    =SYSCTL_RCGCGPIO_R
    LDR R0, [R1]
    ORR R0, R0, #0x20
    STR R0, [R1] ; Port
F
    NOP ; time for clock to
finish
    NOP ; 2)
no need to unlock PF2
    LDR R1,
    =GPIO_PORTF_DIR_R
    LDR R0, [R1]
    ORR R0, #0x04 ; PF2
output
    STR R0, [R1]
    LDR R1,
    =GPIO_PORTF_DEN_R
```

```
void LED_Init(void){
    SYSCTL_RCGCGPIO_R |= 0x20;
    // 1) activate clock for Port F
    while((SYSCTL_PGPIO_R&0x20)
        == 0){}; // ready?
    // 2) no need to unlock PF2
    GPIO_PORTF_DIR_R |= 0x04;
    // 5) output on PF2
    GPIO_PORTF_DEN_R |= 0x04;
    // 7) digital enable on PF2
}
```

<b>LDR R0, [R1]</b> <b>ORR R0, #0x04 ; digital I/O</b> <b>STR R0, [R1]</b> <b>BX LR</b>	<b>void LED_Off(void){</b> <b>    PF2 = 0; // turn off LED</b> <b>}</b>
<b>LED_Off</b> <b>LDR R1, =PF2 ; R1 is 0x40025010</b> <b>MOV R0, #0</b> <b>STR R0, [R1] ; clear just PF2</b> <b>BX LR</b>	<b>void LED_On(void){</b> <b>    PF2 = 0x04; // turn on LED</b> <b>}</b>
<b>LED_On</b> <b>LDR R1, =PF2 ; R1 is 0x40025010</b> <b>MOV R0, #0x04</b> <b>STR R0, [R1] ; set just PF2</b> <b>BX LR</b>	<b>void LED_Toggle(void){</b> <b>    PF2 = PF2 ^ 0x04; // toggle LED</b> <b>}</b>
<b>LED_Toggle</b> <b>LDR R1, =PF2 ; R1 is 0x40025010</b> <b>LDR R0, [R1] ; previous value</b> <b>EOR R0, R0, #0x04 ; flip bit 2</b> <b>STR R0, [R1] ; affect just PF2</b> <b>BX LR</b>	

### Program 4.3. Software interface for an LED on PF2 (SSR\_xxx.zip).

The **LDR R0,[R1]** instruction reads Port F bit 2 into Register 0. Let  $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$  be the original values. The **EOR R0,R0,#0x04** instruction toggles bit 2, and the **STR R0,[R1]** instruction stores the result back to Port F. Because of bit-specific addressing only PF2 is affected by these functions.

$b_7 b_6 b_5 b_4 b_3 \mathbf{b}_2 b_1 b_0$  original value

0 0 0 0 0 1 0 0 **0x04** constant

$b_7 b_6 b_5 b_4 b_3 \sim \mathbf{b}_2 b_1 b_0$  result of the **EOR** instruction

### Example 4.1: Generate two out of phase square waves on PF2 and PF1.

**Solution:** Out of phase means one signal goes high when the other one goes low. During the initialization we specify PF2 and PF1 as outputs, then establish the initial values as 0 and 1 respectively. We use the **exclusive or** operation to toggle both bits at the same time. The infinite loop program will repeat the exclusive or operation over and over, creating the out of phase square waves on Port F bit 2 and 1. The other six bits of Port F remain unchanged. We create a bit-specific address constant to access just PF2 and PF1:

```
#define PF21 (*((volatile uint32_t *)0x40025018))
```

```
PF21 EQU 0x40025018
```

```
// C implementation
```

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

```
Start
```

```
// 1) Port F clock
```

```
    SYSCTL_RCGCGPIO_R |= 0x20;  
    while((SYSCTL_PRGPIO_R&0x20)  
        == 0){};// ready?
```

```
// 2) no need to unlock
```

```
// 5) PF2 PF1 outputs
```

```
    GPIO_PORTF_DIR_R |= 0x06;
```

```
// 7) digital I/O on PF2-1
```

```
    GPIO_PORTF_DEN_R |= 0x06;
```

```
// initial output
```

```
    PF21 = 0x02; // PF2=0; PF1=1
```

```
    while(1){
```

```
        PF21 ^= 0x06; // toggle
```

```
    }
```

```
}
```

```
LDR R1,
```

```
=SYSCTL_RCGCGPIO_R
```

```
    LDR R0, [R1] ;1) clock
```

```
    ORR R0, R0, #0x20 ; set bit 5
```

```
    STR R0, [R1]
```

```
    NOP ; time to finish
```

```
    NOP ; 2) no need to unlock
```

```
PF1,PF2
```

```
    LDR R1, =GPIO_PORTF_DIR_R
```

```
    LDR R0, [R1] ; 5) set direction
```

```
    ORR R0, #0x06 ; PF1,PF2 output
```

```
    STR R0, [R1]
```

```
    LDR R1, =GPIO_PORTF_DEN_R
```

```
    LDR R0, [R1] ; 7) enable Port F
```

```
    ORR R0, #0x06
```

```
    STR R0, [R1]
```

```
    LDR R1, =PF21
```

```
    MOV R0, #0x02 ; PF2=0, PF1=1
```

```
loop STR R0, [R1] ; write PF1,PF2
```

```
    EOR R0, R0, #0x06 ; toggle
```

```
PF1,PF2
```

```
B loop
```

Program 4.4. Software interface that creates two out of phase square waves (Squarewaves\_xxx.zip).

**Checkpoint 4.8:** Assume the instructions **STR EOR B loop** in Program 4.4 take 2, 1, and 3 bus cycles respectively to execute, and assume the bus frequency is 16 MHz. Estimate the frequency of each square wave in Program 4.4. See logic analyzer measurement in Figure 4.17.

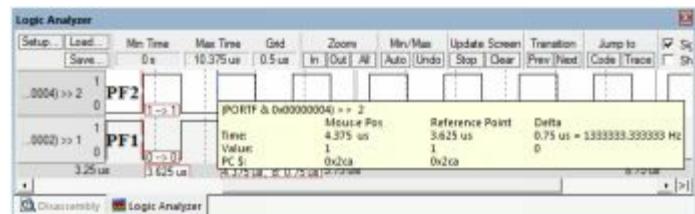


Figure 4.17. Logic analyzer in the Keil uVision simulator measures the frequency of the squarewaves.

**Example 4.2:** The goal is to develop a means for the microcontroller to turn on and to turn off an AC-powered appliance. The interface will use a solid state relay (SSR) having a control portion equivalent to an LED with parameters of 2V and 10 mA. Include appropriate functions.

**Solution:** Since we need to interface an LED, we use an open collector NOT gate just like Figure 2.9. We choose an electronic circuit that has an output current larger than the 10 mA needed by the SSR. Since the maximum  $I_{CE}$  of the PN2222 is 150 mA, it can sink the 10 mA required by the SSR. A 7405 or 7406 could also have been used, but they require a +5V supply. The resistor is selected to control the current to the diode. Using the LED design equation,  $R = (3.3 - V_d - V_{CE}) / I_d = (3.3 - 2 - 0.3V) / 0.01A = 100 \Omega$ . There is a standard value 5% resistor at 100  $\Omega$ . The specification  $V_{CE} = 0.3V$  is a maximum. If  $V_{CE}$  is actually between 0.1 and 0.3V, then 10 to 12 mA will flow, and the relay will still activate properly. When the input to the PN2222 is high ( $p=3.3V$ ), the output is low ( $q=0.3V$ ), see Figure 4.18. In this state, a 10 mA current is applied to the diode, and relay switch activates. This causes 120 VAC power to be delivered to the appliance. But, when the input is low ( $p=0$ ), the output floats ( $q=HiZ$ , which is neither high nor low). This floating output state causes the LED current to be zero, and the relay switch opens. In this case, no AC power is delivered to the appliance.

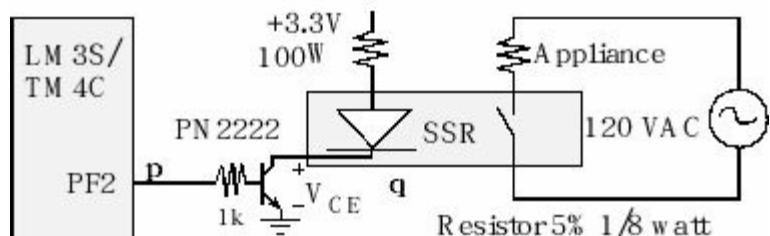


Figure 4.18. Solid state relay interface using a PN2222 NPN transistor.

The software in Program 4.3 can be used to control the device (SSR\_xxx.zip). The initialization will set bit 2 of the direction register to make PF2 an output, see **LED\_Init**. This function should be called once at the start of the system. After initialization, the **LED\_On** and **LED\_Off** functions can be called to control the appliance.

Some problems are so unique that they require the engineer to invent completely original solutions. Most of the time, however, the engineer can solve even complex problems by building the system from components that already exist. Creativity will still be required in selecting the proper components, making small changes in their behavior (tweaking), arranging them in an effective and

efficient manner, and then verifying the system satisfies both the requirements and constraints. When young engineers begin their first job, they are sometimes surprised to see that education does not stop with college graduation, but rather is a life-long activity. In fact, it is the educational goal of all engineers to continue to learn both processes (rules about how to solve problems) and products (hardware and software components). As the engineer becomes more experienced, he or she has a larger toolbox from which processes and components can be selected.

The hardest step for most new engineers is the first one: where to begin? We begin by analyzing the problem to create a set of specifications and constraints in the form of a requirements document. Next, we look for components, in the form of previously debugged solutions, which are similar to our needs. Often during the design process, additional questions or concerns arise. We at that point consult with our clients to clarify the problem. Next we rewrite the requirements document and get it reapproved by the clients.

It is often difficult to distinguish whether a parameter is a specification or a constraint. In actuality, when designing a system it often doesn't matter into which category a parameter falls, because the system must satisfy all specifications and constraints. Nevertheless, when documenting the device it is better to categorize parameters properly. Specifications generally define in a quantitative manner the overall system objectives as given to us by our customers.

Constraints, on the other hand, generally define the boundary space within which we must search for a solution to the problem. If we must use a particular component, it is often considered a constraint. In this book, we constrain most designs to include an LM3S/TM4C microcontroller. Constraints also are often defined as an inequality, such as "the cost must be less than \$50", or "the battery must last for at least one week". Specifications on the other hand are often defined as a quantitative number, and the system satisfies the requirement if the system operates within a specified tolerance of that parameter. Tolerance can be defined as a percentage error or as a range with minimum and maximum values.

The high-level design uses data flow graphs. We then combine the pieces and debug the system. As the pieces are combined we can draw a call graph to organize the parts. If new components are designed, we can use flowcharts to develop new algorithms. The more we can simulate the system, the more design possibilities we can evaluate, and the quicker we can make changes. Debugging involves both making sure it works, together with satisfying all requirements and constraints.

**Observation:** Defining realistic tolerances on specifications profoundly affects system cost.

**Checkpoint 4.9:** What are the effects of specifying a tighter tolerance (e.g., 1% when the problem asked for 5%)?

**Checkpoint 4.10:** What are the effects of specifying a looser tolerance (e.g., 10% when the problem asked for 5%)?

---

**Example 4.3:** Design an embedded system that flashes LEDs in a 0101, 0110, 1010, 1001 binary repeating pattern.

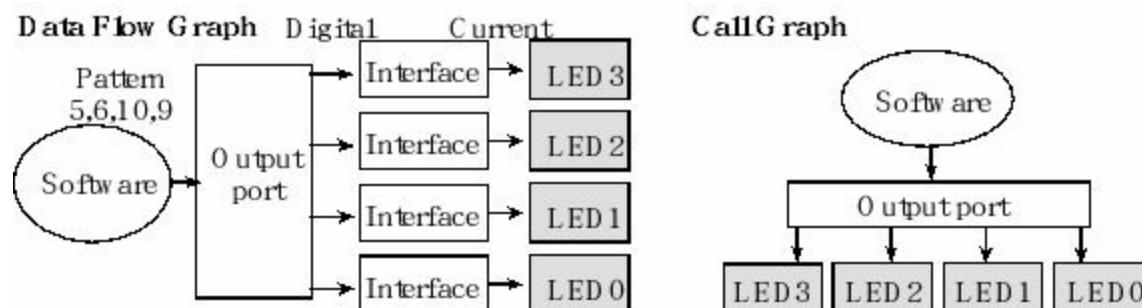
**Solution:** This system will need four LEDs, and the computer must be able to activate/deactivate them. Since the problem didn't specify power source, speed, color, or brightness, we could either put off these decisions until the engineering design stage in order to simplify the design or minimize cost, or we could go back to the clients and ask for additional requirements. In this case, the clients didn't care about power, speed, color, or brightness but did think minimizing cost was a good idea. In this book, we will constrain all our designs to include a LM3S/TM4C microcontroller. Because we have +3.3 V microcontroller systems, we will specify the system to run on +3.3 V power. We have in stock HLMP-4740 green LEDs that operate at 1.9 V and 2 mA, so we will use them. Table 4.8 summarizes the specifications and constraints. We will use standard 5% resistors to minimize cost.

Specifications		Constraints
Repeating pattern of 5, 6, 10, 9		TM4C123-based
Four 1.9 V, 2 mA green LEDs		Minimize cost
+3.3V power supply		Standard 5% resistors

**Table 4.8. Specifications and constraints of the LED output system.**

Tolerance for this LED output system says it is acceptable if it has four LEDs, but unacceptable if it has three or five of them. Similarly, it will be acceptable as long as the LED current is between 1.5 and 2.5 mA. If the current drops below 1.5 mA, we won't be able to see the LED, and if it goes above 2.5 mA, it might damage the LED.

The data flow graph in Figure 4.19 shows information as it flows from the controller software to the four LEDs. The data flow graph will be important during the subsequent design phases because the hardware blocks can be considered as a preliminary hardware block diagram of the system. The call graph, also shown in Figure 4.19, illustrates this master/slave configuration where the controller software will manipulate the four LEDs. The hardware design of this system could have used four copies of the LED interface presented earlier in Figure 2.9. The TM4C microcontroller can source or sink up to 8 mA. We can save money by using low-current LEDs, which can be connected directly to the microcontroller without a driver.



**Figure 4.19. Data flow graph and call graph of the LED output system.**

Figure 4.20 shows four simple negative logic LED interfaces. A low output will turn on the LED, and a high output will turn it off. Notice the similarity of the data flow graph in Figure 4.19 with the hardware circuit in Figure 4.20. If the  $V_{OL}$  of the microcontroller is 0.4V, and the voltage across the LED is 1.9V, then the voltage across the resistor should be  $3.3 - 1.9 - 0.4$ V or 1V. We calculate the resistor value using Ohm's Law,  $R = 1V/2mA$  or  $500\Omega$ . Using standard resistor values with a 5% tolerance will be cheaper to build (see Section 9.1). In particular,  $470\Omega$  and  $510\Omega$  are two standard resistor values near  $500\Omega$ . If we were to use  $470\Omega$ , then the LED current would be  $(3.3 - 1.9 - 0.4)V/470\Omega$  or 2.1mA. Similarly, if we were to use  $510\Omega$ , then the LED current would be  $(3.3 - 1.9 - 0.4)V/510\Omega$  or 1.96mA. Both would have been acceptable, but we will use the  $510\Omega$  resistor because it is acceptable for a wide range of microcontroller output voltages. More specifically, if  $V_{OL}$  ranges from 0.13 to 0.63V, then the LED current remains within the 1.5 to 2.5 mA specification. It would have been more expensive to design this system with  $500\Omega$  resistors.

Pseudo-code is similar to high-level languages, but without a rigid syntax. This means we utilize whatever syntax we like. Flowcharts are good when the software involves complex algorithms with many decisions points causing control paths. On the other hand, pseudo-code may be better when the software is more sequential and involves complex mathematical calculations.

**Portability** is a measure of how easy it is to convert software that runs on one machine to run on another machine. Notice how the main program does not refer to Port D at all. This way we can use this main program on a different microcontroller. C is more portable than assembly.

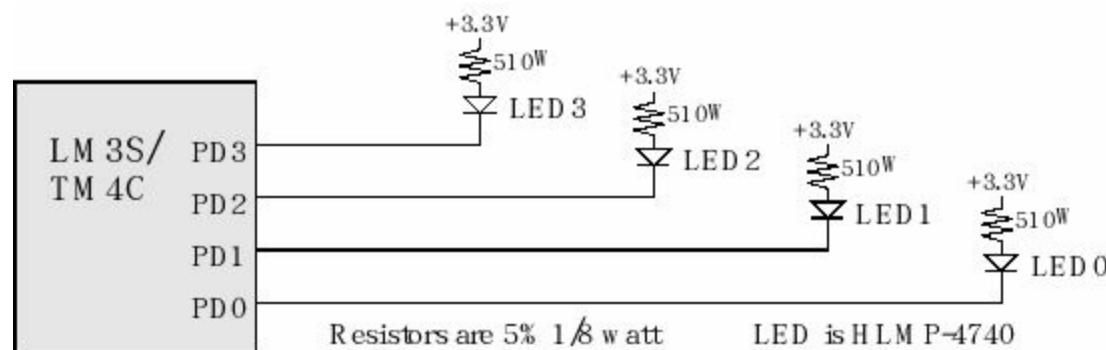


Figure 4.20. Hardware circuit for the LED output system.

The software design of this system also involves using examples presented earlier with some minor tweaking. The only data required in this problem is the 5–6–10–9 sequence. Later in Chapter 6, we will consider solutions to this type of problem using data structures, but in this first example, we will take a simple approach, not using a data structure. Figure 4.21 illustrates a software design process using flowcharts. We start with a general approach on the left. Flowchart 1 shows the software will initialize the output port and perform the output sequence. As we design the software system, we fill in the details. This design process is called successive refinement. It is also classified as top-down, because we begin with high-level issues and end at the low level. In Flowchart 2, we set the direction register and then output the sequence 5–6–10–9. It is at this stage we figured out how to create the repeating sequence. Flowchart 3 fills in the remaining details. To output the negative logic pattern 1010 to the LEDs, we will output a 5 to the bottom 4 bits of Port D on the TM4C microcontroller.

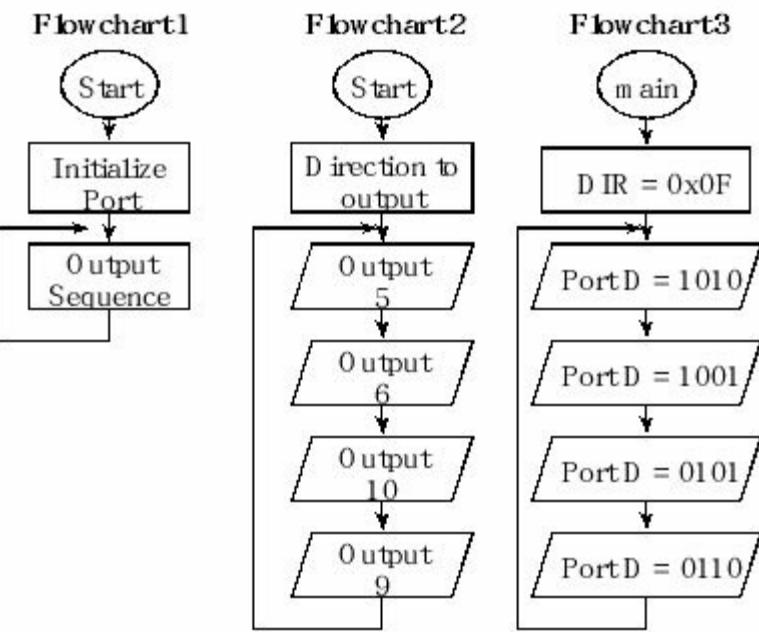


Figure 4.21. Software design for the LED output system using flowcharts.

Many software developers use pseudo-code rather than flowcharts, because the pseudo-code itself can be embedded into the software as comments. Program 4.5 shows the C implementation for this system. Notice the similarity in structure between Flowchart 3 and this code. The **LEDS** definition implements friendly access to pins PD3 – PD0.

```
#define LEDS (*((volatile uint32_t *)0x4000703C))
```

<pre> LEDS EQU 0x4000703C GPIO_Init   LDR R1,   =SYSCTL_RCGCGPIO_R   LDR R0, [R1] ; 1) D clock   ORR R0, R0, #0x00000008   STR R0, [R1]   NOP ; time to finish   NOP ; 2) no need to unlock   LDR R1,   =GPIO_PORTD_DIR_R   LDR R0, [R1] ; 5) direction   ORR R0, R0,#0xF ; PD3-0 output   STR R0, [R1]   LDR R1,   =GPIO_PORTD_DEN_R   LDR R0, [R1] </pre>	<pre> // C implementation void GPIO_Init(void){ // 1) Port D clock   SYSCTL_RCGCGPIO_R  = 0x08;  while((SYSCTL_PRGPIO_R&amp;0x08)     == 0){}; // ready?  // 2) no need to unlock PD3-0  // 5) PD3-0 outputs   GPIO_PORTD_DIR_R  = 0x0F;  // 7) digital I/O on PD3-0   GPIO_PORTD_DEN_R  = 0x0F; }  void main(void){   GPIO_Init();   while(1){ </pre>
---	--

```

ORR R0, R0, #0x0F ;
7)PD3-0 digital
STR R0, [R1]
BX LR
Start BL GPIO_Init          }
LDR R0, =LEDS ; R0 =      }
0x4000703C
    MOV R1, #10 ; R1 = 10
    MOV R2, #9  ; R2 = 9
    MOV R3, #5  ; R3 = 5
    MOV R4, #6  ; R4 = 6
loop STR R1, [R0] ; LEDS =
10
    STR R2, [R0] ; LEDS = 9
    STR R3, [R0] ; LEDS = 5
    STR R4, [R0] ; LEDS = 6
B loop

```

Program 4.5. C software for the LED output system (GPIO\_xxx.zip).

In order to test the system we need to build a prototype. One option is simulation. A second option is to use a development system like a LaunchPad. In this approach, you build the external circuits on a protoboard and use the debugger to download and test the software. A third approach is typically used after a successful evaluation with one of the previous methods. In this approach, we design a printed circuit board (PCB) including both the external circuits and the microcontroller itself.

During the testing phase of the project we observe that all four of the LEDs are continuously on. We use the software debugger to single step our program, which correctly outputs the 1010, 1001, 0101, 0110 binary repeating pattern. During this single stepping the LEDs do come on and off in the proper pattern. Using a voltmeter on the circuit we observe a 0.25V signal on the output of the microcontroller and a 1.9V voltage drop across the diode whenever the software wishes to turn the LED on. Because the LEDs are flashing faster than our eyes can see, we test the system at full speed and observe the four outputs on a logic analyzer (Analog Discovery by Digilent Inc.), collecting data presented as Figure 4.22. If we wished to be able to see the LEDs flash with our eyes, we could add a 0.1 second delay after each output.

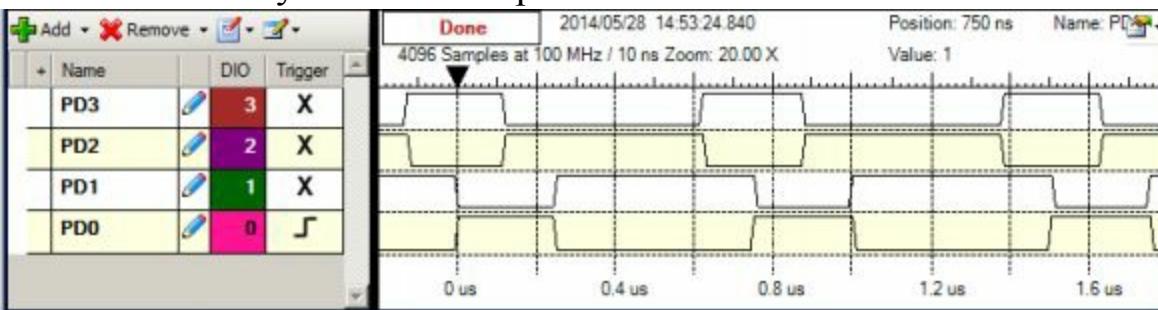


Figure 4.22. Logic analyzer waveforms collected during the testing of the LED output system.

## 4.3. Phase-Lock-Loop

Normally, the execution speed of a microcontroller is determined by an external crystal. The Stellaris EKK-LM3S1968 evaluation board has an 8 MHz crystal. The Texas Instruments Tiva EK-LM4F120XL, EK-TM4C123GXL, and EK-TM4C1294-XL boards have a 16 MHz crystal. Most microcontrollers have a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the bus clock will require less power to operate and generate less heat. Speeding up the bus clock obviously allows for more calculations per second.

The default bus speed of the LM3S1968 and TM4C microcontrollers is that of the internal oscillator, also meaning that the PLL is not initially active. For example, the default bus speed for the LM3S1968 kit is  $12\text{ MHz} \pm 30\%$ . The default bus speed for the TM4C internal oscillator is  $16\text{ MHz} \pm 1\%$ . The internal oscillator is significantly less precise than the crystal, but it requires less power and does not need an external crystal. This means for most applications we will activate the main oscillator and the PLL so we can have a stable bus clock.

There are two ways to activate the PLL. We could call a library function, or we could access the clock registers directly. In general, using library functions creates a better design because the solution will be more stable (less bugs) and will be more portable (easier to switch microcontrollers). However, the objective of the book is to present microcontroller fundamentals. Showing the direct access does illustrate some concepts of the PLL.

First, we can include the Stellaris/Tiva library and call the **SysCtlClockSet** function to change the speed. This function is defined in the **sysctl.c** file. Assume we wish to run an LM3S with an 8 MHz crystal at 50 MHz. The desired bus speed is set by the **SYSCTL\_SYSDIV\_4** parameter, which in this case will be 200 MHz divided by 4. The library function activates the PLL because of the **SYSCTL\_USE\_PLL** parameter. The main oscillator is the one with the external crystal attached. The last parameter specifies the frequency of the attached crystal.

```
SysCtlClockSet( SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
    SYSCTL_OSC_MAIN | SYSCTL_XTAL_8MHZ);
```

Assume we wish to run an LM3S microcontroller with a 6 MHz crystal at 20 MHz. The divide by 10 reduces the 200 MHz base frequency to 20 MHz.

**SysCtlClockSet( SYSCTL\_SYSDIV\_10 | SYSCTL\_USE\_PLL |  
SYSCTL\_OSC\_MAIN | SYSCTL\_XTAL\_6MHZ);**

Assume we wish to run an TM4C with a 16 MHz crystal at 80 MHz. The divide by 2.5 creates a bus frequency of 80 MHz, implemented as 400 MHz divided by 5.

**SysCtlClockSet( SYSCTL\_SYSDIV\_2\_5 | SYSCTL\_USE\_PLL |  
SYSCTL\_OSC\_MAIN | SYSCTL\_XTAL\_16MHZ);**

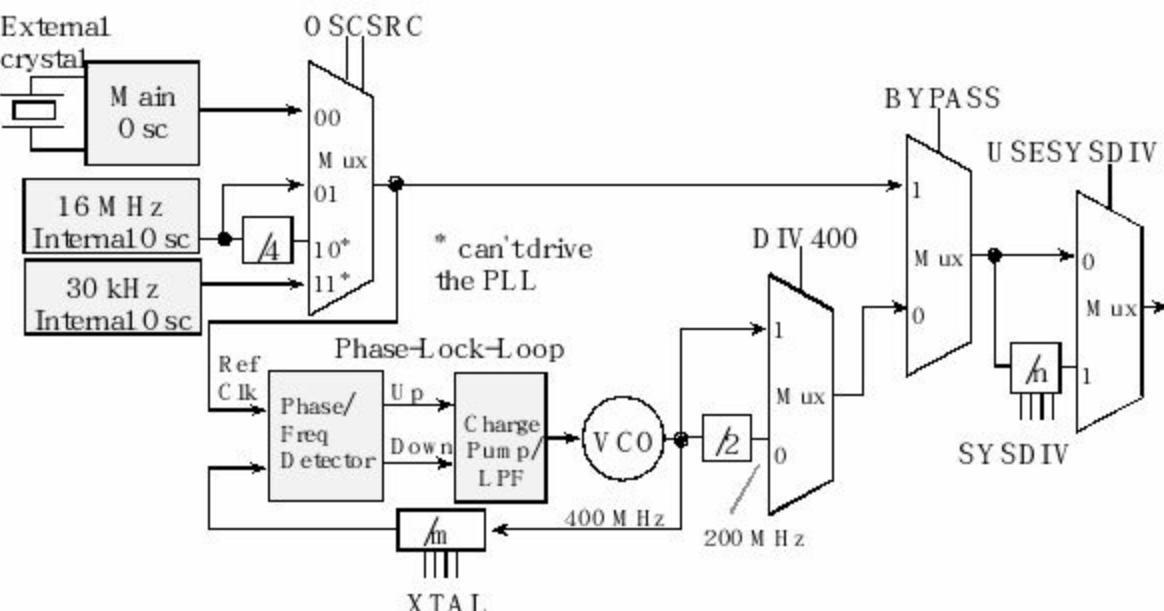


Figure 4.23. Block diagram of the main clock tree on the TM4C123 including the PLL.

To make our code more portable, it is a good idea to use library functions whenever possible. However, we will present an explicit example illustrating how the PLL works. An external crystal is attached to the TM4C microcontroller, as shown in Figure 4.23. The PLLs on the other Stellaris/Tiva microcontrollers operate in the same basic manner. Table 4.9 shows the clock registers used to define what speed the processor operates. The output of the main oscillator (Main Osc) is a clock at the same frequency as the crystal. By setting the OSCSRC bits to 0, the multiplexer control will select the main oscillator as the clock source.

Program 4.6 shows a program to activate a microcontroller with a 16 MHz main oscillator to run at 80 MHz. 0) Use RCC2 because it provides for more options. 1) The first step is set BYPASS2 (bit 11). At this point the PLL is bypassed and there is no system clock divider. 2) The second step is to specify the crystal frequency in the four XTAL bits using the code in Table 4.9. The OSCSRC2 bits are cleared to select the main oscillator as the oscillator clock source. 3) The third step is to clear PWRDN2 (bit 13) to activate the PLL. 4) The fourth step is to configure and enable the clock divider using the 7-bit SYSDIV2 field. If the 7-bit SYSDIV2 is **n**, then the clock will be divided by **n+1**. To get the desired 80 MHz from the 400 MHz PLL, we need to divide by 5. So, we place a 4 into the SYSDIV2 field. 5) The fifth step is to wait for the PLL to stabilize by waiting for PLLRIS (bit 6) in the **SYSCTL\_RIS\_R** to become high. 6) The last step is to connect the PLL by clearing the BYPASS2 bit. To modify this program to operate on other microcontrollers, you will need to change the crystal frequency and the system clock divider. Program 4.6 is on the book web site as **PLL\_xxx.zip**.

XTAL	Crystal Freq (MHz)	XTAL	Crystal Freq (MHz)
0x0	Reserved	0x10	10.0 MHz
0x1	Reserved	0x11	12.0 MHz
0x2	Reserved	0x12	12.288 MHz
0x3	Reserved	0x13	13.56 MHz
0x4	3.579545 MHz	0x14	14.31818 MHz

0x5	3.6864 MHz	0x15	16.0 MHz
0x6	4 MHz	0x16	16.384 MHz
0x7	4.096 MHz	0x17	18.0 MHz
0x8	4.9152 MHz	0x18	20.0 MHz
0x9	5 MHz	0x19	24.0 MHz
0xA	5.12 MHz	0x1A	25.0 MHz
0xB	6 MHz (reset value)	0x1B	Reserved
0xC	6.144 MHz	0x1C	Reserved
0xD	7.3728 MHz	0x1D	Reserved
0xE	8 MHz	0x1E	Reserved
0xF	8.192 MHz	0x1F	Reserved

**Table 4.9a. XTAL field used in the SYSCTL\_RCC\_R register of the TM4C123.**

Address	26-23	22	13	11	10-6	5-4	Name
\$400FE060	SYSDIV	USESYSDIV	PWRDN	BYPASS	XTAL	OSCSRC	SYSCTL_RCC_R
\$400FE050					PLLRI		SYSCTL_RIS_R
	31	30	28-22	13	11	6-4	
\$400FE070	USERCC2	DIV400	SYSDIV2	PWRDN2	BYPASS2	OSCSRC2	SYSCTL_RCC2_R

**Table 4.9b. Main clock registers for the TM4C123.**

```
#define SYSDIV2 4
void PLL_Init(void){
// 0) Use RCC2
SYSCTL_RCC2_R |= 0x80000000; // USERCC2
// 1) bypass PLL while initializing
SYSCTL_RCC2_R |= 0x00000800; // BYPASS2, PLL bypass
// 2) select the crystal value and oscillator source
SYSCTL_RCC_R = (SYSCTL_RCC_R & ~0x000007C0) // clear bits 10-6
    + 0x00000540; // 10101, configure for 16 MHz crystal
SYSCTL_RCC2_R &= ~0x00000070; // configure for main oscillator source
// 3) activate PLL by clearing PWRDN
SYSCTL_RCC2_R &= ~0x00002000;
// 4) set the desired system divider
SYSCTL_RCC2_R |= 0x40000000; // use 400 MHz PLL
SYSCTL_RCC2_R = (SYSCTL_RCC2_R & ~0x1FC00000)+(SYSDIV2<<22); // 80 MHz
// 5) wait for the PLL to lock by polling PLLRI
while((SYSCTL_RIS_R & 0x00000040)==0){}; // wait for PLLRI bit
// 6) enable use of PLL by clearing BYPASS
SYSCTL_RCC2_R &= ~0x00000800;
}
```

Program 4.6a. Activate the TM4C123 with a 16 MHz crystal to run at 80 MHz (PLL\_xxx.zip).

**PLL\_Init ; 0) configure the system to use RCC2 for advanced features**

LDR R3, =SYSCTL\_RCC2\_R ; R3 = &SYSCTL\_RCC2\_R

LDR R2, [R3]

ORR R2, R2, #0x80000000 ; USERCC2

STR R2, [R3]

; 1) bypass PLL while initializing

ORR R2, R2, #0x00000800 ; BYPASS2

STR R2, [R3]

; 2) select the crystal value and oscillator source

LDR R1, =SYSCTL\_RCC\_R ; R1 = &SYSCTL\_RCC\_R

LDR R0, [R1]

BIC R0, R0, #0x000007C0 ; clear XTAL field

ORR R0, R0, #0x00000540 ; configure for 16 MHz crystal

STR R0, [R1]

BIC R2, R2, #0x00000070 ; MOSC

; 3) activate PLL by clearing PWRDN

BIC R2, R2, #0x00002000 ; Power-Down PLL

; 4) set the desired system divider

ORR R2, R2, #0x40000000 ; use 400 MHz PLL

BIC R2, R2, #0x1FC00000 ; clear system clock divider field

ADD R2, R2, #(SYSDIV2<<22) ; SYSDIV2 = 4 (80 MHz clock)

STR R2, [R3] ; set RCC2

; 5) wait for the PLL to lock by polling PLLRIS

LDR R1, =SYSCTL\_RIS\_R ; R1 = &SYSCTL\_RIS\_R

**PLL\_Init\_loop**

LDR R0, [R1] ; R0 = [R1] (value)

ANDS R0, R0, #0x00000040 ; PLL RIS

BEQ PLL\_Init\_loop ; if(R0 == 0), keep polling

; 6) enable use of PLL by clearing BYPASS

BIC R2, R2, #0x00000800 ; BYPASS2

STR R2, [R3] ; enable PLL

Program 4.6b. Activate the TM4C123 with a 16 MHz crystal to run at 80 MHz (PLL\_xxxasm.zip).

**Checkpoint 4.11:** How would you change Program 4.6 if your microcontroller had an 8 MHz crystal and you wish to run at 50 MHz?

## 4.4. SysTick Timer

SysTick is a simple counter that we can use to create time delays and generate periodic interrupts. It exists on all Cortex -M microcontrollers, so using SysTick means the system will be easy to port to other microcontrollers. Table 4.10 shows some of the register definitions for SysTick. The basis of SysTick is a 24-bit down counter that runs at the bus clock frequency. There are four steps to initialize the SysTick timer. First, we clear the **ENABLE** bit to turn off SysTick during initialization. Second, we set the **RELOAD** register. Third, we write to the **NVIC\_ST\_CURRENT\_R** value to clear the counter. Lastly, we write the desired mode to the control register, **NVIC\_ST\_CTRL\_R**. We set the **CLK\_SRC** bit specifying the core clock will be used. We must set **CLK\_SRC=1**, because **CLK\_SRC=0** external clock mode is not implemented on the LM3S/TM4C family. In Chapter 9, we will set **INTEN** to enable interrupts, but in this first example we clear **INTEN** so interrupts will not be requested. We need to set the **ENABLE** bit so the counter will run. When the **CURRENT** value counts down from 1 to 0, the **COUNT** flag is set. On the next clock, the **CURRENT** is loaded with the **RELOAD** value. In this way, the SysTick counter (**CURRENT**) is continuously decrementing. If the **RELOAD** value is n, then the SysTick counter operates at modulo n+1 (...n, n-1, n-2 ... 1, 0, n, n-1, ...). In other words, it rolls over every n+1 counts. The **COUNT** flag could be configured to trigger an interrupt. However, in this first example interrupts will not be generated.

Address	31-	23-	16	15-3	2	1	0	Name
\$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
\$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
\$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

**Table 4.10. SysTick registers.**

If we activate the PLL to run the microcontroller at 80 MHz, then the SysTick counter decrements every 12.5 ns. In general, if the period of the core bus clock is t, then the **COUNT** flag will be set every (n+1)t. Reading the **NVIC\_ST\_CTRL\_R** control register will return the **COUNT** flag in bit 16 and then clear the flag. Also, writing any value to the **NVIC\_ST\_CURRENT\_R** register will reset the counter to zero and clear the **COUNT** flag.

Program 4.7 uses the SysTick timer to implement a time delay. For example, the user calls **SysTick\_Wait10ms(123)**; and the function returns 1.23 seconds later. The **RELOAD** register is set to the number of bus cycles one wishes to wait. If the PLL function of Program 4.6 has been executed, then the units of this delay will be 12.5 ns. Writing to **CURRENT** will clear the counter and will clear the count flag (bit 16) of the **CTRL** register. After SysTick has been decremented **delay** times, the count flag will be set and the while loop will terminate. Since SysTick is only 24 bits, the maximum time one can wait with **SysTick\_Wait** is  $2^{24} \times 12.5\text{ns}$ , which is about 200 ms. To provide for longer delays, the function **SysTick\_Wait10ms** calls the function **SysTick\_Wait** repeatedly. Notice that  $800,000 \times 12.5\text{ns}$  is 10ms. Program 4.7 is on the book web site as **SysTick\_xxx.zip**.

```
void SysTick_Init(void){
```

```

NVIC_ST_CTRL_R = 0;           // 1) disable SysTick during setup
NVIC_ST_RELOAD_R = 0x00FFFFFF; // 2) maximum reload value
NVIC_ST_CURRENT_R = 0;        // 3) any write to current clears it
NVIC_ST_CTRL_R = 0x00000005;  // 4) enable SysTick with core clock
}
// The delay parameter is in units of the 80 MHz core clock. (12.5 ns)
void SysTick_Wait(uint32_t delay){
    NVIC_ST_RELOAD_R = delay-1; // number of counts to wait
    NVIC_ST_CURRENT_R = 0;     // any value written to CURRENT clears
    while((NVIC_ST_CTRL_R&0x00010000)==0){ // wait for count flag
    }
} // 10000us equals 10ms
void SysTick_Wait10ms(uint32_t delay){
    uint32_t i;
    for(i=0; i<delay; i++){
        SysTick_Wait(800000); // wait 10ms
    }
}

```

Program 4.7a. Timer functions that implement a time delay (SysTick\_xxx.zip).

Checkpoint 4.12: How would you change `SysTick_Wait10ms` in Program 4.7 if your microcontroller were running at 12 MHz?

### SysTick\_Init

```

LDR R1, =NVIC_ST_CTRL_R
MOV R0, #0                 ; disable SysTick during setup
STR R0, [R1]
LDR R1, =NVIC_ST_RELOAD_R   ; R1 = &NVIC_ST_RELOAD_R
LDR R0, =0x00FFFFFF;       ; maximum reload value
STR R0, [R1]                ; [R1] = R0 = NVIC_ST_RELOAD_M
LDR R1, =NVIC_ST_CURRENT_R ; R1 = &NVIC_ST_CURRENT_R
MOV R0, #0                  ; any write to current clears it
STR R0, [R1]                ; clear counter
LDR R1, =NVIC_ST_CTRL_R    ; enable SysTick with core clock
MOV R0, #0x05
STR R0, [R1]                ; ENABLE and CLK_SRC bits set
BX LR

```

-----SysTick\_Wait-----

```

; Time delay using busy wait.
; Input: R0 delay parameter in units of the core clock (20 nsec)
; Output: none ; Modifies: R0, R1, R3
SysTick_Wait

```

```

    LDR R1, =NVIC_ST_RELOAD_R ; R1 = &NVIC_ST_RELOAD_R

```

```

SUB R0, #1
STR R0, [R1]          ; delay-1, number of counts to wait
LDR R1, =NVIC_ST_CTRL_R ; R1 = &NVIC_ST_CTRL_R
SysTick_Wait_loop
    LDR R3, [R1]          ; R3 = NVIC_ST_CTRL_R
    ANDS R3, R3, #0x00010000 ; Count set?
    BEQ SysTick_Wait_loop
    BX LR

;-----SysTick_Wait10ms-----
; Time delay using busy wait. This assumes 50 MHz clock
; Input: R0 number of times to wait 10 ms before returning
; Output: none           ; Modifies: R0
DELAY10MS EQU 500000 ; clock cycles in 10 ms (assumes 50 MHz clock)
SysTick_Wait10ms
    PUSH {R4, LR}          ; save current value of R4 and LR
    MOVS R4, R0              ; R4 = R0 = remainingWaits
    BEQ SysTick_Wait10ms_done ; R4 == 0, done
SysTick_Wait10ms_loop
    LDR R0, =DELAY10MS      ; R0 = DELAY10MS
    BL SysTick_Wait         ; wait 10 ms
    SUBS R4, R4, #1          ; R4 = R4 - 1; remainingWaits--
    BHI SysTick_Wait10ms_loop ; if(R4 > 0), wait another 10 ms
SysTick_Wait10ms_done
    POP {R4, PC}

```

Program 4.7b. Timer functions that implement a time delay (SysTick\_xxxasm.zip).

**Example 4.4.** Design a system with four outputs, making them 5, 6, 10, 9 over and over separated by 5 ms.

**Solution:** There are no inputs to this system. However, the four outputs will be generated using port pins. We could have used any port, but in this example we will use PD3-0. This example illustrates how development time is reduced by reusing previously developed code. In this example we need software that implements a 5-ms delay. Rather than starting over from scratch, we will reuse the timer subroutines previously developed in Program 4.7. Notice, when we think about **SysTick\_Wait** in Program 4.7 we can focus on what it does (takes a parameter in Register R0 and waits that many cycles), rather than worry about how it works. During initialization, we set the direction register to make PD3-0 outputs and enable the timer. After each output, the system waits 5 ms by calling **SysTick\_Wait** with Register R0 equal to 400,000. We create a bit-specific definition for the 4 pins PD3-PD0:

```
#define OUTPUTS (*((volatile uint32_t *)0x4000703C))
```

OUTPUTS EQU 0x4000703C ; PD3-0	#define DELAY 400000 // 5ms
--------------------------------	-----------------------------

```

DELAY EQU 400000 ; 5ms
Start BL PLL_Init ; 80 MHz, Prog 4.6
BL SysTick_Init ; init
LDR R1, = SYSCTL_RCGCGPIO_R
LDR R0, [R1]
ORR R0, R0, #0x08
STR R0, [R1] ; 1) Port D clock
NOP
NOP ; 2) no need to unlock
LDR R1, =GPIO_PORTD_DIR_R
LDR R0, [R1] ; 5) direction
ORR R0, R0, #0x0F ; PD3-0 output
STR R0, [R1]
LDR R1, =GPIO_PORTD_DEN_R
LDR R0, [R1] ; 7) enable
ORR R0, R0, #0x0F ; digital I/O
STR R0, [R1]
LDR R0, =DELAY ; 5ms
LDR R4, =OUTPUTS ; R4=0x4000703C
loop MOV R1, #0x05
STR R1, [R4] ; out 0xA
BL SysTick_Wait ; wait
MOV R1, #0x06
STR R1, [R4] ; out 0x09
BL SysTick_Wait ; wait
MOV R1, #0x0A
STR R1, [R4] ; out 0x05
BL SysTick_Wait ; wait
MOV R1, #0x09
STR R1, [R4] ; out 0x06
BL SysTick_Wait ; wait
B loop

```

```

void main(void){
    PLL_Init(); // 80 MHz, Prog 4.6
    SysTick_Init(); // Program 4.7
    // 1) activate clock for Port D
    SYSCTL_RCGCGPIO_R |= 0x08;
    while((SYSCTL_PGPIO_R&0x20)
        == 0){};// ready?

    // 2) no need to unlock PD3-0

    // 5) set direction register
    GPIO_PORTD_DIR_R |= 0x0F;

    // 7) enable digital port
    GPIO_PORTD_DEN_R |= 0x0F;

    while(1){
        OUTPUTS = 0x05;
        SysTick_Wait(Delay); // Prog 4.7

        OUTPUTS = 0x06;
        SysTick_Wait(Delay);

        OUTPUTS = 0x0A;
        SysTick_Wait(Delay);

        OUTPUTS = 0x09;
        SysTick_Wait(Delay);
    }
}

```

## Program 4.8. A system to output a four-bit pattern.

**Observation:** If the PD3-0 outputs of Example 4.4 were connected to a stepper motor as shown in Figure 8.25, this software would cause the motor to spin at a constant rate.

Switches have mass, friction, and a spring. Depending on the magnitudes of these three properties the switch will bounce (oscillate a few times) or not bounce when pressed or released. If it does occur, the contact bounce is usually on the order of 1 ms. One way to remove the bounce is to wait at least 10 ms between readings of the switch. The flowcharts in Figure 4.24 show algorithms that wait until a switch is pressed. The one on the left does not consider bounce, while the one in the middle will remove bounce. The flowchart on the right uses the wait function to count the number of times a switch is pressed.

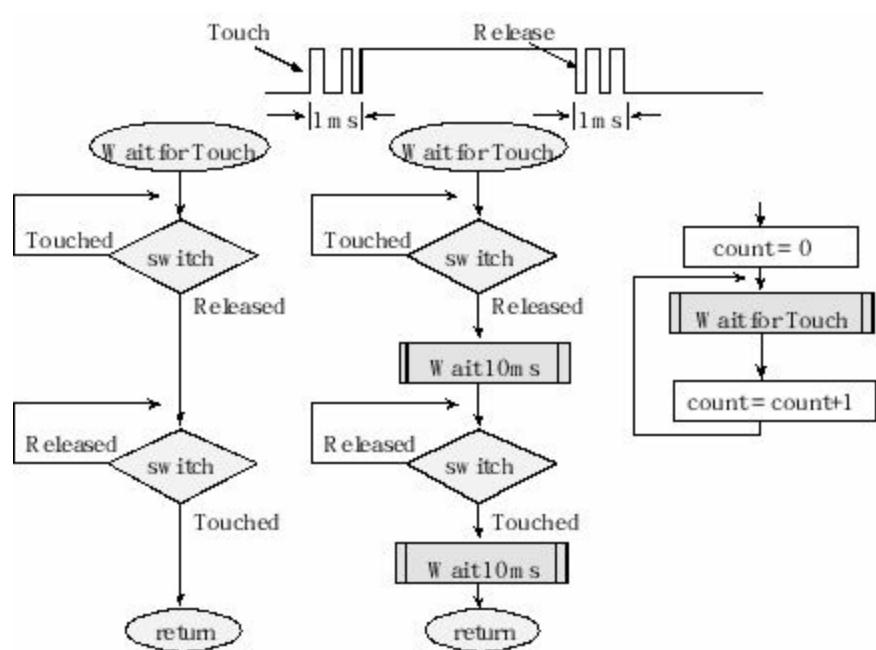


Figure 4.24. Flowcharts illustrating how to debounce a switch.

# 4.5. Standard I/O Driver and the printf Function

A very powerful approach to I/O is to provide a high-level abstraction in such a way that the I/O device itself is hidden from the user. There are three printf projects on the book's web site. The overall purpose of each of these examples is to provide an output stream using the standard **printf** function. Using the project **Printf\_UART\_xxx.zip**, we send the output data stream through UART0 to the PC. The project **Printf\_Nokia\_xxx.zip** sends data through SSI0 to a low cost Nokia LCD. The project **Printf\_ST7735\_xxx.zip** implements a similar approach sending data through SSI0 to a color LCD ST7735. In each implementation, there is an **Output\_Init()** function that initializes the output device, and a general function **printf()** we use to output data in a standard way.

At the low level, we implement how the output actually happens by writing a **fputc** function. The **fputc** function is a private and implemented inside the **UART.c**, **Nokia5110.c** or **ST7735.c** file. It sends characters to the display and manages the line feed and carriage return functionalities. There is another function called **getc** to handle standard input. The UART will implement input, but the two LCD implementations will not have input capabilities.

At the high level, the user performs output by calling **printf**. This abstraction clearly separates what it does (**printf** outputs information) from how it works (**fputc** sends data to the display over UART or SSI). In these three examples all output is sent to a display; however, we could modify the **fputc** function and redirect the output stream to other devices such as a parallel LCD, USB, Ethernet, wireless link, or solid-state disk.

**printf** is a function from the standard input and output library that allows you to display messages. To make the standard input and output library available to your program, you must include one of these **UART.c**, **Nokia5110.c** or **ST7735.c** files in your project, and add these lines, replacing **UART.h** with **Nokia5110.h** or **ST7735.h** as appropriate:

```
#include <stdio.h>
#include "UART.h"
```

You will need to initialize the display by executing this line at the beginning of your program:

```
Output_Init(); // initialize the output display
```

The call to **printf** has a string parameter followed by a list of values to display. Assume **cc** is an 8-bit variable containing 0x56 ('V'), **xx** is a 32-bit variable containing 100, and **yy** is a 16-bit variable containing -100, **zz** is a 32-bit floating containing 3.14159265. The following illustrate the use of **printf**. After the format parameter, **printf** requires at least as many additional arguments as specified in the format.

Example code	Output
<b>printf("Hello world\n");</b>	<b>Hello world</b>

<code>printf("cc = %c %d %#x\n",cc,cc,cc);</code>	<b>cc = V 86 0x56</b>
<code>printf("xx = %c %d %#x\n",xx,xx,xx);</code>	<b>xx = d 100 0x64</b>
<code>printf("yy = %d %#x\n",yy,yy);</code>	<b>yy = -100 0xffffffff9c</b>
<code>printf("zz = %f %3.2f\n",zz,zz);</code>	<b>zz = 3.141593 3.14</b>

Escape sequences are used to display non-printing and hard-to-print characters. In general, these characters control how text is positioned on the screen, for example, newlines and tabs, see Table 4.11.

Character	Value	Escape Sequence
alert (beep)	<b>0x07</b>	\a
backslash	<b>0x5C</b>	\\\
backspace	<b>0x08</b>	\b
carriage return	<b>0x0D</b>	\r
double quote	<b>0x22</b>	\"
form feed	<b>0x0C</b>	\f
horizontal tab	<b>0x08</b>	\t
newline	<b>0x0A</b>	\n
null character	<b>0x00</b>	\0
single quote	<b>0x27</b>	\'
STX	<b>0x02</b>	\x02 (this syntax works for any 2-digit hex value)
vertical tab	<b>0x0B</b>	\v
question mark	<b>0x3F</b>	\?

**Table 4.11. Escape sequences.**

When the program is executed, the control string will be displayed exactly as it appears in the program with two exceptions. First, the computer will replace each conversion specification with a value given in the other arguments part of the `printf` statement. Second, escape sequences will be replaced with special non-printing and hard-to-print characters. To display the contents of a variable we add a % tag into the format string the specifier defines the type as listed in Table 4.12. The floating-point specifiers have been omitted.

**%[flags][width].[precision]specifier**

Specifier	Output	Example
<b>c</b>	Character	a
<b>d or i</b>	Signed decimal integer	392
<b>ld</b>		1234567890

	Signed 32-bit long decimal integer	
e	Scientific notation	6.022141e23
E	Scientific notation, capital letter	6.022141E23
f	Floating point	3.14159
o	Unsigned octal	610
s	String of characters	sample
u	Unsigned decimal integer	7235
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (capital letters)	7FA
%	%% will write % to stdout	%

**Table 4.12. Format specifiers.**

The tag can also contain flags, width, .precision, and length sub-specifiers. The **flags** are listed in Table 4.13. If the **width** is present, it specifies the minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger. The **.precision** sub-specifier specifies the minimum number of digits to be written (d, i, o, u, x, X). If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated if the result requires more digits. A precision of 0 means that no character is written for the value 0. For s the **.precision** is the maximum number of characters to be printed. For c type is **.precision** has no effect. For floating point **.precision** is the number of digits after the decimal.

Flags	Description
-	Left-justify within the given field width
+	Forces the result to have a plus or minus sign
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

**Table 4.13. Flag sub-specifiers.**

If successful, **printf** will return the total number of characters written. On failure, a negative number is returned. The start of a format specifier is signified by a percent sign and the end is signified by one of the letter codes in Table 4.12. Each format specifier will be replaced by a value from the argument list converted to the specified format. These optional fields typical occur in this order

The pound sign ( '#' ) specifies that the value should be converted to an alternate form. The alternate form for hexadecimal adds the 0x or 0X. The alternate form for octal is a leading zero.

```
printf("%ox", 11); // prints just 'b'
```

```
printf("%#x", 11); // prints '0xb'  
printf("%X", 11); // prints just 'B'  
printf("%#X", 11); // prints '0XB'  
printf("%o", 11); // prints just '13'  
printf("%#o", 11); // prints '013'
```

The zero( '0' ) specifies zero-padding. The converted value is padded on the left with the specified number of zeros minus the number of digits to be printed. This is described in more detail below.

```
printf("%d", 9); // prints '9'  
printf("%4d", 9); // prints ' 9'  
printf("%04d", 9); // prints '0009'  
printf("%04d", 123); // prints '0123'
```

A minussign ( '-' ) specifies left justification. Without the minus, the format is right justified.

```
printf("%5d", 12); // prints ' 12' (right justified)  
printf("%-5d", 12); // prints '12 ' (left justified)
```

A space( ' ' ) specifies that a blank should be left before a positive number.

```
printf("% d", 9); // prints ' 9'  
printf("% d", -9); // prints '-9'
```

The plus sign( '+' ) specifies that a sign always be placed before the value. The plus sign overrides a space if both are used.

```
printf("%+d", 9); // prints '+9'  
printf("%+d", -9); // prints '-9'
```

A decimal digit specifies the minimum field width. Using the minus sign makes the format is left justified, otherwise it is right justified. Used with the zero-modifier for numeric conversions, the value is right-padded with zeros to fill the field width.

```
printf("%3d", 12); // prints ' 12' (right justified)  
printf("%-3d", 12); // prints '12 ' (left justified) printf("%3d", 123); // prints '123'  
(filled up)  
printf("%3d", 1234); // prints '1234' (bigger than 3 width)
```

A precision value in the form of a period ( '.' ), followed by an optional digit string. If the digit string is omitted, a precision of zero is used. When used with decimal, hexadecimal or octal integers, it specifies the minimum number of digits to print. For floating point output, it specifies the number of digits after the decimal place. For the 's' (string) conversion, it specifies the maximum number of characters of the string to print, which is quite useful to make sure long strings don't exceed their field width.

```
printf("%.3d", 7); // prints '007'  
printf("%.3d", 12345); // prints '12345'
```

```

printf("%3s", "Jonathan"); // prints 'Jonathan'
printf("%.3s", "Jonathan"); // prints 'Jon'
printf("%3s", "JV");      // prints 'JV '
printf("%.3s", "JV");      // prints 'JV'

```

Consider a decimal fixed-point number with units 0.001 cm. For example, if the value of **distance** is equal to 1234, this means the distance is 1.234 cm. Assume the distance varies from 0 to 99.999 cm. This C code could be used to print the value of the number in such a way that exactly 20 characters are printed for all values of **distance** from 0 to 99999. The first format specifier ( **%2d** ) prints the integer part in exactly two characters, and the second format specifier ( **.3d** ) prints the fractional part in exactly three characters.

```
printf("Distance = %2d.%3d cm", distance/1000,distance%1000);
```

<u>Value</u>	<u>Output</u>
0	<b>Distance = 0.000 cm</b>
1	<b>Distance = 0.001 cm</b>
99	<b>Distance = 0.099 cm</b>
123	<b>Distance = 0.123 cm</b>
1234	<b>Distance = 1.234 cm</b>
12345	<b>Distance = 12.345 cm</b>

Program 4.9 demonstrates the use of input/output using the standard library. The low-level UART includes both standard input and standard output. The Nokia and ST7735 examples only include standard output. The **scanf** function will wait for user input and return with a value as specified by the type. In this case it returns with a signed decimal number in the variable **side** . The body of the main program,

- 1) Asks for input,
- 2) Waits for input,
- 3) Performs a calculation, and
- 4) Displays the results.

```

//**** 0. Documentation Section
// This program calculates the area of square shaped rooms
// Author: Ramesh Yerraballi & Jon Valvano
// Date: 5/24/2014
//
// 1. Pre-processor Directives Section
#include <stdio.h> // Diamond braces for sys lib: Standard I/O
#include <stdint.h> // C99 variable types
void Output_Init(void);

// 2. Global Declarations section
int32_t side; // room wall meters

```

```
int32_t area; // size squared meters
// Function Prototypes

// 3. Subroutines Section
// MAIN: Mandatory routine for a C program to be executable
int main(void) {
    Output_Init();           // initialize output device
    printf("This program calculates areas of square-shaped rooms\n");
    while(1){
        printf("Give room side:"); // 1) ask for input
        scanf("%ld", &side);     // 2) wait for input
        area = side*side;        // 3) calculation
        printf("\nside = %ld m, area = %ld sqr m\n", side, area); // 4) out
    }
}
```

Program 4.9. Software to calculate the area of a square room  
(Scanf\_UART\_xxx.zip).

## 4.6. Debugging monitor using an LED

One of the important tasks in debugging a system is to observe when and where our software is executing. A debugging tool that works well for real-time systems is the monitor. In a real-time system, we need the execution time of the debugging tool to be small compared to the execution time of the program itself. **Intrusiveness** is defined as the degree to which the debugging code itself alters the performance of the system being tested. A monitor is an independent output process, somewhat similar to the print statement, but one that executes much faster and thus is much less intrusive. An LED attached to an output port of the microcontroller is an example of a BOOLEAN monitor. You can place LEDs on unused output pins. Software toggles these LEDs to let you know where and when your program is running. Assume an LED is attached to Port F bit 2. Program 4.10 will toggle the LED.

<pre>TogglePF2 LDR R1, = GPIO_PORTF_DATA_R LDR R0, [R1] EOR R0, R0, #0x04 STR R0, [R1] BX LR</pre>	<pre>void TogglePF2(void){     GPIO_PORTF_DATA_R ^= 0x04; // toggle LED }</pre>
--	---

Program 4.10. An LED monitor using regular port access.

Program 4.11 will also toggle the LED on PF2, but this version using bit-specific addressing to access just PF2.

<pre>PF2 EQU 0x40025010 TogglePF2 LDR R1, =PF2 LDR R0, [R1] EOR R0, R0, #0x04 STR R0, [R1] BX LR</pre>	<pre>#define PF2 ((volatile uint32_t *)0x40025010)) void TogglePF2(void){     PF2 ^= 0x04; // toggle LED }</pre>
--	--

Program 4.11. An LED monitor using bit-specific addressing to access the port (TM4C123).

A **heartbeat** is a pulsing output that is not required for the correct operation of the system, but it is useful to see while the program is running. In particular, you add **BL TogglePF2** statements at strategic places within your system. It only takes 13 bus cycles to execute. Port F must be initialized so that bit 2 is an output before the debugging begins. You can either observe the LED directly or look at the LED control signals with a high-speed oscilloscope or logic analyzer. The LCD, which will be explained in Section 7.8, can be an effective monitor for small amounts of information. The Nokia 5110 LCD cost less than \$10 and can display 72 characters. Unfortunately, the Nokia 5110 requires about 20  $\mu$ s to output each character, so the use of an LCD monitor might be intrusive. When using LED monitors it is better to modify just the one bit, leaving the other 7 as is. In this way, you can have additional LED monitors.

# 4.7. Performance Debugging

Performance debugging involves the verification of the timing behavior of our system. Performance debugging is a dynamic process where the system is run and the dynamic behavior of the input/outputs is compared against the expected results. Two methods of are presented, and then the techniques are applied to measure execution speed.

## 4.7.1. Instrumentation

**Instrumentation** is software code added for the purpose of debugging. SysTick is a 24-bit counter, decremented at the bus clock frequency. It automatically rolls over when it gets to 0. If we are sure the execution speed of our function is less than ( $2^{24}$  bus cycles), we can use this timer to collect timing information with only a modest amount of intrusiveness. We initialize the timer by calling **SysTick\_Init** as shown in Program 4.7. Keep in mind the fact that although the registers and variables are 32 bits, but the data values are only 24 bits. To perform a measurement we first read **NVIC\_ST\_CURRENT\_R**, execute some software, and then read **NVIC\_ST\_CURRENT\_R** again. The difference represents the elapsed time for the executing software. In this example **before** and **elapsed** are 32-bit unsigned variables:

```
before = NVIC_ST_CURRENT_R;  
// software we wish to test  
elapsed = (before-NVIC_ST_CURRENT_R)&0x00FFFFFF;
```

Another method to observe time-dependent behavior of our software involves an output port and a logic analyzer or oscilloscope. Assume an oscilloscope is attached to Port F bit 2. The two subroutines in Program 4.3 can be used to set and clear the bit. Next, you add calls to **LED\_On** and **LED\_Off** statements at strategic places within the system. Port F must be initialized so that bit 2 is an output( **LED\_Init** ) before the debugging begins. You can observe the signal with a high-speed oscilloscope.

```
LED_On();  
// software we wish to test  
LED_Off();
```

## 4.7.2. Measurement of Dynamic Efficiency

We will present three ways to measure dynamic efficiency of our software. To illustrate these three methods, we will consider measuring the execution time of the **sqrt** function. The first method is to count bus cycles using the assembly listing. This approach is only appropriate for very short programs, and it becomes difficult for long programs with many conditional branch instructions. This method is very tedious and often only approximate. A portion of the assembly output is presented in Program 4.12. The Cortex™-M Technical Reference Manual Section 3.3 lists the execution times for

each instruction. These times are listed in the assembly version of Program 4.12. The “16\*” calculations are because the loop is executed 16 times. The divide instruction lists an execution time from 2 to 12 cycles, depending on the data. The branch instructions may or may not have to refill the pipeline, so their execution varies. The total cycle count for the **sqrt** function ranges from 150 to 344 cycles. At 50 MHz, 150 to 344 cycles ranges from 3 to 6.88  $\mu$ s. For most programs it is actually very difficult to get an accurate time measurement using this technique.

<pre>; Input: R0 unsigned integer ; Output: R0 squareroot of input sqrt MOV r1,r0      ;1     MOVS r3,#0x01      ;1     ADD r0,r3,r1,LSR #4 ;1     MOVS r2,#0x10      ;1 loop MLA r3,r0,r0,r1 ;16*2     UDIV r3,r3,r0      ;16*(2to12)     LSRS r0,r3,#1      ;16*1     SUBS r2,r2,#1      ;16*1     CMP r2,#0x00      ;16*1     BNE loop          ;16*(2to4)     BX lr             ;2to4</pre>	<pre>// Newton's method // s is an integer // sqrt(s) is an integer uint32_t sqrt(uint32_t s){     uint32_t t; // t*t will become s     int n;      // loop counter     t = s/16+1; // initial guess     for(n = 16; n; --n){ // will finish         t = ((t*t+s)/t)/2;     }     return t; }</pre>
---	---

Program 4.12. Integer sqrt function (assembly language version shows number of cycles).

The second method uses the SysTick functions defined in Program 4.7. Since the execution speed may be dependent on the input data, it is often wise to measure the execution speed for a wide range of input parameters. There is a slight overhead in the measurement process itself. To be more accurate you could measure this overhead and subtract it off your measurements.

There is an optimization level for the ARM Keil™ uVision® compiler (0, 1, 2, or 3) and a selector button labeled “Optimize for time”. Measurements were performed with inputs 100 and 230400. The assembly version took 201 to 220 cycles. The C program at optimization level 3 and “optimize for time” took 178 to 198 cycles. Interestingly, the **sqrt** routine in “math.h” took about 1840 cycles to execute.

```
void main(void){ uint32_t before,elapsed,ss,tt;
SysTick_Init(); // initialize, Program 4.7
ss = 230400;
before = NVIC_ST_CURRENT_R;
tt = sqrt(ss);
elapsed = (before-NVIC_ST_CURRENT_R)&0x00FFFFFF;
while(1){};
}
```

The third technique can be used in situations where SysTick is unavailable or where the execution time might be larger than  $2^{24}$  cycles. In this empirical technique we attach an unused output pin to an oscilloscope or to a logic analyzer. We will set the pin high before the call to the function and set the pin low after the function call. In this way a pulse is created on the digital output with duration equal to the execution time of the function. Assume Port F bit 2 is available and connected to the scope. By placing the function call in a loop, the scope can be triggered. With a storage scope or logic analyzer, the function need be called only once.

```
void main(void){ uint32_t tt;
    LED_Init(); // initialize PF2 (Program 4.3)
    while(1){
        LED_On(); // see Figure 4.25
        tt = sqrt(230400);
        LED_Off();
    }
}
```

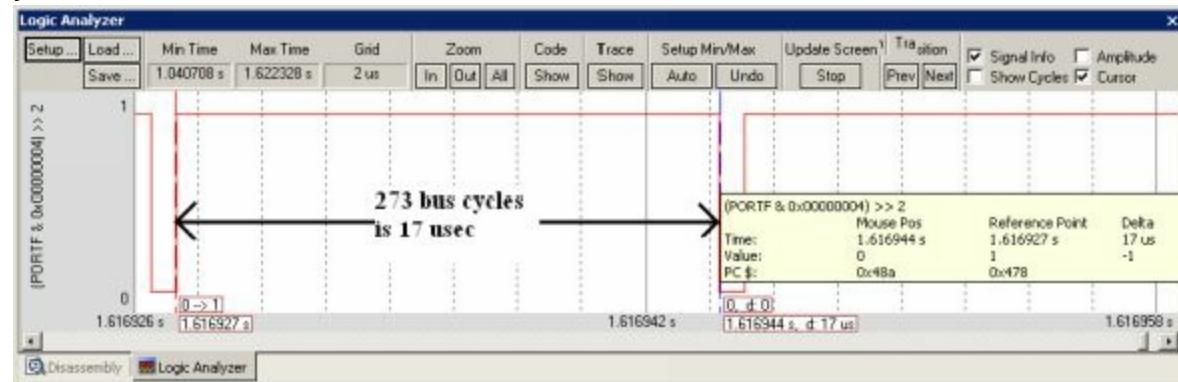


Figure 4.25. Measurement of execution time using a port pin and a logic analyzer (ProfileSqrt\_4F120).

# 4.8. Exercises

- 4.1 What parallel ports are available on the TM4C123?
- 4.2 What parallel ports are available on the TM4C1294?
- 4.3 What is a direction register? Why does the microcontroller have direction registers?
- 4.4 What is the alternative function register?
- 4.5 Write software that initializes TM4C Port A, so pins 7,5,3,1 are output and the rest are input.
- 4.6 Write software that initializes TM4C Port A, so pins 5,4 are output and the rest are input.
- 4.7 Write software that initializes TM4C Port C in a friendly manner, so pins 2 and 3 are output.
- 4.8 Fill in the following table with the bit-specific assembly definitions (the first one is done)

Port	Bits	Definition (using AHB)
A	5	<b>PA5 EQU 0x40004080</b>
B	1,0	
F	7,5	
G	3,2,0	

- 4.9 Fill in the following table with the bit-specific C definitions (the first one is done)

Port	Bits	Definition (using AHB)
A	5	<b>#define PA5 ((volatile uint32_t *)0x40004080))</b>
C	3,1	
D	7,6,5	
F	7,0	

- 4.10 Write software that initializes TM4C Port A, so pins 5, 4, and 3 are output. Make the initialization friendly. Design an output function that takes a 3-bit parameter (0 to 7) and writes the value to these three pins. Use bit-specific addressing for the output.

- 4.11 Write software that initializes TM4C Port E, so pin 1 is an output. Make the initialization friendly. Design an output function that takes a 1-bit parameter (0 or 1) and writes the value to this pin. Use bit-specific addressing for the output.

- 4.12 Redesign the SSR interface shown in Figure 4.18 using a +5V source. In particular, recalculate the required resistor value if we were to change the +3.3V to +5V.

- 4.13 Redesign the LED interface shown in Figure 4.20 if the four LEDs operated at 1.9 V and 1 mA. In particular, recalculate the required resistor values if we were to change to these LEDs.

- 4.14 Rewrite the software in Program 4.5 so the LED pattern changes every 0.1 sec.

**4.15** Design a negative logic switch interface on PA5. I.e., the input is low if the switch is pressed and high if the switch is not pressed.

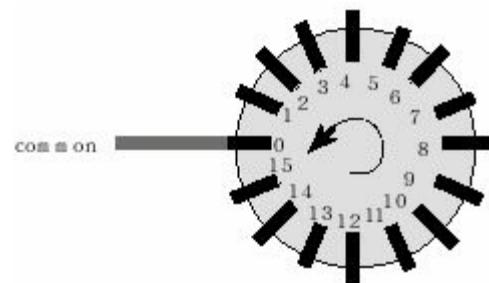
**4.16** Using the software in Program 4.11, write a function that delays 1 second.

**D4.17** Show the circuit diagram to interface a switch to PA6 and an LED to PC3. Write software that initializes the ports. In the body of the main program, toggle the LED on and off every one second if the switch is pressed, and turn the LED off if the switch is not pressed.

**D4.18** Show the circuit diagram to interface two switches to PA1 and PA0 and one LED to PD0. Write software that initializes the ports. In the body of the main program, toggle the LED every 1 second if the two switches are either both on or both off. Turn the LED off if the PA1 switch is pressed and the PA0 switch is not pressed. Turn the LED on if the PA0 switch is pressed and the PA1 switch is not pressed.

**D4.19** Show the circuit diagram to interface one switch to PB0 and four LEDs to PA3-PA0. The four LEDs will display a number from 0 to 15 in binary. Write software that initializes the ports. In the body of the main program, increment the number every time the switch is pressed and released. Wait at least 10 ms in between reading the switch. Once the number gets to 15, do not increment it any more.

**D4.20** Show the circuit diagram to interface four switches to PA3-PA0 and four LEDs to PE3-PE0. Switches PA3-PA2 represent a 2-bit unsigned number 0, 1, 2, or 3. Switches PA1-PA0 represent a second 2-bit unsigned number 0, 1, 2, or 3. The four LEDs will display a number from 0 to 15 in binary. Write software that initializes the ports. In the body of the main program, read the switches, form the two numbers, multiply the numbers together, and output the result on the LEDs.



**D4.21** You are given a 16-position rotary switch, which has 17 wires. There is one wire called common, and the other 16 wires are labeled S0 through S15. The common wire is connected to exactly one of other 16 wires. You are to design an interface that creates a 4-bit digital signal representing the switch position. These signals are to be connected to Port D bits 3,2,1,0. Write an initialization ritual. Write an input subroutine that reads Port D and returns in Register R0 the current switch position 0 to 15.

# 4.9. Lab Assignments

The labs in this book involve the following steps:

Part a) During the analysis phase of the project determine additional specifications and constraints. In particular, discover which microcontroller you are to use, whether you are to develop in assembly language or in C, and whether the project is to be simulated then built, just built or just simulated. For example, inputs can be created with switches and outputs can be generated with LEDs. The UART can be interfaced to a PC, and a communication program like **PuTTY** or **TExaSdisplay** can be used to interact with the system.

Part b) Design, build, and test the hardware interfaces. Use a computer-aided-drawing (CAD) program to draw the hardware circuits. Label all pins, chips, and resistor values. In this chapter, there will be one switch for each input and one LED for each output. Connect the switch interfaces to microcontroller input pins, and connect the LED interfaces to microcontroller output pins. Pressing the switch will signify a high input logic value. You should activate the LED to signify a high output logic value.

Part c) Design, implement and test the software that initializes the I/O ports and performs the specified function. Often a main program is used to demonstrate the system.

**Lab 4.1 NOT gate.** The overall objective is to create a NOT gate. The system has one digital input and one digital output, such that the output is the logical complement of the input. Implement the design such that the complement function occurs in the software of the microcontroller.

**Lab 4.2 AND gate.** The overall objective is to create a 3-input AND gate. The system has three digital inputs and one digital output, such that the output is the logical and of the three inputs. Implement the design such that the AND function occurs in the software of the microcontroller.

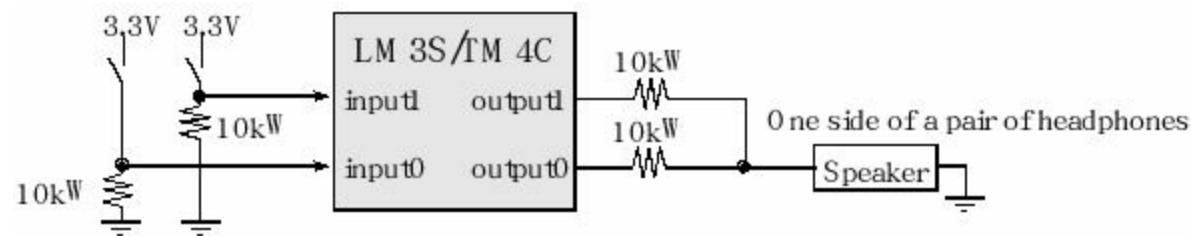
**Lab 4.3 OR gate.** The overall objective is to create a 3-input OR gate. The system has three digital inputs and one digital output, such that the output is the logical or of the three inputs. Implement the design such that the OR function occurs in the software of the microcontroller.

**Lab 4.4 EOR gate.** The overall objective is to create a 2-input EXCLUSIVE OR gate. The system has two digital inputs and one digital output, such that the output is the logical exclusive or of the two inputs. Implement the design such that the EXCLUSIVE OR function occurs in the software of the microcontroller.

**Lab 4.5 Voting logic.** The overall objective is to create a 3-input voting logic. The system has three digital inputs and one digital output, such that the output is high if and only if two or more inputs are high. This means the output will be low if two or more inputs are low. Implement the design such that the voting function occurs in the software of the microcontroller.

**Lab 4.6 Counter.** The overall objective is to create a 4-bit counter. The system has one digital input and four digital outputs. The counter is incremented each time the switch is pressed and released. Wait at least 10 ms in between reading the switch to remove switch bounce. Once the counter reaches 15 roll it back over to 0 on the next press/release.

**Lab 4.7** Oscillator. The overall objective is to a variable frequency oscillator. The system has two digital inputs and two digital outputs. If input1 is true the digital output1 oscillates at 262 Hz. If the input1 is false the output1 remains low. If input2 is true the digital output2 oscillates at 392 Hz. If the input2 is false the output2 remains low. If you connect each output to a  $10\text{ k}\Omega$  resistor as shown in the figure, then you can hear the tones as middle C and middle G. If both inputs are true attempt to oscillate both outputs (output1 at 262 Hz and output2 at 392 Hz).



# 5. Modular Programming

## Chapter 5 objectives are to:

- Present the C syntax of keywords and punctuation
- Describe how to program in a modular way
- Use conditional branching to perform decisions
- Implement for-loops, macros, and recursion
- Implement modular programming using subroutines
- Present functional debugging as a method to test software

In this chapter, we will begin by presenting a general approach to modular design. In specific, we will discuss how to organize software blocks in an effective manner. The ultimate success of an embedded system project depends both on its software and hardware. Computer scientists pride themselves in their ability to develop quality software. Similarly electrical engineers are well-trained in the processes to design both digital and analog electronics. Manufacturers, in an attempt to get designers to use their products, provide application notes for their hardware devices. The main objective of this book is to combine effective design processes together with practical software techniques in order to develop quality embedded systems. As the size and especially the complexity of the software increase, the software development changes from simple "coding" to "software engineering", and the required skills also vary along this spectrum. These software skills include modular design, layered architecture, abstraction, and verification. Real-time embedded systems are usually on the small end of the size scale, but never the less these systems can be quite complex. Therefore, both hardware and software skills are essential for developing embedded systems. Writing good software is an art that must be developed, and cannot be added on at the end of a project. Just like any other discipline (e.g., music, art, science, religion), expertise comes from a combination of study and practice. The watchful eye of a good mentor can be invaluable, so take the risk and show your software to others inviting praise and criticism. Good software combined with average hardware will always outperform average software on good hardware. In this chapter we will outline various techniques for developing quality software.

# 5.1. C Keywords and Punctuation

C has predefined tokens, called **keywords**, which have specific meaning in C programs, as listed in Table 5.1. This section describes keywords and punctuation.

Keyword	Meaning
<b>asm</b>	Specify a function is written in assembly code (specific to ARM Keil™ uVision®)
<b>auto</b>	Specifies a variable as automatic (created on the stack)
<b>break</b>	Causes the program control structure to finish
<b>case</b>	One possibility within a <b>switch</b> statement
<b>char</b>	Defines a number with a precision of 8 bits(C99 defines <b>int8_t</b> and <b>uint8_t</b> )
<b>const</b>	Defines parameter as constant in ROM, and defines a local parameter as fixed value
<b>continue</b>	Causes the program to go to beginning of loop
<b>default</b>	Used in <b>switch</b> statement for all other cases
<b>do</b>	Used for creating program loops
<b>double</b>	Specifies variable as double precision floating point
<b>else</b>	Alternative part of a conditional
<b>extern</b>	Defined in another module
<b>float</b>	Specifies variable as single precision floating point
<b>for</b>	Used for creating program loops
<b>goto</b>	Causes program to jump to specified location
<b>if</b>	Conditional control structure
<b>int</b>	Defines a number with a precision that will vary from compiler to compiler
<b>long</b>	Defines a number with a precision of 32 bits(C99 defines <b>int32_t</b> and <b>uint32_t</b> )
<b>register</b>	Specifies how to implement a local
<b>return</b>	Leave function
<b>short</b>	Defines a number with a precision of 16 bits(C99 defines <b>int16_t</b> and <b>uint16_t</b> )
<b>signed</b>	Specifies variable as signed (default)
<b>sizeof</b>	Built-in function returns the size of an object
<b>static</b>	Stored permanently in memory, accessed locally
<b>struct</b>	Used for creating data structures
<b>switch</b>	Complex conditional control structure

<b>typedef</b>	Used to create new data types
<b>unsigned</b>	Always greater than or equal to zero
<b>void</b>	Used in parameter list to mean no parameter
<b>volatile</b>	Can change implicitly outside the direct action of the software.
<b>while</b>	Used for creating program loops

**Table 5.1. Keywords have predefined meanings.**

The **volatile** keyword disables compiler optimization, forcing the compiler to fetch a new value each time. We will use **volatile** when defining I/O ports because the value of ports can change outside of software action. We will also use **volatile** when sharing a global variable between the main program and an interrupt service routine. It is a good programming practice not to use these keywords for your variable or function names.

**Punctuation marks** are very important in C. It is one of the most frequent sources of errors for both beginning and experienced programmers.

**Semicolons** are used as statement terminators. Strange and confusing syntax errors may be generated when you forget a semicolon, so this is one of the first things to check when trying to remove syntax errors. Notice that one semicolon is placed at the end of every simple statement in Program 5.1. When executed, the function **Step** will output the pattern 10, 9, 5, 6 to Port D. The **#define** statement creates a substitution rule, such that every instance of **STEPPER** in the program is replaced with (\*((volatile uint32\_t \*)0x4000703C)).

```
#define STEPPER (*((volatile uint32_t *)0x4000703C))
void Step(void){
    STEPPER = 10;
    STEPPER = 9;
    STEPPER = 5;
    STEPPER = 6;
}
```

Program 5.1. Semicolons are used to separate one statement from the next.

Preprocessor directives do not end with a semicolon since they are not actually part of the C language proper. Preprocessor directives begin in the first column with the # and conclude at the end of the line. Program 5.2 will fill the array **DataBuffer** with data read from the input Port A. We assume in this example that Port A has been initialized as an input. Notice that semicolons are used to separate the three fields of the for loop statement.

```
uint8_t DataBuffer[100];
#define GPIO_PORTA_DATA_R (*((volatile uint32_t *)0x400043FC))
void Fill(void){ uint32_t j;
    for(j=0; j<100; j++){
        DataBuffer[j] = GPIO_PORTA_DATA_R;
    }
}
```

## Program 5.2. Semicolons are used to separate three fields of the for statement.

We can define a label using the **colon**. Although C has a **goto** statement, it is strongly discouraged to use **goto**. Software is easier to understand using the block-structured control statements (if, if else, for, while, do while, and switch case.) Program 5.3 will return after the Port A input reads the same value 10000 times in a row. Again we assume Port A has been initialized as an input. Notice that every time the current value on Port A is different from the previous value the counter is reinitialized. Notice the use of colons in Program 5.3.

```
int32_t Debounce(void){ uint32_t cnt; uint32_t last,new;
start:  cnt = 0;      // number of times Port C is the same
        last = GPIO_PORTA_DATA_R;
loop:   if(++cnt==10000) goto done; // count 10000 times
        new = GPIO_PORTA_DATA_R;
        if(last != new) goto start; // changed??
        goto loop;
done:   return(last);
}
```

Program 5.3. Colons are used to define labels (places to which we can jump).

Colons also terminate **case**, and **default** prefixes that appear in **switch** statements. In Program 5.4 one output to the stepper motor produced each time the function **OneStep** is called. The proper stepper motor sequence is 10–9–5–6. The default case is used to restart the pattern. For both applications of the colon (goto and switch), we see that a label is created that is a potential target for a transfer of control. Notice the use of colons in Program 5.4.

```
uint8_t Last=10;
void OneStep(void){
    uint8_t theNext;
    switch(Last){
        case 10: theNext = 9; break; // 10 to 9
        case 9: theNext = 5; break; // 9 to 5
        case 5: theNext = 6; break; // 5 to 6
        case 6: theNext = 10; break; // 6 to 10
        default: theNext = 10;
    }
    GPIO_PORTD_DATA_R = theNext;
    Last = theNext; // set up for next call
}
```

Program 5.4. Colons are also used with the switch statement.

**Commas** separate items that appear in lists. We can create multiple variables of the same type using commas.

```
uint32_t beginTime,endTime,elapsedTime;
```

Lists are also used with functions having multiple parameters, both when the function is defined and called. Program 5.5 adds two 16-bit signed numbers, implementing ceiling and floor. Notice the use of commas in Program 5.5.

```
int16_t add(int16_t x, int16_t y){ int16_t z;
z = x+y;
if((x>0)&&(y>0)&&(z<0))z = 32767;
if((x<0)&&(y<0)&&(z>0))z = -32768;
return(z);
}
void main(void){ int16_t a,b;
a = add(2000,2000)
b = 0
while(1){
    b = add(b,1);
}
```

Program 5.5. Commas separate the parameters of a function.

Lists can also be used in general expressions. Sometimes it adds clarity to a program if related variables are modified at the same place. The value of a list of expressions is always the value of the last expression in the list. In the following example, first **thetime** is incremented, next **thedate** is decremented, and then **x** is set to **k+2**.

```
x = (thetime++, thedate--, k+2);
```

Apostrophes are used to specify character literals. Assuming the function **OutChar** will display a single ASCII character, Program 5.6 will display the lower case alphabet.

```
void Alphabet(void){ char mych;
for(mych='a'; mych<='z'; mych++){
    OutChar(mych); // Print next letter
}
}
```

Program 5.6. Apostrophes are used to specify characters.

Quotation marks are used to specify string literals. Strings are stored as a sequence of ASCII characters followed by a termination code, 0. Program 5.7 will display “Hello World” twice.

```
const char Msg[] = "Hello World"; // string constant
void OutString(const char str[]){ int i;
i = 0;
while(str[i]){ // output until the 0 termination
    OutChar(str[i]); // Print next letter
    i = i+1;
}
}
```

```
void PrintHelloWorld(void){  
    OutString("Hello World");  
    OutString(Msg);  
}
```

Program 5.7. Quotation marks are used to specify strings.

**Braces** {} are used throughout C programs. The most common application is for creating a compound statement. Each open brace { must be matched with a closing brace }. Notice the use of indenting in Programs 5.1 through 5.7 that helps to match up braces. Each time an open brace is used, the source code is tabbed over using 2 spaces. In this way, it is easy to see at a glance the brace pairs.

Square **brackets** enclose array dimensions (in declarations) and subscripts (in expressions). The following defines an integer array named **Fifo** consisting of 100 16-bit signed numbers.

```
int16_t Fifo[100];
```

The following assigns the variable **PutPt** to the address of the first entry of the array.

```
PutPt = &Fifo[0];
```

**Parentheses** enclose argument lists that are associated with function declarations and calls. They are required even if there are no arguments. As with all programming languages, C uses parentheses to control the order in which expressions are evaluated. Thus,  $(11+3)/2$  yields 7, whereas  $11+3/2$  yields 12. Parentheses are very important when writing expressions.

# 5.2. Modular Design using Abstraction

In Section 2.4, we presented successive refinement as a method to convert a problem statement into a software algorithm. Successive refinement is the transformation from the general to the specific. In this section, we introduce the concept of modular programming and demonstrate that it is an effective way to organize our software projects. There are four reasons for forming modules. First, functional abstraction allows us to reuse a software module from multiple locations. Second, complexity abstraction allows us to divide a highly complex system into smaller less complicated components. The third reason is portability. If we create modules for the I/O devices, then we can isolate the rest of the system from the hardware details. This approach is sometimes called a hardware abstraction layer. Since all the software components that access an I/O port are grouped together, it will be easier to redesign the embedded system on a machine with different I/O ports. Finally, another reason for forming modules is security. Modular systems by design hide the inner workings from other modules and provide a strict set of mechanisms to access data and I/O ports. Hiding details and restricting access generates a more secure system.

Software must deal with **complexity**. Most real systems have many components, which interact in a complex manner. The size and interactions will make it difficult to conceptualize, abstract, visualize, and document. In this chapter we will present data flow graphs and call graphs as tools to describe interactions between components. Software must deal with **conformity**. All design, including software design, must interface with existing systems and with systems yet to be designed. Interfacing with existing systems creates an additional complexity. Software must deal with **changeability**. Most of the design effort involves change. Creating systems that are easy to change will help manage the rapid growth occurring in the computer industry.

## 5.2.1. Definition and Goals

The key to completing any complex task is to break it down into manageable subtasks. Modular programming is a style of software development that divides the software problem into distinct well-defined modules. The parts are as small as possible, yet relatively independent. Complex systems designed in a modular fashion are easier to debug because each module can be tested separately. Industry experts estimate that 50 to 90% of software development cost is spent in maintenance. All five aspects of software maintenance

- Correcting mistakes,
- Adding new features,
- Optimizing for execution speed or program size,
- Porting to new computers or operating systems, and
- Reconfiguring the software to solve a similar related program

are simplified by organizing the software system into modules. The approach is particularly useful when a task is large enough to require several programmers.

A **program module** is a self-contained software task with clear entry and exit points. There is a distinct difference between a module and the assembly language subroutine or C language function. A module is usually a collection of subroutines or functions that in its entirety performs a well-defined set of tasks. A collection of 32-bit trigonometry functions is an example of a module. The device driver in Program 4.3 is another example of a module. Modular programming involves both the specification of the individual modules and the connection scheme whereby the modules are interfaced together to form the software system. While the module may be called from many locations throughout the software, there should be well-defined **entry points**. In C, the entry point of a module is defined in the header file and is specified by a list of function prototypes for the public functions. Similarly in assembly, the entry point of a module is also a list of public subroutines that can be called.

**Common Error:** In many situations the input parameters have a restricted range. It would be inefficient for the module and the calling routine to both check for valid input. On the other hand, an error may occur if neither checks for valid input.

An **exit point** is the ending point of a program module. The exit point of a subroutine is used to return to the calling routine. We need to be careful about exit points. It is important that the stack be properly balanced at all exit points. Similarly, if the subroutine returns parameters, then all exit points should return parameters in an acceptable format. If the main program has an exit point it either stops the program or returns to the debugger. In most embedded systems, the main program does not exit.

**Common Error:** It is an error if all the exit points of an assembly subroutine do not balance the stack and return parameters in the same way.

In this section, an object refers to either a subroutine or a data element. A **public** object is one that is shared by multiple modules. This means a public object can be accessed by other modules. Typically, we make the most general functions of a module public, so the functions can be called from other modules. For a module performing I/O, typical public functions include initialization, input, and output. A **private** object is one that is not shared. I.e., a private object can be accessed by only one module. Typically, we make the internal workings of a module private, so we hide how a private function works from user of the module. In an object-oriented language like C++ or Java, the programmer clearly defines a function or data object as public or private. Later in this chapter, we will present a naming convention for assembly language or C that can be used in an equivalent manner to define a function or data object as public or private.

At a first glance, I/O devices seem to be public. For example, Port D resides permanently at the fixed address of 0x400073FC, and the programmer of every module knows that. In other words, from a syntactic viewpoint, any module has access to any I/O device. However, in order to reduce the complexity of the system, we will restrict the number of modules that actually do access the I/O device. From a “what do we actually do” perspective, however, we will write software that considers I/O devices as private, meaning an I/O device should be accessed by only one module. In general, it will be important to clarify which modules have access to I/O devices and when they are allowed to access them. When more than one module accesses an I/O device, then it is important to develop ways to arbitrate or synchronize. If two or more want to access the device simultaneously

arbitration determines which module goes first. Sometimes the order of access matters, so we use synchronization to force a second module to wait until the first module is finished. Most microcontrollers do not have architectural features that restrict access to I/O ports, because it is assumed that all software burned into its ROM was designed for a common goal, meaning from a security standpoint one can assume there are no malicious components. However, as embedded systems become connected to the Internet, providing the power and flexibility, security will become important issue.

**Checkpoint 5.1:** What conflict could arise if multiple modules use the same port, but module initialization functions are not friendly? How do you resolve the conflict?

**Information hiding** is similar to minimizing coupling. It is better to separate the mechanisms of software from its policies. We should separate “what the function does” from “how the function works”. What a function does is defined by the relationship between its inputs and outputs. It is good to hide certain inner workings of a module and simply interface with the other modules through the well-defined input/output parameters. For example we could implement a variable size buffer by maintaining the current byte count in a global variable, **Count**. A good module will hide how **Count** is implemented from its users. If the user wants to know how many bytes are in the buffer, it calls a function that returns the count. A badly written module will not hide **Count** from its users. The user simply accesses the global variable **Count**. If we update the buffer routines, making them faster or better, we might have to update all the programs that access **Count** too. Allowing all software to access **Count** creates a security risk, making the system vulnerable to malicious or incompetent software. The object-oriented programming environments provide well-defined mechanisms to support information hiding. This separation of policies from mechanisms is discussed further in the section on layered software.

**Maintenance Tip:** It is good practice to make all permanently-allocated data and all I/O devices private. Information is transferred from one module to another through well-defined public function calls.

The **Keep It Simple Stupid** approach tries to generalize the problem so that the solution uses an abstract model. Unfortunately, the person who defines the software specifications may not understand the implications and alternatives. As a software developer, we always ask ourselves these questions:

“How important is this feature?”

“What if it worked this different way?”

Sometimes we can restate the problem to allow for a simpler and possibly more powerful solution.

## 5.2.2. Functions, Procedures, Methods, and Subroutines

A program module that performs a well-defined task can be packaged up and defined as a single entity. Functions in that module can be invoked whenever a task needs to be performed. Object-oriented high-level languages like C++ and Java define program modules as methods. Functions and procedures are defined in some high-level languages like Pascal, FORTRAN, and Ada. In these languages, functions return a parameter and procedures do not. Most high-level languages however

define program modules as functions, whether they return a parameter or not. A subroutine is the assembly language version of a function. Consequently, subroutines may or may not have input or output parameters. Formally, there are two components to a subroutine: definition and invocation. The subroutine **definition** specifies the task to be performed. Examples of three subroutine definitions can be seen as the SysTick functions in Program 4.7. In other words, it defines what will happen when executed. The syntax for a subroutine definition was presented previously in Section 2.8. It begins with a label, which will be the name of the subroutine and ends with a return instruction. The definition of a subroutine includes a formal specification its input parameters and output parameters. In well-written software, the task performed by a subroutine will be well-defined and logically complete. The subroutine **invocation** is inserted to the software system at places when and where the task should be performed. An example of a subroutine invocation can be seen in Program 3.9. We define software that invokes the subroutine as “the calling program” because it calls the subroutine. There are three parts to a subroutine invocation: pass input parameters, subroutine call, and accept output parameters. If there are input parameters, the calling program must establish the values for input parameters before it calls the subroutine. A **BL** instruction is used to call the subroutine. After the subroutine finishes, and if there are output parameters, the calling program accepts the return value(s). In this chapter, we will pass parameters using the registers. If the register contains a value, the parameter is classified as **call by value**. If the register contains an address, which points to the value, then the parameter is classified as **call by reference**.

For example, consider a subroutine that samples the 12-bit ADC, as drawn in Figure 5.1. An analog input signal is connected to ADC0. The details of how the ADC works will be presented later in Chapter 10, but for now we focus on the defining and invoking subroutines. The execution sequence begins with the calling program setting up the input parameters. In this case, the calling program sets Register R0 equal to the channel number, **MOV R0,#0**. The instruction **BLADC\_In** will save the return address in the LR register and jump to the **ADC\_In** subroutine. The subroutine performs a well-defined task. In this case, it takes the channel number in Register R0 and performs an analog to digital conversion, placing the digital representation of the analog input into Register R0. The **BX LR** instruction will move the return address into the PC, returning the execution thread to the instruction after the **BL** in the calling program. In this case, the output parameter in Register R0 contains the result of the ADC conversion. It is the responsibility of the calling program to accept the return parameter. In this case, it simply stores the result into variable **n**. In this example, both the input and output parameters are call by value.

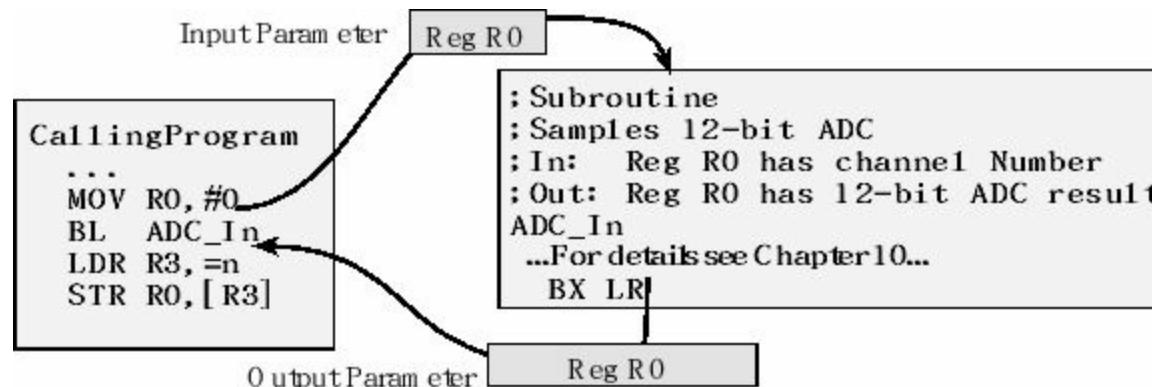


Figure 5.1. The calling program invokes the **ADC\_In** subroutine passing parameters in registers.

## 5.2.3. Dividing a Software Task into Modules

The overall goal of modular programming is to enhance clarity. The smaller the task, the easier it will be to understand. **Coupling** is defined as the influence one module's behavior has on another module. In order to make modules more independent we strive to minimize coupling. Obvious and appropriate examples of coupling are the input/output parameters explicitly passed from one module to another. A quantitative measure of coupling is the number of bytes per second (bandwidth) that are transferred from one module to another. On the other hand, information stored in public global variables can be quite difficult to track. In a similar way, shared accesses to I/O ports can also introduce unnecessary complexity. Public global variables cause coupling between modules that complicate the debugging process because now the modules may not be able to be separately tested. On the other hand, we must use global variables to pass information into and out of an interrupt service routine and from one call to an interrupt service routine to the next call. When passing data into or out of an interrupt service routine, we group the functions that access the global into the same module, thereby making the global variable private. Another problem specific to embedded systems is the need for fast execution, coupled with the limited support for local variables. On many microcontrollers it is inefficient to implement local variables on the stack. Consequently, many programmers opt for the less elegant yet faster approach of global variables. Again, if we restrict access to these globals to function in the same module, the global becomes private. It is poor design to pass data between modules through public global variables; it is better to use a well-defined abstract technique like a FIFO queue.

We should assign a logically complete task to each module. The module is logically complete when it can be separated from the rest of the system and placed into another application. The interface design is extremely important. The interface to a module is the set of public functions that can be called and the formats for the input/output parameters of these functions. The interfaces determine the policies of our modules: "What does the module do?" In other words, the interfaces define the set of actions that can be initiated. The interfaces also define the coupling between modules. In general we wish to minimize the bandwidth of data passing between the modules yet maximize the number of modules. Of the following three objectives when dividing a software project into subtasks, it is really only the first one that matters

- Make the software project easier to understand
- Increase the number of modules
- Decrease the interdependency (minimize bandwidth between modules).

**Checkpoint 5.2:** List some examples of coupling.

We will illustrate the process of dividing a software task into modules with an abstract but realistic example. The overall goal of the example shown in Figure 5.2 is to sample data using an ADC, perform calculations on the data, and output results. The organic light emitting diode (OLED) could be used to display data to the external world. Notice the typical format of an embedded system in that it has some tasks performed once at the beginning, and it has a long sequence of tasks performed over and over. The structure of this example applies to many embedded systems such as a diagnostic medical instrument, an intruder alarm system, a heating/AC controller, a voice recognition module, automotive emissions controller, or military surveillance system. The left side of Figure 5.2 shows the complex software system defined as a linear sequence of ten steps, where each step represents

many lines of assembly code. The linear approach to this program follows closely to linear sequence of the processor as it executes instructions. This linear code, however close to the actual processor, is difficult to understand, hard to debug, and impossible to reuse for other projects. Therefore, we will attempt a modular approach considering the issues of functional abstraction, complexity abstraction, and portability in this example. The modular approach to this problem divides the software into three modules containing seven subroutines. In this example, assume the sequence Step4-Step5-Step6 causes data to be sorted. Notice that this sorting task is executed twice.

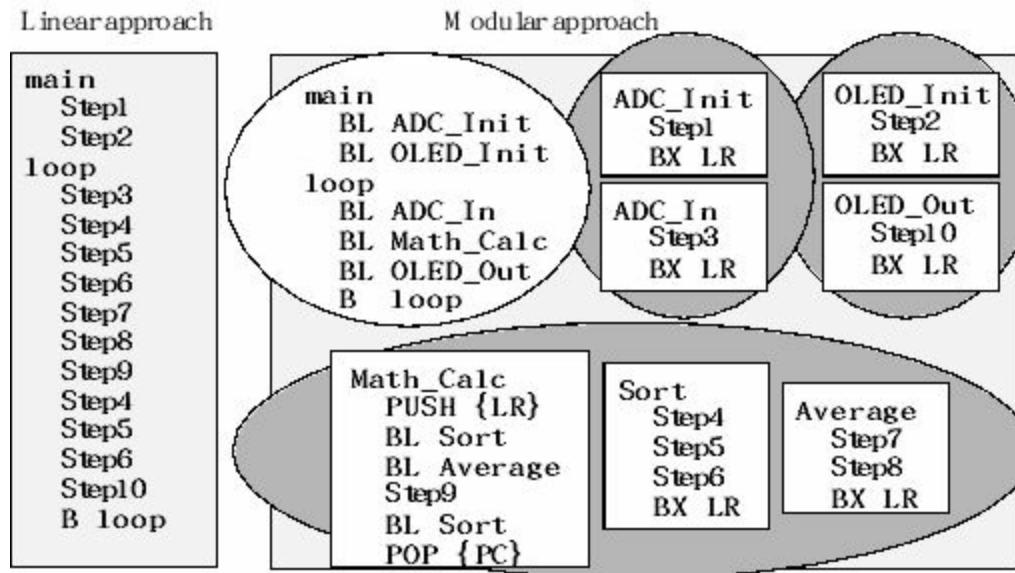


Figure 5.2. A complex software system is broken into three modules containing seven subroutines.

Functional abstraction encourages us to create a **Sort** subroutine allowing us to write the software once, but execute it from different locations. Complexity abstraction encourages us to organize the ten-step software into a main program with multiple modules, where each module has multiple subroutines. For example, assume the assembly instructions in Step1 cause the ADC to be initialized. Even though this code is executed only once, complexity abstraction encourages us to create an **ADC\_Init** subroutine so the system is easier to understand and easier to debug. In a similar way assume Step2 initializes the OLED port, Step3 samples the ADC, the sequence Step7-Step8 performs an average, and Step10 outputs to the OLED. Therefore, each well-defined task is defined as a separate subroutine. The subroutines are then grouped into modules. For example, the ADC module is a collection of subroutines that operate the ADC. The complex behavior of the ADC is now abstracted into two easy to understand tasks: turn it on, and use it. In a similar way, the OLED module includes all functions that access the OLED. Again, at the abstract level of the main program, understanding how to use the OLED is a matter knowing we first turn it on then we transmit data. The math module is a collection of subroutines to perform necessary calculations on the data. In this example, we assume sort and average will be private subroutines, meaning they can be called only by software within the math module and not by software outside the module. Making private subroutines is an example of “information hiding”, separating what the module does from how the module works. When we **port** a system, it means we take a working system and redesign it with some minor but critical change. The OLED device is used in this system to output results. We might be asked to port this system onto a device that uses an LCD in place of the OLED for its output. In this case, all we need to do is design, implement and test an LCDmodule with two

subroutines **LCD\_Init** and **LCD\_Out** that function in a similar manner as the existing OLED routines. The modular approach performs the exact same ten steps in the exact same order. However, the modular approach is easier to debug, because first we debug each subroutine, then we debug each module, and finally we debug the entire system. The modular approach clearly supports code reuse. For example, if another system needs an ADC, we can simply use the ADC module software without having to debug it again.

**Observation:** When writing modular code, notice its two-dimensional aspect. Down the y-axis still represents time as the program is executed, but along the x-axis we now visualize a functional block diagram of the system showing its data flow: input, calculate, output.

**Observation:** When writing modular code, we hide details that are likely to change. Furthermore, we take details that are unlikely to change and use them to define the interfaces between modules.

## 5.2.4. How to Draw a Call Graph

Defined previously in Figure 2.5, we recall that a **call graph** is a graphical representation of the organizational structure of the modules pieced together to construct a system. In this section, we will work through the process of drawing a call graph. A **software module** is a collection of public functions, private functions, and private global variables that together perform a complete task. Modular programming places multiple related subroutines into a single module. I/O devices are essential in all computers, but they are particularly relevant when developing software for an embedded system. Just like our software, it is appropriate to group I/O ports into **hardware modules**, which together perform a complete I/O task. The main program is at the top, and the I/O ports are at the bottom. In a hierarchical system, the modules are organized both in a horizontal and vertical fashion. Modules at the same horizontal level perform similar but distinct functions (e.g., we could place all I/O modules at the same horizontal level in the call graph hierarchy). From a vertical perspective, we place modules responsible for overall policy decisions at the top and modules performing implementations at the bottom of the call graph hierarchy. Since one of the advantages of breaking a large software project into subtasks is concurrent development, it makes sense to consider concurrency when dividing the tasks. In other words, the modules should be partitioned in such a way that multiple programmers can develop and test the subtasks as independently as possible. On the other hand, careful and constant supervision is required as modules are connected together and tested.

An arrow represents a software linkage, i.e., one software module calling another. We draw the tail of the arrow in the software module that initiates the call, and we point the head of the arrow at the software module it calls. When programming in C, including a header file in the implementation file of a module defines an arrow in the call graph. The exception to this rule is including a header file that contains constants and has no corresponding implementation file. The file **tm4c123ge6pm.h** contains the I/O port definitions for the TM4C123, and has no code file. Therefore, **tm4c123ge6pm** is not a module. On the other hand we can create an **ADC** module by placing all the ADC functions in the **ADC.c** file and defining the prototypes for the public functions in the **ADC.h** file. If we place an **#include "ADC.h"** statement in our **main.c** code, we create a call graph arrow from the **main** module to the **ADC** module, because software in the **main** module can call the public functions of the **ADC** module. In a large complex system, we will add call graph arrows for situations where it **can**

call rather than where is **does** call. It is easier in a larger system to draw the can-call arrows than the does-call arrows, because we just have to look at the header files each code file includes. In contrast, we usually draw only the does-call arrows for accesses to I/O devices. In other words, a **device driver** is a collection of I/O software for a particular I/O device. This approach will also simplify maintaining a call graph during phases while the software is being designed, written, debugged, or upgraded. Changes to the list of header files included by a module are much less frequent than changes to the list of functions actually called. On the other hand, most embedded systems are simple enough that it is more appropriate to show just the does-call arrows.

A global variable is one which is allocated in permanent RAM. These variables are a necessary and important component of an embedded system, because some information is permanent in nature. Good programming style however suggests we restrict access to these global variables to a single module. On the other hand, a public global variable is accessed by more than one module. Public globals represent poor programming style, because they add complexity to the system. Reading and writing public globals add arrows to the call graph. If module A reads a global variable in module B, then we add an arrow from B to A, because activities in B cause changes in A. If module A writes to a global variable in module B, then we add an arrow from A to B, because activities in A cause changes in B. If there is an arrow from A to B, and a second arrow from B to A, then modules A and B must be tested together. Coupling through shared global variables is a very bad style because debugging will be difficult.

Typically, hardware modules are at the lowest level, because hardware responds to software. An arrow from an oval to a rectangle represents a hardware access, i.e., the software reads from or writes to an I/O port. An arrow from an oval to a rectangle signifies the usual read/write access to the hardware module or public global. We will study interrupts in detail in Chapter 9. With interrupts, a hardware triggering event causes the software interrupt service routine to execute. Therefore with interrupts, we add an arrow from the hardware module to the software module. It can be drawn with two single-headed arrows or one double-headed arrow. Defining arrows between hardware and software modules allows us to identify problems such as conflict (two modules writing to the same I/O configuration registers), or race conditions (e.g., one module reading a port before another module initializes it).

Figure 5.3 shows a call graph of the example presented in Figure 5.2. To draw a call graph, we first represent all the software modules as ovals. Inside the oval lives the functions and variables of that module. Normally, there is not space to list all the subroutines of each module inside the oval, but they are drawn here in this figure so you can see the details of how the graph is drawn. In this example, there is a main program and three software modules. Since this main program calls the Math module, the main program is at a higher level than the Math module, therefore the oval for main will be drawn above the oval for Math. The ADC, Math, and OLED modules do not call each other and each is called by main, so they exist at the same level. In this example, there are two hardware modules, and they are drawn as rectangles. To draw the arrows, we search for subroutine call instructions. The tail of an arrow is placed in the module containing the calling program, and the head of an arrow is placed in the module with the subroutine.

If there are multiple calls from one module to another, only one arrow is needed. For example, there are two calls from main to ADC, but only one arrow is drawn. No arrows will be drawn to describe subroutine calls within a module. For example, we do not need to draw arrows representing the Math

routine **Math\_Calc** calling **Sort** and **Average**, because these are all within the same module. Two arrows from software to hardware are drawn, because the ADC module accesses the ADC hardware, and the OLED module accesses the OLED hardware.

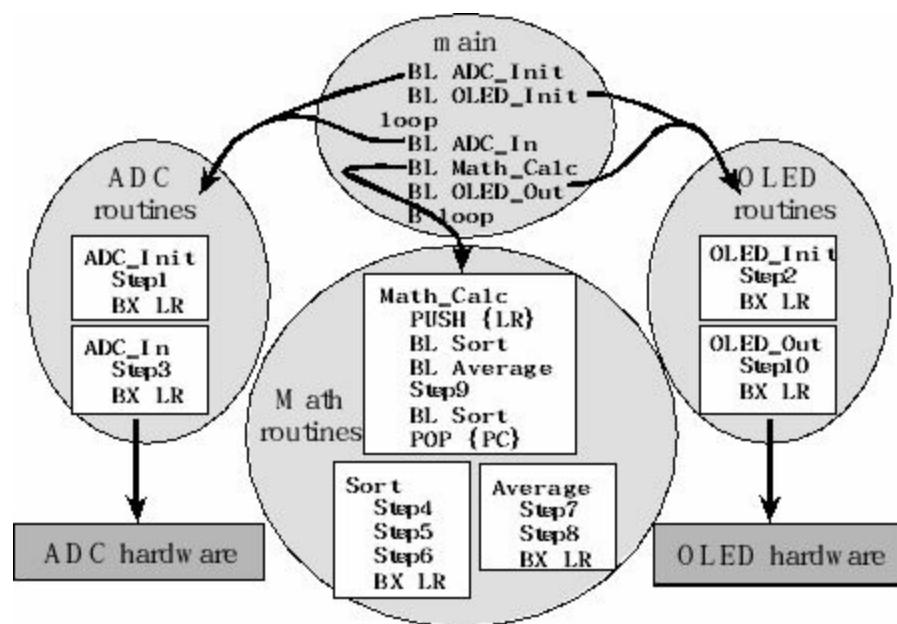


Figure 5.3. A call graph of the system of Figure 5.2.

We can develop and connect modules in a **hierarchical** manner. Construct new modules by combining existing modules. In general, to reduce complexity of the system we want to maximize the number of modules and minimize the number of arrows between them. More specifically, we want to minimize the bandwidth of data flowing from one module to the other.

**Observation:** If module A calls module B, and module B calls module A, then these two modules must be tested together.

**Maintenance Tip:** It is good practice to have one hardware module (e.g., the ADC or OLED) accessed by exactly one software module.

**Checkpoint 5.3:** In what way are I/O devices considered as public?

**Checkpoint 5.4:** How can you implement a system that considers I/O devices as private?

## 5.2.5. How to Draw a Data Flow Graph

As shown previously in Figure 2.4, a **data flow graph** is a graphical representation of the data as it traverses the system. Figure 5.4 shows the data flow graph for the example presented in Figure 5.2. In general, the data flow graph contains the same software and hardware modules as the call graph. There are two fundamental differences, however. The arrows in a data flow graph specify the direction, data type, and rate of data transfer. Conversely, arrows in a call graph specify which module invoked which other module. The second difference is in general we draw modules in a data flow graph from left to right as data enters as inputs on the left and exits as outputs on the right. In a call graph, we draw modules top to bottom from high-level to low-level functions.

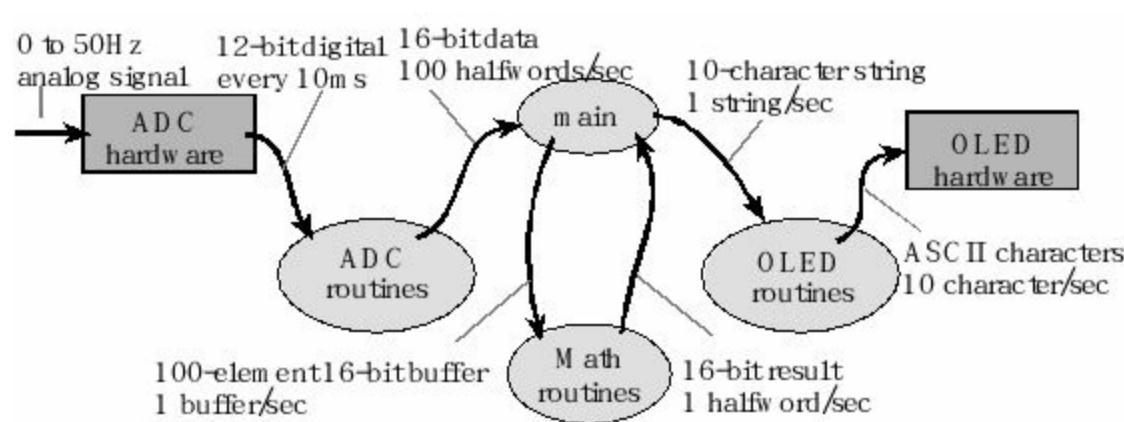


Figure 5.4. A data flow graph of the system of Figure 5.2.

Assume in this example, the analog input contains frequency components from 0 to 50 Hz. We classify the signal as analog and specify the bandwidth of the analog signal to be 50 Hz. The output of the ADC hardware is 12-bit digital samples. If the 12-bit ADC is sampled 100 times a second, we define the bandwidth of the digital data out of the ADC software module as 100 samples/sec, 100 halfwords/sec or 200 bytes/sec. Assume once a second, the main program fills a 100-element buffer and passes it to the math module. The math module takes in 100 samples and generates one 16-bit result. In this case, we define the output of the math module to be 1 halfword/sec. If once a second each result is printed as ten ASCII characters using the OLED, then the bandwidth into and out of the OLED software module will be 10 characters/sec.

## 5.2.6. Top-down versus Bottom-up Design

Hierarchical systems have tree-structured call graphs, like system in Figure 5.3. Layered systems have call graphs that group the modules into layers, such that the linkage arrows only go from a high level to a lower level or within the same level. A lower level module is not allowed to call a higher level. If at all possible, we should avoid cyclic graphs. A cycle in the call graph will make testing difficult. Recall that we design top down and test bottom up. When there is a cycle in the call graph, there is no good place to start debugging. There are two approaches to hierarchical programming. The top-down approach starts with a general overview, like an outline of a paper, and builds refinement into subsequent layers. Most engineers believe top down is the proper approach to design. A top-down programmer was once quoted as saying,

“Write no software until every detail is specified”

Top down provides an excellent global approach to the problem. Managers like top down because it gives them tighter control over their workers. The top-down approach works well when an existing operational system is being upgraded or rewritten.

On the other hand the bottom-up approach starts with the smallest detail, builds up the system “one brick at a time.” The bottom-up approach provides a realistic appreciation of the problem because we often cannot appreciate the difficulty or the simplicity of a problem until we have tried it. It allows programmers to start immediately coding and gives programmers more input into the design. For example, a low level programmer may be able to point out features that are not possible and suggest other features that are even better. Some software projects are flawed from their conception. With bottom-up design, the obvious flaws surface early in the development cycle.

Bottom-up is a better approach when designing a complex system and specifications are open-ended. For example, when researching new technologies or exploring new markets, you can’t perform a top-down design because there are no specifications or constraints with which to work. However, a bottom-up approach allows you to brainstorm putting pieces together in new and creative ways. In a bottom-up design, questions begin with “I wonder what would happen if...”

On the other hand, top down is better when you have a very clear understanding of the problem specifications and the constraints of your system.

# 5.3. Making Decisions

The previous section presented fundamental concepts and general approaches to solving problems on the computer. In the subsequent sections, detailed implementations will be presented.

## 5.3.1. Conditional Branch Instructions

Normally the computer executes one instruction after another in a sequential or linear fashion. In particular, the next instruction to execute is found immediately following the current instruction. We use branch instructions to deviate from this straight line path. The branch instructions were presented earlier in Chapter 3. The following unsigned conditional branch instructions must follow a subtract or compare, such as **SUBS CMN** and **CMP**.

<b>BLO target</b>	;Branch if unsigned less than	if C=0,	same as
<b>BCC</b>			
<b>BLS target</b>	;Branch if unsigned less than or equal to	if C=0 or Z=1	
<b>BHS target</b>	;Branch if unsigned greater than or equal to		if C=1, same as
<b>BCS</b>			
<b>BHI target</b>	;Branch if unsigned greater than	if C=1 and Z=0	

After a subtraction the carry bit is 0 if there is an error and 1 if there is no error. To understand exactly how unsigned conditional branches work, let's start with the **BLO** instruction. As stated earlier, we bring the first unsigned number into a register, and then subtract a second unsigned number from the first. Let's call the first number First and the second number Second. The **BLO** instruction is supposed to branch if the first unsigned number is strictly less than the second. The two possibilities, branch or no branch, are illustrated in number wheels drawn in Figure 5.5.

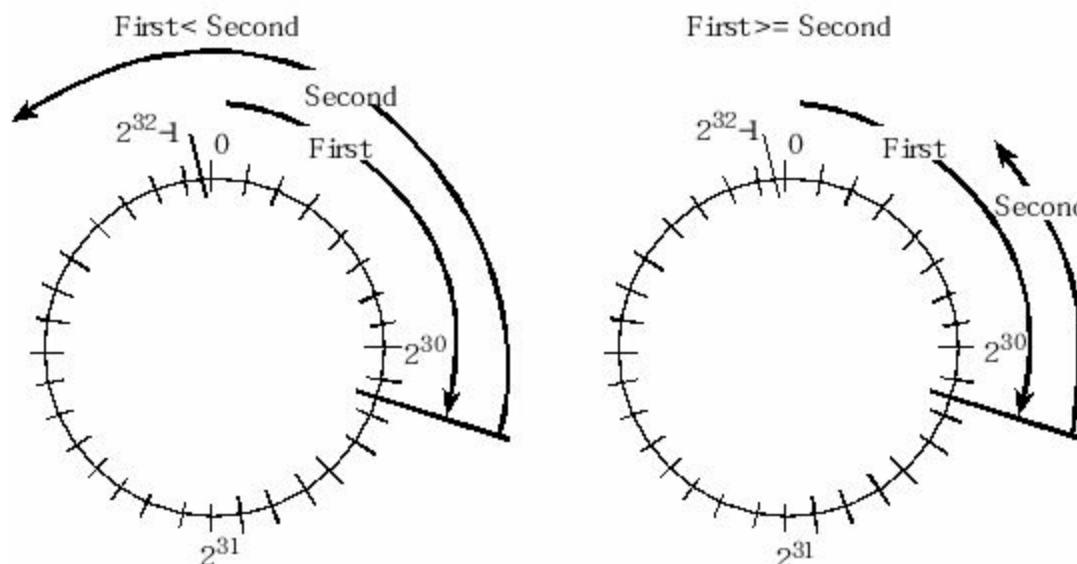


Figure 5.5. Number wheel on left shows the result of subtracting a big unsigned number from a little number, and the one on the right occurs when subtracting a small unsigned number from a large one.

Assume for a moment that the condition is true, meaning  $\text{First} < \text{Second}$ . Since  $\text{First} < \text{Second}$ ,  $\text{First} - \text{Second}$  should be a negative number. I.e., if we subtract a big unsigned number from a small unsigned number, an unsigned overflow must occur, because the correct result of the subtraction is negative, but there are no negative numbers in the unsigned format. Thus, the C bit must be clear ( $C=0$  means overflow error). The left side of Figure 5.5 shows the subtraction will always cross the  $0-(2^{32}-1)$  barrier because  $\text{First} < \text{Second}$ . Conversely, assume the condition is false, meaning the first unsigned number is greater than or equal the second. The right side of Figure 5.5 shows when we subtract the smaller second number from the bigger first number we get the correct result. In this case, the C bit will be set ( $C=1$  means no overflow error). Thus, the **BLO** instruction can be defined as branch if  $C=0$ . The **BHS** instruction is the logical complement of **BLO**, so **BHS** instruction will branch if the C bit is set. The **BLS** instruction will branch if the first number is less than the second ( $C=0$ ) or if the two numbers are equal ( $Z=1$ ). Hence, the operation of the **BLS** instruction can be defined as branch if ( $C=0$  or  $Z=1$ ). Lastly, the **BHI** instruction is the logical complement of **BLS**, so **BHI** instruction will branch if ( $C=1$  and  $Z=0$ ).

The following signed branch instructions must follow a subtract compare or test instruction, such as **SUBS CMN** and **CMP**.

<b>BLT target</b>	; if signed less than	if $(\sim N \& V   N \& \sim V) = 1$ if $N \neq V$
<b>BGE target</b>	; if signed greater than or equal to	if $(\sim N \& V   N \& \sim V) = 0$ if $N = V$
<b>BGT target</b> $N = V$	; if signed greater than	if $(Z   \sim N \& V   N \& \sim V) = 0$ if $Z = 0$ and
<b>BLE target</b>	; if signed less than or equal to	if $(Z   \sim N \& V   N \& \sim V) = 1$ if $Z = 1$ or $N \neq V$

To understand exactly how signed conditional branches work, we will begin with the **BLT** instruction. We bring the first signed number into a register, and then subtract a second signed number from the first. The **BLT** instruction is supposed to branch if the first signed number is strictly less than the second. Assume for a moment that  $\text{First} < \text{Second}$ , thus the branch should occur. Since  $\text{First} < \text{Second}$ ,  $\text{First} - \text{Second}$  should be a negative number. Let's further dissect this case into two subcases. If the V bit is clear, the subtraction is correct and the N bit will be 1. This subcase defines the  $N \& \sim V$  term. If the V bit is set, the subtraction is incorrect and the result will be incorrectly positive, making N bit 0. This subcase defines the  $\sim N \& V$  term. Conversely, assume the condition is false, meaning  $\text{First} \geq \text{Second}$ , and the branch should not occur. Since  $\text{First} \geq \text{Second}$ ,  $\text{First} - \text{Second}$  should be a positive number. If the V bit is clear, the subtraction is correct and the N bit will be 0. If the V bit is set, the subtraction is incorrect and the result will be incorrectly negative, making N bit 1. Thus, the **BLT** instruction can be defined as branch if  $(\sim N \& V | N \& \sim V) = 1$ . The **BGE** instruction is the logical complement of **BLT**, so **BGE** instruction will branch if  $(\sim N \& V | N \& \sim V) = 0$ . The **BLE** instruction will branch if the first number is less than the second ( $(\sim N \& V | N \& \sim V) = 1$ ) or if the two numbers are equal ( $Z=1$ ). Combining the less than with the equal conditions, the operation of **BLE** instruction can be defined as branch if  $(Z | ((\sim N \& V) | (N \& \sim V))) = 1$ . Lastly, the **BGT** instruction is the logical complement of **BLE**, so **BGT** instruction will branch if  $(Z | ((\sim N \& V) | (N \& \sim V))) = 0$ .

Normally, we will never use the **CMN** instruction explicitly. However, when we write the instruction **CMP R0,#n** the number **n** is specified by the flexible second operand. Thus, there are only a limited number of choices we can select for the constant **n**. For example, **n** cannot equal -2. Luckily for us, the assembler will automatically convert **CMP R0,#-2** into the equivalent instruction **CMN R0,#2**.

**Common Error:** It is usually an error to follow a compare instruction with **BPL** or **BMI**.

### 5.3.2. Conditional if-then Statements

Decision making is an important aspect of software programming. Two values are compared and certain blocks of program are executed or skipped depending on the results of the comparison. In assembly language it is important to know the precision (e.g., 8-bit, 16-bit, 32-bit) and the format of the two values (e.g., unsigned, signed). It takes three steps to perform a comparison. You begin by reading the first value into a register. If the second value is not a constant, it must be read into a register, too. The second step is to compare the first value with the second value. You can use either a subtract instruction with the **S** (**SUBS**) or a compare instruction (**CMPCMN**). The **CMP CMN SUBS** instructions set the condition code bits. The last step is a conditional branch.

**Observation:** Think of the three steps 1) bring first value into a register, 2) compare to second value, 3) conditional branch, **bxx** (where xx is eq ne lo ls hi hs gt ge lt or le). The branch will occur if (first is xx second).

In Programs 5.8 and 5.9, we assume **G** is a 32-bit unsigned variable. Program 5.8 contains two separate if-then structures involving testing for equal or not equal. It will call **GEqual7** if **G** equals 7, and **GNotEqual7** if **G** does not equal 7. When testing for equal or not equal it doesn't matter whether the numbers are signed or unsigned. However, it does matter if they are 8-bit or 16-bit. To convert these examples to 16 bits, use the **LDRH R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction. To convert these examples to 8bits, use the **LDRB R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction.

Assembly code	C code
<b>LDR R2, =G ; R2 = &amp;G</b>	<b>uint32_t G;</b>
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G == 7){</b>
<b>CMP R0, #7 ; is G == 7 ?</b>	<b>    GEqual7();</b>
<b>BNE next1 ; if not, skip</b>	<b>}</b>
<b>BL GEqual7 ; G == 7</b>	
<b>next1</b>	
<b>LDR R2, =G ; R2 = &amp;G</b>	
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G != 7){</b>
<b>CMP R0, #7 ; is G != 7 ?</b>	<b>    GNotEqual7();</b>
<b>BEQ next2 ; if not, skip</b>	<b>}</b>
<b>BL GNotEqual7 ; G != 7</b>	
<b>next2</b>	

## Program 5.8. Conditional structures that test for equality (this works with signed and unsigned numbers).

When testing for greater than or less than, it does matter whether the numbers are signed or unsigned. Program 5.9 contains four separate unsigned if-then structures. In each case, the first step is to bring the first value in R0; the second step is to compare the first value with a second value; and the third step is to execute an unsigned branch **Bxx**. The branch will occur if the first unsigned value is **xx** the second unsigned value.

Assembly code	C code
<b>LDR R2, =G ; R2 = &amp;G</b>	<b>uint32_t G;</b>
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G &gt; 7){</b>
<b>CMP R0, #7 ; is G &gt; 7?</b>	<b>    GGreater7();</b>
<b>BLS next1 ; if not, skip</b>	<b>}</b>
<b>BL GGreater7 ; G &gt; 7</b>	
<b>next1</b>	
<b>LDR R2, =G ; R2 = &amp;G</b>	
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G &gt;= 7){</b>
<b>CMP R0, #7 ; is G &gt;= 7?</b>	<b>    GGreaterEq7();</b>
<b>BLO next2 ; if not, skip</b>	<b>}</b>
<b>BL GGreaterEq7 ; G &gt;= 7</b>	
<b>next2</b>	
<b>LDR R2, =G ; R2 = &amp;G</b>	
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G &lt; 7){</b>
<b>CMP R0, #7 ; is G &lt; 7?</b>	<b>    GLess7();</b>
<b>BHS next3 ; if not, skip</b>	<b>}</b>
<b>BL GLess7 ; G &lt; 7</b>	
<b>next3</b>	
<b>LDR R2, =G ; R2 = &amp;G</b>	
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G &lt;= 7){</b>
<b>CMP R0, #7 ; is G &lt;= 7?</b>	<b>    GLessEq7();</b>
<b>BHI next4 ; if not, skip</b>	<b>}</b>
<b>BL GLessEq7 ; G &lt;= 7</b>	
<b>next4</b>	

## Program 5.9. Unsigned conditional structures.

It will call **GGreater7** if **G** is greater than 7 , **GGreaterEq7** if **G** is greater than or equal to 7 , **GLess7** if **G** is less than 7 , and **GLessEq7** if **G** is less than or equal to 7 . When comparing unsigned values, the instructions **BHI** **BLO** **BHS** and **BLS** should follow the subtraction or comparison instruction. A conditional if-then is implemented by bringing the first number in a register, subtracting the second number, then using the branch instruction with complementary logic to skip over the body of the if-then. To convert these examples to 16 bits, use the **LDRH R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction. To convert these examples to 8bits, use the **LDRB R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction.

**Example 5.1.** Assuming G1 is 8-bit unsigned, write software that sets G2=1 if G1 is greater than 100.

**Solution:** First, we draw a flowchart describing the desired algorithm, see Figure 5.6. Next, we restate the conditional as “skip over if G1 is less than or equal to 100”. To implement the assembly code we bring G1 into Register R0 using **LDRB** to load an unsigned byte, subtract 100, then branch to next if G1 is less than or equal to 100, as presented in Program 5.10. We will use an unsigned conditional branch because the data format is unsigned.

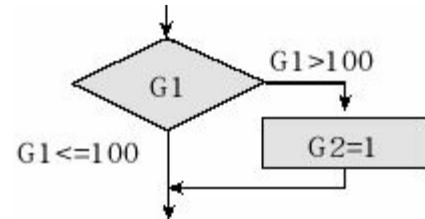


Figure 5.6. Flowchart of an if-then structure.

<b>LDR R2, =G1 ; R2 = &amp;G1</b> <b>LDRB R0, [R2] ; R0 = G1</b> <b>CMP R0, #100 ; is G1 &gt; 100?</b> <b>BLS next ; if not, skip to end</b> <b>MOV R1, #1 ; R1 = 1</b> <b>LDR R2, =G2 ; R2 = &amp;G2</b> <b>STRB R1, [R2] ; G2 = 1</b> <b>next</b>	<b>uint8_t G1, G2;</b> <b>if(G1&gt;100){</b> <b>    G2 = 1;</b> <b>}</b>
--	---

Program 5.10. An unsigned if-then structure. LDRB used because 8-bit, BLS used because it is unsigned.

**Checkpoint 5.5:** Assume you have an 8-bit unsigned global variable **N** . Write assembly code that implements **if(N==25)isEqual()**;

**Checkpoint 5.6:** Assume **H1** and **H2** are two 16-bit unsigned variables. Write assembly code that implements **if(H1==H2)isEqual()**;

Program 5.11 contains four separate signed if-then structures, where **G** is signed 32 bits. In each case, the first step is to bring the first value in R0; the second step is to compare the first value with a second value; and the third step is to execute a signed branch **Bxx** . The branch will occur if the first signed value is **xx** the second signed value.

Assembly code	C code
<b>LDR R2, =G ; R2 = &amp;G</b>	<b>int32_t G;</b>
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G &gt; 7){</b>
<b>CMP R0, #7 ; is G &gt; 7?</b>	<b>    GGreater7();</b>
<b>BLE next1 ; if not, skip</b>	<b>}</b>
<b>BL GGreater7 ; G &gt; 7</b>	
<b>next1</b>	
<b>LDR R2, =G ; R2 = &amp;G</b>	
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G &gt;= 7){</b>
<b>CMP R0, #7 ; is G &gt;= 7?</b>	<b>    GGreaterEq7();</b>
<b>BLT next2 ; if not, skip</b>	<b>}</b>
<b>BL GGreaterEq7 ; G &gt;= 7</b>	
<b>next2</b>	
<b>LDR R2, =G ; R2 = &amp;G</b>	
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G &lt; 7){</b>
<b>CMP R0, #7 ; is G &lt; 7?</b>	<b>    GLess7();</b>
<b>BGE next3 ; if not, skip</b>	<b>}</b>
<b>BL GLess7 ; G &lt; 7</b>	
<b>next3</b>	
<b>LDR R2, =G ; R2 = &amp;G</b>	
<b>LDR R0, [R2] ; R0 = G</b>	<b>if(G &lt;= 7){</b>
<b>CMP R0, #7 ; is G &lt;= 7?</b>	<b>    GLessEq7();</b>
<b>BGT next4 ; if not, skip</b>	<b>}</b>
<b>BL GLessEq7 ; G &lt;= 7</b>	
<b>next4</b>	

## Program 5.11. Signed conditional structures.

Similar to Program 5.9, Program 5.11 will call **GGreater7** if **G** is greater than 7 , **GGreaterEq7** if **G** is greater than or equal to 7 , **GLess7** if **G** is less than 7 , and **GLessEq7** if **G** is less than or equal to 7 . When comparing signed values, the instructions **BGT** **BLT** **BGE** and **BLE** should follow the subtraction or comparison instruction. A conditional if-then is implemented by bringing the first number in a register, subtracting the second number, then using the branch instruction with complementary logic to skip over the body of the if-then. To convert these examples to 16 bits, use the **LDRSH R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction. To convert these examples to 8bits, use the **LDRSB R0,[R2]** instruction instead of the **LDR R0, [R2]** instruction.

**Checkpoint 5.7:** When implementing **if(N>25)isGreater();** why is it important to know if **N** is signed or unsigned?

**Common error:** It is an error to use an unsigned conditional branch when comparing two signed values. Similarly, it is a mistake to use a signed conditional branch when comparing two unsigned values.

**Observation:** One cannot directly compare a signed number to an unsigned number. The proper method is to first convert both numbers to signed numbers of a higher precision and then compare.

---

**Example 5.2.** Redesign the Example 5.1 code assuming G1 is 8-bit signed.

**Solution:** We can use the same flowchart shown previously in Figure 5.6. Again we bring G1 into Register R0 this time using **LDRSB** to load a signed byte, subtract 100, then branch to next if G1 is less than or equal to 100, as presented in Program 5.12. However, we will use a signed conditional branch because the data format is signed.

<b>LDR R2, =G1 ; R2 = &amp;G1</b>	<b>int8_t G1, G2;</b>
<b>LDRSB R0, [R2] ; R0 = G1 (signed)</b>	<b>if(G1&gt;100){</b>
<b>CMP R0, #100 ; is G1 &gt; 100?</b>	<b>G2 = 1;</b>
<b>BLE next ; if not, skip to end</b>	<b>}</b>
<b>MOV R1, #1 ; R1 = 1</b>	
<b>LDR R2, =G2 ; R2 = &amp;G2</b>	
<b>STRB R1, [R2] ; G2 = 1</b>	
<b>next</b>	

---

Program 5.12. A signed if-then structure LDRSB is a signed 8-bit load. BLE is a signed branch.

Notice that the C code for Program 5.9 looks similar to Program 5.11, and the C code for Program 5.10 looks similar to Program 5.12. This is because the compiler knows the type of variables G1 and G2; therefore, it knows whether to utilize unsigned or signed branches. Unfortunately, this similarity can be deceiving. When writing code whether it be assembly or C, you still need to keep track of whether your variables are signed or unsigned. Furthermore, when comparing two objects, they must have comparable types. E.g., “Which is bigger, 2 unsigned apples or -3 signed dollars?” The compiler does not seem to reject comparisons between signed and unsigned variables as an error. However, I recommend that you do not compare a signed variable to an unsigned variable. When comparing objects of different types, it is best to first convert both objects to the same format, and then perform the comparison. Conversely, we see that all numbers are converted to 32 bits before they are compared. This means there is no difficulty comparing variables of differing precisions: e.g., 8-bit, 16-bit, and 32-bit as long as both are signed or both are unsigned.

We can use the unconditional branch to add an **else** clause to any of the previous **if then** structures. A simple example of an unsigned conditional is illustrated in the Figure 5.7 and presented in Program 5.13. The first three lines test the condition **G1 > G2**. If **G1 > G2**, the software branches to **high**. Once at **high**, the software calls the **isGreater** subroutine then continues. Conversely, if **G1≤G2**, the software does not branch and the **isLessEq** subroutine is executed. After executing the **isLessEq** subroutine, there is an unconditional branch, so that only one and not both subroutines are called.

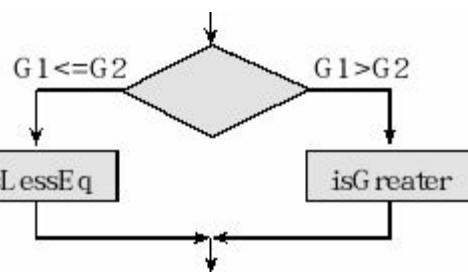


Figure 5.7. Flowchart of an if-then-else structure.

<pre> LDR R2, =G1 ; R2 = &amp;G1 LDR R0, [R2] ; R0 = G1 LDR R2, =G2 ; R2 = &amp;G2 LDR R1, [R2] ; R1 = G2 CMP R0, R1 ; is G1 &gt; G2 ? BHI high ; if so, skip to high low BL isLessEq ; G1 &lt;= G2 B next ; unconditional high BL isGreater ; G1 &gt; G2 next </pre>	<pre> uint32_t G1,G2; if(G1&gt;G2){     isGreater(); } else{     isLessEq(); } </pre>
---	---

Program 5.13. An unsigned if-then-else structure (unsigned 32-bit).

**Checkpoint 5.8:** Assume you have a 16-bit signed global variable M. Write assembly code that implements **if(M > 1000) isGreater(); else isLess();**

The **selection operator** takes three input parameters and yields one output result. The format is

**Expr1 ? Expr2 : Expr3**

The first input parameter is an expression, **Expr1**, which yields a Boolean(0 for false, not zero for true). **Expr2** and **Expr3** return values that are regular numbers. The selection operator will return the result of **Expr2** if the value of **Expr1** is true, and will return the result of **Expr3** if the value of **Expr1** is false. The type of the expression is determined by the types of **Expr2** and **Expr3**. If **Expr2** and **Expr3** have different types, then promotion is applied. The left and right side perform identical functions. If **b** is 1 set **a** equal to 10, otherwise set **a** to 1.

<pre> a = (b==1) ? 10 : 1; </pre>	<pre> if(b==1)     a=10; else     a=1; </pre>
-----------------------------------	---

### 5.3.3. switch Statements

Switch statements provide a non-iterative choice between any number of paths based on specified conditions. They compare an expression to a set of constant values. Selected statements are then executed depending on which value, if any, matches the expression. The expression between the parentheses following **switch** is evaluated to a number and compared one by one to the explicit cases. Figure 5.8 draws a flowchart describing Program 5.4, which performs one output each time the function **OneStep** is called. The **break** causes execution to exit the switch statement. The **default** case is run if none of the explicit case statements match. The operation of the switch statement in Program 5.4 performs this list of actions:

If **Last** is equal to 10, then **theNext** is set to 9.

If **Last** is equal to 9, then **theNext** is set to 5.

If **Last** is equal to 5, then **theNext** is set to 6.

If **Last** is equal to 6, then **theNext** is set to 10.

If **Last** is not equal any of the above, then **theNext** is set to 10.

When using **break**, only the first matching case will be invoked. In other words, once a match is found, no other tests are performed. The body of the switch is not a normal compound statement since local declarations are not allowed in it or in subordinate blocks.

Assume the output port is connected to a stepper motor, and the motor has 24 steps per rotation. Calling **OneStep** will cause the motor to rotate by exactly 15 degrees. 15 degrees is 360 degrees divided by 24. For more information on stepper motors, see Example 6.1, Program 7.6, and Section 8.7.

```
uint32_t Last=10;
void OneStep(void){
    uint32_t next;
    switch(Last){
        case 10: next = 9; break;
        case 9:  next = 5; break;
        case 5:  next = 6; break;
        case 6:  next = 10; break;
        default: next = 10;
    }
    GPIO_PORTD_DATA_R = next;
    Last = next;
}
```

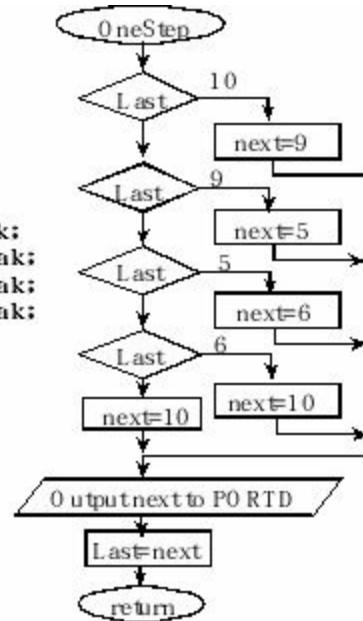


Figure 5.8. The switch statement is used to make multiple comparisons.

Program 5.14 converts an ASCII character to the equivalent decimal value. This example of a switch statement shows that the multiple tests can be performed for the same condition.

```
uint8_t Convert(char letter){
    uint8_t digit;
    switch (letter) {
        case 'A':
```

```

case 'B':
case 'C':
case 'D':
case 'E':
case 'F':
    digit = letter+10-'A'; break;
case 'a':
case 'b':
case 'c':
case 'd':
case 'e':
case 'f':
    digit = letter+10-'a'; break;
default:
    digit = letter-'0';
}
return digit;
}

```

Program 5.14. A switch statement is used to convert an ASCII character to numeric value.

## 5.3.4. While Loops

Quite often the microcomputer is asked to wait for events or to search for objects. Both of these operations are solved using the **while** or **do-while** structure. A simple example of while loop is illustrated in the Figure 5.9 and presented in Program 5.15. Assume G1 and G2 are unsigned 32-bit variables. The operation is defined by the C code

```
while(G2 > G1){Body();}
```

specifies the function **Body()** will be executed over and over as long as  $G2 > G1$ .

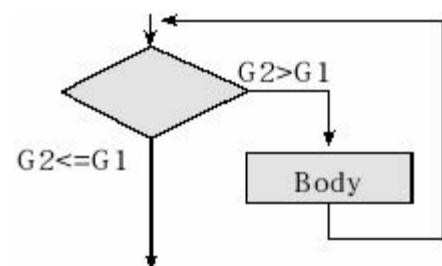


Figure 5.9. Flowchart of a while structure. Execute **Body()** over and over until  $G2 \leq G1$ .

Program 5.15 begins with a test of **G2>G1**. If **G2≤G1** then the body of the while loop is skipped. The unconditional branch( **B loop** )after the body causes **G2** and **G1** to be tested again. In this way, the body is executed repeatedly until **G2≤G1** .

<pre> LDR R4, =G1 ; R4 = &amp;G1 LDR R5, =G2 ; R5 = &amp;G2 loop LDR R0, [R5] ; R0 = G2     LDR R1, [R4] ; R1 = G1     CMP R0, R1 ; is G2 &lt;= G1?     BLS next ; if so, skip to next     BL Body ; body of the loop     B loop next </pre>	<pre> uint32_t G1,G2;  while(G2 &gt; G1){     Body(); } </pre>
--	--

Program 5.15. A while loop structure.

**Observation:** The body of a while loop may execute zero or more times, but the body of a do-while loop is executed at least once.

One of the conventions when writing assembly is whether or not subroutines should save registers. According to AAPCS, we will allow subroutines to freely modify R0–R3 and R12. Conversely, if a subroutine wishes to use R4 through R11, it will preserve the values using the stack. Similarly, if the subroutine wishes to use LR (e.g., to call another subroutine) it must save and restore LR. AAPCS requires us to push and pop an even number of registers. AAPCS guarantees the address pointers in R4 and R5 only need to be set once in Program 5.15, because we can assume that the call to **Body()** will not corrupt them. However, since the variables themselves are held in RAM and may therefore be changed by some other piece of code, it does make sense to reload the values of the variables each time through the loop.

**Checkpoint 5.9:** Assume you have a 16-bit unsigned global variable **N**. Write assembly code that implements **while(N!=25){body();}**

### 5.3.5. Do-while Loops

A do-while loop performs the body first, and the test for completion second. It will execute the body at least once. Assume **PF1** and **PA5** are definitions of I/O pins using bit-specific addressing. Program 5.16 will toggle the output PF1 as long as input PA5 is low.

<pre> LDR R1, =PF1 ; R1 = &amp;PF1 LDR R5, =PA5 ; R5 = &amp;PA5 loop LDR R0, [R1]     EOR R0, #2 ; toggle bit 1     STR R0, [R1]     LDR R2, [R5] ; R2 = PA5     ANDS R2, #0x20 ; bit 5 set?     BEQ loop ; spin while low next </pre>	<pre> // toggle PF1 while PA5 low do{     PF1 = PF1 ^ 0x02; } while((PA5 &amp; 0x20) == 0); </pre>
--	--

Program 5.16. A do-while loop structure.

## 5.3.6. For Loops

A **for-loop** control structure is a special case of the while loop. For loops can iterate up or down. To show the similarity between the while loop and for loop these two C functions are identical. The <init> code is executed once. The <test> code returns a true/false and is tested before each iteration. The <body> and <end> codes are executed each iteration.

<pre>&lt;init&gt;; while(&lt;test&gt;){     &lt;body&gt;;     &lt;end&gt;; }</pre>	<pre>for(&lt;init&gt;; &lt;test&gt;; &lt;end&gt;){     &lt;body&gt;; }</pre>
--	--

For-loops are a convenient way to perform repetitive tasks. As an example, we write code that calls **Process()** 100 times. Two possible solutions are illustrated in Figure 5.10. The solution on the left starts at 0 and counts up to 100, while the solution on the right starts at 100 and counts down to 0. The first field is the starting task (e.g., **i=0** ). The next field specifies the conditions with which to continue execution (e.g., **i<100** ), and the last field is the operation to perform after each interactions (e.g., **i++** ). Similar to a while loop, the test occurs before each execution of the body.

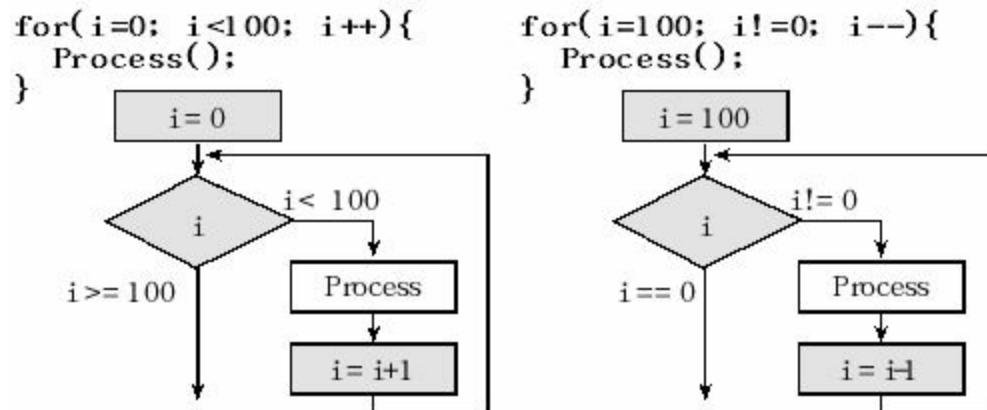


Figure 5.10. Two flowcharts of a for - loop structure.

The count-up implementation places the loop counter in the Register R4, as shown in Program 5.17. As mentioned earlier, we assume the subroutine **Process** preserves the value in R4.

<pre>MOV R4, #0 ; R4 = 0 loop CMP R4, #100 ; index &gt;= 100?     BHS done ; if so, skip to done     BL Process ; process function     ADD R4, R4, #1 ; R4 = R4 + 1     B loop done</pre>	<pre>for(i=0; i&lt;100; i++){     Process(); }</pre>
---	--

Program 5.17. A simple for-loop.

If we assume the body will execute at least once, we can execute a little faster, as shown in Program 5.18, by counting down. Counting down is one instruction faster than counting up.

<b>MOV R4, #100 ; R4 = 100</b>	<b>MOV R4, #0 ; R4 = 0</b>
<b>loop BL Process ; body</b>	<b>loop BL Process ; body</b>
<b>SUBS R4, R4, #1 ; R4 = R4-</b>	<b>ADD R4, R4, #1 ; R4 = R4+1</b>
<b>1</b>	<b>CMP R4, #100 ; done?</b>
<b>BNE loop</b>	<b>BLO loop ; if not,repeat</b>
<b>done</b>	

Program 5.18. Optimized for-loops.

## 5.4. \*Assembly Macros

A macro is a template for a sequence of instructions. The macro definition includes a name and a code sequence. This name becomes the mnemonic by which the macro is subsequently invoked. Invoking a macro means replacing the macro name with its code sequence, analogous to a copy-paste operation in the editor. Macros are like subroutines in the sense we use them to encapsulate operations we wish to perform as a single higher-level function. With subroutines, we call a function at run time by moving the PC into the LR and jumping to the code that defines the function. The return from subroutine also occurs at run time, by moving return address from LR back into the PC. Invoking a macro, on the other hand, occurs at assembly time. When the assembler sees a macro in your code, it performs a character substitution (copy/paste), replacing the macro name with the macro definition.

The syntax used in this section is compatible with ARM Keil™ uVision® compiler. The macro definition may also include macro arguments. A macro definition may contain any code or directive except nested macro definitions. Invoking previously defined macros is allowed inside a macro definition. The code sequence of the macro is inserted in the source file at the position where the macro is invoked. To invoke a macro, write the macro name in the operation field of a source statement. Place the arguments, if any, in the operand field. The macro may contain conditional assembly directives that cause the assembler to produce parameter-dependent variations of the macro definition. The definition of a macro consists of three parts:

The header statement, a **MACRO** directive with a label that names the macro.  
The code sequence, with argument placeholders as needed.  
The **MEND** directive, terminating the macro definition.

We can use macros to define what looks like new instructions. For example, this macro creates a new assembly instruction that performs the modulus. MOD = DIVIDEND % DIVISOR

**MACRO**

```
UMOD $Mod,$Divnd,$Divsr ;MOD,DIVIDEND,DIVISOR
UDIV $Mod,$Divnd,$Divsr ;Mod = DIVIDEND/DIVISOR
MUL $Mod,$Mod,$Divsr ;Mod = DIVISOR*(DIVIDEND/DIVISOR)
SUB $Mod,$Divnd,$Mod ;Mod = DIVIDEND-DIVISOR*(DIVIDEND/DIVISOR)
MEND
```

The registers **MOD** and **DIVIDEND** must not be the same register. We invoke this macro in our code simply by placing **UMOD** as an operation in our program.

Our code	Result created by assembler
<b>UMOD R3,R0,R1 ;R3=R0%R1</b>	<b>UDIV R3,R0,R1</b>
	<b>MUL R3,R3,R1</b>
	<b>SUB R3,R0,R3</b>
<b>STR R3,[R7]</b>	<b>STR R3,[R7]</b>
<b>UMOD R3,R2,R4 ;R3=R2%R4</b>	<b>UDIV R3,R2,R4</b>
	<b>MUL R3,R3,R4</b>

```
STR R3,[R6]
```

```
SUB R3,R2,R3
```

```
STR R3,[R7]
```

**Observation:** Subroutines optimize memory space with a cost of execution speed. There is one copy of the code but requires a **BL** to call and a **BX LR** to return.

**Observation:** Macros optimize execution speed with a cost of memory space. There is one copy of the code for each invocation but does not require call or return instructions. If a macro is invoked once, then it will execute faster and require less memory as compared to defining the operation as a subroutine.

---

**Example 5.3.** Redesign the I/O driver for an LED on PF2, originally solved in Program 4.3b.

**Solution:** We will leave the initialization as a function, because it is only called once. However, we will rewrite the three operations that output to the pin. The initialization will define PE0 as an output. The subroutine solution, shown on the left of Program 5.19, is repeated from Program 4.3b. These macros do not have any parameters. Assume **PF2** is a bit-specific label, **PF2 EQU 0x40025010**.

#### LED\_Off

```
LDR R1, =PF2  
MOV R0, #0  
STR R0, [R1]  
BX LR
```

#### MACRO

```
LED_Off  
LDR R1, =PF2  
MOV R0, #0  
STR R0, [R1]  
MEND
```

#### LED\_On

```
LDR R1, =PF2  
MOV R0, #1  
STR R0, [R1]  
BX LR
```

#### MACRO

```
LED_On  
LDR R1, =PF2  
MOV R0, #1  
STR R0, [R1]  
MEND
```

#### LED\_Toggle

```
LDR R1, =PF2  
LDR R0, [R1]  
EOR R0, R0, #1  
STR R0, [R1]  
BX LR
```

#### MACRO

```
LED_Toggle  
LDR R1, =PF2  
LDR R0, [R1]  
EOR R0, R0, #1  
STR R0, [R1]  
MEND
```

---

Program 5.19. I/O port drivers using subroutines and macros.

**Checkpoint 5.10:** What advantage does the macro solution have over the subroutine solution?

Sometimes a macro is only one line long. This example creates a new assembly instruction that performs multiply by 7. This approach can be used to multiply by any constant in the form of  $2^n \pm 1$ . For multiply by 7, we use the fact that if x is a variable, then  $7x = (x \ll 3) - x$ .

### MACRO

```
MUL7 $Rd,$Rn  
RSB $Rd,$Rn,$Rn,LSL #3  
MEND
```

We invoke this macro in our code simply by placing **MUL7** as an operation in our program. Notice that this macro generates much smaller object code and runs much faster than using the **MUL** instruction.

Our code	Result created by assembler
<b>MUL7 R3,R2 ;R3=7*R2</b>	<b>RSB R3,R2,R2,LSL #3</b>
STR R3,[R7]	STR R3,[R7]
<b>MUL7 R4,R4 ;R4=7*R4</b>	<b>RSB R4,R4,R4,LSL #3</b>
STR R4,[R6]	STR R4,[R7]

This approach can also be used to multiply by any constant in the form of  $1 \pm 2^{-n}$ . For example, to multiply by 15/16 we implement  $x - (x \gg 4)$ . This macro is unsigned multiply by 15/16.

### MACRO

```
MUL15_16 $Rd,$Rn  
SUB $Rd,$Rn,$Rn,LSR #4  
MEND
```

# 5.5. \*Recursion

A **recursive** subroutine is one that calls itself. Each time the subroutine is started a new instantiation occurs. There is a unique set of parameters, registers, and local variables for each instantiation. The stack is a convenient way to separate the parameters and variables of one instantiation from another. In order for the recursive function to finish, there must be a situation where a direct result is generated, which is called the **end condition**. The body of a recursive relation defines the operation in terms of calls to itself that are closer to the end condition(s). For example, the factorial has two possibilities

$$\text{Fact}(1) = 1$$

end condition, base case, or anchor case  
recursion

$$\text{Fact}(n) = n \cdot \text{Fact}(n-1) \text{ if } n > 1$$

Program 5.20 shows two implementations of factorial. The one on the top uses iteration, and the one on the bottom uses recursion. It is usually the case that a recursive algorithm can be rewritten in iterative form. Nevertheless, sometimes it is more convenient to implement the algorithm in recursive form.

```
; iterative implementation (22 bytes)
; Input: R0 is n
; Output: R0 is Fact(n)
; Assumes: R0 <= 12 (13! overflows)
Fact MOV R1, #1 ; R1 = 1 = total
loop CMP R0, #1 ; is n (R0) <= 1?
    BLS done ; if so, skip to done
    MUL R1, R0, R1 ; total = total*n
    SUB R0, R0, #1 ; n = n - 1
    B loop
done MOV R0, R1 ; total = Fact(n)
BX LR
; recursive implementation (30 bytes)
; Input: R0 is n
; Output: R0 is Fact(n)
; Assumes: R0 <= 12 (13! overflows)
Fact CMP R0, #1 ; is n (R0) <= 1?
    BLS endcase ; if so, to endcase
    PUSH {R0, LR} ; save R0 and LR
    SUB R0, R0, #1 ; n = n - 1
    BL Fact ; R0 = Fact(n-1)
    POP {R1, LR} ; restore R1, LR
    MUL R0, R0, R1 ; R0 = n*Fact(n-1)
    BX LR ; normal return
endcase
MOV R0, #1 ; R0 = 1
BX LR ; end case return
```

```
// iterative implementation
// Assumes: n <= 12
uint32_t Fact(uint32_t n){
    uint32_t r;
    r = 1;
    for(; n>1; n--){
        r = r*n;
    }
    return r;
}

// recursive implementation
// Assumes: n <= 12
uint32_t Fact(uint32_t n){
    if(n <= 1){ // end condition
        return 1;
    }
    return n*Fact(n-1); // recursion
}
```

Program 5.20. Iterative and recursive implementations of factorial.

Table 5.2 shows the execution time in cycles for these two assembly implementations. Notice that the recursive implementation is slightly longer, and the execution speed is slightly slower.

Input	Iterative	Recursive
1	11	9
2	18	23
3	25	37
4	32	51
5	39	65

Table 5.2. Execution times in cycles for Program 5.20 running on LM3S1968 including all branches.

Recursive Fact(0) and Fact(1) do not push anything onto the stack. Fact(n) ( $2 \leq n \leq 12$ ) pushes  $8*(n-1)$  bytes (32-bit n and 32-bit LR per call, above 1). The iterative also uses the stack, creating the digits right to left pushing them onto the stack, and then popping off the digits in the correct order.

**Checkpoint 5.11:** How many stack bytes are required for each instantiation of **Fact** ? How much stack space is required to execute **Fact(5)** ?

The power function,  $y = x^n$ , can be designed in a recursive manner. We will assume the inputs x and n are 32-bit unsigned integers. We begin with some simple observations that may lead to the base case. At this point we write them all down and see later if any can be used:

$$0^n = 0 \text{ for all } n$$

$$1^n = 1 \text{ for all } n$$

$$x^0 = 1 \text{ for all } x \text{ not equal to } 0$$

$$x^1 = x \text{ for all } x$$

$$x^2 = x*x \text{ for all } x$$

Next we try to rewrite the function in terms of itself so that the problem becomes simpler

$$x^n = x*x*x*x...*x \quad (n \text{ times, this is the iterative approach})$$

$$x^n = x^{n+1}/x \quad (\text{This is more complex})$$

$$x^n = x*x^{n-1} \quad (\text{This is simpler})$$

The last rewrite is written in a recursive manner and will eventually lead to the  $x^0 = 1$  base.

```
uint32_t power (uint32_t x, uint32_t n){
    if(x == 0) return 0;
    if(n == 0) return 1;
    return x*power(x,n - 1);
}
```

The power function has two base cases. The first one is needed to handle the power(0,0) possibility. For x not zero, the second base case will become true eventually. If n=10, then the recursive function will call itself 10 times. Fibonacci can be defined recursively

```
uint32_t fibonacci(uint32_t n){
    if(n == 0) return 0;
    if(n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);}
```

}

**Example 5.4.** You are given a subroutine, **OutChar**, which outputs one ASCII character. Design a function that outputs a 32-bit unsigned integer.

**Solution:** We will solve this two ways, iteratively and recursively. As always, we ask “what is our starting point?”, “how do we make progress?”, and “when are we done?” The input, N, is a 32-bit unsigned number (in R0), and we are done when 1 to 10 ASCII characters are displayed, representing the value of N. Figure 5.11 demonstrates the successive refinement approach to solving this problem iteratively. The iterative solution has three phases: initialization, creation of digits, and output of the ASCII characters. The digits are created from the remainders occurring by dividing the input, N by 10. To get all the digits we divide by 10 until the quotient is 0. Because the digits are created in the opposite order, each digit will be pushed on the stack during the creation phase and popped off the stack during the output stage. The counter is needed so the output stage knows how many digits to pop from the stack.

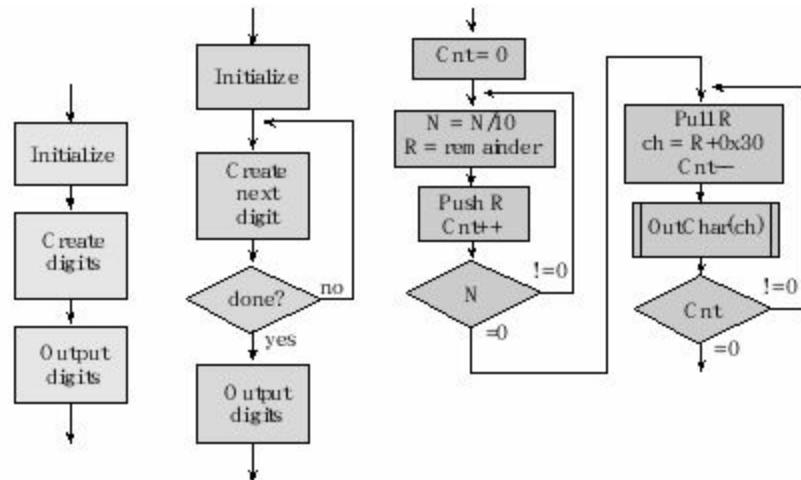
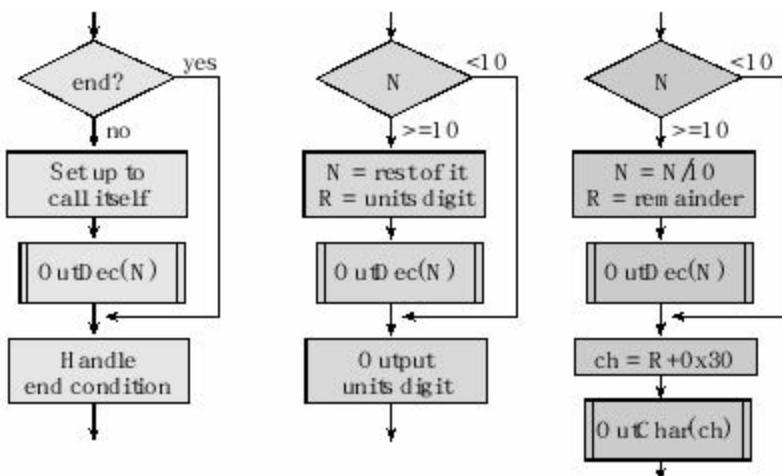


Figure 5.11. Successive refinement method for the iterative approach.

Figure 5.12 demonstrates the successive refinement approach to solving this problem recursively. Most recursive functions first check for the end condition. If the end condition is true, it handles the simple case directly. If the end condition is not true, it simplifies the problem (in this case  $N = N/10$ ) and calls itself. Just like the iterative solution, the digits (calculated as R) are calculated in reverse order, and the stack is used to save the intermediate results, so the digits are displayed in proper order.



## Figure 5.12. Successive refinement method for the recursive approach.

Program 5.21 shows two assembly language implementations of this 32-bit output decimal function. The iteration solution actually has two loops; the first loop determines the digits in opposite order, and the second loop outputs the digits in proper order. The recursive solution also uses the stack to calculate the least significant digit first, but output the most significant digit first. There is no fundamental rule that states which is better iteration or recursion. A good programmer has both in his or her toolbox, and uses whichever is easier to understand and easier to debug.

```
; iterative implementation
; uses up to 48 bytes of stack
; Input: R0 is 32-bit number N
; Output: none
; Modifies: R1, R2, R3
OutUDec
    PUSH {R4, LR}
    MOV R2, #10 ; R2 = 10 = divisor
    MOV R4, #0 ; R4 = 0 = cnt
ODloop
    UDIV R3, R0, R2 ; R3 = N/10
    MUL R1, R3, R2 ; R1 = N/10*10
    SUB R1, R0, R1 ; R1 = N%10
    PUSH {R1} ; save value
    ADD R4, R4, #1 ; cnt = cnt + 1
    MOVS R0, R3 ; R0 N = N/10
    CMP R0, #0 ; is N == 0?
    BNE ODloop ; if not continue
ODout POP {R0} ; restore into R0
    ADD R0, R0, #'0'; convert ASCII
    BL OutChar ; print character
    SUBS R4, R4, #1 ; cnt = cnt - 1
    CMP R4, #0 ; is cnt == 0?
    BNE ODout ; if not continue
    POP {R4, LR} ; restore
    BX LR ; return
```

```
; recursive implementation
; uses a maximum of 76 bytes of stack
; Input: R0 is 32-bit number N
; Output: none
; Modifies: R1, R2, R3
OutUDec
    PUSH {LR}
    CMP R0, #10 ; character < 10?
    BLO ODend ; if so, end
    MOV R2, #10 ; R2 = divisor
    UDIV R3, R0, R2 ; R3 = N/10
    MUL R1, R3, R2 ; R1 = N/10*10
    SUB R1, R0, R1 ; R1 = N%10
    PUSH {R1} ; save
    MOVS R0, R3 ; N = N/10
    BL OutUDec ; OutUDec(N/10)
    POP {R0} ; restore into R0
ODend
    ADD R0, R0, #'0'; convert ASCII
```

```
// iterative method
void OutUDec(uint32_t n){
    uint32_t cnt=0;
    char buffer[11];
    do{
        buffer[cnt] = n%10;// digit
        n = n/10;
        cnt++;
    }
    while(n); // repeat until n==0
    for(; cnt; cnt--){
        OutChar(buffer[cnt-1]+'0');
    }
}
```

```
// recursive method
void OutUDec(uint32_t n){
    if(n >= 10){
        OutUDec(n/10); // ms digits
        n = n%10; // n is 0-9
    }
    OutChar(n+'0');
}
```

```
BL OutCh    ; print character  
POP {LR}    ; restore  
BX LR      ; return
```

## Program 5.21. Iterative and recursive implementations of output decimal.

To observethe execution of the recursive implementation of **OutUDec** , we can place a breakpoint on the first line and observe the stack and Register R0.

---

**Observation:** In general, recursive algorithms are shorter to write, but require additional stack space.

# 5.6. Writing Quality Software

## 5.6.1. Style Guidelines

The objective of this section is to present style rules when developing software. This set of rules is meant to guide not control. In other words, they serve as general guidelines rather than fundamental law. Choosing names for variables and functions involves creative thought, and it is intimately connected to how we feel about ourselves as programmers. Of the policies presented in this section, naming conventions may be the hardest habit for us to break. The difficulty is that there are many conventions that satisfy the “easy to understand” objective. Good names reduce the need for documentation. Poor names promote confusion, ambiguity, and mistakes. Poor names can occur because code has been copied from a different situation and inserted into our system without proper integration (i.e., changing the names to be consistent with the new situation.) They can also occur in the cluttered mind of a second-rate programmer, who hurries to deliver software before it is finished.

Names should have meaning. If we observe a name away from the place where it is defined, the meaning of the object should be obvious. The object **TxFifo** is clearly the transmit first in first out circular queue. The function **LCD\_OutString** will output a string to the LCD.

Avoid ambiguities. Don't use variable names in our system that are vague or have more than one meaning. For example, it is vague to use **temp**, because there are many possibilities for temporary data, in fact, it might even mean temperature. Don't use two names that look similar, but have different meanings.

Give hints about the type. We can further clarify the meaning of a variable by including phrases in the variable name that specify its type. For example, **dataPt** **timePt** **putPt** are pointers.

Similarly, **voltageBuf** **timeBuf** **pressureBuf** are data buffers. Other good phrases include **Flag** **Mode** **U L Index Cnt**, which refer to Boolean flag, system state, unsigned 16-bit, signed 32-bit, index into an array, and a counter respectively.

Use the same name to refer to the same type of object. For example, everywhere we need a local variable to store an ASCII character we could use the name **letter**. Another common example is to use the names **i j k** for indices into arrays. The names **V1 R1** might refer to a voltage and a resistance. The exact correspondence is not part of the policies presented in this section, just the fact that a correspondence should exist. Once another programmer learns which names we use for which types of object, understanding our code becomes easier.

Use a prefix to identify public objects. A public variable is shared between two modules. A public function is a function in one module that can be called from another module. An underline character will separate the module name from the function name. Public objects have the underline and private objects do not. As an exception to this rule, we can use the underline to delimit words in all uppercase name (e.g., **MIN\_PRESSURE equ 10**). Functions that can be accessed outside the scope of a module (i.e., public) will begin with a prefix specifying the module to which it belongs. It is poor

style to create public variables, but if they need to exist, they too would begin with the module prefix. The prefix matches the module name containing the object. For example, if we see a function call, **BL LCD\_OutString** we know the public function belongs to the LCD module. Notice the similarity between this syntax (e.g., **LCD\_Init**) and the corresponding syntax we would use if programming the module in C++ (e.g., **LCD.Init()**). Using this convention, we can distinguish public and private objects.

Use upper and lower case to specify the allocation of an object. We will define I/O port addresses and other constants using no lower-case letters, like typing with caps-lock on. In other words, names without lower-case letters refer to objects with fixed values. **TRUE FALSE** and **NULL** are good examples of fixed-valued objects. As mentioned earlier, constant names formed from multiple words will use an underline character to delimit the individual words. E.g., **MAX\_VOLTAGE UPPER\_BOUND\_FIFO\_SIZE**. Permanently allocated variables are global, with a name beginning with a capital letter, but including some lower-case letters. Temporarily allocated variables are called local, and the name will begin with a lower-case letter, and may or may not include upper case letters. Since all functions are permanently allocated, we can start function names with either an upper-case or lower-case letter. Using this convention, we can distinguish constants, globals and locals.

An object's properties (public/private, local/global, constant/variable) are always perfectly clear at the place where the object is defined. The importance of the naming policy is to extend that clarity also to the places where the object is used.

Use capitalization to delimit words. Names that contain multiple words should be defined using a capital letter to signify the first letter of the word. Recall that the case of the first letter specifies whether it is a local or global variable. Some programmers use the underline as a word-delimiter, but except for constants, we will reserve underline to separate the module name from the name of a public object. Table 5.3 overviews the naming convention presented in this section.

Object type	Examples of names that identify type
Constants	<b>CR_SAFE_TO_RUN PORTA_STACK_SIZE START_OF_RAM</b>
Local variables	<b>maxTemperature lastCharTyped errorCnt</b>
Private global variable	<b>MaxTemperature LastCharTyped ErrorCnt RxFifoPt</b>
Public global variable	<b>DAC_MaxTemperature Key_LastCharTyped Network_ErrCnt</b>
Private function	<b>ClearTime wrapPointer InChar</b>
Public function	<b>Timer_ClearTime RxFifo_Put Key_InChar</b>

Table 5.3. Examples of names.

**Checkpoint 5.12:** Just by looking at its name, how can you tell if a function is private or public?

**Checkpoint 5.13:** Just by looking at its name, how can you tell if a variable is local or global?

The Single Entry Point is at the Top. In assembly language, we place a single entry point of a subroutine at the first line of the code. By default, C functions have a single entry point. Placing the entry point at the top provides a visual marker for the beginning of the subroutine.

The Single Exit Point is at the Bottom. Most programmers prefer to use a single exit point as the last line of the subroutine. Some programmers employ multiple exit points for efficiency reasons. In general, we must guarantee the registers, stack, and return parameters are at a similar and consistent state for each exit point. In particular, we must deallocate local variables properly. If you do employ multiple exit points, then you should develop a means to visually delineate where one subroutine ends and the next one starts. You could use one line of comments to signify the start a subroutine and a different line of comments to show the end of it. Program 5.22 employs distinct visual markers to see the beginning and end of the subroutine.

Label Names. Some of the assembly examples used weak label names like “**loop**”, “**done**”, “**end**”, and “**out**”. These names technically work, and they might make sense for very short examples. But they can cause problems when functions are copied from one file to another, resulting in multiple “**loop**” labels. One or both needs to be changed before it even assembles. Labels inside functions can be precededwith the function name, such as “**AbsOK**” or with an underscore “**Abs\_OK**”.

<pre>;-----Abs----- ; Take the absolute value of a number. ; Input: R0 is 32-bit signed number ; Output: R0 is 31-bit absolute value Abs CMP R0, #0 ; is number (R0) &gt;= 0?     BPL AbsOK ; if so, already positive     EOR R0, R0, #0xFFFFFFFF ; invert bits     ADD R0, R0, #1      ; add one AbsOK BX LR      ; return ;-----end of Abs-----</pre>	<pre>*****Abs***** // Input: signed 32-bit // Output: absolute value uint32_t Abs(int32_t n){     if(n&lt;0){         n = -n;     }     return (uint32_t) n; }</pre>
---	--

Program 5.22. Examples that use comments to delineate its beginning and end.

**Observation:** Having the first and last lines of a subroutine be the entry and exit points makes it easier to debug, because it will be easy to place debugging instruments (like breakpoints).

**Common error:** If you place a debugging breakpoint on the last BL LR of a subroutine with multiple exit points, then sometimes the subroutine will return without generating the break.

Write Structured Programs. A structured program is one that adheres to a strict list of program structures, previously defined in Section 1.8and further elaborated in Section 5.2. When we program in C (with the exception of **goto** , which by the way you should only use with extreme care) we are forced to write structured programs due to the syntax of the language. One technique for writing structured assembly language is to adhere to the program structures shown in Figure 1.29. In other words, restrict the assembly language branching to configurations that mimic the software behavior of **if** , **if-else** , **do-while** , **while** , **for** and **switch** . Structured programs are much easier to debug, because execution proceeds only through a limited number of well-defined pathways. When we use well-understood assembly branching structures, then our debugging can focus more on the overall function and less on how the details are implemented.

The Registers Must Be Saved. When working on a software team it is important to establish a rule whether or not subroutines will save/restore registers. Establishing this convention is especially important when a mixture of assembly and high-level language is being used, or if the software project remains active for long periods of time. It is safest to save and restore registers that are modified (most programmers do not save/restore the PSR) and output parameter(s) returned in a register. Exceptions to this rule can be made for those portions of the code where speed is most critical. According to AAPCS, we will preserve R4 through R11, but not preserve R0–R3, R12, or the PSR. Remember to always save the link register if your function calls another function. Stack operations function more efficiently if the SP remains aligned on an 8-byte boundary. We maintain the stack on an 8-byte alignment by pushing and popping an even number of registers. Subsequent function calls over-write the return location for your function, making yours unable to return. It is also important to ensure that the SP is the same on exit from your function as it was at entry.

**Common Error:** If the calling routine expects a subroutine to save/restore registers, and it doesn't, then information will be lost.

**Observation:** If the calling routine does not expect a subroutine to save/restore registers, and it does, then the system executes a little slower and the object code is a little bigger than it could be.

**Common Error:** When a mixture of C and assembly language programs are integrated, then an error may occur if the AAPCS rules are changed because there may be a change in if registers are saved/restored, or how parameters are passed.

Use High-Level Languages Whenever Possible. It may seem odd to have a rule about high-level languages in a section about assembly language programming. It is even odder to make this statement in a book devoted to assembly language programming. In general, we should use high-level languages when memory space and execution speed are less important than portability and maintenance. When execution speed is important, you could write the first version in a high-level language, run a profiler (that will tell you which parts of your program are executed the most), then optimize the sections of code using up the most execution time by writing them in assembly language. If a C language implementation just doesn't run fast enough, you could consider a more powerful compiler or a faster microcomputer.

**Observation:** High-level language programmers who are well acquainted with the underlying assembly language of the machine have a better understanding of how their machine and software work.

Minimize Conditional Branching. Every time software makes a conditional branch, there are two possible outcomes that must be tested (branch or not branch.) In the example shown in Program 5.23, assume we wish to set a 32-bit Flag if Port G bit 7 is true. A flag will be true if it is any nonzero value, and false if it is zero. A conditional branch could be avoided by solving the problem in another way. We will define a PG7 label to be the bit-specific address for this pin (0x40026200).

**Observation:** Software can be made easier to understand by reworking the approach in order to reduce the number of conditional branches.

**Checkpoint 5.14:** If a system has 20 conditional branches, how many potential execution paths might there be through the software?

```

PG7 EQU 0x40026200          // uses conditional branch
; set flag with conditional branch
; Input: none
; Output: none
; Modifies: R1, R2
SetFlagConditional
    LDR R2, =PG7 ; R2 = 0x40026200
    LDR R1, [R2] ; R1 = PG7
    LDR R2, =Flag ; R2 = &Flag
    CMP R1, #0x80 ; is PG7 == 0x80?
    BNE SFCClr
SFCCSet MOV R1, #-1 ; PG7 is high
    STR R1, [R2] ; Flag = -1
    B SFCEnd
SFCCClr MOV R1, #0 ; PG7 is low
    STR R1, [R2] ; Flag = 0
SFCEnd BX LR
; set flag with no conditional branch
; Input: none
; Output: none
; Modifies: R1, R2
SetFlagNoConditional
    LDR R2, =PG7 ; R2 = 0x40026200
    LDR R1, [R2] ; R1 = PG7
    LDR R2, =Flag ; R2 = &Flag
    STR R1, [R2] ; Flag = 0x00 or 0x80
    BX LR

```

Program 5.23. Sometimes we can remove a conditional branch and simplify the program.

## 5.6.2. Comments

Discussion about comments was left for last, because they are the least important aspect involved in writing quality software. It is much better to write well-organized software with simple interfaces having operations so easy to understand that comments are not necessary.

The beginning of every file should include the file name, purpose, hardware connections, programmer, date, and copyright. E.g.,

```

; filename adtest.s
; Test of TM4C123 12-bit ADC
; 1 Hz sampling and output to the serial port
; Last modified 5/19/14 by Jonathan W. Valvano
; Copyright 2014 by Jonathan W. Valvano, valvano@mail.utexas.edu
; You may use, edit, run or distribute this file
; as long as the above copyright notice remains

```

The beginning of every function should include a line delimiting the start of the function, purpose, input parameters, output parameters, and special conditions that apply. The comments at the beginning of the function explain the policies (e.g., how to use the function.) These comments, which are similar to the comments for the prototypes in the header file, are intended to be read by the client. E.g.,

```
;-----UART_InUDec-----  
; Accepts ASCII input from the UART in unsigned decimal format  
; and converts to a 32-bit unsigned number with a maximum of 65535  
; If a number is above 2^32, it truncates without reporting error  
; Backspace will remove last digit typed  
; Inputs: none  
; Outputs: Register R0 is the unsigned 32-bit value
```

Comments can be added to a variable or constant definition to clarify the usage. In particular, comments can specify the units of the variable or constant. For complicated situations, we can use additional lines and include examples. E.g.,

```
V1    SPACE 2 ; voltage at node 1 in mV, range -5000 mV to +5000 mV  
Fs    SPACE 2 ; sampling rate in Hz  
FoundFlag SPACE 1 ; 0 if keyword not yet found, 1 if found  
RunMode  SPACE 1 ; 0, 1, 2, or 3 specifies system mode  
; 0 means idle  
; 1 means startup  
; 2 means active run  
; 3 means stopped
```

Comments can be used to describe complex algorithms. These types of comments are intended to be read by our coworkers. The purpose of these comments is to assist in changing the code in the future, or applying this code into a similar but slightly different application. Comments that restate the function provide no additional information, and actually make the code harder to read. Examples of bad comments include:

```
ADD R0,#1 ; add one to R0  
MOV R1,#0 ; set R1 to 0
```

Good comments explain why the operation is performed, and what it means:

```
ADD R0,#1 ; maintain elapsed time in msec  
MOV R1,#0 ; switch to idle mode because no more data is available
```

We can add spaces so the comment fields line up. We should avoid tabs because they often do not translate well from one computer to another. In this way, the software is on the left and the comments can be read on the right.

I taught a large programming class one semester, and being an arrogant and lazy fellow, I thought I could write a grading program that accepts the students' programming assignments and automatically generates and records their grades. The second step will be to design a Massive Open Online Class, MOOC, on edX, and then I could teach the masses without ever having to show up for work. My grading program worked OK for the functional aspects of the students' software. My program

generated inputs, called the students' program and compared the results with expected behavior. Where I utterly failed was in my attempts to automatically grade their software on style. I used the following three part "quality" statistic. First, I measured execution speed the student's software,  $s_i$ . Smaller times represent improved dynamic efficiency. Next, I measured the number of bytes in the object code,  $b_i$ . Again, a smaller number represents better static efficiency. Third, I used the number of ASCII characters in the source code,  $c_i$ , as a quantitative measure of documentation. For this parameter, bigger is better. In a typical statistical fashion, I used the average and standard deviation to calculate

$$\text{quality} = \frac{\bar{s} - s_i}{s_s} + \frac{\bar{b} - b_i}{s_b} + \frac{c_i - \bar{c}}{s_c}$$

Half way through the semester, I happened to look at some assignments and was horrified to find the all-time worst software ever written from both a style and content basis. To improve speed and reduce size, the students cut so many corners that their code didn't really work anymore, it just appeared to work to my grading program. Then they took the ugly mess and filled it with nonsense comments, giving it the appearance of extensive documentation. To my students in that class that semester, I sincerely apologize. We should write comments for coworkers who must change our software, or clients who will use our software.

### 5.6.3. Inappropriate I/O and Portability

One of the biggest mistakes beginning programmers make is the inappropriate usage of I/O calls (e.g., screen output and keyboard input). An explanation for their foolish behavior is that they haven't had the experience yet of trying to reuse software they have written for one project in another project. Software portability is diminished when it is littered with user input/output. To reuse software with user I/O in another situation, you will almost certainly have to remove the input/output statements. In general, we avoid interactive I/O at the lowest levels of the hierarchy, rather return data and flags and let the higher level program do the interactive I/O. Often we add keyboard input and screen output calls when testing our software. It is important to remove the I/O that is not directly necessary as part of the module function. This allows you to reuse these functions in situations where screen output is not available or appropriate. Obviously screen output is allowed if that is the purpose of the routine.

**Common Error:** Performing unnecessary I/O in a subroutine makes it harder to reuse at a later time.

# 5.7. How Assemblers Work

Assemblers are development tools that process assembly language source program statements and translate them into executable machine language object files. The symbolic language used to code source programs to be processed by the assembler is called assembly language. The language is a collection of mnemonic symbols representing: operations (i.e., machine instruction mnemonics or directives to the assembler), symbolic names, operators, and special symbols. The assembly language provides mnemonic operation codes for all machine instructions in the instruction set. The assembly language also contains mnemonic directives that specify auxiliary actions to be performed by the assembler. These directives or pseudo-ops are not always translated into machine language.

Most assemblers require two passes. During the first pass, the source program is analyzed in order to develop the symbol table. A **symbol table** is a mapping between symbolic names (e.g., **GPIO\_PORTD\_DIR\_R**) and their numeric values (e.g., **0x40007400**). During the second pass, the object file is created (assembled) using the symbol table developed in pass one. It is during the second pass that the source program listing is also produced. The symbol table is recreated in the second pass. A phasing error occurs if the symbol table values calculated in the two passes are different. Errors that occur during the assembly process (e.g., undefined symbol, illegal op code, illegal operand, etc.) are explained in the listing file.

The **source code** is a file of characters usually created with an editor. Each line within the source code is processed completely before the next line is read. As each line is processed, the assembler examines the label, operation code, and operand fields. The operation code table is scanned for a match with a known opcode. During the processing of a standard operation code mnemonic, the standard machine code is inserted into the object file. If an assembler directive is being processed, the proper action is taken.

Any errors that are detected by the assembler are displayed after the actual line containing the error is printed. **Object code** is the binary values (instructions and data) that, when executed by the computer, perform the intended function. The listing file contains the address, object code, and a copy of the source code. The listing file also provides a symbol table describing where in memory the program and data will be loaded. The symbol table is a list of all the names used in the program along with the values. A symbol is created when you put a label starting in column 1. The symbol table value for this type is the absolute memory address where the instruction, variable or constant will reside in memory. The second type of label is created by the **EQU**,

**GPIO\_PORTD\_DIR\_R EQU 0x40007400**

The value for this type of symbol is simply the number specified in the operand field. When the assembler processes an instruction with a symbol in it, it simply substitutes the fixed value in place of the symbol. Therefore we will use symbols to clarify (make it easier to understand) our programs. The symbol table for this example is given at the end of the listing file. An assembler error will occur if the operand cannot be formed with the flexible second operand:

**MOV R0,#GPIO\_PORTD\_DIR\_R ;will not assemble**

**LDR R0,=GPIO\_PORTD\_DIR\_R ;this is OK**

A compiler converts high-level language source code into object code. A cross-compiler also converts source code into object code and creates a listing file except that the object code is created for a target machine that is different from the machine running the cross-compiler. uVision® and Code Composer Studio™ include both a cross-assembler and a cross-compiler because they run on the Windows PC and creates Cortex™ object code.

**Checkpoint 5.15:** What does the assembler do in pass 1?

**Checkpoint 5.16:** What does the assembler do in pass 2?

# 5.8. Functional debugging

## 5.8.1. Stabilization

**Functional debugging** involves the verification of input/output parameters. Functional debugging is a static process where inputs are supplied, the system is run, and the outputs are compared against the expected results. Four methods of functional debugging are presented in this section, and two more functional debugging methods are presented in the next chapter after indexed addressing mode is presented. There are two important aspects of debugging: control and observability. The first step of debugging is to **stabilize** the system. In the debugging context, we stabilize the problem by creating a test routine that fixes (or stabilizes) all the inputs. In this way, we can reproduce the exact inputs over and over again. Stabilization is an effective approach to debugging because we can control exactly what software is being executed. Once stabilized, if we modify the program, we are sure that the change in our outputs is a function of the modification we made in our software and not due to a change in the input parameters. When a system has a small number of possible inputs (e.g., less than a million), it makes sense to test them all. When the number of possible inputs is large we need to choose a set of inputs. There are many ways to make this choice. We can select values:

Near the extremes and in the middle

Most typical of how our clients will properly use the system

Most typical of how our clients will improperly attempt to use the system

That differ by one

You know your system will find difficult

Using a random number generator

To stabilize the system we define a fixed set of inputs to test, run the system on these inputs, and record the outputs. Debugging is a process of finding patterns in the differences between recorded behavior and expected results. The advantage of modular programming is that we can perform modular debugging. We make a list of modules that might be causing the bug. We can then create new test routines to stabilize these modules and debug them one at a time. Unfortunately, sometimes all the modules seem to work, but the combination of modules does not. In this case we study the interfaces between the modules, looking for intended and unintended (e.g., unfriendly code) interactions.

## 5.8.2. Single Stepping

Many debuggers allow you to set the program counter to a specific address then execute one instruction at a time. The debugger provides three stepping commands **Step**, **StepOver** and **StepOut** commands. **Step** is the usual execute one assembly instruction. However, when debugging C we can also execute one line of C. **StepOver** will execute one assembly instruction, unless that instruction is

a subroutine call, in which case the debugger will execute the entire subroutine and stop at the instruction following the subroutine call. **StepOut** assumes the execution has already entered a subroutine, and will finish execution of the subroutine and stop at the instruction following the subroutine call.

## 5.8.3. Breakpoints with Filtering

A **breakpoint** is a mechanism to tag places in our software, which when executed will cause the software to stop. Normally, you can break on any line of your program.

One of the problems with breakpoints is that sometimes we have to observe many breakpoints before the error occurs. One way to deal with this problem is the conditional breakpoint. To illustrate the implementation of conditional breakpoints, add a global variable called **Count** and initialize it to 32 in the initialization ritual. Add the following conditional breakpoint to the appropriate location in your software. Using the debugger, we set a regular breakpoint at **bkpt**. We run the system again (you can change the 32 to match the situation that causes the error.)

```
PUSH {R1, R2} ; save R1 and R2
LDR R2, =Count ; R2 = Count           if(--Count==0)
LDR R1, [R2] ; R1 = Count           bkpt
SUBS R1, R1, #1 ; Count = Count - 1
STR R1, [R2] ; store to Count
BNE DEBUG_skip ; if Count != 0, skip
DEBUG_bkpt NOP ; put breakpoint here
DEBUG_skip POP {R1, R2} ; restore R1 and
R2
```

Notice that the breakpoint occurs only on the 32<sup>nd</sup> time the break is encountered. Any appropriate condition can be substituted. Most modern debuggers allow you to set breakpoints that will trigger on a count. However, this method allows flexibility of letting you choose the exact conditions that cause the break.

## 5.8.4. Instrumentation: Print Statements

The use of print statements is a popular and effective means for functional debugging. One difficulty with print statements in embedded systems is that a standard “printer” may not be available. Another problem with printing is that most embedded systems involve time-dependent interactions with its external environment. The print statement itself may be so slow, that the debugging process itself causes the system to fail. In this regard, the print statement is **intrusive**. Therefore, throughout this book we will utilize debugging methods that do not rely on the availability of a standard output device. If `printf` is implemented using the UART (`printf_UARTxxx.zip`), the baud rate is 115200 bits/sec. This translates to 11520 characters per second. Therefore, it takes about 868  $\mu$ s to output 10 characters using `printf`. However, because the UART has a 16-character buffer, if we output less than 16 characters, and then wait more than 1.4ms before we output again, the use of `printf` would be minimally intrusive.

## 5.8.5. Desk checking

We perform a desk check (or dry run) by determining in advance, either by analytical algorithm or explicit calculations, the expected outputs of strategic intermediate stages and final results for typical inputs. We then run our program and compare the actual outputs with this template of expected results.

**Focus.** The first step is to specify a small piece of code you think may be incorrect. This step will get easier as you develop experience writing and debugging code. If a particular output is incorrect, we suspect each of the modules used to transform input into the incorrect output. Rather than testing the entire system we debug a very small piece (10 to 20 lines).

**Preparation.** For a particular set of inputs (e.g., inputs that result in incorrect outputs), we hand-execute our code, writing down all intermediate results as we expect them to be. It is called desk checking because this step is performed at our desk with paper and pencil.

**Control.** Set a breakpoint at the start of the code, and run to that point. We should force the input parameters to be one of the sets of inputs for which we prepared.

**Observability.** Configure the debugger so we can observe variables and intermediate calculations. Single step the software and compare the actual outputs with the expected results.

# 5.9. Exercises

**5.1** Assume you have a 16-bit unsigned global variable H. Write assembly code that implements **if(H > 1234)isGreater();**

**5.2** Assume you have a 16-bit signed global variable H. Write assembly code that implements **if(H > -1234)isGreater();**

**5.3** Assume you have an 8-bit unsigned global variable G. Write assembly code that implements **if(G < 50) isLess(); else isMore();**

**5.4** Assume you have a 16-bit signed global variable H. Write assembly code that implements **if(H < -500) isLess(); else isMore();**

**5.5** Assume you have an 8-bit global variable G. Write assembly code that implements **while(G&0x80)body();**

**5.6** Write assembly code that implements  
**while(GPIO\_PORTD\_DATA\_R &0x01)body();**

**5.7** You will write three assembly language versions of the following C code  
**n=100; while(n!=0){n--; body();}**

- a) Assume the variable **n** is implemented as a 16-bit global variable.
- b) Assume the variable **n** is implemented as an 8-bit global variable.
- c) Assume the variable **n** is implemented as a 32-bit variable in Register R2.

**5.8** You will write three assembly language versions of the following C code  
**n=0; while(n<100){n++; body();}**

- a) Assume the variable **n** is implemented as a 16-bit global variable.
- b) Assume the variable **n** is implemented as an 8-bit global variable.
- c) Assume the variable **n** is implemented as a 32-bit variable in Register R4.

**5.9** You will write two assembly language versions of the following C code  
**n=1000; while(n!=0){n--; body();}**

- a) Assume the variable **n** is implemented as a 16-bit global variable.
- b) Assume the variable **n** is implemented as a 32-bit variable in Register R5.

**5.10** There are two 16-bit signed variables, called **Input** and **Output**. Write assembly code that checks the **Input**, and if **Input** is less than -100, then the code sets the **Output** to 200. Conversely if **Input** is greater than or equal to -100, then the code does not modify **Output**.

**5.11** Assume Register R0 contains an ASCII character. Write assembly code that converts any lower case letters (a-z) to upper case (A-Z). For example, if Register R0 is initially ‘g’, convert it to ‘G’. Leave all other characters unchanged.

**5.12** Assume Register R0 contains an ASCII character. Write assembly code that converts any upper case letters (A-Z) to lower case (a-z). For example, if Register R0 is initially ‘G’, convert it to ‘g’. Leave all other characters unchanged.

**5.13** Write an assembly subroutine that implements a median filter. The three 32-bit unsigned numbers are passed into the subroutine by value in Registers R0, R1 and R2. The median is the middle value of the three, sorted by size. The return parameter is passed back in Register R0.

**5.14** Write an assembly subroutine that finds the least common multiple of two numbers. The inputs are passed in as 32-bit unsigned numbers in Registers R0 and R1. The result is returned as a 32-bit unsigned number in Register R0.

**5.15** You are given a stopwatch module with the following functions. Try and guess what each function does. **Watch\_SetTimerResolution**, **Watch\_StartTimer**, **Watch\_StopTimer**, **Watch\_DisplayTime**

**D5.16** Assume there are two 8-bit unsigned digital inputs attached to Ports F and G respectively. Port F contains the measured motor speed in rotations per second (rps), and Port G contains the desired motor speed also in rps. There is an 8-bit unsigned output on Port A interfaced to the motor that controls power to the motor. PORTA=0 means no power, PORTA =255 is full power. After initialization the incremental motor controller should implement this: If the actual speed is less than the desired and if PORTA <255, then increment PORTA

If the actual speed equals the desired, then do not change PORTA

If the actual speed is greater than the desired and if PORTA >0, then decrement PORTA

After initialization, the body of the **main** program execute over and over.

**D5.17** The goal is to design a one-wheeled balancing robot. Assume there is an 8-bit signed digital input attached to Port F. Port F contains the measured angle of the robot with respect to the ground in degrees. The desired position is straight up, which is 0 degrees. There is an 8-bit signed output on Port G interfaced to the wheel that controls torque to the wheel. Port G = -128 means full clockwise torque, Port G = 0 means no torque, and Port G = +127 means full counterclockwise torque. After initialization the incremental controller should:

If the angle is less than zero and if Port G < 127, then increment Port G

If the angle equals zero, then do not change Port G

If the angle is greater than zero and if Port G > -128, then decrement Port G

After initialization, the body of the **main** program execute over and over.

# 5.10. Lab Assignments

There are many types of recursion. The factorial and the decimal output functions in Section 5.4 are examples of **linear recursion**, because only one call is made to the function within the function. A **tail recursive** function has the recursive call as the last action taken by the function. A tail recursive function can be implemented in an iterative manner by removing the recursive call and substituting it with a loop. A **binary recursive** function calls itself twice during the course of its execution. For Labs 5.1 and 5.2, pass parameters in registers and place local variables also in registers. Implement 32-bit unsigned arithmetic. Design a main program to test the functionality of your solution. Measure the execution speed and required stack space of both versions for 5 different input values. Generalize the results.

**Lab 5.1** Tail Recursion. Implement the following recursive greatest common divisor function in assembly language. Convert the operation to a nonrecursive algorithm, and implement it also in assembly language.

```
uint32_t gcd(uint32_t m, uint32_t n){  
    uint32_t r;  
    if(m < n){  
        return gcd(n,m);  
    }  
    r = m%on;  
    if(r == 0){  
        return(n);  
    }  
    return(gcd(n,r));  
}
```

**Lab 5.2** Binary Recursion.  ${}_nC_k$  is the number of combinations of choosing  $n$  elements out of a set of  $k$  elements. Implement the following recursive function in assembly language. Convert the operation to a nonrecursive algorithm, and implement it also in assembly language.

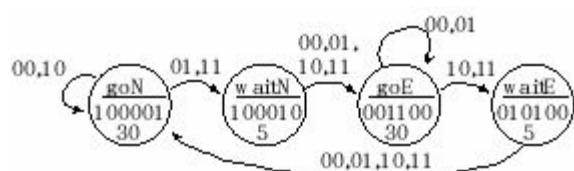
```
uint32_t nCk(uint32_t n, uint32_t k){  
    if((k == 0)||(n == k)){  
        return(1);  
    }  
    return(nCk(n-1,k) + nCk(n-1,k-1));  
}
```

# 6. Pointers and Data Structures

## Chapter 6 objectives are to:

- Implement pointers using indexed addressing modes
- Use pointers to access arrays, strings, structures, tables, and matrices
- Present finite state machines as an abstractive design methodology
- Use tables and interpolation to implement nonlinear functions
- Present minimally intrusive methods for functional debugging

Data are brought into registers temporarily for manipulation and decision making. However, on a long term basis, we store data in memory. If the data values are known at design time, and do not change, we place them in ROM. If we do not know the values at assembly time, or if the values vary with time, we need to place them in RAM. When we write software that manipulates the exact same data each time, then we can know its address at assembly time and use direct or extended addressing to access the data. For example, because the addresses of the I/O ports are fixed, we typically use direct or extended addressing to access I/O. Conversely, sometimes we write software that operates on different data at different times (e.g., calculating the average of a set of numbers in various buffers, outputting different strings on an LCD.) In these cases, we need a way to access data, where which data we are operating on is determined at run-time. For these situations, we use pointers. A pointer is simply an address. A pointer (e.g., Pt in Figure 6.1) is a variable where the contents of the variable is not data, but rather an address. Our software can be extremely flexible if we allow the address to change dynamically. On the Cortex™-M processor, pointers are 32 bits containing the address of the data of interest. Before we use a pointer, we must initialize it, so it points to an object. We can also change a pointer at run time, so it points to a different object, as shown in Figure 6.1. In this book, we will use the address 0 as the null pointer, meaning the pointer is not valid. In this chapter, the objects addressed by pointers will be data, but in Chapter 7, we will see an example of function pointers. A function pointer is an address pointing to a subroutine. The reset vector, stored at 0x0000.0004, is an example of function pointer, because it contains a pointer to the main program.



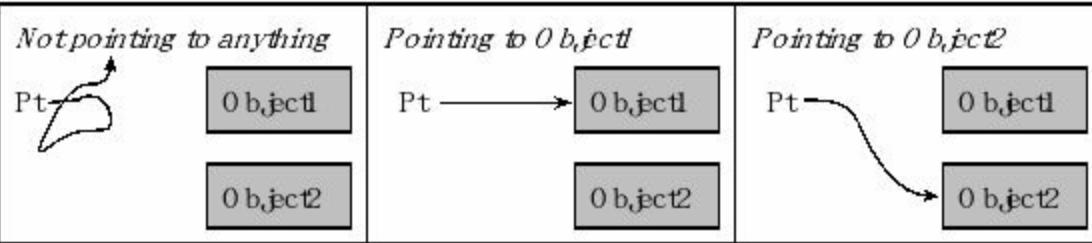


Figure 6.1. Pointers are addresses pointing to objects. The objects may be data, functions, or other pointers.

# 6.1. Indexed Addressing and Pointers

At the assembly level, we implement **pointers** using indexed addressing mode. For example, a register contains an address, and the instruction reads or writes memory specified by that address. Basically, we place the address into a register, then use indexed addressing mode to access the data. In this case, the register holds the pointer. Figure 6.2 illustrates three examples that utilize pointers. In this figure, **Pt** **SP** **GetPt** **PutPt** are pointers, where the arrows show to where they point, and the shaded boxes represent data. An **array or string** is a simple structure containing multiple equal-sized elements. We set a pointer to the address of the first element, then use indexed addressing mode to access the elements inside. We have introduced the stack previously, and will cover it in more detail in Chapter 7. The stack pointer (SP) points to the top element on the stack. A linked list contains some elements that are pointers themselves. The pointers are used to traverse the data structure. Example linked lists will be presented in Section 6.6. The first in first out (FIFO) queue is an important data structure for I/O programming because it allows us to pass data from one module to another. One module puts data into the FIFO and another module gets data out of the FIFO. There is a **GetPt** that points to the oldest data (to be removed next) and a **PutPt** that points to an empty space (location to be stored into next). The FIFO queue will be presented in detail in Chapter 11.

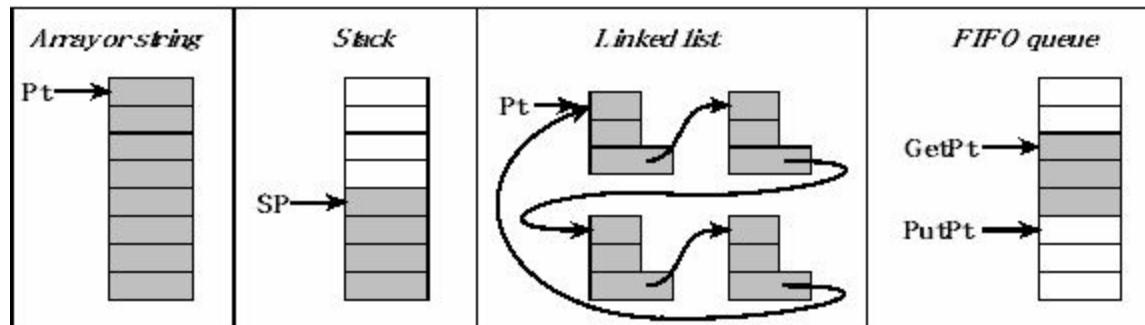


Figure 6.2. Examples of data structures that utilize pointers.

1
2
3
5
7
11
13
17
19
23

**16 bits** Accessing 16-bit data structures with indexed addressing is slightly different in assembly versus in C. For example, if we create an array of the first ten prime numbers stored as 16-bit integers, we could allocate the structure in ROM using the **DCW** pseudo-op. E.g.,

**Prime DCW 1,2,3,5,7,11,13,17,19,23**

The equivalent ROM-based definition in C would be

```
uint16_t const Prime[10]={1,2,3,5,7,11,13,17,19,23};
```

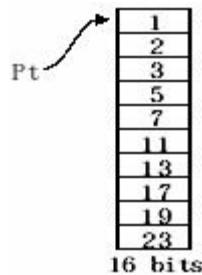
By convention, we define **Prime[0]** as the first element, **Prime[1]** as the second element, etc. The address of the first element can be written as **&Prime[0]** or just **Prime**. In C, if we want the 5<sup>th</sup> element, we use the expression **Prime[4]** to fetch the 7 out of the structure. In assembly, however, we are responsible for knowing each element is two bytes and the 5<sup>th</sup> element is actually at bytes number 8 and 9. In general, the n<sup>th</sup> element of a 16-bit array is at bytes 2n-2 and 2n-1. E.g., to read the 5<sup>th</sup> element into Register R0 we need to perform

```
LDR R1,=Prime ;pointer to the structure
LDRH R0,[R1,#8] ;read 16-bit unsigned Prime[4]
```

Either way, manipulating addresses in assembly always involves the physical byte-address regardless of the precision of the data. Similarly assume we have a pointer to **Prime**, and we want to increment the pointer to the next element. In C, we define the pointer as

```
uint16_t const *Pt;
```

In this case, the **const** does not indicate the pointer is fixed. Rather, the pointer refers to constant 16-bit data in ROM. We initialize the pointer at run time using



```
Pt = Prime; // Pt points to Prime
```

or

```
Pt = &Prime[0]; // Pt points to Prime
```

Similarly in assembly, we can define the pointer in RAM as

```
Pt SPACE 4 ; pointer to Prime
```

and initialize it as

```
LDR R1,=Prime ;pointer to the structure
```

```
LDR R0,Pt ;pointer to the Pt
```

```
STR R1,[R0] ;Pt is a pointer to Prime[0]
```

You should not use **DCD** to define/allocate RAM-based variables in microcontrollers used for embedded systems, because these RAM-based variables have no initial value when power is applied to the microcontroller. One must explicitly initialize variables using assembly code like the above three lines. In C however, at start-up all RAM-based variables are initialized to an explicit value, or to zero if no explicit initial value is given.

Now, to increment the pointer to the next element in C, use the expression **Pt++**. In C, **Pt++**, which is the same thing as **Pt=Pt+1**; actually adds two to the pointer because it points to halfwords. However, in assembly we have to explicitly add 2 to the pointer. E.g.,

```

LDR R0,Pt    ;pointer to the Pt
LDR R1,[R0]  ;R1 is value of Pt
ADD R1,#2
STR R1,[R0] ;update Pt

```

**Observation:** We normally add/subtract one to the pointer when accessing an 8-bit array, add/subtract two when accessing a 16-bit array, and add/subtract four when accessing a 32-bit array.

The subroutines thus far in the book have utilized **call by value** parameter passing. With an input parameter using call by value, the data itself is passed into the subroutine. For an output parameter using return by value, the result of the subroutine is a value, and the value itself is returned. The most efficient mechanism to pass parameters is the registers. In Chapter 7 we will learn a more flexible, but less efficient technique to pass parameters using the stack. Alternatively, if you pass a pointer to the data, rather than the data itself, we will be able to pass large amounts of data. Passing a pointer to data is classified as **call by reference**. For large amounts of data, call by reference is also very fast, because the data need not be copied from calling program to subroutine. In call by reference, the one copy of the data exists in the calling program, and a pointer to it is passed to the subroutine. In this way, the subroutine actually performs read/write access to the original data. Call by reference is also a convenient mechanism to return data as well. Passing a pointer to an object allows this object to be an input parameter and an output parameter.

As an example, consider the situation where we wish to pass 100 bytes into the subroutine **Sort** . In this case, we have one or more buffers, defined in RAM, which initially contains data in an unsorted fashion. The buffers shown here are uninitialized, but assume previously executed software has filled these buffers with corresponding voltage and pressure data. In C, we could have

```

uint8_t VBuffer[100]; // voltage data
uint8_t PBuffer[100]; // pressure data

```

A similar definition in assembly would be

```

VBuffer SPACE 100 ;voltage data
PBuffer SPACE 100 ;pressure data

```

Since 100 bytes is more than will fit in the registers, we will use call by reference. In C, to declare a parameter call by reference we use the \*.

```

void Sort(uint8_t *bufPt){uint8_t data;
  data = *bufPt; // example read data from original buffer
  *bufPt = data; // example write data back to the buffer
}

```

In C, to invoke a function using call by reference we pass a pointer to the object. These two calling sequences are identical, because in C the array name is equivalent to a pointer to its first element. The & operator is used to get the address of a variable.

```

void main(void){
  Sort(VBuffer);
  Sort(PBuffer);

```

```

void main(void){
  Sort(&VBuffer[0]);
  Sort(&PBuffer[0]);

```

}

In assembly, we use a register as a pointer to the data. The calling sequence for sorting the voltage data in **VBuffer** could be

**LDR R0,=VBuffer ;R0 = &VBuffer (pointer to first element)**  
**BL Sort**

The calling sequence for sorting the pressure data in **PBuffer** could be

**LDR R0,=PBuffer ;R0 = &PBuffer (pointer to first element)**  
**BL Sort**

One advantage of call by reference in this example is the same buffer can be used also as the return parameter. In particular, this sort routine could shuffle the data around in the same original buffer. Since RAM is a scarce commodity on most microcontrollers, not having to allocate two buffers will reduce RAM requirements for the system.

From a security perspective, call by reference is more vulnerable than call by value. If we have important information, then a level of trust is required to pass a pointer to the original data to a subroutine. Since call by value creates a copy of the data at the time of the call, it is slower but more secure. With call by value, the original data is protected from subroutines that are called.

## 6.2. Arrays

**Random access** means one can read and write any element in any order. Random access is allowed for all indexable data structures. An indexed data structure has elements of the same size and can be accessed knowing the name of the structure, the size of each element, and the element number. In C, we use the syntax `[]` to access an indexed structure. Arrays, matrices, and tables are examples of indexed structures presented in this chapter.

**Sequential access** means one reads and writes the elements in order. Pointers are usually employed in these types of data structures. Strings, linked-lists, stacks, queues, and trees are examples of sequential structures. The first in first out circular queue (**FIFO**) is useful for data flow problems, and it will be presented in Chapters 9 and 11.

An **array** is made of elements of equal precision and allows random access. The **precision** is the size of each element. Typically, precision is expressed in bits or bytes. The **length** is the number of elements. The **origin** is the index of the first element. A data structure with the first element existing at index zero is called **zero-origin indexing**. In C, zero-origin index is almost always used. For example, **Prime[0]** is the first element of the array **Prime**.

---

**Example 6.1.** Write a software module to control the read/write (R/W) head of an audio tape recorder. From the perspective shown in Figure 6.3, the stepper motor causes the R/W head to move up and down. This motion affects which audio track on the tape is under the head. The goal is to be able to move the motor one step at a time.

**Solution:** This module requires three public functions: one for initialization, one to rotate one step clockwise, and one to rotate one step counter-clockwise. By rotating the motor one step at a time, the software can control which audio track on the tape is under the R/W head. A stepper motor has four digital control lines. To make the stepper motor spin, we output the sequence 5, 6, 10, and 9 over and over on these four lines. To make it spin in the other direction, we output the sequence in the other direction. This motor has 24 steps per revolution; therefore one step will change the shaft angle by exactly  $15^\circ$ . To make the motor step once, we output just the next number in the sequence. For example, if the output is currently at 5, and we wish to rotate the shaft by  $15^\circ$ , we simply output a 6. In this solution, we will store the 5, 6, 10, and 9 data in an array, as shown in Figure 6.4. For more information on the hardware interfacing of stepper motors see Section 8.7.

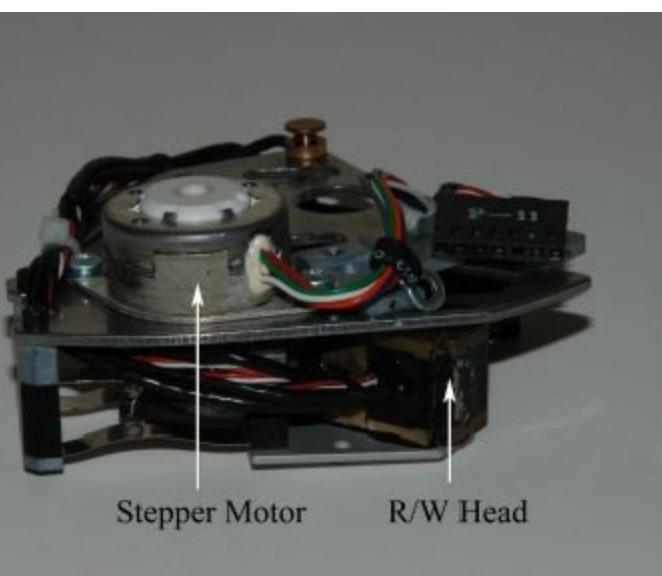


Figure 6.3. A stepper motor is used in a cassette tape recorder to select the track.

0x2000.0100	0x05
0x2000.0101	0x06
0x2000.0102	0x0A
0x2000.0103	0x09

Figure 6.4. A byte array with 4 elements (addresses are made up to illustrate the array is in ROM).

In C, we can access an element of the array using its name and an index. Assume Port D bits 3–0 are connected to the stepper motor. The initialization function makes those pins an output, and the **Index** is initialized to zero. In assembly, we can perform a similar function using indexed addressing, see Program 6.1. Assume **Index** is a 32-bit private global variable, defined in RAM, and initialized to zero. **Index** takes on the values 0, 1, 2, and 3. The instruction **LDRB R3,[R0,R2]** adds the base address in Register R2 to the index value in Register R0, fetching the contents of the array at that index. Since the first output generated by **Stepper\_CW** will be a 5, we will initialize the motor to 9; this way, the first call to **Stepper\_CW** will move the motor. In this example, the subroutine is public but has no input or output parameters. Port D, the array, and the index are private to this module. This means if another module wishes to move the motor, it can call the public function **Stepper\_CW**, but does not have access to Port D bits 3–0, **Data**, or **Index**. The third public function, **Stepper\_CCW**, is left as Homework problem 6.29. To make the code friendly, we will use this bit-specific **STEPPER** definition of PD3-0, the same as **LEDS** back in Program 4.5.

```
#define STEPPER (*((volatile uint32_t *)0x4000703C))
```

<pre>AREA DATA, ALIGN=2 Index SPACE 4  AREA .text, CODE, READONLY, ALIGN=2 THUMB  Data DCB 0x05, 0x06, 0x0A, 0x09 STEPPER EQU 0x4000703C  Stepper_Init     PUSH {R4,LR}     BL GPIO_Init ; Program 4.5     LDR R1, =STEPPER     MOV R0, #0x09</pre>	<pre>const uint8_t Data[4]= {0x05,0x06,0x0A,0x09}; uint32_t static Index;  void Stepper_Init(void){     GPIO_Init(); // Prog 4.5     STEPPER = 0x09;     Index = 0; // first index }</pre>
---	--

```

STR R0, [R1] ; PortD = 9
MOV R0, #0
LDR R1, =Index
STR R0, [R1] ; Index = 0
POP {R4,PC}
;Move one step clockwise
Stepper_CW
    LDR R1, =Index
    LDR R0, [R1] ; R0 = Index
    LDR R2, =Data ; R2 = &Data
    LDRB R3, [R0, R2] ; R3 = Data[Index]
    LDR R2, =STEPPER
    STR R3, [R2] ; PortD = Data[Index]
    ADD R0, R0, #1 ; Index+1
    AND R0, R0, #3 ; 0 <= Index <= 3
    STR R0, [R1]
    BX LR

```

```

void Stepper_CW(void){
    STEPPER = Data[Index];
    Index = (Index+1)&0x03;
}

```

## Program 6.1. Stepper motor software that uses a byte array.

When constraining a sequence of numbers to a power of two, we can use a logical AND. If a sequence runs from 0 to  $2^n - 1$ , then the mask will be  $2^n - 1$ . To get the next number in the sequence we execute  $I = (I+1) \& (2^n - 1)$ . Before connecting an actual stepper motor, check its datasheet to determine its maximum current load. The TM4C123 microcontroller can only source 8mA, and the TM4C1294 can source 12mA, which is far less than what a motor will need. Beware that connecting the motor directly to digital logic pins will break your microcontroller. This stepper motor interface will be completed in Section 8.8.

---

### Example 6.2. Design an exponential function, $y = 10^x$ , with a 32-bit output.

**Solution:** Since the output is less than 4,294,967,295, the input must be between 0 and 9. One simple solution is to employ a constant word array, as shown in Figure 6.5. Each element is 32 bits. In assembly, we define a word constant using **DCD**, making sure it exists in ROM.

In C, the syntax for accessing all array types is independent of precision. See Program 6.2. The compiler automatically performs the correct address correction. We will assume the input is less than or equal to 9. If **x** is the index and **Base** is the base address of the array, then the address of the element at **x** is **Base + 4\*x**. In assembly, we can access the array using indexed addressing. We will assume the Register R0 input is less than or equal to 9.

0x00000134	1
0x00000138	10
0x0000013C	100
0x00000140	1,000
0x00000144	10,000
0x00000148	100,000
0x0000014C	1,000,000
0x00000150	10,000,000

<b>0x00000154</b>	<b>100,000,000</b>
<b>0x00000158</b>	<b>1,000,000,000</b>

Figure 6.5. A word array with 10 elements. Addresses illustrate the array is stored in ROM as 4 bytes each.

<b>AREA</b> <b>.text CODE,READONLY,ALIGN=2</b> <b>Powers DCD 1, 10, 100, 1000, 10000</b> <b>    DCD 100000, 1000000, 10000000</b> <b>    DCD 100000000, 1000000000</b> <b>; Input: R0=x    Output: R0=10^x</b> <b>power LSL R0, R0, #2 ; x = x*4</b> <b>    LDR R1, =Powers ; R1 = &amp;Powers</b> <b>    LDR R0, [R0, R1] ; y=Powers[x]</b> <b>    BX LR</b>	<b>const uint32_t Powers[10]</b> <b>={1,10,100,1000,10000,</b> <b>    100000,1000000,10000000,</b> <b>    100000000,1000000000};</b>  <b>uint32_t power(uint32_t x){</b> <b>    return Powers[x];</b> <b>}</b>
--	---

## Program 6.2. Array implementation of a nonlinear function.

In the previous examples, the length of the array was known. Sometimes, it is desirable to allow the length to vary dynamically. There are many mechanisms that allow for a variable length array. One simple mechanism saves the length of the array as the first element. In this way, we could add run time checking to make sure the index bounds are not exceeded. The **Powers** function could have been defined as

```
const uint32_t Powers[11]={10, 1, 10, 100, 1000, 10000, 100000,
    1000000, 10000000, 100000000, 1000000000};

uint32_t power(uint32_t x){
    if(x<Powers[0])
        return Powers[x+1];
    return 0xFFFFFFFF; // overflow
}
```

Another common mechanism to handle variable length is a termination code. Typical codes for ASCII character data are shown in Table 6.1. This method can only be used if it is not possible for the termination code to be present in the data.

ASCII	C	code	name
NUL	\0	0x00	null
ETX	\x03	0x03	end of text
EOT	\x04	0x04	end of transmission
FF	\f	0x0C	form feed
CR	\r	0x0D	carriage return
ETB	\x17	0x17	end of transmission block

Table 6.1. Typical termination codes

For arrays of numbers, we can use one of the extreme values as the termination code. For example, if the data were 16-bit unsigned integers we could use 65535 as the termination code. The other values from 0 to 65534 would represent actual data. Here are three examples of variable length arrays of 16-bit unsigned integers. The first array has 4 elements, the second 9 elements and the third array is empty.

```
const uint16_t Data1[5]={0,3,4,1,65535};
const uint16_t Data2[10]={1,2,3,4,5,6,7,8,9,65535};
const uint16_t Data3[1]={65535};
```

If we wished to add up all values in this variable length array, we would need to pass the array into the function using call by reference. In C, the first parameter is passed in R0, and the return parameter is returned in R0. The assembly implementation passes parameters the same way, see Program 6.3. The elements are 16-bit unsigned, so the instruction **LDRH** is used to fetch entries from the array.

<pre>; Input: R0=&amp;Array  Output: R0=sum Sum MOV R1,#0       LDR R3,=65535 loop LDRH R2,[R0] ; value from array       CMP R2,R3 ; termination?       BEQ done       ADD R1,R1,R2       ADD R0,#2 ; next halfword       B loop done MOV R0,R1 ; return result BX LR</pre>	<pre>uint16_t Sum(uint16_t *pt){     uint16_t result=0;     while(*pt != 65535){         result = result + (*pt);         pt++;     }     return result; }</pre>
---	--

Program 6.3. Function to sum all elements of a variable length array. Data are 16 bits unsigned.

Similarly, if we have 16-bit signed data, we could use -32768 as the termination code

```
const int16_t Data4[5]={0,3,-4,1,-32768};
const int16_t Data5[10]={1,-2,3,4,-5,6,-7,8,-9,-32768};
const int16_t Data6[1]={-32768};
```

The functions in Program 6.4 will sum all values in a variable length array using a -32768 termination. The array is passed into the function using call by reference. The elements are 16-bit signed, so the instruction **LDRSH** is used to fetch entries.

<pre>; Input: R0=&amp;Array  Output: R0=sum Sum MOV R1,#0 loop LDRSH R2,[R0] ; value from array       CMP R2,#-32768 ; termination?       BEQ done       ADD R1,R1,R2       ADD R0,#2 ; next halfword       B loop done MOV R0,R1 ; return result BX LR</pre>	<pre>int16_t Sum(int16_t *pt){     int16_t result=0;     while(*pt != -32768){         result = result + (*pt);         pt++;     }     return result; }</pre>
---	--

Program 6.4. Function to sum all elements of a variable length array. Data are 16

bits signed.

In general, let **n** be the precision of a zero-origin indexed array in bytes. If **I** is the index and **Base** is the beginning address of the array, then the address of the element at **I** is

$$\text{Base} + n * I$$

The origin of an array is the index of the first element. The origin of a zero-origin indexed array is zero. In general, if **o** is the origin of the array, then the address of the element at **I** is

$$\text{Base} + n * (I - o)$$

## 6.3. Strings

H
e
l
l
o
w
o
r
l
d
13
10
0
8 bits

A **string** is a data structure with equal size elements that only allows sequential access. The bytes of the string are always read in order from the first to the last. In contrast, an **array** allows random access to any element in any order. The same mechanisms introduced for variable length arrays will apply also to strings. In general, we store the length of the string in the first position, when the data can take on any value, negating the possibility of using a termination code. From a programming perspective we access strings in the same manner as arrays. In C99, we can define a constant string of ASCII characters with null termination:

```
const char Hello[] = "Hello world\n\r";
```

When defining constant strings or arrays we must specify their value because the data will be loaded into ROM and cannot be changed at run time. In the previous constant arrays we specified the size; however for constant arrays the size can be left off and the compiler will calculate the size automatically. Notice also the string can contain special characters, some of which were listed previously in Table 4.11. The above string has 13 characters followed by a null (0) termination. The equivalent definition in assembly would be

```
Hello DCB "Hello world\n\r",0
```

In C, ASCII strings are stored with null-termination. In C, the compiler automatically adds the zero at the end, but in assembly, the zero must be explicitly defined.

---

**Example 6.3.** Write software to output an ASCII string an output device.

**Solution:** Because the length of the string may be too long to place all the ASCII characters into the registers at the same time, call by reference parameter passing will be used. With call by reference, a pointer to the string will be passed. The function **OutString**, shown in Program 6.5, will output the string data to the display. A version of the function **OutChar** will be developed later in Chapter 8 and shown as Program 8.1 which sends data out the UART. When using a development kit, this UART data is observable on the PC running a terminal program like PuTTY or TExaSdisplay. For now all we need to know is that it outputs a single ASCII character. In the assembly version R4 is used because we know by convention the function **OutChar** will preserve R4 to R11. However, by

convention this function will preserve R4 by saving and restoring it. R4 is a pointer to the string; one is added to the pointer each time through the loop because each element in the string is one byte. Since this function calls a subfunction it must save LR. The POP PC operation will perform the function return.

<b>;Input: R0 points to string</b>	<b>// displays a string</b>
<b>OutString</b>	<b>void OutString(char *pt){</b>
<b>PUSH {R4, LR}</b>	<b>while(*pt){</b>
<b>MOV R4, R0</b>	<b>OutChar(*pt); // output</b>
<b>loop LDRB R0, [R4]</b>	<b>pt++; // next</b>
<b>ADD R4, #1 ;next</b>	<b>}</b>
<b>CMP R0, #0 ;done?</b>	
<b>BEQ done ;0 termination</b>	
<b>BL OutChar ;print character</b>	
<b>B loop</b>	
<b>done POP {R4, PC}</b>	

### Program 6.5. A variable length string contains ASCII data.

**Observation:** Most C compilers have standard libraries. If you include “string.h” you will have access to many convenient string operations.

When dealing with strings we must remember that they are arrays of characters with null termination. In C, we can pass a string as a parameter, but doing so creates a constant string and implements call by reference. Assuming **Hello** is as defined above, these three invocations are identical:

```
OutString(Hello);
OutString(&Hello[0]);
OutString("Hello world\n\r");
```

Previously we dealt with constant strings. With string variables, we do not know the length at compile time, so we must allocate space for the largest possible size the string could be. E.g., if we know the string size could vary from 0 to 19 characters, we would allocate 20 bytes.

```
char String1[20];
char String2[20];
```

In C, we cannot assign one string to another. I.e., these are illegal

```
String1 = "Hello"; //*****illegal*****
String2 = String1; //*****illegal*****
```

We can make this operation occur by calling a function called **strcpy** , which copies one string to another. This function takes two pointers. We must however make sure the destination string has enough space to hold the string being copied.

```
strcpy(String1,"Hello"); // copies "Hello" into String1
strcpy(String2, String1); // copies String1 into String2
```

Program 6.6 shows two implementations of this string copy function. R0 and R1 are pointers, and R2 contains the data as it is being copied. In this case, `dest++`; is implemented as an “add 1” because the data is one byte each. In other situations, the increment pointer would be “add 2” for halfword data and would be “add 4” for word data.

<pre>; Input: R0=&amp;dest  R1=&amp;source strcpy LDRB R2,[R1] ;source data     STRB R2,[R0] ;copy     CMP R2,#0 ;termination?     BEQ done     ADD R1,#1 ;next     ADD R0,#1     B strcpy done BX LR ;faster version strcpy LDRB R2,[R1],#1 ;source data     STRB R2,[R0],#1 ;copy     CMP R2,#0 ;termination?     BEQ done     B strcpy done BX LR</pre>	<pre>// copy string from source to dest void strcpy(char *dest, char *source){     while(*source){         *dest = *source; // copy         dest++;           // next         source++;     }     *dest = '\0'; // termination } // another version void strcpy(char *dest, char *source){ char data;     do{         data = *dest++ = *source++;     } while(data); }</pre>
--	--

Program 6.6. Simple string copy functions.

## 6.4. Structures

A **structure** has elements with different types and/or precisions. In C, we use **struct** to define a structure. The **const** modifier causes the structure to be allocated in ROM. Without the **const**, the C compiler will place the structure in RAM, allowing it to be dynamically changed. If the structure were to contain an ASCII string of variable length, then we must allocate space to handle its maximum size. In this first example, the structure will be allocated in RAM so no **const** is included. The following code defines a structure with three elements. We give separate names to each element. The **typedef** command creates a new data type based on the structure, but no memory is allocated.

```
struct player{  
    uint8_t Xpos;      // first element  
    uint8_t Ypos;      // second element  
    uint16_t LifePoints; // third element  
};  
typedef struct player playerType;
```

We can allocate a variable called **Sprite** of this type, which will occupy four bytes in RAM:

```
playerType Sprite;
```

We can access the individual elements of this variable using the syntax **name.element**. After these three lines are executed we have the data structure as shown in Figure 6.6, assuming the variable occupies the four bytes starting at 0x2000.0250.

```
Sprite.Xpos = 10;  
Sprite.Ypos = 20;  
Sprite.LifePoints = 12000;
```

0x2000.0250	10
0x2000.0251	20
0x2000.0252	12000

Figure 6.6. A structure collects elements of different sizes and/or types into one object.

We can also define pointers to structures. We define pointer in a similar way as other pointers

```
playerType *Ptr;
```

Before we can use a pointer, we must make sure it points to something

```
Ptr = &Sprite;
```

We access the individual fields using the syntax **pointer->element**

```
Ptr->Xpos = 10;  
Ptr->Ypos = 20;  
Ptr->LifePoints = 12000;
```

We can create something similar to structures in assembly by using **EQU** definitions, see Program 6.7. Since structures have multiple elements, we will employ call by reference when passing parameters. This C function takes a player, moves it to location 50,50 and adds one life-point. We call the function by passing an address to a **playerType** variable. For example, we execute **MoveCenter(&Sprite);**

<pre>; Input: R0 = &amp;PlayerType Xpos    EQU 0 Ypos    EQU 1 LifePoints EQU 2 MoveCenter MOV R1,#50           STRB R1,[R0,#Xpos]           STRB R1,[R0,#Ypos]           LDRH R1,[R0,#LifePoints]           LDR R2,=65535           CMP R1,R2    ;at max?           BHS skip           ADD R1,#1    ;more life           STRH R1,[R0,#LifePoints] skip   BX LR</pre>	<pre>// move to center and add life void MoveCenter(playerType *pt){     pt-&gt;Xpos = 50;     pt-&gt;Ypos = 50;     if(pt-&gt;Life Points &lt; 65535){         pt-&gt;Life Points++;     } }</pre>
---	---

Program 6.7. A function that accesses a structure.

**Observation:** Most C compilers will align 16-bit elements within structures to an even address and will align 32-bit elements to a word-aligned address.

**Observation:** Call by reference allows the single parameter to be both an input parameter and an output parameter.

Without the **const**, the C compiler will place the structure in RAM, allowing it to be dynamically changed. If the structure resides in RAM, then the system will have to initialize the data structure explicitly by executing software. If the structure is in ROM, we must initialize it at compile time. The next section shows examples of ROM-based structures.

# 6.5. Finite State Machines with Linked Structures

## 6.5.1. Abstraction

Software abstraction allows us to define a complex problem with a set of basic abstract principles. If we can construct our software system using these abstract building blocks, then we have a better understanding of both the problem and its solution. This is because we can separate what we are doing (policies) from the details of how we are getting it done (mechanisms). This separation also makes it easier to optimize. Abstraction provides for a proof of correct function and simplifies both extensions and customization. The abstraction presented in this section is the **Finite State Machine** (FSM). The abstract principles of FSM development are the inputs, outputs, states, and state transitions. The FSM state graph defines the time-dependent relationship between its inputs and outputs. If we can take a complex problem and map it into a FSM model, then we can solve it with simple FSM software tools. Our FSM software implementation will be easy to understand, debug, and modify. Other examples of software abstraction include **Proportional Integral Derivative** digital controllers, fuzzy logic digital controllers, neural networks, and linear systems of differential equations. In each case, the problem is mapped into a well-defined model with a set of abstract yet powerful rules. Then, the software solution is a matter of implementing the rules of the model. In our case, once we prove our software correctly solves one FSM, then we can make changes to the state graph and be confident that our software solution correctly implements the new FSM.

The FSM controller employs a well-defined model or framework with which we solve our problem. The state graph will be specified using either a linked or table data structure. An important aspect of this method is to create a 1-1 mapping from the state graph into the data structure. The three advantages of this abstraction are 1) it can be faster to develop because many of the building blocks preexist; 2) it is easier to debug (prove correct) because it separates conceptual issues from implementation; and 3) it is easier to change. In a **Moore FSM** the output value depends only on the current state, and the inputs affect the state transitions. On the other hand, the outputs of a **Mealy FSM** depend both on the current state and the inputs. See Figure 6.7.

When designing a FSM, we begin by defining what constitutes a state. In a simple system like a single intersection traffic light, a state might be defined as the pattern of lights (i.e., which lights are on and which are off). In a more sophisticated traffic controller, what it means to be in a state might also include information about traffic volume at this and other adjacent intersections. The next step is to make a list of the various states in which the system might exist. As in all designs, we add outputs so the system can affect the external environment, and inputs so the system can collect information about its environment or receive commands as needed. The execution of a Moore FSM repeats this sequence over and over

1. Perform output, which depends on the current state

2. Wait a prescribed amount of time (optional)
3. Input
4. Go to next state, which depends on the input and the current state

The execution of a Mealy FSM repeats this sequence over and over

1. Wait a prescribed amount of time (optional)
2. Input
3. Perform output, which depends on the input and the current state
4. Go to next state, which depends on the input and the current state

There are other possible execution sequences. Therefore, it is important to document the sequence before the state graph is drawn. The high-level behavior of the system is defined by the state graph. The states are drawn as circles. Descriptive states names help explain what the machine is doing. Arrows are drawn from one state to another, and labeled with the input value causing that state transition.

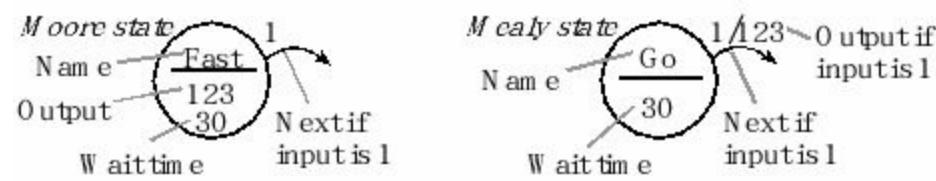


Figure 6.7. The output in a Moore depends just on the state. In a Mealy the output depends on state and input.

**Observation:** If the machine is such that a specific output value is necessary “to be a state”, then a Moore implementation will be more appropriate.

**Observation:** If the machine is such that no specific output value is necessary “to be a state”, but rather the output is required to transition the machine from one state to the next, then a Mealy implementation will be more appropriate.

A linked structure consists of multiple identically-structured nodes. Each node of the linked structure defines one state. One or more of the entries in the node is a pointer (or link) to other nodes. In an embedded system, we usually use statically-allocated fixed-size linked structures, which are defined at compile time and exist throughout the life of the software. In a simple embedded system the state graph is fixed, so we can store the linked data structure in nonvolatile memory. For complex systems where the control functions change dynamically (e.g., the state graph itself varies over time), we could implement dynamically-allocated linked structures, which are constructed at run time and number of nodes can grow and shrink in time. We can also use a table structure to define the state graph, which consists of contiguous multiple identically-structured elements. Each element of the table defines one state. One or more of the entries is an index to other elements. An important factor when implementing FSMs is that there should be a clear and one-to-one mapping between the FSM state graph and the data structure. I.e., there should be one element of the structure for each state. If each state has four arrows, then each node of the linked structure should have four links.

## 6.5.2. Moore Finite State Machines

The outputs of Moore FSM are only a function of the current state. In contrast, the outputs are a function of both the input and the current state in a Mealy FSM. Often, in a Moore FSM, the specific output pattern defines what it means to be in the current state. In the first example, the inputs and outputs are simple binary numbers read from and written to a parallel port.

---

**Example 6.4.** Design a traffic light controller for the intersection of two equally busy one-way streets. The goal is to maximize traffic flow, minimize waiting time at a red light, and avoid accidents.

**Solution:** The intersection has two one-ways roads with the same amount of traffic: North and East, as shown in Figure 6.8. Controlling traffic is a good example because we all know what is supposed to happen at the intersection of two busy one-way streets. We begin the design defining what constitutes a state. In this system, a state describes which road has authority to cross the intersection. The basic idea, of course, is to prevent southbound cars to enter the intersection at the same time as westbound cars. In this system, the light pattern defines which road has right of way over the other. Since an output pattern to the lights is necessary to remain in a state, we will solve this system with a Moore FSM. It will have two inputs (car sensors on North and East roads) and six outputs (one for each light in the traffic signal.) The six traffic lights are interfaced to Port B bits 5–0, and the two sensors are connected to Port E bits 1–0,

PE1=0, PE0=0 means no cars exist on either road

PE1=0, PE0=1 means there are cars on the East road

PE1=1, PE0=0 means there are cars on the North road

PE1=1, PE0=1 means there are cars on both roads

The next step in designing the FSM is to create some states. Again, the Moore implementation was chosen because the output pattern (which lights are on) defines which state we are in. Each state is given a symbolic name:

goN ,    PB5-0 = 100001 makes it green on North and red on East

waitN ,    PB5-0 = 100010 makes it yellow on North and red on East

goE ,    PB5-0 = 001100 makes it red on North and green on East

waitE ,    PB5-0 = 010100 makes it red on North and yellow on East

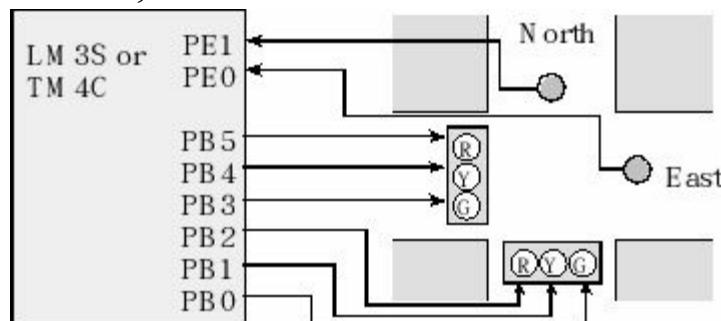


Figure 6.8. Traffic light interface with two sensors and 6 lights.

The output pattern for each state is drawn inside the state circle. The time to wait for each state is also included. How the machine operates will be dictated by the input-dependent state transitions. We create decision rules defining what to do for each possible input and for each state. For this design we can list heuristics describing how the traffic light is to operate:

If no cars are coming, stay in a green state, but which one doesn't matter.

To change from green to red, implement a yellow light of exactly 5 seconds.

Green lights will last at least 30 seconds.

If cars are only coming in one direction, move to and stay green in that direction.

If cars are coming in both directions, cycle through all four states.

Before we draw the state graph, we need to decide on the sequence of operations.

1. Initialize timer and direction registers

2. Specify initial state

3. Perform FSM controller

- a) Output to traffic lights, which depends on the state

- b) Delay, which depends on the state

- c) Input from sensors

- d) Change states, which depends on the state and the input

We implement the heuristics by defining the state transitions, as illustrated in Figure 6.9. Instead of using a graph to define the finite state machine, we could have used a table, as shown in Table 6.2.

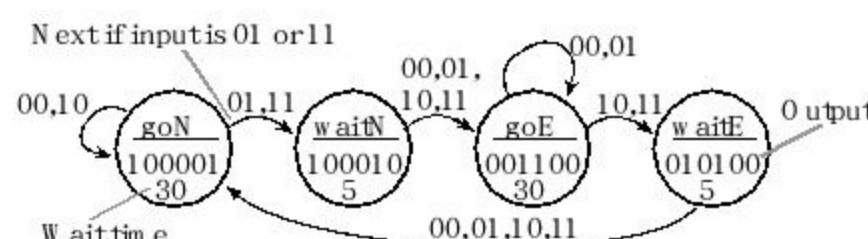


Figure 6.9. Graphical form of a Moore FSM that implements a traffic light.

State \ Input	00	01	10	11
goN (100001,30)	goN	waitN	goN	waitN
waitN (100010,5)	goE	goE	goE	goE
goE (001100,30)	goE	goE	waitE	waitE
waitE (010100,5)	goN	goN	goN	goN

Table 6.2. Tabular form of a Moore FSM that implements a traffic light.

The next step is to map the FSM graph onto a data structure that can be stored in ROM. Program 6.8 uses a linked data structure, where each state is a node, and state transitions are defined as pointers to other nodes. The four **Next** parameters define the input-dependent state transitions. The wait times are defined in the software as fixed-point decimal numbers with units of 0.01 seconds, giving a range of 10 ms to about 10 minutes. Using good labels makes the program easier to understand, in other words **goN** is more descriptive than **&fsm[0]**.

The main program begins by specifying the Port E bits 1 and 0 to be inputs and Port B bits 5–0 to be

outputs. The initial state is defined as **goN**. The main loop of our controller first outputs the desired light pattern to the six LEDs, waits for the specified amount of time, reads the sensor inputs from Port E, and then switches to the next state depending on the input data. The timer functions were presented earlier as Program 4.7. The function **SysTick\_Wait10ms** will wait 10ms times the parameter in Register R0. These two bit-specific definitions will facilitate friendly access to Ports B and E. **SENSOR** accesses PE1–PE0, and **LIGHT** accesses PB5–PB0. LM3S systems can skip initialization steps 2, 3, and 4.

```
#define SENSOR (*((volatile uint32_t *)0x4002500C))
#define LIGHT (*((volatile uint32_t *)0x400050FC))
```

<pre>;Linked data structure ;Put in ROM OUT EQU 0 ;offset for output WAIT EQU 4 ;offset for time NEXT EQU 8 ;offset for next goN DCD 0x21 ;North green, East red     DCD 3000 ;30 sec     DCD goN,waitN,goN,waitN waitN DCD 0x22 ;North yellow, East red     DCD 500 ;5 sec     DCD goE,goE,goE,goE goE DCD 0x0C ;North red, East green     DCD 3000 ;30 sec     DCD goE,goE,waitE,waitE waitE DCD 0x14 ;North red, East yellow     DCD 500 ;5 sec     DCD goN,goN,goN,goN Start     BL PLL_Init      ;50 MHz clock     BL SysTick_Init ;enable SysTick     LDR R1, =SYSCTL_RCGCGPIO_R     LDR R0, [R1]     ORR R0, R0, #0x12 ; activate B E     STR R0, [R1]     NOP     NOP ; allow time to finish     LDR R1, =GPIO PORTE_AMSEL_R     LDR R0, [R1]     BIC R0, R0, #0x03 ; no analog     STR R0, [R1]     LDR R1, =GPIO PORTE_PCTL_R     LDR R0, [R1]     BIC R0, R0, #0x000000FF ; PE1-0     STR R0, [R1]     LDR R1, =GPIO PORTE_DIR_R     LDR R0, [R1]     BIC R0, R0, #0x03 ; PE1-0 input     STR R0, [R1]     LDR R1, =GPIO PORTE_AFSEL_R     LDR R0, [R1]     BIC R0, R0, #0x03 ; no alt funct     STR R0, [R1]     LDR R1, =GPIO PORTE_DEN_R</pre>	<pre>// Linked data structure struct State {     uint32_t Out;     uint32_t Time;     const struct State *Next[4]; } typedef const struct State STyp; #define goN &amp;FSM[0] #define waitN &amp;FSM[1] #define goE &amp;FSM[2] #define waitE &amp;FSM[3] STyp FSM[4] = { {0x21,3000,{goN,waitN,goN,waitN}}, {0x22, 500,{goE,goE,goE,goE}}, {0x0C,3000,{goE,goE,waitE,waitE}}, {0x14, 500,{goN,goN,goN,goN}}};  int main(void) {     STyp *Pt; // state pointer     uint32_t Input;     PLL_Init(); // 50 MHz, Prog 4.6     SysTick_Init(); // Program 4.7     SYSCTL_RCGCGPIO_R  = 0x12; // 1) B E     while((SYSCTL_PGPIO_R&amp;0x12)         != 0x12){};// ready?     // 2) no need to unlock     // 3) disable analog function on PE1-0     GPIO_PORTE_AMSEL_R &amp;= ~0x03;     // 4) enable regular GPIO     GPIO_PORTE_PCTL_R &amp;=     ~0x000000FF;     // 5) inputs on PE1-0     GPIO_PORTE_DIR_R &amp;= ~0x03;     // 6) regular function on PE1-0     GPIO_PORTE_AFSEL_R &amp;= ~0x03;     // 7) enable digital on PE1-0     GPIO_PORTE_DEN_R  = 0x03;      // 3) disable analog function on PB5-0     GPIO_PORTB_AMSEL_R &amp;= ~0x3F;     // 4) enable regular GPIO     GPIO_PORTB_PCTL_R &amp;=     ~0x00FFFFFF;     // 5) outputs on PB5-0</pre>
--	--

```

LDR R0, [R1]
ORR R0, R0, #0x03 ; enable PE1-0
STR R0, [R1]
LDR R1, =GPIO_PORTB_AMSEL_R
LDR R0, [R1]
BIC R0, R0, #0x3F ; no analog
STR R0, [R1]
LDR R1, =GPIO_PORTB_PCTL_R
LDR R0, [R1]
BIC R0, R0, #0x00FFFFFF ; PB5-0
STR R0, [R1]
LDR R1, =GPIO_PORTB_DIR_R
LDR R0, [R1]
ORR R0, R0, #0x3F ; PB5-0 output
STR R0, [R1]
LDR R1, =GPIO_PORTB_AFSEL_R
LDR R0, [R1]
BIC R0, R0, #0x3F ; no alt funct
STR R0, [R1]
LDR R1, =GPIO_PORTB_DEN_R
LDR R0, [R1]
ORR R0, R0, #0x3F ; enable PB5-0
STR R0, [R1]
LDR R4,=goN ;state pointer
LDR R5,=SENSOR ;0x4002500C
LDR R6,=LIGHT ;0x400050FC
FSM LDR R0,[R4,#OUT] ;output value
STR R0,[R6] ;set lights
LDR R0,[R4,#WAIT] ;time delay
BL SysTick_Wait10ms
LDR R0,[R5] ;read input
LSL R0,R0,#2 ;4 bytes/address
ADD R0,R0,#NEXT ;8,12,16,20
LDR R4,[R4,R0] ;go to next state
B FSM

```

Pt = goN;

```

while(1){
    LIGHT = Pt->Out; // set lights
    SysTick_Wait10ms(Pt->Time);
    Input = SENSOR; // read sensors
    Pt = Pt->Next[Input];
}

```

## Program 6.8. Linked data structure implementation of the traffic light controller (PointerTrafficLightxxx.zip).

In order to make it easier to understand, which will simplify verification and modification, we have made a 1-to-1 correspondence between the state graph in Figure 6.9 and the **fsm[4]** data structure in Program 6.8. Notice also how this implementation separates the civil engineering policies (the data structure specifies what the machine does), from the computer engineering mechanisms (the executing software specifies how it is done.) Once we have proven the executing software to be operational, we can modify the policies and be confident that the mechanisms will still work. When an accident occurs, we can blame the civil engineer that designed the state graph.

On microcontrollers that have EEPROM, we can place the FSM data structure in EEPROM. This allows us to make minor modifications to the finite state machine (add/delete states, change input/output values) by changing the data structure. In this way small modifications/upgrades/options to the finite state machine can be made by reprogramming the EEPROM reusing the hardware.

The FSM approach makes it easy to change. To change the wait time for a state, we simply change the value in the data structure. To add more states (e.g., put a red/red state after each yellow state, which will reduce accidents caused by bad drivers running the yellow light), we simply increase the size of the **fsm[]** structure and define the **Out**, **Time**, and **Next** fields for these new states.

To add more output signals (e.g., walk and left turn lights), we simply increase the precision of the **Out** field. To add two more input lines (e.g., wait button, left turn car sensor), we increase the size of the next field to **Next[16]**. Because now there are four input lines, there are 16 possible combinations, where each input possibility requires a **Next** value specifying where to go if this combination occurs. In this simple scheme, the size of the **Next[]** field will be 2 raised to the power of the number of input signals.

**Checkpoint 6.1:** Why is it good to use labels for the states? E.g., **goN** is better than **&fsm[0]**.

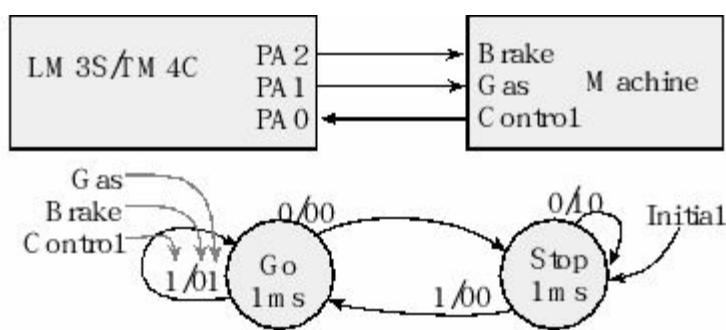
**Observation:** In order to make the FSM respond quicker, we could implement a time delay function that returns immediately if an alarm condition occurs. If no alarm exists, it waits the specified delay.

### 6.5.3. Mealy Finite State Machines

A Mealy FSM has the outputs that depend on both the input and the current state. The state transition arrows in a Mealy FSM specify both the output and the next state. In general, we employ a Mealy machine when the output is needed to cause the state to change. We will implement the following Mealy Finite State machine using a linked structure. There is one input and two outputs. The initial state is Stop. The heuristics of this machine are:

If we are stopped, and the control is low, we press the brake and remain stopped.  
If we are stopped, and the control is high, we release the brake and begin to go.  
If we are going, and the control is low, we release the gas and begin to stop.  
If we are going, and the control is high, we press the gas and continue to go.  
There must be at least 1 ms of no brake, no gas as it switches between go and stop.

This is a Mealy FSM because the 00 output pattern is needed to change states (Stop to Go, or Go to Stop). There is a 1 ms wait parameter for this simple machine. Most controllers that manipulate real hardware do require well-specified delays from input to output, which are implemented as delays in the states. The hardware and FSM are shown in Figure 6.10.



## Figure 6.10. Engine controller implemented with a Mealy FSM.

There is one input, **Control**, connected to PA0. There are two outputs, **Brake** connected to PA2, and **Gas** connected to PA1. Each state has two next states and two outputs which depend on the current input. The controller continuously repeats the sequence

Input from **Control** (PA0)

Output to **Brake**, **Gas** (PA2,PA1) which depends on the input **Control**

Delay as specified by this state

Next state which depends on the input **Control**

E.g., if the state is in **Stop**, and the **Control** is 0, then the **Brake** output is 1 and the **Gas** output is 0 and the next state is **Stop**. Program 6.9 shows the implementation. The main program begins by specifying the Port A bit 0 to be input and Port A bits 2–1 to be outputs. The initial state is defined as **Stop**. The main loop of our controller first inputs from PA0, performs an output based on the input, and then switches to the next state depending on the input data. These two bit-specific definitions will facilitate friendly access to Port A. **INPUT** accesses PA0, and **OUTPUT** accesses PA2–PA1.

```
#define INPUT (*((volatile uint32_t *)0x40004004))
#define OUTPUT (*((volatile uint32_t *)0x40004018))
```

```
;Linked data structure
;Put in ROM
OUT EQU 0 ;offset for output
DELAY EQU 8 ;offset for delay
NEXT EQU 12 ;offset for next
Stop DCD 2,0 ;Outputs for 0,1
    DCD 50000 ;1 ms
    DCD Stop,Go ;Next for 0,1
Go DCD 0,1 ;Outputs for 0,1
    DCD 50000 ;1 ms
    DCD Stop,Go ;Next for 0,1
Start
    BL PLL_Init ; 50 MHz clock
    BL SysTick_Init ; enable SysTick
    LDR R1, =SYSCTL_RCGCGPIO_R
    LDR R0, [R1]
    ORR R0, R0, #0x01 ; activate A
    STR R0, [R1]
    NOP
    NOP ; allow time to finish
    LDR R1, =GPIO_PORTA_AMSEL_R
    LDR R0, [R1]
    BIC R0, R0, #0x07 ; no analog
    STR R0, [R1]
    LDR R1, =GPIO_PORTA_PCTL_R
    LDR R0, [R1]
    MOV R2, #0x00000FFF
    BIC R0, R0, R2 ; GPIO PA2-0
    STR R0, [R1]
    LDR R1, =GPIO_PORTA_DIR_R
```

```
// Linked data structure
struct State {
    uint32_t Out[2];
    uint32_t Delay;
    const struct State *Next[2];
} typedef const struct State STyp;
#define Stop &FSM[0]
#define Go &FSM[1]
STyp FSM[2]={
{{2,0},50000,{Stop,Go}},
{{0,1},50000,{Stop,Go}}};

int main(void){
    STyp *Pt; // state pointer
    uint32_t Input;
    PLL_Init(); // 50 MHz, Prog 4.6
    SysTick_Init(); // program 4.7
    // 1) activate port A
    SYSCTL_RCGCGPIO_R |= 0x01;
    while((SYSCTL_PRGPI0_R&0x01) == 0)
    {};

    // 2) no need to unlock Port A

    // 3) disable analog on PA2-0
    GPIO_PORTA_AMSEL_R &= ~0x07;

    // 4) configure PA2-0 as GPIO
    GPIO_PORTA_PCTL_R &=
~0x00000FFF;
```

```

LDR R0, [R1]
BIC R0, R0, #0x01 ; PA0 input
ORR R0, R0, #0x06 ; PA2-1 output
STR R0, [R1]
LDR R1, =GPIO_PORTA_AFSEL_R
LDR R0, [R1]
BIC R0, R0, #0x07 ; no alt funct
STR R0, [R1]
LDR R1, =GPIO_PORTA_DEN_R
LDR R0, [R1]
ORR R0, R0, #0x07 ; enable PA2-0
STR R0, [R1]
LDR R4,=Stop ; State pointer
LDR R5,=INPUT ; 0x40004004
LDR R6,=OUTPUT ; 0x40004018
FSM LDR R3,[R5] ; Read input
LSL R3,R3,#2 ; 4 bytes each
ADD R1,R3,#OUT ; R1 is 0 or 4
LDR R2,[R4,R1] ; Output value
LSL R2,R2,#1 ; into bits 2,1
STR R2,[R6] ; set outputs
LDR R0,[R4,#DELAY] ; 20ns delays
ADD R1,R3,#NEXT ;R1 is 12 or 16
LDR R4,[R4,R1] ;find next state
BL SysTick_Wait ;program 4.7
B FSM

```

```

// 5) make PA0 in and PA2-1 out
GPIO_PORTA_DIR_R &= ~0x01;
GPIO_PORTA_DIR_R |= 0x06;

// 6) disable alt func on PA2-0
GPIO_PORTA_AFSEL_R &= ~0x07;

// 7) enable digital I/O on PA2-0
GPIO_PORTA_DEN_R |= 0x07;

```

Pt = Stop; // initial state: stopped

```

while(1){
    Input = INPUT;
    // get new input from Control
    OUTPUT = Pt->Out[Input]<<1;
    // output depends on input and state
    SysTick_Wait(Pt->Delay); // wait
    Pt = Pt->Next[Input]; // next
}
}

```

Program 6.9. Linked data structure implementation of the motor controller (MealyEngineControl\_xxx.zip).

# 6.6. \*Dynamically Allocated Data Structures

In order to reuse memory and provide for efficient use of RAM, we need dynamic memory allocation. The previous examples in this chapter used fixed allocation, meaning the size of the data structures is decided in advance and specified in the source code. In addition, the location of these structures is determined by the assembler at assembly time. With a dynamic allocation the size and location will be determined at run time. To implement dynamic allocation we will manage a heap. The heap is a chunk of RAM that is

1. Dynamically allocated by the program when it creates the data structure
2. Used by the program to store information
3. Dynamically released by the program when the structure is no longer needed

The heap manager provides the system two operations:

```
pt = malloc(size); // returns a pointer to a block of size bytes  
free(pt); // deallocates the block at pt
```

The implementation of this general memory manager is beyond the scope of this book. Instead, we will develop a very useful, but simple heap manager with these two operations:

```
pt = Heap_Allocate(); // returns a pointer to a block of fixed size  
Heap_Release(pt); // deallocates the block at pt
```

Once we allocate space, it is important to keep track of the pointer to that space. If we lose the pointer, that space is lost forever. Similarly, once we are finished using the space, we should deallocate that space. If we continue to allocate space without ever deallocating it, then we will run out of space and the system will crash.

## 6.6.1. \*Fixed Block Memory Manager

In general, the heapmanager allows the program to allocate a variable block size, but in this section we will develop a simplified heap manager handles just fixed size blocks. In this example, the block size is specified by **SIZE**. The initialization will create a **linked list** of all the free blocks (Figure 6.11). A list is a collection of dissimilar objects, typically implemented in C with **struct**. In this case, the list is an array where the first element is a pointer, and the remaining elements are the memory to be allocated. A linked list is a collection of lists that are connected together with pointers, as shown in Figure 6.11. Programs 6.10 through 6.13 combine to create a simple memory manager.

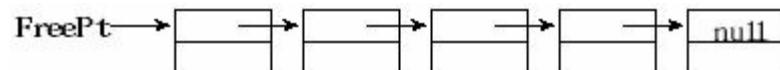


Figure 6.11. The initial state of the heap has all of the free blocks linked in a list.

Program 6.10 shows the global structures for the heap. These entries are defined in RAM. **SIZE** is the number of 32-bit words in each block. All blocks allocated and released with this memory manager will be of this fixed size. **NUM** is the number of blocks to be managed. **FreePt** points to the first free block. **NULL** is defined as 0, meaning undefined pointer.

<b>SIZE EQU 4</b>	<b>#define SIZE 4</b>
<b>NUM EQU 5</b>	<b>#define NUM 5</b>
<b>NULL EQU 0</b>	<b>#define NULL 0 // empty pointer</b>
<b>FreePt SPACE 4</b>	<b>int32_t *FreePt;</b>
<b>Heap SPACE SIZE*NUM*4</b>	<b>int32_t Heap[SIZE*NUM];</b>

Program 6.10. Private global structures for the fixed-block memory manager (HeapFixedBlock\_xxx.zip).

Initialization must be performed before the heap can be used. Program 6.11 shows the software that partitions the heap into blocks and links them together, creating the situation illustrated in Figure 6.11. **FreePt** points to a linear linked list of free blocks. Initially these free blocks are contiguous and in order, but as the manager is used the positions and order of the free blocks can vary. It will be the pointers that will thread the free blocks together.

<b>Heap_Init LDR R0,=Heap</b>	<b>void Heap_Init(void){ int i;</b>
<b>LDR R1,=FreePt</b>	<b>int32_t *pt;</b>
<b>STR R0,[R1] ;FreePt=&amp;Heap[0];</b>	<b>pt = FreePt = &amp;Heap[0];</b>
<b>MOV R2,#SIZE</b>	<b>for(i=1; i&lt;NUM; i++){</b>
<b>MOV R3,#NUM-1</b>	<b>*pt = (int32_t)(pt+SIZE);</b>
<b>imLoop ADD R1,R0,R2,LSL #2 ;pt+SIZE</b>	<b>pt = pt+SIZE;</b>
<b>STR R1,[R0] ;*pt=pt+SIZE;</b>	<b>}</b>
<b>MOV R0,R1 ;pt=pt+SIZE;</b>	<b>*(int32_t*)pt = NULL;</b>
<b>SUBS R3,R3,#1</b>	<b>}</b>
<b>BNE imLoop</b>	
<b>MOV R1,#NULL</b>	
<b>STR R1,[R0] ;last ptr is NULL</b>	
<b>BX LR</b>	

Program 6.11. Functions to initialize the heap (HeapFixedBlock\_xxx.zip).

To allocate a block, the manager just removes one block from the front of the free list. Program 6.12 shows the allocate function. The **Heap\_Allocate** function will fail and return a null pointer when the heap becomes empty.

<b>Heap_Allocate ; R0 points to new block</b>	<b>int32_t *Heap_Allocate(void){</b>
<b>LDR R1,=FreePt</b>	<b>int32_t *pt;</b>
<b>LDR R0,[R1] ;R0=FreePt;</b>	<b>pt = FreePt;</b>
<b>CMP R0,#NULL</b>	<b>if(pt != NULL){</b>
<b>BEQ aDone ;if(pt!=NULL)</b>	<b>FreePt = (int32_t*)*pt; // next</b>
<b>LDR R2,[R0] ;link next</b>	<b>}</b>
<b>STR R2,[R1] ;FreePt=*pt;</b>	<b>return(pt);</b>
<b>aDone BX LR</b>	<b>}</b>

Program 6.12. Function to allocate memory blocks (returns NULL if empty) (HeapFixedBlock\_xxx.zip).

The **Heap\_Release** returns a block to the free list, see Program 6.13. This function will link it to the front of the list. This system does not check to verify a released block actually was previously allocated.

```

; R0 => block being released
Heap_Release
    LDR R1,=FreePt
    LDR R2,[R1] ;R2=oldFreePt
    STR R0,[R1] ;FreePt=pt
    STR R2,[R0] ;*pt=oldFreePt;
    BX LR

```

```

void Heap_Release(int32_t *pt){
    int32_t *oldFreePt;
    oldFreePt = FreePt;
    FreePt = pt;
    *pt = (int32_t)oldFreePt;
}

```

Program 6.13. Function to allocate and release memory blocks (HeapFixedBlock\_xxx.zip).

**Checkpoint 6.2:** Consider a system that needs variable-size memory allocation, where the size can range from 2 to a maximum of 20 words. How might this simple heap be used?

## 6.6.2. \*Linked List FIFO

An example application of a dynamically allocated data structure is a first in first out queue (FIFO). The FIFO is used to pass data from one module to another. One module creates data (the producer) and puts it into the FIFO. A second module will get data from the FIFO and perform operations on the data (the consumer). Data are passed from the producer to the consumer in an order-preserved manner. In this implementation, **GetPt** points to the oldest node (the one to get next) and **PutPt** points to the newest node, the place to add more data. The pointer for the newest node (if it exists) is a null. The **Fifo\_Put** operation fails (full) when the heap runs out of space. The **Fifo\_Get** operation fails (empty) when **GetPt** equals **NULL**. Program 6.14 shows the global variables defined in RAM.

<pre> ;put in RAM Next EQU 0 ;next Data EQU 4 ;32-bit data for node GetPt SPACE 4 ; GetPt is pointer to oldest node PutPt SPACE 4 ; PutPt is pointer to newest node </pre>	<pre> struct Node{     struct Node *Next;     int32_t Data; };  typedef struct Node NodeType; NodeType *PutPt; // place to put NodeType *GetPt; // place to get </pre>
--	--

Program 6.14. Definition of the linked list structure (LLFifo\_xxx.zip).

Figure 6.12 shows an example FIFO with three elements (after running with lots of putting and getting). In this example, element 1 is the oldest because it was put first. This system uses Programs 6.11, 6.12, 6.13, and 6.14 with **SIZE** equal to 2 words.

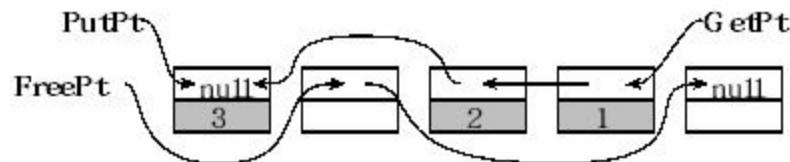


Figure 6.12. A linked list FIFO after putting 1,2,3.

Program 6.15 shows the functions to initialize the FIFO. When the **GetPt** is **NULL**, the FIFO is empty. Since there is no newest node, the **PutPt** is also **NULL**.

```

Fifo_Init
PUSH {R4,LR}
MOV R1,#NULL

```

```

void Fifo_Init(void){
    GetPt = NULL; // Empty when null
    PutPt = NULL;
}

```

```

LDR R0,=GetPt
STR R1,[R0] ;GetPt=NULL
LDR R0,=PutPt
STR R1,[R0] ;PutPt=NULL
BL Heap_Init
POP {R4,PC}

```

```

    Heap_Init();
}

```

## Program 6.15. Initialization of the linked list FIFO (LLFifo\_xxx.zip).

Figure 6.13 is a flowchart of the Put and Get functions. The Put function is shown in Program 6.16. The first step is to allocate a new block. The FIFO is full only when the heap is full (**Heap\_Allocate** returns a failure). The Put operation first allocates space for the new entry, and then stores the new information into the **Data** field. Since this element will be last, its **Next** field is set to null. The last part of Put links this new node at the end of the linked list.

```

; Inputs: R0 value, data to put
; Outputs: R0=1 if successful
;          R0=0 if unsuccessful
Fifo_Put
    PUSH {R4,LR}
    MOV R4,R0 ;data to put
    BL Heap_Allocate
    CMP R0,#NULL
    BEQ PFul ;skip if full
    STR R4,[R0,#Data] ;store data
    MOV R1,#NULL
    STR R1,[R0,#Next] ;next=NULL
    LDR R2,=PutPt ;R2 = &PutPt
    LDR R3,[R2] ;R3 = PutPt
    CMP R3,#NULL ;previously MT?
    BEQ PMT
    STR R0,[R3,#Next] ;link previous
    B PCon
PMT LDR R1,=GetPt ;R1 = &GetPt
    STR R0,[R1] ;Now one entry
PCon STR R0,[R2] ;points to newest
    MOV R0,#1 ;success
    B PDon
PFul MOV R0,#0 ;failure, full
PDon POP {R4,PC}

```

```

int Fifo_Put(int32_t theData){
    NodeType *pt;
    pt = (NodeType*)Heap_Allocate();
    if(!pt){
        return(0); // full
    }
    pt->Data = theData; // store
    pt->Next = NULL;
    if(PutPt){
        PutPt->Next = pt; // Link
    }
    else{
        GetPt = pt; // first one
    }
    PutPt = pt;
    return(1); // successful
}

```

## Program 6.16. Implementation of the Put function for the linked list FIFO (LLFifo\_xxx.zip).

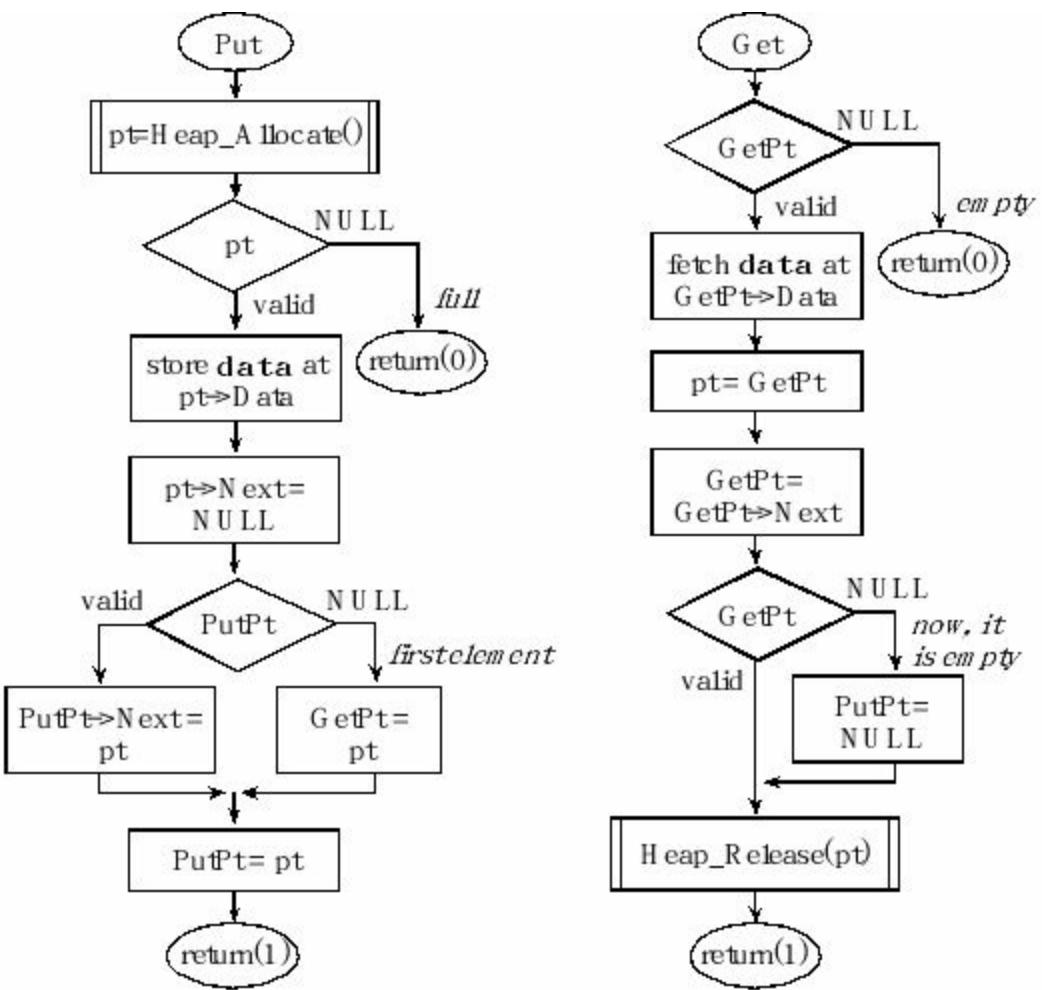


Figure 6.13. Flowcharts of a linked list FIFO Put and Get operations.

One of the difficulties with Get is the need to return two parameters. The R0 parameter (the regular return parameter in C) returns 0 if unsuccessful and 1 if successful. If successful, a second return parameter will contain the data. For this implementation, the second parameter is implemented with call by reference. This means the calling program passes a pointer to an empty variable and the Get function will place the data into this empty space. An example of how to call these routines can be found in the projects LLFifo\_xxx.zip

```

int main(void){ int32_t r1,r2,d1;
Fifo_Init(); // initialize linked list FIFO
r1 = Fifo_Put(55); // save the number 55
r2 = Fifo_Get(&d1); // d1 will be 55, r2=1
while(1){}
}

```

The Get function first checks to make sure the FIFO is not empty, see Program 6.17. Next, the **Data** field is retrieved from the node. This node is then unlinked from the linked list, and the memory block is released to the heap. There is a special case that handles the situation where you get the one remaining node in the linked list. In this case both **PutPt** and **GetPt** point to this node. When you get this node, both **PutPt** and **GetPt** are set to null, signifying the FIFO is now empty.

<p>; Inputs: R0 points to an empty place  ; Outputs: data removed to place  ; R0=0 if successful  ; R0=1 if empty</p>
---

<pre> int Fifo_Get(int32_t *datapt){ Node *pt; if(!GetPt){     return(0); // empty } </pre>
---

```

Fifo_Get
PUSH {R4,LR}
LDR R1,=GetPt ;R1 = &GetPt
LDR R2,[R1] ;R2 = GetPt
CMP R2,#NULL
BEQ GMT ;empty if NULL
LDR R3,[R2,#Data] ;read
STR R3,[R0] ;by reference
LDR R3,[R2,#Next] ;next link
STR R3,[R1] ; update GetPt
CMP R3,#NULL
BNE GCon
LDR R1,=PutPt ;Now empty
STR R3,[R1] ;update PutPt
GCon MOV R0,R2 ;old data
BL Heap_Release
MOV R0,#1 ;success
B GDon
GMT MOV R0,#0 ;failure, empty
GDon POP {R4,PC}
}
*datapt = GetPt->Data;
pt = GetPt;
GetPt = GetPt->Next;
if(GetPt==NULL){ // one entry
    PutPt = NULL;
}
Heap_Release((int32_t*)pt);
return(1); // success
}

```

Program 6.17. Implementation of the Get function for the linked list FIFO (LLFifo\_xxx.zip).

# 6.7. Matrices and Graphics

A **matrix** is a two-dimensional data structure accessed by row and column. Each element of a matrix is the same type and precision. In C, we create matrices using two sets of brackets. Figure 6.14 shows this byte matrix with six 8-bit elements. The figure also shows two possible ways to map the two-dimensional data structure into the linear address space of memory.

```
uint8_t M[2][3]; // byte matrix with 2 rows and 3 columns
```

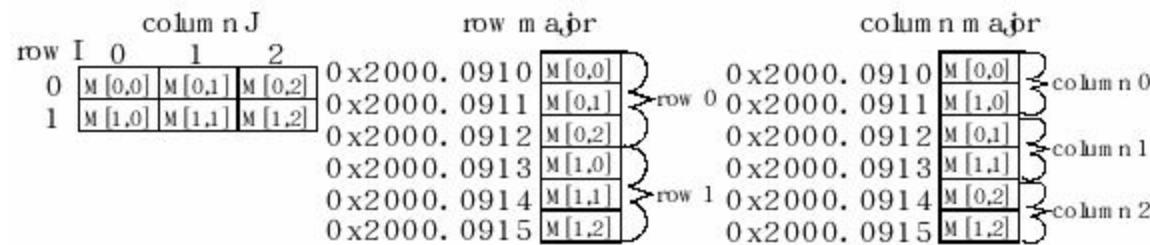


Figure 6.14. A byte matrix with 2 rows and 3 columns.

With row-major allocation, the elements of each row are stored together. Let **i** be the row index, **j** be the column index, **n** be the number of bytes in each row (equal to the number of columns), and **Base** is the base address of the byte matrix, then the address of the element at **i,j** is

```
Base+n*i+j
```

An assembly language subroutine that reads elements from this row-major matrix is shown in Program 6.18. The matrix data is passed using call by reference, and the indices are passed using call by value. The row index (0 or 1) is passed in Register R1. The column index (0, 1, or 2) is passed in Register R2. The base address of the matrix is passed in Register R0. The subroutine returns the value in Register R0. This function works for a 2 by 3 matrix

```
uint8_t M[2][3]; // byte matrix, 2 rows and 3 columns
```

<pre>; Read an 8-bit value from (i, j) ; Input: Base (R0) pointer to matrix ;      i (R1) is the row index ;      j (R2) is the column index ; Output: R0 is retrieved value ; Assumes: (0 &lt;= R1 &lt;= 1) ;      and (0 &lt;= R2 &lt;= 2) Matrix_Read     MOV R3, #3      ; R3 = 3 col     MUL R3, R3, R1 ; R3 = 3*i     ADD R3, R3, R2 ; R3 = 3*i+j     ADD R3, R3, R0 ; R3 = Base+3*i+j     LDRB R0, [R3]   ; R0 = M[i,j]     BX LR          ; return</pre>	<pre>// Read an 8-bit value from (i, j) // row or column major by compiler. // Input: i is the row index //         j is the column index // Output: retrieved value // Assumes: (0 &lt;= i &lt;= 1) and //         (0 &lt;= j &lt;= 2) uint8_t Matrix_Read(     uint8_t base[][],     uint8_t i, uint8_t j){     return base[i][j]; }</pre>
--	--

Program 6.18. Function to access a two by three row-major matrix.

With column-major allocation, the elements of each column are stored together. Let **i** be the row index, **j** be the column index, **m** be the number of bytes in each column (equal to the number of rows), and **Base** is the base address of the byte matrix, then the address of the element at **i,j** is

**Base+m\*j+i**

With a halfwordmatrix, each element requires two bytes of storage. Let **i** be the row index, **j** be the column index, **n** be the number of halfwords in each row (equal to the number of columns), and **Base** is the base address of the word matrix, then the address of the element at **i,j** is

**Base+2\*(n\*i+j)**

With a wordmatrix, each element requires four bytes of storage. Let **i** be the row index, **j** be the column index, **n** be the number of words in each row (equal to the number of columns), and **Base** is the base address of the word matrix, then the address of the element at **i,j** is

**Base+4\*(n\*i+j)**

As an example of a matrix, we will develop a set of driver functions to manipulate a 48 by 84 by 1-bit graphics display. The Nokia 5110 is 48 by 84 pixel LCD. We will use a bit array to store pixel values for the LCD, see Figure 6.15.

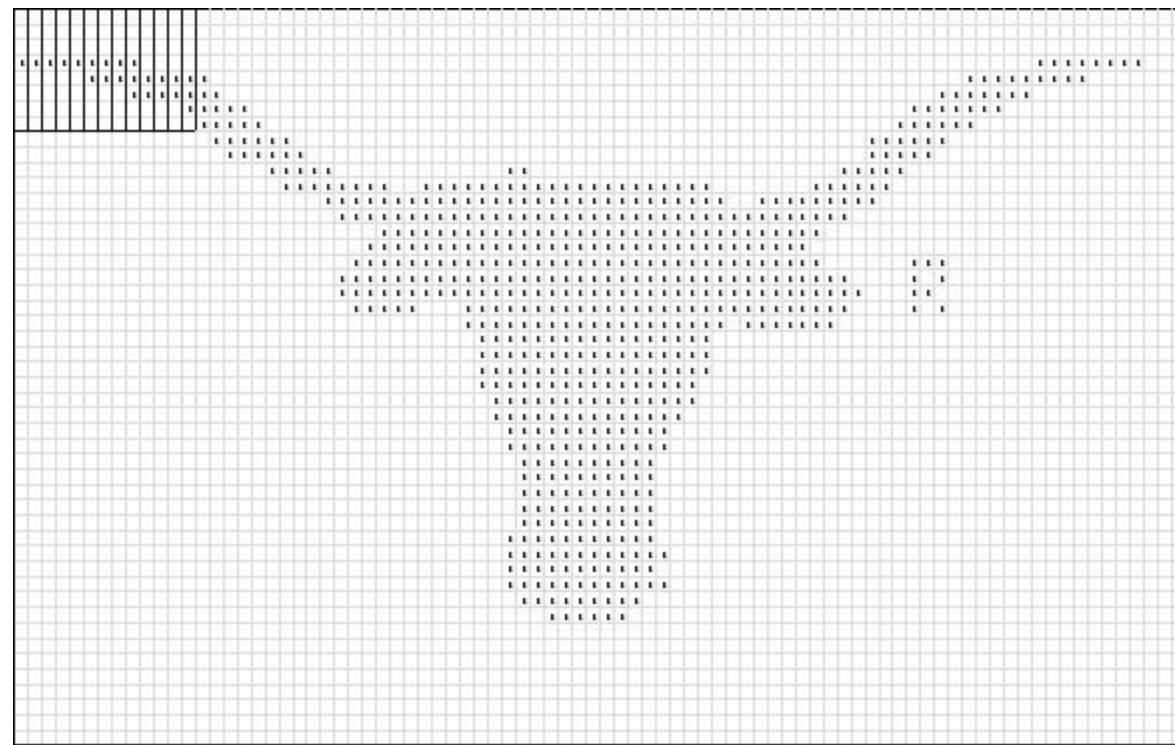


Figure 6.15. A 1-bit matrix with 48 rows and 84 columns, each pixel is 1 bit. The first few bytes are outlined in the top left: 0x08, 0x08, 0x08, 0x08, 0x08, 0x18, 0x18, 0x18, 0x38, 0x38, 0x30, 0x30, 0x30, 0x70.

Placing a 0 into a pixel location will turn the pixel off and a placing a 1 will turn it on. In this display, the first bit is the top left corner of the display, and the last bit is the bottom right corner. The graphical image on this 48 by 84 display will be stored in the 1-bit array called **Screen**. Since there are a total of 4032 pixels, and each byte can store 8 pixels, we need 504 bytes to store the entire image. We define an array in global RAM to hold one image,

```
uint8_t Screen[504]; // stores the next image to be printed on the screen
```

An implementation of this example can be found in Nokia5110\_xxx.zip. Similar graphics driver

implementations for the Kentec and ST7735 LCDs can be found as Kentec\_xxx.zip and ST7735\_xxx.zip. The description in this section apply to the Nokia 5110, but the general approach is similar in all three displays. The Figure 6.15 was created using the Excel file 48x84 Image.xls. On the Nokia 5110 LCD, 8 pixels are packed into a byte such that all pixels of one byte exist in the same column, as shown in the upper left corner of Figure 6.15. E.g., the letter ‘B’ is created with 6 columns and 8 rows: 0x7f, 0x49, 0x49, 0x49, 0x36, 0x00. Since characters are 6 pixels wide,  $84/6=12$  characters fit per row. Since characters are 8 pixels tall, there can be  $48/8=6$  rows of ASCII text.

0	0	0	0	0	0
1	1	1	1	0	0
1	0	0	0	1	0
1	0	0	0	1	0
1	1	1	1	0	0
1	0	0	0	1	0
1	0	0	0	1	0
1	1	1	1	0	0

The basic idea of graphics is to create the desired image in a RAM buffer by clearing and setting bits in the buffer. Once the image is complete it is punched into the display by transferring the data to the hardware display. Once data is transferred, the graphics hardware will send the pixels to the display over and over so it looks like a fixed image to human eyes. When we wish to change the image, the software creates another image and sends it to the display.

The function **Nokia5110\_DisplayBuffer** sends the **Screen** buffer to the LCD. Let **i** be the row index (y-coordinate), where **i** ranges from 0 to 47. 0 is on the top and 47 is on the bottom. Let **j** be the column index (x-coordinate), where **j** ranges from 0 to 83. 0 is on the left and 83 is on the right. There are 84 bytes creating a 8-pixel high 84-pixel wide group. Therefore, the starting address of the group containing **i,j** is

**Screen + 84\*(i>>3)**

where right shift is integer math without rounding. Notice that if **i** is 0 to 47, then **(i>>3)** will be 0 to 5. The column index specifies which byte. The address of the byte containing the information at **(i,j)** is

**Screen + 84\*(i>>3) + j**

Let **k** be the bottom 3 bits of **i**.

**k = i&0x07;**

A mask will specify the bit location within the byte. This array can be defined in ROM as

**Masks FCB 0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80**

For example, if **k** is 0 then we use the bit mask of 0x01 to access the information stored in the appropriate bit of the **Screen** buffer. Program 6.19 takes the row and column index values and calculates the memory address and bit mask to access that pixel in the **Screen** matrix. **Access** is a private function for this module. A **helper function** is another name for private functions used inside a module, but not called by software outside the module. Conversely, the other four functions of this module are public.

```

; Input: R0 the row index i (0 to 47 in this case), y-coordinate
;       R1 the column index j (0 to 83 in this case), x-coordinate
; Output: R0 points to the byte of interest
;       R1 the Mask to access that 1-bit pixel
Access ; Access the Image pixel at (i, j).
LDR R3, =Masks ; R3 = Masks (pointer)
AND R2, R0, #0x07 ; R2 = k = i&0x07
LDRB R2, [R3, R2] ; R2 = Masks[k]
LDR R3, =Screen ; R3 = Screen (pointer)
LSR R0, R0, #3 ; R0 = R0>>3 (i>>3), divide by 8
MOV R2, #84
MUL R2, R0, R2 ; R2 = 84*(i>>3)
ADD R0, R2, R3 ; R0 = Screen + 84*(i>>3) (pointer)
ADD R0, R0, R1 ; R0 = Screen + 84*(i>>3)+j (pointer)
MOV R1, R2 ; R1 is Mask
BX LR

```

Program 6.19. A helper function to access a bit-matrix.

Functions to clear, and set bits in the **Screen** buffer are shown in Program 6.20. For the public functions, the parameters **i**, **j** as passed by value, and the video buffer itself is a private global within this module. The public functions are callable from C or assembly. For example to set the pixel in row=5 column=3, we could call **Nokia5110\_SetPxl(5,3);**

```

-----Nokia5110_ClrPxl-----
; Clear the Image pixel at (i, j), turning it dark.
; Input: R0 the row index i (0 to 47 in this case), y-coordinate
;       R1 the column index j (0 to 83 in this case), x-coordinate
; Output: none ; Modifies: R0,R1,R2,R3
Nokia5110_ClrPxl
PUSH {LR}
BL Access ; get pointer to pixel to change
LDRB R3, [R0] ; R3 = [R0] = 8 pixels
BIC R3, R3, R1 ; R3 = R3&~R1 (clear proper pixel, save other)
STRB R3, [R0] ; [R0] = R3 = 8 pixels
POP {PC}
----- Nokia5110_SetPxl-----
; Set the Image pixel at (i, j) to the given value.
; Input: R0 the row index i (0 to 47 in this case), y-coordinate
;       R1 the column index j (0 to 83 in this case), x-coordinate
; Output: none ; Modifies: R0,R1,R2,R3
Nokia5110_SetPxl
PUSH {LR}
BL Access ; get pointer to pixel to change
LDRB R3, [R0] ; R3 = [R0] = 8 pixels

```

**ORR R3, R3, R1 ; R3 = R3|R1 (set proper pixel, save other)**

**STRB R3, [R0] ; [R0] = R3 = write 8 pixels**

**POP {PC}**

## Program 6.20. Functions that modify the bit-matrix.

In order for the image to appear on the display, there must be a hardware interface that translates the data in the video buffer onto the graphics hardware. A typical way this translation occurs is for the video buffer to exist in the display hardware itself. The software reads and writes this buffer in a similar way as described in this example. The graphics hardware is then responsible for copying the data from the buffer onto the display. These 7 functions operate directly on the LCD, and will be described in more detail later in Section 8.4.

```
void Nokia5110_Init(void);
void Nokia5110_OutChar(char data);
void Nokia5110_OutString(char *ptr);
void Nokia5110_OutUDec(uint16_t n);
void Nokia5110_SetCursor(uint8_t newX, uint8_t newY);
void Nokia5110_Clear(void);
void Nokia5110_DrawFullImage(const uint8_t *ptr);
```

These 5 functions operate on the **Screen** buffer. 16-color BMP files store images in a matrix format very similar to Figure 6.15. There is also a file converter that translates standard 16-color BMP files to C code that can be displayed on the LCD.

```
void Nokia5110_ClearBuffer(void);
void Nokia5110_ClrPxl(uint32_t i, uint32_t j);
void Nokia5110_SetPxl(uint32_t i, uint32_t j);
void Nokia5110_PrintBMP(uint8_t xpos, uint8_t ypos,
                        const uint8_t *ptr, uint8_t threshold);
void Nokia5110_DisplayBuffer(void);
```

## 6.8. \*Tables

A **table** is a collection of identically-sized structures. Program 6.21 and Figure 6.16 show a table containing a simple data base. Each entry in the table records the name, life span, and the year of inauguration. The names are variable length, but a fixed size will be allocated so that each table entry will be exactly 36 bytes. The C compiler will fill the unused bytes in the Name field with zeros. Each entry must be the same size, so we can calculate the address quickly.

```
struct entry{  
    char Name[30]; // null-terminated string  
    uint16_t life[2]; // birth year, year died  
    uint16_t year; // year of inauguration  
};  
typedef const struct entry entryType;  
entryType Presidents[3]={  
    {"George Washington", {1732, 1799}, 1789},  
    {"John Adams", {1735, 1826}, 1797},  
    {"Thomas Jefferson", {1743, 1826}, 1801}  
};
```

Program 6.21. A simple data base with three entries.

"George Washington"	
1732	1799
1789	
"John Adams"	
1735	1826
1797	
"Thomas Jefferson"	
1743	1826
1801	

Figure 6.16. A table collects structures of same size into one object.

Many applications require the representation of complex waveforms in digital form. A typical application is calibration curves that describe the input/output behavior of the system. The electronic hardware takes the measurand (y is position, pressure, temperature etc.) as input and has the ADC conversion (x is 0 to 1023) as output. In this situation, the software algorithm is asked to reverse the process, taking as input (x) the ADC measurement and giving as output (y) is the measurand. One of the most efficient, yet simple, techniques for describing nonlinear equations is to provide a small table of (x, y) points then use linear interpolation between the points. In this way, the response is piece-wise linear. This technique only works for any single-valued data set (a unique y for each x). There is a clear tradeoff between accuracy and software efficiency (static and dynamic). E.g., you can add more points to improve accuracy, but it requires more memory and runs slower.

As an example, we will design a fixed-point **sin()** function using table lookup and interpolation. The input is an 8-bit unsigned fixed point with a resolution of  $2\pi/256$ , and the output is an 8-bit signed fixed point with a resolution of  $1/127$ . Rather than have a complete table of all 256 possibilities, we will store a subset of individual points and use interpolation between the points, as shown in Table 6.3 and Figure 6.17. Let  $x$  be the input ( $0 \leq x < 2\pi$ ) and  $\mathbf{Ix}$  be the integer portion (0 to 255).

Similarly,  $y$  is the output ( $-1 \leq y \leq +1$ ) and  $\mathbf{Iy}$  is the integer portion (-127 to 127). First, we find two points in the table  $(x_1, y_1)$  and  $(x_2, y_2)$  such that  $x_2 > x \geq x_1$ . If we assume a straight line between the points, we can calculate  $y$  from  $x$ .

$$y = (y_2 - y_1) * (x - x_1) / (x_2 - x_1) + y_1$$

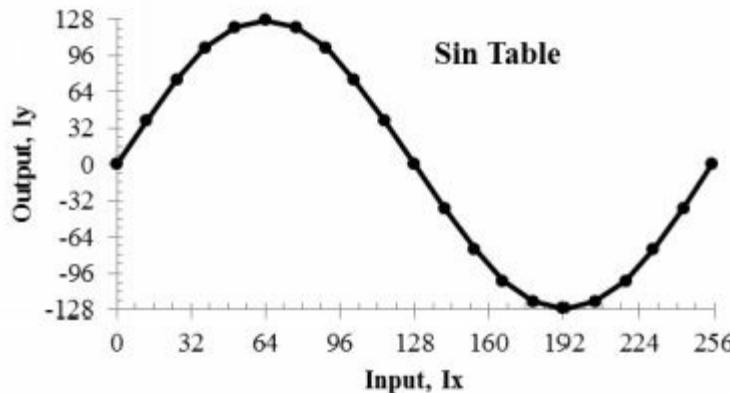


Figure 6.17. A table contains specific points and the software will use linear interpolation to fill in the gaps.

index	$x$	$y=\sin(x)$	$\mathbf{Ix}$	$\mathbf{Iy}$
0	0.000	0.000	0	0
1	0.314	0.309	13	39
2	0.628	0.588	26	75
3	0.942	0.809	38	103
4	1.257	0.951	51	121
5	1.571	1.000	64	127
6	1.885	0.951	77	121
7	2.199	0.809	90	103
8	2.513	0.588	102	75
9	2.827	0.309	115	39
10	3.142	0.000	128	0
11	3.456	-0.309	141	-39
12	3.770	-0.588	154	-75
13	4.084	-0.809	166	-103
14	4.398	-0.951	179	-121
15	4.712	-1.000	192	-127
16	5.027	-0.951	205	-121
17	5.341	-0.809	218	-103
18	5.655	-0.588	230	-75

19	5.969	-0.309	243	-39
20	6.283	0.000	255	0

**Table 6.3. Fixed-point implementation of a sin function.**

```
IxTab DCD 0,13,26,38,51,64,77
      DCD 90,102,115,128,141
      DCD 154,166,179,192,205
      DCD 218,230,243,255,256
IyTab DCD 0,39,75,103,121
      DCD 127,121,103,75,39,0
      DCD -39,-75,-103,-121,-127
      DCD -121,-103,-75,-39,0,0
;*****Sin*****
;Inputs: R0 is 0 to 255, Ix
;Output: R0 is -127 to +127 Iy
Sin PUSH {R4-R6,LR}
      LDR R1,=IxTab ;find x1<=Ix<x2
      LDR R2,=IyTab
lookx1 LDR R6,[R1,#4] ;x2
      CMP R0,R6 ;check Ix<x2
      BLO found ;R1 => x1
      ADD R1,#4
      ADD R2,#4
      B lookx1
found LDR R4,[R1] ;x1
      SUB R4,R0,R4 ;Ix-x1
      LDR R5,[R2,#4] ;y2
      LDR R2,[R2] ;y1
      SUB R5,R2 ;y2-y1
      LDR R1,[R1] ;x1
      SUB R6,R1 ;x2-x1
      MUL R0,R4,R5 ;(y2-y1)*(Ix-x1)
      SDIV R0,R0,R6
      ADD R0,R2 ;Iy
      POP {R4-R6,PC}
```

```
const int32_t IxTab[22]={
 0,13,26,38,51,64,77,90,102,
 115,128,141,154,166,179,192,205,
 218,230,243,255,256};

const int32_t IyTab[22]={
 0,39,75,103,121,127,121,103,
 75,39,0,-39,-75,-103,-121,-127,
 -121,-103,-75,-39,0,0};

// Ix is 0 to 255 (pi/128)
// Iy is -127 to +127 (1/256)
int32_t Sin(int32_t Ix){
  int32_t x1,x2,y1,y2;
  int i=0;
  while(Ix >= IxTab[i+1]){
    i++;
  }
  x1 = IxTab[i];
  x2 = IxTab[i+1];
  y1 = IyTab[i];
  y2 = IyTab[i+1];
  return ((y2-y1)*(Ix-x1))/(x2-x1)+y1;
}
```

**Program 6.22. Linear interpolation (LinearInterpolation\_xxx.zip).**

In software we manipulate the integer portion of variable to calculate **Iy** from **Ix**.

$$Iy = ((y2-y1)*(Ix-x1))/(x2-x1) + y1$$

The two arrays consist of multiple unsigned ( **Ix** , **Iy** ) pairs, which define a piece-wise linear function. The first **Ix** entry must be less than or equal to minimum input, and the last **Ix** entry must be bigger than maximum input. One entry was added at the end of the table to make sure the operation works for **Ix** =255. The table must be monotonic in **Ix**. See Program 6.22.

# 6.9. Functional Debugging

## 6.9.1. Instrumentation: Dump into Array without Filtering

There are three limitations of using print statements to debug. First, many embedded systems do not have a standard output device onto which we could stream debugging information. A second difficulty with print statements is that they can significantly slow down the execution speed in real-time systems. The bandwidth of the print functions often cannot keep pace with the real-time execution. For example, our system may wish to call a function 1000 times a second (or every 1 ms). If we add print statements to it that require more than 1 ms to perform, the presence of the print statements will cause the system to crash. In this situation, the print statements would be considered extremely intrusive. Another problem with print statements occurs when the system is using the same output hardware for its normal operation, as is required to perform the print function. For example, your watch may have an LCD, but that display is used to implement the watch functionality. If we output debugging information to the LCD, the debugger output and normal system output are intertwined.

To solve these limitations, we can add a debugger instrument that dumps strategic information into an array at run time. We can then observe the contents of the array at a later time. One of the advantages of dumping is that the JTAG debugger allows you to visualize memory even when the program is running. So this technique will be quite useful in systems with a JTAG debugger.

Assume **happy** and **sad** are strategic 8-bit variables. The first step when instrumenting a dump is to define a buffer in RAM to save the debugging measurements.

SIZE EQU 20	#define SIZE 20
HappyBuf SPACE SIZE	uint8_t HappyBuf[SIZE];
SadBuf SPACE SIZE	uint8_t SadBuf[SIZE];
Cnt SPACE 4	uint32_t Cnt;

The **Cnt** will be used to index into the buffers. **Cnt** must be initialized to zero, before the debugging begins. The debugging instrument, shown in Program 6.23, dumps the strategic variables into the buffers. When writing debugging instruments it is good style to preserve all registers.

Save PUSH {R0-R4,LR}	
LDR R0,=Cnt ;R0 = &Cnt	
LDR R1,[R0] ;R1 = Cnt	
CMP R1,#SIZE	
BHS done ;full?	
LDR R3,=Happy	
LDRB R3,[R3] ;R3 is happy	
LDR R2,=HappyBuf	
STRB R3,[R2,R1] ;save happy	

void Save(void){	
if(Cnt < SIZE){	
HappyBuf[Cnt] = happy;	
SadBuf[Cnt] = sad;	
Cnt++;	
}	
}	

```

LDR R3,=Sad
LDRB R3,[R3] ;R3 is sad
LDR R2,=SadBuf
STRB R3,[R2,R1] ;save sad
ADD R1,#1
STR R1,[R0] ;save Cnt
done POP {R0-R4,PC}

```

### Program 6.23. Instrumentation dump.

Next, you add **BLSave** statements at strategic places within the system. You can either use the debugger to display the results, or add software that prints the results after the program has run and stopped.

## 6.9.2. Instrumentation: Dump into Array with Filtering.

One problem with dumps is that they can generate a tremendous amount of information. If you suspect a certain situation is causing the error, you can add a **filter** to the instrument. A filter is a software/hardware condition that must be true in order to place data into the array. In this situation, if we suspect the error occurs when another variable gets large, we could add a filter that saves in the array only when the variable is above a certain value. In the example shown in Program 6.24, the instrument dumpsonly when **sad** is greater than 100.

```
Save PUSH {R0-R4,LR}
```

```

LDR R3,=Sad
LDRB R3,[R3] ;R3 is sad
CMP R3,#100
BLS done ;assuming unsigned
LDR R0,=Cnt ;R0 = &Cnt
LDR R1,[R0] ;R1 = Cnt
CMP R1,SIZE
BHS done ;full?
LDR R2,=SadBuf
STRB R3,[R2,R1] ;save sad
LDR R3,=Happy
LDRB R3,[R3] ;R3 is happy
LDR R2,=HappyBuf
STRB R3,[R2,R1] ;save happy
ADD R1,#1
STR R1,[R0] ;save Cnt
done POP {R0-R4,PC}

```

```
void Save(void){
```

```

if(sad > 100){
    if(Cnt < SIZE){
        HappyBuf[Cnt] = happy;
        SadBuf[Cnt] = sad;
        Cnt++;
    }
}
}
```

### Program 6.24. Instrumentation dump with filter.

**Observation:** You should save registers at the beginning and restore them back at the end, so the debugging instrument itself doesn't cause the software to crash.

# 6.10. Exercises

**6.1** Write a subroutine to converts a null-terminated string to upper case. In particular, convert all lower case ASCII characters to upper case. The original data is in RAM, so this routine overwrites the string. A pointer to the string is passed by reference in Register R0.

**6.2** Write a subroutine to converts a null-terminated string to lower case. In particular, convert all upper case ASCII characters to lower case. The original data is in RAM, so this routine overwrites the string. A pointer to the string is passed by reference in Register R0.

**6.3** Write a subroutine that compares two null-terminated strings. A pointer to the first string is passed by reference in Register R0. A pointer to the second string is passed by reference in Register R1. The return parameter is in Register R0; it will be 0 if the strings do not match, and will be nonzero if the strings match.

**6.4** Write a subroutine that adds two equal sized arrays. The size of the arrays is passed in Register R2. A pointer to the first array is passed by reference in Register R0. A pointer to the second array is passed by reference in Register R1. The first array, pointed to by R0, should be added to the second array, pointed to by R1, and the sum placed back in the second array. Assume the data is 8-bit unsigned, and implement a ceiling operation (set result to 255) on overflow.

**6.5** Write a subroutine that implements the dot-product two equal sized arrays. The arrays contain 8-bit unsigned numbers. Register R0 contains the size of the array, and registers R0 and R1 are call by reference pointers to the arrays. The return parameter is an unsigned 32-bit number in Register R0.

For example consider these two arrays

**Vector1 DCB 10,20,30 ; 3-D vector**

**Vector2 DCB 1,0,2 ; 3-D vector**

The dot product is  $10*1+20*0+30*2 = 70$ . The calling sequence is

```
MOV R2,#3      ; size of arrays  
LDR R0,=Vector1 ; pointer to first array  
LDR R1,=Vector2 ; pointer to second array  
BL DotProduct
```

**6.6** Write a subroutine that counts the number of characters in a string. The string is null-terminated. Register R0 is a call by reference pointer to the string. The number of characters in the string is returned in Register R0. For example, consider this string

**Name DCB “Valvano”**

**DCB 0**

The size is 7. The calling sequence is

```
LDR R0,=Name    ; pointer to string  
BL Count
```

**6.7** Write a subroutine that finds the maximum number in an array. The array contains 8-bit signed numbers. The first element of the array is the size. Register R0 is a call by reference pointer to the array. The maximum value in the array is returned in Register R0.

**6.8** Write a subroutine that finds the largest absolute value in an array. The array contains 8-bit signed numbers. The first element of the array is the size. Register R0 is a call by reference pointer to the array. The maximum absolute value in the array is returned in Register R0.

**6.9** Write a subroutine that counts the frequency of occurrence of letters in a text buffer. Register R0 points to a null-terminated ASCII buffer. There is a 26-element array into which the frequency data will be entered. For example, the first element of **Freq** will contain the number of A's and a's. Count only the upper case and lower case letters.

**Freq SPACE 26 ;twenty six 16-bit counters**

**6.10** Assume we have some 6-row by 8-column matrix data structures. The precision of each entry is 16 bits. The information is stored in column-major format (the data for each column is stored contiguously) with zero indexing. I.e., the row index, I, ranges  $0 \leq I \leq 5$ , and the column index, J, ranges  $0 \leq J \leq 7$ . Write the assembly language subroutine that accepts a pointer to the array, the I and J indices and returns the 16-bit contents.

**;Inputs R0 row index I=0,1,...,5**

**; R1 column index J=0,1,...,7**

**; R2 pointer to a 6 by 8 matrix**

**;Outputs R0 16-bit contents at matrix[I,J]**

**6.11** Assume we have some 5-row by 10-column matrix data structures. The precision of each entry is 32 bits. The information is stored in column-major format (the data for each column is stored contiguously) with zero indexing. I.e., the row index, I, ranges  $0 \leq I \leq 4$ , and the column index, J, ranges  $0 \leq J \leq 9$ . Write the assembly language subroutine that accepts a pointer to the array, the I and J indices and returns the 16-bit contents. Don't save/restore registers.

**;Inputs R0 row index I=0,1,...,4**

**; R1 column index J=0,1,...,9**

**; R2 pointer to a 5 by 10 matrix**

**;Outputs R0 32-bit contents at matrix[I,J]**

**D6.12** Write an assembly main program that implements this Mealy finite state machine. The FSM data structure, shown below, is given and cannot be changed. The next state links are defined as 32-bit pointers. Each state has 8 outputs and 8 next-state links. The input is on Port A bits 2, 1, and 0 and the output is on Port B bits 5, 4, 3, 2, 1, and 0. There are three states (S0, S1, and S2), and initial state is S0. Show all assembly software required to execute this machine. The repeating execution sequence is input, output (depends on input and current state), next (depends on input and current state).

**S0 DCB 0,0,5,6,3,9,3,0 ; Outputs for inputs 0 to 7**

**DCD S0,S0,S1,S1,S1,S2,S2,S2 ; Next states for inputs 0-7**

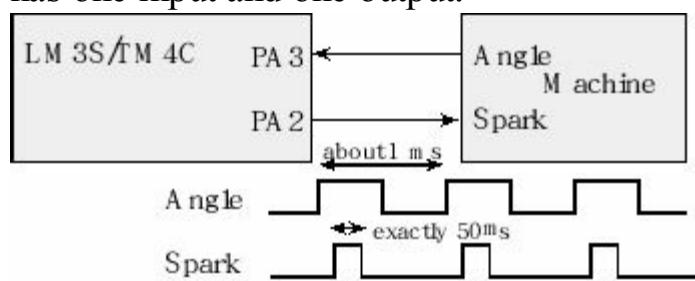
**S1 DCB 1,2,3,9,6,5,3,3 ; Outputs for inputs 0 to 7**

**DCD S2,S0,S0,S0,S2,S2,S2,S1 ; Next states for inputs 0-7**

**S2 DCB 1,2,3,9,6,5,3,3 ; Outputs for inputs 0 to 7**

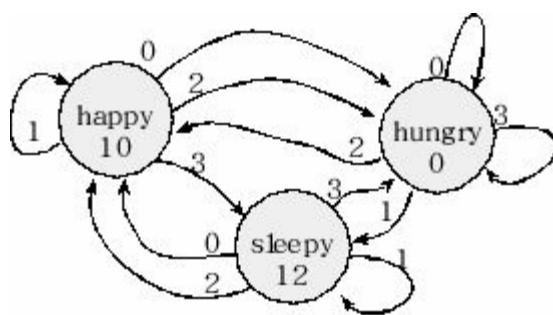
**DCD S2,S2,S2,S2,S0,S0,S2,S1 ; Next states for inputs 0-7**

**D6.13** Design a microcomputer-based controller using a linked list finite state machine. The system has one input and one output.



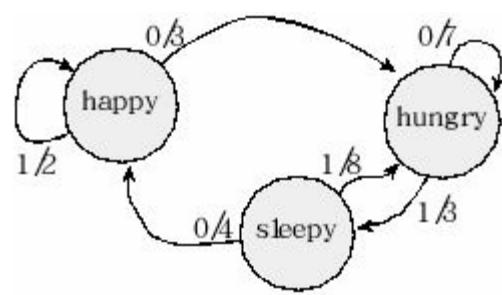
The input, **Angle**, is a periodic signal with a frequency of 20 to 1000 Hz (has a period of 1 to 50 ms). The output, **Spark**, should be a positive pulse (exactly 50  $\mu$ s wide) every time **Angle** goes from 0 to 1. The delay between the rising edge of **Angle** and the start of the **Spark** pulse should be as short as possible. The period of **Angle** can vary from 1 ms to 50 ms. Since **Angle** is an input you cannot control it, only respond to its rising edge.

- Design the one input, one output finite state machine for this system. Draw the FSM graph. Use descriptive state names (i.e., don't call them S0, S1, S2...)
- Show the assembly code to create the statically-allocated linked list. Include statement(s) to place it in the proper location on your microcomputer.
- Show the assembly language controller. Include statement(s) to place it in the proper location on a microcomputer. Assume this is the only task that the microcomputer executes. I.e., show ALL the instructions necessary. Make the program automatically start on a RESET.



**D6.14** Write an assembly main program that implements this Moore finite state machine. The FSM state graph, shown below, is given and cannot be changed. The input is on Port A bits 1 and 0, and the output is on Port B bits 4, 3, 2, 1, and 0. There are three states (happy, hungry, and sleepy), and initial state is happy.

- Show the ROM-based FSM data structure
- Show the initialization and controller software. Initialize the direction registers, making all code friendly. You may add variables in any appropriate manner (registers, stack, or global RAM). The repeating execution sequence is ...output, input, next.... Please make your code friendly.



**D6.15** Write an assembly main program that implements this Mealy finite state machine. The FSM state graph, shown below, is given and cannot be changed. The input is on Port A bit 0 and the output is on Port B bits 3,2,1,0. There are three states (happy, hungry, sleepy), and initial state is happy.

- Show the ROM-based FSM data structure
- Show the initialization and controller software. Initialize the direction registers, making all code friendly. You may add variables in any appropriate manner (registers, stack, or global RAM). The repeating execution sequence is ... input, output, next.... Please make your code friendly.

---

# 6.11. Lab Assignments

**Lab 6.1** Minimally Intrusive Debugging. Take one of the labs from Chapter 4 or 5 and add minimally intrusive debugging instruments. The goal is to collect a set of measurements that prove the lab is operational. Measure the execution time of your dump, and quantify the intrusiveness as the execution time divided by the time between dumps.

**Lab 6.2** Traffic Light Controller. Implement the traffic light controller of Example 6.4. Use push button switches for the sensors and LEDs for the lights. Add a walk switch and walk/don't walk light to the traffic light controller. Show the Moore FSM graph and demonstrate the 1-1 relationship between the graph and the software structure.

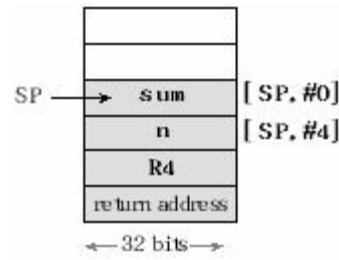
**Lab 6.3** Graphics Driver. Start with the bit matrix routines in Nokia5110\_xxx.zip. Design and implement a line draw function that takes two points and a line color and adds pixels to the **Screen**. Design and implement a circle draw that takes a point, a radius and a line color and adds pixels to the **Screen**. Use these routines to draw a clock face (without letters and numbers).

**Lab 6.4** Cubic Interpolation. Redesign the sin function shown in Program 6.22. The input and output are 16-bit fixed-point numbers with a resolution of  $2^{-12}$ . The particular interpolation mechanism we will use is called cubic interpolation. The idea behind this technique is given in the following document online: <http://paulbourke.net/miscellaneous/interpolation/>

# 7. Variables, Numbers, and Parameter Passing

## Chapter 7 objectives are to:

- Explain how to implement local variables on the stack
- Show how various C compilers implement local variables and pass parameters
- Compare and contrast call by value versus call by reference parameter passing
- Show the interface to a liquid crystal display
- Present fixed-point and floating-point formats
- Develop algorithms to convert one format into another



Variables are an important component of software design, and there are many factors to consider when creating variables. Some of the obvious considerations are the size and format of the data. Another factor is the scope of a variable. The scope of a variable defines which software modules can access the data. Variables with an access that is restricted to one software module are classified as private, and variables shared between multiple modules are public. In general, a system is easier to design (because the modules are smaller and simpler), easier to change (because code can be reused), and easier to verify (because interactions between modules are well-defined) when we limit the scope of our variables. However, since modules are not completely independent we need a mechanism to transfer information from one to another. The ARM Application Binary Interface (ABI) has detailed descriptions of how to develop software interfaces. However, in this chapter, we will discuss the fundamentals of software interfaces. Because their contents are allowed to change, all variables must be allocated in RAM and not ROM. On the one hand, global variables contain information that is permanent and are usually assigned a fixed location in RAM. On the other hand, local variables contain temporary information and are stored in a register or allocated on the stack. One of the important objectives of this chapter is to present design steps for creating, using, and destroying local variables on the stack. In summary, there are three types of variables public globals (shared permanent), private globals (unshared permanent), and private locals (unshared temporary). Because there is no appropriate way to create a public local variable, we usually refer to private local variables simply as local variables, and the fact that they are private is understood.

# 7.1. Local versus global

A **local variable** contains temporary information. Since we will implement local variables on the stack or in registers, this information cannot be shared with other software modules. Therefore, under most situations, we can further classify these variables as private. Local variables are allocated, used, and then deallocated, in this specific order. For speed reasons, we wish to assign local variables to a register. When we assign a local variable to a register, we can do so in a formal manner. There will be a certain line in the assembly software at which the register begins to contain the variable (allocation), followed by lines where the register contains the information (access or usage), and a certain line in the software after which the register no longer contains the information (deallocation). As an example, consider the register allocation used in engine controller Mealy FSM, similar to Program 6.9, shown here as Program 7.1.

Line	Program	R0	R1	R2	R4	R5	R6
	Start						
1	LDR				ClkPt		
2	R1,=SYSCTL_RCGCGPIO_R	Clk		ClkPt			
3	LDR R0,[R1]	Clk		ClkPt			
4	ORR R0,R0,#0x01	Clk		ClkPt			
5	STR R0,[R1]						
6	NOP						
7	NOP			DIRpt			
8	LDR R1,=PORTA_DIR_R	DIR		DIRpt			
9	LDR R0,[R1]	DIR		DIRpt			
10	BIC R0,R0,#0x01	DIR		DIRpt			
11	ORR R0,R0,#0x06	DIR		DIRpt			
12	STR R0,[R1]			AFSELpt			
13	LDR	AFSEL		AFSELpt			
14	R1,=PORTA_AFSEL_R	AFSEL		AFSELpt			
15	LDR R0,[R1]	AFSEL		AFSELpt			
16	BIC			DENpt			
17	R0,R0,#0x07	DEN		DENpt			
18	STR R0,[R1]	DEN		DENpt			
19	LDR R1,=PORTA_DEN_R	DEN		DENpt			
20	LDR R0,[R1]				Pt		
21	ORR R0,R0,#0x07				Pt	INpt	
22	STR R0,[R1]				Pt	INpt	OUTpt
23	LDR R4,=Stop	Input			Pt	INpt	OUTpt
24	LDR R5,=INPUT	Input			Pt	INpt	OUTpt
25	LDR R6,=OUTPUT	Input	OutIndex		Pt	INpt	OUTpt
26	FSM LDR R0,[R5]	Input	OutIndex	Out	Pt	INpt	OUTpt
27	LSL R0,R0,#2	Input		Out	Pt	INpt	OUTpt
28	ADD R1,R0,#OUT	Input		Out	Pt	INpt	OUTpt
29	LDR R2,[R4,R1]	Input	NextIndex		Pt	INpt	OUTpt
30	LSL R2,R2,#1	Input	NextIndex		Pt	INpt	OUTpt
31	STR R2,[R6]				Pt	INpt	OUTpt
	ADD R1,R0,#NEXT						
	LDR R4,[R4,R1]						
	B FSM						

## Program 7.1. Register assignments in a finite state machine controller.

R1 is allocated for holding the **OutIndex** value (0 or 4) in Line 25, used in Line 25, and then deallocated, such that after Line 26, R1 can be used for other purposes. Registers R0, R1, and R2 are used in this program to temporarily hold information, and hence are classified as local variables. Contrast this to how Registers R4, R5, and R6 are used. This is a VERY simple program and in such, the usage of Registers R4, R5, and R6 is unusual. This main program assigns Register R4 to hold the state pointer (**Pt**) in Line 20. From that point in time, Register R4 always contains **Pt**, and hence we classify this assignment of Register R4 as permanent. It is appropriate to assign a register as permanent only in very simple situations (e.g., less than a 50-line program with no interrupts). Registers R5 and R6 also permanently contain pointers to the input and output ports respectively.

**Observation:** Use Registers R0, R1, R2, R3, and R12 for temporary data such as function parameters and use Registers R4–R11 for more permanent data (AAPCS).

The information stored in a local variable is not permanent. This means if we store a value into a local variable during one execution of the module, the next time that module is executed the previous value is not available. Examples include loop counters and temporary sums. We use a local variable to store data that is temporary in nature. We can implement a local variable using the stack or registers. Reasons why we choose local variables over global variables include

- Dynamic allocation/release allows for reuse of RAM
  - Limited scope of access (making it private) provides for data protection
- Only the program that created the local variable can access it
- Since an interrupt will save registers, the code is reentrant
  - Since absolute addressing is not used, the code is relocatable

Reasons why we place local variables on the stack rather than using registers include

- We can use symbolic names for the variables, making it easier to understand
- The number of variables is only limited by the size of the stack
- Because it is more general, it will be easier to add additional variables

A **global variable** is allocated at a permanent and fixed location in RAM. A public global variable contains information that is shared by more than one program module. We must use global variables to pass data between the main program (i.e., foreground thread) and an ISR (i.e., background thread). If a function called from the foreground belongs to the same module as the ISR, then a global variable used to pass data between the function and the ISR can be classified as a private global. Private means software outside the module does not directly access the data. Global variables are allocated at assembly time and never deallocated. Allocation of a global variable means the assembler assigns the variable a fixed location in RAM. The information they store is permanent. Examples include time of day, date, calibration tables, user name, temperature, FIFO queues, and message boards. We use **LDR R0,=Data** to get a pointer to the variable **Data**. When dealing with complex data structures like the ones presented in Chapter 6, we choose to make the pointers to the data either public (bad) or private (good). In general, it is a poor design practice to employ an excessive number of public global variables. On the other hand, private global variables are necessary to store information that is permanent in nature.

**Checkpoint 7.1:** How do you create a local variable in C?

**Checkpoint 7.2:** How do you create a global variable in C?

**Observation:** Sometimes we store temporary information in global variables because it is easier to observe the contents using the debugger. This usage is appropriate during the early stages of development, but once the module is initially tested, temporary information should be converted to local, and the system should be tested again.

In C, a **static** local has permanent allocation, which means it maintains its value from one call to the next. It is still local in scope, meaning it is only accessible from within the function. I.e., modifying a local variable with **static** changes its allocation (it is now permanent), but doesn't change its scope (it is still private). In the following example, **count** contains the number of times **MyFunction** is called. The initialization of a static local occurs just once, during startup.

```
void MyFunction(void){ static uint32_t count=0;  
    count++;  
}
```

In C, we create a private global variable using the **static** modifier. Modifying a global variable with **static** does not change its allocation (it is still permanent), but does reduce its scope. Regular globals can be accessed from any function in the system (public), whereas a **static** global can only be accessed by functions within the same file. A **static** global is private. Functions can be static also, meaning they can be called only from other functions in the file. E.g.,

```
static int32_t myPrivateGlobalVariable; // accessible by this file only  
void static MyPrivateFunction(void){  
}
```

In C, a **const** global is read-only. It is allocated in the ROM portion of memory. Constants, of course, must be initialized at compile time. E.g.,

```
const int16_t Slope=21;  
const uint8_t SinTable[8]={0,50,98,142,180,212,236,250};
```

**Common Error:** If you leave off the **const** modifier in the **SinTable** example, the table will be allocated twice, once in ROM containing the initial values, and once in RAM containing data to be used at run time. Upon startup, the system copies the ROM-version into the RAM-version.

**Maintenance Tip:** It is good practice to specify whether an assembly variable is signed or unsigned in the comments. If the information has units (e.g., volts, seconds etc.) this should be included also.

## 7.2. Stack rules

In the last section we discussed the important issue of global versus local variables. One of the more flexible means to create local variables will be the stack. In this section, we define a set of rules for proper use of the stack. A last in first out (LIFO) stack is implemented in hardware by most computers. The stack can be used for local variables (temporary storage), saving return addresses during subroutine calls, passing parameters to subroutines, and to save registers during the processing of an interrupt. The first advantage of placing local variables on the stack is that the storage can be dynamically allocated before usage and deallocated after usage. The second advantage is the facilitation of reentrant software.

The stack pointer (SP) on the Cortex™-M processor points to the **top** entry of the stack, as shown in Figure 7.1. Also entries on the stack are 32-bits wide. If it exists, we define the data immediately below the top (larger memory address) as **next to top**. To **push** a word on the stack, we first decrement the stack pointer (SP) by 4, then we store the word at the location pointed to by the SP. To **pop** a byte from the stack, first we read the word from memory pointed to by SP, then we increment the SP by 4. Interrupts, the **PUSH** instruction and the **POP** instruction are the three common ways to modify the stack. Furthermore, subtracting multiples of 4 from the SP will allocate stack space, and adding multiples of 4 to the SP will deallocate stack space.

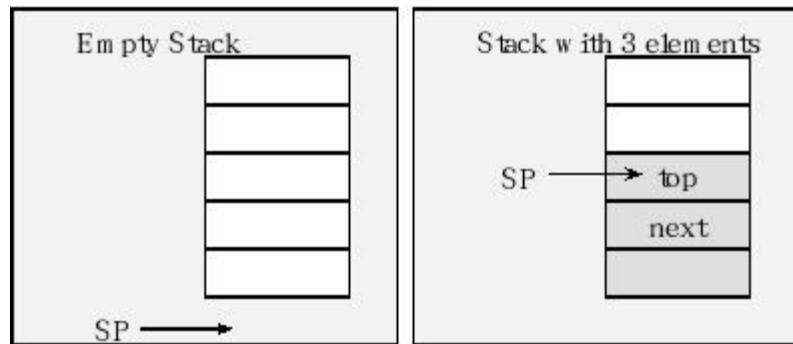


Figure 7.1. Each entry on the stack is 32 bits. The white boxes are free, and the shaded boxes contain data.

We can read and write previously allocated locations on the stack using indexed mode addressing. For example to read the 32-bit value from the next to the top word,

**LDR R0,[SP,#4] ;R0 = the next to the top byte**

The LIFO stack has a few rules (repeated from Chapter 3)

1. Program segments should have an equal number of pushes and pops
2. Stack accesses (push or pop) should not be performed outside the allocated area
3. Stack reads and writes should not be performed within the free area
4. Stack push should first decrement SP by 4, then store the data
5. Stack pop should first read the data, and then increment SP by 4

Programs that violate rule number 1 will probably crash when an illegal address is moved into the PC. Violations of rule number 2 can be caused by a stack underflow or overflow. Stack underflow is caused when there are more pops than pushes, and it is always the result of a software bug. A stack overflow can be caused by two reasons. If the software mistakenly pushes more than it pops, then the stack pointer will eventually overflow its bounds. Even when there is exactly one pop for each push, a stack overflow can occur if the stack is not allocated large enough. Stack overflow is a very difficult bug to recognize, because the first consequence occurs when the computer pushes data onto the stack and overwrites data stored in a global variable. At this point the local variables and global variables exist at overlapping addresses. Setting a breakpoint at the first address of the allocated stack area allows you to detect a stack overflow situation.

**Checkpoint 7.3:** How do you specify the size of the stack (look in startup.s)?

Figures 7.1 and 7.2 show the **free area** as white boxes. The following assembly code violates rule 3 and will not work if interrupts are active. The objective is to save Register R0 onto the stack. When an interrupt occurs registers will automatically be pushed on the stack, destroying the data.

```
SUB R1,SP,#8 ;R1 points to the free area  
STR R0,[R1] ;Store contents of R0 into free area (**illegal***)  
LDR R2,[R1] ;Read contents of free area into R2 (**illegal***)
```

The proper technique is to allocate first, then store.

```
PUSH {R0,R1} ;Store contents of R0,R1 onto the stack
```

## 7.3. Local variables allocated on the stack

The following assembly codeshows the **PUSH** and **POP** instructions can be used to store temporary information on the stack. If a subroutine modifies a register, it is a matter of programmer style as to whether or not it should save and restore the register. According to AAPCS a subroutine canfreely change R0–R3 and R12 but save and restore any other register it changes. In particular, if one subroutine calls another subroutine, then it must save and restore the LR. In the following example, assume the function modifies Register R0, R4, R8 and calls another function. The programming style dictates registers R4, R8, and LR be saved. Notice the return address is pushed on the stack as LR but popped off into PC. When multiple registers are pushed or popped, the data exist in memory with the lowest numbered register using the lowest memory address. In other words, the registers in the {} can be specified in any order, but the order in which they appear on the stack is fixed. According to AAPCS we must push and pop an even number of registers. Of course remember to balance the stack by having the same number of pops as pushes.

**Func PUSH {R4,R5,R8,LR} ; save registers as needed**

**;1) allocate local variables**

**;2) body of the function, access local variables**

**;3) deallocate local variables**

**POP {R4,R5,R8,PC} ; restore registers and return**

The ARM processor has a lot of registers, and we appropriately should use them for temporary information such as function parameters and local variables. However, when there are a lot of parameters or local variables, we can place them on the stack. Program 7.2 has a large **data** buffer that is private to this function. It is inconvenient to store arrays in registers. Rather it is appropriate to place the array in memory and use indexed addressing mode to access the information. Because this buffer is private and temporary we will place it on the stack. 1) The **SUB** instruction allocates 10 words on the stack. Figure 7.2 shows the stack before and after the allocation. 2) During the execution of the function, the SP points to the first location of **data**. The local variable **i** is held in R0. R1 will contain **i \*4** as an offset into the buffer, because each buffer entry is 4 bytes. The addressing mode **[SP,R1]** accesses data on the stack without pushing or popping. 3) The **ADD** instruction deallocates the local variable, balancing the stack.

**Set SUB SP,SP,#40 ;1)allocate 10 words**

```
MOVS R0,#0x00 ;2)i=0
B test ;2)
loop LSL R1,R0,#2 ;2)4*i
STR R0,[SP,R1] ;2)access
ADDS R0,R0,#1 ;2)i++
test CMP R0,#10 ;2)
BLT loop ;2)
```

**// C language implementation**

```
void Set(void){
    uint32_t data[10];
    int i;
    for(i=0; i<10; i++){
        data[i] = i;
    }
}
```

```
ADD SP,SP,#40 ;3)deallocate  
BX LR
```

```
}
```

Program 7.2. Assembly and C versions that initialize a local array of ten elements.

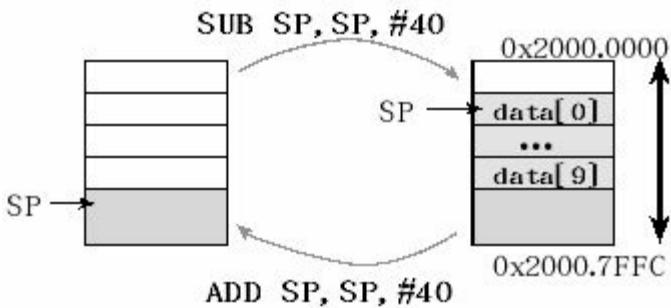


Figure 7.2. A stack picture showing a local array of ten elements.

Stack implementation of local variables has four stages: binding, allocation, access, and deallocation.

1. **Binding** is the assignment of the address (not value) to a symbolic name. The symbolic name will be used by the programmer when referring to the local variable. The assembler binds the symbolic name to a stack index, and the computer calculates the physical location during execution. In the following example, the local variable will be at address SP+0, and the programmer will access the variable using [SP,#sum] addressing:

**sum EQU 0 ;32-bit local variable, stored on the stack**

2. **Allocation** is the generation of memory storage for the local variable. The computer allocates space during execution by decrementing the SP. In this first example, the software allocates the local variable by pushing a register on the stack. According to AAPCS, we must allocate space in multiples of 8 bytes. The contents of the register become the initial value of the variable.

**MOV R0,#0**

**MOV R1,#2**

**PUSH {R0,R1} ;allocate and initialize two 32-bit variables**

In this next example, the software allocates the local variable by decrementing the stack pointer. This local variable is also uninitialized. This method is most general, allowing the allocation of an arbitrary amount of data.

**SUB SP,#8 ;allocate two 32-bit variables**

**Checkpoint 7.4:** Assume Register R0 contains the size in 32-bit words of an array, determined at run-time. Write assembly code to allocate the array on the stack.

3. The **access** to a local variable is a read or write operation that occurs during execution. Because we use SP addressing with offset, we will only use **LDR** and **STR** to access local variables on the stack. In the first code fragment, we will add the contents of R0 to the local variable **sum**.

**LDR R1,[SP,#sum] ; R1=sum**

**ADD R1,R0 ; R1=R0+sum**

**STR R1,[SP,#sum] ; sum=R0+sum**

In the next code fragment, the local variable **sum** is divided by 16.

**LDR R0,[SP,#sum] ; R0=sum**

**LSR R0,R0,#4**

**STR R0,[SP,#sum] ; sum=sum/4**

4. **Deallocation** is the release of memory storage for the location variable. The computer deallocated space during execution by incrementing SP. The software deallocates the local variable by incrementing the stack pointer.

**ADD SP,#4 ;deallocate sum**

**Checkpoint 7.5:** Write a subroutine that allocates then deallocates three 32-bit locals.

## 7.4. Stack frames

Each time a function is called a **stack frame** is created. There are four types of data that may be saved in the stack frame. By convention, if there are more than 4 input parameters, additional parameters above 4 will be pushed on the stack by the calling program. If the function calls another function, the LR (return address) must be pushed on the stack. By convention if the function uses registers R4–R11, it will push them on the stack so their values are preserved. Lastly, the function may allocate local variables on the stack. If the input parameters are passed on the stack they must be pushed first, otherwise there is no particular rule about the stack order of the data stored on the stack frame:

- Parameters
- Return address
- Saved registers
- Local variables

One limitation of SP indexed addressing mode to access local variables is the difficulty of pushing additional data onto the stack during the execution of the function. In particular, if the body of the function pushes additional items on the stack, the symbolic binding becomes incorrect. There are two approaches to this problem. First, we could recompute the binding after each stack push/pop. Second, we could assign a second register to point into the stack.

To employ a **stack frame pointer** we execute the initial steps of the function: saving LR, saving registers, and allocating local variables on the stack. Once these initial steps are complete, we set another register to point into the stack. Because R4–R11 will be saved and restored any of these would be appropriate for the stack frame pointer. E.g.,

**MOV R11,SP**

This stack frame pointer (R11) points to the local variables and parameters of the function. It is important in this implementation that once the stack frame pointer is established (e.g., using the **MOV R11,SP** instruction), that the stack frame register (R11) not be modified. The term **frame** refers to the fact that the pointer value is fixed. If R11 is a fixed pointer to the set of local variables, then a fixed binding (using the **EQU** pseudo op) can be established between Register R11 and the local variables and parameters, even if additional information is pushed on the stack. Because the stack frame pointer should not be modified, every subroutine will save the old stack frame pointer of the function that called the subroutine and restore it before returning. Local variable access uses indexed addressing mode using Register R11.

**Observation:** One advantage of using a stack frame is that you can push and pop within the body of the function, and still be able to access local variables using their symbolic name.

**Observation:** With a processor like the ARM with lots of registers, it is not a disadvantage to dedicate a register as a stack frame pointer, and thus making it unavailable for general use.

In C, we can define a local variable after any open brace { . The compiler will usually allocate local variables in registers, but in this section we will place all local variables on the stack. Programs 7.3 and 7.4 calculate the 32-bit sum of the first 1000 numbers. The purpose of these simple programs is to demonstrate various implementations of local variables. In these programs, the result will be returned by value in Register R0. The Figure 7.3 shows the local variables with SP indexed addressing. This simple program did not need to push R4 and LR, but pushing these two illustrates typical components of a stack frame.

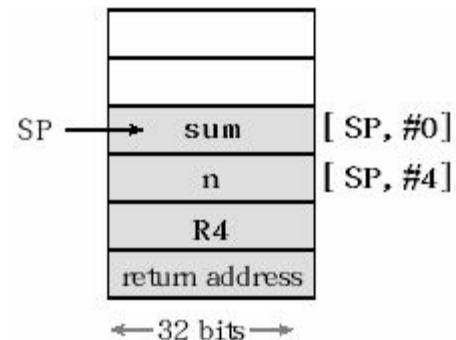


Figure 7.3. The stack frame includes return address, registers, and local variables. The local variables are accessed with SP-indexed addressing mode.

Program 7.3 shows an implementation using regular stack pointer addressing, drawn in Figure 7.3. The binding is not necessary, but its usage greatly improves understanding. For example the access to the variable **n** could be performed using [SP,#4] but [SP,#n] addressing mode is easier to understand. The binding creates exactly the same machine code as without binding, but it is easier to understand because the variables are referred to by symbolic names.

```
; *****binding phase *****
sum EQU 0 ;32-bit unsigned number
n EQU 4 ;32-bit unsigned number
; 1)*****allocation phase *****
calc PUSH {R4,LR}
    SUB SP,#8 ;allocate n,sum
; 2)*****access phase *****
    MOV R0,#0
    STR R0,[SP,sum] ;sum=0
    MOV R1,#1000
    STR R1,[SP,n] ;n=1000
loop LDR R1,[SP,n] ;R1=n
    LDR R0,[SP,sum] ;R0=sum
    ADD R0,R1 ;R0=sum+n
    STR R0,[SP,sum] ;sum=sum+n
    LDR R1,[SP,n] ;R1=n
    SUBS R1,#1 ;n-1
    STR R1,[SP,n] ;n=n-1
    BNE loop
; 3)*****deallocation phase *****
    ADD SP,#8 ;deallocation
    POP {R4,PC} ;R0=sum
```

```
uint32_t calc(void){
    uint32_t sum,n;
    sum = 0;
    for(n=1000;n>0;n--){
        sum=sum+n;
    }
    return sum;
}
```

Program 7.3. Stack pointer implementation of a function with two local 32-bit variables.

Program 7.4 shows an implementation using stack frame pointer addressing, drawn in Figure 7.4. The program establishes the frame pointer in R11, and then it allocates the variables. In Program 7.4, the variable **n** is accessed using the [R11,#4] addressing mode. Notice the similarity between Programs 7.3 and 7.4. However, the body of Program 7.4 is free to push additional data on the stack.

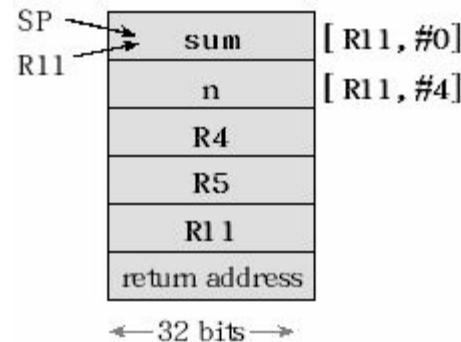


Figure 7.4. The stack frame includes return address, registers, and local variables. The local variables are accessed with R11-indexed addressing modes.

**Common Error:** One does not allocate/deallocate stack space by changing R11. We must modify SP to allocate/deallocate space.

```
; *****binding phase*****
sum EQU 0 ;32-bit unsigned number
n EQU 4 ;32-bit unsigned number
; 1)*****allocation phase *****
calc PUSH {R4,R5,R11,LR}
    SUB SP,#8 ;allocate n,sum
    MOV R11,SP ;frame pointer
; 2)*****access phase *****
    MOV R0,#0
    STR R0,[R11,#sum] ;sum=0
    MOV R1,#1000
    STR R1,[R11,#n] ;n=1000
loop LDR R1,[R11,#n] ;R1=n
    LDR R0,[R11,#sum] ;R0=sum
    ADD R0,R1 ;R0=sum+n
    STR R0,[R11,sum] ;sum=sum+n
    LDR R1,[R11,#n] ;R1=n
    SUB R1,#1 ;n-1
    STR R1,[R11,#n] ;n=n-1
    BNE loop
; 3)*****deallocation phase *****
    ADD SP,#8 ;deallocation
    POP {R4,R5,R11,PC} ;R0=sum
```

```
uint32_t calc(void){
    uint32_t sum,n;
    sum = 0;
    for(n=1000;n>0;n--){
        sum=sum+n;
    }
    return sum;
}
```

Program 7.4. Stack frame pointer implementation of a function with two local 32-bit variables.

# 7.5. Parameter Passing

Up to this point in the book, we used registers to pass data into and out of subroutines. The **input parameters** (or arguments) are pieces of data passed from the calling routine into the subroutine during execution. The **output parameter** (or argument) is information returned from the subroutine back to the calling routine after the subroutine has completed its task. As previously defined in Chapter 6, there are two methods to pass parameters: **call by reference** and **call by value**. With call by reference a pointer to the object is passed. In this way the subroutine and the module that calls the subroutine have access to the exact same object. Call by reference can be used to pass a large quantity of data, and it can be used to implement a parameter that is both an input and an output parameter. With call by value, a copy of the data itself is passed.

- |   |  |
|---|--|
| <ul style="list-style-type: none"><li>• Call by value</li><li>• Call by reference</li></ul> | <p>Safe, simple, good for small amounts of data<br/>Parameter can be input or output, good for large amounts</p> |
|---|--|

Using the stack to pass parameters provides a much greater flexibility not possible with just the registers. Passing parameters via global variables is extremely poor style.

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>• Registers</li><li>• Stack</li><li>• Global variables</li></ul> | <p>Fast and simple<br/>Flexible, good for large amounts of data<br/>Simple and poor style</p> |
|--|---|

## 7.5.1. Parameter Passing in C

The **call by value** method passes a copy of the current value. By AAPCS convention, the first four parameters are passed in Registers R0–R3. In this scheme, variables passed to a function cannot be affected by the function. With call by value parameter passing, there are two copies of **angle** (the original and the copy in R0). See Program 7.5. Notice at the time of the call to **next**, R0 equals the value of **angle**. In this case changes to the variable **angle** occur in the program that owns or defined **angle**.

<pre>; R0 is the angle next ADD R0,#1 ;add to copy       CMP R0,#200       BNE skip       MOV R0,#0 ;roll over skip BX LR angle EQU 0 ;0 to 199 main SUB SP,#4 ;allocate       MOV R0,#0       STR R0,[SP,#angle]       BL Stepper_Init loop BL Stepper_Step       LDR R0,[SP,#angle] ;R0=angle       BL next</pre>	<pre>uint32_t next(uint32_t ang){     ang++;     if(ang == 200){         ang = 0;     }     return ang; } void main(void){     uint32_t angle=0; // 0 to 199     Stepper_Init();     while(1){         Stepper_Step();         angle = next(angle);     } }</pre>
---	---

```

STR R0,[SP,#angle] ;update
B loop

```

## Program 7.5. An input/output parameter is implemented using call by value.

The **call by reference** method passes a pointer to the object. In other words, references (pointers) to the actual arguments are passed, instead of copies of the actual arguments themselves. In this scheme, assignment statements have implied side effects on the actual arguments; that is, variables passed to a function are affected by changes to the formal arguments. Sometimes side effects are beneficial, and sometimes they are not. As an example consider a stepper motor program shown in Program 7.6.

```

; R0 points to the angle
next LDR R1,[R0] ;*pt
    ADD R1,#1 ;++
    CMP R1,#200
    BNE skip
    MOV R1,#0 ;roll over
skip STR R1,[R0] ;update
    BX LR
angle EQU 0 ;0 to 199
main SUB SP,#4 ;allocate
    MOV R0,#0
    STR R0,[SP,#angle]
    BL Stepper_Init
loop BL Stepper_Step
    MOV R0,SP ;R0=&angle
    BL next
    B loop

```

```

void next(uint32_t *pt){
    (*pt) = (*pt)+1;
    if((*pt) == 200){
        (*pt) = 0;
    }
}
void main(void){
    uint32_t angle=0; // 0 to 199
    Stepper_Init();
    while(1){
        Stepper_Step();
        next(&angle);
    }
}

```

## Program 7.6. An input/output parameter is implemented using call by reference.

In Program 7.6, the parameter is passed in R0. Both assembly and C versions are shown. With call by reference parameter passing, there is one copy of the information, and the calling program (e.g., main) passes an address (R0 in the assembly version) to the function. The read and write accesses to the parameter affect the original variable. The local variable **angle** is put on the stack because we need a reference (pointer) to it. Notice at the time of the call to **next**, R0 points to **angle**.

Since C supports only one formal output parameter, we can implement additional output parameters using call by reference. The calling program passes pointers to empty objects (R0 and R1 in the assembly version), and the **where** function fills the objects with data. Program 7.7 shows a function that returns two parameters using call by reference. Assume global variables **Xx** **Yy** are private to the **where** function and contain the true current position.

```

Xx SPACE 4 ; private to where
Yy SPACE 4
where LDR R2,=Xx
    LDR R2,[R2] ;value of Xx
    STR R2,[R0] ;pass data
    LDR R3,=Yy
    LDR R3,[R3] ;value of Yy
    STR R3,[R1] ;pass data
    BX LR
myX EQU 0 ;32-bit
myY EQU 4
func PUSH {R4,LR}

```

```

static int32_t Xx,Yy; // position

void where(int32_t *xpt,
           int32_t *ypt){
    (*xpt) = Xx; // return Xx
    (*ypt) = Yy; // return Yy
}

void func(void){

```

```

SUB SP,#8 ;allocate
MOV R0,SP ;R0=&myX
ADD R1,SP,#myY ;R1=&myY
BL where
;do something based on myX,myY
ADD SP,#8 ;deallocate
POP {R4,PC}

```

```

int32_t myX,myY;
where(&myX,&myY);
// do something based on myX,myY
}

```

## Program 7.7. Multiple output parameters is implemented using call by reference.

An important point to remember about passing arguments by value in C is that there is no connection between an actual argument and its source. Changes to the arguments made within a function have no affect what so ever on the objects that might have supplied their values. They can be changed and the original values will not be affected. This removes a burden of concern from the programmer since he may use arguments as local variables without side effects. It also avoids the need to define temporary variables just to prevent side effects.

It is precisely because C uses call by value that we can pass expressions, not just variables, as arguments. The value of an expression can be copied, but it cannot be referenced since it has no existence in memory. Therefore, call by value adds important generality to the language. Since expressions may include assignment, increment, and decrement operators, it is possible for argument expressions to affect the values of arguments lying to their right. Consider, for example,

```
func(y=x+1, 2*y);
```

where the first argument has the value **x+1** and the second argument has the value **2\*(x+1)**. However, the value of the second argument depends on whether the arguments are evaluated right-to-left or left-to-right. This kind of situation should be avoided, since the C language does not guarantee the order of argument evaluation. The safe way to write this is

```
y=x+1;
func(y, 2*y);
```

The value of the expression is calculated at the time of the call, and that value is passed into the subroutine. This function call passes the ASCII character represented by the **digit** (0-9)

```
OutChar(digit + '0');
```

**Checkpoint 7.6:** What is the difference between call by value and call by reference?

## 7.5.2. Parameter Passing in Assembly Language

In contrast to C, it is easy to return multiple parameters in assembly language. If just a few parameters need to be returned we can use the registers. In Program 7.8, the values of ports A, B, C, and D are to be returned. For this microcontroller the 8-bit port data is returned as 32-bit.

```
; Reg R0 = Port A, Reg R1 = Port B
; Reg R2 = Port C, Reg R3 = Port D
```

## GetPorts LDR R0,=GPIO\_PORTA\_DATA\_R

```
LDR R0,[R0]      ; value of Port A  
LDR R1,=GPIO_PORTB_DATA_R  
LDR R1,[R1]      ; value of Port B  
LDR R2,=GPIO_PORTC_DATA_R  
LDR R2,[R2]      ; value of Port C  
LDR R3,=GPIO_PORTD_DATA_R  
LDR R3,[R3]      ; value of Port D  
BX LR
```

;\*\*\*\*\*calling sequence\*\*\*\*\*

BL GetPorts

; Reg R0,R1,R2,R3 have four results

Program 7.8. Multiple return parameters implemented with registers (not AAPCS compatible).

If many parameters are needed, then the stack can be used. In the next four programs, we compare and contrast four ways to implement a simple subroutine with two input parameters and one output parameter. The function will add the two 32-bit inputs and return the 32-bit sum as the result. Since these subroutines will not be callable from C, we will not follow parameter passing conventions compatible with C. In each case, the calling program will pass in the values of variables **A** and **B**, and then store the result in variable **C**.

Registers. The most efficient method of parameter passes is to use registers. R0 and R1 are input parameters and R2 is the return parameter. On the left of Program 7.9 is the subroutine and on the right is the calling sequence that produces **C=A-B**.

<b>;Inputs:</b> R0,R1 <b>;Outputs:</b> R2=R0-R1 Sub1 SUB R2,R0,R1 BX LR	<b>LDR R0,=A</b> <b>LDR R0,[R0]</b> ;R0 has the value of A <b>LDR R1,=B</b> <b>LDR R1,[R1]</b> ;R1 has the value of B <b>BL Sub1</b> <b>LDR R0,=C</b> <b>STR R2,[R0]</b> ;C=A-B
--	---

Program 7.9. Using registers to pass two inputs and return one output call by value (not AAPCS).

Stack. The stack can be used with passing many input and/or output parameters. When passing an input parameter on the stack, its value is pushed. When returning an output parameter on the stack, the calling program allocates an empty stack position, the subroutine fills the space with data, and then the calling program pops the data from the stack. It is important to draw a stack picture created during the calling sequence so symbolic binding can occur. Figure 7.5 shows the stack during the execution of the body of the subroutine. On the left of Program 7.10 is the subroutine and on the right is the calling sequence that produces **C=A-B**. There are many possible orders of the parameters on the stack. Any order is fine as long as the calling program and the subroutine are consistent.

<b>;Inputs:</b> In1 In2 on stack <b>;Outputs:</b> Out=In1-In2 on stack	<b>LDR R1,=A</b> <b>LDR R1,[R0]</b> ;R0 has the value of A
---	---

In1 EQU 8	LDR R0,=B
In2 EQU 4	LDR R0,[R1] ;R1 has the value of B
Out EQU 0	PUSH {R0,R1} ;input parameters
Sub2 LDR R0,[SP,#In1]	SUB SP,#4 ;place for output
LDR R1,[SP,#In2]	BL Sub2
ADD R2,R1,R0	POP {R2} ;result
STR R2,[SP,#Out]	LDR R0,=C
BX LR	STR R2,[R0] ;C=A-B
	ADD SP,#8 ;balance stack

Program 7.10. Using the stack to pass two inputs and return one output call by value (not AAPCS).

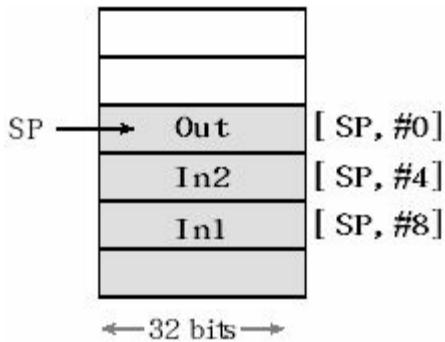


Figure 7.5. The stack includes three parameters, call by value.

Stack frame. Again the stack is used to pass parameters. With this example, we will save the return address, preserve R11, and allocate a local variable. We would need to save the LR if this function called another function (which it doesn't in this case). We would need to save the registers if the convention were to require it. In this case, we did not need a local variable, but we allocated one anyway, so you could see all 4 parts of a stack frame. The calling sequence is identical to the previous example. Figure 7.6 shows the stack during the execution of the body of the subroutine. On the left of Program 7.11 is the subroutine, and on the right is the calling sequence that produces  $C=A-B$ .

;Inputs: In1 In2 on stack	LDR R1,=A
;Outputs: Out=In1-In2 on stack	LDR R1,[R0] ;R0 has the value of A
In1 EQU 20	LDR R0,=B
In2 EQU 16	LDR R0,[R1] ;R1 has the value of B
Out EQU 12	PUSH {R0,R1} ;input parameters
local EQU 0	SUB SP,#4 ;place for output
Sub3 PUSH {R11,LR}	BL Sub3
SUB SP,#4 ;allocate	POP {R2} ;result
MOV R11,SP ;frame pointer	LDR R0,=C
LDR R0,[R11,#In1]	STR R2,[R0] ;C=A+B
LDR R1,[R11,#In2]	ADD SP,#8 ;balance stack
SUB R2,R0,R1	
STR R2,[R11,#Out]	
ADD SP,#4 ;deallocate	
POP {R11,PC}	

Program 7.11. Using the stack frame to pass two inputs and return one output call by value (not AAPCS).

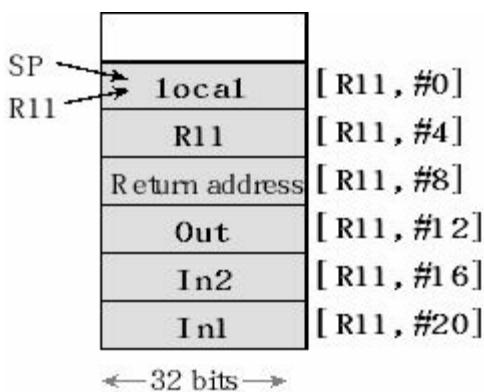


Figure 7.6. A stack frame contains up to four components: parameters, return address, saved registers, and locals.

Global variables. A simple but completely inappropriate method is to pass parameters using global variables, Program 7.12. In this method the information is contained in global memory variables. Many embedded systems use this approach if the processor has very few registers and has limited or no facilities with handling data on the stack.

<pre>;Inputs: A,B ;Outputs: C=A+B Sub4 LDR R0,A LDR R0,[R0] ;R0 = value of A LDR R1,B LDR R1,[R1] ;R1 = value of B ADD R2,R1,R0 ;A+B LDR R0,C STR R2,[R0] ;C=A+B BX LR</pre>	<b>BL Sub4</b>
--	----------------

Program 7.12. Using global variables to pass two inputs and return one output (not AAPCS).

It is good style to use registers or the stack to pass parameters into and out of subroutines. When interrupts are enabled, it is possible to have multiple threads active at the same time. There is still only one processor, so exactly one thread is actually running at a time, but we define concurrent programming as the state where multiple threads are “ready to run” at the same time. The interrupt hardware provides the mechanism to switch from one thread to the next. Invoking the interrupt will push registers on the stack and the return from interrupt will pull those registers from the stack, restoring the registers back to their previous values. Because concurrent threads have “separate” registers and stack areas, software that uses the stack will operate properly in a concurrent environment. Conversely, extreme care is required when using global variables (including the I/O ports) in a concurrent environment. On the other hand, since each thread has “separate” registers and stack, when passing data from one thread to another we must use variables allocated permanently in RAM. Program 7.13 shows an interrupt service routine on the right that sets a global variable called **Flag**, and a subroutine on the left, which is called by the main program, which waits for the **Flag** to be set.

<pre>;Wait for Flag to become 1 Wait LDR R0,=Flag loop LDR R1,[R0] ;R1 = Flag     CMP R1,#1     BNE loop ;wait until 1</pre>	<b>SysTick_Handler</b> <pre>LDR R0,=Flag ;R0 = &amp;Flag MOV R1,#1 STR R1,[R0] ;Flag=1 BX LR ;return from interrupt</pre>
--	---

```
MOV R1,#0  
STR R1,[R0] ;Flag=0  
BX LR
```

Program 7.13. RAM-based variables must be used to pass information from an interrupt service routine to a main program. If Flag is only accessed by these two routines, we can classify it as a private global.

### 7.5.3. C Compiler Implementation of Local and Global Variables

One of the most important applications of learning assembly language involves analyzing assembly listings when programming in a high-level language. When one programs in a high-level language there are many design decisions to be made, affecting accuracy (e.g., overflow, drop-out), reliability (e.g., buffer overflow, critical section, race condition), speed, and code size. Often, these decisions can be best understood at the assembly language level. In fact, one cannot tell if a section of high-level language code is critical without looking at the associated assembly language generated by the compiler. Critical sections will be discussed in Chapter 10. For another example, assume you are designing a finite state machine in C. You could implement the FSM using a linked data structure (next states are pointers) or with a table (next states are indices). Because of the complexity of modern processors, it is almost impossible to determine how fast code will run by just observing the listings generated by the compiler. It is much more accurate to run actual code with actual inputs adding debugging instruments and measuring execution speed with an oscilloscope or logic analyzer. On the other hand, sometimes we have a high-level language program that we know doesn't work, but we just can't seem to find the bug. Often it is easier to visualize bugs by looking at the assembly listing in and around the bugged code. Another application of observing assembly listing generated by the compiler involves proving program correctness. For example, we might ask if the following C code causes an overflow error, assuming both **In** and **Out** are 8-bit **uint8\_t**.

**Out = (99\*In)/100;**

There are two ways to determine if overflow could occur. First, we could exhaustively test the software giving all possible inputs and verifying the correct output for each test case. Second, knowing the architecture and assembly language of the machine, we could look at the general rules about how C handles promotion/demotion and prove that overflow cannot occur. In general the compiler will load variable contents into a register by promoting to the natural size of the processor (in this case 32 bits), perform the calculations at this natural size, and then demote the result back into the size of the target variable. The following assembly code was generated by the ARM Keil™ uVision® compiler (optimization level 3). Notice that the input is promoted to 32 bits, where the multiply and divide occur with 32-bit precision. **In** has a range of 0 to 255, so  $99 * \text{In}$  has a range of 0 to 25245 (fits in 32 bits). As you can see, the compiler will attempt to optimize; in this case substituting one multiplication by two additions. The result is demoted back to 8 bits. However, because **Out** will always be less than **In**, the demotion cannot overflow. Looking at Section 3.3 of the Cortex™-M Technical Reference Manual, we can estimate the execution time for this code. Loads and stores take 2 cycles, moves and adds take 1 cycle, and the divide takes from 2 to 12 cycles. Therefore, this code takes 11 to 21 cycles to execute.

```

0x00000356 4934 LDR r1,[pc,#208] ; R1 = &In
0x00000358 2264 MOVS r2,#0x64 ; R2 = 100
0x0000035A 7848 LDRB r0,[r1,#0x01] ; R0 = In
0x0000035C EB001040 ADD r0,r0,r0,LSL #5 ; R0 = R0+32*R0 = 33*In
0x00000360 EB000040 ADD r0,r0,r0,LSL #1 ; R0 = R0+2*R0 = 99*In
0x00000364 FBB0F0F2 UDIV r0,r0,r2 ; 99*In/100
0x00000368 7088 STRB r0,[r1,#0x02] ; Out=99*In/100

```

Here is another example where observing the assembly code illuminates the bug. The goal of the software is to combine two 8-bit variables into one 16-bit variable. If you cannot see the bug, look up the precedence of the two operators `<<` and `+`.

<b>combine</b> <code>MOV r2,r0 ;R0=msb ADD r3,r1,#0x08 ;lsb+8 LSL r0,r2,r3 ;msb&lt;&lt;(8+lsb) BX lr</code>	<code>uint32_t combine(     uint8_t msb,     uint8_t lsb){     return msb&lt;&lt;8 + lsb; }</code>
--	--

**Observation:** If you want to understand what your C programs are doing, set the optimization level to low (no optimization). If you want your C programs to run fast, set the optimization level to high.

**Common Error:** It would be a grievous programming error to access the local variables of the main program. Therefore, in assembly language, it is essential to make the distinction between local variables and data passed on the stack to the subroutine.

The specific goal of this section is to study how compilers implement local variables and pass parameters. However, in the big picture, we can improve our understanding of both the machine architecture and our high-level language programs by looking at the assembly code generated by the compiler. Program 7.14 shows assembly code was generated by the ARM Keil™ uVision® compiler (optimization level 0).

;R0 is \*pt ;R1 is index  
;R2 is value

int32\_t G; // global  
int32\_t sub(int32\_t \*pt, // R0

```

sub MOV r3,r0 ;R3 is *pt
    LDR r0,[r3,r1,LSL #2]
    SUBS r0,r0,r2
    STR r0,[r3,r1,LSL #2]
    MOV r0,r2 ;return value
    BX lr
main PUSH {r4,lr}
    SUB sp,sp,#0x50 ;allocate z
    MOVS r0,#0x05
    LDR r1,[pc,#340] ;R1 = &G
    STR r0,[r1,#0x00] ;G=5
    MOVS r0,#0x06
    STR r0,[sp,#0x00] ;z[0]=6
    MOVS r2,#0x02 ;value
    MOVS r1,#0x01 ;index
    MOV r0,sp ;*pt
    BL.W sub
    LDR r1,[pc,#320] ;R1 = &G
    STR r0,[r1,#0x00] ;store G
    ADD sp,sp,#0x50 ;deallocate
    POP {r4,pc}

```

```

        int32_t index, // R1
        int32_t value){ // R2
    pt[index] -= value;
    return value;
}

void main(void){
int32_t z[20]; // local
    G = 5;      // access global
    z[0] = 6;   // access local
    G = sub(z,1,2);
}

```

Program 7.14. An example used to illustrate the C compiler's access to globals, locals and parameters.

## 7.6. Fixed-point Numbers

We will use fixed-point numbers when we wish to express values in our software that have noninteger values. In order to design a fixed-point system the range of values must be known. A **fixed-point number** contains two parts. The first part is a **variable integer**, called  $I$ . This variable integer may be signed or unsigned. An unsigned fixed-point number is one that has an unsigned variable integer. A signed fixed-point number is one that has a signed variable integer. The **precision** of a number system is the total number of distinguishable values that can be represented. The precision of a fixed-point number is determined by the number of bits used to store the variable integer. Therefore, to use a fixed-point system, the precision must be less than or equal to 32 bits or  $2^{32}$  alternatives. On the Cortex™-M processor, we typically use 32 bits, but 8 or 16 bits could be used. The variable integer is saved in memory and is manipulated by software. These manipulations include but are not limited to load, store, shift, add, subtract, multiply, and divide. The second part of a fixed-point number is a **fixed constant**, called  $\Delta$ . The fixed constant is defined at design time and cannot be changed at run time. The fixed constant defines the resolution of the number system. The fixed constant is not stored in memory. Usually we specify the value of this fixed constant using software comments to explain our fixed-point algorithm. The value of the fixed-point number is defined as the product of the variable integer times the fixed constant:

$$\text{Fixed-point number} = I \cdot \Delta$$

The **resolution** of a number is the smallest difference that can be represented. In the case of fixed-point numbers, the resolution is equal to the fixed constant,  $\Delta$ . Sometimes we express the resolution of the number as its units. For example, a decimal fixed-point number with a resolution of 0.001 volts is really the same thing as an integer with units of mV. When inputting numbers from a keyboard or outputting numbers to a display, it is usually convenient to use **decimal fixed point**. With decimal fixed point the fixed constant is a power of 10.

$$\text{Decimal fixed-point number} = I \cdot 10^m \text{ for some constant integer } m$$

Again, the integer  $m$  is fixed and is not stored in memory. Decimal fixed point will be easy to input or output to humans, while **binary fixed point** will be easier to use when performing mathematical calculations. With binary fixed point the fixed constant is a power of 2.

$$\text{Binary fixed-point number} = I \cdot 2^n \text{ for some constant integer } n$$

**Observation:** If the range of numbers is known and small, then the numbers can be represented in a fixed-point format.

**Checkpoint 7.7:** Give an approximation of  $\pi$  using the decimal fixed-point ( $\Delta = 0.001$ ) format.

**Checkpoint 7.8:** Give an approximation of  $\pi$  using the binary fixed-point ( $\Delta = 2^{-8}$ ) format.

In the first example, we will develop the equations that a microcontroller would need to implement a digital voltmeter. The LM3S/TM4C family of microcontrollers has a built-in analog to digital converter (ADC) that can be used to transform an analog signal into digital form. The 12-bit ADC analog input range is 0 to +3 V, and the ADC digital output varies 0 to 4095 respectively. Let  $V_{in}$  be the analog voltage in volts and  $n$  be the digital ADC output, then the equation that relates the analog to digital conversion is

$$V_{in} = 3*n/4095 = 0.0007326 *n$$

Resolution is defined as the smallest change in voltage that the ADC can detect. This ADC has a resolution of about 0.7 mV. In other words, the analog voltage must increase or decrease by 0.7 mV for the digital output of the ADC to change by at least one bit. It would be inappropriate to save the voltage as an integer, because the only integers in this range are 0, 1, 2, and 3. Even though the TM4C supports floating point, the voltage data will be saved in fixed-point format, because it will take less memory and execute faster. Decimal fixed point is chosen because the voltage data for this voltmeter will be displayed. A fixed-point resolution of  $\Delta=0.001$  V is chosen because it is about equal to the ADC resolution. Table 7.1 shows the performance of the system. The table shows us that we need to store the variable part of the fixed-point number in at least 16 bits.

$V_{in}$ (V) Analog input	$n$ ADC digital output	$I$ (0.001 V) variable part of the fixed- point data
0.000	0	0
0.001	1	1
1.000	1365	1000
1.500	2048	1500
3.000	4095	3000

**Table 7.1. Performance data of a microcomputer-based voltmeter.**

One possible software formula to convert  $n$  into  $I$  is as follows.

$$I = (3000*n+2048)/4095, \text{ where } I \text{ is defined as } V_{in} = I*0.001V$$

It is very important to carefully consider the order of operations when performing multiple integer calculations. There are two mistakes that can happen. The first error is **overflow**, and it is easy to detect. Overflow occurs when the result of a calculation exceeds the range of the number system. The two solutions of the overflow problem were discussed earlier, promotion and ceiling/floor. The other error is called **drop-out**. Drop-out occurs after a right shift or a divide, and the consequence is that an intermediate result loses its ability to represent all of the values. To avoid drop-out, it is very important to divide last when performing multiple integer calculations. If you divided first, e.g.,  $I=3000*(n/4095)$ , then the values of I would be only 0, or 3000. The addition of “2048” has the effect of rounding to the closest integer. The value 2048 is selected because it is about one half of the denominator. For example, the calculation  $(3000*n)/4095=0$  for  $n=1$ , whereas the  $“(3000*n+2048)/4096”$  calculation yields the better answer of 1. A display algorithm for this decimal fixed-point format is shown the next section.

When adding or subtracting two fixed-point numbers with the same  $\Delta$ , we simply add or subtract their integer parts. First, let  $x$ ,  $y$ , and  $z$  be three fixed-point numbers with the same  $\Delta$ . Let  $x=I\cdot\Delta$ ,  $y=J\cdot\Delta$ , and  $z=K\cdot\Delta$ . To perform  $z = x+y$ , we simply calculate  $K = I+J$ . Similarly, to subtract  $z = x-y$ , we simply calculate  $K=I-J$ . When adding or subtracting fixed-point numbers with different fixed parts, we must first convert the two inputs to the format of the result before adding or subtracting. This is where binary fixed point is more convenient, because the conversion process involves shifting rather than multiplication/division.

In this next example, let  $x$ ,  $y$ , and  $z$  be three binary fixed-point numbers with different resolutions. In particular, we define  $x$  to be  $\square I\cdot2^{-5}$ ,  $y$  to be  $\square J\cdot2^{-2}$ , and  $z$  to be  $\square K\cdot2^{-3}$ . To convert  $x$  to the format of  $z$ , we divide  $I$  by 4 (right shift twice). To convert  $y$  to the format of  $z$ , we multiply  $J$  by 2 (left shift once). To perform  $z = x+y$ , we calculate

$$K = (I>>2)+(J<<1)$$

For the general case, we define  $x$  to be  $\square I\cdot2^n$ ,  $y$  to be  $\square J\cdot2^m$ , and  $z$  to be  $\square K\cdot2^p$ . To perform any general operation, we derive the fixed-point calculation by starting with desired result. For addition, we have  $z = x+y$ . Next, we substitute the definitions of each fixed-point parameter

$$K \cdot 2^p = I \cdot 2^n + J \cdot 2^m$$

Lastly, we solve for the integer part of the result

$$K = I \cdot 2^{n-p} + J \cdot 2^{m-p}$$

For multiplication, we have  $z=x\cdot y$ . Again, we substitute the definitions of each fixed-point parameter

$$K \cdot 2^p = I \cdot 2^n \cdot J \cdot 2^m$$

Lastly, we solve for the integer part of the result

$$K = I \cdot J \cdot 2^{n+m-p}$$

For division, we have  $z=x/y$ . Again, we substitute the definitions of each fixed-point parameter

$$K \cdot 2^p = I \cdot 2^n / J \cdot 2^m$$

Lastly, we solve for the integer part of the result

$$K = I/J \cdot 2^{n-m-p}$$

Again, it is very important to carefully consider the order of operations when performing multiple integer calculations. We must worry about overflow and drop out. In particular, in the division example, if  $(n-m-p)$  is positive then the left shift ( $I \cdot 2^{n-m-p}$ ) should be performed before the divide ( $/J$ ). We can use these fixed-point algorithms to perform complex operations using the integer functions on our microcontroller.

---

**Example 7.1.** Rewrite the following digital filter using fixed-point calculations.

$$y = x - 0.0532672 \cdot x_1 + x_2 + 0.0506038 \cdot y_1 - 0.9025 \cdot y_2$$

**Solution:** In this case, the variables  $y$ ,  $y_1$ ,  $y_2$ ,  $x$ ,  $x_1$ , and  $x_2$  are all integers, but the constants will be expressed in binary fixed-point format. The value  $-0.0532672$  can be approximated by  $-14 \cdot 2^{-8}$ . The value  $0.0506038$  can be approximated by  $13 \cdot 2^{-8}$ . Lastly, the value  $-0.9025$  can be approximated by  $-231 \cdot 2^{-8}$ . The fixed-point implementation of this digital filter is

---

$$y = x + x_2 + (-14 \cdot x_1 + 13 \cdot y_1 - 231 \cdot y_2) \gg 8$$

**Common Error:** Lazy or incompetent programmers use floating point in many situations where fixed-point would be preferable.

**Observation:** As the fixed constant is made smaller, the resolution of the fixed-point representation is improved, but the variable integer part also increases. Unfortunately, larger integers will require more bits for storage and calculations.

**Checkpoint 7.9:** Using a fixed constant of  $2^{-8}$ , rewrite the digital equation  $F = 1.8 \cdot C + 32$  in binary fixed-point format.

**Checkpoint 7.10:** Using a fixed constant of  $10^{-3}$ , rewrite the digital filter  $y = x - 0.0532672 \cdot x_1 + x_2 + 0.0506038 \cdot y_1 - 0.9025 \cdot y_2$  in decimal fixed-point format.

**Checkpoint 7.11:** Assume resistors  $R_1$ ,  $R_2$ ,  $R_3$  are the integer parts of 16-bit unsigned binary fixed-point numbers with a fixed constant of  $2^{-4}$ . Write an equation to calculate  $R_3 = R_1 \parallel R_2$  (parallel combination.)

The purpose of this example is to study overflow and drop-out errors during integer calculations. The objective of the software is to calculate the circumference of a circle given its radius.

$$c = 2\pi r$$

Assume  $r$  is an unsigned 32-bit fixed-point number with a resolution of 0.001 cm.  $c$  is also fixed-point with the same resolution. I.e.,  $c = C * 0.001$  cm and  $r = R * 0.001$  cm, where  $C$  and  $R$  are unsigned 32-bit variable integers. Given 32-bit variables, the values of  $c$  can range from 0.000 to 4,294,967.295 cm. If we divide this by  $2\pi$ , this calculation should work for values of  $r$  ranging from 0 to 683,565.275 cm. We substitute the definitions of  $c$  and  $r$  into the equation to get an exact relationship between input  $R$  and output  $C$ ,

$$C = 2 * \pi * R$$

We need to convert this equation to a function with integer operations. One simple possibility is

$$C = 6283 * R / 1000$$

The difficulty with this equation is the multiply 6283 is the possibility of overflow. The largest value  $r$  can be without overflow is  $2^{32} / 6283 * 0.001$  cm = 683 cm, which is a 1000 times smaller than the range predicted by the  $c = 2\pi r$  equation. There are two approaches to reducing the effect of overflow. The first approach would be to promote to 64 bits, perform the operation, and then demote back to 32 bits. The second approach is to find a better approximation for  $2\pi$ . If we search the space of all integers  $(I_1, I_2)$  less than 255, such that  $I_1/I_2$  is as close to  $2\pi$  as possible, we find this possibility

$$C = 245 * R / 39$$

Notice that  $2\pi - 245/39 = 2\pi - 6.28205 = 0.0011$ , which means this calculation is just as accurate as the 6283/1000 approximation. However, the multiply by 245 is less likely to cause an overflow error as compared to the multiply by 6283. When dividing by an unsigned number we can implement rounding by adding half of the divisor to the dividend. In this example, we add 20.

$$C = (245 * R + 20) / 39$$

## 7.7. Conversions

In this section we will develop methods to convert between ASCII strings and binary numbers. Let's begin with a simple example. Let **Data** be a fixed length string of three ASCII characters. Each entry of **Data** is an ASCII character 0 to 9. Let **Data[0]** be the ASCII code for the hundred's digit, **Data[1]** be the ten's digit and **Data[2]** be the one's digit. Let **n** be an unsigned 32-bit integer. We will also need an index, **i**. The decimal digits 0 to 9 are encoded in ASCII as 0x30 to 0x39. So, to convert a single ASCII digit to decimal number, we simply subtract 0x30. To convert this string of 3 decimal digits into binary we calculate

$$n = 100*(\text{Data}[0]-0x30) + 10*(\text{Data}[1]-0x30) + (\text{Data}[2]-0x30);$$

This 3-digit ASCII string could also be calculated as

$$n = (\text{Data}[2]-0x30) + 10*((\text{Data}[1]-0x30) + 10*(\text{Data}[0]-0x30));$$

If **Data** were a string of 9 decimal digits we could put the above function into a loop

```
n = 0;  
for (i=0; i<9 ;i++){  
    n = 10*n + (Data[i]-0x30);  
}
```

If **Data** were a variable length string of ASCII characters terminated with a null character (0), we could convert it to binary using a while loop, as shown in Program 7.15. A pointer to the string is passed using call by reference. In the assembly version, the pointer R0 is incremented as the string is parsed. R1 contains the local variable **n**, R2 contains the data from the string, and R3 contains the constant 10.

```
; Input: R0 points to string,  
;       null-terminated string  
; Output: R0 contains number value  
Str2UDec  
    MOV R1,#0 ; n = 0  
    MOV R3,#10 ; R3 = 10  
loop LDRB R2,[R0] ; R2 = *pt  
    CMP R2,#0 ; null?  
    BEQ done ; if so, done  
    ADD R0,R0,#1 ; next pointer  
    SUB R2,R2,#0x30 ; ASCII to num  
    MUL R1,R1,R3 ; n = n*10  
    ADD R1,R1,R2 ; n*10+num  
    B loop  
done MOV R0,R1 ; return n  
    BX LR
```

```
// Convert ASCII string to  
//   unsigned 32-bit decimal  
// string is null-terminated  
uint32_t Str2UDec(char *pt){  
    uint32_t n = 0; // number  
    while (*pt != 0){  
        n = 10*n + (*pt)-0x30;  
        pt++;  
    }  
    return n;  
}
```

Program 7.15. Unsigned ASCII string to decimal conversion.

The example, shown in Program 7.16, uses an I/O device capable of sending and receiving ASCII characters. When using a development board, we can send serial data to/from the PC using the UART. The function **InChar()** returns an ASCII character from the I/O device. The function **OutChar()** sends an ASCII character to the I/O device. The function **InUDec()** will accept characters from the device until a carriage return (the **Enter** key) is typed. Only the numbers are echoed.

```
#define CR 0x0D
// Accept ASCII input in unsigned decimal format, up to 4294967295
// If n> 4294967295, it will truncate without reporting the error
uint32_t InUDec(void){ uint32_t n=0; char character;
    while((character=InChar()) != CR){ // accepts until <enter>
        if((character >= '0') && (character <= '9')){
            n = 10*n+(character-0x30); // overflows if above 4294967295
            OutChar(character);      // echo this character
        }
    }
    return n;
}
```

Program 7.16. Input an unsigned decimal number.

If the ASCII characters were to contain optional “+” and “-” signs, we could look for the presence of the sign character in the first position. If there is a minus sign, then set a flag. Next use our unsigned conversion routine to process the rest of the ASCII characters and generate the unsigned number, **n**. If the flag was previously set, we can negate the value **n**. Be careful to guarantee the + and – are only processed as the first character.

To convert an unsigned integer into a fixed length string of ASCII characters, we could use the integer divide. Assume **n** is an unsigned integer less than or equal to 999. In this simple program, the number 5 is converted to the string “005”, see Program 7.17.

```
// Data is a pointer to a 4-byte empty buffer
// n is the input 0 to 999
void UDec2Str(uint32_t n, char Data[4]){
    Data[0] = n/100 + 0x30;
    n = n%100;           // n is now between 0 and 99
    Data[1] = n/10 + 0x30;
    n = n%10;           // n is now between 0 and 9
    Data[2] = n + 0x30;
    Data[3] = 0;          // null termination
}
```

Program 7.17. Unsigned decimal to ASCII string conversion.

The functions in Program 7.18 convert numbers into the corresponding ASCII characters, including units. Instead of creating an output string, these functions output each character to display device by calling **OutChar**. The program **OutUDec3** is a simple program that outputs exactly three characters. For example the number 5 is displayed as “005”. The resolution is 0.001 V. The program **OutUHex** is a recursive function (similar to **OutUDec** in Program 5.21) that outputs a variable number of characters to display the number in hexadecimal.

```

void OutUDec3(uint32_t n){ // n is the input 0 to 999
    OutChar(n/100 + 0x30); n = n%100; // n is now between 0 and 99
    OutChar(n/10 + 0x30); n = n%10; // n is now between 0 and 9
    OutChar(n + 0x30);
}
void OutUHex(uint32_t number){ // Output a hexadecimal number
    if(number >= 0x10){
        OutUHex(number/0x10); // all but last digit
        OutUHex(number%0x10); // last hex digit
    }
    else{ // base case 0 to 15
        if(number < 0xA){
            OutChar(number + '0'); // 0 to 9
        }
        else{
            OutChar(number -10 +'A'); // A to F
        }
    }
}
}

void OutFDec(uint32_t i){ // fixed constant is 0.001
    OutUDec(i/1000); // left of the decimal point (Program 5.21)
    OutChar('.'); // decimal point
    OutChar(0x30+(i%1000)/100); // tenths digit
    OutChar(0x30+(i%100)/10); // hundredths digit
    OutChar(0x30+i%10); // thousandths digit
    OutChar('V');} // units
}

```

Program 7.18. Print 3-digit decimal, 32-bit hexadecimal, decimal fixed-point number to an output device.

## 7.8. \*IEEE Floating-point numbers

If the range of numbers is unknown or large, then the numbers must be represented in a floating-point format. Conversely, we can use fixed point when the range of values is small and known. Therefore, we will not need floating-point operations for most embedded system applications because fixed point is sufficient. Furthermore, if the processor does not have floating-point instructions then a floating-point implementation will run much slower than the corresponding fixed-point implementation. However, it is appropriate to know the definition of floating point. NASA believes that there are on the order of  $10^{21}$  stars in our Universe. Manipulating large numbers like these is not possible using integer or fixed-point formats. Another limitation with integer or fixed-point numbers is there are some situations where the range of values is not known at the time the software is being designed. In a Physics research project, you might be asked to count the rate at which particles strike a sensor. Since the experiment has never been performed before, you do not know in advance whether there will be 1 per second or 1 trillion per second. The applications with numbers of large or unknown range can be solved with floating-point numbers. **Floating point** is similar in format to fixed point, except the exponent is allowed to change at run time. Consequently, both the exponent and the mantissa will be stored. Just like with fixed-point numbers we will use binary exponents for internal calculations, and decimal exponents when interfacing with humans. This number system is called floating point because as the exponent varies, the binary point or decimal point moves.

The IEEE Standard for Binary Floating-Point Arithmetic or ANSI/IEEE Std 754-1985 is the most widely-used format for floating-point numbers. There are three common IEEE formats: single-precision (32-bit), double-precision (64-bit), and double-extended precision (80-bits). The 32-bit short real format as implemented by the TM4C123 is presented here. The floating-point format, f, for the single-precision data type is shown in Figure 7.11. Computers use binary floating point because it is faster to shift than it is to multiply/divide by 10.

Bit 31	Mantissa sign, s=0 for positive, s=1 for negative
Bits 30:23	8-bit biased binary exponent $0 \leq e \leq 255$
Bits 22:0	24-bit mantissa, m, expressed as a binary fraction, A binary 1 as the most significant bit is implied. $m = 1.m_1m_2m_3\dots m_{23}$

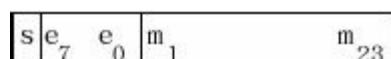


Figure 7.11. 32-bit single-precision floating-point format.

The value of a single-precision floating-point number is

$$f = (-1)^s \cdot 2^{e-127} \cdot m$$

The range of values that can be represented in the single-precision format is about  $\pm 10^{-38}$  to  $\pm 10^{+38}$ . The 24-bit mantissa yields a precision of about 7 decimal digits. The floating-point value is zero if both e and m are zero. Because of the sign bit, there are two zeros, positive and negative, which behave the same during calculations. To illustrate floating point, we will calculate the single-precision representation of the number 10. To find the binary representation of a floating-point number, first extract the sign.

$$10 = (-1)^0 \cdot 10$$

Step 2, multiply or divide by two until the mantissa is greater than or equal to 1, but less than 2.

$$10 = (-1)^0 \cdot 2^3 \cdot 1.25$$

Step 3, the exponent e is equal to the number of divide by twos plus 127.

$$10 = (-1)^0 \cdot 2^{130-127} \cdot 1.25$$

Step 4, separate the 1 from the mantissa. Recall that the 1 will not be stored.

$$10 = (-1)^0 \cdot 2^{130-127} \cdot (1+0.25)$$

Step 5, express the mantissa as a binary fixed-point number with a fixed constant of  $2^{-23}$ .

$$10 = (-1)^0 \cdot 2^{130-127} \cdot (1+2097152 \cdot 2^{-23})$$

Step 6, convert the exponent and mantissa components to hexadecimal.

$$10 = (-1)^0 \cdot 2^{0x82-127} \cdot (1+0x200000 \cdot 2^{-23})$$

Step 7, extract s, e, m terms, convert hexadecimal to binary

$$10 = (0,0x82,0x200000) = (0,10000010,010000000000000000000000)$$

Sometimes this conversion does not yield an exact representation, as in the case of 0.1. In particular, the fixed-point representation of 0.6 is only an approximation.

Step 1       $0.1 = (-1)^0 \cdot 0.1$

Step 2       $0.1 = (-1)^0 \cdot 2^{-4} \cdot 1.6$

Step 3       $0.1 = (-1)^0 \cdot 2^{123-127} \cdot 1.6$

Step 4       $0.1 = (-1)^0 \cdot 2^{123-127} \cdot (1+0.6)$

Step 5       $0.1 \approx (-1)^0 \cdot 2^{123-127} \cdot (1+5033165 \cdot 2^{-23})$

Step 6       $0.1 \approx (-1)^0 \cdot 2^{0x7B-127} \cdot (1+0x4CCCCD \cdot 2^{-23})$

Step 7       $0.1 \approx (0,0x7B,0x4CCCCD) = (0,0111011,1001100110011001101)$

The following example shows the steps in finding the floating-point approximation for  $\pi$ .

Step 1       $\pi = (-1)^0 \cdot \pi$

Step 2       $\pi \approx (-1)^0 \cdot 2^1 \cdot 1.570796327$

Step 3       $\pi \approx (-1)^0 \cdot 2^{128-127} \cdot 1.570796327$

Step 4       $\pi \approx (-1)^0 \cdot 2^{128-127} \cdot (1+0.570796327)$

Step 5       $\pi \approx (-1)^0 \cdot 2^{128-127} \cdot (1+4788187 \cdot 2^{-23})$

Step 6       $\pi \approx (-1)^0 \cdot 2^{0x80-127} \cdot (1+0x490FDB \cdot 2^{-23})$

Step 7       $\pi \approx (0,0x80,0x490FDB) = (0,10000000,1001001000011111011011)$

There are some special cases for floating-point numbers. When e is 255, the number is considered as plus or minus infinity, which probably resulted from an overflow during calculation. When e is 0, the number is considered as **denormalized**. The value of the mantissa of a denormalized number is less than 1. A denormalized short result number has the value,

$$f = (-1)^s \cdot 2^{-126} \cdot m$$

where  $m = 0.m_1m_2m_3\dots m_r$

**Observation:** The floating-point zero is stored in denormalized format

When two floating-point numbers are added or subtracted, the smaller one is first **unnormalized**. The mantissa of the smaller number is shifted right and its exponent is incremented until the two numbers have the same exponent. Then, the mantissas are added or subtracted. Lastly, the result is normalized. To illustrate the floating-point addition, consider the case of  $10+0.1$ . First, we show the original numbers in floating-point format. The mantissa is shown in binary format.

$$10.0 = (-1)^0 \cdot 2^3 \cdot 1.010000000000000000000000000000$$

$$+ 0.1 = (-1)^0 \cdot 2^{-4} \cdot 1.10011001100110011001101$$

Every time the exponent is incremented the mantissa is shifted to the right. Notice that 7 binary digits are lost. The 0.1 number is unnormalized, but now the two numbers have the same exponent. Often the result of the addition or subtraction will need to be normalized. In this case the sum did not need normalization.

$$\begin{aligned}
 10.0 &= (-1)^0 \cdot 2^3 \cdot 1.01000000000000000000000000 \\
 + 0.1 &= (-1)^0 \cdot 2^3 \cdot 0.0000001100110011001100110011001 \\
 10.1 &\equiv (-1)^0 \cdot 2^3 \cdot 1.01000011001100110011001
 \end{aligned}$$

When two floating-point numbers are multiplied, their mantissas are multiplied and their exponents are added. When dividing two floating-point numbers, their mantissas are divided and their exponents are subtracted. After multiplication and division, the result is normalized. To illustrate the floating-point multiplication, consider the case of  $10 \times 0.1$ . Let  $m_1, m_2$  be the values of the two mantissas. Since the range is  $1 \leq m_1, m_2 < 2$ , the product  $m_1 \times m_2$  will vary from  $1 \leq m_1 \times m_2 < 4$ .

$$\begin{aligned}10.0 &= (-1)^0 \cdot 2^3 \cdot 1.01000000000000000000000000 \\ * 0.1 &= (-1)^0 \cdot 2^{-4} \cdot 1.10011001100110011001101 \\ 1.0 &= (-1)^0 \cdot 2^{-1} \cdot 10.00000000000000000000000000\end{aligned}$$

The result needs to be normalized.

$$1.0 = (-1)^0 \cdot 2^0 \cdot 1.00000000000000000000000000$$

**Roundoff** is the error that occurs as a result of an arithmetic operation. For example, the multiplication of two 64-bit mantissas yields a 128-bit product. The final result is normalized into a normalized floating-point number with a 64-bit mantissa. Roundoff is the error caused by discarding the least significant bits of the product. Roundoff during addition and subtraction can occur in two places. First, an error can result when the smaller number is shifted right. Second, when two n-bit numbers are added the result is  $n+1$  bits, so an error can occur as the  $n+1$  sum is squeezed back into an n-bit result.

**Truncation** is the error that occurs when a number is converted from one format to another. For example, when an 80-bit floating-point number is converted to 32-bit floating-point format, 40 bits are lost as the 64-bit mantissa is truncated to fit into the 24-bit mantissa. Recall, the number 0.1 could not be exactly represented as a short real floating-point number. This is an example of truncation as the true fraction was truncated to fit into the finite number of bits available.

If the range is known and small and a fixed-point system can be used, then a 32-bit fixed-point number system will have better resolution than a 32-bit floating-point system. For a fixed range of values (i.e., one with a constant exponent), a 32-bit floating-point system has only 23 bits of precision, while a 32-bit fixed-point system has 9 more bits of precision.

Figure 7.11 shows the floating-point registers on the Cortex M4. Software can access these registers in any combination of 32 single-precision registers named S0 to S31 or 16 double-precision registers D0 to D15. In particular, registers S0 and S1 are the same as register D0. This section will focus on single precision floating-point operations.

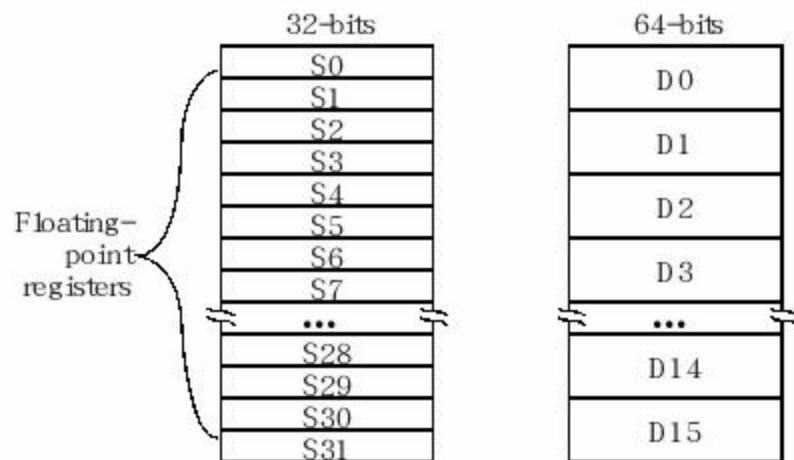


Figure 7.12. The TM4C has 32 single-precision floating-point registers that overlap with 16 double-precision floating-point registers.

The following lists the general form for some of the load and store instructions. Because the constant is stored into memory, and the assembly creates a PC relative access, the constant can be any single-precision floating-point value. **St** **Sd** **Sn** and **Sm** represent any of the 32 single-precision floating-point registers. **Rn** and **Rd** are regular integer registers.

**VLDR.F32 Sd, [Rn] ; load 32-bit float at [Rn] to Sd**

**VSTR.F32 St, [Rn] ; store 32-bit St to memory at [Rn]**

**VLDR.F32 Sd, [Rn, #n] ; load 32-bit memory at [Rn+n] to Sd**

**VSTR.F32 St, [Rn, #n] ; store 32-bit St to memory [Rn+n]**

**VLDR.F32 Sd, =constant ; load 32-bit constant into Sd**

The move instructions get their data from the machine instruction or from within the processor and do not require additional memory access instructions. The immediate value is any number that can be expressed as  $\pm n \cdot 2^r$ , where  $16 \leq n \leq 31$ , and  $0 \leq r \leq 7$ .

**VMOV.F32 Sd, Sn ; set Sd equal to the value in Sn**

**VMOV.F32 Sd, #imm ; set Sd equal to imm**

**VMOV Rd, Sn ; set Rd equal to the value in Sn**

**VMOV Sd, Rn ; set Sd equal to the value in Rn**

These are some of the arithmetic operations, which operate on the floating-point registers. Arithmetic operations can cause overflow, underflow, divide by zero floating-point exceptions. In particular, bits in the **SYSEXC\_RIS\_R** register will get set if there is a floating-point error.

```
VADD.F32 Sd, Sn, Sm ; set Sd equal to Sn+Sm
VSUB.F32 Sd, Sn, Sm ; set Sd equal to Sn-Sm
VMUL.F32 Sd, Sn, Sm ; set Sd equal to Sn*Sm
VDIV.F32 Sd, Sn, Sm ; set Sd equal to Sn/Sm
VNEG.F32 Sd, Sm      ; set Sd equal to -Sm
VABS.F32 Sd, Sm      ; set Sd equal to the absolute value of Sm
VSQRT.F32 Sd, Sm     ; set Sd equal to the square root of Sm
```

To enable the floating-point hardware on the Cortex M4, we need to set bits 23-20 of the Coprocessor Access Control ( **NVIC\_CPAC\_R** ) register.

```
NVIC_CPAC_R equ 0xE000ED88
```

**EnableFPU**

```
LDR R0, =NVIC_CPAC_R
LDR R1, [R0]          ; Read CPAC
ORR R1, R1, #0x00F00000 ; enable floating coprocessor
STR R1, [R0]          ; Write CPAC
BX LR
```

The following example calculates the area of a circle. The radius input is passed by value in register S0 and the output is returned by value also in register S0. In C, we define a single-precision floating-point variable using **float** .

```
AREA DATA, ALIGN=2
In SPACE 4
Area SPACE 4
AREA [.text],CODE,READONLY,ALIGN=2
THUMB
; Input: S0 is radius r in cm
; Output: S0 is area in cm^2
CircleArea
    VMUL.F32 S0,S0,S0 ; r*r
    VLDR.F32 S1,-3.14159265
    VMUL.F32 S0,S0,S1 ; pi*r*r
    BX LR
Test PUSH {R4,LR}
    LDR R0,=In
    VLDR.F32 S0,[R0] ;S0 is In
    BL CircleArea
    LDR R0,=A
    VSTR.F32 S0,[R0] ;A = pi*In*In
    POP {R4,PC}
```

```
float In; // radius in cm
float A; // area in cm^2
```

```
float CircleArea(float r){
    return 3.14159265*r*r;
}
```

```
void Test(void){
    A = CircleArea(In);
}
```

Program 7.22. Floating-point function to calculate the area of a circle (Float\_xxx.zip).

**Observation:** The assembly version of CircleArea executes in 22 bus cycles, while the C version requires 110 bus cycles. A C version of CircleArea running on a LM3S1968 without floating point hardware runs in 165 bus cycles.

## 7.9. Exercises

- 7.1 What does it mean to say a function is public versus private? Why is this distinction important?
- 7.2 What does it mean to say a variable is public versus private? Why is this distinction important?
- 7.3 What does it mean to say a variable is local versus global?
- 7.4 Consider the reasons why one chooses which technique to create a variable.
- a) List three reasons why one would implement a variable using a register.
  - b) List three reasons why one would implement a variable on the stack and access it using R11 indexed mode addressing.
  - c) List three reasons why one would implement a variable in permanently allocated RAM.
- 7.5 Consider reasons for implementing "call by value" versus "call by reference"
- a) List two reasons for implementing "call by value".
  - b) List two reasons for implementing "call by reference"
- 7.6 Give an approximation of  $\sqrt{2}$  using the decimal fixed-point ( $\Delta= 0.001$ ) format.
- 7.7 Give an approximation of  $\sqrt{2}$  using the binary fixed-point ( $\Delta= 2^{-8}$ ) format.
- 7.8 Assume **M** and **N** are two integers, each less than 1000. Find the best set of **M** and **N**, such that **M/N** is approximately  $\sqrt{2}$ . (Like 7/5, but much more accurate).
- 7.9 Assume **M** and **N** are two integers, each less than 1000. Find the best set of **M** and **N**, such that **M/N** is approximately  $\pi$ . (Like 22/7, but much more accurate).
- 7.10 Give an approximation of  $\sqrt{101}$  using the decimal fixed-point ( $\Delta= 0.01$ ) format.
- 7.11 Give an approximation of  $\sqrt{99}$  using the binary fixed-point ( $\Delta= 2^{-4}$ ) format.
- 7.12 A signed 16-bit binary fixed-point number system has a  $\Delta$  resolution of 1/256. What is the corresponding value of the number if the integer part stored in memory is 384?
- 7.13 An unsigned 16-bit decimal fixed-point number system has a  $\Delta$  resolution of 1/100. What is the corresponding value of the number if the integer part stored in memory is 384?
- D7.14 Write assembly code that finds the average value of a 10-element array. Each element is unsigned 32 bits. A pointer to the array is passed in R0 and you return the result in R0. A typical calling sequence:

**LDR R0,=mydata ; pointer to 10-element structure**  
**BL Average**  
**;result in R0**

**D7.15** Write assembly code that calculates the average of three unsigned 32-bit numbers. The three parameters are passed by value on the stack. You return the result in R0. A typical calling sequence:

```
MOV R0,#100
MOV R1,#200
MOV R2,#400
PUSH {R0-R2}
BL Average
;result in R0
ADD SP,#12 ;balance stack
```

**D7.16** Write assembly code that finds the maximum of value of a 10-element array. Each element is unsigned 16 bits. The two parameters are passed by reference on the stack. A typical calling sequence:

```
LDR R0,=mydata ;pointer to 10-element structure
PUSH {R0}
SUB SP,#4 ;place for the result
BL max
POP {R0} ;result
ADD SP,#4 ;balance stack
```

**D7.17** Write assembly code that sorts three unsigned numbers. The three parameters are passed by value in R0, R1, and R2. You return the results back in the same registers such that  $R0 \leq R1 \leq R2$ . A typical calling sequence:

```
MOV R0,#400
MOV R1,#200
MOV R2,#100
BL Sort ;shifts numbers around such that R0≤R1≤R2
```

**D7.18** Write assembly code that converts temperature in Fahrenheit to temperature in Centigrade. Both input and output are fixed-point 0.1. The input parameter is passed by value on the stack. You return the result in R0.

```
MOV R0,#720 ; 72.0F
PUSH {R0}
BL FtoC
ADD SP,#4 ; balance stack
```

**D7.19** Write assembly code that converts temperature in Centigrade to temperature in Fahrenheit. Both input and output are fixed-point 0.01. The input parameter is passed by value in R0. You return the result in R0. A typical calling sequence:

```
MOV R0,#2500 ;25.00C
```

**BL CtoF**  
**;result in R0**

**D7.20** Write assembly code that finds the median of value of a 5-element array. Each element is unsigned 8 bits. The input parameter is passed by reference in R0. You return the result in R0. A typical calling sequence:

**LDR R0,=mydata ; pointer to 5-element structure**  
**BL Median**  
**;result in R0**

**D7.21** Using recursion, write a subroutine that calculates the Fibonacci function. In particular,

**fib(0) = 1**

**fib(1) = 1**

**fib(n) = fib(n-1)+fib(n-2) for n>1**

The input is passed by value in R0, and the result is also returned by value in R0.

**D7.22** First, rewrite the following digital filter using decimal fixed-point math. Assume the inputs are unsigned 10-bit values (0 to 1023). Then, rewrite it so that it can be calculated with integer math using the fact that 0.11111 is about 1/9 and 0.088889 is about 4/45 and 0.8 is 4/5. In both cases, the calculations are to be performed in 32-bit unsigned integer form without overflow.

$$y = 0.11111 \cdot x + 0.08889 \cdot x_1 + 0.80000 \cdot y_1$$

**7.23** Does the associative principle hold for signed integer multiply and divide? Assume **Out1 Out2** **A B C** are all the same precision (e.g., 32 bits). In particular do these two C calculations always achieve identical outputs? If not, give an example.

**Out1 = (A\*B)/C;**  
**Out2 = A\*(B/C);**

**7.24** Does the associative principle hold for signed integer addition and subtraction? Assume **Out3 Out4** **A B C** are all the same precision (e.g., 32 bits). In particular do these two C calculations always achieve identical outputs? If not, give an example.

**Out3 = (A+B)-C;**  
**Out4 = A+(B-C);**

**D7.25** Write an assembly subroutine that implements an averaging filter. The three 16-bit unsigned numbers are passed into the subroutine by value in Registers R0, R1 and R2. The average is (first+second+third)/3. The return parameter is passed back in Register R0. If you need a temporary variable, you should use the stack.

**7.26** Give the short real floating-point representation of  $\sqrt{2}$ . Give the short real floating-point representation of -134.4. Give the short real floating-point representation of -0.0123.

**D7.27** Assume we have 10-dimensional vectors, stored as 10-element arrays. For example, let the vector **X** equal  $(x_0, x_1, \dots, x_9)$ . Each value is a signed 32-bit integer. Write assembly code that finds the dot-product of two 10-element vectors. Neglect overflow.

$$\mathbf{X} \cdot \mathbf{Y} = x_0 * y_0 + x_1 * y_1 + \dots + x_9 * y_9$$

The two parameters are passed by reference using registers, R0 and R1. The result is to be returned as a 32-bit signed value in Register R0.

**D7.28** Write a subroutine to implement linear regression. A typical calling sequence:

**LDR R0,= DataSet1 ; pointer to 10-element structure**

**BL Regression**

**; R0 = m =slope as a fixed point**

**; R1 = b =offset as a fixed point**

**; R2 = e =average error as a fixed point**

Input x, y numbers will be in signed two's complement 32-bit decimal fixed point with a resolution,  $\Delta$ , of 0.01. Outputs m,b,e will be in signed two's complement 32-bit decimal fixed point with a resolution,  $\Delta$ , of 0.0001. Ignore overflow. A typical array structure looks like:

**DataSet1 DCD 3 ; number of data points**

**DCD 0,1 ; (x,y) = (0 , 0.01)**

**DCD 10,2 ; (x,y) = (0.1 , 0.02)**

**DCD 20,3 ; (x,y) = (0.2 , 0.03)**

For this example **b=0.01** , **m=0.1** , **e=0** , so R0 is returned as 100, R1 is 1000, and R2 is 0. Let  $(x_0, y_0)$  and  $(x_1, y_1)$  be two points, then the slope and intercept of the “ $y=mx+b$ ” line through those points is given by

$m = \frac{y_1 - y_0}{x_1 - x_0}$        $b = y_0 - mx_0$       and  $e = 0$

In general, let  $x(i)$  and  $y(i)$  be arrays of length  $n > 2$ . Each of the following sums range from  $i=0$  to  $n-1$ .  
 $m = \frac{\sum_{i=0}^{n-1} (y_i - mx_i)}{n}$       and  $b = y_0 - mx_0$

For  $n > 2$ , the average error is defined as

$e = \frac{\sum_{i=0}^{n-1} (y_i - mx_i)^2}{n}$

---

## 7.10. Lab Assignments

**Lab 7.1** Fast conversion. The goal of this lab is to device a high speed number to ASCII conversion function. First implement program 7.17 and write a main program that calls it. Use the SysTick technique decribed in Section 4.7 to measure how fast the **UDec2Str** program requires to execute. Automate the measurement so the function can be profiled for all input data values from 0 to 999. Next, implement a fast version using table lookup. Goal is to improve execution speed at the expense of memory.

**Lab 7.2** Fast conversion. In a similar way write three functions that implement ASCII to hexadecimal conversion. The input is one ASCII character ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’ and the output is a 4-bit number representing the hexadecimal value of the input. The first function uses if-then, the second function uses a switch statement, and the third function uses a table lookup. Use the SysTick technique decribed in Section 4.7 to measure how fast each function requires to execute. Automate the measurement so the function can be profiled for all possible input values.

**Lab 7.3** Sort. Write a function that sorts an array of signed 32-bit numbers. Implement it at least two ways. Design a method to test the functionality of the sort programs. Compare the sort speed of the two methods. Test the functions with sizes 10, 100, and 1000 elements.

# 8. Serial and Parallel Port Interfacing

## Chapter 8 objectives are to:

- Describe the UART and SSI serial ports,
- Discuss how to interface a keyboard using scanning,
- Explain and interface electromechanical devices that are binary in nature,
- Use pulse width modulation (PWM) to control power delivered to a DC motor,
- Interface and control a stepper motor.

A more detailed approach to interfacing can be found in Volume 2 of these series. However, in this chapter is an introduction to I/O interfacing. The common theme of this chapter is I/O interfacing connected to serial and parallel ports. The chapter begins with a discussion of UART and SSI ports, which can be used to interface external devices to the microcontroller such as GPS, DAC, LCD, OLED, and ADC devices. Various I/O devices such as keyboards, optical sensors, relays, solenoids, DC motors, and stepper motors will be interfaced. In addition, the pulse width modulation will be used when interfacing DC motors so that the software can control power delivered to the motor. Advances in the number and sophistication of the I/O ports have contributed greatly to the long term growth of applications of embedded systems. This book covers just some of the ports and some of the features of the LM3S/TM4C family of microcontrollers. For a complete list of I/O ports refer to the respective data sheets.

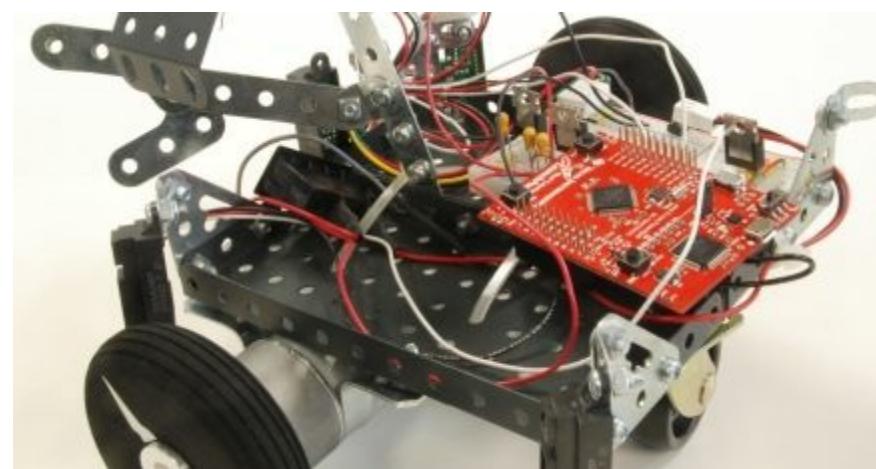


Figure 8.1. Geared DC motors provide a good torque and speed for light-weight robots.

# 8.1. General Introduction to Interfacing

There are three components to microcomputer interfacing. Since many external devices have physical characteristics, the first step is the mechanical design of the physical components. Often, the mechanical design is simply selecting the physical devices from a list of available components. The next step is the analog and digital electronics used to connect the physical devices to the computer. The voltage levels of the external device must be translated into values compatible with the microcontroller. The RS232 interface using the MAX3232 interface in Figure 8.3 is a typical example of this translation. Some external devices need the interface to source or sink current, and the interfaces in Figures 8.14, 8.18 and 8.23 can be used for these applications. The input/output information may be encoded as simple digital signals or variable analog signals. Interfacing with analog signals will be presented in Chapter 11. More complex systems may use frequency, period, phase, or pulse width to represent the signals. Interfacing with time-based signals will be presented in Chapter 9. The third component of interfacing is the low-level software that transforms the mechanical and electrical devices into objects that perform the desired tasks. The group of these low-level functions is often designated as an I/O device driver. Since this book serves as an introduction to interfacing, most of the hardware circuits are given, and the software design is explained. More details can be found in Volume 2.

Most microcontrollers are built with CMOS logic. Interfacing with CMOS logic involves consideration of voltage, current, and capacitance. First, let's consider a digital output, e.g., a port pin with its direction register equal to 1.  $I_{OH}$  is the largest current a port pin can source when the output is high.  $V_{OH}$  is the smallest voltage a port pin will be if the output is high and the current is less than  $I_{OH}$ . If the microcontroller is powered by  $V_{DD}$ , then the output high voltage will be between  $V_{OH}$  and  $V_{DD}$ .  $I_{OL}$  is the largest current a port pin can sink when the output is low.  $V_{OL}$  is the largest voltage a port pin will be if the output is low and the current is less than  $I_{OL}$ . The output low voltage will be between 0 and  $V_{OL}$ . Next, let's consider a digital input, e.g., a port pin with its direction register equal to 0.  $I_{IH}$  is the current the input port pin will require when its input is high.  $V_{IH}$  is the voltage above which the input will be considered high.  $I_{IL}$  is the current the input port pin will require when its input is low.  $V_{IL}$  is the voltage below which the input will be considered low. Figure 8.2 shows current parameters for various digital logic families, and Table 8.1 shows voltage parameters. In summary, if the input is between 0 and  $V_{IL}$ , it is considered a low. If the input is between  $V_{IH}$  and  $V_{DD}$ , it is considered a high. Refer back to the transistor-level implementation in Figure 3.5. If the voltage on an input pin remains between  $V_{IL}$  and  $V_{IH}$  for a long time, both the p-type and n-type transistors will be active, causing a short circuit from power to ground. With some CMOS microcomputers one must define unused I/O pins either as outputs or specify them as inputs and tie the pin high (or low) in hardware. On the LM3S/TM4C family of microcontrollers we can leave an unused pin disconnected if the software leaves its **DEN** pins zero. Because of the extremely high impedance of CMOS inputs, an unconnected input pin may oscillate, dissipating power unnecessarily. In order for the output to properly drive all the inputs of the next stage, the maximum available output current must be larger than the sum of all the required input currents for both the high and low conditions.

$|I_{OL}|^3 \leq |I_{IH}|$

and

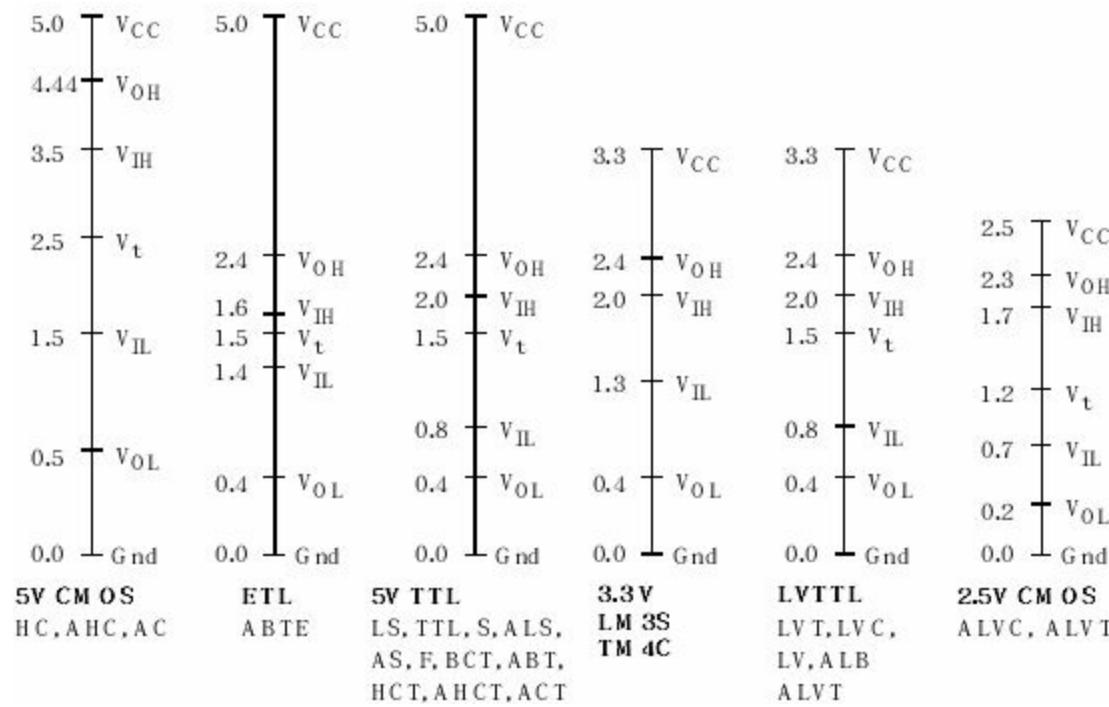
$|I_{OL}|^3 \geq |I_{IL}|$

In order for the digital information to be properly transferred from the output of one module to the input of the next, we need the output high voltage to be more than the required input high voltage, and the output low voltage to be less than the input low voltage, see Figure 8.2.

$V_{OH} \geq V_{IH}$  and  $V_{OL} \leq V_{IL}$

Family	Example	$I_{OH}$	$I_{OL}$	$I_{IH}$	$I_{IL}$
Standard TTL	7404	0.4 mA	16 mA	40 $\mu$ A	1.6 mA
Schottky TTL	74S04	1 mA	20 mA	50 $\mu$ A	2 mA
Low Power Schottky	74LS04	0.4 mA	4 mA	20 $\mu$ A	0.4 mA
High Speed CMOS	74HC04	4 mA	4 mA	1 $\mu$ A	1 $\mu$ A
TM4C 2mA-drive	TM4C123	2 mA	2 mA	2 $\mu$ A	2 $\mu$ A
TM4C 4mA-drive	TM4C123	4 mA	4 mA	2 $\mu$ A	2 $\mu$ A
TM4C 8mA-drive	TM4C123	8 mA	8 mA	2 $\mu$ A	2 $\mu$ A
TM4C 12mA-drive	TM4C1294	12 mA	12 mA	2 $\mu$ A	2 $\mu$ A

**Table 8.1. The input and output currents of various digital logic families and microcontrollers.**



**Figure 8.2. Voltage thresholds for various digital logic families.**

The last consideration for interfacing with CMOS logic is capacitance. Capacitance loading occurs with each input and with long cables. ADC pins on the LM3S/TM4C microcontrollers have an input capacitance around 1 pF. A typical input capacitance on CMOS logic is about 5 pF. Consider a situation where the output of one circuit is attached to the input of another. If the output goes from 0 to +3.3V, the voltage as perceived at the input of the next stage will be

$$v(t) = 3.3 - 3.3e^{-t/\tau}$$

where R is the resistance in the circuit, and C is the capacitive load.  $\tau = R \cdot C$  is called the time constant. If the time constant is very small, the input goes from 0 to +3.3 almost immediately after the output goes 0 to +3.3V. If the signal is a square wave with period T, the interface will only work for situations where the period T is large compared to the time constant  $\tau$ .

I/O ports are the specific components of a microcomputer that allow it to interact with its environment. A **device driver** is a collection of software functions that allow higher level software to utilize an I/O device. In other words, the set of low-level functions that input/output directly with the hardware are grouped together in a single module and called a device driver.

## 8.2. Universal Asynchronous Receiver Transmitter (UART)

In this section we will develop a simple device driver using the Universal Asynchronous Receiver/Transmitter (UART). This serial port allows the microcontroller to communicate with devices such as other computers, printers, input sensors, and LCDs. Serial transmission involves sending one bit at a time, such that the data is spread out over time. The total number of bits transmitted per second is called the **baud rate**. The reciprocal of the baud rate is the **bit time**, which is the time to send one bit. Most microcontrollers have at least one UART. Before discussing the detailed operation on the LM3S/TM4C, we will begin with general features common to all devices. Each UART will have a baud rate control register, which we use to select the transmission rate. Each device is capable of creating its own serial clock with a transmission frequency approximately equal to the serial clock in the computer with which it is communicating. A **frame** is the smallest complete unit of serial transmission. Figure 8.3 plots the signal versus time on a serial port, showing a single frame, which includes a **start bit** (which is 0), 8 bits of data (least significant bit first), and a **stop bit** (which is 1). There is always only one start bit, but the Stellaris ® and Tiva ® UARTs allow us to select the 5 to 8 data bits and 1 or 2 stop bits. The UART can add even, odd, or no parity bit. However, we will employ the typical protocol of 1 start bit, 8 data bits, no parity, and 1 stop bit. This protocol is used for both transmitting and receiving. The information rate, or **bandwidth**, is defined as the amount of data or useful information transmitted per second. From Figure 8.3, we see that 10 bits are sent for every byte of usual data. Therefore, the bandwidth of the serial channel (in bytes/second) is the baud rate (in bits/sec) divided by 10.

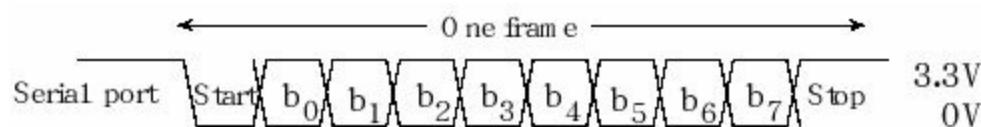


Figure 8.3. A serial data frame with 8-bit data, 1 start bit, 1 stop bit, and no parity bit.

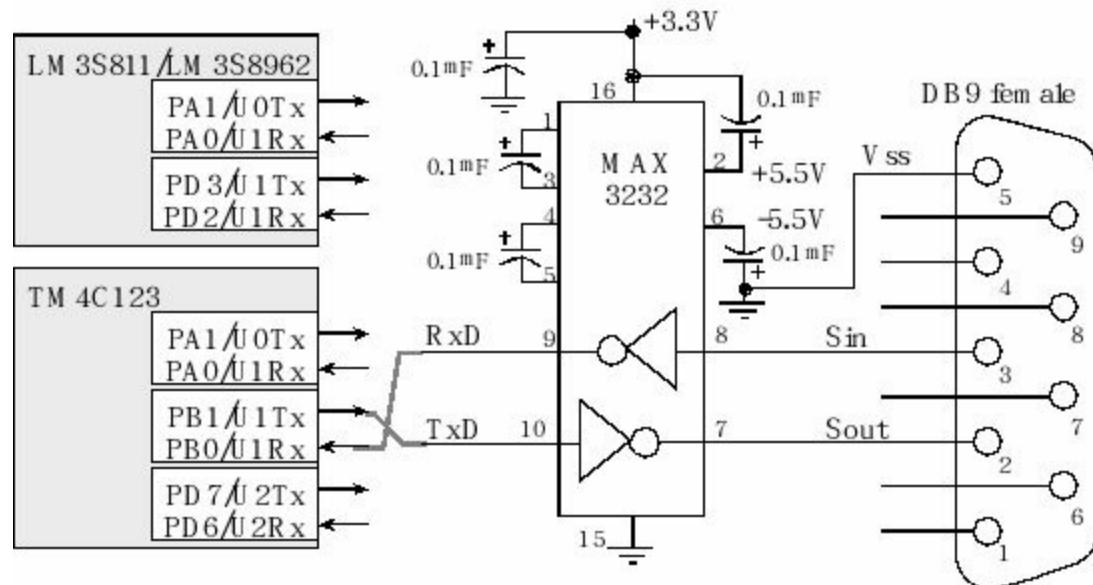
**Common Error:** If you change the bus clock frequency without changing the baud rate register, the UART will operate at an incorrect baud rate.

**Checkpoint 8.1:** Assuming the protocol drawn in Figure 8.3 and a baud rate of 1000 bits/sec, what is the bandwidth in bytes/sec?

Table 8.2 shows the three most commonly used RS232 signals. The RS232 standard uses a DB25 connector that has 25 pins. The EIA-574 standard uses RS232 voltage levels and a DB9 connector that has only 9 pins. The most commonly used signals of the full RS232 standard are available with the EIA-574 protocols. Only **TxD**, **RxD**, and **SG** are required to implement a simple bidirectional serial channel, thus the other signals are not shown (Figure 8.4). We define the **data terminal equipment** (DTE) as the computer or a terminal and the **data communication equipment** (DCE) as the modem or printer.

DB25 Pin	RS232 Name	DB9 Pin	EIA-574 Name	Signal	Description	True	DTE	DCE
2	BA	3	103	TxD	Transmit Data	-5.5V	out	in
3	BB	2	104	RxD	Receive Data	-5.5V	in	out
7	AB	5	102	SG	Signal Ground			

**Table 8.2. The commonly-used signals on the RS232 and EIA-574 protocols.**



**Figure 8.4. Hardware interface implementing an asynchronous RS232 channel. The TM4C123 and TM4C1294 have eight UART ports (see Tables 4.3 and 4.4).**

**Observation:** Most LM3S/TM4C development kits send UART0 channel through the USB cable, so the circuit shown in Figure 8.4 will not be needed. On the PC side of the cable, the serial channel becomes a virtual COM port.

RS232 is a non-return-to-zero (NRZ) protocol with true signified as a voltage between -5 and -15 V. False is signified by a voltage between +5 and +15 V. A MAX3232 converter chip is used to translate between the +5.5/-5.5 V RS232 levels and the 0/+3.3 V digital levels. The capacitors in this circuit are important, because they form a charge pump used to create the ±5.5 voltages from the +3.3 V supply. The RS232 timing is generated automatically by the UART. During transmission, the Maxim chip translates a digital high on microcontroller side to -5.5V on the RS232/EIA-574 cable, and a digital low is translated to +5.5V. During receiving, the Maxim chip translates negative voltages on RS232/EIA-574 cable to a digital high on the microcontroller side, and a positive voltage is translated to a digital low. The computer is classified as DTE, so its serial output is pin 3 in the EIA-574 cable, and its serial input is pin 2 in the EIA-574 cable. When connecting a DTE to another DTE, we use a cable with pins 2 and 3 crossed. I.e., pin 2 on one DTE is connected to pin 3 on the other DTE and pin 3 on one DTE is connected to pin 2 on the other DTE. When connecting a DTE to a DCE, then the cable passes the signals straight across. In all situations, the grounds are connected together using the SG wire in the cable. This channel is classified as **full-duplex**, because transmission can occur in both directions simultaneously.

## 8.2.1. Asynchronous Communication

We will begin with transmission, because it is simple. The transmitter portion of the UART includes a data output pin, with digital logic levels as drawn in Figure 8.5. The transmitter has a 16-element FIFO and a 10-bit shift register, which cannot be directly accessed by the programmer (Figure 8.5). The FIFO and shift register in the transmitter are separate from the FIFO and shift register associated with the receiver. To output data using the UART, the software will first check to make sure the transmit FIFO is not full (it will wait if TXFF is 1) and then write to the transmit data register (e.g., **UART0\_DR\_R**). The bits are shifted out in this order: start,  $b_0$ ,  $b_1$ ,  $b_2$ ,  $b_3$ ,  $b_4$ ,  $b_5$ ,  $b_6$ ,  $b_7$ , and then stop, where  $b_0$  is the LSB and  $b_7$  is the MSB. The transmit data register is write only, which means the software can write to it (to start a new transmission) but cannot read from it. Even though the transmit data register is at the same address as the receive data register, the transmit and receive data registers are two separate registers.

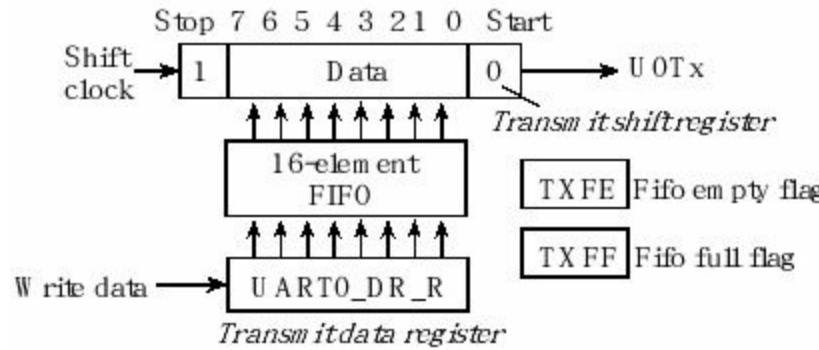


Figure 8.5. Data and shift registers implement the serial transmission.

When a new byte is written to **UART0\_DR\_R**, it is put into the transmit FIFO. Byte by byte, the UART gets data from the FIFO and loads them into the 10-bit transmit shift register. The 10-bit shift register includes a start bit, 8 data bits, and 1 stop bit. Then, the frame is shifted out one bit at a time at a rate specified by the baud rate register. If there are already data in the FIFO or in the shift register when the **UART0\_DR\_R** is written, the new frame will wait until the previous frames have been transmitted, before it too is transmitted. The FIFO guarantees the data are transmitted in the order they were written. The serial port hardware is actually controlled by a clock that is 16 times faster than the baud rate, referred to in the datasheet as **Baud16**. When the data are being shifted out, the digital hardware in the UART counts 16 times in between changes to the **U0Tx** output line.

The software can actually write 16 bytes to the **UART0\_DR\_R**, and the hardware will send them all one at a time in the proper order. This FIFO reduces the software response time requirements of the operating system to service the serial port hardware. Unfortunately, it does complicate the hardware/software timing. At 9600 bits/sec, it takes 1.04 ms to send a frame. Therefore, there will be a delay ranging from 1.04 and 16.7 ms between writing to the data register and the completion of the data transmission. This delay depends on how much data are already in the FIFO at the time the software writes to **UART0\_DR\_R**.

Receiving data frames is a little trickier than transmission because we have to synchronize the receive shift register with the incoming data. The receiver portion of the UART includes a **U0Rx** data input pin with digital logic levels. At the input of the microcontroller, true is 3.3V and false is 0V. There is also a 16-element FIFO and a 10-bit shift register, which cannot be directly accessed by the programmer (Figure 8.6). Again the receive shift register and receive FIFO are separate from those in the transmitter. The receive data register, **UART0\_DR\_R**, is read only, which means write operations to this address have no effect on this register (recall write operations activate the transmitter). The receiver obviously cannot start a transmission, but it recognizes a new frame by its start bit. The bits are shifted in using the same order as the transmitter shifted them out: start, **b<sub>0</sub>**, **b<sub>1</sub>**, **b<sub>2</sub>**, **b<sub>3</sub>**, **b<sub>4</sub>**, **b<sub>5</sub>**, **b<sub>6</sub>**, **b<sub>7</sub>**, and then stop.

There are six status bits generated by receiver activity. The Receive FIFO empty flag, **RXFE**, is clear when new input data are in the receive FIFO. When the software reads from **UART0\_DR\_R**, data are removed from the FIFO. When the FIFO becomes empty, the **RXFE** flag will be set, meaning there are no more input data. There are other flags associated with the receiver. There is a Receive FIFO full flag **RXFF**, which is set when the FIFO is full. There are four status bits associated with each byte of data. For this reason, the receive FIFO is 12 bits wide. The overrun error, **OE**, is set when input data are lost because the FIFO is full and more input frames are arriving at the receiver. An overrun error is caused when the receiver interface latency is too large. The break error, **BE**, is set when the input is held low for more than a frame. The **PE** bit is set on a parity error. Because the error rate is so low, most systems do not implement parity. The framing error, **FE**, is set when the stop bit is incorrect. Framing errors are probably caused by a mismatch in baud rate.

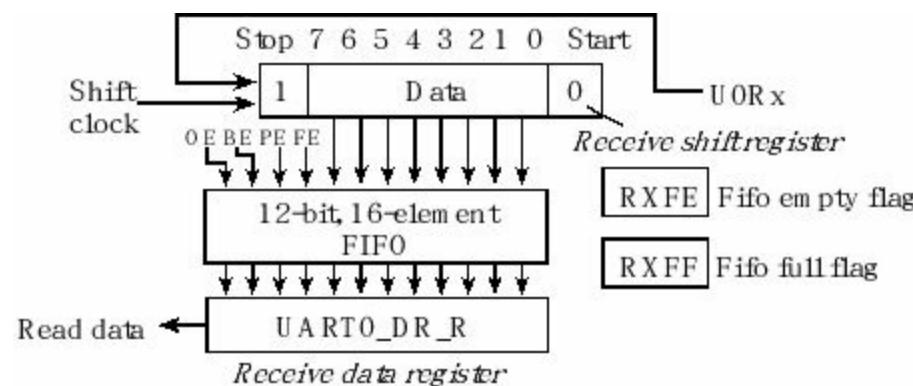


Figure 8.6. Data register shift registers implement the receive serial interface.

The receiver waits for the 1 to 0 edge signifying a start bit, then shifts in 10 bits of data one at a time from the **U0Rx** line. The internal **Baud16** clock is 16 times faster than the baud rate. After the 1 to 0 edge, the receiver waits 8 **Baud16** clocks and samples the start bit. 16 **Baud16** clocks later it samples **b<sub>0</sub>**. Every 16 **Baud16** clocks it samples another bit until it reaches the stop bit. The UART needs an internal clock faster than the baud rate so it can wait the half a bit time between the 1 to 0 edge beginning the start bit and the middle of the bit window needed for sampling. The start and stop bits are removed (checked for framing errors), the 8 bits of data and 4 bits of status are put into the receive FIFO. The FIFO implements hardware buffering so data can be safely stored if the software is performing other tasks.

**Observation:** If the receiving UART device has a baud rate mismatch of more than 5%, then a framing error can occur when the stop bit is incorrectly captured.

An overrun occurs when there are 16 elements in the receive FIFO, and a 17<sup>th</sup> frame comes into the receiver. In order to avoid overrun, we can design a real-time system, i.e., one with a maximum latency. The latency of a UART receiver is the delay between the time when new data arrives in the receiver (**RXFE=0**) and the time the software reads the data register. If the latency is always less than 160 bit times, then overrun will never occur.

**Observation:** With a serial port that has a shift register and one data register (no FIFO buffering), the latency requirement of the input interface is the time it takes to transmit one data frame.

In the example illustrated in Figure 8.7, assume the UART receive shift register and receive FIFO are initially empty (**RXFE=1**). 17 incoming serial frames occur one right after another (letters A – Q), but the software does not respond. At the end of the first frame, the 0x41 (letter ‘A’) goes into the receive FIFO, and the **RXFE**flag is cleared. Normally, the **UART\_InChar** function would respond to **RXFE** being clear and read the data from the UART. In this scenario however, the software is busy doing other things and does not respond to the presence of data in the receive FIFO. Next, 15 more frames are shifted in and entered into the receive FIFO. At the end of the 16th frame, the FIFO is full (**RXFF=1**). If the software were to respond at this point, then all 16 characters would be properly received. If the 17<sup>th</sup> frame occurs before the first is read by the software, then an overrun error occurs, and a frame is lost. We can see from this worst case scenario that the software must read the data from the UART within 160 bit times of the clearing of **RXFE**.

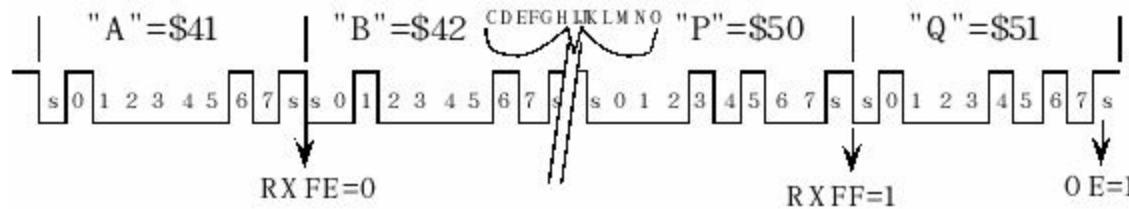


Figure 8.7. Seventeen receive data frames result in an overrun (OE) error.

## 8.2.2. LM3S/TM4C UART Details

Next we will overview the specific UART functions on the LM3S/TM4C microcontrollers. This section is intended to supplement rather than replace the Texas Instruments manuals. When designing systems with any I/O module, you must also refer to the reference manual of your specific microcontroller. It is also good design practice to review the errata for your microcontroller to see if any quirks (mistakes) exist in your microcontroller that might apply to the system you are designing.

The LM3S/TM4C microcontrollers have one to eight UARTs. The specific port pins used to implement the UARTs vary from one chip to the next. To find which pins your microcontroller uses, you will need to consult its datasheet. Table 8.3 shows some of the registers for the UART0. If the microcontroller has a second or third UART, the register names will replace the 0 with a 1 – 7. For the exact register addresses, you should include the appropriate header file (e.g., **tm4c1294ncpdt.h**). To activate a UART you will need to turn on the UART clock in the **SYSCTL\_RCGCUART\_R** register. You should also turn on the clock for the digital port in the **SYSCTL\_RCGCGPIO\_R** register. You need to enable the transmit and receive pins as digital signals. The alternative function for these pins must also be selected.

The **OE**, **BE**, **PE**, and **FE** are error flags associated with the receiver. You can see these flags in two places: associated with each data byte in **UART0\_DR\_R** or as a separate error register in **UART0\_RSR\_R**. The overrun error (**OE**) is set if data has been lost because the input driver latency is too long. **BE** is a break error, meaning the other device has sent a break. **PE** is a parity error (however, we will not be using parity). The framing error (**FE**) will get set if the baud rates do not match. The software can clear these four error flags by writing any value to **UART0\_RSR\_R**.

The status of the two FIFOs can be seen in the **UART0\_FR\_R** register. The **BUSY** flag is set while the transmitter still has unsent bits, even if the transmitter is disabled. It will become zero when the transmit FIFO is empty and the last stop bit has been sent. If you implement busy-wait output by first outputting then waiting for **BUSY** to become 0 (right flowchart of Figure 8.8), then the routine will write new data and return after that particular data has been completely transmitted.

The **UART0\_CTL\_R** control register contains the bits that turn on the UART. **TXE** is the Transmitter Enable bit, and **RXE** is the Receiver Enable bit. We set **TXE**, **RXE**, and **UARTEN** equal to 1 in order to activate the UART device. However, we should clear **UARTEN** during the initialization sequence.

	31–12	11	10	9	8	7–0			Name	
\$4000.C000		OE	BE	PE	FE	DATA			UART0_DR_R	
	31–3				3	2	1	0		
\$4000.C004				OE	BE	PE	FE		UART0_RSR_R	
	31–8	7	6	5	4	3	2–0			
\$4000.C018		TXFE	RXFF	TXFF	RXFE	BUSY				UART0_FR_R
	31–16	15–0								
\$4000.C024		DIVINT							UART0_IBRD_R	
	31–6			5–0						
\$4000.C028		DIVFRAC							UART0_FBRD_R	
	31–8	7	6 – 5	4	3	2	1	0		
\$4000.C02C		SPS	WPEN	FEN	STP2	EPS	PEN	BRK	UART0_LCRH_R	
	31–10	9	8	7	6–3	2	1	0		
\$4000.C030		RXE	TXE	LBE		SIRLP	SIREN	UARTEN	UART0_CTL_R	

**Table 8.3. Some UART registers. Each register is 32 bits wide. Shaded bits are zero.**

The **UART0\_IBRD\_R** and **UART0\_FBRD\_R** registers specify the baud rate. The baud rate **divider** is a 22-bit binary fixed-point value with a resolution of  $2^{-6}$ . The **Baud16** clock is created from the system bus clock, with a frequency of (Bus clock frequency)/**divider**. The baud rate is 16 times slower than **Baud16**

$$\text{Baud rate} = \text{Baud16}/16 = (\text{Bus clock frequency})/(16 * \text{divider})$$

For example, if the bus clock is 8 MHz and the desired baud rate is 19200 bits/sec, then the **divider** should be  $8,000,000/16/19200$  or 26.04167. As a binary fixed-point number, this number is about 11010.000011. We can establish this baud rate by putting the 11010 into **UART0\_IBRD\_R** and the 000011 into **UART0\_FBRD\_R**. In reality, 11010.000011 is equal to  $1667/64$  or 26.046875. The baud rates in the transmitter and receiver must match within 5% for the channel to operate properly. The error for this example is 0.02%.

The three registers **UART0\_LCRH\_R**, **UART0\_IBRD\_R**, and **UART0\_FBRD\_R** form an internal 30-bit register. This internal register is only updated when a write operation to **UART0\_LCRH\_R** is performed, so any changes to the baud-rate divisor must be followed by a write to the **UART0\_LCRH\_R** register for the changes to take effect. Out of reset, both FIFOs are disabled and act as 1-byte-deep holding registers. The FIFOs are enabled by setting the **FEN** bit in **UART0\_LCRH\_R**.

**Checkpoint 8.2:** Assume the bus clock is 10 MHz. What is the baud rate if **UART0\_IBRD\_R** equals 2 and **UART0\_FBRD\_R** equals 32?

**Checkpoint 8.3:** Assume the bus clock is 50 MHz. What values should you put in **UART0\_IBRD\_R** and **UART0\_FBRD\_R** to make a baud rate of 38400 bits/sec?

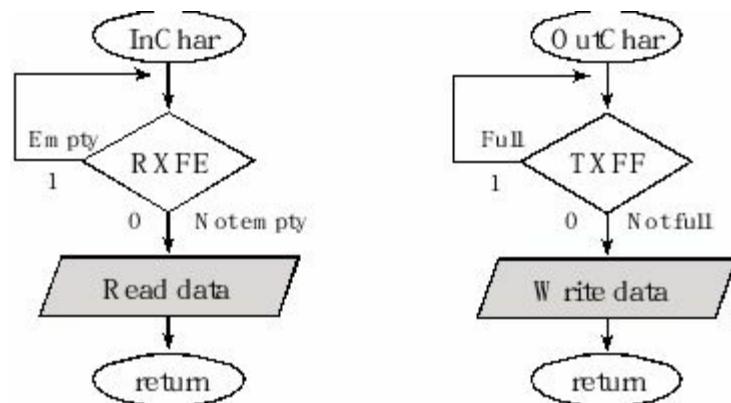


Figure 8.8. Flowcharts of InChar and OutChar using busy-wait synchronization.

### 8.2.3. UART Device Driver

Software that sends and receives data must implement a mechanism to synchronize the software with the hardware. In particular, the software should read data from the input device only when data is indeed ready. Similarly, software should write data to an output device only when the device is ready to accept new data. With busy-wait synchronization, the software continuously checks the hardware status waiting for it to be ready. In this section, we will use busy-wait synchronization to write I/O programs that send and receive data using the UART. After a frame is received, the receive FIFO will

be not empty (**RXFE** becomes 0) and the 8-bit data is available to be read. To get new data from the serial port, the software first waits for **RXFE** to be zero, then reads the result from **UART0\_DR\_R**. Recall that when the software reads **UART0\_DR\_R** it gets data from the receive FIFO. This operation is illustrated in Figure 8.8 and shown in Program 8.1. In a similar fashion, when the software wishes to output via the serial port, it first waits for **TXFF** to be clear, then performs the output. When the software writes **UART0\_DR\_R** it puts data into the transmit FIFO. An interrupt synchronization method will be presented in Chapter 11.

The initialization program, **UART\_Init**, enables the UART device and selects the baud rate. The **PCTL** bits were defined back in Tables 4.3, 4.4. **PCTL** bits 7-0 are set to 0x11 to select U0Tx and U0Rx on PA1 and PA0. The input routine waits in a loop until **RXFE** is 0 (FIFO not empty), then reads the data register. The output routine first waits in a loop until **TXFF** is 0 (FIFO not full), then writes data to the data register. Polling before writing data is an efficient way to perform output. **UART2\_xxx.zip** is the interrupt-driven version, which will be presented in Chapter 11.

```
// Assumes a 50 MHz bus clock, creates 115200 baud rate
void UART_Init(void){      // should be called only once
    SYSCTL_RCGCUART_R |= 0x0001; // activate UART0
    SYSCTL_RCGCGPIO_R |= 0x0001; // activate port A
    UART0_CTL_R &= ~0x0001;   // disable UART
    UART0_IBRD_R = 27; // IBRD=int(50000000/(16*115,200)) = int(27.1267)
    UART0_FBRD_R = 8; // FBRD = round(0.1267 * 64) = 8
    UART0_LCRH_R = 0x0070; // 8-bit word length, enable FIFO
    UART0_CTL_R = 0x0301; // enable RXE, TXE and UART
    GPIO_PORTA_PCTL_R = (GPIO_PORTA_PCTL_R&0xFFFF00)+0x00000011; // UART
    GPIO_PORTA_AMSEL_R &= ~0x03; // disable analog function on PA1-0
    GPIO_PORTA_AFSEL_R |= 0x03; // enable alt funct on PA1-0
    GPIO_PORTA_DEN_R |= 0x03; // enable digital I/O on PA1-0
}
// Wait for new input, then return ASCII code
char UART_InChar(void){
    while((UART0_FR_R&0x0010) != 0); // wait until RXFE is 0
    return((char)(UART0_DR_R&0xFF));
}
// Wait for buffer to be not full, then output
void UART_OutChar(char data){
    while((UART0_FR_R&0x0020) != 0); // wait until TXFF is 0
    UART0_DR_R = data;
}
```

Program 8.1. Device driver functions that implement serial I/O (**UART\_xxx.zip**).

**Checkpoint 8.4:** How does the software clear RXFE?

**Checkpoint 8.5:** How does the software clear TXFF?

**Checkpoint 8.6:** Describe what happens if the receiving computer is operating on a baud rate that is twice as fast as the transmitting computer?

**Checkpoint 8.7:** Describe what happens if the transmitting computer is operating on a baud rate that is twice as fast as the receiving computer?

**Checkpoint 8.8:** How do you change Program 8.1 to run at the same baud rate, but the system clock is now 10 MHz.

## 8.3. Synchronous Serial Interface, SSI

Microcontrollers employ multiple approaches to communicate synchronously with peripheral devices and other microcontrollers. The synchronous serial interface (SSI) system can operate as a master or as a slave. The channel can have one master and one slave, or it can have one master and multiple slaves. With multiple slaves, the configuration can be a star (centralized master connected to each slave), or a ring (each node has one receiver and one transmitter, where the nodes are connected in a circle.) The master initiates all data communication.

Stellaris® and Tiva® microcontrollers have 0 to 4 Synchronous Serial Interface or SSI modules. Another name for this protocol is Serial Peripheral Interface or SPI. The fundamental difference between a UART, which implements an asynchronous protocol, and a SSI, which implements a synchronous protocol, is the manner in which the clock is implemented. Two devices communicating with asynchronous serial interfaces (UART) operate at the same frequency (baud rate) but have two separate clocks. With a UART protocol, the clock signal is not included in the interface cable between devices. Two UART devices can communicate with each other as long as the two clocks have frequencies within  $\pm 5\%$  of each other. Two devices communicating with synchronous serial interfaces (SSI) operate from the same clock (synchronized). With a SSI protocol, the clock signal is included in the interface cable between devices. Typically, the master device creates the clock, and the slave device(s) uses the clock to latch the data (in or out.) The SSI protocol includes four I/O lines. The slave select SSI0Fss is an optional negative logic control signal from master to slave signal signifying the channel is active. The second line, SCK, is a 50% duty cycle clock generated by the master. The SSI0Tx (master out slave in, MOSI) is a data line driven by the master and received by the slave. The SSI0Rx (master in slave out, MISO) is a data line driven by the slave and received by the master. In order to work properly, the transmitting device uses one edge of the clock to change its output, and the receiving device uses the other edge to accept the data. Figure 8.9 shows the I/O port locations of the SSI ports discussed in this book, see Tables 4.3 and 4.4.

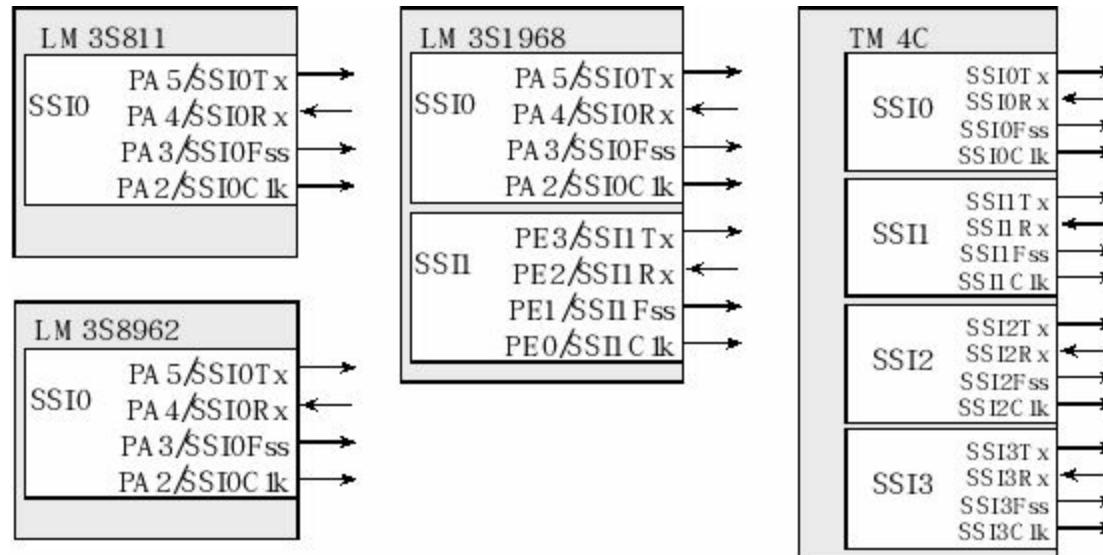


Figure 8.9. Synchronous serial port pins on Stellaris® LM3Sand Tiva® TM4C microcontrollers.

On the LM3S/TM4C the shift register can be configured from 4 to 16 bits. The shift register in the master and the shift register in the slave are linked to form a distributed register. Figure 8.10 illustrates communication between master and slave. Typically, the microcontroller and the I/O device slave are so physically close we do not use interface logic.

The interface is classified as synchronous because the hardware clock is shared between devices. The SSI on the LM3S/TM4C employs two hardware FIFOs. Both FIFOs are 8 elements deep and 4 to 16 bits wide, depending on the selected data width. When performing I/O the software puts into the transmit FIFO by writing to the **SSI0\_DR\_R** register and gets from the receive FIFO by reading from the **SSI0\_DR\_R** register.

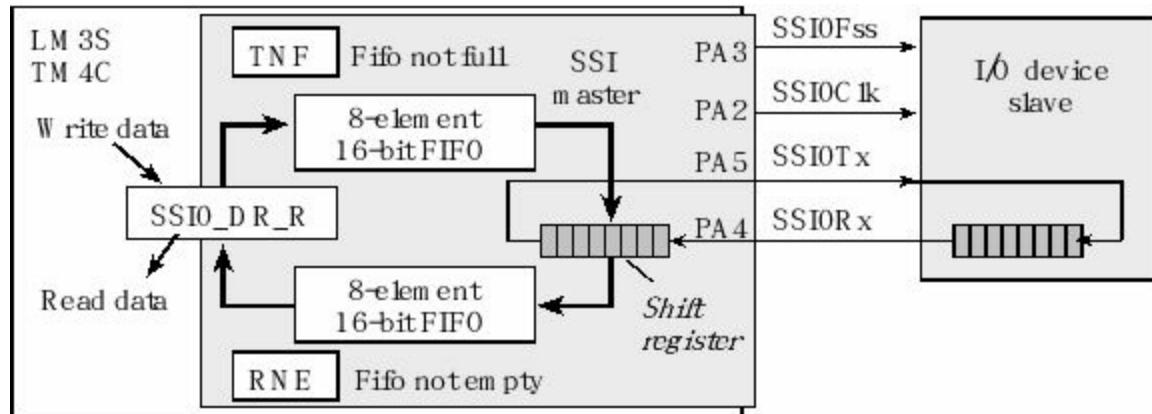


Figure 8.10. A synchronous serial interface between a microcontroller and an I/O device.

Table 8.4 lists the SSI0 registers on the TM4C. The TM4C can operate in slave mode, but we will focus on master mode. The **PCTL** bits are defined in Tables 4.3 and 4.4.

Address	31-6								3	2	1	0	Name
\$400F.E61C									SSI3	SSI2	SSI1	SSI0	SYSCTL_RCGCSSI_R
\$4000.8000	31-16	15-8		7	6	5-4	3-0						SSI0_CR0_R
\$4000.8008	31-16	15-0		SPH	SPO	FRF	DSS						SSI0_DR_R
\$4000.8004	Data								7	6	5	4	SSI0_CR1_R
\$4000.800C									SOD	MS	SSE	LBM	SSI0_SR_R
\$4000.8010	BSY								RFF	RNE	TNF	TFE	CPSDVS_R
\$4000.8014									TXIM	RXIM	RTIM	RORIM	SSI0_IM_R
\$4000.8018									TXRIS	RXRIS	RTRIS	RORRIS	SSI0_RIS_R
\$4000.801C									TXMIS	RXMIS	RTMIS	RORMIS	SSI0_MIS_R
\$4000.8020									RTIC	RORIC			SSI0_ICR_R
\$4000.4420	SEL	GPIO_PORTA_AFSEL_R											
\$4000.441C	DEN	GPIO_PORTA_DEN_R											
\$4000.4400	DIR	GPIO_PORTA_DIR_R											
\$400F.E608	GPIOH	GPIOG	GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	GPIOH	GPIOG	GPIOF	GPIOE	SYSCTL_RCGCGPIO_R

Table 8.4. The TM4C SSI0 registers. Each register is 32 bits wide. Bits 31 – 8 are zero.

If there is data in the transmit FIFO, the SSI module will transmit it. With SSI it transmits and receives bits at the same time. When a data transfer operation is performed, this distributed 8- to 32-bit register is serially shifted 4 to 16 bit positions by the SCK clock from the master so the data is effectively exchanged between the master and the slave. Data in the master shift register are transmitted to the slave. Data in the slave shift register are transmitted to the master. Typically, the microcontroller is master and the I/O module is the slave, but one can operate the microcontroller in slave mode. When designing with SSI, you will need to consult the data sheets for your specific microcontroller.

The SSI clock frequency is established by the 8-bit field **SCR**field in the **SSI0\_CR0\_R** register and the 8-bit field **CPSDVSR**field in the **SSI0\_CPSR\_R** register. **SCR** can be any 8-bit value from 0 to 255. **CPSDVSR** must be an even number from 2 to 254. Let  $f_{BUS}$  be the frequency of the bus clock. The frequency of the SSI is

$$f_{SSI} = f_{BUS} / (CPSDVSR * (1 + SCR))$$

Common control features for the SSI module include:

Baud rate control register, used to select the transmission rate

Data size 4 to 16 bits

Mode bits in the control register to select master versus slave

Freescale mode with clock polarity and clock phase

TI synchronous serial mode

Microwire mode

Interrupt arm bit

Ability to make the outputs open drain (open collector)

Common status bits for the SPI module include:

BSY, SSI is currently transmitting and/or receiving a frame, or the transmit FIFO is not empty

RFF, SSI receive FIFO is full

RNE, SSI receive FIFO is not empty

TNF, SSI transmit FIFO is not full

TFE, SSI transmit FIFO is empty

The key to proper transmission is to select one edge of the clock (shown as “T” in Figure 8.11) to be used by the transmitter to change the output, and use the other edge (shown as “R”) to latch the data in the receiver. In this way data is latched during the time when it is stable. Data available is the time when the output data is actually valid, and data required is the time when the input data must be valid.

The LM3S/TM4C output transmission is valid (S5) from 0 to 1 bus clock after the clock. I.e., the maximum S5 time is 1 system bus cycle and the minimum is 0. When receiving the setup time (S8) is 1 system bus cycle and the hold time (S9) is 2 system bus cycles. In order for the communication to occur without error, the data available from the device that is driving the data line must overlap (start before and end after) the data required by the other device that is receiving the data. It is this overlap that will determine the maximum frequency at which synchronous serial communication can occur. The concepts of data available and data required will be presented in much more detail in Volume 2.

**Checkpoint 8.9:** What are the definitions of setup time and hold time?

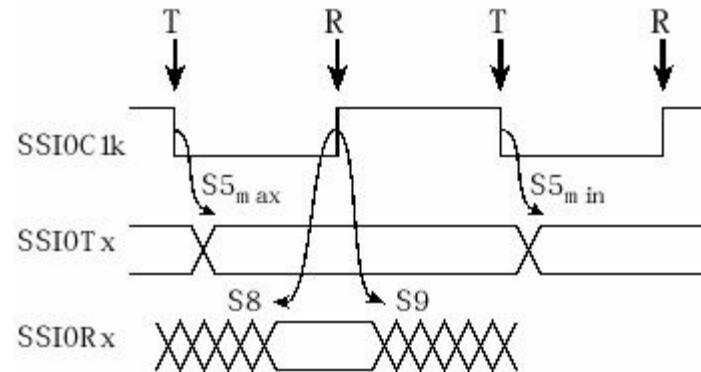


Figure 8.11. Synchronous serial timing showing the data available interval overlaps the data required interval.

**Observation:** Because the clocks are shared, if you change the bus clock frequency, the transfer rate will change in both master and slave.

The Freescale SPI timing (with **SPO=SPH=0**) is shown in Figure 8.12. The SPI transmits data at the same time as it receives input. In the Freescale modes, the SPI changes its output on the opposite edge of the clock as it uses to shift data in. There are three mode control bits (**MS**, **SPO**, **SPH**) that affect the transmission protocol. If the device is a master (**MS=0**), it generates the **SCLK**, and data is output on the **SSI0Tx** pin and input on the **SSI0Rx** pin. The **SPO** control bit specifies the polarity of the **SCLK**. In particular, the **SPO** bit specifies the logic level of the clock when data is not being transferred. The **SPH** bit affects the timing of the first bit transferred and received. If **SPH** is 0, then the device will shift data in on the first (and 3<sup>rd</sup>, 5<sup>th</sup>, 7<sup>th</sup>, ... etc.) clock edge. If **SPH** is 1, then the device will shift data in on the second (and 4<sup>th</sup>, 6<sup>th</sup>, 8<sup>th</sup>, ... etc.) clock edge. The data is transmitted MSB first. There are SSI examples on the book web site (MAX5353\_xxx.zip, SDC\_xxx.zip). Furthermore, most Stellaris® evaluation boards utilize the SSI to interface the organic light emitting diode display. So most of the OLED example projects also include example SSI code.

There is also a TI synchronous serial mode and another protocol called Microwire. Refer to the data sheets for details of these modes.

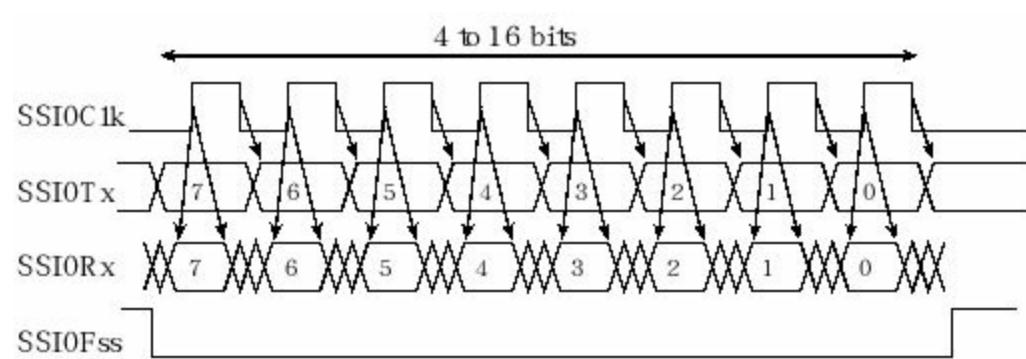


Figure 8.12. Synchronous serial Freescale single transfer mode (SPO=0, SPH=0).

Analog Devices, Maxim, and Texas Instruments will send free samples of DACs and ADCs to students, so these SPI interface examples are inexpensive to build. The data sheets of most devices will assist you when interfacing it to your microcontroller. I do suggest you get plastic dual in-line packages (PDIP), so you can plug the parts into a protoboard. Also you should consider voltage levels, making sure the chips operate on whatever voltage supply you have on your system.

<http://www.analog.com/en/index.html>

<http://www.maxim-ic.com/>

<http://www.ti.com>

## 8.4. Nokia 5110 Graphics LCD Interface

In this section we will interface a Nokia 5110 LCD using busy-wait synchronization. See Figure 8.13. Before we output data or commands to the display, we will check a status flag and wait for the previous operation to complete. Busy-wait synchronization is very simple and is appropriate for I/O devices that are fast and predictable.

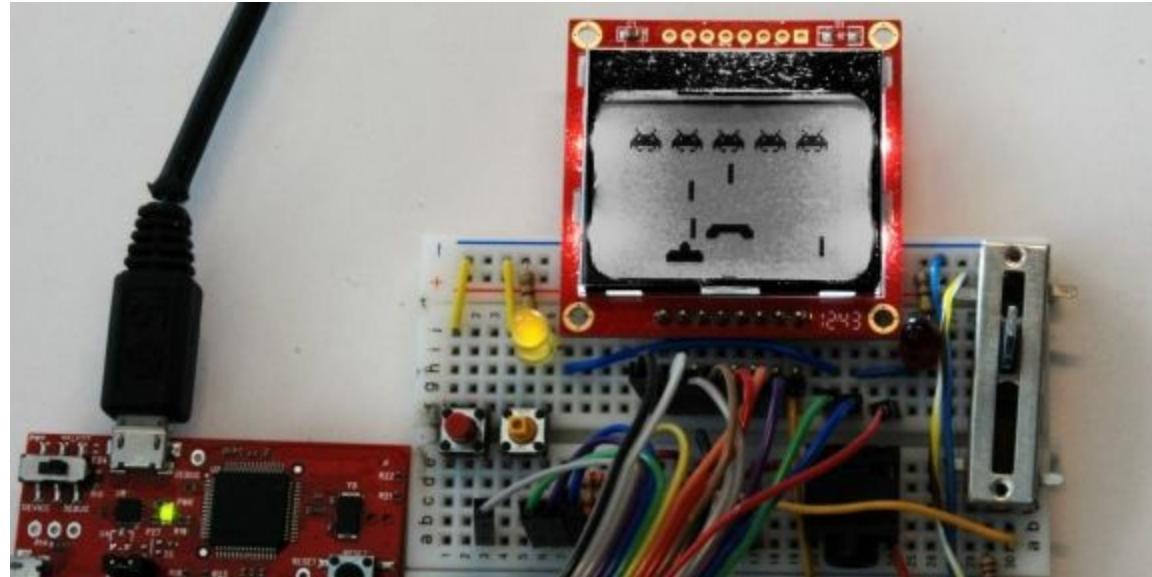


Figure 8.13. Nokia 5110 display with 84 by 48 monochrome pixels.

The Nokia 5110 uses the synchronous serial interface (SSI) described in the last section to control PA5 (MOSI), PA3 (Fss), and PA2 (Sclk), as shown in Figure 8.14. Pins PA7 and PA6 are regular GPIO pins. The microcontroller will be master and the LCD slave. There are multiple Nokia 5110 displays for sale on the market with the same LCD but different pin locations for the signals. Figure 8.14 shows two of the possible pin configurations. Please look on your actual display for the pin name and not the pin number. Be careful when connecting the back light, because at 3.3V the back light draws 80 mA. If you want a dimmer back light connect 3.3V to a 100 ohm resistor, and the other end of the resistor to the LED/BL pin.

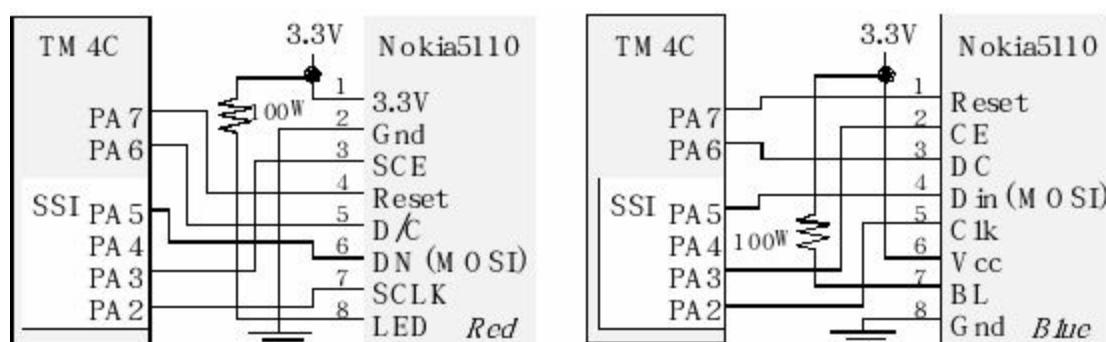


Figure 8.14. Nokia 5110 interface to a TM4C123.

Program 8.2 lists the prototypes for public functions available in the software starter project. The **Init** function must be called once, before any of the other functions can be called. The **SetCursor** function, defines where on the screen subsequent character output will occur. Each ASCII character is 7 pixels

wide and 8 pixels high. This means there can be  $84/7 = 12$  characters by  $48/8 = 6$  rows. The cursor is defined by character position, not pixel location, so  $0 \leq \text{newX} \leq 11$  and  $0 \leq \text{newY} \leq 5$ , with 0,0 being the top row on left. The **Clear** function erases the entire screen. It takes 4,032 bits, or 504 bytes, to represent an entire 84 by 48 pixel image. The **DrawFullImage** function takes a 504-byte array and copies it onto the display. Additional graphics functions were presented previously in Section 6.7.

```
void Nokia5110_Init(void);
void Nokia5110_SetCursor(uint8_t newX, uint8_t newY);
void Nokia5110_Clear(void);
void Nokia5110_DrawFullImage(const uint8_t *ptr);
void Nokia5110_OutChar(char data);
void Nokia5110_OutString(char *ptr);
void Nokia5110_OutUDec(unsigned uint16_t n);
```

Program 8.2. Software prototypes for Nokia 5110 display (Nokia5110xxx.zip).

0	1	0	0	0	1	0
0	1	0	0	0	1	0
0	1	1	1	1	1	0
0	1	0	1	0	1	0
0	1	0	1	0	1	0
0	0	1	1	1	0	0
0	0	0	1	0	0	0
00	1f	24	7c	24	1f	00

The **OutChar** **OutString** and **OutUDec** functions draw ASCII characters on the screen. These three functions maintain a cursor so you can call these three functions in any order. The matrix **ASCII[][][5]** contains the pixel image for each character. Notice the 5 by 8 image for ASCII 0x7F is the University of Texas UT symbol. There is one blank line before the 5 by 8 character and one blank line after making each character 7 wide by 8 pixels tall, see Program 8.3.

```
static const uint8_t ASCII[][][5] = {
    {0x00, 0x00, 0x00, 0x00, 0x00} // 20, space
    ,{0x00, 0x00, 0x5f, 0x00, 0x00} // 21 !
...
    ,{0x1f, 0x24, 0x7c, 0x24, 0x1f} // 7f UT sign
};

#define PA6 (*((volatile uint32_t *)0x40004100))
void static lcddatawrite(uint8_t data){
    while((SSI0_SR_R&0x02)==0){} // busy-wait on TNF
    PA6 = 0x20;                // DC is data
    SSI0_DR_R = data;          // data out
}
void Nokia5110_OutChar(char data){int i;
    lcddatawrite(0x00);        // blank vertical line padding
```

```
for(i=0; i<5; i=i+1){ // 5 by 8 image
    lcddatawrite(ASCII[data - 0x20][i]);
}
lcddatawrite(0x00); // blank vertical line padding
}
```

Program 8.3. Low-level functions to output characters on the Nokia 5110 display (Nokia5110xxx.zip).

## 8.5. Scanned Keyboards

In a **scanned interface**, the switches are placed in a row/column matrix. In this way, many keys can be interfaced with just a few I/O pins. Figure 8.15 shows a matrix keyboard with 4 rows and 4 columns. In general, if there are **n** rows and **m** columns, there could be **n\*m** switches, but we would need only **n+m** I/O pins. The **at the four outputs signifies open collector** (an output with two states HiZ and low.)

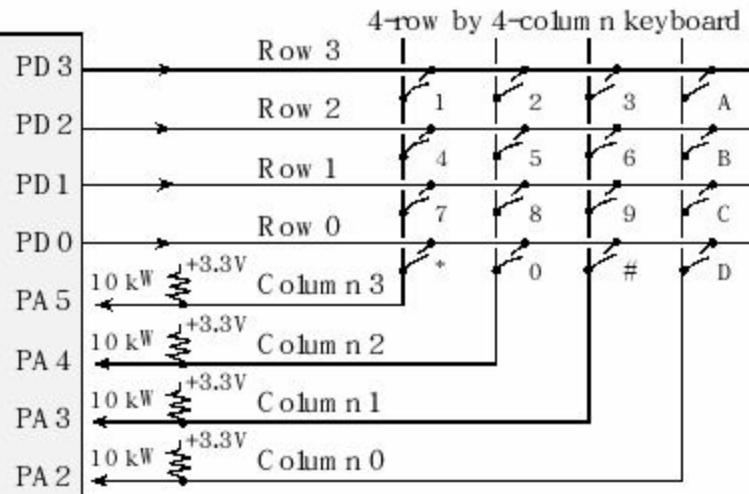


Figure 8.15. A matrix keyboard interfaced to the microcomputer.

The computer drives one row at a time to zero, while leaving the other rows at HiZ. By reading the column, the software can detect if a key is pressed in that row. The software “scans” the device by checking all rows one by one. For most microcontrollers, the open collector functionality can be implemented by toggling the direction register. Remember, open collectors have two states low and off. The output low state can be made by making the pin an output and setting the output data to zero. The output off state can be made by making the pin an input.

# 8.6. Binary actuators

## 8.6.1. Interface

Relays, solenoids, and DC motors are grouped together because their electrical interfaces are similar. We can add speakers to this group if the sound is generated with a square wave. In each case, there is a coil, and the computer must drive (or not drive) current through the coil. To interface a coil, we consider **voltage**, **current**, and **inductance**. We need a power supply at the desired voltage requirement of the coil. If the only available power supply is larger than the desired coil voltage, we use a voltage regulator (rather than a resistor divider) to create the desired voltage. We connect the power supply to the positive terminal of the coil, shown as **+V** in Figure 8.16. We will use a transistor device to drive the negative side of the coil to ground. The computer can turn the current on and off using this transistor. The second consideration is current. In particular, we must however select the power supply and an interface device that can support the coil current. The 7406 is a digital inverter with open collector outputs (HiZ and low). The 2N2222 is a bipolar junction transistor (BJT), NPN type, with moderate current gain. The TIP120 is a Darlington transistor, also NPN type, that can handle larger currents. The IRF540 is a MOSFET transistor that can handle even more current. BJT and Darlington transistors are current-controlled (meaning the output is a function of the input current), while the MOSFET is voltage-controlled (output is a function of input voltage). When interfacing a coil to the microcontroller, we use information like Table 8.5 to select an interface device capable of sinking the current necessary to activate the coil. It is a good design practice to select a driver with a maximum current at least twice the required coil current. When the digital **Port** output is high, the interface transistor is active, and current flows through the coil. When the digital **Port** output is low, the transistor is not active, and no current flows through the coil.

Device	Type	Maximum current
TM4C123	CMOS	8 mA
TM4C1294	CMOS	12 mA
7406/7407	TTL logic	40 mA
2N2222	BJT NPN	500 mA
TIP120	Darlington NPN	5 A
IRF540	power MOSFET	28 A

**Table 8.5. Four possible devices that can be used to interface a coil compared to the LM3S/TM4C.**

Similar to the solenoid and EM relay, the DC motor has a frame that remains motionless, and an armature that moves. In this case, the armature moves in a circular manner (shaft rotation). A DC motor has an electro-magnet as well. When current flows through the coil, a magnetic force is created causing a rotation of the shaft. Brushes positioned between the frame and armature are used to

alternate the current direction through the coil, so that a DC current generates a continuous rotation of the shaft. When the current is removed, the magnetic force stops, and the shaft is free to rotate. The resistance in the coil ( $R$ ) comes from the long wire that goes from the + terminal to the - terminal of the motor. The inductance in the coil ( $L$ ) arises from the fact that the wire is wound into coils to create the electromagnetics. The coil itself can generate its own voltage (emf) because of the interaction between the electric and magnetic fields. If the coil is a DC motor, then the emf is a function of both the speed of the motor and the developed torque (which in turn is a function of the applied load on the motor.) Because of the internal emf of the coil, the current will depend on the mechanical load. For example, a DC motor running with no load might draw 50 mA, but under load (friction) the current may jump to 500 mA.

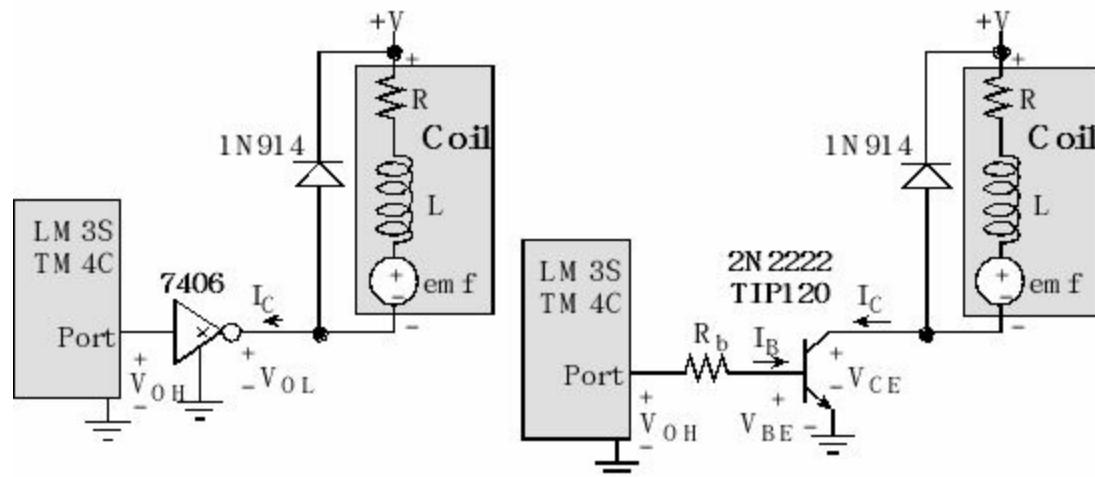


Figure 8.16. Binary interface to EM relay, solenoid, DC motor, or speaker.

**Observation:** It is important to realize that many devices cannot be connected directly up to the microcontroller. In the specific case of motors, we need an interface that can handle the voltage and current required by the motor.

The third consideration is inductance in the coil. The 1N914 diode in Figure 8.16 provides protection from the back emf generated when the switch is turned off, and the large  $dI/dt$  across the inductor induces a large voltage (on the negative terminal of the coil), according to  $V=L \cdot dI/dt$ . For example, if you are driving 0.1 A through a 0.1 mH coil (**Port** output = 1) using a 2N2222, then disable the driver (**Port** output = 0), the 2N2222 will turn off in about 20ns. This creates a  $dI/dt$  of at least  $5 \cdot 10^6$  A/s, producing a back emf of 500 V! The 1N914 diode shorts out this voltage, protecting the electronic from potential damage. The 1N914 is called a **snubber diode**.

**Example 8.1.** Design an interface for two +12V 1A geared DC motors. These two motors will be used to propel a robot with two independent drive wheels, shown back in Figure 8.1.

**Solution:** We will use two copies of the TIP120 circuit in Figure 8.16 because the TIP120 can sink at least three times the current needed for this motor. We select a +12V supply and connect it to the +V in the circuit. The needed base current is

$$I_b = I_{coil} / h_{fe} = 1A / 1000 = 1mA$$

The desired interface resistor.

$$R_b \leq (V_{OH} - V_{be}) / I_b = (5 - 2.5) / 1mA = 2.5 k\Omega$$

To cover the variability in  $h_{fe}$ , we will use a 1 kΩ resistor instead of the 2.5 kΩ. The actual voltage on the motor when active will be  $+12 - 2 = 10$ V. The coils and transistors can vary a lot, so it is appropriate to experimentally verify the design by measuring the voltages and currents.

---

## 8.6.2. Electromagnetic and Solid State Relays

A relay is a device that responds to a small current or voltage change by activating switches or other devices in an electric circuit. It is used to remotely switch signals or power. The input control is usually electrically isolated from the output switch. The input signal determines whether the output switch is open or closed. Relays are classified into three categories depending upon whether the output switches power (i.e., high currents through the switch) or electronic signals (i.e., low currents through the switch). Another difference is how the relay implements the switch. An electromagnetic (EM) relay uses a coil to apply EM force to a contact switch that physically opens and closes. The solid state relay (SSR) uses transistor switches made from solid state components to electronically allow or prevent current flow across the switch. The three types are

1. The classic general purpose relay has an EM coil and can switch AC power
2. The reed relay has an EM coil and can switch low level DC electronic signals
3. The solid state relay (SSR) has an input triggered semiconductor power switch

Three solid state relays are shown in Figure 8.17. Interfacing a SSR is identical to interfacing an LED, which was previously described in Figure 2.9. A SSR interface was presented earlier as Figure 4.17. SSRs allow the microcontroller to switch AC loads from 1 to 30A. They are appropriate in situations where the power is turned on and off many times.



Figure 8.17. Solid state relays can be used to control power to an AC appliance.

The input circuit of an EM relay is a coil with an iron core. The output switch includes two sets of silver or silver-alloy contacts (called **poles**.) One set is fixed to the relay **frame**, and the other set is located at the end of leaf spring poles connected to the **armature**. The contacts are held in the “normally closed” position by the armature return spring. When the input circuit energizes the EM coil, a “pull in” force is applied to the armature and the “normally closed” contacts are released (called **break**) and the “normally open” contacts are connected (called **make**.) The armature pull in can either energize or de-energize the output circuit depending on how it is wired. Relays are mounted in special sockets, or directly soldered onto a PC board.

The number of poles (e.g., single pole, double pole, 3P, 4P etc.) refers to the number of switches that are controlled by the input. The relay shown below is a double pole because it has two switches.

**Single throw** means each switch has two contacts that can be open or closed. **Double throw** means each switch has three contacts. The common contact will be connected to one of the other two contacts (but not both at the same time.) The parameters of the output switch include maximum AC (or DC) power, maximum current, maximum voltage, on resistance, and off resistance. A DC signal will weld the contacts together at a lower current value than an AC signal, therefore the maximum ratings for DC are considerably smaller than for AC. Other relay parameters include turn on time, turn off time, life expectancy, and input/output isolation. **Life expectancy** is measured in number of operations. Figure 8.18 illustrates the various configurations available. The sequence of operation is described in Table 8.6.

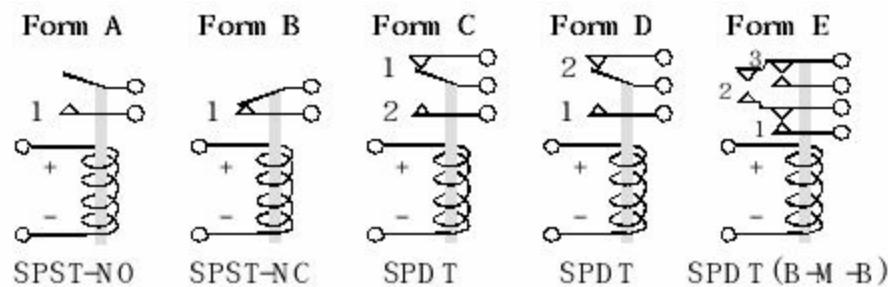


Figure 8.18. Standard relay configurations.

Form	Activation Sequence	Deactivation Sequence
A	Make 1	Break 1
B	Break 1	Make 1
C	Break 1, Make 2	Break 2, Make 1
D	Make 1, Break 2	Make 2, Break 1
E	Break 1, Make 2, Break 3	

Table 8.6. Standard definitions for five relay configurations.

### 8.6.3. Solenoids

Solenoids are used in discrete mechanical control situations such as door locks, automatic disk/tape ejectors, and liquid/gas flow control valves (on/off type). Much like an EM relay, there is a frame that remains motionless, and an armature that moves in a discrete fashion (on/off). A solenoid has an electro-magnet. When current flows through the coil, a magnetic force is created causing a discrete motion of the armature. Each of the solenoids shown Figure 8.19 has a cylindrically-shaped armature that moves in the horizontal direction relative to the photograph. The solenoid on the top is used in a door lock, and the second from top is used to eject the tape from a video cassette player. When the current is removed, the magnetic force stops, and the armature is free to move. The motion in the opposite direction can be produced by a spring, gravity, or by a second solenoid.

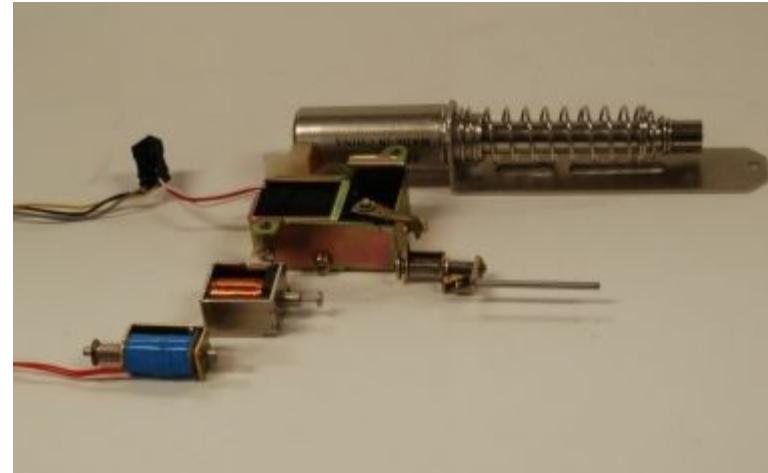


Figure 8.19. Photo of four solenoids.

## 8.7. \*Pulse-width modulation

In the previous interfaces the microcontroller was able to control electrical power to a device in a binary fashion: either all on or all off. Sometimes it is desirable for the microcontroller to be able to vary the delivered power in a variable manner. One effective way to do this is to use pulse width modulation (PWM). The basic idea of PWM is to create a digital output wave of fixed frequency, but allow the microcontroller to vary its duty cycle. Figure 8.20 shows various waveforms that are high for **H** cycles and low for **L** cycles. The system is designed in such a way that **H+L** is constant (meaning the frequency is fixed). The duty cycle is defined as the fraction of time the signal is high:

$$\text{duty} = \frac{H}{H+L}$$

Hence, duty cycle varies from 0 to 1. We interface this digital output wave to an external actuator (like a DC motor), such that power is applied to the motor when the signal is high, and no power is applied when the signal is low. We purposely select a frequency high enough so the DC motor does not start/stop with each individual pulse, but rather responds to the overall average value of the wave. The average value of a PWM signal is linearly related to its duty cycle and is independent of its frequency. Let **P** ( $P=V*I$ ) be the power to the DC motor, shown in Figure 8.20, when the PP0 signal is high. Notice the circuit in Figure 8.20 is one of the examples previously described in Figure 8.16. Under conditions of constant speed and constant load, the delivered power to the motor is linearly related to duty cycle.

$$\text{duty} * P = \frac{H}{H+L} * P$$

**Delivered Power =**

Unfortunately, as speed and torque vary, the developed emf will affect delivered power. Nevertheless, PWM is a very effective mechanism, allowing the microcontroller to adjust delivered power. See PWMxxx.zip examples.

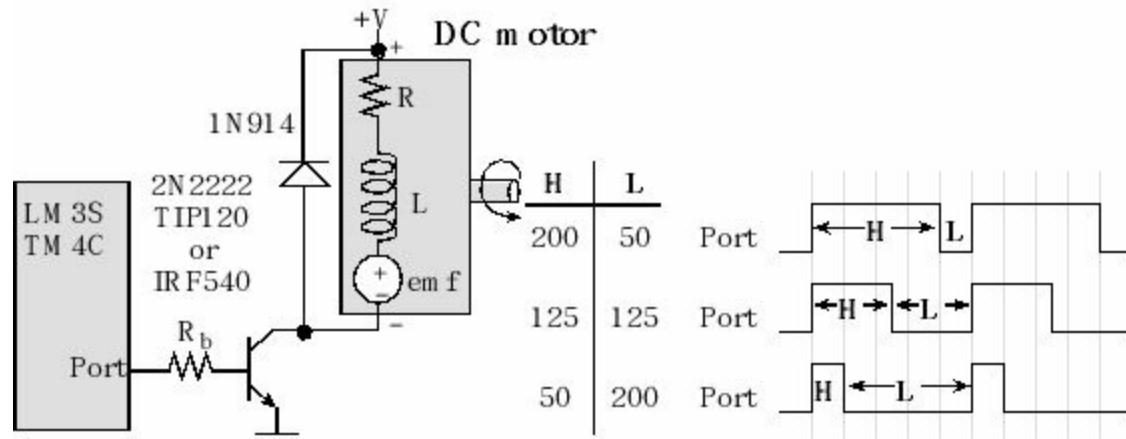


Figure 8.20. Pulse width modulation used to vary power delivered to a DC motor.

## 8.8. \*Stepper motors

A motor can be evaluated in terms of its maximum speed (RPM), its torque (N-m), and the efficiency in which it translates electrical power into mechanical power. Sometimes however, we wish to use a motor to control the rotational position ( $\theta$ =motor shaft angle) rather than to control the rotational speed ( $\omega=d\theta/dt$ ). Stepper motors are used in applications where precise positioning is more important than high RPM, high torque, or high efficiency. Stepper motors are very popular for microcontroller-based embedded systems because of their inherent digital interface. Figure 8.21 shows three stepper motors.



Figure 8.21. Photo of three stepper motors.

The larger motors provide more torque, but require more current. It is easy for a computer to control both the position and velocity of a stepper motor in an open-loop fashion. Although the cost of a stepper motor is typically higher than an equivalent DC permanent magnetic field motor, the overall system cost is reduced because stepper motors may not require feedback sensors. They are used in printers to move paper and print heads, tapes/disks to position read/write heads, and high-precision robots. For example, the stepper motor shown previous in Figure 6.3 moves the R/W head from one track to another on an audio tape recorder.

A bipolar stepper motor has two coils on the stator (the frame of the motor), labeled **A** and **B** in Figures 8.22 and 8.23. Typically, there is always current flowing through both coils. When current flows through both coils, the motor does not spin (it remains locked at that shaft angle). Stepper motors are rated in their holding torque, which is their ability to hold stationary against a rotational force (torque) when current is constantly flowing through both coils. To move a bipolar stepper, we reverse the direction of current through one (not both) of the coils, see Figure 8.20. To move it again we reverse the direction of current in the other coil. Remember, current is always flowing through both coils. Let the direction of the current be signified by up and down. To make the current go up, the microcontroller outputs a binary 01 to the interface. To make the current go down, it outputs a binary 10. Since there are 2 coils, four outputs will be required (e.g.,  $0101_2$  means up/up). To spin the motor, we output the sequence  $0101_2$ ,  $0110_2$ ,  $1010_2$ ,  $1001_2$ ... over and over. Each output causes the motor to rotate a fixed angle. To rotate the other direction, we reverse the sequence ( $0101_2$ ,  $1001_2$ ,  $1010_2$ ,  $0110_2$ ...). There is a North and a South permanent magnet on the rotor (the part that spins). The amount of rotation caused by each current reversal is a fixed angle depending on the number of teeth on the permanent magnets.

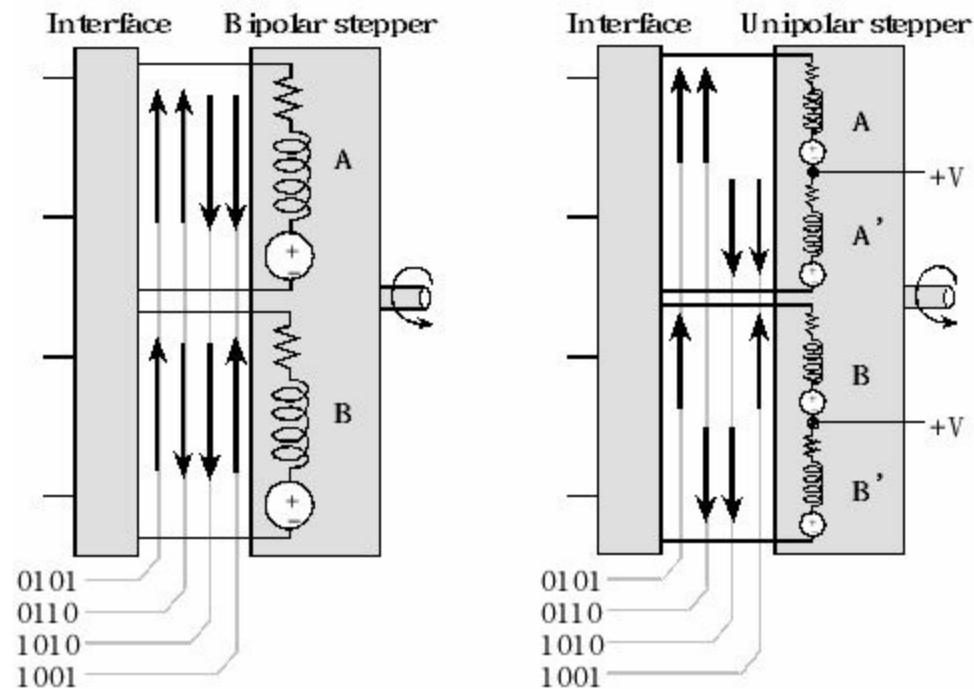


Figure 8.22. A bipolar stepper has 2 coils, but a unipolar stepper divides those two coils into four parts.

For example, the rotor in Figure 8.23 is drawn with one North tooth and one South tooth. If there are  $n$  teeth on the South magnet (also  $n$  teeth on the North magnet), then the stepper will move at  $90/n$  degrees. This means there will be  $4n$  steps per rotation. Because moving the motor involves accelerating a mass (rotational inertia) against a load friction, after we output a value, we must wait an amount of time before we can output again. If we output too fast, the motor does not have time to respond. The speed of the motor is related to the number of steps per rotation and the time in between outputs. For information on stepper motors see the data sheets web page at <http://users.ece.utexas.edu/~valvano/Datasheets/>

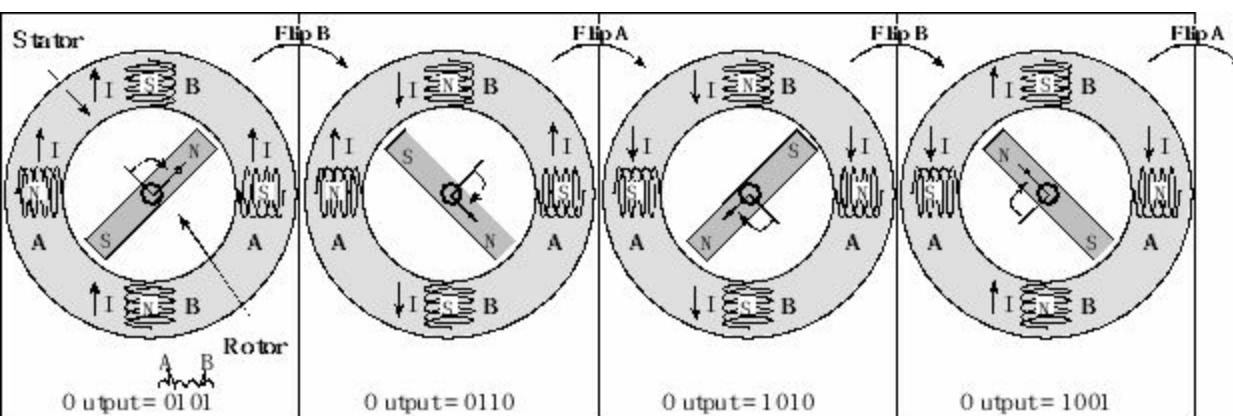


Figure 8.23. To rotate this stepper by  $18^\circ$ , the interface flips the direction of one of the currents.

The unipolar stepper motor provides for bi-directional currents by using a center tap, dividing each coil into two parts. In particular, coil **A** is split into coil **A** and **A'**, and coil **B** is split into coil **B** and **B'**. The center tap is connected to the  $+V$  power source and the four ends of the coils can be controlled with open collector drivers. Because only half of the electro-magnets are energized at one time, a unipolar stepper has less torque than an equivalent-sized bipolar stepper. However, unipolar steppers are easier to interface. For example, you can use four copies of the circuit in Figure 8.16 to interface a stepper motor.

Figure 8.24 shows a circular linked graph containing the output commands to control a stepper motor. This simple FSM has no inputs, four output bits and four states. There is one state for each output pattern in the usual stepper sequence 5,6,10,9... The circular FSM is used to spin the motor in a clockwise direction. Notice the 1-to-1 correspondence between the state graph in Figure 8.24 and the **fsm[4]** data structure in Program 8.3.

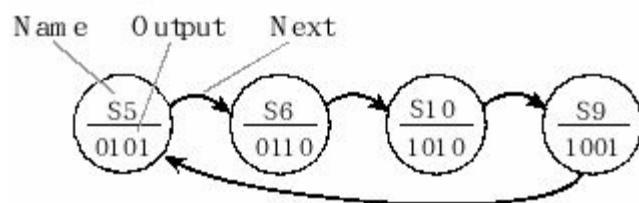


Figure 8.24. This stepper motor FSM has four states. The 4-bit outputs are given in binary.

---

**Example 8.2.** Design a stepper motor controller than spins the motor at 6 RPM.

**Solution:** We choose a stepper motor according to the speed and torque requirements of the system. A stepper with 200 steps/rotation will provide a very smooth rotation while it spins. Just like the DC motor, we need an interface that can handle the currents required by the coils. We can use a L293 to interface either unipolar or bipolar steppers that require less than 1 A per coil. In general, the output current of a driver must be large enough to energize the stepper coils. We control the interface using an output port of the microcontroller, as shown in Figure 8.25. The circuit shows the interface of a unipolar stepper, but the bipolar stepper interface is exactly the same except there is no  $+V$  connection to the motor.

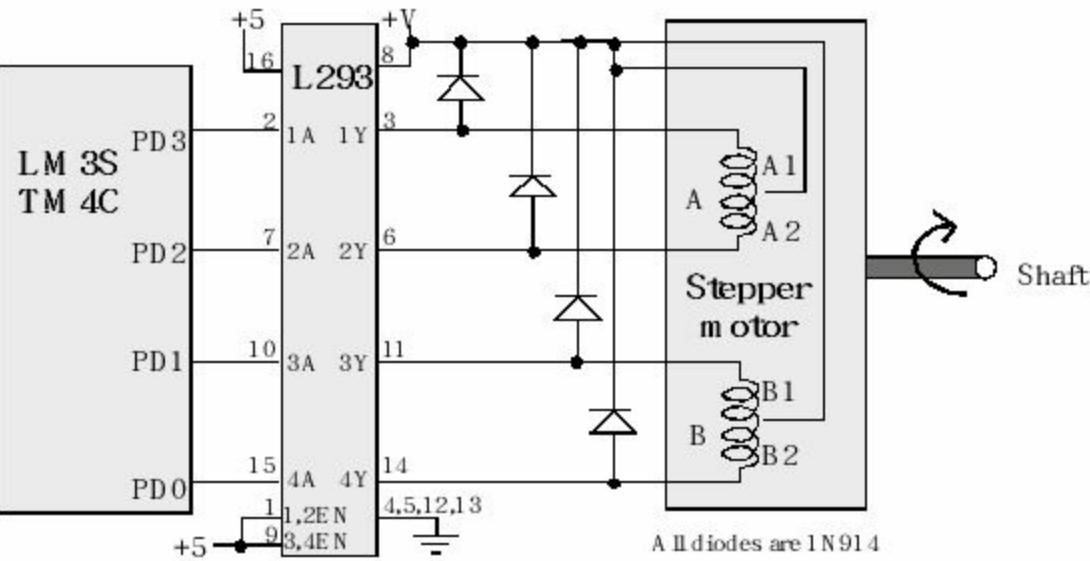


Figure 8.25. A unipolar stepper motor interfaced to a microcontroller.

The main program, Program 8.3, begins by initializing the Port D output and the state pointer, **pt**. Every 50 ms the program outputs a new stepper command using the bit-specific address. The function **SysTick\_Wait10ms()** from Program 4.7 generates an appropriate delay between outputs to the stepper. For a 200 step/rotation stepper, we need to wait 50 ms between outputs to spin at 6 RPM. When calculating speed, it is important to keep track of the units.

$$\text{Speed} = (1 \text{ rotation}/200 \text{ steps}) * (1000 \text{ ms/s}) * (60 \text{ sec/min}) * (1 \text{ step}/\mathbf{50 \text{ ms}}) = 6 \text{ RPM}$$

This is a FSM with no input, because it always spins at 6 rps.

```

struct State{
    uint8_t Out;
    const struct State *Next;
    uint32_t Delay;
};
typedef const struct State StateType;
typedef StateType *StatePtr;
StateType fsm[4] = {
    {10,&fsm[1],5},
    {9,&fsm[2],5},
    {5,&fsm[3],5},
    {6,&fsm[0],5}
};
const struct State *Pt; // Current State
#define STEPPER (*((volatile uint32_t *)0x4000703C))
int main(void){
    PLL_Init(); // Program 4.6
    SysTick_Init(); // Program 4.7
    SYSCTL_RCGCGPIO_R |= 0x08; // 1) port D clock enabled
    Pt = &fsm[0];

```

```

// 2) no need to unlock PD3-0
GPIO_PORTD_AMSEL_R &= ~0x0F; // 3) disable analog function
GPIO_PORTD_PCTL_R &= ~0x0000FFFF; // 4) GPIO
GPIO_PORTD_DIR_R |= 0x0F; // 5) make PD3-0 out
GPIO_PORTD_AFSEL_R &= ~0x0F; // 6) disable alt func on PD3-0
GPIO_PORTD_DR8R_R |= 0x0F; // enable 8 mA drive on PD3-0
GPIO_PORTD_DEN_R |= 0x0F; // 7) enable digital I/O on PD3-0
while(1){
    STEPPER = Pt->Out; // step motor
    SysTick_Wait10ms(Pt->Delay);
    Pt = Pt->Next; // circular linked list
}

```

### Program 8.3. Stepper motor controller.

---

To illustrate how easy it is to make changes to this implementation, let's consider these three modifications. To make it spin in the other direction, we simply change pointers to sequence in the other direction. We could also add an input pin and have it spin clockwise or counterclockwise depending on the input. Each would have two next states depending on the input. To make it spin at a different rate, we change the wait time. To implement an eight-step sequence (the half-stepping outputs are 5, 4, 6, 2, 10, 8, 9, 1...), we add the four new states and link all eight states in the desired sequence.

**Checkpoint 8.10:** If the stepper motor were to have 36 steps per rotation, how fast would the motor spin using Program 8.3?

**Checkpoint 8.11:** What would you change in Program 8.3 to make the motor spin at 30 RPM?

**Performance tip:** Use a DC motor for applications requiring high torque or high speed, and use a stepper motor for applications requiring accurate positioning at low speed.

**Performance tip:** To get high torque at low speed, use a geared DC motor (the motor spins at high speed, but the shaft spins slowly).

## 8.9. Exercises

- 8.1** Assume the baud rate is 9600 bits/sec. Show the serial port output versus time waveform that occurs when the ASCII characters “ABC” are transmitted one right after another. What is the total time to transmit the three characters?
- 8.2** Assume the baud rate is 19200 bits/sec. Show the serial port output versus time waveform that occurs when the ASCII characters “125” are transmitted one right after another. What is the total time to transmit the three characters?
- 8.3** Assume the bus clock is 50 MHz. Write an assembly language subroutine that initializes the serial port to communicate at 9600 bits/sec, 8-bit data, 1 start bit, and 1 stop bit.
- 8.4** Considering the voltages shown in Figure 8.2, prove that you can connect an LM3S/TM4C output to a 7404 input. Similarly, prove that you can connect a 7404 output to an LM3S/TM4C input. Which logic family types shown in Figure 8.2 allow the output of the digital gate to be connected to an LM3S/TM4C input?
- D8.5** Design an interface for a 64-key keyboard, which is configured with 8 rows and 8 columns. Show the hardware interface to the microcontroller. Show the initialization ritual. Assume there is either no key or one key pressed. Write an input subroutine that returns the key number 0 to 63 if a key is pressed or -1 if no key is pressed. Assume the keys do not bounce.
- D8.6** Design an interface for a 20-key keyboard, which is configured with 4 rows and 5 columns. Show the hardware interface to the microcontroller. Show the initialization ritual. Assume there is either no key or one key pressed. Write an input subroutine that returns the key number 0 to 19 if a key is pressed or -1 if no key is pressed. Assume the keys do not bounce.
- D8.7** Interface an electromagnetic relay (2 wires) to the microcontroller. The coil requires 250 mA at 5V. Write a ritual to initialize the interface. Write a subroutine, called **On**, that activates the relay, and a subroutine, called **Off**, that deactivates the relay.
- D8.8** Interface a solenoid (2 wires) to the microcontroller. The coil requires 100 mA at 12V. Write a ritual to initialize the interface. Write a subroutine, called **Pulse**, that activates the solenoid for 10 ms (then shuts off). No interrupts needed, use **SysTick\_Wait**.
- D8.9** Interface a DC motor (2 wires) to the microcontroller. The coil requires 500 mA at 12V. In addition to the motor output, there are two inputs. When the Go input is high the motor spins, (when Go is low, no power is delivered). When the motor is spinning, the other input (Direction) determines the CCW/CW rotational direction. Use a L293 H-bridge driver.

**D8.10** There is a microcontroller digital output connected to a microcontroller digital input across a long cable. The connection has an equivalent capacitance of 25 pF into a 10 M $\Omega$  resistance. The capacitance results from the long cable, and the resistance results from the input impedance of the receiver microcontroller. What is the time constant of this system? If we operate 10 times slower than the time constant, what is the maximum period allowed for this system? List two ways to speed up this transmission.

**D8.11** Interface an 8-bit DAC, MAX549 to the LM3S/TM4C SSI port. Write two functions, one to initialize and one to update both DAC analog outputs.

**D8.12** Interface a unipolar stepper motor (5 wires) to the microcontroller. Each coil requires 500 mA at 12V. There are 200 steps per revolution. There is also a switch input, if the input is low the motor spins CW at 5 rps. If the input is high the motor spins CCW at 10 rps. Use **SysTick\_Wait** and a FSM.

**D8.13** Interface a unipolar stepper motor (5 wires) to the microcontroller. Each coil requires 100 mA at 6V. There are 36 steps per revolution. There is also a switch input, if the input is low the motor stops. If the input is high the motor spins at 10 rps. Use **SysTick\_Wait** and a FSM.

**D8.14** Interface a bipolar stepper motor (4 wires) to the microcontroller. Each coil requires 500 mA at 12V. There are 200 steps per revolution. There is also a switch input, if the input is low the motor stops. If the input is high the motor spins at 5 rps. Use **SysTick\_Wait** and a FSM.

---

## 8.10. Lab Assignments

**Lab 8.1** Keyboard Device Driver. Interface a matrix keyboard and design a software device driver to support the functionality of the keyboard.

**Lab 8.2** Stepper motor. Interface a stepper motor and three switches. Design a FSM controller that spins the motor as specified by the switches. One switch determines CW or CCW rotation, one switch determines fast or slow speed, and the third switch determines stop or go operation.

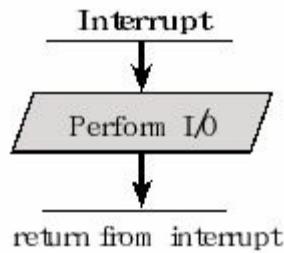
**Lab 8.3** Calculator. Interface both a keyboard and an LCD. Design, implement and test a four-function calculator using decimal fixed-point math.

**Lab 8.4** LCD interface. Interface an LCD to the microcontroller. Develop a software driver (a set of functions) similar to the Nokia5110 example project.

# 9. Interrupt Programming and Real-time Systems

## Chapter 9 objectives are to:

- Explain the fundamentals of interrupt programming,
- Introduce interrupt-driven I/O, and implement periodic interrupts,
- Explain edge-triggered interrupts and use them to interface individual switches,
- Present the timer-based modules needed for real-time systems,
- Develop methods to debug real-time events.



An embedded system uses its input/output devices to interact with the external world. Input devices allow the computer to gather information, and output devices can display information. Output devices also allow the computer to manipulate its environment. The tight-coupling between the computer and external world distinguishes an embedded system from a regular computer system. Given a change in input, it is not only necessary to get the correct response, but it will be necessary to get the correct response at the correct time. The challenge is under most situations the software executes much faster than the hardware. E.g., the software may ask the hardware to clear the LCD, but within the hardware this action might take 1 ms to complete. During this time, the software could execute thousands and thousands of instructions. Therefore, the synchronization between the executing software and its external environment is critical for the success of an embedded system. This chapter begins with an overview of I/O synchronization. We then present general concepts about interrupts, and specific details for the Cortex™-M microcontroller. We will then use periodic interrupts to cause a software task to be executed on a periodic basis. This chapter describes the timer-based modules used to design real-time embedded systems. If a GPIO pin is configured as an input, it can also be armed to invoke an interrupt on falling edges, rising edges or both falling and rising edges. This way software can respond quickly to changes in the external environment.

# 9.1. I/O Synchronization

**Latency** is the time between when the I/O device indicated service is required and the time when service is initiated. Latency includes hardware delays in the digital hardware plus computer software delays. For an input device, software latency (or software response time) is the time between new input data ready and the software reading the data. For an output device, latency is the delay from output device idle and the software giving the device new data to output. In this book, we will also have periodic events. For example, in our data acquisition systems, we wish to invoke the analog to digital converter (ADC) at a fixed time interval. In this way we can collect a sequence of digital values that approximate the continuous analog signal. Software latency in this case is the time between when the ADC conversion is supposed to be started, and when it is actually started. The microcomputer-based control system also employs periodic software processing. Similar to the data acquisition system, the latency in a control system is the time between when the control software is supposed to be run, and when it is actually run. A **real-time** system is one that can guarantee a worst case latency. In other words, the software response time is small and bounded. **Throughput** or **bandwidth** is the maximum data flow in bytes/second that can be processed by the system. Sometimes the bandwidth is limited by the I/O device, while other times it is limited by computer software. Bandwidth can be reported as an overall average or a short-term maximum. **Priority** determines the order of service when two or more requests are made simultaneously. Priority also determines if a high-priority request should be allowed to suspend a low priority request that is currently being processed. We may also wish to implement equal priority, so that no one device can monopolize the computer. In some computer literature, the term "soft-real-time" is used to describe a system that supports priority.

The purpose of our interface is to allow the microcontroller to interact with its external I/O device. There are five mechanisms to synchronize the microcontroller with the I/O device. Each mechanism synchronizes the I/O data transfer to the busy to done transition. The methods are discussed in the following paragraphs.

**Blind cycle** is a method where the software simply waits a fixed amount of time and assumes the I/O will complete before that fixed delay has elapsed. For an input device, the software triggers (starts) the external input hardware, waits a specified time, then reads data from device, see the left part of Figure 9.1. For an output device, the software writes data to the output device, triggers (starts) the device, then waits a specified time. We call this method **blind**, because there is no status information about the I/O device reported to the computer software. It is appropriate to use this method in situations where the I/O speed is short and predictable. One application of blind-cycle synchronization is the LCD, presented previously in Chapter 7. We can ask the LCD to display an ASCII character, wait 37  $\mu$ s, and then we are sure the operation is complete. This method works because the LCD speed is short and predictable. Many stepper motor interfaces use blind-cycle synchronization. If we repeat this 8-step sequence over and over 1) output a 0x05, 2) wait 1ms, 3) output a 0x06, 4) wait 1ms, 5) output a 0x0A, 6) wait 1ms, 7) output a 0x09, 8) wait 1ms, the motor will spin at a constant speed.

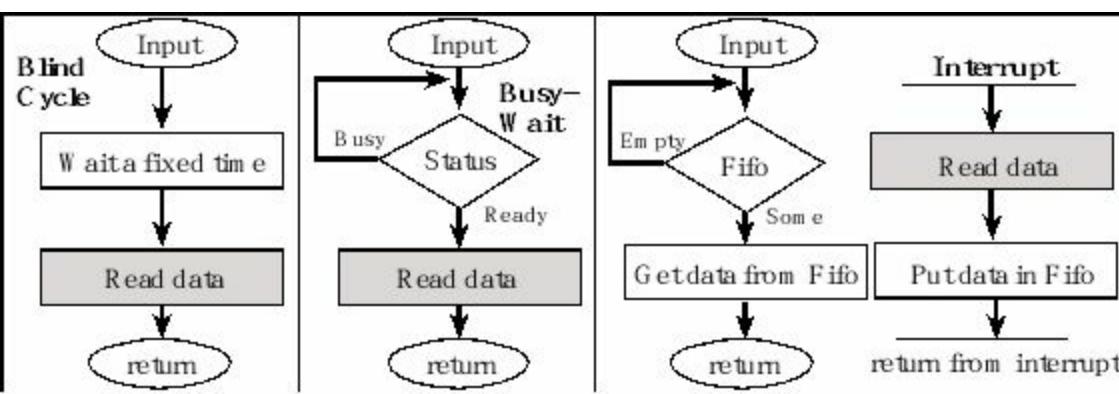


Figure 9.1. The input device sets a flag when it has new data.

**Busy wait or polling** is a software loop that checks the I/O status waiting for the done state. For an input device, the software waits until the input device has new data, and then reads it from the input device, see the middle part of Figure 9.1. For an output device, the software writes data, triggers the output device then waits until the device is finished. Another approach to output device interfacing is for the software to wait until the output device has finished the previous output, write data, and then trigger the device. Busy-wait synchronization will be used in situations where the software system is relatively simple and real-time response is not important. The UART software in Chapter 8 employed busy-wait synchronization.

An **interrupt** uses hardware to cause special software execution. With an input device, the hardware will request an interrupt when input device has new data. The software interrupt service will read from the input device and save in global RAM, see the right part of Figure 9.1. With an output device, the hardware will request an interrupt when the output device is idle. The software interrupt service will get data from a global structure, and then write to the device. Sometimes we configure the hardware timer to request interrupts on a periodic basis. The software interrupt service will perform a special function. A data acquisition system needs to read the ADC at a regular rate. Interrupt synchronization will be used in situations where the system is fairly complex (e.g., a lot of I/O devices) or when real-time response is important.

**Periodic polling** uses a clock interrupt to periodically check the I/O status. At the time of the interrupt the software will check the I/O status, performing actions as needed. With an input device, a ready flag is set when the input device has new data. At the next periodic interrupt after an input flag is set, the software will read the data and save them in global RAM. With an output device, a ready flag is set when the output device is idle. At the next periodic interrupt after an output flag is set, the software will get data from a global structure, and write it. Periodic polling will be used in situations that require interrupts, but the I/O device does not support interrupt requests directly.

**DMA**, or direct memory access, is an interfacing approach that transfers data directly to/from memory. With an input device, the hardware will request a DMA transfer when the input device has new data. Without the software's knowledge or permission the DMA controller will read data from the input device and save it in memory. With an output device, the hardware will request a DMA transfer when the output device is idle. The DMA controller will get data from memory, and then write it to the device. Sometimes we configure the hardware timer to request DMA transfers on a periodic basis. DMA can be used to implement a high-speed data acquisition system. DMA synchronization will be used in situations where high bandwidth and low latency are important. DMA will be discussed in Volume 3.

One can think of the hardware being in one of three states. The **idle** state is when the device is disabled or inactive. No I/O occurs in the idle state. When active (not idle) the hardware toggles between the **busy** and **ready** states. The interface includes a **flag** specifying either busy (0) or ready (1) status. Hardware-software synchronization revolves around this flag:

- The hardware will set the flag when the hardware component is complete.
- The software can read the flag to determine if the device is busy or ready.
- The software can clear the flag, signifying the software component is complete.
- This flag serves as the hardware triggering event for an interrupt.

For an input device, a status flag is set when new input data is available. The “busy to ready” state transition will cause a busy-wait loop to complete, see middle of Figure 9.1. Once the software recognizes the input device has new data, it will read the data and ask the input device to create more data. It is the **busy to ready** state transition that signals to the software that the hardware task is complete, and now software service is required. When the hardware is in the ready state the I/O transaction is complete. Often the simple process of reading the data will clear the flag and request another input.

The problem with I/O devices is that they are usually much slower than software execution. Therefore, we need synchronization, which is the process of the hardware and software waiting for each other in a manner such that data is properly transmitted. A way to visualize this synchronization is to draw a state versus time plot of the activities of the hardware and software. For an input device, the software begins by waiting for new input. When the input device is busy it is in the process of creating new input. When the input device is ready, new data is available. When the input device makes the transition from busy to ready, it releases the software to go forward. In a similar way, when the software accepts the input, it can release the input device hardware. The arrows in Figure 9.2 represent the synchronizing events. In this example, the time for the software to read and process the data is less than the time for the input device to create new input. This situation is called I/O bound, meaning the bandwidth is limited by the speed of the I/O hardware.

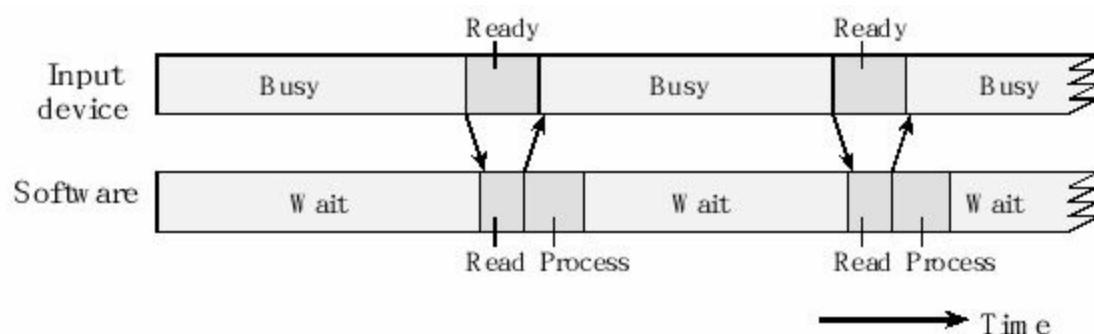


Figure 9.2. The software must wait for the input device to be ready (I/O bound input interface).

If the input device were faster than the software, then the software waiting time would be zero. This situation is called CPU bound (meaning the bandwidth is limited by the speed of the executing software). In real systems the bandwidth depends on both the hardware and the software. Another characteristic of real systems is the data can vary over time, like traffic arriving and leaving an intersection. In other words, the same I/O channel can sometimes be I/O bound, but at other times the channel could be CPU bound.

The busy-wait method is classified as unbuffered because the hardware and software must wait for each other during the transmission of each piece of data. The interrupt solution (shown in the right part of Figure 9.1) is classified as buffered, because the system allows the input device to run continuously, filling a FIFO with data as fast as it can. In the same way, the software can empty the buffer whenever it is ready and whenever there is data in the buffer. We will implement a buffered interface for the serial port input in Chapter 11 using interrupts. The buffering used in an interrupt interface may be a hardware FIFO, a software FIFO, or both hardware and software FIFOs. We will see the FIFO queues will allow the I/O interface to operate during both situations: I/O bound and CPU bound.

For an output device, a status flag is set when the output is idle and ready to accept more data. The “busy to ready” state transition causes a busy-wait loop to complete, see the middle part of Figure 9.3. Once the software recognizes the output is idle, it gives the output device another piece of data to output. It will be important to make sure the software clears the flag each time new output is started.

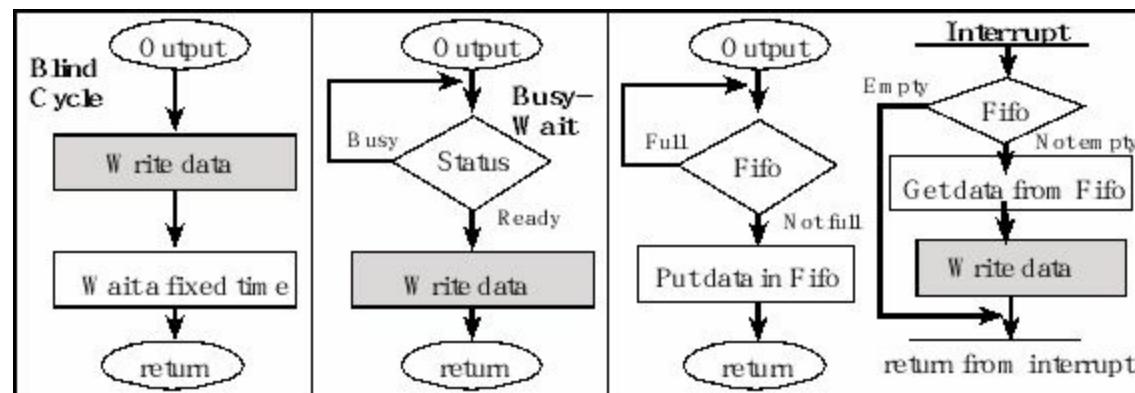


Figure 9.3. The output device sets a flag when it has finished outputting the last data.

Figure 9.4 contains a state versus time plot of the activities of the output device hardware and software. For an output device, the software begins by generating data then sending it to the output device. When the output device is busy it is processing the data. Normally when the software writes data to an output port, that only starts the output process. The time it takes an output device to process data is usually longer than the software execution time. When the output device is done, it is ready for new data. When the output device makes the transition from busy to ready, it releases the software to go forward. In a similar way, when the software writes data to the output, it releases the output device hardware. The output interface illustrated in Figure 9.4 is also I/O bound because the time for the output device to process data is longer than the time for the software to generate and write it. Again, I/O bound means the bandwidth is limited by the speed of the I/O hardware.

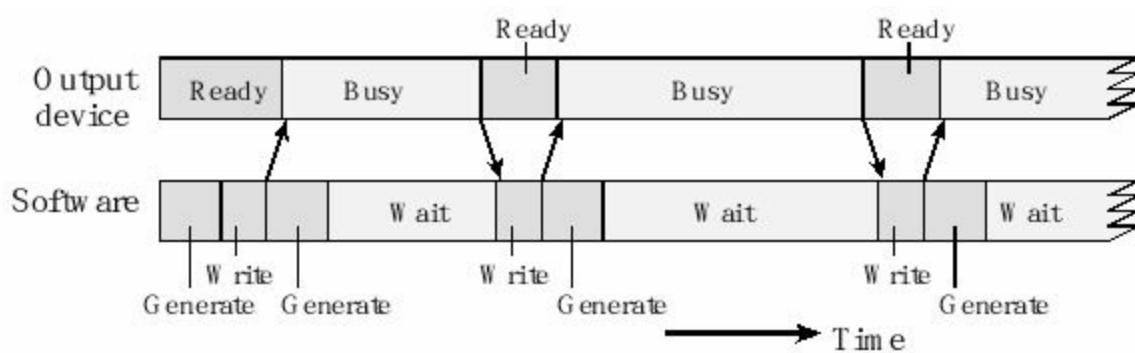


Figure 9.4. The software must wait for the output device to finish the previous operation (I/O bound).

The busy-wait solution for this output interface is also unbuffered, because when the hardware is done, it will wait for the software and after the software generates data, it waits for the hardware. On the other hand, the interrupt solution (shown as the right part of Figure 9.3) is buffered, because the system allows the software to run continuously, filling a FIFO as fast as it wishes. In the same way, the hardware can empty the buffer whenever it is ready and whenever there is data in the FIFO. We will implement a buffered interface for the serial port output in Chapter 11 using interrupts. Again, FIFO queues allow the I/O interface to operate during both situations: I/O bound and CPU bound.

On some systems an interrupt will be generated on a hardware failure. Examples include power failure, temperature too high, memory failure, and mechanical tampering of secure systems. Usually, these events are extremely important and require immediate attention. The Cortex-Mprocessor will execute special software (**fault**) when it tries to execute an illegal instruction, access an illegal memory location, or attempt an illegal I/O operation.

## 9.2. Interrupt Concepts

An **interrupt** is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution. This hardware event is called a **trigger**. The hardware event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer). When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag. A **thread** is defined as the path of action of software as it executes. The execution of the interrupt service routine is called a background thread. This thread is created by the hardware interrupt request and is killed when the interrupt service routine returns from interrupt (e.g., by executing a **BX LR**). A new thread is created for each interrupt request. It is important to consider each individual request as a separate thread because local variables and registers used in the interrupt service routine are unique and separate from one interrupt event to the next interrupt. In a **multi-threaded** system, we consider the threads as cooperating to perform an overall task. Consequently we will develop ways for the threads to communicate (e.g., FIFO) and synchronize with each other. Most embedded systems have a single common overall goal. On the other hand, general-purpose computers can have multiple unrelated functions to perform. A **process** is also defined as the action of software as it executes. Processes do not necessarily cooperate towards a common shared goal. Threads share access to I/O devices, system resources, and global variables, while processes have separate global variables and system resources. Processes do not share I/O devices.

There are no standard definitions for the terms mask, enable, and arm in the professional, Computer Science, or Computer Engineering communities. Nevertheless, in this book we will adhere to the following specific meanings. To **arm (disarm)** a device means to enable (shut off) the source of interrupts. Each potential interrupting trigger has a separate arm bit. One arms (disarms) a trigger if one is (is not) interested in interrupts from this source. To **enable (disable)** means to allow interrupts at this time (postponing interrupts until a later time). On the ARM ® Cortex™-M processor there is one interrupt enable bit for the entire interrupt system. We disable interrupts if it is currently not convenient to accept interrupts. In particular, to disable interrupts we set the I bit in **PRIMASK**.

The software has dynamic control over some aspects of the interrupt request sequence. First, each potential interrupt trigger has a separate **arm** bit that the software can activate or deactivate. The software will set the arm bits for those devices from which it wishes to accept interrupts, and will deactivate the arm bits within those devices from which interrupts are not to be allowed. In other words it uses the arm bits to individually select which devices will and which devices will not request interrupts. The second aspect that the software controls is the interrupt enable bit.

Specifically, bit 0 of the special register **PRIMASK** is the interrupt mask bit, **I**. If this bit is 1 most interrupts and exceptions are not allowed, which we will define as **disabled**. If the bit is 0, then interrupts are allowed, which we will define as **enabled**. The third aspect is priority. The **BASEPRI** register prevents interrupts with lower priority interrupts, but allows higher priority interrupts. For example if the software sets the **BASEPRI** to 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. The software can also specify the priority

level of each interrupt request. If **BASEPRI** is zero, then the priority feature is disabled and all interrupts are allowed. Five conditions must be true for an interrupt to be generated: device arm, NVIC enable, global enable, level, and trigger. A device must be armed by setting bits in its interrupt mask register; enabled in the NVIC; I bit must be 0; the level of the requested interrupt must be less than **BASEPRI**; and an external event must occur setting its trigger flag (**RIS** register). These five conditions must be simultaneously true but can occur in any order.

An interrupt causes the following sequence of five events. First, the current instruction is finished. Second, the execution of the currently running program is suspended, pushing eight registers on the stack (**R0**, **R1**, **R2**, **R3**, **R12**, **LR**, **PC**, and **PSR** with the **R0** on top). If the floating point unit on the TM4C123/TM4C1294 is active, an additional 18 words will be pushed on the stack representing the floating point state, making a total of 26 words. Third, the **LR** is set to a specific value signifying an interrupt service routine (ISR) is being run (bits [31:8] to 0xFFFFFFF, bits [7:1] specify the type of interrupt return to perform, bit 0 will always be 1 on the Cortex-M meaning Thumb mode). In our examples we will see LR is set to 0xFFFFFFFF9. If the floating point registers were pushed, the LR will be 0xFFFFFFF9. Fourth, the **IPSR** is set to the interrupt number being processed. Lastly, the **PC** is loaded with the address of the ISR (vector). These five steps, called a **context switch**, occur automatically in hardware as the context is switched from a foreground thread to a background thread. We can also have a context switch from a lower priority ISR to a higher priority ISR. Next, the software executes the ISR.

If a trigger flag is set, but the interrupts are disabled (I=1), the interrupt level is not high enough, or the flag is disarmed, the request is not dismissed. Rather the request is held **pending**, postponed until a later time, when the system deems it convenient to handle the requests. In other words, once the trigger flag is set, under most cases it remains set until the software clears it. The five necessary events (device arm, NVIC enable, global enable, level, and trigger) can occur in any order. For example, the software can set the I bit to prevent interrupts, run some code that needs to run to completion, and then clear the I bit. A trigger occurring while running with I=1 is postponed until the time the I bit is cleared again. In particular we will need to disable interrupts when executing nonreentrant code but disabling interrupts will have the effect of increasing the response time of software to external events.

Clearing a trigger flag is called **acknowledgement**, which occurs only by specific software action. Each trigger flag has a specific action software must perform to clear that flag. We will pay special attention to these enable/disable software actions. The SysTick periodic interrupt will be the only example of an automatic acknowledgement. For SysTick, the periodic timer requests an interrupt, but the trigger flag will be automatically cleared. For all the other trigger flags, the ISR must explicitly execute code that clears the flag.

The **interrupt service routine** (ISR) is the software module that is executed when the hardware requests an interrupt. There may be one large ISR that handles all requests (polled interrupts), or many small ISRs specific for each potential source of interrupt (vectored interrupts). The design of the interrupt service routine requires careful consideration of many factors. Except for the SysTick interrupt, the ISR must clear the trigger flag that caused the interrupt (acknowledge). After the ISR provides the necessary service, it will execute **BX LR**. Because LR contains a special value (e.g., 0xFFFFFFFF9), this instruction pops the 8 registers from the stack, which returns control to the main program. If the LR is 0xFFFFFFF9, then 28 registers will be popped by **BX LR**. There are two stack

pointers: PSP and MSP. The software in this book will exclusively use the MSP. It is imperative that the ISR software balance the stack before exiting. Execution of the previous thread will then continue with the exact stack and register values that existed before the interrupt. Although interrupt handlers can create and use local variables, parameter passing between threads must be implemented using shared global memory variables. A private global variable can be used if an interrupt thread wishes to pass information to itself, e.g., from one interrupt instance to another. The execution of the main program is called the foreground thread, and the executions of the various interrupt service routines are called background threads.

An axiom with interrupt synchronization is that the ISR should execute as fast as possible. The interrupt should occur when it is time to perform a needed function, and the interrupt service routine should perform that function, and return right away. Placing backward branches (busy-wait loops, iterations) in the interrupt software should be avoided if possible. The percentage of time spent executing any one ISR should be minimized. For an input device, the **interface latency** is the time between when new input is available, and the time when the software reads the input data. We can also define **device latency** as the response time of the external I/O device. For example, if we request that a certain sector be read from a disk, then the device latency is the time it takes to find the correct track and spin the disk (seek) so the proper sector is positioned under the read head. For an output device, the interface latency is the time between when the output device is idle, and the time when the software writes new data. A **real-time** system is one that can guarantee a worst case interface latency.

Many factors should be considered when deciding the most appropriate mechanism to synchronize hardware and software. One should not always use busy wait because one is too lazy to implement the complexities of interrupts. On the other hand, one should not always use interrupts because they are fun and exciting. Busy-wait synchronization is appropriate when the I/O timing is predictable and when the I/O structure is simple and fixed. Busy wait should be used for dedicated single thread systems where there is nothing else to do while the I/O is busy. Interrupt synchronization is appropriate when the I/O timing is variable, and when the I/O structure is complex. In particular, interrupts are efficient when there are I/O devices with different speeds. Interrupts allow for quick response times to important events. In particular, using interrupts is one mechanism to design real-time systems, where the interface latency must be short and bounded. **Bounded** means it is always less than a specified value. **Short** means the specified value is acceptable to our consumers.

Interrupts can also be used for infrequent but critical events like power failure, memory faults, and machine errors. Periodic interrupts will be useful for real-time clocks, data acquisition systems, and control systems. For extremely high bandwidth and low latency interfaces, DMA should be used.

An **atomic** operation is a sequence that once started will always finish, and cannot be interrupted. All instructions on the ARM ® Cortex™-M processor are atomic except store and load multiple, **STM**, **LDM**, **PUSH**, **POP**. If we wish to make a section of code atomic, we can run that code with I=1. In this way, interrupts will not be able to break apart the sequence. Again, requested interrupts that are triggered while I=1 are not dismissed, but simply postponed until I=0. In particular, to implement an atomic operation we will 1) save the current value of the **PRIMASK**, 2) disable interrupts, 3) execute the operation, and 4) restore the **PRIMASK** back to its previous value.

**Checkpoint 9.1:** What five conditions must be true for an interrupt to occur?

**Checkpoint 9.2:** How do you enable interrupts?

**Checkpoint 9.3:** What are the steps that occur when an interrupt is processed?

As you develop experience using interrupts, you will come to notice a few common aspects that most computers share. The following paragraphs outline three essential mechanisms that are needed to utilize interrupts. Although every computer that uses interrupts includes all three mechanisms, there are a wide spectrum of implementation methods.

All interrupting systems must have the **ability for the hardware to request action from computer**. In general, the interrupt requests can be generated using a separate connection to the processor for each device. The LM3S/TM4C microcontrollers use separate connections to request interrupts.

All interrupting systems must have the **ability for the computer to determine the source**. A vectored interrupt system employs separate connections for each device so that the computer can give automatic resolution. You can recognize a vectored system because each device has a separate interrupt vector address. With a polled interrupt system, the interrupt software must poll each device, looking for the device that requested the interrupt. Most interrupts on the LM3S/TM4C microcontrollers are vectored, but there are some triggers that share the same vector. For these interrupts the ISR must poll to see which trigger caused the interrupt. For example, all eight input pins on a GPIO port can trigger an interrupt, but the eight trigger flags share the same vector. So if multiple pins on a GPIO port are armed, the shared ISR must poll to determine which one(s) requested service.

The third necessary component of the interface is the **ability for the computer to acknowledge the interrupt**. Normally there is a trigger flag in the interface that is set on the busy to ready state transition, i.e., when the device needs service. In essence, this trigger flag is the cause of the interrupt. Acknowledging the interrupt involves clearing this flag. It is important to shut off the request, so that the computer will not mistakenly request a second (and inappropriate) interrupt service for the same condition. The first Intel x86 processors used a hardware acknowledgment that automatically clears the request. Except for periodic SysTick, LM3S/TM4C microcontrollers use software acknowledge. So when designing an interrupting interface, it will be important to know exactly what hardware conditions will set the trigger flag (and request an interrupt) and how the software will clear it (acknowledge) in the ISR.

**Common Error:** The system will crash if the interrupt service routine doesn't either acknowledge or disarm the device requesting the interrupt.

**Common Error:** The ISR software should not disable interrupts at the beginning nor should it reenable interrupts at the end. Which interrupts are allowed to run is automatically controlled by the priority set in the NVIC.

## 9.3. Interthread Communication and Synchronization

For regular function calls we use the registers and stack to pass parameters, but interrupt threads have logically separate registers and stack. In particular, registers are automatically saved by the processor as it switches from main program (foreground thread) to interrupt service routine (background thread). Exiting an ISR will restore the registers back to their previous values. Thus, all parameter passing must occur through global memory. One cannot pass data from the main program to the interrupt service routine using registers or the stack.

In this chapter, multi-threading means one main program (foreground thread) and multiple ISRs (background threads). An operating system allows multiple foreground threads (see Volume 3). Synchronizing threads is a critical task affecting efficiency and effectiveness of systems using interrupts. In this section, we will present in general form three constructs to synchronize threads: binary semaphore, mailbox, and FIFO queue.

A **binary semaphore** is simply a shared flag, as described in Figure 9.5. There are two operations one can perform on a semaphore. **Signal** is the action that sets the flag. **Wait** is the action that checks the flag, and if the flag is set, the flag is cleared and important stuff is performed. This flag must exist as a private global variable with restricted access to only the **Wait** and **Signal** functions. In C, we add the qualifier **static** to a global variable to restrict access to software within the same file. In order to reduce complexity of the system, it will be important to limit the access to this flag to as few modules as possible.

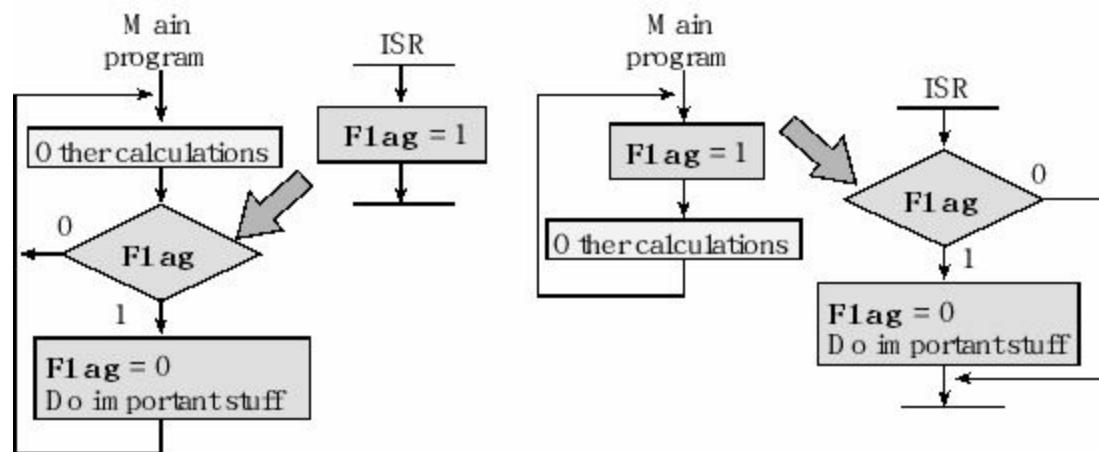


Figure 9.5. A semaphore can be used to synchronize threads.

A flag of course has two states: 0 and 1. However, it is good design to assign a meaning to this flag. For example, 0 might mean the switch has not been pressed, and 1 might mean the switch has been pressed. Figure 9.5 shows two examples of the binary semaphore. The big arrows in this figure signify the synchronization link between the threads. In the example on the left, the ISR signals the semaphore and the main program waits on the semaphore. Notice the “important stuff” is run in the foreground once per execution of the ISR. In the example on the right, the main program signals the semaphore and the ISR waits. It is good design to have NO backwards jumps in an ISR. In this particular application, if the ISR is running and the semaphore is 0, the action is just skipped and the computer returns from the interrupt.

The second interthread synchronization scheme is the **mailbox**. The mailbox is a binary semaphore with associated data variable. Figure 9.6 illustrates an input device interfaced using interrupt synchronization. The big arrow in this figure signifies the communication and synchronization link between the background and foreground. The mailbox structure is implemented with two shared global variables. **Mail** contains data, and **Status** is a semaphore flag specifying whether the mailbox is full or empty. The interrupt is requested when its trigger flag is set, signifying new data are ready from the input device. The ISR will read the data from the input device and store it in the shared global variable **Mail**, then update its status to full. The main program will perform other calculations, while occasionally checking the status of the mailbox. When the mailbox has data, the main program will process it. This approach is adequate for situations where the input bandwidth is slow compared to the software processing speed.

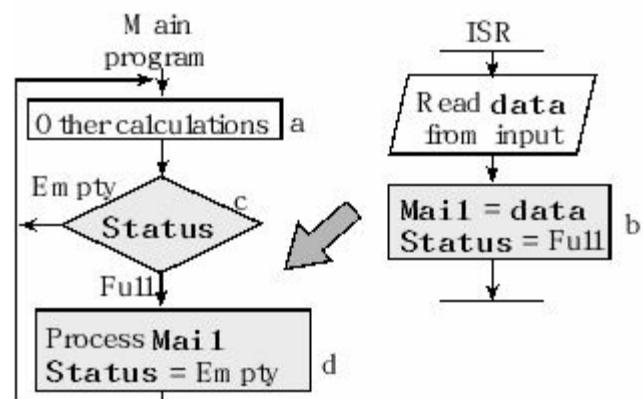
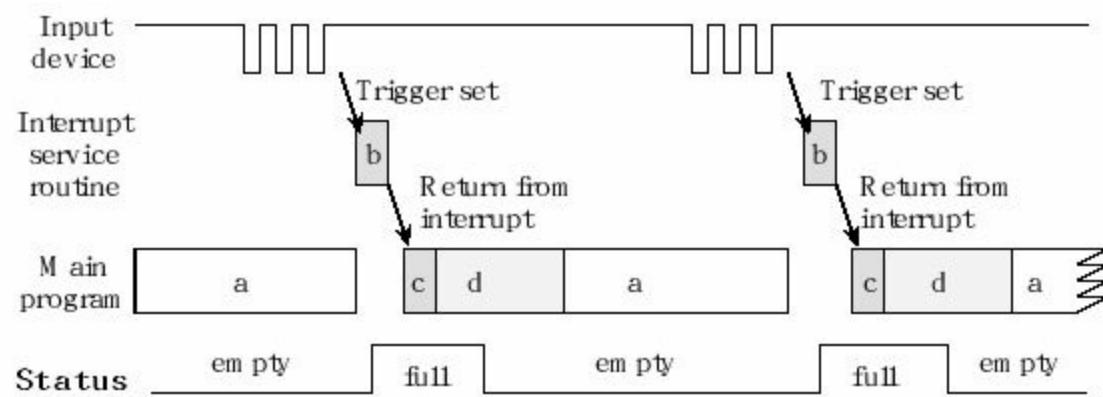


Figure 9.6. A mailbox can be used to pass data between threads.

One way to visualize the interrupt synchronization is to draw a state versus time plot of the activities of the hardware, the mailbox, and the two software threads (Figure 9.7).



## Figure 9.7. Hardware/software timing of an input interface using a mailbox.

Figure 9.7 shows that at time (a) the mailbox is empty, the input device is idle and the main program is performing other tasks, because mailbox is empty. When new input data are ready, the trigger flag will be set, and an interrupt will be requested. At time (b) the ISR reads data from input device and saves it in **Mail**, and then it sets **Status** to full. At time (c) the main program recognizes **Status** is full. At time (d) the main program processes data from **Mail**, sets **Status** to empty. Notice that even though there are two threads, only one is active at a time. The interrupt hardware switches the processor from the main program to the ISR, and the return from interrupt switches the processor back.

The third synchronization technique is the FIFO queue. Details of the FIFO will be presented in Chapter 11.

There are other types of interrupt that are not an input or output. For example we will configure the computer to request an interrupt on a periodic basis. This means an interrupt handler will be executed at fixed time intervals. This periodic interrupt will be essential for the implementation of real-time data acquisition and real-time control systems. For example if we are implementing a digital controller that executes a control algorithm 100 times a second, then we will set up the internal timer hardware to request an interrupt every 10 ms. The interrupt service routine will execute the digital control algorithm and then return to the main thread. In a similar fashion, we will use periodic interrupts in Chapter 10 to perform analog input and/or analog output. For example if we wish to sample the ADC 100 times a second, then we will set up the internal timer hardware to request an interrupt every 10 ms. The interrupt service routine will sample the ADC, process (or save) the data, and then return to the main thread.

**Performance Tip:** It is poor design to employ backward jumps in an ISR, because they may affect the latency of other interrupt requests. Whenever you are thinking about using a backward jump, consider redesigning the system with more or different triggers to reduce the number of backward jumps.

## 9.4. NVIC on the ARM Cortex-M Processor

On the ARM® Cortex™-M processor, exceptions include resets, software interrupts and hardware interrupts. Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located. Vectors are stored in ROM at the beginning of memory. Program 9.1 shows the first few vectors as defined in the **Startup.s** file. **DCD** is an assembler pseudo-op that defines a 32-bit constant. ROM location 0x0000.0000 has the initial stack pointer, and location 0x0000.0004 contains the initial program counter, which is called the reset vector. It points to a function called the reset handler, which is the first thing executed following reset. There are up to 240 possible interrupt sources and their 32-bit vectors are listed in order starting with location 0x0000.0008. From a programming perspective, we can attach ISRs to interrupts by writing the ISRs as regular assembly subroutines or C functions with no input or output parameters and editing the **Startup.s** file to specify those functions for the appropriate interrupt. For example, if we wrote a Port C interrupt service routine named **PortCISR**, then we would

replace **GPIOPortC\_Handler** with **PortCISR**. In this book, we will write our ISRs using standard function names so that the **Startup.s** file need not be edited. I.e., we will simply name the ISR for edge-triggered interrupts on Port C as **GPIOPortC\_Handler**. The ISR for this interrupt is a 32-bit pointer located at ROM address 0x0000.0048. Because the vectors are in ROM, this linkage is defined at compile time and not at run time. For more details see the **Startup.s** files within the interrupt examples posted on the book web site. Each processor is a little different so check the data sheet.

**Checkpoint 9.4:** Where is the vector for SysTick? What is the standard name for this ISR?

### EXPORT \_\_Vectors

Vectors	; address ISR
DCD StackMem + Stack	; 0x00000000 Top of Stack
DCD Reset_Handler	; 0x00000004 Reset Handler
DCD NMI_Handler	; 0x00000008 NMI Handler
DCD HardFault_Handler	; 0x0000000C Hard Fault Handler
DCD MemManage_Handler	; 0x00000010 MPU Fault Handler
DCD BusFault_Handler	; 0x00000014 Bus Fault Handler
DCD UsageFault_Handler	; 0x00000018 Usage Fault Handler
DCD 0	; 0x0000001C Reserved
DCD 0	; 0x00000020 Reserved
DCD 0	; 0x00000024 Reserved
DCD 0	; 0x00000028 Reserved
DCD SVC_Handler	; 0x0000002C SVCall Handler
DCD DebugMon_Handler	; 0x00000030 Debug Monitor Handler
DCD 0	; 0x00000034 Reserved
DCD PendSV_Handler	; 0x00000038 PendSV Handler

```

DCD SysTick_Handler ; 0x0000003C SysTick Handler
DCD GPIOPortA_Handler ; 0x00000040 GPIO Port A
DCD GPIOPortB_Handler ; 0x00000044 GPIO Port B
DCD GPIOPortC_Handler ; 0x00000048 GPIO Port C
DCD GPIOPortD_Handler ; 0x0000004C GPIO Port D
DCD GPIOPortE_Handler ; 0x00000050 GPIO Port E
DCD UART0_Handler ; 0x00000054 UART0
DCD UART1_Handler ; 0x00000058 UART1
DCD SSI0_Handler ; 0x0000005C SSI
DCD I2C0_Handler ; 0x00000060 I2C
DCD PWM0Fault_Handler ; 0x00000064 PWM Fault
DCD PWM0_Handler ; 0x00000068 PWM Generator 0

```

Program 9.1. Software syntax to set the interrupt vectors for the LM3S/TM4C.

Program 9.2 shows that the syntax for a ISR looks like a function with no parameters. Notice that each ISR (except for SysTick) must acknowledge the interrupt.

<b>GPIOPortC_Handler</b>	<b>void</b>
LDR	<b>GPIOPortC_Handler(void){</b>
R0,=GPIO_PORTC_ICR_R	<b>GPIO_PORTC_ICR_R =</b>
MOV R1,#0x10	<b>0x10; // ack</b>
STR R1,[R0] ; ack	// stuff
;stuff	}
<b>BX LR ;return from interrupt</b>	

Program 9.2. Typical interrupt service routine.

Vector address	Number	IRQ	ISR name in Startup.s	NVIC	Priority bits
0x00000038	14	-2	PendSV_Handler	NVIC_SYS_PRI3_R	23 – 21
0x0000003C	15	-1	SysTick_Handler	NVIC_SYS_PRI3_R	31 – 29
0x00000040	16	0	GPIOPortA_Handler	NVIC_PRI0_R	7 – 5
0x00000044	17	1	GPIOPortB_Handler	NVIC_PRI0_R	15 – 13
0x00000048	18	2	GPIOPortC_Handler	NVIC_PRI0_R	23 – 21
0x0000004C	19	3	GPIOPortD_Handler	NVIC_PRI0_R	31 – 29
0x00000050	20	4	GPIOPortE_Handler	NVIC_PRI1_R	7 – 5
0x00000054	21	5	UART0_Handler	NVIC_PRI1_R	15 – 13
0x00000058	22	6	UART1_Handler	NVIC_PRI1_R	23 – 21
0x0000005C	23	7	SSI0_Handler	NVIC_PRI1_R	31 – 29
0x00000060	24	8	I2C0_Handler	NVIC_PRI2_R	7 – 5
0x00000064	25	9	PWM0Fault_Handler	NVIC_PRI2_R	15 – 13
0x00000068	26	10	PWM0_Handler	NVIC_PRI2_R	23 – 21
0x0000006C	27	11	PWM1_Handler	NVIC_PRI2_R	31 – 29
0x00000070	28	12	PWM2_Handler	NVIC_PRI3_R	7 – 5
0x00000074	29	13	Quadrature0_Handler	NVIC_PRI3_R	15 – 13
0x00000078	30	14	ADC0_Handler	NVIC_PRI3_R	23 – 21
0x0000007C	31	15	ADC1_Handler	NVIC_PRI3_R	31 – 29

0x000000080	32	16	ADC2_Handler	NVIC_PRI4_R	7 – 5
0x000000084	33	17	ADC3_Handler	NVIC_PRI4_R	15 – 13
0x000000088	34	18	WDT_Handler	NVIC_PRI4_R	23 – 21
0x00000008C	35	19	Timer0A_Handler	NVIC_PRI4_R	31 – 29
0x000000090	36	20	Timer0B_Handler	NVIC_PRI5_R	7 – 5
0x000000094	37	21	Timer1A_Handler	NVIC_PRI5_R	15 – 13
0x000000098	38	22	Timer1B_Handler	NVIC_PRI5_R	23 – 21
0x00000009C	39	23	Timer2A_Handler	NVIC_PRI5_R	31 – 29
0x0000000A0	40	24	Timer2B_Handler	NVIC_PRI6_R	7 – 5
0x0000000A4	41	25	Comp0_Handler	NVIC_PRI6_R	15 – 13
0x0000000A8	42	26	Comp1_Handler	NVIC_PRI6_R	23 – 21
0x0000000AC	43	27	Comp2_Handler	NVIC_PRI6_R	31 – 29
0x0000000B0	44	28	SysCtl_Handler	NVIC_PRI7_R	7 – 5
0x0000000B4	45	29	FlashCtl_Handler	NVIC_PRI7_R	15 – 13
0x0000000B8	46	30	GPIOPortF_Handler	NVIC_PRI7_R	23 – 21
0x0000000BC	47	31	GPIOPortG_Handler	NVIC_PRI7_R	31 – 29
0x0000000C0	48	32	GPIOPortH_Handler	NVIC_PRI8_R	7 – 5
0x0000000C4	49	33	UART2_Handler	NVIC_PRI8_R	15 – 13
0x0000000C8	50	34	SSI1_Handler	NVIC_PRI8_R	23 – 21
0x0000000CC	51	35	Timer3A_Handler	NVIC_PRI8_R	31 – 29
0x0000000D0	52	36	Timer3B_Handler	NVIC_PRI9_R	7 – 5
0x0000000D4	53	37	I2C1_Handler	NVIC_PRI9_R	15 – 13
0x0000000D8	54	38	Quadrature1_Handler	NVIC_PRI9_R	23 – 21
0x0000000DC	55	39	CAN0_Handler	NVIC_PRI9_R	31 – 29
0x0000000E0	56	40	CAN1_Handler	NVIC_PRI10_R	7 – 5
0x0000000E4	57	41	CAN2_Handler	NVIC_PRI10_R	15 – 13
0x0000000E8	58	42	Ethernet_Handler	NVIC_PRI10_R	23 – 21
0x0000000EC	59	43	Hibernate_Handler	NVIC_PRI10_R	31 – 29
0x0000000F0	60	44	USB0_Handler	NVIC_PRI11_R	7 – 5
0x0000000F4	61	45	PWM3_Handler	NVIC_PRI11_R	15 – 13
0x0000000F8	62	46	uDMA_Handler	NVIC_PRI11_R	23 – 21
0x0000000FC	63	47	uDMA_Error	NVIC_PRI11_R	31 – 29

**Table 9.1. Some of the interrupt vectors for the LM3S/TM4C123.**

Interrupts on the Cortex-M are controlled by the Nested Vectored Interrupt Controller (NVIC). To activate an interrupt source we need to set its priority and enable that source in the NVIC. This activation is in addition to the arm and enable steps. Table 9.1 lists the interrupt sources available on the LM3S/TM4C family of microcontrollers. Interrupt numbers 0 to 15 contain the faults, software interrupt and SysTick; these interrupts will be handled differently from interrupts 16 to 63.

Table 9.2 shows some of the priority registers on the NVIC. Each register contains an 8-bit priority field for four devices. On the LM3S/TM4C microcontrollers, only the top three bits of the 8-bit field are used. This allows us to specify the interrupt priority level for each device from 0 to 7, with 0 being the highest priority. The interrupt number (number column in Table 9.1) is loaded into the **IPSR** register. The servicing of interrupts does not set the I bit in the **PRIMASK**, so a higher priority interrupt can suspend the execution of a lower priority ISR. If a request of equal or lower priority is generated while an ISR is being executed, that request is postponed until the ISR is completed. In particular, those devices that need prompt service should be given high priority.

Address	31 – 29	23 – 21	15 – 13	7 – 5	Name
0xE000E400	GPIO Port D	GPIO Port C	GPIO Port B	GPIO Port A	NVIC_PRI0_R
0xE000E404	SSI0, Rx Tx	UART1, Rx Tx	UART0, Rx Tx	GPIO Port E	NVIC_PRI1_R
0xE000E408	PWM Gen 1	PWM Gen 0	PWM Fault	I2C0	NVIC_PRI2_R
0xE000E40C	ADC Seq 1	ADC Seq 0	Quad Encoder	PWM Gen 2	NVIC_PRI3_R
0xE000E410	Timer 0A	Watchdog	ADC Seq 3	ADC Seq 2	NVIC_PRI4_R
0xE000E414	Timer 2A	Timer 1B	Timer 1A	Timer 0B	NVIC_PRI5_R
0xE000E418	Comp 2	Comp 1	Comp 0	Timer 2B	NVIC_PRI6_R
0xE000E41C	GPIO Port G	GPIO Port F	Flash Control	System Control	NVIC_PRI7_R
0xE000E420	Timer 3A	SSI1, Rx Tx	UART2, Rx Tx	GPIO Port H	NVIC_PRI8_R
0xE000E424	CAN0	Quad Encoder 1	I2C1	Timer 3B	NVIC_PRI9_R
0xE000E428	Hibernate	Ethernet	CAN2	CAN1	NVIC_PRI10_R
0xE000E42C	uDMA Error	uDMA Soft Tfr	PWM Gen 3	USB0	NVIC_PRI11_R
0xE000ED20	SysTick	PendSV	--	Debug	NVIC_SYS_PRI3_R

**Table 9.2. The LM3S/TM4C NVIC registers. Each register is 32 bits wide. Bits not shown are zero.**

There are two enable registers **NVIC\_EN0\_R** and **NVIC\_EN1\_R**. The 32 bits in register **NVIC\_EN0\_R** control the IRQ numbers 0 to 31 (interrupt numbers 16 – 47). In Table 9.1 we see UART0 is IRQ=5. To enable UART0 interrupts we set bit 5 in **NVIC\_EN0\_R**, see Table 9.3. The bottom 16 bits in **NVIC\_EN1\_R** control the IRQ numbers 32 to 47 (interrupt numbers 48 – 63). In Table 9.1 we see UART2 is IRQ=33. To enable UART2 interrupts we set bit 1 (33-32=1) in **NVIC\_EN1\_R**, see Table 9.3. Not every interrupt source is available on every LM3S/TM4C microcontroller, so you will need to refer to the data sheet for your microcontroller when designing I/O interfaces. Writing zeros to the **NVIC\_EN0\_R** **NVIC\_EN1\_R** registers has no effect. To disable interrupts we write ones to the corresponding bit in the **NVIC\_DIS0\_R** or **NVIC\_DIS1\_R** register.

Address	31	30	29-	6	5	4	3	2	1	0	Name
0xE000E100	G	F	...	UART1	UART0	E	D	C	B	A	NVIC_EN0_R
0xE000E104			...						UART2	H	NVIC_EN1_R

**Table 9.3. The LM3S/TM4C NVIC interrupt enable registers.**

Figure 9.8 shows the context switch from executing in the foreground to running an edge-triggered ISR from Port C. Assume Port C interrupts are configured for a priority level of 2. The I bit in the PRIMASK is 0 signifying interrupts are enabled. The interrupt number (ISRNUM) in the IPSR register is 0, meaning we are running in **Thread mode** (i.e., the main program, and not an ISR). **Handler mode** is signified by a nonzero value in IPSR. When BASEPRI register is zero, all interrupts are allowed and the BASEPRI register is not active.

When a Port C interrupt is triggered, the current instruction is finished. (a) Eight registers are pushed on the stack with **R0** on top. These registers are pushed onto the stack using whichever stack pointer is active: either the **MSP** or **PSP**. (b) The vector address is loaded into the **PC** (“Vector address” column in Table 9.1). (c) The **IPSR** register is set to 18 (“Number” column in Table 9.1) (d) The top 24 bits of **LR** are set to 0xFFFFFFF, signifying the processor is executing an ISR. The bottom eight bits specify how to return from interrupt.

0xE1 Return to Handler mode MSP (using floating point state on TM4C)

0xE9 Return to Thread mode MSP (using floating point state on TM4C)

0xED Return to Thread mode PSP (using floating point state on TM4C)

0xF1 Return to Handler mode MSP

0xF9 Return to Thread mode MSP ← we will mostly be using this one

0xFD Return to Thread mode PSP

After pushing the registers, the processor always uses the main stack pointer (**MSP**) during the execution of the ISR. Events b, c, and d can occur simultaneously

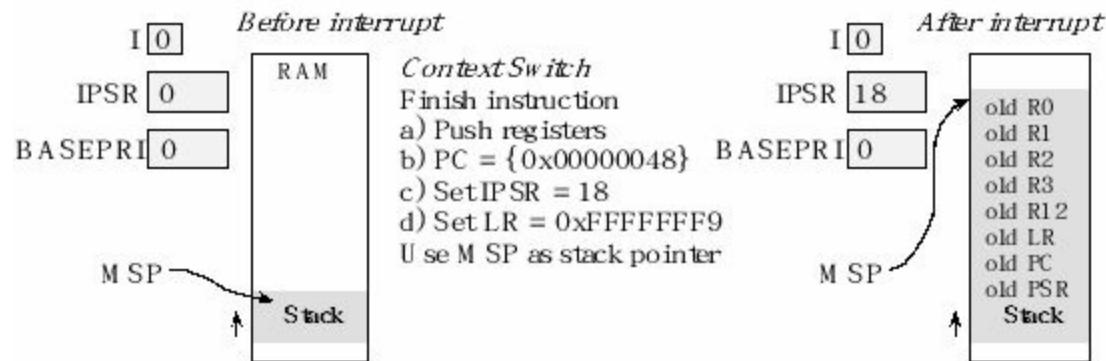


Figure 9.8. Stack before and after an interrupt.

To **return from an interrupt**, the ISR executes the typical function **return BX LR**. However, since the top 24 bits of **LR** are 0xFFFFFFF, it knows to return from interrupt by popping the eight registers off the stack. Since the bottom eight bits of **LR** in this case are 0b11111001, it returns to thread mode using the **MSP** as its stack pointer. Since the **IPSR** is part of the **PSR** that is popped, it is automatically reset its previous state.

A **nested interrupt** occurs when a higher priority interrupt suspends an ISR. The lower priority interrupt will finish after the higher priority ISR completes. When one interrupt preempts another, the **LR** is set to 0xFFFFFFFF1, so it knows to return to handler mode. **Tail chaining** occurs when one ISR executes immediately after another. Optimization occurs because the eight registers need not be popped only to be pushed once again. If an interrupt is triggered and is in the process of stacking registers when a higher priority interrupt is requested, this **late arrival interrupt** will be executed first.

On the Cortex™-M4, if an interrupt occurs while in the floating point state, an additional 18 words are pushed on the stack. These 18 words will save the state of the floating point processor. Bits 7-4 of the **LR** will be 0b1110 (0xE), signifying it was interrupted during a floating point state. When the ISR returns, it knows to pull these 18 words off the stack and restore the state of the floating point processor.

**Priority** determines the order of service when two or more requests are made simultaneously. Priority also allows a higher priority request to suspend a lower priority request currently being processed. Usually, if two requests have the same priority, we do not allow them to interrupt each other. NVIC assigns a priority level to each interrupt trigger. This mechanism allows a higher priority trigger to interrupt the ISR of a lower priority request. Conversely, if a lower priority request occurs while running an ISR of a higher priority trigger, it will be postponed until the higher priority service is complete.

**Observation:** There are many interrupt sources, but an effective system will use only a few.

Program 9.3 gives the definitions in **startup.s** that allow the software to enable and disable interrupts. These functions are callable from either assembly or C code.

```
;***** DisableInterrupts *****
; disable interrupts
; inputs: none           outputs: none
DisableInterrupts CPSID I ;set I=1
    BX    LR
```

```
;***** EnableInterrupts *****
; enable interrupts
; inputs: none           outputs: none
EnableInterrupts CPSIE I ;set I=0
    BX    LR
```

Program 9.3. Assembly functions needed for interrupt enabling and disabling.

## 9.5. Edge-triggered Interrupts

Synchronizing software to hardware events requires the software to recognize when the hardware changes states from busy to done. Many times the busy to done state transition is signified by a rising (or falling) edge on a status signal in the hardware. For these situations, we connect this status signal to an input of the microcontroller, and we use edge-triggered interfacing to configure the interface to set a flag on the rising (or falling) edge of the input. Using edge-triggered interfacing allows the software to respond quickly to changes in the external world. If we are using busy-wait synchronization, the software waits for the flag. If we are using interrupt synchronization, we configure the flag to request an interrupt when set. Each of the digital I/O pins on the LM3S/TM4C family can be configured for edge triggering. Table 4.5 listed some of the I/O registers. Table 9.4 expands this list to include many of the registers available for Port A. The differences between members of the LM3S/TM4C family include the number of ports (e.g., the TM4C123 has ports A – F, while the TM4C1294 has ports A,B,C,D, E,F,G,H,J,K,L,M,N,P, and Q) and the number of pins in each port (e.g., the TM4C123 has pins 7 – 0 in Port B, while the TM4C1294 only has pins 5 – 0 in Port B). For more details, refer to the datasheet for your specific microcontroller. Any or all of digital I/O pins can be configured as an edge-triggered input. When writing C code using these registers, include the header file for your particular microcontroller (e.g., **tm4c123ge6pm.h**). To use any of the features for a digital I/O port, we first enable its clock in the **SYSCTL\_RCGCGPIO\_R** register. For each bit we wish to use we must set the corresponding **DEN** (Digital Enable) bit. To use a pin as regular digital input or output, we clear its **AFSEL** (Alternate Function Select) bit. Setting the **AFSEL** will activate the pin's special function (e.g., UART, I<sup>2</sup>C, CAN etc.) For regular digital input/output, we clear **DIR** (Direction) bits to make them input, and we set **DIR** bits to make them output. On the TM4C123, only pins PD7 and PF0 need to be unlocked. We clear bits in the **AMSEL** register to disable analog function. See Tables 4.3, 4.4 to see the **PCTL** bits.

Address	7	6	5	4	3	2	1	0	Name
\$4000.43FC	DATA	GPIO_PORTA_DATA_R							
\$4000.4400	DIR	GPIO_PORTA_DIR_R							
\$4000.4404	IS	GPIO_PORTA_IS_R							
\$4000.4408	IBE	GPIO_PORTA_IBE_R							
\$4000.440C	IEV	GPIO_PORTA_IEV_R							
\$4000.4410	IME	GPIO_PORTA_IM_R							
\$4000.4414	RIS	GPIO_PORTA_RIS_R							
\$4000.4418	MIS	GPIO_PORTA_MIS_R							
\$4000.441C	ICR	GPIO_PORTA_ICR_R							
\$4000.4420	SEL	GPIO_PORTA_AFSEL_R							
\$4000.4500	DRV2	GPIO_PORTA_DR2R_R							
\$4000.4504	DRV4	GPIO_PORTA_DR4R_R							
\$4000.4508	DRV8	GPIO_PORTA_DR8R_R							
\$4000.450C	ODE	GPIO_PORTA_ODR_R							
\$4000.4510	PUE	GPIO_PORTA_PUR_R							
\$4000.4514	PDE	GPIO_PORTA_PDR_R							
\$4000.4518	SLR	GPIO_PORTA_SLR_R							

\$4000.451C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTA_DEN_R
\$4000.4524	CR	CR	CR	CR	CR	CR	CR	CR	GPIO_PORTA_CR_R
\$4000.4528	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTA_AMSEL_R
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0	
\$4000.452C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTA_PCTL_R
\$4000.4520	LOCK (32 bits)								GPIO_PORTA_LOCK_R

**Table 9.4. Some TM4C123 port A registers. For PMC bits, see Tables 4.3, 4.4.**

To configure an edge-triggered pin, we first enable the clock on the port and configure the pin as a regular digital input. Clearing the **IS** (Interrupt Sense) bit configures the bit for edge triggering. If the **IS** bit were to be set, the trigger occurs on the level of the pin. Since most busy to done conditions are signified by edges, we typically trigger on edges rather than levels. Next we write to the **IBE** (Interrupt Both Edges) and **IEV** (Interrupt Event) bits to define the active edge. We can trigger on the rising, falling, or both edges, as listed in Table 9.5. We clear the **IME** (Interrupt Mask Enable) bits if we are using busy-wait synchronization, and we set the **IME** bits to use interrupt synchronization.

The hardware sets an **RIS** (Raw Interrupt Status) bit (called the trigger) and the software clears it (called the acknowledgement). The triggering event listed in Table 9.5 will set the corresponding **RIS** bit in the **GPIO\_PORTA\_RIS\_R** register regardless of whether or not that bit is allowed to request an interrupt. In other words, clearing an **IME** bit disables the corresponding pin's interrupt, but it will still set the corresponding **RIS** bit when the interrupt would have occurred. To use interrupts, clear the **IME** bit, configure the bits in Table 9.5, and then set the **IME** bit. The software can acknowledge the event by writing ones to the corresponding **IC**(Interrupt Clear) bit in the **GPIO\_PORTA\_IC\_R** register. The **RIS** bits are read only, meaning if the software were to write to this registers, it would have no effect. For example, to clear bits 2, 1, and 0 in the **GPIO\_PORTA\_RIS\_R** register, we write a 0x07 to the **GPIO\_PORTA\_IC\_R** register. Writing zeros into **IC** bits will not affect the **RIS** bits.

DIR	AFSEL	IS	IBE	IEV	Port mode
0	0	0	0	0	Input, falling edge trigger
0	0	0	0	1	Input, rising edge trigger
0	0	0	1	-	Input, both edges trigger
0	0	1	0	0	Input, low level trigger
0	0	1	0	1	Input, high-level trigger

**Table 9.5. Edge-triggered and level-active interrupt modes (set **IME=1** to arm interrupt).**

For input signals we have the option of adding either a pull-up resistor or a pull-down resistor. If we set the corresponding **PUE** (Pull-Up Enable) bit on an input pin, the equivalent of a 13 kΩ □ to 30 kΩ resistor to +3.3 V power is internally connected to the pin. Similarly, if we set the corresponding **PDE** (Pull-Down Enable) bit on an input pin, the equivalent of a 13 kΩ to 35 kΩ resistor to ground is internally connected to the pin. We cannot have both pull-up and a pull-down resistor, so setting a bit in one register automatically clears the corresponding bit in the other register.

A typical application of pull-up and pull-down mode is the interface of simple switches. Using these modes eliminates the need for an external resistor when interfacing a switch. Compare the interfaces on Port A to the interfaces on Port B illustrated in Figure 9.9. The PA2 and PA3 interfaces will use software-configured internal resistors, while the PB2 and PB3 interfaces use actual resistors. The PA2 and PB2 interfaces in Figure 9.9a) implement negative logic switch inputs, and the PA3 and PB3 interfaces in Figure 9.9b) implement positive logic switch inputs.

**Checkpoint 9.5:** What do negative logic and positive logic mean in this context?

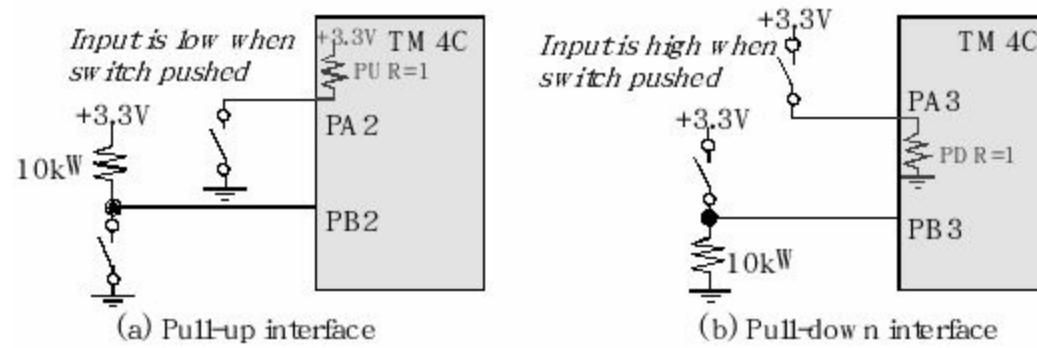


Figure 9.9. Edge-triggered interfaces can generate interrupts on a switch touch.

Using edge triggering to synchronize software to hardware centers around the operation of the trigger flags, **RIS**. A busy-wait interface will read the appropriate **RIS** bit over and over, until it is set. When the **RIS** bit is set, the software will clear the **RIS** bit (by writing a one to the corresponding **IC** bit) and perform the desired function. With interrupt synchronization, the initialization phase will arm the trigger flag by setting the corresponding **IME** bit. In this way, the active edge of the pin will set the **RIS** and request an interrupt. The interrupt will suspend the main program and run a special interrupt service routine (ISR). This ISR will clear the **RIS** bit and perform the desired function. At the end of the ISR it will return, causing the main program to resume. In particular, five conditions must be simultaneously true for an edge-triggered interrupt to be requested:

- The trigger flag bit is set (**RIS**)
- The arm bit is set (**IME**)
- The level of the edge-triggered interrupt must be less than **BASEPRI**
- The edge-triggered interrupt must be enabled in the **NVIC\_EN0\_R**
- Bit 0 of the special register **PRIMASK** is 0

**Checkpoint 9.6:** What values do you write into **DIR**, **AFSEL**, **PUE**, and **PDE** to configure the switch interfaces of PA2 and PA3 in Figure 9.9?

Table 9.4 listed the registers for Port A. The other ports have similar registers. We will begin with a simple example that counts the number of rising edges on Port F bit 4 (Program 9.4). The initialization requires many steps. (a) The clock for the port must be enabled. (b) The global variables should be initialized. (c) The appropriate pins must be enabled as inputs. (d) We must specify whether to trigger on the rise, the fall, or both edges. In this case we will trigger on the rise of PF4. (e) It is good design to clear the trigger flag during initialization so that the first interrupt occurs due to the first rising edge after the initialization has been run. We do not wish to trigger on a rising edge that might have occurred during the power up phase of the system. (f) We arm the edge-trigger by setting the corresponding bits in the **IM** register. (g) We establish the priority of Port F by setting bits 23 – 21 in the **NVIC\_PRI7\_R** register as listed in Table 9.2. We activate Port F interrupts in the NVIC by setting bit 30 in the **NVIC\_EN0\_R** register, Table 9.3. There is no need to unlock PF4.

```

volatile uint32_t FallingEdges = 0;
void EdgeCounter_Init(void){
    SYSCTL_RCGCGPIO_R |= 0x20; // (a) activate clock for port F
    FallingEdges = 0; // (b) initialize counter
    GPIO_PORTF_DIR_R &= ~0x10; // (c) make PF4 in (built-in button)
    GPIO_PORTF_AFSEL_R &= ~0x10; // disable alt funct on PF4
    GPIO_PORTF_DEN_R |= 0x10; // enable digital I/O on PF4
    GPIO_PORTF_PCTL_R &= ~0x000F0000; // configure PF4 as GPIO
    GPIO_PORTF_AMSEL_R &= ~0x10; // disable analog functionality on PF4
    GPIO_PORTF_PUR_R |= 0x10; // enable weak pull-up on PF4
    GPIO_PORTF_IS_R &= ~0x10; // (d) PF4 is edge-sensitive
    GPIO_PORTF_IBE_R &= ~0x10; // PF4 is not both edges
    GPIO_PORTF_IEV_R &= ~0x10; // PF4 falling edge event
    GPIO_PORTF_ICR_R = 0x10; // (e) clear flag4
    GPIO_PORTF_IM_R |= 0x10; // (f) arm interrupt on PF4
    NVIC_PRI7_R = (NVIC_PRI7_R & 0xFF00FFFF) | 0x00A00000; // (g) priority 5
    NVIC_EN0_R = 0x40000000; // (h) enable interrupt 30 in NVIC
    EnableInterrupts(); // (i) Program 9.3
}
void GPIOPortF_Handler(void){
    GPIO_PORTF_ICR_R = 0x10; // acknowledge flag4
    FallingEdges = FallingEdges + 1;
}

```

Program 9.4. Interrupt-driven edge-triggered input that counts rising edges of PF4 (EdgeInterrupt\_xxx.zip).

This initialization is shown to enable interrupts in step (i). However, in most systems we would not enable interrupts in the device initialization. Rather, it is good design to initialize all devices in the system, then enable interrupts. All ISRs must acknowledge the interrupt by clearing the trigger flag that requested the interrupt. For edge-triggered PF4, the trigger flag is bit 4 of the **GPIO\_PORTF\_RIS\_R** register. This flag can be cleared by writing a 0x10 to **GPIO\_PORTF\_ICR\_R**.

If two or more triggers share the same vector, these requests are called **polled interrupts**, and the ISR must determine which trigger generated the interrupt. If the requests have separate vectors, then these requests are called **vectored interrupts** and the ISR knows which trigger caused the interrupt. Example 9.1 illustrates these differences.

**Example 9.1.** Interface two switches and signal associated semaphores when each switch is pressed.

**Solution:** We will assume the switches do not bounce (interfacing switches that bounce will be covered later in the chapter). The semaphore SW1 will be signaled when switch SW1 is pressed, and similarly, semaphore SW2 will be signaled when switch SW2 is pressed. In the first solution, we will use vectored interrupts by connecting one switch to Port C and the other switch to Port E (left side of Figure 9.10). Since the two sources have separate vectors, the switch on Port C will automatically activate **GPIOPortC\_Handler** and switch on Port E will automatically activate **GPIOPortE\_Handler**. The left side of Figures 9.10 and 9.11 show the solution with vectored interrupts.

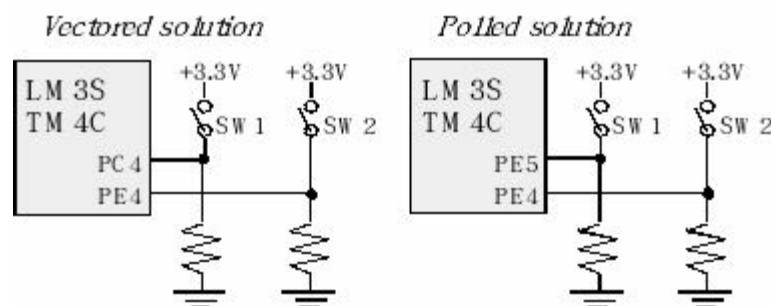


Figure 9.10. Two solutions of switch-triggered interrupts.

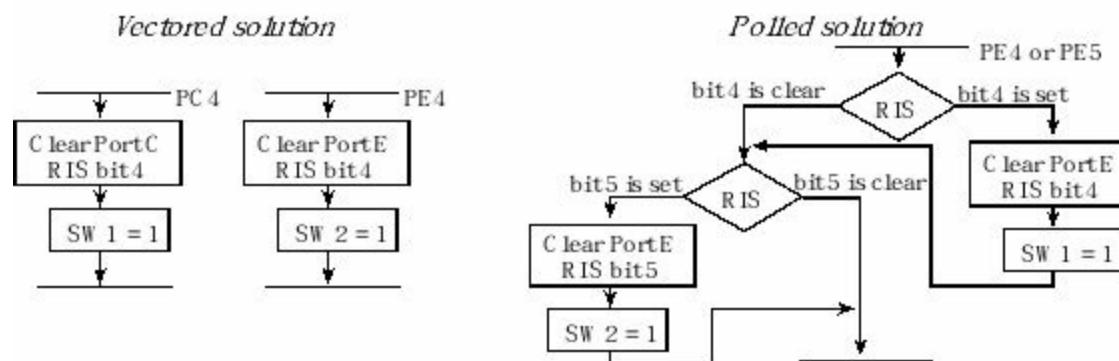


Figure 9.11. Flowcharts for a vectored and polled interrupt.

The software solution using vectored interrupts is in Program 9.5. We initialize two I/O pins as inputs with rising edge interrupt triggers. In this way, we get an interrupt request when the switch is touched. I.e., an interrupt occurs on the 0 to 1 rising edge either of PC4 or PE4. To acknowledge an interrupt we clear the trigger flag. Writing a 0x10 to the interrupt clearregister, **GPIO\_PORTn\_ICR\_R**, will clear bit 4 in the **RIS** register without affecting the other bits in the **RIS** register. Notice that the acknowledgement uses an “=” instead of an “|=” because this register is write-only, such that writing ones to **IC** register will clear corresponding bits in the **RIS** register. Writing zeros to the **IC** register has no effect.

```

volatile uint8_t SW1, SW2; // semaphores
void VectorButtons_Init(void){
    SYSCTL_RCGCGPIO_R |= 0x14; // activate port C and port E
}

```

```

SW1 = 0;          // clear semaphores
SW2 = 0;
GPIO_PORTC_AMSEL_R &= ~0x10; // disable analog function on PC4
GPIO_PORTC_PCTL_R &= ~0x000F0000; // configure PC4 as GPIO
GPIO_PORTC_DIR_R &= ~0x10; // make PC4 in
GPIO_PORTC_AFSEL_R &= ~0x10; // disable alt funct on PC4
GPIO_PORTC_DEN_R |= 0x10; // enable digital I/O on PC4
GPIO_PORTC_IS_R &= ~0x10; // PC4 is edge-sensitive
GPIO_PORTC_IBE_R &= ~0x10; // PC4 is not both edges
GPIO_PORTCIEV_R |= 0x10; // PC4 rising edge event
GPIO_PORTC_ICR_R = 0x10; // clear flag4
GPIO_PORTC_IM_R |= 0x10; // enable interrupt on PC4
NVIC_PRI0_R = (NVIC_PRI0_R&0xFF00FFFF)|0x00400000; // priority 2

GPIO PORTE_AMSEL_R &= ~0x10; // disable analog function on PE4
GPIO PORTE_PCTL_R &= ~0x000F0000; // configure PE4 as GPIO
GPIO PORTE_DIR_R &= ~0x10; // make PE4 in
GPIO PORTE_AFSEL_R &= ~0x10; // disable alt funct on PE4
GPIO PORTE_DEN_R |= 0x10; // enable digital I/O on PE4
GPIO PORTE_IS_R &= ~0x10; // PE4 is edge-sensitive
GPIO PORTE_IBE_R &= ~0x10; // PE4 is not both edges
GPIO PORTEIEV_R |= 0x10; // PE4 rising edge event
GPIO PORTE_ICR_R = 0x10; // clear flag4
GPIO PORTE_IM_R |= 0x10; // enable interrupt on PE4
NVIC_PRI1_R = (NVIC_PRI1_R&0xFFFFFFF0)|0x00000040; // priority 2
NVIC_EN0_R = 0x00000014; // enable interrupts 2 and 4 in NVIC
EnableInterrupts();
}

void GPIOPortC_Handler(void){
    GPIO_PORTC_ICR_R = 0x10; // acknowledge flag4
    SW1 = 1; // signal SW1 occurred
}

void GPIOPortE_Handler(void){
    GPIO PORTE_ICR_R = 0x10; // acknowledge flag4
    SW2 = 1; // signal SW2 occurred
}

```

Program 9.5. Example of a vectored interrupt (TwoButtonVector\_xxx.zip).

The right sides of Figures 9.10 and 9.11 show the solution with polled interrupts. Touching either switch will cause a Port E interrupt. The ISR must poll to see which one or possibly both caused the interrupt. Fortunately, even though they share a vector, the acknowledgements are separate. The code **GPIO\_PORTE\_ICR\_R=0x10**; will clear bit 4 in the status register without affecting bit 5, and the code **GPIO\_PORTE\_ICR\_R=0x20**; will clear bit 5 in the status register without affecting bit 4. This means the timing of one switch does not affect whether or not pushing the other switch will signal its semaphore. On the other hand, whether we are using polled or vectored interrupt, because there is only one processor, the timing of one interrupt may delay the servicing of another interrupt.

The polled solution is Program 9.6. It takes three conditions to cause an interrupt. 1) The PE4 and PE5 are armed in the initialization; 2) The LM3S/TM4C1s enabled for interrupts with the **EnableInterrupts()** function; 3) The trigger **GPIO\_PORTE\_RIS\_R** is set on the rising edge of PE4 or the trigger **GPIO\_PORTE\_RIS\_R** is set on the rising edge of PE5. Because the two triggers have separate acknowledgments, if both triggers are set, both will get serviced. Furthermore, the polling sequence does not matter.

```

volatile uint8_t SW1, SW2; // semaphores
void PolledButtons_Init(void){
    SYSCTL_RCGCGPIO_R |= 0x10; // activate port E
    SW1 = 0; // clear semaphores
    SW2 = 0;
    GPIO_PORTE_AMSEL_R &=~0x30;// disable analog function on PE5-4
    GPIO_PORTE_PCTL_R &=~0x00FF0000; // configure PE5-4 as GPIO
    GPIO_PORTE_DIR_R &=~0x30; // make PE5-4 in
    GPIO_PORTE_AFSEL_R &=~0x30;// disable alt funct on PE5-4
    GPIO_PORTE_DEN_R |= 0x30; // enable digital I/O on PE5-4
    GPIO_PORTE_IS_R &=~0x30; // PE5-4 is edge-sensitive
    GPIO_PORTEIBE_R &=~0x30; // PE5-4 is not both edges
    GPIO_PORTEIEV_R |= 0x30; // PE5-4 rising edge event
    GPIO_PORTE_ICR_R = 0x30; // clear flag5-4
    GPIO_PORTE_IM_R |= 0x30; // enable interrupt on PE5-4
    NVIC_PRI1_R = (NVIC_PRI1_R&0xFFFFF00)|0x00000040; // priority 2
    NVIC_EN0_R = 0x00000010; // enable interrupt 4 in NVIC
    EnableInterrupts();
}
void GPIOPortE_Handler(void){
    if(GPIO_PORTE_RIS_R&0x10){ // poll PE4
        GPIO_PORTE_ICR_R = 0x10; // acknowledge flag4
        SW1 = 1; // signal SW1 occurred
    }
    if(GPIO_PORTE_RIS_R&0x20){ // poll PE5
        GPIO_PORTE_ICR_R = 0x20; // acknowledge flag5
        SW2 = 1; // signal SW2 occurred
    }
}
```

}

## Program 9.6. Example of a polled interrupt (TwoButtonPoll\_xxx.zip).

One of the problems with switches is called **switch bounce**. Many inexpensive switches will mechanically oscillate for up to a few milliseconds when touched or released. It behaves like an underdamped oscillator. These mechanical oscillations cause electrical oscillations such that a port pin will oscillate high/low during the bounce. In some cases this bounce should be removed. To remove switch bounce we can ignore changes in a switch that occur within 10 ms of each other. In other words, recognize a switch transition, disarm interrupts for 10ms, and then rearm after 10 ms. Alternatively, we could record the time of the switch transition. If the time between this transition and the previous transition is less than 10ms, ignore it. If the time is more than 10 ms, then accept and process the input as a real event.

# 9.6. SysTick Periodic Interrupts

One application of periodic interrupts is called “intermittent polling” or “periodic polling”. Figure 9.12 shows busy wait side by side with periodic polling. In busy-wait synchronization, the main program polls the I/O devices continuously. With periodic polling, the I/O devices are polled on a regular basis (established by the periodic interrupt.) If no device needs service, then the interrupt simply returns.

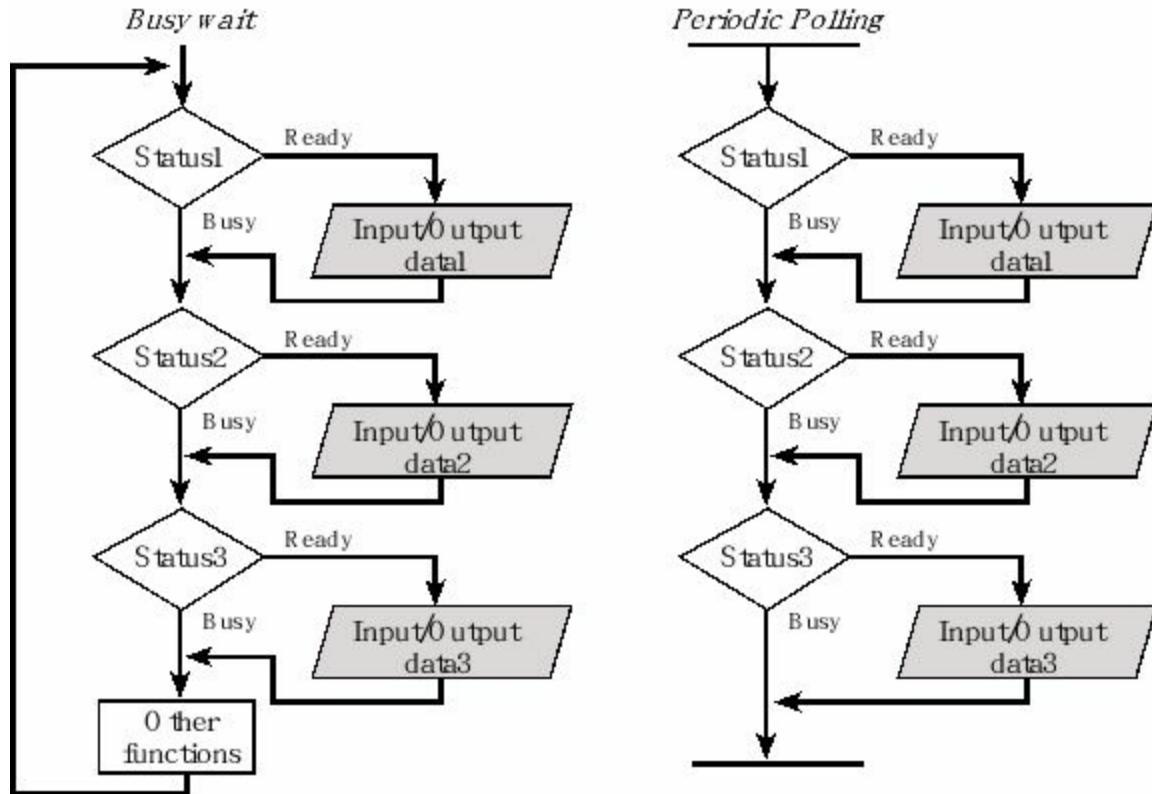


Figure 9.12. On the left is busy-wait, and on the right is periodic polling.

If the polling period is  $\Delta t$ , then on average the interface latency will be  $\frac{1}{2}\Delta t$ , and the worst case latency will be  $\Delta t$ . Periodic polling is appropriate for low bandwidth devices where real-time response is not necessary. This method frees the main program from the I/O tasks. We use periodic polling if the following two conditions apply:

1. The I/O hardware cannot generate interrupts directly
2. We wish to perform the I/O functions in the background

For a data acquisition system, it is important to establish an accurate sampling rate. The time in between ADC samples must be equal and known in order for the digital signal processing to function properly. Similarly for microcontroller-based control systems, it is important to maintain both the ADC and DAC timing. In the next section, we will see the general purpose timers can also create periodic interrupts.

The SysTick timer is a simple way to create periodic interrupts. A periodic interrupt is one that is requested on a fixed time basis. This interfacing technique is required for data acquisition and control systems, because software servicing must be performed at accurate time intervals.

Table 9.6 shows the SysTick registers used to create a periodic interrupt. SysTick has a 24-bit counter that decrements at the bus clock frequency. Let  $f_{BUS}$  be the frequency of the bus clock, and let  $n$  be the value of the **RELOAD** register. The frequency of the periodic interrupt will be  $f_{BUS}/(n+1)$ . First, we clear the **ENABLE** bit to turn off SysTick during initialization. Second, we set the **RELOAD** register. Third, we write to the **NVIC\_ST\_CURRENT\_R** value to clear the counter. Lastly, we write the desired mode to the control register, **NVIC\_ST\_CTRL\_R**. We must set **CLK\_SRC=1**, because **CLK\_SRC=0** external clock mode is not implemented on the LM3S/TM4C family. We set **INTEN** to enable interrupts. We establish the priority of the SysTick interrupts using the **TICK** field in the **NVIC\_SYS\_PRI3\_R** register. We need to set the **ENABLE** bit so the counter will run. When the **CURRENT** value counts down from 1 to 0, the **COUNT** flag is set. On the next clock, the **CURRENT** is loaded with the **RELOAD** value. In this way, the SysTick counter (**CURRENT**) is continuously decrementing. If the **RELOAD** value is  $n$ , then the SysTick counter operates at modulo  $n+1$  (... $n$ ,  $n-1$ ,  $n-2$  ... 1, 0,  $n$ ,  $n-1$ , ...). In other words, it rolls over every  $n+1$  counts. Thus, the **COUNT** flag will be configured to trigger an interrupt every  $n+1$  counts. Program 9.7 shows a simple example of SysTick. SysTick is the only interrupt on the LM3S/TM4C that has an automatic acknowledge. Notice there is no explicit software step in the ISR to clear the trigger flag.

Address	31-	23-	16	15-3	2	1	0	Name
	24	17						
\$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
\$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
\$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

Address	31-29	28-	23-21	20-	7-5	4-0	Name
	24			8			
\$E000ED20	TICK	0	PENDSV	0	DEBUG	0	NVIC_SYS_PRI3_R

**Table 9.6. SysTick registers.**

```

volatile uint32_t Counts;
#define PD0 (*((volatile uint32_t *)0x40007004))
void SysTick_Init(uint32_t period){
    SYSCTL_RCGCGPIO_R |= 0x08; // activate port D
    Counts = 0;
    GPIO_PORTD_AMSEL_R &=~0x01; // no analog
    GPIO_PORTD_PCTL_R &=~0x0000000F; // regular GPIO function
    GPIO_PORTD_DIR_R |= 0x01; // make PD0 out
    GPIO_PORTD_AFSEL_R &=~0x01; // disable alt funct on PD0
    GPIO_PORTD_DEN_R |= 0x01; // enable digital I/O on PD0
    NVIC_ST_CTRL_R = 0; // disable SysTick during setup
    NVIC_ST_RELOAD_R = period - 1;// reload value
    NVIC_ST_CURRENT_R = 0; // any write to current clears it
    NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R&0x00FFFFFF)|0x40000000; //priority 2
    NVIC_ST_CTRL_R = 0x00000007; // enable with core clock and interrupts
    EnableInterrupts();
}

```

```
void SysTick_Handler(void){  
    PD0 ^= 0x01;      // toggle PD0  
    Counts = Counts + 1;  
}
```

Program 9.7. Implementation of a periodic interrupt using SysTick  
(PeriodicSysTickInts\_xxx.zip).

# 9.7. Timer Periodic Interrupts

The LM3S/TM4C microcontrollers have timers that are separate and distinct from SysTick, see Figure 9.14. In periodic timer mode the timer is configured as a 16-bit down-counter with an optional 8-bit prescaler that effectively extends the counting range of the timer to 24 bits. The timer is compared to a preloaded constant, and when the timer equals the constant a trigger flag is set and the output pin is inverted. We select periodic timer mode by setting the 2-bit TAMR (or TBMR) field of the **TIMER0\_TAMR\_R** (or **TIMER0\_TBMR\_R**) to 0x02. If we set this field to 0x01, the timer is in one shot mode. In periodic mode the timer runs continuously, and in one shot mode, it runs once and stops. The periodic mode can also be used to create pulse width modulated outputs.

We will use output compare to create time delays, trigger a periodic interrupts, and control ADC sampling. We will also use output compare together with input capture to measure frequency. Output compare and input capture can also be combined to measure period and frequency over a wide range of ranges and resolutions. We may run the output compare modes with or without an external output pin attached.

Each periodic timer module has

An external output pin, e.g., CCP0,

A flag bit, e.g., TATORIS

A control bit to connect the output to the ADC as a trigger, e.g., TAOTE,

An interrupt arm bit, e.g., TATOIM

A 16-bit output compare register, e.g., **TIMER0\_TAILR\_R**

An 8-bit prescaleregister, e.g., **TIMER0\_TAPR\_R**

An 8-bit prescale match register, e.g., **TIMER0\_TAPMR\_R**

The members of the LM3S/TM4C family have varying number of timers, see Figure 9.13. When designing a system using the timers, you will need to consult the datasheet for your particular microcontroller. In particular, some of the channels do not have an associated output pin. The TM4C123 has six timers, and the TM4C1294 has eight timers. See Tables 4.3, 4.4 to see which I/O pins the TM4C123/TM4C1294 uses for the timers.

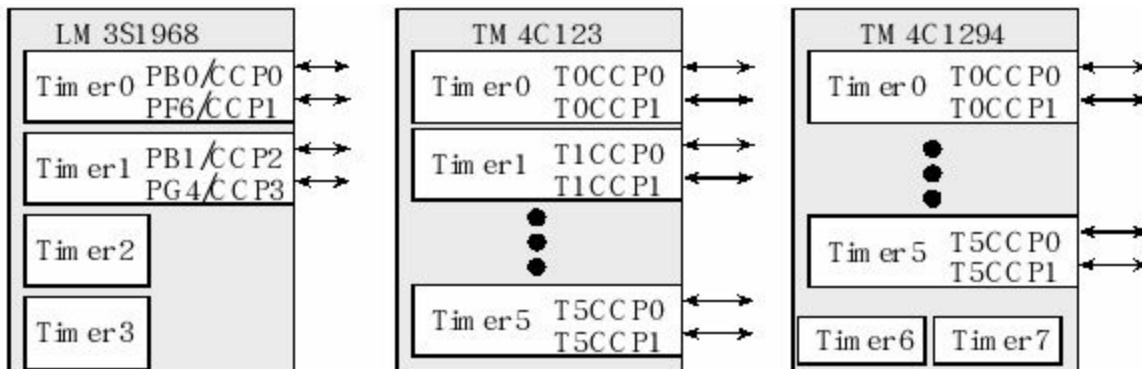


Figure 9.13. Timer pins on the LM3S/TM4C microcontroller are labeled CCP.

To connect the output pin to the timer, we must set the alternative function bit for that pin. The output compare pin is an output of the computer, hence can be used for debugging or to control an external device. An output compare event occurs, changing the state of the output pin, when the 16-bit timer matches the 16-bit **TIMER0\_TAILR\_R** register. The timer will be used without the output pin if the corresponding alternative function bit is clear.

The output compare event occurs when a timer counts down to zero. The timer mode specifies what effect the output compare event will have on the output pin or the rest of the system. If the timer is in one-shot or periodic timer mode, the **TATORIS** (or **TBTORIS**) bit of the Raw Interrupt Status register (**TIMER0\_RIS\_R**) is set. If the arm bit **TATOIM** (or **TBTOIM**) in the **TIMER0\_IMR\_R** register is set, a timer interrupt is requested. The hardware can also trigger an ADC conversion at this time. If the timer is in one-shot mode, it stops counting after the first output compare event. In periodic timer mode, the timer continues counting indefinitely until explicitly disabled by clearing the **TAEN** (or **TBEN**) enable bit in the **TIMER0\_CTL\_R** register. Just like the input capture, the output compare flag is cleared by writing a 1 to its corresponding bit in the Interrupt Clear Register (**TIMER0\_ICR\_R**).

One simple application of output compare is to create a fixed time delay. Let **delay** be the number of bus cycles you wish to wait, up to 65,535. The steps to create the delay are:

- 0) Enable the General-Purpose Timer Module in of **SYSCTL\_RCGCTIMER\_R**
- 1) Ensure that the timer is disabled before making any changes (clear **TAEN**)
- 2) Put the timer module in 16-bit mode by writing 0x4 to **TIMER0\_CFG\_R**
- 3) Write 0x1 to **TAMR**
- 4) Load the desired **delay** into **TIMER0\_TAILR\_R**
- 5) Write a 1 to **CATOCINT** of **TIMER0\_ICR\_R** to clear the time-out flag
- 6) Set the **TAEN** bit to start the timer and begin counting down from **delay**
- 7) Poll **TATORIS** of **TIMER0\_RIS\_R**, wait is over when this bit is set

A second application of output compare is to create a periodic interrupt. Let **prescale** be an 8-bit number loaded into **TIMER0\_TAPR\_R**. The timer frequency will be bus frequency divided by **prescale**+1. The default prescale is 0, meaning the timer frequency equals the bus frequency. Let **period** be the 32-bit value loaded into **TIMER0\_TAILR\_R**. The steps to create the periodic interrupt are:

- 0) Enable the General-Purpose Timer Module in of **SYSCTL\_RCGCTIMER\_R**
- 1) Ensure that the timer is disabled before making any changes (clear **TAEN**)
- 2) Put the timer module in 32-bit mode by writing 0x00 to **TIMER0\_CFG\_R**
- 3) Write 0x2 to **TAMR** to configure for periodic mode
- 4) Load **period** into **TIMER0\_TAILR\_R**
- 5) Load **prescale** into **TIMER0\_TAPR\_R**
- 6) Write a 1 to **CATOCINT** of **TIMER0\_ICR\_R** to clear the time-out flag
- 7) Write a 1 to **TATOIEN** of **TIMER0\_IMR\_R** to arm the time-out interrupt
- 8) Set the priority in the correct NVIC Priority register

- 9) Enable the correct interrupt in the correct NVIC Interrupt Enable register  
 10) Set the **TAEN** bit to start the timer and begin counting down from **period**

If the bus period is  $\Delta t$ , then the timer interrupt period will be

$$\Delta t * (\text{prescale} + 1) * (\text{period} + 1).$$

A few cycles of instructions should separate Steps 0 and 1 to ensure that the timer is receiving a clock before the program attempts to use it. Move Step 0 earlier in your program or insert dummy instructions between Steps 0 and 1 if you get a Hardware Fault. The maximum **period** can be 32 bits without the prescaler. Resolution is 1 bus cycle. Please check the errata for your microcontroller. Many LM3S microcontrollers have hardware bugs associated with 32-bit timer modes. However, we will use 32-bit mode on the TM4C microcontrollers.

**Checkpoint 9.7:** When is TATORIS trigger flag set in periodic timer mode?

**Example 9.2.** Design a system to execute a user task at a periodic rate with units of  $\mu\text{s}$ .

**Solution:** We will generate a periodic interrupt and call the user task from the ISR. Assuming a 80 MHz bus clock, we multiply the period by 80 to match the units of 1  $\mu\text{s}$ . To define the user task we will create a private global variable containing a pointer to the user's function. We will set the variable during initialization and call that function at run time. Another name for a dynamically set function pointer is a **hook**. The **TIMER0\_TAILR\_R** is 32 bits, so the maximum interrupt period is  $12.5\text{ns} * 2^{32}$ , which is about 53 seconds.

**void (\*PeriodicTask)(void); // user function**

The initialization sequence follows the 1 – 10 outline listed above (Program 9.8).

```
void Timer3_Init(void(*task)(void), uint32_t period){
  SYSCTL_RCGCTIMER_R |= 0x0008; // 0) activate timer3
  PeriodicTask = task; // user function (also delay)
  TIMER3_CTL_R = 0x00000000; // 1) disable timer3 during setup
  TIMER3_CFG_R = 0x00000000; // 2) configure for 32-bit timer mode
  TIMER3_TAMR_R = 0x00000002; // 3) configure for periodic mode
  TIMER3_TAILR_R = (period*80)-1; // 4) reload value
  TIMER3_TAPR_R = 0; // 5) 12.5ns timer3
  TIMER3_ICR_R = 0x00000001; // 6) clear timer3 time out flag
  TIMER3_IMR_R |= 0x00000001; // 7) arm time out interrupt
  NVIC_PRI8_R = (NVIC_PRI8_R & 0x00FFFFFF) | 0x40000000; // 8) priority 2
  NVIC_EN1_R = 1 << (35-32); // 9) enable IRQ 35 in NVIC
  TIMER3_CTL_R |= 0x00000001; // 10) enable timer3
  EnableInterrupts();
}

void Timer3A_Handler(void){
  TIMER3_ICR_R = 0x00000001; // acknowledge timer3 time out
```

```
(*PeriodicTask)(); // execute user task  
}
```

## Program 9.8. Implementation of a periodic interrupt using Timer3A (PeriodicTimer0AInts\_xxx.zip).

---

**Example 9.3.** Design an interface 32 Ω speaker and use it to generate a loud 1 kHz sound.

**Solution:** At 3V, a 32 Ω speaker will require a current of about 100 mA. We will use the 2N2222 circuit in Figure 8.16 because it can sink at least three times the current needed for this speaker. In this example the interface will be connected to PA5 (any port pin could have been used). We select a +3.3V supply and connect it to the +V in the circuit. The needed base current is  $I_b = I_{coil}/h_{fe} = 100\text{mA}/100 = 1.0\text{mA}$ .

The desired interface resistor is  $R_b \leq (V_{OH} - V_{be})/I_b = (3.3 - 0.6)/1.0\text{mA} = 2.7\text{k}\Omega$ . To cover the variability in  $h_{fe}$ , we will use a 1.5 kΩ resistor instead of the 2.7 kΩ. The actual voltage on the speaker when active will be  $+3.3 - 0.3 = 3\text{V}$ . We can make the sound quieter by using a larger resistor for  $R_b$ . To generate the 1 kHz sound we need a 1 kHz square wave. There are many good methods to generate square waves. In this example we will implement one of the simplest methods: period interrupt and toggle an output pin in the ISR. To generate a 1 kHz wave we will toggle the PA5 pin every 500 μs. Assuming a 80 MHz crystal and no prescale, **TIMER3\_TAILR\_R** value will equal  $500*80-1 = 39999$ . See Program 9.9.

```
#define PA5 (*((volatile uint32_t *)0x40004080))  
void PA5toggle(void){  
    PA5 = PA5^0x20; // make output sound  
}  
void Sound_Init(void){  
    SYSCTL_RCGCGPIO_R |= 0x01; // 1) activate clock for Port A  
    while((SYSCTL_PGPIO_R&0x01) == 0){};  
    GPIO_PORTA_AMSEL_R &= ~0x20; // no analog  
    GPIO_PORTA_PCTL_R &= ~0x00F00000; // regular GPIO function  
    GPIO_PORTA_DIR_R |= 0x20; // make PA5 out  
    GPIO_PORTA_AFSEL_R &= ~0x20; // disable alt funct on PA5  
    GPIO_PORTA_DEN_R |= 0x20; // enable digital I/O on PA5  
    Timer3_Init(&PA5toggle,500); // Program 9.8 (counts at 1us)  
}
```

---

Program 9.9. Sound output using a periodic interrupt.

**Observation:** To make a quieter sound, we could use a larger resistor between the PA5 output and the 2N2222 base.

## 9.8. Hardware debugging tools

Microcomputer related problems often require the use of specialized equipment to debug the system hardware and software. Two very useful tools are the **logic analyzer** and the oscilloscope. A logic analyzer is essentially a multiple channel digital storage scope with many ways to trigger, see Figure 9.14. As a troubleshooting aid, it allows the experimenter to observe numerous digital signals at various points in time and thus make decisions based upon such observations. As with any debugging process, it is necessary to select which information to observe out of a vast set of possibilities. Any digital signal in the system can be connected to the logic analyzer. Figure 9.14 shows an 8-channel logic analyzer, but real devices can support 128 or more channels. One problem with logic analyzers is the massive amount of information that it generates. With logic analyzers (similar to other debugging techniques) we must strategically select which signals in the digital interfaces to observe and when to observe them. In particular, the triggering mechanism can be used to capture data at appropriate times eliminating the need to sift through volumes of output. Sometimes there are extra I/O pins on the microcontroller, not needed for the normal operation of the system (shown as the bottom two wires in Figure 9.14). In this case, we can connect the pins to a logic analyzer, and add software debugging instruments that set and clear these pins at strategic times within the software. In this way we can visualize the hardware/software timing.

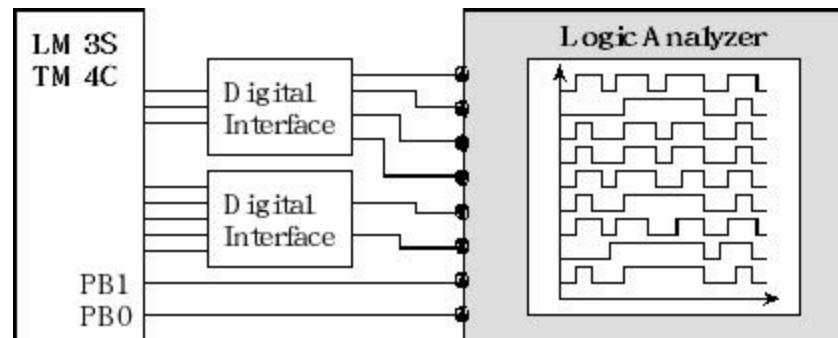


Figure 9.14. A logic analyzer and example output.

An **oscilloscope** can be used to capture voltage versus time data. You can adjust the voltage range and time scale. The oscilloscope trigger is how and when the data will be capture. In normal mode, we measure patterns that repeat over and over, and we use the trigger (e.g., rising edge of channel 1) to freeze the image. In single shot mode, the display is initially blank, and once the trigger occurs, one trace is captured and display.

# 9.9. Profiling

Profiling is similar to performance debugging because both involve dynamic behavior. Profiling is a debugging process that collects the time history of strategic variables. For example if we could collect the time-dependent behavior of the program counter, then we could see the execution patterns of our software. We can profile the execution of a multiple thread software system to detect reentrant activity. We can profile a software system to see which of two software modules is run first. For a real-time system, we need to guarantee the time between when software should be run and when it actually runs is short and bounded. Profiling allows us to measure when software is actually run, experimentally verifying the system is real time.

**Checkpoint 9.8:** Write two friendly debugging instruments, one that sets Port B bit 3 high, and the other makes it low.

**Observation:** Debugging instruments need to save and restore registers so the original function is not disrupted.

## 9.9.1 Profiling using a software dump to study execution pattern

In this section, we will use a software instrument to study the execution pattern of our software. In order to collect information concerning execution we will define a debugging instrument that saves the time and location in an array (like a dump), as shown in Program 9.10. The debugging session will initialize the private global **N** to zero. In this profile, the place **p** will be an integer, uniquely specifying from which place in the software **Profile** is called. The compiled version of **Profile** with optimization level 0 requires about 40 cycles to execute. If the microcontroller is running at 80 MHz, this debugging instrument consumes about 0.5  $\mu$ s per call. This amount of time would usually be classified as minimally intrusive.

```
uint32_t Time[100];
uint32_t Place[100];
uint32_t N;
void Profile(uint32_t p){
    if(N<100){
        Time[N] = NVIC_ST_CURRENT_R; // record time
        Place[N] = p;                // record place
        N++;
    }
}
```

Program 9.10. Debugging instrument for profiling.

Next, we add calls to the debugging instrument at strategic locations in the software, giving a different

number for each place, as shown in Program 9.11. By observing these data, we can determine both a time profile (when=SysTick timer) and an execution profile (where= **p**) of the software execution. From Section 4.7 we previously estimated this function runs in about 100 to 300 cycles. 18 calls to the debugger, each at 40 cycles, will slow down execution by over 700 cycles. Therefore, profiling this program with a dump would be highly intrusive. This profiling method is appropriate for situations where the time between dumps is much longer than 40 cycles.

```
uint32_t sqrt(uint32_t s){ int n; // loop counter
    uint32_t t; // t*t will become s
    Profile(0);
    t = s/16+1; // initial guess
    for(n = 16; n; --n){ // will finish
        Profile(1);
        t = ((t*t+s)/t)/2;
    }
    Profile(2);
    return t;
}
```

Program 9.11. A time/position profile dumping into a data array.

## 9.9.2. Profiling using an Output Port

In this section, we will discuss a hardware/software combination to visualize program activity. Our debugging instrument will set output port bits. We will place these instruments at strategic places in the software. If we are using a regular oscilloscope, then we must stabilize the system so that the function is called over and over. We connect the output pins to an oscilloscope or logic analyzer and observe the program activity. Program 9.12 uses an output port to profile. Assume Port A pins 4 and 5 are initialized as outputs and connected to the logic analyzer or scope. Analysis of the assembly code generated by the compiler shows this code will execute at most 5 cycles. Therefore this is less intrusive than the dump in Program 9.11.

```
#define PA54 (*((volatile uint32_t *)0x400040C0))
uint32_t sqrt(uint32_t s){ int n; // loop counter
    uint32_t t; // t*t will become s
    PA54 = 0x10;
    t = s/16+1; // initial guess
    for(n = 16; n; --n){ // will finish
        PA54 = 0x20;
        t = ((t*t+s)/t)/2;
        PA54 = 0x30;
    }
    PA54 = 0x00;
    return t;
}
```

Program 9.12. A time/position profile using two output bits.

### 9.9.3. \*Thread Profile

When more than one thread is active, you could use the previous technique to visualize the thread that is currently running. For each thread, we assign an output pin. The debugging instrument would set the corresponding bit high when the thread starts and clear the bit when the thread stops. We would then connect the output pins to a multiple channel scope or logic analyzer to visualize in real time the thread that is currently running. Program 9.13 shows a simple thread profile of a system with a foreground thread (main program) and two background threads (SysTick and Timer). Three Port A outputs are used to visualize execution

PA5 will toggle when the software is running in the foreground

PA4 will pulse high then low when executing the SysTick ISR

PA3 will pulse high then low when executing the Timer ISR

The SysTick and Timer initializations were shown previously in Program 9.7 and 9.8 respectively. The timer is running at priority3 and SysTick is running at priority 2 (higher). The debugging instruments are shown in bold.

```
volatile uint32_t Counts;
#define PA5  ((volatile uint32_t *)0x40004080))
#define PA4  ((volatile uint32_t *)0x40004040))
#define PA3  ((volatile uint32_t *)0x40004020))
void Timer0A_Handler(void){
PA3 = 0x08;
    TIMER0_ICR_R = TIMER_ICR_TATOCINT;// acknowledge timer0A timeout
PA3 = 0;
}
void SysTick_Handler(void){
PA4 = 0x10;
    Counts = Counts + 1;
PA4 = 0;
}
int main(void){
    PLL_Init()           // configure for 50 MHz clock
    SYSCTL_RCGCGPIO_R |= 0x01; // 1) activate clock for Port A
    while((SYSCTL_PRGPIO_R&0x01) == 0){};
    GPIO_PORTA_AMSEL_R &= ~0x38; // disable analog function
    GPIO_PORTA_PCTL_R &= ~0x00FFF000; // GPIO
    GPIO_PORTA_DIR_R |= 0x38; // make PA5-3 outputs
    GPIO_PORTA_AFSEL_R &= ~0x38; // disable alt func on PA5-3
    GPIO_PORTA_DEN_R |= 0x38; // enable digital I/O on PA5-3
    Timer0A_Init(5);        // 200 kHz
    SysTick_Init(304);      // 164 kHz
```

```

EnableInterrupts();
while(1){
    PA5 = PA5^0x20;
}
}

```

Program 9.13. Thread profile using output pins a logic analyzer (Profile\_xxx.zip).

Bit-specific addresses are used so the two accesses to Port A do not interact with each other. The results shown in Figure 9.15 demonstrate the two interrupts occurs periodically (measured with Analog Discovery by Digilent Inc.) Furthermore the results show, most of the time the software is running in the foreground. The time to execute the ISR is short compared to the time between interrupt requests. This represents a good interrupt design. The following labels in Figure 9.15 illustrate these events

- A) The main program is running
- B) A Timer0 interrupt service has begun
- C) A SysTick interrupt service has begun
- D) The SysTick ISR is finished
- E) The Timer0 ISR is finished
- F) The main program resumes

The A-B-C-D-E-F labels illustrate the case when that the SysTick interrupt has suspended the execution of the Timer ISR. The other interrupts on this trace do not overlap.

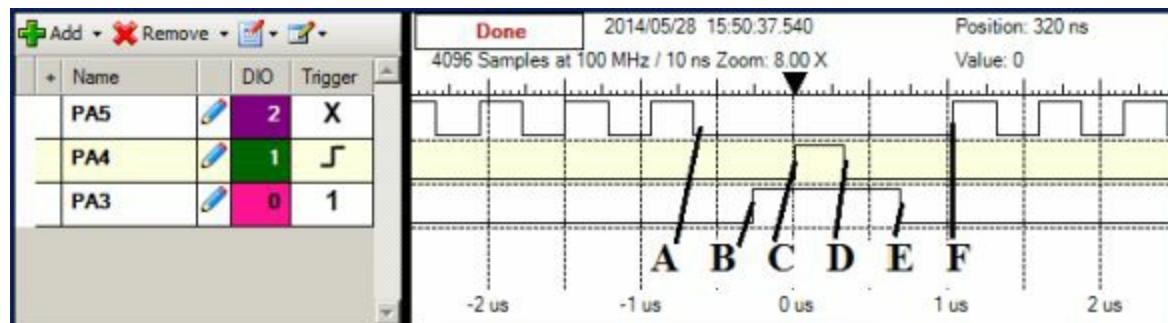


Figure 9.15. Real-time thread profile measured with a logic analyzer. Trigger is set for condition PA3 is high (running Timer0 ISR) and rising edge of PA4 (start of SysTick ISR).

# 9.10. Exercises

**9.1** If you are using SysTick interrupts, how do you set the priority to level 6?

**9.2** If you are using Timer0A interrupts, how do you set the priority to level 5?

**9.3** If you are using edge trigger F and edge trigger G, can you set their priority to the same level? What happens if the two triggers occur at the same time?

**9.4** List the steps automatically occurring in hardware as the context switches from foreground to background.

**9.5** While executing an ISR can the software tell if this trigger suspended the main program or a lower priority interrupt?

**9.6** Explain tail chaining. Explain late arriving interrupt.

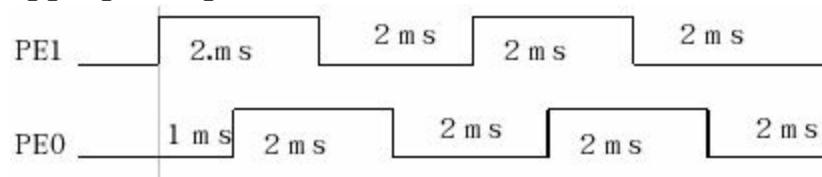
**9.7** What performance specification will degrade if your software sets I=1, executes for 1 ms, and then clears I back to 0?

**D9.8** Create the repeating waveform on PD0 output as shown below. Design the software system using SysTick periodic interrupts. Show all the software for this system: direction registers, global variables, SysTick initialization, main program, and SysTick ISR. The main program initializes the system, and then executes a do-nothing loop. The SysTick ISR performs output to Port D. Please make your code that accesses Port D friendly. Variables you need should be allocated in the appropriate places.



**D9.9** Perform the design described in D9.8 using Timer0A interrupts.

**D9.10** Create the repeating waveform on PE0,PE1 output as shown below. Design the software system using SysTick periodic interrupts. Show all the software for this system: direction registers, global variables, SysTick initialization, main program, and SysTick ISR. The main program initializes the system, and then executes a do-nothing loop. The SysTick ISR performs output to Port E. Please make your code that accesses Port E friendly. Variables you need should be allocated in the appropriate places.



**D9.11** Perform the design described in D9.10 using Timer0A interrupts.

**D9.12** Write interrupting SysTick software that counts a global variable at 1 Hz. Give the initialization, and the ISR.

**D9.13** Write interrupting SysTick software in C or assembly that maintains the time of day. Give the initialization, and the ISR. The initial time of day is passed in when initialization is called. Register R0 contains the initial hour, Register R1 contains the initial minute. Assume the initial seconds are 0. Implement military time, where the hour goes from 0 to 23. Include a function called **Time\_Get** that returns the time in hours:minutes:seconds.

**D9.14** Redesign the FSM in Homework 6.13 in C or assembly to run in the background using Timer interrupts. There are no backward jumps in the ISR.

**D9.15** Redesign the FSM in Homework 6.14 in C or assembly to run in the background using SysTick interrupts. There are no backward jumps in the ISR. Execute the FSM every 2 ms.

**D9.16** Redesign the FSM in Homework 6.15 in C or assembly to run in the background using SysTick interrupts. There are no backward jumps in the ISR. Execute the FSM every 10 ms.

**D9.17** Interface a unipolar stepper motor (5 wires) to the microcontroller. Each coil requires 500 mA at 12V. There are 200 steps per revolution. Write software that spins the motor at 1 rps, using timer interrupts.

**D9.18** Interface a unipolar stepper motor (5 wires) to the microcontroller. Each coil requires 100 mA at 6V. There are 36 steps per revolution. Write software in C or assembly that spins the motor at 10 rps, using Timer0A interrupts.

**D9.19** Interface a bipolar stepper motor (4 wires) to the microcontroller. Each coil requires 500 mA at 12V. There are 200 steps per revolution. Write software in C or assembly that spins the motor at 5 rps, using Timer0A interrupts.

**D9.20** Interface a  $32 \Omega$  speaker (2 wires) to the microcontroller. To make a sound, output a 1 kHz square wave to the interface, creating about 1 V peak-to-peak on the speaker (about 10 mA pulsed current). Use the +3.3V supply and a 2N2222 transistor. Write one subroutine to activate the sound (arm Timer0A interrupts), and a second subroutine to stop the sound (disarm Timer0A). Write in C or assembly.

**D9.21.** The students in a class are specified as an array of structures. Write C code to navigate through the class array and print all student records in the following format: first initial, last initial, id, and teammate's id.

```
struct Student {  
    char Initials[2];  
    int16_t id;  
    struct Student *teammate;  
};  
typedef struct Student SType;
```

```
#define JVpt &class[4]  
#define RVpt &class[5]  
SType class[6] = {  
    {{'X','Y'},123, RSpt}, // XY  
    {{'A','B'}, 23, RVpt}, // AB  
    {{'R','S'}, 11, XYpt}, // RS
```

```
#define XYpt &class[0]          {{'X','Y'}, 44, JVpt}, // CD
#define ABpt &class[1]          {{'A','B'}, 42, CDpt}, // JV
#define RSpt &class[2]          {{'R','Y'},457, ABpt}}; // RY
#define CDpt &class[3]
```

---

# 9.11. Lab Assignments

**Lab 9.1** Traffic light controller. Redesign the traffic light controller from Lab 6.2 to run within the SysTick handler.

**Lab 9.2** Stepper controller. Redesign the stepper motor controller from Lab 8.2 to run within the SysTick handler.

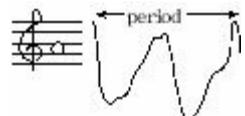
**Lab 9.3** Stop watch. Design, implement and test a stop watch. There should be at least 3 buttons and a display (LCD or OLED).

**Lab 9.4** Alarm clock. Design, implement and test an alarm clock. There should be at least 3 buttons, one buzzer, and a display (LCD or OLED). Have the buttons request edge-triggered interrupts. It will probably be necessary to debounce the switches.

# 10. Analog I/O Interfacing

## Chapter 10 objectives are to:

- Discuss sampling and the Nyquist Theorem,
- Present a simple way to build a DAC,
- Use the DAC to generate sounds and music,
- Present some simple ADC conversion methods,
- Describe the internal ADC on the LM3S/TM4C.



The common theme of this chapter is analog I/O interfacing. The chapter begins with a discussion of representing continuous signals with digital approximations. A digital to analog converter will be used to generate waveforms and sound. This chapter covers some ADC modes built into the microcontroller. The ADC is then used to design measurement systems. A control system includes both inputs and outputs.

# 10.1. Approximating continuous signals in the digital domain

An **analog signal** is one that is continuous in both amplitude and time. Neglecting quantum physics, most signals in the world exist as continuous functions of time in an analog fashion (e.g., voltage, current, position, angle, speed, force, pressure, temperature, and flow etc.) In other words, the signal has an amplitude that can vary over time, but the value cannot instantaneously change. To represent a signal in the digital domain we must approximate it in two ways: amplitude quantizing and time quantizing. From an amplitude perspective, we will first place limits on the signal restricting it to exist between a minimum and maximum value (e.g., 0 to +3V), and second, we will divide this amplitude range into a finite set of discrete values. The **range** of the system is the maximum minus the minimum value. The **precision** of the system defines the number of values from which the amplitude of the digital signal is selected. Precision can be given in number of alternatives, binary bits, or decimal digits. The **resolution** is the smallest change in value that is significant.

Figure 10.1 shows a temperature waveform (solid line), with a corresponding digital representation sampled at 1 Hz and stored as a 5-bit integer number with a range of 0 to 31 °C. Because it is digitized in both amplitude and time, the digital samples (individual dots) in Figure 10.1 must exist at an intersection of grey lines. Because it is a time-varying signal (mathematically, this is called a function), we have one amplitude for each time, but it is possible for there to be 0, 1, or more times for each amplitude.

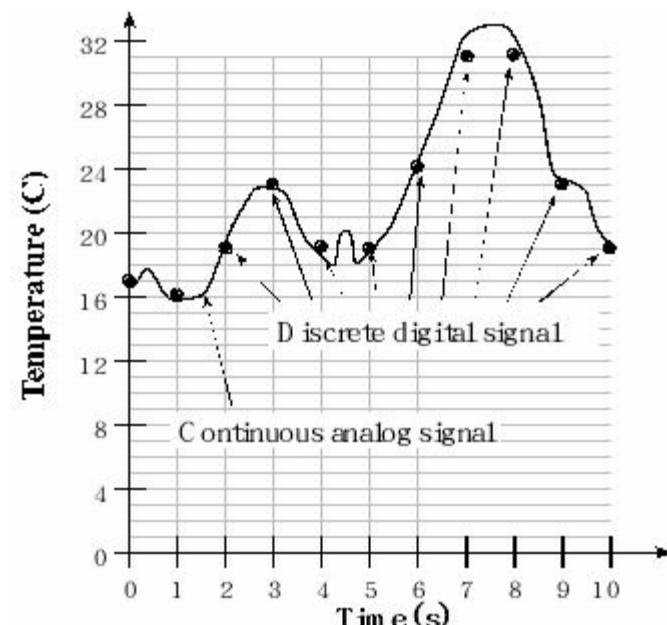


Figure 10.1. An analog signal is represented in the digital domain as discrete samples

The second approximation occurs in the time domain. Time quantizing is caused by the finite sampling interval. For example, the data are sampled every 1 second in Figure 10.1. In practice we will use a periodic timer to trigger an analog to digital converter (ADC) to digitize information, converting from the analog to the digital domain. Similarly, if we are converting from the digital to the analog domain, we use the periodic timer to output new data to a digital to analog converter (DAC). The **Nyquist Theorem** states that if the signal is sampled with a frequency of  $f_s$ , then the digital samples only contain frequency components from 0 to  $\frac{1}{2} f_s$ . Conversely, if the analog signal does contain frequency components larger than  $\frac{1}{2} f_s$ , then there will be an **aliasing** error during the sampling process. Aliasing is when the digital signal appears to have a different frequency than the original analog signal.

**Checkpoint 10.1:** Why can't the digital samples represent the little wiggles in the analog signal?

**Checkpoint 10.2:** Why can't the digital samples represent temperatures above 31 °C?

**Checkpoint 10.3:** What range of frequencies is represented in the digital samples when the ADC is sampled once a second, like Figure 10.1?

**Checkpoint 10.4:** If I wanted to create an analog output wave with frequencies components from 0 to 1000 Hz, what is the slowest rate at which I could output to the DAC?

## 10.2. Digital to Analog Conversion

A DAC converts digital signals into analog form as illustrated in Figure 10.2. Although one can interface a DAC to a regular output port, most DACs are interfaced using high-speed synchronous protocols, like the SSI. The DAC output can be current or voltage. Additional analog processing may be required to filter, amplify or modulate the signal. We can also use DACs to design variable gain or variable offset analog circuits.

The DAC **precision** is the number of distinguishable DAC outputs (e.g., 1024 alternatives, 10 bits). The DAC **range** is the maximum and minimum DAC output (volts, amps). The DAC resolution is the smallest distinguishable change in output. The units of resolution are in volts or amps depending on whether the output is voltage or current. The **resolution** is the change in output that occurs when the digital input changes by 1.

$$\text{Range(volts)} = \text{Precision(alternatives)} \cdot \text{Resolution(volts)}$$

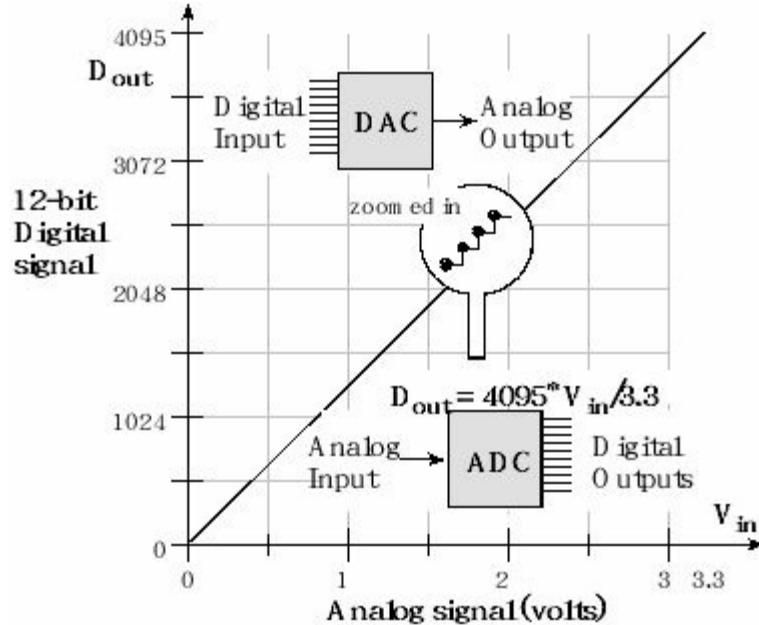


Figure 10.2. A 12-bit DAC provides analog output. A 12-bit ADC provides analog input.

The DAC **accuracy** is  $(\text{Actual} - \text{Ideal}) / \text{Ideal}$  where Ideal is referred to the National Institute of Standards and Technology (NIST). One can choose the full scale **range** of the DAC to simplify the use of fixed-point math. For example, if an 8-bit DAC had a full scale range of 0 to 2.55 volts, then the resolution would be exactly 10 mV. This means that if the DAC digital input were 123, then the DAC output voltage would be 1.23 volts.

**Checkpoint 10.5:** An 8-bit DAC has a range of 0 to 2.5V, what is the approximate resolution?

**Checkpoint 10.6:** You need a DAC with a range of 0 to 2V, and a resolution of 1 mV. What is the smallest number of bits could you use for the DAC?

A DAC **gain error** is a shift in the slope of the  $V_{out}$  versus digital input static response. A DAC **offset error** is a shift in the  $V_{out}$  versus digital input static response. The DAC transient response has three components: delay phase, slewing phase, ringing phase. During the delay phase, the input has changed but the output has not yet begun to change. During the slewing phase, the output changes rapidly. During the ringing phase, the output oscillates while it stabilizes. For purposes of **linearity**, let  $m, n$  be digital inputs, and let  $f(n)$  be the analog output of the DAC, see Figure 10.3. One quantitative measure of linearity is the correlation coefficient of a linear regression fit of the  $f(n)$  responses. If  $\Delta$  is the DAC resolution, it is linear if

$$f(n+1) - f(n) = f(m+1) - f(m) = \Delta \quad \text{for all } n, m$$

The DAC is **monotonic** if

$$\text{sign}(f(n+1) - f(n)) = \text{sign}(f(m+1) - f(m)) \quad \text{for all } n, m$$

Conversely, the DAC is **nonlinear** if

$$f(n+1) - f(n) \neq f(m+1) - f(m) \quad \text{for some } n, m$$

Practically speaking all DACs are nonlinear, but the worst nonlinearity is nonmonotonicity. The DAC is **nonmonotonic** if

$$\text{sign}(f(n+1) - f(n)) \neq \text{sign}(f(m+1) - f(m)) \quad \text{for some } n, m$$

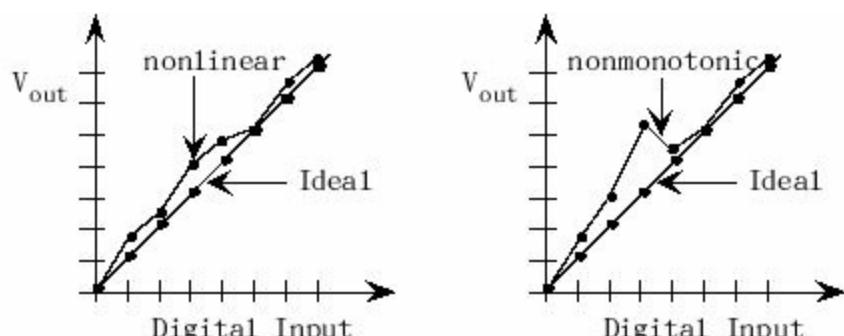


Figure 10.3. Nonlinear and nonmonotonic DACs.

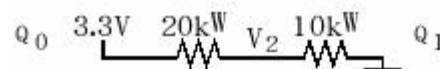
**Example 10.1.** Design a 2-bit binary-weighted DAC with a range of 0 to +3.3V using resistors.

**Solution:** We begin by specifying the desired input/output relationship of the 2-bit DAC. There are two possible solutions depending upon whether we want a resolution of 0.825 V or 1.1 V, as shown as  $V_1$  and  $V_2$  in Table 10.1. Both solutions are presented in Figure 10.4.

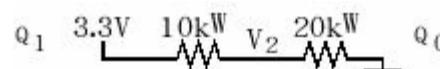
N	$Q_1$	$Q_0$	$V_1$ (V)	$V_2$ (V)
0	<b>0</b>	<b>0</b>	0.000	0.0
1	<b>0</b>	<b>3.3</b>	0.825	1.1
2	<b>3.3</b>	<b>0</b>	1.650	2.2
3	<b>3.3</b>	<b>3.3</b>	2.475	3.3

Table 10.1. Specifications of the 2-bit binary-weighted DAC.

Assume the output high voltage ( $V_{OH}$ ) of the microcontroller is 3.3 V, and its output low voltage ( $V_{OL}$ ) is 0. With a binary-weighted DAC, we choose the resistor ratio to be 2/1 so  $Q_1$  bit is twice as significant as the  $Q_0$  bit, as shown in Figure 10.4. Considering the circuit on the right, if both  $Q_1$  and  $Q_0$  are 0, the output  $V_2$  is zero. If  $Q_1$  is 0 and  $Q_0$  is +3.3V, the output  $V_2$  is determined by the resistor divider network



which is 1.1V. If  $Q_1$  is +3.3V and  $Q_0$  is 0, the output  $V_2$  is determined by the network



which is 2.2V. If both  $Q_1$  and  $Q_0$  are +3.3V, the output  $V_2$  is +3.3V. The output impedance of this DAC is approximately 4 k $\Omega$ , which means it cannot source or sink much current.

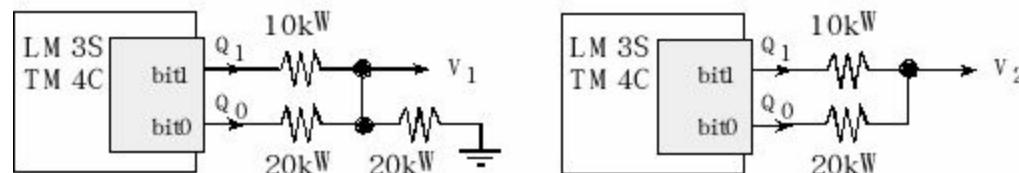


Figure 10.4. Two solutions for a 2-bit binary-weighted DAC.

You can realistically build a 6-bit DAC using the binary-weighted method.

**Checkpoint 10.7:** How do you build a 3-bit binary-weighted DAC using this method?

# 10.3. Music Generation

Most digital music devices rely on high-speed DACs to create the analog waveforms required to produce high-quality sound. In this section, we will discuss a very simple sound generation system that illustrates this application of the DAC. The hardware consists of a DAC and a speaker interface. You can drive headphones directly from a DAC output, but to drive a regular speaker, you will need to add an audio amplifier, as illustrated in Figure 10.5.

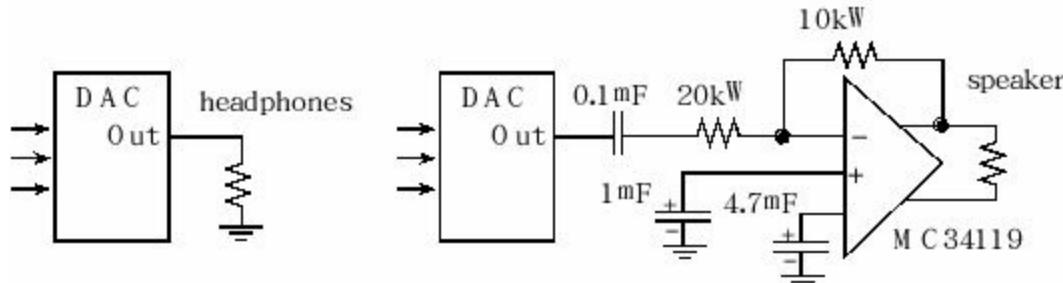


Figure 10.5. DAC allows the software to create music.

For more information on the audio amplifier, refer to the data sheet of the MC34119. To generate sound we need a table of data and a periodic interrupt. Program 10.1 shows C code that defines a 64-element 6-bit sine wave. The **const** modifier will place the data in ROM. The **static** modifier for the variable **i** causes the allocation to be in permanent RAM, with a one-time initialization to 0. The interrupt software will output one value to the DAC. See Figure 10.6. In this example, the 6-bit DAC is interfaced to output pins PB5-0. A 6-bit binary-weighted DAC was made with six resistors having values which were powers of 2, similar to Figure 10.4. To output to this DAC we simply write to Port B. In order to create the sound, it is necessary to output just one number to the DAC each interrupt. The DAC range is 0 to +3.3 V.

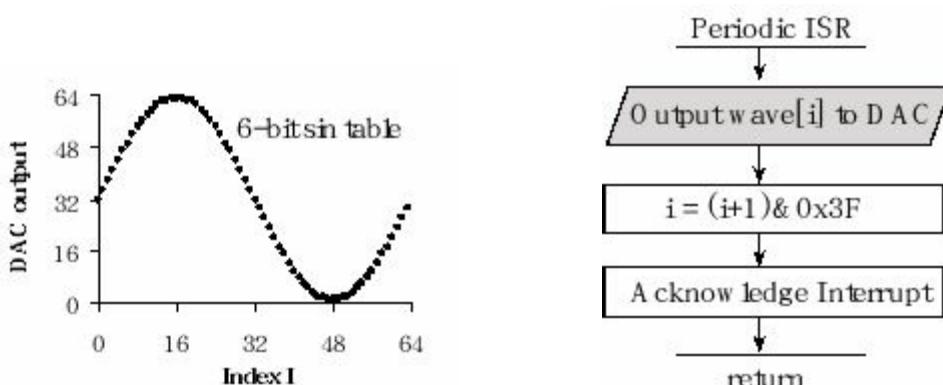


Figure 10.6. A DAC and a periodic interrupt are used to create sound. Output is discrete in time and voltage.

```
const uint8_t wave[64] = {  
    32,35,38,41,44,47,49,52,54,56,58,59,61,62,62,63,63,63,62,62,61,59,58,  
    56,54,52,49,47,44,41,38,35,32,29,26,23,20,17,15,12,10,8,6,5,3,2,2,1,  
    1,1,2,2,3,5,6,8,10,12,15,17,20,23,26,29};
```

```

#define DAC (*((volatile uint32_t *)0x400050FC)) // PB5-0
void Timer3A_Handler(void){
static uint32_t i=0;    // i varies from 0 to 63
    DAC = wave[i];      // output one value each interrupt
    i = (i+1)%0x3F;
    TIMER3_ICR_R = TIMER_ICR_TATOCINT; // ack
}

```

Program 10.1. The periodic interrupt outputs one value to the DAC. The initialization is in Program 9.8.

The quality of the music will depend on both hardware and software factors. The precision of the DAC, external noise, and the dynamic range of the speaker are some of the hardware factors. Software factors include the DAC output rate and the complexity of the stored sound data. If you output a sequence of numbers to the DAC that form a sine wave, then you will hear a continuous tone on the speaker, as shown in Figure 10.7. The **loudness** of the tone is determined by the amplitude of the wave. The **pitch** is defined as the frequency of the wave. Table 10.2 contains frequency values for the notes in one octave. The frequency of the wave,  $f_{\text{sin}}$ , will be determined by the frequency of the interrupt,  $f_{\text{int}}$ , divided by the size of the table n. The size of the table in Program 10.1 is n=64.

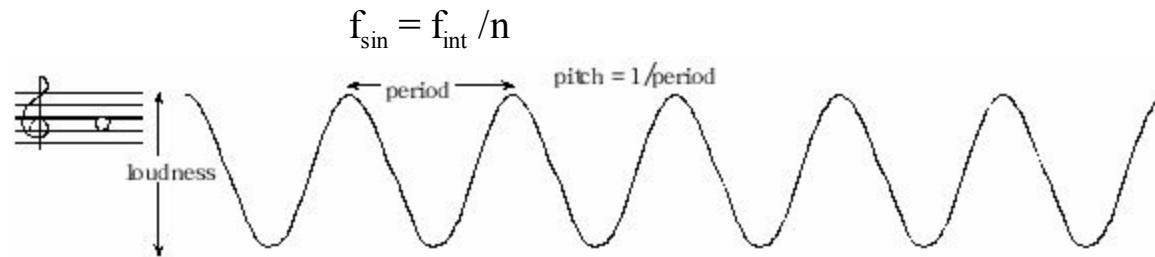


Figure 10.7. The loudness and pitch are controlled by the amplitude and frequency.

The frequency of each note can be calculated by multiplying the previous frequency by  $\sqrt[12]{2}$ . You can use this method to determine the frequencies of additional notes above and below the ones in Table 10.2. There are twelve notes in an octave, therefore moving up one octave doubles the frequency.

Note	frequency
C	523 Hz
B	494 Hz
B <sup>b</sup>	466 Hz
A	440 Hz
A <sup>b</sup>	415 Hz
G	392 Hz
G <sup>b</sup>	370 Hz
F	349 Hz
E	330 Hz
E <sup>b</sup>	311 Hz

D	294 Hz
D <sup>b</sup>	277 Hz
C	262 Hz

**Table 10.2. Fundamental frequencies of standard musical notes. The frequency for ‘A’ is exact.**

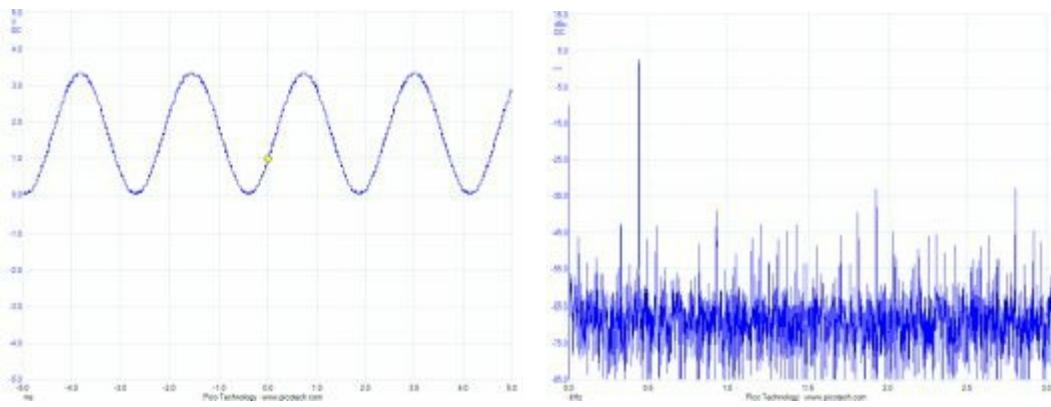


Figure 10.8. A 440Hz sine wave generated with a 6-bit DAC. The plot on the right is the Fourier Transform(frequency spectrum dB versus kHz) of the data plotted on the left.

The measured data in Figure 10.8 was collected using this DAC. The plot on the left was measured with a digital scope (without the headphones being attached). The measured waveform on the left of Figure 10.8 is approximately  $V(t) = 1.6 + 1.6\sin(2\pi 440 t)$  volts. The plot on the right shows the frequency response of this data, plotting amplitude (in dB) versus frequency (in kHz). A **decibel** (dB) is a measure of the relative magnitude of two voltages. Figure 10.8 it compares the input voltage to the full scale voltage using this definition

$$20 \cdot \log_{10}(\text{signal}/\text{fullscale})$$

The two peaks in the spectrum are at DC and 440 Hz. The DC and 440 Hz points are signal, and all the other points in the spectrum are considered noise. The 440 Hz signal has magnitude of about 3 dB and the noise peaks are less than -33 dB. We can calculate the signal to noise

$$\text{db} = 20 \cdot \log_{10}(\text{signal/noise}) = 20 \cdot \log_{10}(\text{signal}/\text{fullscale}) - 20 \cdot \log_{10}(\text{noise}/\text{fullscale})$$

For this data,  $20\log_{10}(\text{signal}/\text{fullscale})$  is +3dB and  $20\log_{10}(\text{noise}/\text{fullscale})$  is -33dB, so we plug this data into the above equation and get  $+36 = 20\log_{10}(\text{signal}/\text{noise})$ . Thus, we calculate  $\text{signal/noise} = 10^{36/20} = 63$ , which is about 6 bits.

Figure 10.9 illustrates the concept of **instrument**. You can define the type of sound by the shape of the voltage versus time waveform. Brass instruments have a very large first harmonic frequency.



Figure 10.9. A waveform shape that generates a trumpet sound.

The **tempo** of the music defines the speed of the song. In 2/4 3/4 or 4/4 music, a **beat** is defined as a quarter note. A moderate tempo is 120 beats/min, which means a quarter note has a duration of  $\frac{1}{2}$  second. A sequence of notes can be separated by pauses (silences) so that each note is heard separately. The **envelope** of the note defines the amplitude versus time. A very simple envelope is illustrated in Figure 10.10. The Cortex<sup>TM</sup>-M processor has plenty of processing power to create these types of waves.

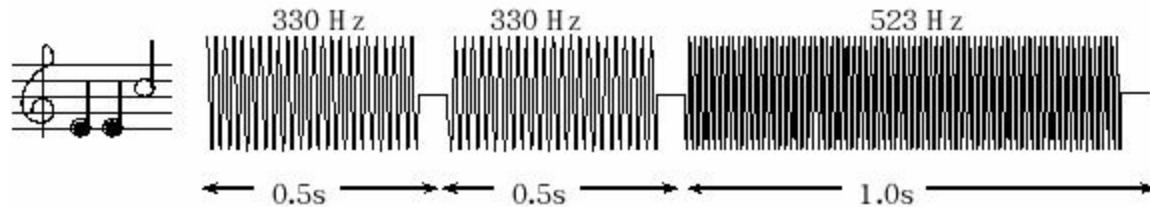


Figure 10.10. You can control the amplitude, frequency and duration of each note (not drawn to scale).

The smooth-shaped envelope, as illustrated in Figure 10.11, causes a less staccato and more melodic sound. This type of sound generation is possible to produce in real time on the Cortex<sup>TM</sup>-M microcontroller.

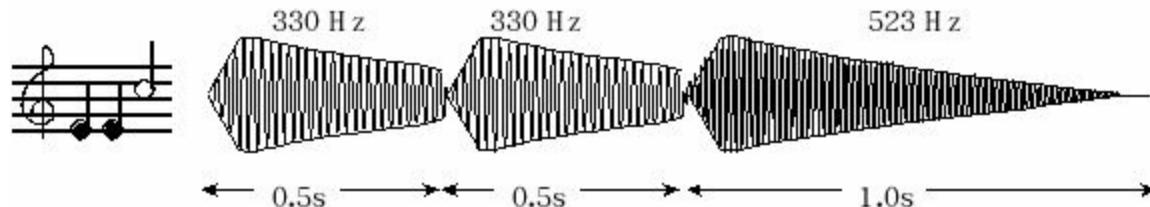


Figure 10.11. The amplitude of a plucked string drops exponentially in time.

A **chord** is created by playing multiple notes simultaneously. When two piano keys are struck simultaneously both notes are created, and the sounds are mixed arithmetically. You can create the same effect by adding two waves together in software, before sending the wave to the DAC. Figure 10.12 plots the mathematical addition of a 262 Hz (low C) and a 392 Hz sine wave (G), creating a simple chord.

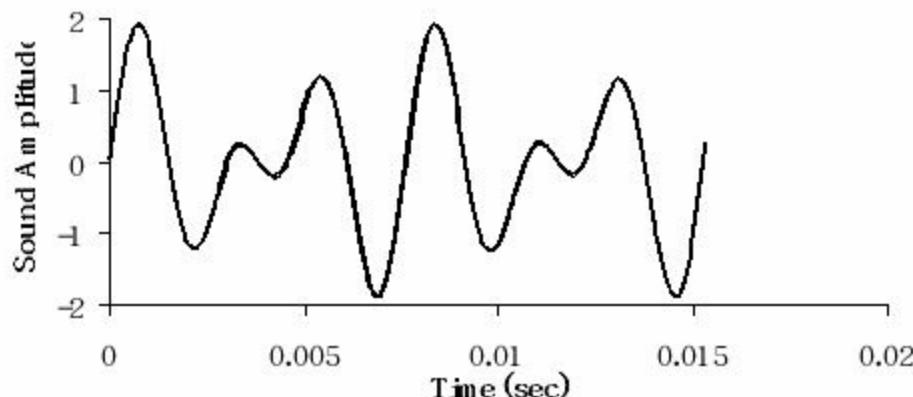


Figure 10.12. A simple chord mixing the notes C and G.



# 10.4. Analog to Digital Conversion

An analog to digital converter (ADC) converts an analog signal into digital form, also shown in Figure 10.2. An embedded system uses the ADC to collect information about the external world (data acquisition system.) The input signal is usually an analog voltage, and the output is a binary number. The ADC precision is the number of distinguishable ADC inputs (e.g., 4096 alternatives, 12 bits). The ADC range is the maximum and minimum ADC input (e.g., 0 to +3.3V). The ADC resolution is the smallest distinguishable change in input (e.g., 0.8 mV). The resolution is the change in input that causes the digital output to change by 1.

$$\text{Range(volts)} = \text{Precision(alternatives)} \cdot \text{Resolution(volts)}$$

Normally we don't specify accuracy for just the ADC, but rather we give the accuracy of the entire system (including transducer, analog circuit, ADC and software). An ADC is **monotonic** if it has no missing codes. This means if the analog signal is a slow rising voltage, then the digital output will hit all values one at a time. The merit of an ADC involves three factors: precision (number of bits), speed (how fast can we sample), and power (how much energy does it take to operate). How fast we can sample involves both the ADC conversion time (how long it takes to convert), and the bandwidth (what frequency components can be recognized by the ADC). The ADC cost is a function of the number and quality of internal components.

## 10.4.1. LM3S/TM4C ADC details

Table 10.3 shows the ADC register bits required to perform sampling on a single channel. For more complex configurations refer to the specific data sheet. The value in **ADC0\_PC\_R** specifies the maximum sampling rate, see Table 10.4. This is not the actual sampling rate. The actual sampling rate is determined by how frequently the ADC is triggered. Refer to the data sheet of your specific microcontroller for maximum possible sampling rate. See Tables 4.3, 4.4 to see which I/O pins the TM4C uses for the ADC analog input channels. On the TM4C, we will need to set bits in the **AMSEL** register to activate the analog interface.

Address	31-2					1	0	Name
\$400F.E638						ADC1	ADC0	SYSCTL_RCGCADC_R
\$4003.8020	31-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0
		SS3		SS2		SS1		SS0
\$4003.8014	31-16			15-12	11-8	7-4	3-0	ADC0_EMUX_R
				EM3	EM2	EM1	EM0	
\$4003.8000	31-4			3	2	1	0	ADC0_ACTSS_R
\$4003.80A0				ASEN3	ASEN2	ASEN1	ASEN0	ADC0_SSMUX3_R
				MUX0				

\$4003.80A4	TS0	IE0	END0	D0	ADC0_SSCTL3_R	
\$4003.8028	SS3	SS2	SS1	SS0	ADC0_PSSI_R	
\$4003.8004	INR3	INR2	INR1	INR0	ADC0_RIS_R	
\$4003.8008	MASK3	MASK2	MASK1	MASK0	ADC0_IM_R	
\$4003.8FC4	Speed				ADC0_PC_R	
	31-12	11-0				
\$4003.80A8	DATA				ADC0_SSFIFO3_R	

**Table 10.3.** The TM4C ADC registers. Each register is 32 bits wide. LM3S has 10-bit data.

The ADC has four sequencers, but we will use only sequencer 3. We set the **ADC0\_SSPRI\_R** register to 0x0123 to make sequencer 3 the highest priority. Because we are using just one sequencer, we just need to make sure each sequencer has a unique priority. We set bits 15–12 (**EM3**) in the **ADC0\_EMUX\_R** register to specify how the ADC will be triggered. Table 10.5 shows the various ways to trigger an ADC conversion. In Volume 2, we will use timer triggering (**EM3**=0x5). However in this first example, we use software start (**EM3**=0x0). The software writes an 8 (**SS3**) to the **ADC0\_PSSI\_R** to initiate a conversion on sequencer 3. Bit 3 (**INR3**) in the **ADC0\_RIS\_R** register will be set when the conversion is complete. We can enable and disable the sequencers using the **ADC0\_ACTSS\_R** register. There are eight ADC channels on the LM3S1968, twelve on the TM4C123, and twenty on the TM4C1294. Which channel we sample is configured by writing to the **ADC0\_SSMUX3\_R** register. The **ADC0\_SSCTL3\_R** register specifies the mode of the ADC sample. We set **TS0** to measure temperature and clear it to measure the analog voltage on the ADC input pin. We set **IE0** so that the **INR3** bit is set on ADC conversion, and clear it when no flags are needed. We will set **IE0** for both interrupt and busy-wait synchronization. When using sequencer 3, there is only one sample, so **END0** will always be set, signifying this sample is the end of the sequence. We set the **D0** bit to activate differential sampling, such as measuring the analog difference between ADC1 and ADC0 pins. In our example, we clear **D0** to sample a single-ended analog input. The **ADC0\_RIS\_R** register has flags that are set when the conversion is complete, assuming the **IE0** bit is set. The **ADC0\_IM\_R** register has interrupt arm bits. The **ADC0\_ISC\_R** register has interrupt trigger bits. The **IN3** bit is set when both **INR3** and **MASK3** are set. We clear the **INR3** and **IN3** bits by writing an 8 to the **ADC0\_ISC\_R** register.

Value	Description
0x7	1M samples/second
0x5	500K samples/second
0x3	250K samples/second
0x1	125K samples/second

**Table 10.4.** The ADC speed bits in the **ADC0\_PC\_R** register.

Value	Event
0x0	Software start
0x1	Analog Comparator 0
0x2	Analog Comparator 1

0x3	Analog Comparator 2
0x4	External (GPIO PB4)
0x5	Timer
0x6	PWM0
0x7	PWM1
0x8	PWM2
0x9	PWM3
0xF	Always (continuously sample)

**Table 10.5. The ADC EM3, EM2, EM1, and EM0 bits in the ADC0\_EMUX\_R register.**

On the LM3S, we can skip steps 1 through 5. We perform the following steps to software start the ADC and sample one channel. Program 10.2 shows a simple initialization of the ADC. It will sample one channel using software start and busy-wait synchronization.

**Step 1.** We enable the port clock for the pin that we will be using for the ADC input.

**Step 2.** Make that pin an input by writing zero to the **DIR** register.

**Step 3.** Enable the alternative function on that pin by writing one to the **AFSEL** register.

**Step 4.** Disable the digital function on that pin by writing zero to the **DEN** register.

**Step 5.** Enable the analog function on that pin by writing one to the **AMSEL** register.

**Step 6.** We enable the ADC clock by setting bit 0 of the **SYSCTL\_RCGCADC\_R** register.

**Step 7.** Bits 3 – 0 of the **ADC0\_PC\_R** register specify the maximum sampling rate of the ADC. In this example, we will sample slower than 125 kHz, so the maximum sampling rate is set at 125 kHz. This will require less power and produce a longer sampling time as described the S/H section, creating a more accurate conversion.

**Step 8.** We will set the priority of each of the four sequencers. In this case, we are using just one sequencer, so the priorities are irrelevant, except for the fact that no two sequencers should have the same priority.

**Step 9.** Before configuring the sequencer, we need to disable it. To disable sequencer 3, we write a 0 to bit 3 (**ASEN3**) in the **ADC0\_ACTSS\_R** register. Disabling the sequencer during programming prevents erroneous execution if a trigger event were to occur during the configuration process.

**Step 10.** We configure the trigger event for the sample sequencer in the **ADC0\_EMUX\_R** register. For this example, we write a 0000 to bits 15–12 (**EM3**) specifying software start mode for sequencer 3.

**Step 11.** For each sample in the sample sequence, configure the corresponding input source in the **ADCSSMUXn** register. In this example, we write the channel number to bits 3–0 in the **ADC0\_SSMUX3\_R** register. In this example, we sample channel 9, which is PE4.

```
void ADC0_InitSWTriggerSeq3_Ch9(void){
```

```

SYSCTL_RCGCGPIO_R |= 0x10; // 1) activate clock for Port E
while((SYSCTL_PRGPIO_R&0x10) == 0){};
GPIO PORTE_DIR_R &= ~0x10; // 2) make PE4 input
GPIO PORTE_AFSEL_R |= 0x10; // 3) enable alternate function on PE4
GPIO PORTE_DEN_R &= ~0x10; // 4) disable digital I/O on PE4
GPIO PORTE_AMSEL_R |= 0x10; // 5) enable analog function on PE4
SYSCTL_RCGCADC_R |= 0x01; // 6) activate ADC0
ADC0_PC_R = 0x01; // 7) configure for 125K
ADC0_SS PRI_R = 0x0123; // 8) Sequencer 3 is highest priority
ADC0_ACTSS_R &= ~0x0008; // 9) disable sample sequencer 3
ADC0_EMUX_R &= ~0xF000; // 10) seq3 is software trigger
ADC0_SSMUX3_R = (ADC0_SSMUX3_R&0xFFFFFFF0) // 11) clear SS3 field
    + 9; // set channel Ain9 (PE4)
ADC0_SSCTL3_R = 0x0006; // 12) no TS0 D0, yes IE0 END0
ADC0_IM_R &= ~0x0008; // 13) disable SS3 interrupts
ADC0_ACTSS_R |= 0x0008; // 14) enable sample sequencer 3
}

```

Program 10.2. Initialization of the ADC using software start and busy-wait (ADCSWTrigger\_xxx.zip).

**Step 12.** For each sample in the sample sequence, we configure the sample control bits in the corresponding nibble in the **ADC0SSCTLn** register. When programming the last nibble, ensure that the **END** bit is set. Failure to set the **END** bit causes unpredictable behavior. Sequencer 3 has only one sample, so we write a 0110 to the **ADC0\_SSCTL3\_R** register. Bit 3 is the **TS0** bit, which we clear because we are not measuring temperature. Bit 2 is the **IE0** bit, which we set because we want the **RIS** bit to be set when the sample is complete. Bit 1 is the **END0** bit, which is set because this is the last (and only) sample in the sequence. Bit 0 is the **D0** bit, which we clear because we do not wish to use differential mode.

**Step 13.** If interrupts are to be used, write a 1 to the corresponding mask bit in the **ADC0\_IM\_R** register. With software start we do not want ADC interrupts, so we clear bit 3.

**Step 14.** We enable the sample sequencer logic by writing a 1 to the corresponding **ASENn**. To enable sequencer 3, we write a 1 to bit 3 (**ASEN3**) in the **ADC0\_ACTSS\_R** register.

Program 10.3 gives a function that performs an ADC conversion. There are four steps required to perform a conversion. The range is 0 to 3V. If the analog input is 0, the digital output will be 0, and if the analog input is 3.3V, the digital output will be 4095.

**Step 1.** The ADC is started using the software trigger. The channel to sample was specified earlier in the initialization.

**Step 2.** The function waits for the ADC to complete by polling the RIS register bit 3.

**Step 3.** The 12-bit digital sample is read out of sequencer 3.

**Step 4.** The RIS bit is cleared by writing to the ISC register.

```

-----ADC0_InSeq3-----
// Busy-wait analog to digital conversion. 0 to 3.3V maps to 0 to 4095
// Input: none
// Output: 12-bit result of ADC conversion
uint32_t ADC0_InSeq3(void){ uint32_t result;
    ADC0_PSSI_R = 0x0008;           // 1) initiate SS3
    while((ADC0_RIS_R&0x08)==0){}; // 2) wait for conversion done
    result = ADC0_SSFIFO3_R&0xFF; // 3) read 12-bit result
    ADC0_ISC_R = 0x0008;           // 4) acknowledge completion
    return result;
}

```

Program 10.3. ADC sampling using software start and busy-wait (ADCSWTrigger\_xxx.zip).

There is software in Volume 2 showing you how to configure the ADC to sample a single channel at a periodic rate using a timer trigger (ADCT0ATrigger\_xxx.zip). The most accurate sampling method is timer-triggered sampling (**EM3**=0x5).

**Checkpoint 10.8:** If the input voltage is 1.0V, what value will the LM3S 10-bit ADC return?

**Checkpoint 10.9:** If the input voltage is 1.0V, what value will the TM4C 12-bit ADC return?

## 10.4.2. ADC Resolution

The ADC resolution is the smallest change in input that can be reliably detected by the system. Figure 10.13 illustrates how ADC resolution should be measured. Because of noise, if we set the ADC input to  $V_{in}$  and sample it many times, we will get a distribution of digital outputs. We plot the number of times we got an output as a function of the output sample. The shape of this response is called a probability density function (pdf) characterizing the noise processes. A pdf plots the number of occurrences versus the ADC sample value. When two pdfs overlap, the two inputs are not distinguishable. If the pdfs do not overlap, we claim the system can resolve the two inputs. For example, white noise has a Gaussian pdf. The standard deviation of repeated measurements (with units of volts) is a simple measure of ADC resolution (in volts). A better measure of resolution would be to repeat the 100 measurements with an input slightly larger,  $V_{in} + \Delta V$ . If we can demonstrate that the second data set is statistically different from the first (regardless of  $V_{in}$ ), we claim the resolution is less than or equal to  $\Delta V$ . For the 12-bit ADC on the TM4C123, Figure 10.13 shows us that we have to increase the input by 1 mV to always be able to recognize the change. For example, the 1.6500V data is statistically different from the 1.6510V data. Therefore, we claim the ADC has a resolution of 1 mV.

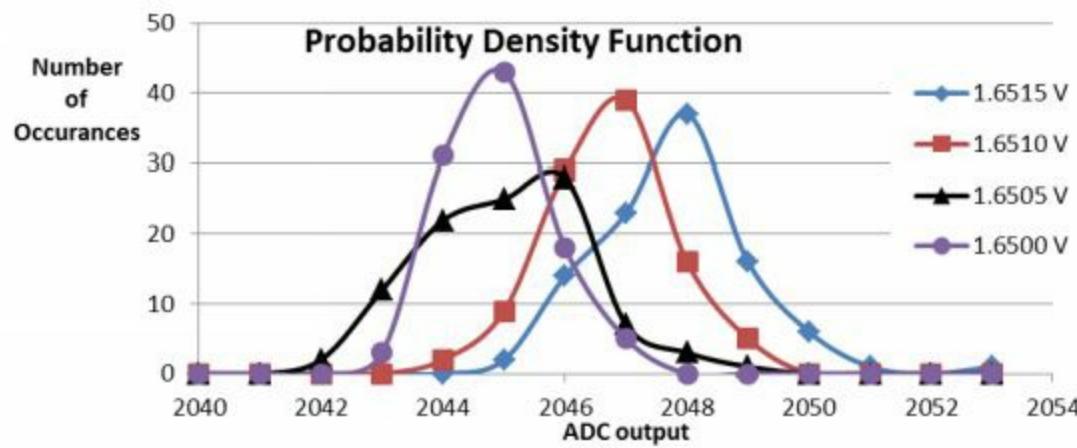


Figure 10.13. A probability density function showing experimental determination of ADC resolution.

**Checkpoint 10.10:** The standard deviation of the data in Figure 10.13 is about 1 ADC sample. Is this the expected result or extremely noisy?

# 10.5. Real-time data acquisition

Whenever we wish convert a continuous analog signal into discrete time digital samples, the rate at which the sampling process occurs is extremely important.

**Nyquist Theorem:** If  $f_{\max}$  is the largest frequency component of the analog signal, then you must sample more than twice  $f_{\max}$  in order to faithfully represent the signal in the digital samples. For example, if the analog signal is  $A + B \sin(2 \pi f t + \phi)$  and the sampling rate is greater than  $2f$ , you will be able to determine  $A$ ,  $B$ ,  $f$ , and  $\phi$  from the digital samples.

The goal of a data acquisition system is to sample the ADC at a regular rate. Let  $f_s$  be the desired sampling rate, and let  $t_i$  be the actual time the ADC creates sample number  $i$ . In a perfect world, we would like to have

$$(t_i - t_{i-1}) = 1/f_s$$

for all  $i$ . When using periodic interrupts to establish the sampling rate (SysTick or Timer), there are two factors that lead to fluctuations in the sample period. We define time jitter,  $\delta t$ , as the maximum variation in the sample-to-sample time.

$$1/f_s - \delta t < (t_i - t_{i-1}) < 1/f_s + \delta t$$

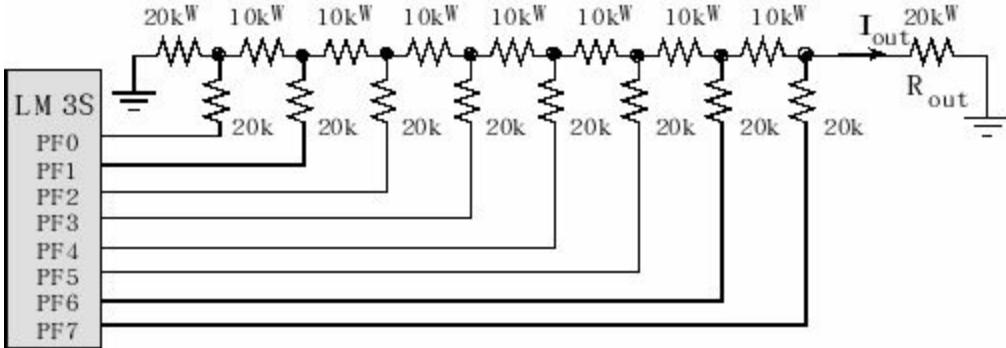
We learned in Chapter 9 that it takes time to process an interrupt (vector fetch, push registers). These cycles, plus the execution of the ISR itself, are equal for every sample. Thus, the time between samples is not affected by this fixed delay. The first factor that does cause jitter is the instruction currently being executed at the time of the interrupt request. The time to execute an instruction on the Cortex™-M processor varies from 1 to 12 cycles. However, other than the divides, most instructions execute in 1, 2, or 3 cycles. We do not know which instruction will be executing or when during that instruction the interrupt will be requested. This uncertainty causes a maximum time jitter of at most 12 cycles, or 240 nsec on a 50 MHz microcontroller. This jitter is usually acceptable. The second source of jitter can be much larger. If there are any portions of the main program that disable interrupts (e.g., because of a critical section), then the time running with interrupts disabled will cause time jitter in the sampling. In a similar fashion, if there are other higher priority interrupts, then the time to execute the other ISR may cause a time jitter. Sampling is an important task that we will assign to a high priority interrupt.

**Observation:** Good software places as little processing in the ISR itself. Perform whatever functions must be done in the ISR, and shift the rest of the processing to the foreground.

**Observation:** Real-time systems must put an upper bound on the time the software is allowed to run with interrupts disabled.

# 10.6. Exercises

**10.1** Consider the 8-bit R-2R resistor ladder shown below. Assume Port F is an output and the digital output voltages from Port F are 0 or +3.3V. Derive a relationship between the 8-bit digital number output to Port F and the current flowing in the resistor labeled  $R_{out}$ . Hint: if one output pin is high and the other pins are low, calculate the current flowing from the pin up through the 20k resistor. Show that this current is the same value regardless of which pin is high (assuming the other pins are low). When a current comes up to a node (drawn with the black dot), it can go one way or another. Again assuming exactly one digital output is high, what happens to currents at each node? I.e., how much goes left and how much goes right? Solve for the basis elements of the 8-bit digital number. I.e., what is  $I_{out}$  if the digital number is 1, 2, 4, 8, 16, 32, 64, and 128? Given the responses for these basis elements, use the law of superposition to derive a general relationship.



**10.2** Assume you have a 12-bit signed ADC. Let  $V_{in}$  be the analog voltage in volts and N be the digital ADC output. The input range of  $-5 \leq V_{in} \leq +5V$ . The ADC digital output range is  $-2048 \leq N \leq +2047$ . First, write a linear equation that relates  $V_{in}$  as a function of N. Next, rewrite the equation in fixed-point math assuming  $V_{in}$  is represented as a decimal fixed-point number with  $\Delta = 0.001$  V.

**10.3** Assume you have an 11-bit signed ADC. Let  $V_{in}$  be the analog voltage in volts and N be the digital ADC output. The input range of  $-10 \leq V_{in} \leq +10V$ . The ADC digital output range is  $-1024 \leq N \leq +1023$ . First, write a linear equation that relates  $V_{in}$  as a function of N. Next, rewrite the equation in fixed-point math assuming  $V_{in}$  is represented as a decimal fixed-point number with  $\Delta = 0.01$  V.

**10.4** An embedded system will use an ADC to measure a parameter. The measurement system range is 0.0 to 9.99 and a resolution of 0.01. What is the smallest number of ADC bits that can be used?

**10.5** An embedded system will use an ADC to measure a distance. The measurement system range is -10 to +10 cm and a resolution of 0.01 cm. What is the smallest number of ADC bits that can be used?

**10.6** An embedded system will use an ADC to measure a force. The measurement system range is 0 to 100 N and a resolution of 0.01 N. What is the smallest number of ADC bits that can be used?

**10.7** An 8-bit ADC (different from the LM3S/TM4C) has an input range of 0 to +2 volts and an output range of 0 to 255 (called straight binary). What digital value will be returned when an input of +1.5 volts is sampled?

**10.8** A 12-bit ADC (different from the LM3S/TM4C) has an input range of -2.5 to +2.5 volts and an output range of 0 to 4095 (called offset binary). What digital value will be returned when an input of +1.25 volts is sampled?

**10.9** A 16-bit ADC (different from the LM3S/TM4C) has an input range of 0 to +2.5 volts and an output range of 0 to 65535 (called straight binary). What digital value will be returned when an input of +0.625 volts is sampled?

**D10.10** Write a function in C or in assembly that samples the ADC and returns a voltage in Register R0 using decimal fixed point with  $\Delta=0.001$  V. Assume the ADC has been initialized.

**D10.11** Write a function in C or in assembly that samples the ADC and returns a voltage in Register R0 using binary fixed point with  $\Delta=2^{-8}$  V. Assume the ADC has been initialized.

**D10.12** Assume an AC waveform is connected to analog channel 0. Write an initialization ritual. Write a subroutine that samples the analog input 256 times, and returns the DC amplitude (average) in Register R0, and the AC amplitude (maximum-minimum) in Register R1. Solve this problem in assembly or in C.

# 10.7. Lab Assignments

**Lab 10.1 Voltmeter.** Design, implement and test a device that measures DC voltage. Use periodic interrupts, ADC, and a display (OLED or LCD). Calibrate the device, then measure accuracy. Use decimal fixed-point numbers

**Lab 10.2 Distance Monitor.** Interface a Sharp GP2Y0A21YK0F infrared object detector to measure distance (<http://www.sharpsma.com>). This sensor creates a continuous analog voltage between 0 and +3V that depends inversely on distance to object, see Figure 10.14.

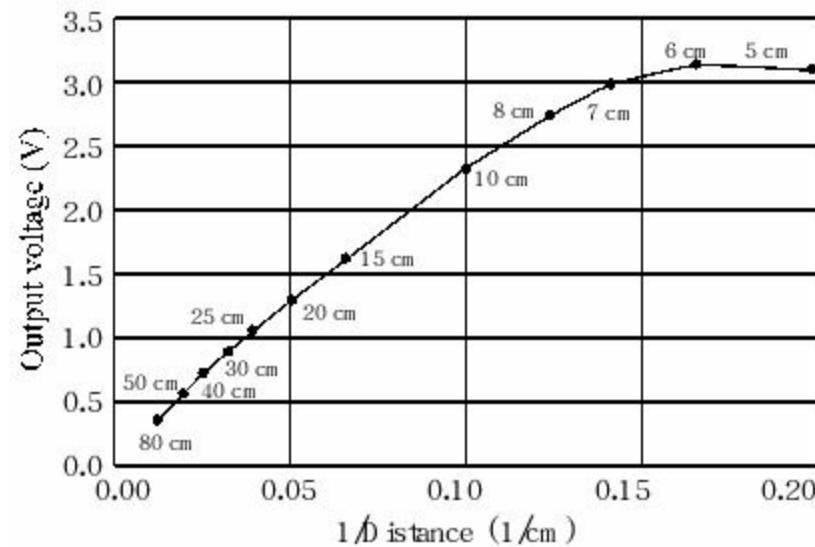


Figure 10.14. Response curve of the Sharp GP2Y0A21YK0F distance sensor.

**Lab 10.3 AC Voltmeter.** Design, implement and test a device that measures AC voltage. Use periodic interrupts, ADC, and a display (OLED or LCD). Calibrate the device, then measure accuracy. Use decimal fixed-point numbers

$$DC = (v[0]+v[1]+\dots+v[255])/256$$

The AC amplitude is calculated as a root-mean-squared value.

$$AC = \sqrt{((v[0]-DC)^2+(v[1]-DC)^2+\dots+(v[255]-DC)^2)/256}$$

**Lab 10.4 Real-Time Position Measurement System.** Interface a slide pot to the microcontroller and use it to measure distance. You will design a position meter with a range of about 3 cm. A linear slide potentiometer (Alpha RA300BF-10-20D1-B54) converts position into resistance ( $0 < R < 50\text{ k}\Omega$ ). You will use an electrical circuit to convert resistance into voltage ( $V_{in}$ ). The potentiometer has three leads. The ADC will convert voltage into a 10-bit digital number (0 to 1023). Your software will calculate position from the ADC sample as a decimal fixed-point number. The position measurements will be displayed on the LCD. A periodic interrupt will be used to establish the real-time sampling.

$$\text{Average accuracy (with units in cm)} = \frac{1}{n} \sum_{i=1}^n |x_{ti} - x_{mi}|$$

**Lab 10.5** Music generation using a Digital to Analog Converter. Design implement and test a 4-bit binary-weighted DAC. Use it to create music. Output a waveform to the DAC during a periodic interrupt.

# 11. Communication Systems

## Chapter 11 objectives are to:

- Present a general model for data flow problems,
- Develop implementations for the first in first out queue,
- Discuss methods to support interthread communication,
- Design show simple networks based on the UART port



The goal of this chapter is to provide a brief introduction to communication systems. Communication theory is a richly developed discipline, and much of the communication theory is beyond the scope of this book. Nevertheless, the trend in embedded systems is to employ multiple intelligent devices, therefore the interconnection will be a strategic factor in the performance of the system. A variety of different manufacturers are involved in the development these devices, thus the interconnection network must be flexible, robust, and reliable. Because the emphasis of this book is on real-time embedded systems, this chapter focuses on implementing communication systems appropriate for embedded systems. The components of an embedded system typically combined to solve a common objective, thus the nodes on the communication network will cooperate towards that shared goal. In particular, requirements of an embedded system, in general, involve relatively low to moderate bandwidth, static configuration, and a low probability of corrupted data. On the other hand reliability and latency are important for real-time systems.

## 11.1. Introduction

A **network** is a collection of interfaces that share a physical medium and a data protocol. A network allows software tasks in one computer to communicate and synchronize with software tasks running on another computer. For an embedded system, the network provides a means for distributed computing. The **topology** of a network defines how the components are interconnected. Examples topologies include rings, busses and multi-hop. Figure 11.1 shows a **ring** network of three microcontrollers. The advantage of this ring network is low cost and can be implemented on any microcontroller with a serial port. Notice that the microcontrollers need not be the same type or speed.

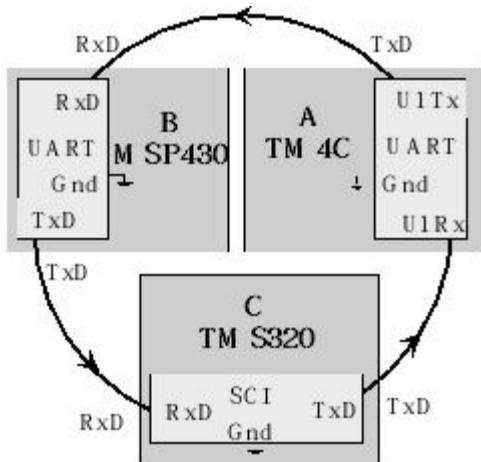


Figure 11.1. A simple ring network with three nodes, linked using the serial ports.

The ZigBee wireless network will be described in Volume 2, and Ethernet will be presented in Volume 3. In Chapter 8, we presented the hardware and software interfaces for the UART channel. We connected the LM3S/TM4C to an I/O device and used the UART to communicate with the human. In this chapter, we will build on those ideas and introduce the concepts of networks by investigating a couple of simple networks. In particular, we will use the UART channel to connect multiple microcontrollers together, creating a network. A communication network includes both the physical channel (hardware) and the logical procedures (software) that allow users or software processes to communicate with each other. The network provides the transfer of information as well as the mechanisms for process synchronization.

When faced with a complex problem, one could develop a solution on one powerful and **centralized** computer system. Alternatively a **distributed** solution could be employed using multiple computers connected by a network. The processing elements in Figure 11.2 may be a powerful computer, a microcontroller, an ASIC, or a smart sensor/actuator.

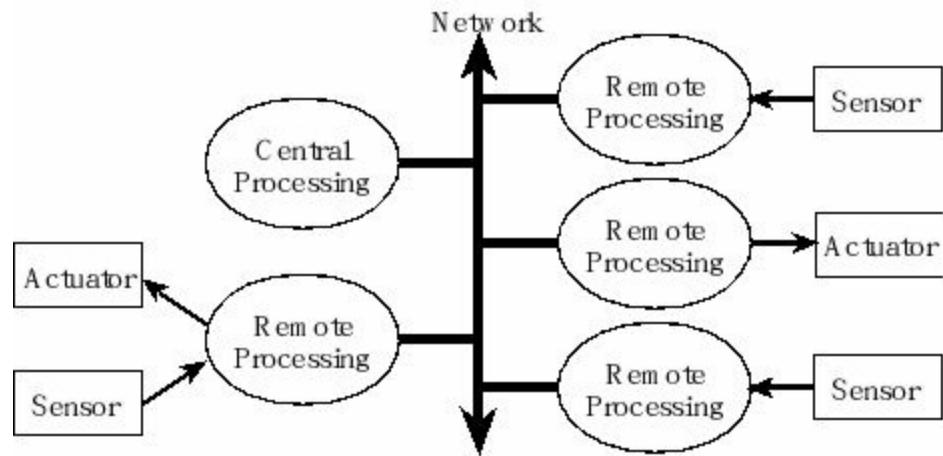


Figure 11.2. Distributed processing places input, output and processing at multiple locations connected together with a network.

There are many reasons to consider a distributed solution (network) over a centralized solution. Often multiple simple microcontrollers can provide a higher performance at lower cost compared to one computer powerful enough to run the entire system. Some embedded applications require input/output activities that are physically distributed. For real-time operation there may not be enough time to allow communication a remote sensor and a central computer. Another advantage of distributed system is improved debugging. For example, we could use one node in a network to monitor and debug the others. Often, we do not know the level of complexity of our problem at design time. Similarly, over time the complexity may increase or decrease. A distributed system can often be deployed that can be scaled. For example, as the complexity increases more nodes can be added, and if the complexity were to decrease nodes could be removed.

Most networks provide an **abstraction** that hides low-level details from high-level operations. This abstraction is often described as layers. The International Standards Organization (ISO) defines a 7-layer model called the **Open Systems Interconnection** (OSI). It provides a standard way to classify network components and operations. The **Physical** layer includes connectors, bit formats, and a means to transfer energy. Examples include RS232, controller area network (CAN), modem V.35, T1, 10BASE-T, 100BASE-TX, DSL, and 802.11a/b/g/n PHY. The **Data link** layer includes error detection and control across a single link (single hop). Examples include 802.3 (Ethernet), 802.11a/b/g/n MAC/LLC, PPP, and Token Ring. The **Network** layer defines end-to-end multi-hop data communication. The **Transport** layer provides connections and may optimize network resources. The **Session** layer provides services for end-user applications such as data grouping and check points. The **Presentation** layer includes data formats, transformation services. The **Application** layer provides an interface between network and end-user programs. A simple three-layer model is shown in Figure 11.3.

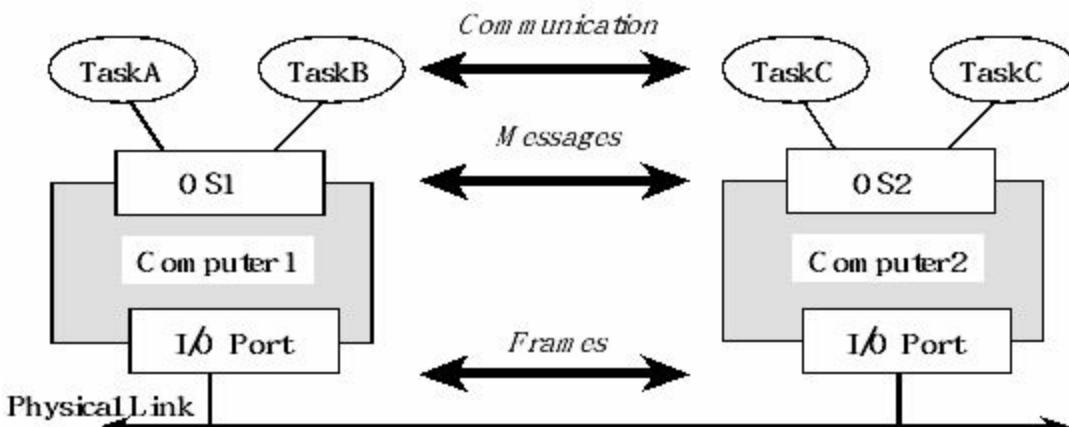


Figure 11.3. A layered approach to communication systems.

At the lowest level, frames are transferred between I/O ports of the two (or more) computers along the physical link or hardware channel. Error detection and correction may be handled at this low level. At the next logical level, the operating system (OS) of one computer sends messages or packets to the OS on the other computer. The message protocol will specify the types and formats of these messages. Error detection and correction may also be handled at this level. Messages typically contain four fields:

1) Address information field

Physical address specifying the destination/source computers

Logical address specifying the destination/source processes (e.g., users)

2) Synchronization or handshake field

Physical synchronization like shared clock, start and stop bits

OS synchronization like request connection or acknowledge

Process synchronization like semaphores

3) Data field

ASCII text (raw or compressed)

Binary (raw or compressed)

4) Error detection and correction field

Longitudinal redundancy check (LRC) (exclusive or of all data)

Checksum (least significant bits of the sum of all the data)

Block correction codes (BCC)

**Observation:** Communication systems often specify bandwidth in total bits/sec, but the important parameter is the data transfer rate.

**Observation:** Often the bandwidth is limited by the software and not the hardware channel.

At the highest level, we consider communication between users or high-level software tasks. Many embedded systems require the communication of command or data information to other modules at either a near or a remote location. Because the focus of this book is embedded systems, we will limit our discussion with communication with devices within the same room. A **full-duplex** channel allows data to transfer in both directions at the same time. In a **half-duplex** system, data can transfer in both directions but only in one direction at a time. Half duplex is popular because it is less expensive (2 wires) and allows the addition of more devices on the channel without change to the existing nodes.

## 11.2. Reentrant Programming and Critical Sections

As the system becomes more complex we need to be careful when sharing data. In general, if two threads access the same global memory and one of the accesses is a write, then there is a **causal dependency** between the threads. This means, the execution order may affect the outcome. Shared global variables are very important in multi-threaded systems because they are required to pass data between threads, but they are complicated and it is hard to find bugs that result with their use. The situation is even more complex in a distributed system.

A program segment is **reentrant** if it can be concurrently executed by two (or more) threads. To implement reentrant software, we place variables in registers or on the stack, and avoid storing into global memory variables. When writing in assembly, we use registers, or the stack for parameter passing to create reentrant subroutines. Typically each thread will have its own set of registers and stack. A nonreentrant subroutine will have a section of code called a **vulnerable window** or **critical section**. An error occurs if

- 1) One thread calls the function in question
- 2) It is executing in the critical section when interrupted by a second thread
- 3) The second thread calls the same function.

There are a number of scenarios that can happen next. In the most common scenario, the second thread is allowed to complete the execution of the function, control is then returned to the first thread, and the first thread finishes the function. This first scenario is the usual case with interrupt programming. In the second scenario, the second thread executes part of the critical section, is interrupted and then re-entered by a third thread, the third thread finishes, the control is returned to the second thread and it finishes, lastly the control is returned to the first thread and it finishes. This second scenario can happen in interrupt programming if the second interrupt has higher priority than the first. A critical section may exist when two different functions that access and modify the same memory-resident data structure.

The fundamental difficulty arises when information exists in multiple places. Program 11.1 shows a C function and the assembly code generated by the ARM Keil™ uVision® compiler. The function is nonreentrant because of the read-modify-write nonatomic access to the global variable, **num** . After executing **LDR r0,[r0,#0x00]** , there are two copies of **num** , one in the variable and a second copy in R0.

```
num SPACE 4
Count LDR r0,[pc,#116] ; R0=
&num
;*****start of critical section***
    LDR r0,[r0,#0x00] ; R0=num
;could be bad if interrupt occurs here
```

```
uint32_t volatile num;
void Count(void){
    num = num + 1;
}
```

```

ADDS r0,r0,#1
;could be bad if interrupt occurs here
LDR r1,[pc,#108] ; R1=&num
;could be bad if interrupt occurs here
STR r0,[r1,#0x00] ; update num
;*****end of critical section***
BX lr
ptr DCD num

```

Program 11.1. This function is nonreentrant because of the read-modify-write access to a global.

Assume there are two concurrent threads (the main program and a background ISR) that both call this function. Concurrent means that both threads are ready to run. Because there is only one computer, exactly one thread will be running at a time. Typically, the operating system switches execution control back and forth using interrupts. There are three places in the assembly code at which if an interrupt were to occur and the ISR called the same function, the end result would be **num** would be incremented only once, even though the function was called twice. Assume for this example **num** is initially 100. An error occurs if:

1. The main program calls **Count**
2. The main executes **LDR r0,[r0,#0x00]** making R0 = 100
3. The OS halts the main (using an interrupt) and starts the ISR
4. the ISR calls **Count**, executing **num=num+1**; making equal to 101
5. The OS returns control back to the main program, R0 is back to its original value of 100
6. The main program finished the function (adding 1 to R0), making **num** equal to 101

Basically, **Count** was called twice, but **num** was only incremented once.

An **atomic operation** is one that once started is guaranteed to finish. In most computers, once an instruction has begun, the instruction must be finished before the computer can process an interrupt. In general, nonreentrant code can be grouped into three categories all involving 1) nonatomic sequences, 2) writes and 3) global variables. We will classify I/O ports as global variables for the consideration of critical sections. We will group registers into the same category as local variables because each thread will have its own registers and stack.

The first group is the **read-modify-write** sequence:

1. The software reads the global variable producing a copy of the data
2. The software modifies the copy (original variable is still unmodified)
3. The software writes the modification back into the global variable.

In the second group, we have a **write followed by read**, where the global variable is used for temporary storage:

1. The software writes to the global variable (only copy of the information)
2. The software reads from the global variable expecting the original data to be there.

In the third group, we have a **non-atomic multi-step write** to a global variable:

1. The software writes part of the new value to a global variable
2. The software writes the rest of the new value to a global variable.

**Observation:** When considering reentrant software and vulnerable windows we classify accesses to I/O ports the same as accesses to global variables.

**Observation:** Sometimes we store temporary information in global variables out of laziness. This practice is to be discouraged because it wastes memory and may cause the module to not be reentrant.

Sometime we can have a critical section between two different software functions (one function called by one thread, and another function called by a different thread). In addition to above three cases, a **non-atomic multi-step read** will be critical when paired with a **multi-step write**. For example, assume a data structure has multiple components (e.g., hours, minutes, and seconds). In this case, the write to the data structure will be atomic because it occurs in a high priority ISR. The critical section exists in the foreground between steps 1 and 3. In this case, a critical section exists even though no software has actually been reentered.

<u>Foreground thread</u>	<u>Background thread</u>
1. The main reads some of the data	2. ISR writes to the data structure
3. The main reads the rest of the data	

In a similar case, a **non-atomic multi-step write** will be critical when paired with a **multi-step read**. Again, assume a data structure has multiple components. In this case, the read from the data structure will be atomic because it occurs in a high priority ISR. The critical section exists in the foreground between steps 1 and 3.

<u>Foreground thread</u>	<u>Background thread</u>
1. The main writes some of the data	2. ISR reads from the data structure
3. The main writes the rest of the data	

When multiple threads are active, it is possible for two threads to be executing the same program. For example, the system may be running in the foreground and calls **Func**. Part way through execution the **Func**, an interrupt occurs. If the ISR also calls **Func**, two threads are simultaneously executing the function. To experimentally determine if a function has been reentered, we could use two flags or two output pins. Set one of them (**PD1, Entered**) at the start and clear it at the end. The thread has been re-entered if this flag or pin is set at the start of the function, as shown in Program 11.2. In this example, Port D bits 1,0 are not part of the original code, but rather used just for the purpose of debugging. **PD1** is 1 when one thread starts executing the function. However, if **PD0** becomes 1, then the function has been reentered. Let **PD0** and **PD1** be bit-specific labels.

```
// function to be tested          // function to be tested
volatile int Entered=0,Flag=0;    void Func(void){
void Func(void){                if(PD1) PD0 = 1;
    if(Entered) Flag = 1;        PD1 = 2;
    Entered = 1;                // the regular function
// the regular function          PD1 = 0;
    Entered = 0;                }
}
```

Program 11.2. Detection of re-entrant behavior using two flags or two output pins.

If critical sections do exist, we can either eliminate it by removing the access to the global variable or implement **mutual exclusion**, which simply means only one thread at a time is allowed to execute in the critical section. In general, if we can eliminate the global variables, then the subroutine becomes reentrant. Without global variables there are no “vulnerable” windows because each thread has its own registers and stack. Sometimes one must access global memory to implement the desired function. Remember that all I/O ports are considered global. Furthermore, global variables are necessary to pass data between threads.

A simple way to implement mutual exclusion is to disable interrupts while executing the critical section. It is important to disable interrupts for as short a time as possible, so as to minimize the effect on the dynamic performance of the other threads. While we are running with interrupts disabled, time-critical events like power failure and danger warnings cannot be processed. Notice also that the interrupts are not simply disabled then enabled. Before the critical section, the interrupt status is saved, and the interrupts disabled. After the critical section, the interrupt status is restored.

1. Save the I bit in a local variable
2. Disable interrupts
3. Run critical section, code that requires mutual exclusion
4. Restore the I bit from the local variable.

You cannot save the interrupt status in a global variable, rather you should save it either on the stack or in a register. We will add the assembly code of Program 11.3 to the **Startup.s** file in our projects that use interrupts.

```
;***** StartCritical *****
```

; make a copy of previous I bit, disable interrupts

; inputs: none

; outputs: previous I bit

StartCritical

MRS R0, PRIMASK ; save old status

CPSID I ; mask all (except faults)

BX LR

;\*\*\*\*\* EndCritical \*\*\*\*\*

; using the copy of previous I bit, restore I bit to previous value

; inputs: previous I bit

; outputs: none

EndCritical

MSR PRIMASK, R0

BX LR

ALIGN

END

Program 11.3. Assembly functions needed for implementing mutual exclusion.

Program 11.4 illustrates how to implement mutual exclusion and eliminate the critical section.

```
uint32_t volatile num;
void Count(void){ uint32_t sr;
    sr = StartCritical(); // 1) save I bit and 2) disable
    num = num + 1;        // 3) run critical code exclusively
    EndCritical(sr);    // 4) restore I bit
}
```

Program 11.4. This function is reentrant because of the read-modify-write access to the global is atomic.

**Checkpoint 11.1:** Consider the situation of nested critical sections. For example, a function with a critical section calls another function that also has a critical section. What would happen if you simply added disable interrupt at the beginning and a reenable interrupt at the end of each critical section?

Another category of timing-dependent bugs, similar to critical sections, is called a **race condition**. A race condition occurs in a multi-threaded environment when there is a causal dependency between two or more threads. In other words, different behavior occurs depending on the order of execution of two threads. In this example of a race condition, Thread-A initializes Port D bits 3 – 0 to be output using **GPIO\_PORTD\_DIR\_R=0x0F**; Thread-B initializes Port D bits 6 – 4 to be output using **GPIO\_PORTD\_DIR\_R=0x70**; In particular, if Thread-A runs first and Thread-B runs second, then Port D bits 3 – 0 will be set to inputs. Conversely, if Thread-B runs first and Thread-A runs second, then Port D bits 6 – 4 will be set to inputs. This is a race condition caused by unfriendly code. The solution to this problem is to write the two initializations in a friendly manner.

In a second example, assume two threads are trying to get data from the same input device. Both call the function **UART\_InChar**. When data arrives at the input, the thread that executes first will capture the data.

# 11.3. Producer-Consumer using a FIFO Queue

## 11.3.1. Basic Principles of the FIFO Queue

The **first in first out** circular queue (**FIFO**) and **double buffer** are useful for data flow situations, as shown in Figure 11.4. These data structures can be used to link a source process (the **producer** is hardware/software that generates data) to a sink process (the **consumer** is hardware/software that consumes data.) In both cases the data is order-preserving, such that the order in which data is saved equals the order in which it is retrieved. There are many **producer-consumer** applications. In Table 11.1 the activities on the left are producers that create or input data, while the activities on the right are consumers that process or output data.

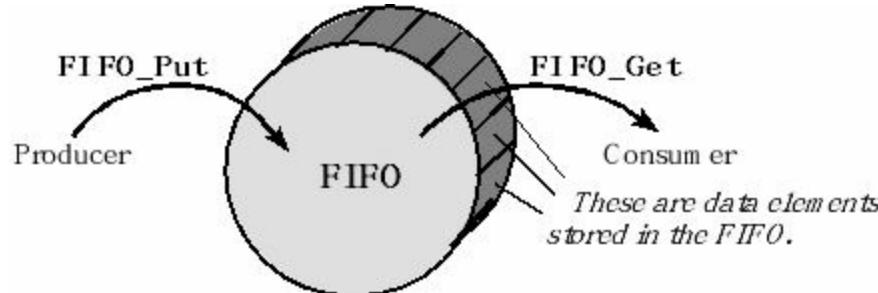


Figure 11.4. FIFO queues and double buffers can be used to pass data from a producer to a consumer.

Source/Producer	Sink/Consumer
Keyboard input	Program that interprets
Program with data	Printer output
Program sends message	Program receives message
Microphone and ADC	Program that saves sound data
Program that has sound data	DAC and speaker

Table 11.1. Producer-consumer examples.

The source process puts data into the FIFO or double buffer. If there is room, the **FIFO\_Put** operation saves data in the structure. If the data structure is full and the user tries to put, the **FIFO\_Put** routine will return a full error signifying the last (newest) data was not properly saved. The sink process removes data from the FIFO or double buffer. After a **FIFO\_Get**, the particular information returned from the **FIFO\_Get** routine is no longer saved. If the structure is empty and the user tries to get, the **FIFO\_Get** routine will return an empty error signifying no data could be retrieved. The FIFO and double buffer are order preserving, such that the information is returned by repeated calls of **FIFO\_Get** in the same order as the data was saved by repeated calls of **FIFO\_Put**. A FIFO typically can store many small chunks of data, whereas a double buffer can store two large fixed-size blocks of data. One can think of a double buffer as a FIFO of two elements, but each element is a large fixed-size block.

### **Checkpoint 11.2:** What conditions might cause the FIFO to become full?

The first in first out circular queue (**FIFO**) is quite useful for implementing a buffered I/O interface. It can be used for both buffered input and buffered output. The order preserving data structure temporarily saves data created by the source (producer) before it is processed by the sink (consumer). After initialization, the FIFO has two functions: **FIFO\_Put** (enters new data) and **FIFO\_Get** (removes the oldest data). You have probably already experienced the convenience of FIFOs. For example, when using an editor, you can continue to type characters while other processing is occurring. The ASCII codes are input from the keyboard as they are typed and put in a FIFO. When the editor is active again, it gets more keyboard data to process. A FIFO is also used when you ask the computer to print a file. Rather than waiting for the actual printing to occur character by character, the print command will put the data in a FIFO. Whenever the printer is free, it will get data from the FIFO. The advantage of the FIFO is it allows you to continue to use your computer while the printing occurs in the background. To implement this magic of background printing we will need interrupts.

Figure 11.5 shows a data flow graph with buffered input and buffered output. FIFOs used in this book will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be carefully shared by more than one program. The advantage of using a FIFO structure for a data flow problem is that we can decouple the producer and consumer threads. Without the FIFO we would have to produce one piece of data, then process it, produce another piece of data, then process it. With the FIFO, the producer thread can continue to produce data without having to wait for the consumer to finish processing the previous data. This decoupling can significantly improve system performance.

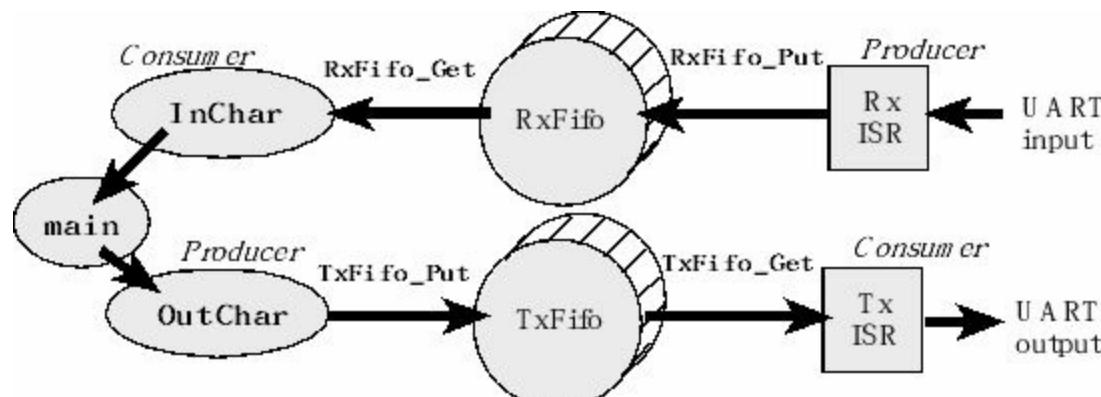


Figure 11.5. A data flow graph showing two FIFOs that buffer data between producers and consumers.

The flowchart for using FIFOs is illustrated in Figure 11.6. With mailbox synchronization, the threads execute in lock-step: one, the other, one, the other... However, with the FIFO queue execution of the threads is more loosely coupled. The classic producer/consumer problem has two threads. One thread produces data and the other consumes data. For an input device, the background thread is the producer because it generates new data, and the foreground thread is the consumer because it uses the data up. For an output device, the data flows in the other direction so the producer/consumer roles are reversed. It is appropriate to pass data from the producer thread to the consumer thread using a FIFO queue.

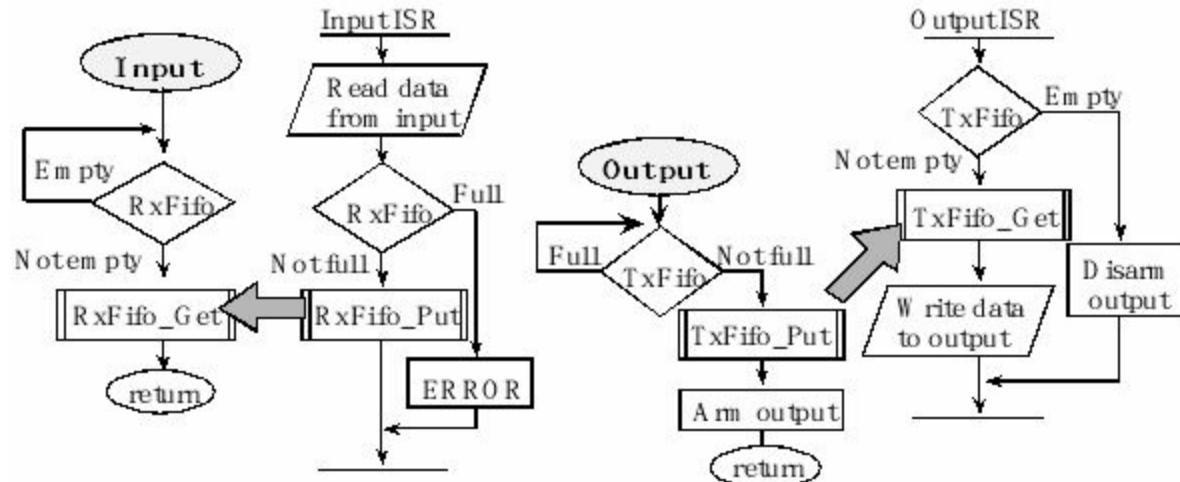


Figure 11.6. In a producer/consumer system, FIFO queues can be used to pass data between threads.

**Observation:** For systems with interrupt-driven I/O on multiple devices, there will be a separate FIFO for each device.

We could process the data within the ISR itself and just report the results of the processing to the main program using the mailbox. Processing data in the ISR is usually poor design because we try to minimize the time running in the ISR, in order to minimize latency of other interrupts.

An input device needs service (busy to done state transition) when new data are available, see Figure 11.7. The interrupt service routine (background) will accept the data and put it into a FIFO. Typically, the ISR will restart the input hardware, causing a done to busy transition.

An output device needs service (busy to done state transition) when the device is idle, ready to output more data. The interrupt service routine (background) will get more data from the FIFO and output it. The output function will restart the hardware causing a done to busy transition. Two particular problems with output device interrupts are

1. How does one generate the first interrupt?

In other words, how does one start the output thread? and

2. What does one do if an output interrupt occurs (device is idle) but there is no more data currently available (e.g., FIFO is empty)?

The foreground thread (main program) executes a loop and accesses the appropriate FIFO when it needs to input or output data. The background threads (interrupts) are executed when the hardware needs service.

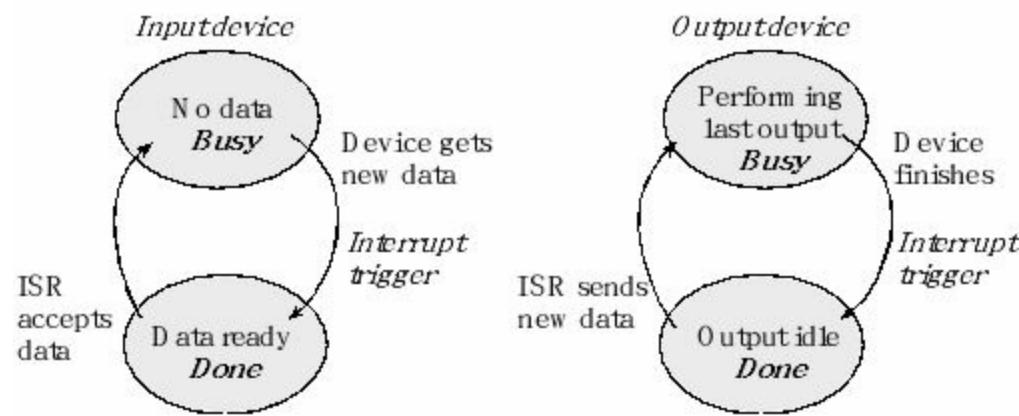


Figure 11.7. The input device interrupts when it has new data, and the output device interrupts when idle.

One way to visualize the interrupt synchronization is to draw a state versus time plot of the activities of the hardware and the two software modules. Figure 11.8 is drawn to describe a situation where the time between inputs is about twice as long as it takes the software to process the data. For this example, the main thread begins by waiting because the FIFO is empty (a). When the input device is busy, it is in the process of creating new input. When the input device is done, new data are available and an interrupt is requested. The interrupt service routine will read the data and put it into the FIFO (b). Once data are in the FIFO, the main program is released to go on because the get function will return with data (c). The main program processes the data (d) and then waits for more input (a). The arrows from one graph to the other represent the synchronizing events. Because the time for the software to read and process the data is less than the time for the input device to create new input, this situation is called **I/O bound**. In this situation, the FIFO has either 0 or 1 entry, and the use of interrupts does not enhance the bandwidth over the busy-wait implementations presented in the previous chapter. Even with an I/O bound device it may be more efficient to utilize interrupts because it provides a straight-forward approach to servicing multiple devices.

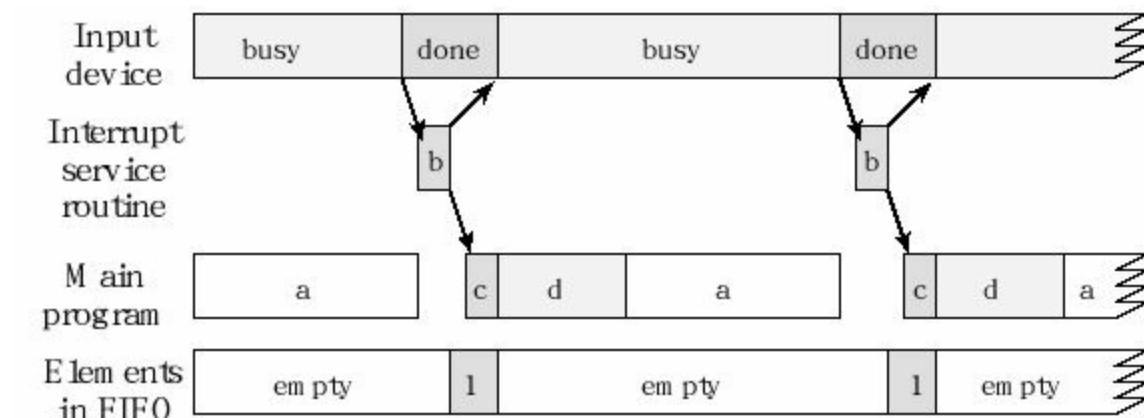


Figure 11.8. Hardware/software timing of an I/O bound input interface.

In this second example, the input device starts with a burst of high bandwidth activity. Figure 11.9 is drawn to describe a situation where the input rate is temporarily two to three times faster than the software can handle. As long as the interrupt service routine is fast enough to keep up with the input device, and as long as the FIFO does not become full during the burst, no data are lost. The software waits for the first data (a), but then does not have to wait until the burst is over. In this situation, the overall bandwidth is higher than it would be with a busy-wait implementation, because the input device does not have to wait for each data byte to be processed (b). This is the classic example of a “buffered” input, because the ISR puts data the FIFO. The main program gets data from the FIFO (c), and then processes it (d). When the I/O device is faster than the software, the system is called **CPU bound**. As we will see later, this system will work only if the producer rate temporarily exceeds the consumer rate (a short burst of high bandwidth input). If the external device sustained the high bandwidth input rate, then the FIFO would become full and data would be lost.

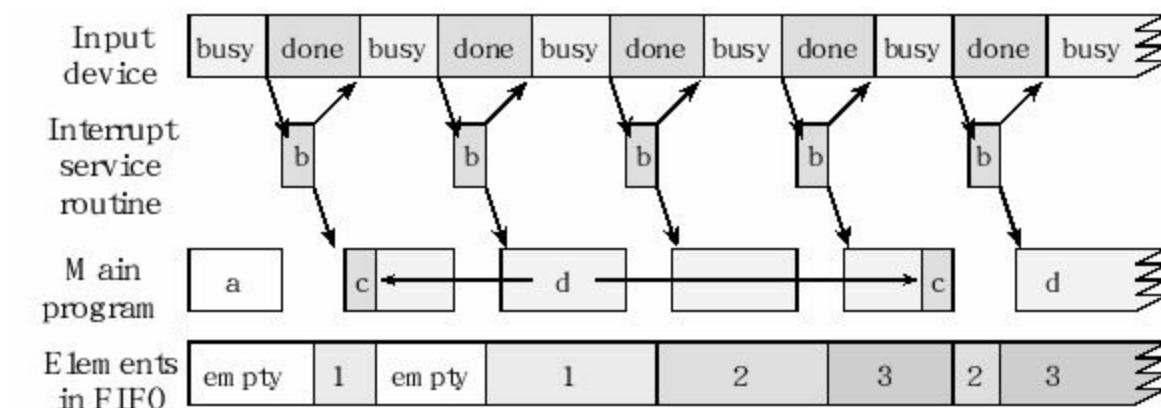


Figure 11.9. Hardware/software timing of an input interface during a high bandwidth burst.

For an input device, if the FIFO is usually empty, the interface is I/O bound. During times when there are many elements, the interface is CPU bound.

For an output device, the interrupt is requested when the output is idle and ready to accept more data. The “busy to done” state transition causes an interrupt. The interrupt service routine gives the output device another piece of data to output. Again, we can visualize the interrupt synchronization by drawing a state versus time plot of the activities of the hardware and the two software modules. Figure 11.10 is drawn to describe a situation where the time between outputs is about half as long as it takes the software to generate new data. For an output device interface, the output device is initially disarmed and the FIFO is empty. The main thread begins by generating new data (a). After the main program puts the data into the FIFO it arms the output interrupts (b). This first interrupt occurs immediately and the ISR gets some data from the FIFO and outputs it to the external device (c). The output device becomes busy because it is in the process of outputting data. It is important to realize that it only takes the software on the order of 1  $\mu$ sec to write data to one of its output ports, but usually it takes the output device much longer to fully process the data. When the output device is done, it is ready to accept more data and an interrupt is requested. If the FIFO is empty at this point, the ISR will disarm the output device (d). If the FIFO is not empty, the interrupt service routine will get from the FIFO, and write it out to the output port. Once data are written to the output port, the output device is released to go on. In this first example, the time for the software to generate data is larger than the time for the external device to output it. This is an example of a **CPU bound** system. In this situation, the FIFO has either 0 or 1 entry, and the use of interrupts does not enhance the

bandwidth over the busy-wait implementations presented in the previous chapter. Nevertheless, interrupts provide a well-defined mechanism for dealing with complex systems.

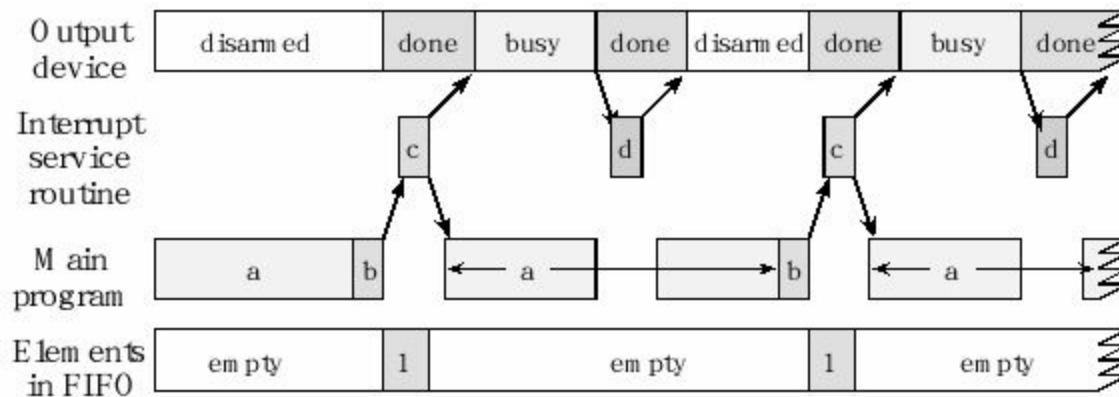


Figure 11.10. Hardware/software timing of a CPU bound output interface.

In this second output example, the software starts with a burst of high bandwidth activity. Figure 11.11 is drawn to describe a situation where the software produces data at a rate that is temporarily four to five times faster than the hardware can handle. As long as the FIFO does not become full, no data are lost. In this situation, the overall bandwidth is higher than it would be with a busy-wait implementation, because the software does not have to wait for each data byte to be processed by the hardware. The software generates data (a) and puts it into the FIFO (b). When the output is idle, it generates an interrupt. The ISR gets data and restarts the output device (c).

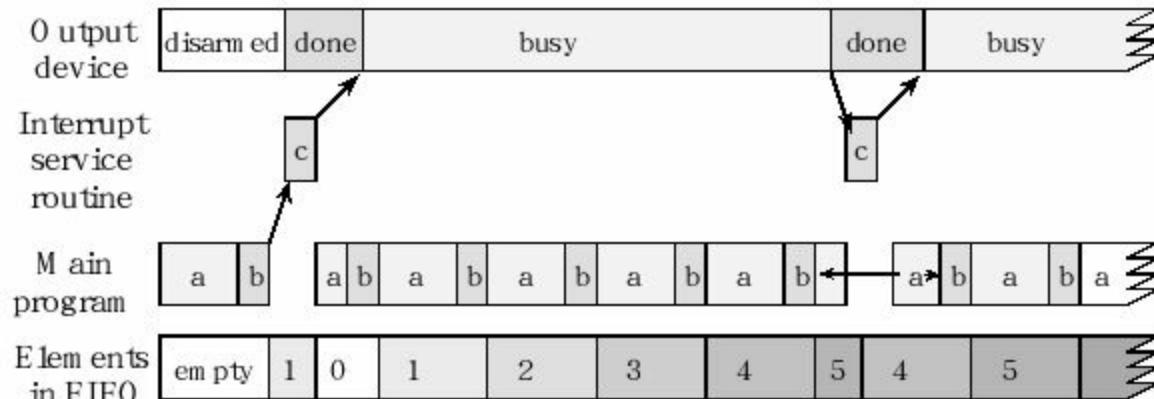


Figure 11.11. Hardware/software timing of an I/O bound output interface.

This is the classic example of a “buffered” output, because data enter the system (via the main program), are temporarily stored in a buffer (put into the FIFO), and then are processed later (by the ISR, get from the FIFO, write to external device.) When the I/O device is slower than the software, the system is called I/O bound. Just like the input scenario, the FIFO might become full if the producer rate is too high for too long.

**Checkpoint 11.3:** What does it mean if the RxFifo is empty?

**Checkpoint 11.4:** What does it mean if the TxFifo is empty?

## 11.3.2. FIFO Queue Analysis

As you recall, the FIFO passes the data from the producer to the consumer. In general, the rates at which data are produced and consumed can vary dynamically. Humans do not enter data into a keyboard at a constant rate. Even printers require more time to print color graphics versus black and white text. Let  $t_p$  be the time (in sec) between calls to **Fifo\_Put**, and  $r_p$  be the arrival rate (producer rate in bytes/sec) into the system, so  $r_p = 1/t_p$ . Similarly, let  $t_g$  be the time (in sec) between calls to **Fifo\_Get**, and  $r_g$  be the service rate (consumer rate in bytes/sec) out of the system, so  $r_g = 1/t_g$ .

If the minimum time between calls to **Fifo\_Put** is greater than the maximum time between calls to **Fifo\_Get**, then a FIFO is not necessary and the data flow could be solved with a mailbox. I.e., no FIFO is needed if  $\min(t_p) \geq \max(t_g)$ . On the other hand, if the time between calls to **Fifo\_Put** becomes less than the time between calls to **Fifo\_Get** because either

- The arrival rate temporarily increases
- The service rate temporarily decreases

then information will be collected in the FIFO. For example, a person might type very fast for a while, followed by long pause. The FIFO could be used to capture without loss all the data as it comes in very fast. Clearly on average the system must be able to process the data (the consumer thread) at least as fast as the average rate at which the data arrives (producer thread). If the average producer rate is larger than the average consumer rate

$$\text{Ave}(r_p) > \text{Ave}(r_g)$$

then the FIFO will eventually overflow no matter how large the FIFO. If the producer rate is temporarily high, and that causes the FIFO to become full, then this problem can be solved by increasing the FIFO size.

There is fundamental difference between an empty error and a full error. Consider the application of using a FIFO between your computer and its printer. This is a good idea because the computer can temporarily generate data to be printed at a very high rate followed by long pauses. The printer is like a turtle. It can print at a slow but steady rate. The computer will put a byte into the FIFO that it wants printed. The printer will get a byte out of the FIFO when it is ready to print another character. A full error occurs when the computer calls **Fifo\_Put** at too fast a rate. A full error is serious, because if ignored data will be lost. On the other hand, an empty error occurs when the printer is ready to print but the computer has nothing in mind. An empty error is not serious, because in this case the printer just sits there doing nothing.

**Checkpoint 11.5:** If the FIFO becomes full, can the situation always be solved by increasing the size?

Consider a FIFO that has a feature where we can determine the number of elements by calling **Fifo\_Size**. If we place this debugging instrument inside the producer, we can measure a histogram of FIFO sizes telling us 1) if the FIFO ever became full; 2) if the interface is CPU bound; or 3) if the interface is I/O bound.

```
uint32_t Histogram[FIFOSIZE];
#define Collect() (Histogram[Fifo_Size()]++);
```

An input interface using interrupts without a FIFO isn't any better than a busy-wait solution. If the next input data arrives before the previous data is processed, then data will be lost. When the I/O bandwidth is fast or unpredictable, it is appropriate to pass data from the producer thread to the consumer thread using a first in first out queue (FIFO). The FIFO will buffer the data between the foreground and background. The presence of the FIFO placed between the producer and consumer greatly improves performance by reducing the time each waits for the other.

### 11.3.3. FIFO Queue Implementation

There are many ways to implement a statically allocated FIFO. We can use either a pointer or and index to access the data in the FIFO. We can use either two pointers (or two indices) or two pointers (or two indices) and a counter. The counter specifies how many entries are currently stored in the FIFO. There are even hardware implementations of FIFO queues. If we were to have infinite memory, as shown in Figure 11.12, a FIFO implementation is easy. **GetI** is the index specifying data that will be removed by the next call to **Fifo\_Get**, and **PutI** is the index to the empty space where the data will be stored by the next call to **Fifo\_Put**. To put data in the FIFO, the new data is stored at **PutI**, and then this index is incremented. To get data from the FIFO, the value at **GetI** is read, and then this index is incremented.

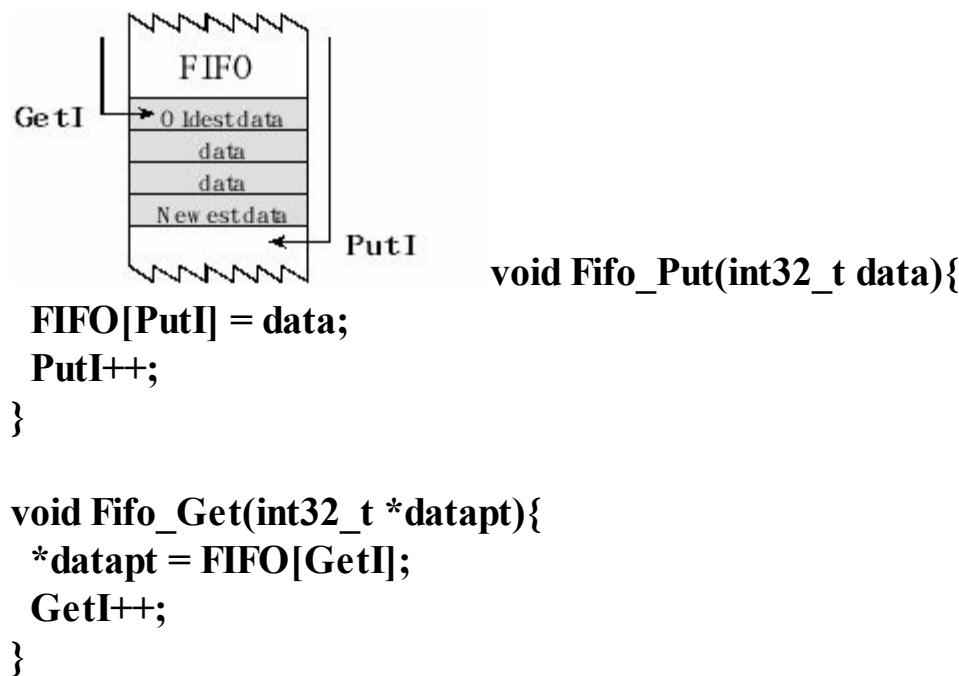


Figure 11.12. The FIFO implementation with infinite memory.

There are three modifications that are required to these functions. If the FIFO is full when **Fifo\_Put** is called then the subroutine should return a full error. Similarly, if the FIFO is empty when **Fifo\_Get** is called, then the subroutine should return an empty error. There is never an infinite amount of memory, so a finite number of bytes will be permanently allocated to the FIFO. Figures 11.13 and 11.14 show an example with 10 words allocated. The **PutI** and **GetI** must be wrapped back up to the top when they reach the bottom. The shaded blocks in these two figures represent valid data saved in the FIFO. Figure 12.8 shows how the FIFO changes as four words are **Put** into it. Figure 11.14 shows the same

FIFO as **Fifo\_Get** is called four times. Observe the order-preserving nature of the FIFO.

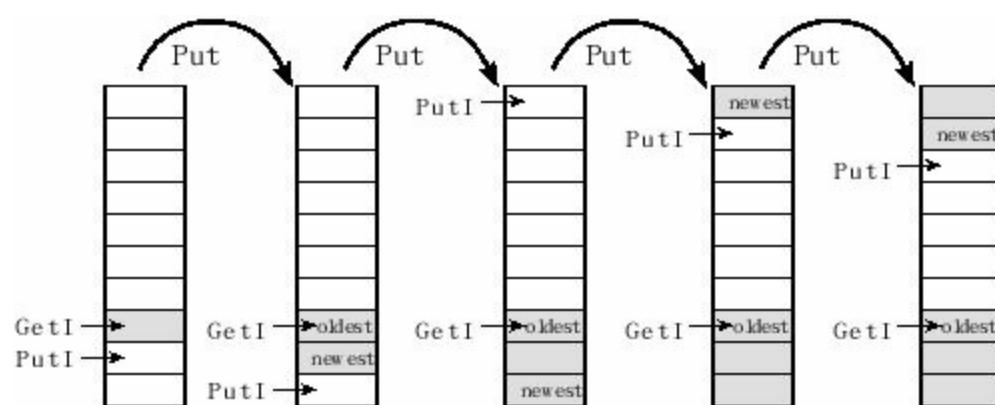


Figure 11.13. The FIFO **Put** operation showing the index wrap.

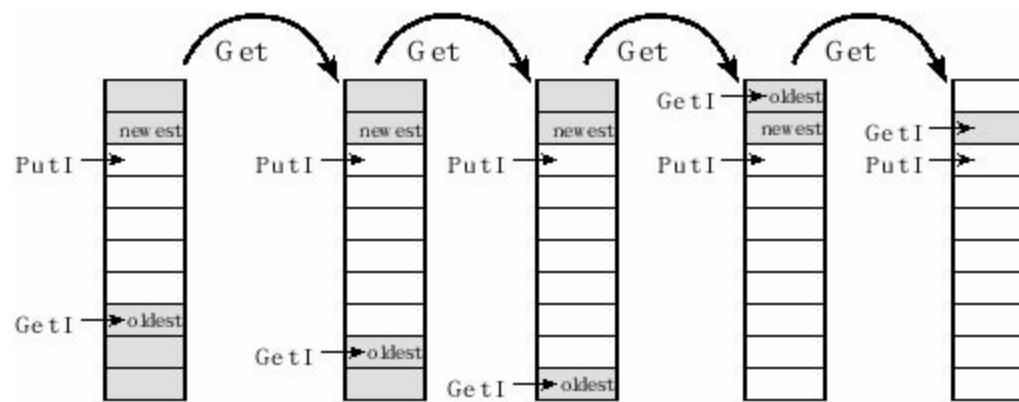


Figure 11.14. The FIFO **Get** operation showing the index wrap.

There are two mechanisms to determine whether the FIFO is empty or full. A simple method is to implement a counter containing the number of elements currently stored in the FIFO. **Fifo\_Get** would decrement the counter and **Fifo\_Put** would increment the counter. The second method, shown in Figure 11.15 and Program 11.5, is to prevent the FIFO from being completely full. For example, if the FIFO had 10 words allocated, then the **Fifo\_Put** subroutine would allow a maximum of 9 words to be stored. If there were already 9 words in the FIFO and another **Fifo\_Put** were called, then the FIFO would not be modified and a full error would be returned. In this way if **PutI** equals **GetI** at the beginning of **Fifo\_Get**, then the FIFO is empty. Similarly, if **PutI+1** equals **GetI** at the beginning of **Fifo\_Put**, then the FIFO is full. Be careful to wrap the **PutI+1** before comparing it to **GetI**. This second method does not require the length to be stored or calculated. The FIFO global structures must be allocated in RAM. **PutI** and **GetI** are private, and not accessible by programs outside the FIFO module.

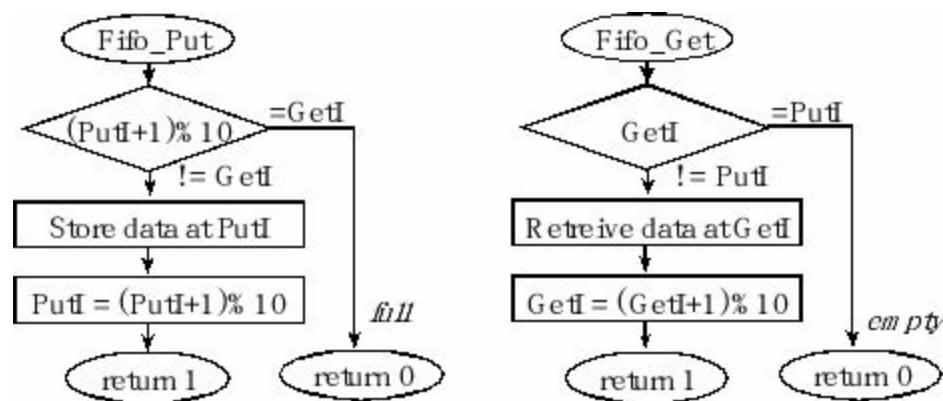


Figure 11.15. Flowcharts of the put and get operations.

The initialization function, **Fifo\_Init**, is usually called once at the start of the system. The FIFO is empty if the **PutI** equals the **GetI**. Both indices should always address locations within the 10-word allocated area. The **Fifo\_Put** routine enters new data in the FIFO. To check for FIFO full, the **Fifo\_Put** routine compares **PutI+1** to **GetI**. If putting would make the FIFO look empty, then the routine is exited without saving the data. This is why a FIFO with 10 allocated words can only hold 9 data points. If not full, then the data is stored and the **PutI** is updated. The **Fifo\_Get** routine removes the oldest data from the FIFO. To check for FIFO empty, the following **Fifo\_Get** routine checks to see if **GetI** equals **PutI**. If they match at the start of the routine, then **Fifo\_Get** returns with the “empty” condition signified. If not empty, the information is retrieved from the FIFO. The **GetI** is incremented signifying that information is no longer in the FIFO. If adding one to an index makes the index go beyond the FIFO buffer, the index is wrapped back to the beginning.

```

#define SIZE 10
uint32_t static PutI; // should be 0 to SIZE-1
uint32_t static GetI; // should be 0 to SIZE-1
int32_t static FIFO[SIZE];
void Fifo_Init(void){
    PutI = GetI = 0; // empty
}
int Fifo_Put(int32_t data){
    if(((PutI+1)%SIZE) == GetI) return 0; // fail if full
    FIFO[PutI] = data; // save in FIFO
    PutI = (PutI+1)%SIZE; // next place to put
    return 1;
}
int Fifo_Get(int32_t *datapt){
    if(PutI == GetI) return 0; // fail if empty
    *datapt = FIFO[GetI]; // retrieve data
    GetI = (GetI+1)%SIZE; // next place to get
    return 1; }

```

Program 11.5. Implementation of a two-index FIFO (FIFO\_xxx.zip).

## 11.3.4. Double Buffer

A double buffer is two buffers of fixed size. One example that uses a double buffer is a disk. Consider the situation where a large amount of data is to be read from a disk. The disk is organized into fixed size blocks. The size of each of the two buffers will match the block size of the disk. In the situation shown in Figure 11.16, the hardware is reading data from the disk filling **Buf1**. The hardware is configured to read an entire block. During this time the software is reading the data previously stored in **Buf2**. The double buffer will preserve order. This means the order in which the characters are input from the disk is the same as the order in which they are processed by the software. The differences between a FIFO queue and a double buffer are data size and queue length. The data size of a FIFO is typically one or two bytes. This means that one puts and gets single bytes into and out of the FIFO queue. The data size of the double buffer is typically large (e.g., 80, 256, 1024 bytes.) This means that one always saves and removes big blocks into and out of the double buffer. The FIFO queue length is large (typically ranging from 16 to 60000 bytes.) The double buffer has exactly 2 buffers. When the software finishes processing **Buf2** and the hardware finishes filling **Buf1**, the buffers are switched (hardware fills **Buf2** and the software processes **Buf1**.) This means if the hardware finishes first, then the disk hardware will have to be paused. Maximum disk efficiency occurs only if the disk can continuously read data as the blocks pass under the read head.

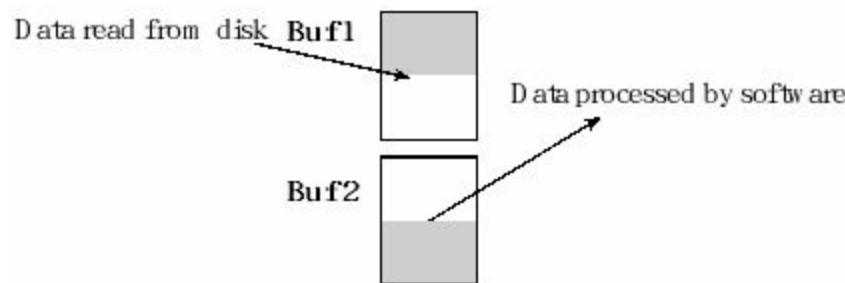


Figure 11.16. A double buffer allows you to store data into one buffer at the same time as retrieving data from the other buffer.

I/O devices which manipulate data in fixed size blocks are candidates for using double buffer data structures. Other examples of such devices include: graphics displays, bar code scanners, UPC readers, credit card readers, and IR receivers. A graphics display uses two buffers called a front buffer and a back buffer. The graphics hardware uses the front buffer to create the visual image on the display, i.e., the front buffer contains the data that you see. The software uses the back buffer to create a new image, i.e., the back buffer contains the data that you see next. When the new image is ready, and the time is right, the two buffers are switched (the front becomes the back and the back becomes the front.) In this way, the user never sees a partially drawn image.

# 11.4. Serial port interface using interrupt synchronization

The system shown in Figure 11.17 has two channels, one for input and one for output, and each channel employs a separate FIFO queue. Program 11.6 shows the interrupt-driven UART device driver. The flowchart for this interface was shown previously as Figure 11.6. During initialization, Port A pins 0 and 1 are enabled as alternate function digital signals. The two software FIFOs of Program 11.5 are initialized. The baud rate is set at 115200 bits/sec, and the hardware FIFOs are enabled. A transmit interrupt will occur as the transmit FIFO goes from 2 elements down to 1 element. Not waiting until the hardware FIFO is completely empty allows the software to refill the hardware FIFO and maintain a continuous output stream, achieving maximum bandwidth. There are two conditions that will request a receive interrupt. First, if the receive FIFO goes from 2 to 3 elements a receive interrupt will be requested. At this time there are still 13 free spaces in the receive FIFO so the latency requirement for this real-time input will be 130 bit times (about 1 ms). The other potential source of receiver interrupts is the receiver time out. This trigger will occur if the receiver becomes idle and there are data in the receiver FIFO. This trigger will allow the interface to receive input data when data comes one or two frames at a time. In the NVIC, the priority is set at 2 and UART0 (IRQ=5) is activated. Normally, one does not enable interrupts in the individual initialization functions. Rather, interrupts should be enabled in the main program, after all initialization functions have completed. Table 11.2 is an expanded version originally presented as Table 8.3. This table includes the registers required when using interrupts.

	31– 12	11	10	9	8	7–0			Name
\$4000.C000		OE	BE	PE	FE	DATA			UART0_DR_R
	31–3				3	2	1	0	
\$4000.C004					OE	BE	PE	FE	UART0_RSR_R
	31– 8	7	6	5	4	3	2–0		
\$4000.C018		TXFE	RXFF	TXFF	RXFE	BUSY			UART0_FR_R
	31– 16	15–0							
\$4000.C024		DIVINT							UART0_IBRD_R
	31–6				5–0				
\$4000.C028					DIVFRAC				UART0_FBRD_R
	31– 8	7	6 – 5	4	3	2	1	0	
\$4000.C02C		SPS	WPEN	FEN	STP2	EPS	PEN	BRK	UART0_LCRH_R

	31- 10	9	8	7	6-3	2	1	0	
\$4000.C030		RXE	TXE	LBE	SIRLP	SIREN	UARTEN	UART0_CTL_R	
	31-6				5-3		2-0		
\$4000.C034					RXIFLSEL		TXIFLSEL	UART0_IFLS_R	
	31- 11	10	9	8	7	6	5	4	
\$4000.C038		OEIM	BEIM	PEIM	FEIM	RTIM	TXIM	RXIM	UART0_IM_R
\$4000.C03C		OERIS	BERIS	PERIS	FERIS	RTRIS	TXRIS	RXRIS	UART0_RIS_R
\$4000.C040		OEMIS	BEMIS	PEMIS	FEMIS	RTMIS	TXMIS	RXMIS	UART0_MIS_R
\$4000.C044		OEIC	BEIC	PEIC	FEIC	RTIC	TXIC	RXIC	UART0_ICR_R

**Table 11.2. More UART registers. Each register is 32 bits wide. Shaded bits are zero.**

To use interrupts we will enable the FIFOs by setting the **FEN**bit in the **UART0\_LCRH\_R** register. **RXIFLSEL** specifies the receive FIFO level that causes an interrupt.

<u>RXIFLSEL</u>	<u>RX FIFO</u>	<u>Set RXMIS interrupt trigger when</u>
0x0	$\geq \frac{1}{8}$ full	Receive FIFO goes from 1 to 2 characters
0x1	$\geq \frac{1}{4}$ full	Receive FIFO goes from 3 to 4 characters
0x2	$\geq \frac{1}{2}$ full	Receive FIFO goes from 7 to 8 characters
0x3	$\geq \frac{3}{4}$ full	Receive FIFO goes from 11 to 12 characters
0x4	$\geq \frac{7}{8}$ full	Receive FIFO goes from 13 to 14 characters

**TXIFLSEL** specifies the transmit FIFO level that causes an interrupt.

<u>TXIFLSEL</u>	<u>TX FIFO</u>	<u>Set TXMIS interrupt trigger when</u>
0x0	$\leq \frac{7}{8}$ empty	Transmit FIFO goes from 15 to 14 characters
0x1	$\leq \frac{3}{4}$ empty	Transmit FIFO goes from 13 to 12 characters
0x2	$\leq \frac{1}{2}$ empty	Transmit FIFO goes from 9 to 8 characters
0x3	$\leq \frac{1}{4}$ empty	Transmit FIFO goes from 5 to 4 characters
0x4	$\leq \frac{1}{8}$ empty	Transmit FIFO goes from 3 to 2 characters

We will employ three of the possible seven interrupt trigger flags, located in the **UART0\_RIS\_R** register. The setting of the **TXRIS** and **RXRIS** flags is defined above. The **RTRIS** is set on a receiver timeout, which is when the receiver FIFO is not empty and no incoming frames have occurred in a 32-bit time period. Each of the seven trigger flags has a corresponding arm bit in the **UART0\_IM\_R** register. A bit in the **UART0\_MIS\_R** register set if the trigger flag is both set and armed. To acknowledge an interrupt (make the trigger flag become zero), software writes a 1 to the corresponding bit in the **UART0\_IC\_R** register.

When the main thread wishes to output it calls **UART\_OutChar**, which will put the data into the software FIFO. Next, it copies as much data from the software FIFO into the hardware FIFO and arms the transmitter. The transmitter interrupt service will also get as much data from the software FIFO and put it into the hardware FIFO. The **copySoftwareToHardware** function has a critical section and is called by both **UART\_OutChar** and the ISR. To remove the critical section the transmitter interrupt is temporarily disarmed in the **UART\_OutChar** function when **copySoftwareToHardware** is called. This helper function guarantees data is transmitted in the same order it was produced.

When input frames are received they are placed into the receive hardware FIFO. If this FIFO goes from 1 to 2elements, or if the receiver becomes idle with data in the FIFO, a receive interrupt occurs. The helper function **copyHardwareToSoftware** will get from the receive hardware FIFO and put into the receive software FIFO. When the main thread wishes to input data it calls **UART\_InChar**. This function simply gets from the software FIFO. If the receive software FIFO is empty, it will spin.

```
#define FIFO_SIZE 16      // size of the FIFOs (must be power of 2)
#define FIFO_SUCCESS 1    // return value on success
#define FIFO_FAIL 0       // return value on failure
AddIndexFifo(Rx, FIFO_SIZE, char, FIFO_SUCCESS, FIFO_FAIL)
AddIndexFifo(Tx, FIFO_SIZE, char, FIFO_SUCCESS, FIFO_FAIL)
```

```
void UART_Init(void){           // should be called only once
    SYSCTL_RCGCUART_R |= 0x01; // activate UART0
    SYSCTL_RCGCGPIO_R |= 0x01; // activate port A
    RxFifo_Init();           // initialize empty FIFOs
    TxFifo_Init();
    UART0_CTL_R &= ~UART_CTL_UARTEN; // disable UART
    UART0_IBRD_R = 27; // IBRD=int(50,000,000/(16*115,200)) = int(27.1267)
    UART0_FBRD_R = 8; // FBRD = round(0.1267 * 64) = 8
    UART0_LCRH_R = (UART_LCRH_WLEN_8|UART_LCRH_FEN); // 8-bit word, FIFOs
    UART0_IFLS_R &= ~0x3F; // clear TX and RX interrupt FIFO level fields
        // configure interrupt for TX FIFO <= 1/8 full
        // configure interrupt for RX FIFO >= 1/8 full
    UART0_IFLS_R += (UART_IFLS_TX1_8|UART_IFLS_RX1_8);
        // enable TX and RX FIFO interrupts and RX time-out interrupt
    UART0_IM_R |= (UART_IM_RXIM|UART_IM_TXIM|UART_IM_RTIM);
    UART0_CTL_R |= 0x0301; // enable RXE TXE UARTEN
    GPIO_PORTA_PCTL_R = (GPIO_PORTA_PCTL_R&0xFFFFFFF0)+0x00000011; // UART
    GPIO_PORTA_AMSEL_R &= ~0x03; // disable analog function on PA1-0
    GPIO_PORTA_AFSEL_R |= 0x03; // enable alt funct on PA1-0
    GPIO_PORTA_DEN_R |= 0x03; // enable digital I/O on PA1-0
    NVIC_PRI1_R = (NVIC_PRI1_R&0xFFFF00FF)|0x00004000; // UART0=priority 2
    NVIC_EN0_R = NVIC_EN0_INT5; // enable interrupt 5 in NVIC
    EnableInterrupts();
```

```

}

// copy from hardware RX FIFO to software RX FIFO
// stop when hardware RX FIFO is empty or software RX FIFO is full
void static copyHardwareToSoftware(void){ char letter;
    while(((UART0_FR_R&UART_FR_RXFE)==0)&&(RxFifo_Size() < (FIFOSIZE-1))){
        letter = UART0_DR_R;
        RxFifo_Put(letter);
    }
}

// copy from software TX FIFO to hardware TX FIFO
// stop when software TX FIFO is empty or hardware TX FIFO is full
void static copySoftwareToHardware(void){ char letter;
    while(((UART0_FR_R&UART_FR_TXFF) == 0) && (TxFifo_Size() > 0)){
        TxFifo_Get(&letter);
        UART0_DR_R = letter;
    }
}

// input ASCII character from UART
// spin if RxFifo is empty
char UART_InChar(void){
    char letter;
    while(RxFifo_Get(&letter) == FIFOFAIL){};
    return(letter);
}

// output ASCII character to SCI
// spin if TxFifo is full
void UART_OutChar(char data){
    while(TxFifo_Put(data) == FIFOFAIL){};
    UART0_IM_R &= ~UART_IM_TXIM;      // disable TX FIFO interrupt
    copySoftwareToHardware();
    UART0_IM_R |= UART_IM_TXIM;      // enable TX FIFO interrupt
}

// at least one of three things has happened:
// hardware TX FIFO goes from 3 to 2 or less items
// hardware RX FIFO goes from 1 to 2 or more items
// UART receiver has timed out
void UART0_Handler(void){
    if(UART0_RIS_R&UART_RIS_RXRIS){    // hardware TX FIFO <= 2 items
        UART0_ICR_R = UART_ICR_TXIC;    // acknowledge TX FIFO
        // copy from software TX FIFO to hardware TX FIFO
        copySoftwareToHardware();
        if(TxFifo_Size() == 0){          // software TX FIFO is empty
            UART0_IM_R &= ~UART_IM_TXIM; // disable TX FIFO interrupt
        }
    }
}

```

```
    }
}

if(UART0_RIS_R&UART_RIS_RXRIS){    // hardware RX FIFO >= 2 items
    UART0_ICR_R = UART_ICR_RXIC;    // acknowledge RX FIFO
    // copy from hardware RX FIFO to software RX FIFO
    copyHardwareToSoftware();
}
if(UART0_RIS_R&UART_RIS_RTRIS){    // receiver timed out
    UART0_ICR_R = UART_ICR_RTIC;    // acknowledge receiver time out
    // copy from hardware RX FIFO to software RX FIFO
    copyHardwareToSoftware();
}
}
```

Program 11.6. Interrupt-driven device driver for the UART uses two hardware FIFOs and two software FIFOs to buffer data (UART2\_xxx.zip).

## 11.5. \*Distributed Systems.

In this section, we will present three communication systems that utilize the UART port. If the distances are short, half duplex can be implemented with simple **open collector** or **open-drain** digital-level logic. Open drain logic has two output states: low and off. In the off state the output is not driven high or low, it just floats. The  $10\text{ k}\Omega$  pull-up resistor will passively make the signal high if none of the open drain outputs are low. The microcontroller can make its **TxD** serial outputs be open drain (**ODE** on the LM3S/TM4C). This mode allows a half-duplex network to be created without any external logic (although pull-up resistors are often used). Three factors will limit the implementation of this simple half-duplex network: 1) the number nodes on the network, 2) the distance between nodes; and 3) presence of corrupting noise. In these situations, a half-duplex RS485 driver chip like the SP483 made by Maxim can be used.

The first communication system is **master-slave** configuration, where the master transmit output is connected to all slave receive inputs, as shown in Figure 11.17. This provides for broadcast of commands from the master. All slave transmit outputs are connected together using wire-or open drain logic, allowing for the slaves to respond one at a time. **Wire-or** means if one slave sends a low, then all devices will see a zero on the line. If all slaves send a high each output will be HiZ, and the  $10\text{k}\Omega$  pull-up resistor will create a  $3.3\text{V}$  on the line. I.e., the low state dominates over the high state. The **ODE** (Open Drain Enable) in the slaves should be set to activate open drain mode on transmitters. The low-level device driver for this communication system is identical to the UART driver developed in the last section. When the master performs UART output it is broadcast to all the slaves. There can be no conflict when the master transmits, because a single output is connected to multiple inputs. When a slave receives input, it knows it is a command from the master. In the other direction, however, a potential problem exists because multiple slave transmitters are connected to the same wire. If the slaves only transmit after specifically being triggered by the master, no collisions can occur.

**Checkpoint 11.6:** What voltage level will the master RxD observe if two slaves simultaneously transmit, one making it a logic high and the other a logic low?

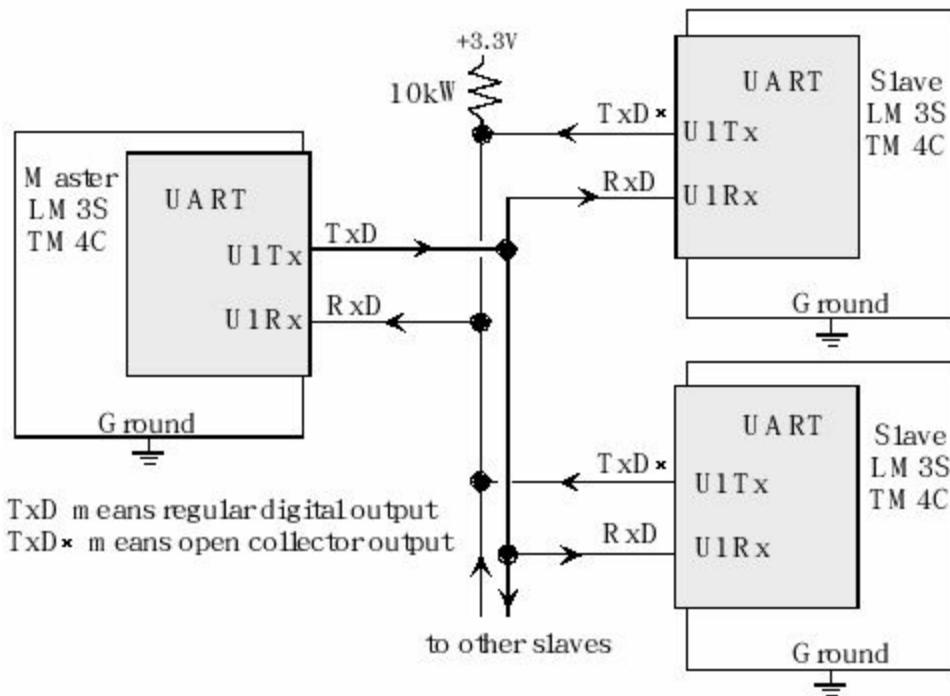


Figure 11.17. A master-slave network implemented with multiple microcontrollers.

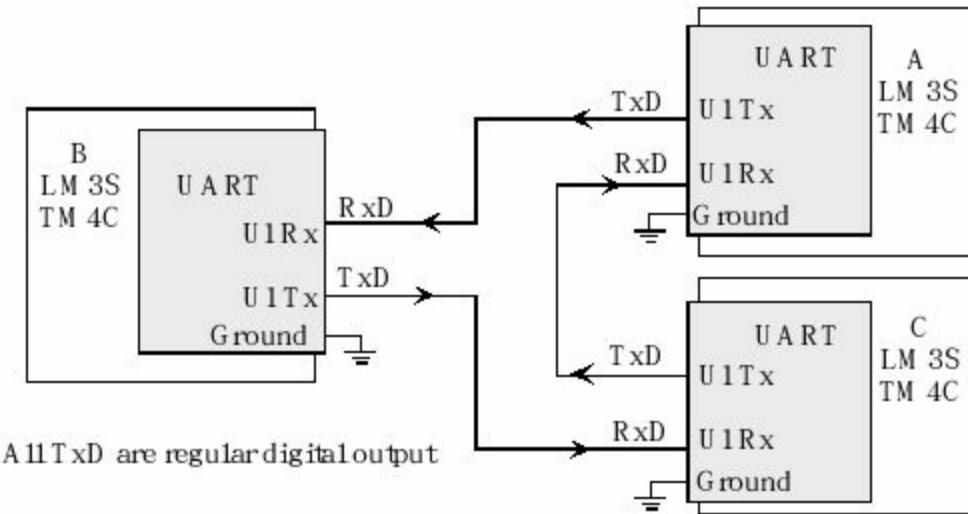


Figure 11.18. A ring network implemented with three microcontrollers.

The next communication system is a **ring network**. This is the simplest distributed system to design, because it can be constructed using standard serial ports. In fact, we can build a ring network simply by chaining the transmit and receive lines together in a circle, as shown in Figure 11.18. Building a ring network is a matter as simple as soldering a RS232 cable in a circle with one DB9 connector for each node. Messages will include source address, destination address and information. If computer A wishes to send information to computer C, it sends the message to B. The software in computer B receives the message, notices it is not for itself, and it resends the message to C. The software in computer C receives the message, notices it is for itself, and it keeps the message. Although simple to build, this system has slow performance (response time and bandwidth), and it is difficult to add/subtract nodes.

**Checkpoint 11.7:** Assume the ring network has 10 nodes, the baud rate is 100,000 bits/sec, and there are 10 bits/frame. What is average time it takes to send a 10 byte message from one computer to another?

The third communication system is a very common approach to distributed embedded systems, called **multi-drop**, as shown in Figure 11.19. To transmit a byte to the other computers, the software activates the SP483 driver and outputs the frame. Since it is half-duplex, the frame is also sent to the receiver of the computer that sent it. This echo can be checked to see if a collision occurred (two devices simultaneously outputting.) If more than two computers exist on the network, we usually send address information first, so that the proper device receives the data. Many collisions can be avoided by waiting for the receiver to finish before transmitting.

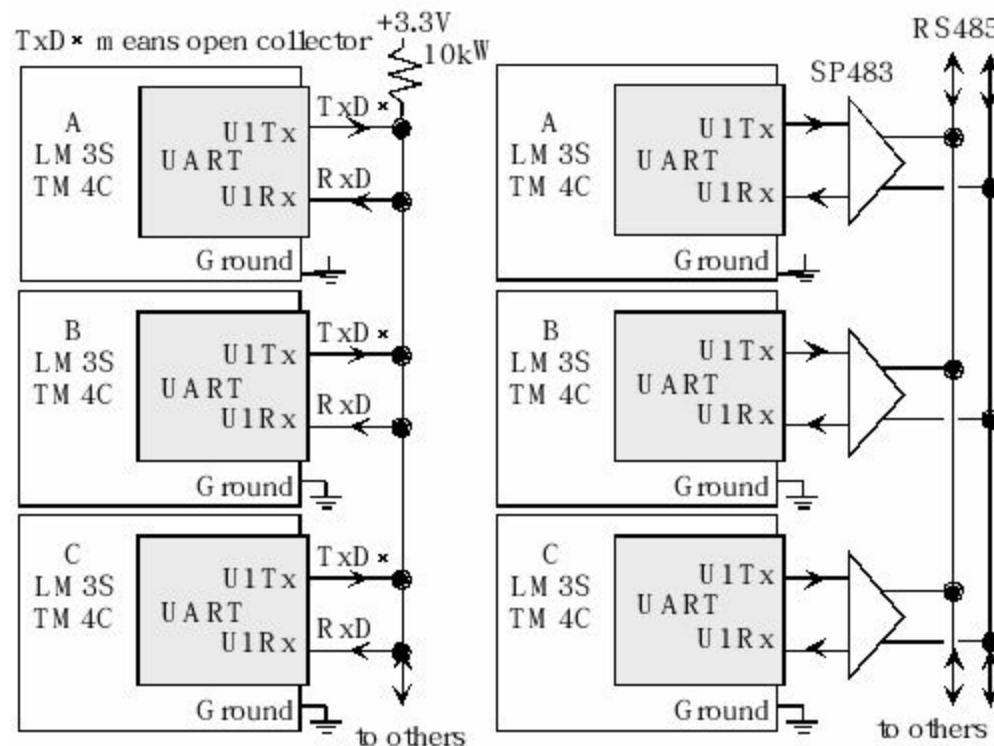


Figure 11.19. A multidrop network is created using a half-duplex serial channel implemented with open drain logic or with RS485 drivers.

**Checkpoint 11.8:** How can the transmitter detect a collision had corrupted its output?

**Checkpoint 11.9:** How can the receiver detect a collision had corrupted its input?

There are many ways to check for transmission errors. You could use a **longitudinal redundancy check** (LRC) or horizontal even parity. The error check byte is simply the **exclusive-OR** of all the message bytes (except the LRC itself). The receiver also performs an **exclusive-OR** on the message as well as the error check byte. The result will equal zero if the block has been transmitted successfully. Another popular method is **checksum**, which is simply the modulo<sub>256</sub> (8-bit) or modulo<sub>65536</sub> (16-bit) sum of the data packet. In addition, each byte could have (but doesn't have to) include even parity.

There are two mechanisms that allow the transmission of variable amounts of data. Some protocols use start (STX=0x02) and stop (ETX=0x03) characters to surround a variable amount of data. The disadvantage of this “termination code” method is that binary data cannot be sent because a data byte might match the termination character (ETX). Therefore, this protocol is appropriate for sending ASCII characters. Another possibility is to use a byte count to specify the length of a message. Many protocols use a byte count. The ZigBee frames, for example, have a byte count in each frame.

# 11.6. Exercises

**11.1** How do you tell if this C code

**GPIO\_PORTA\_DATA\_R |= 0x80; // set PA7**

has a critical section with this C code?

**GPIO\_PORTA\_DATA\_R |= 0x40; // set PA6**

**11.2** How do you tell if the following C code is reentrant?

**Counter++;**

**11.3** Consider the memory manager programs of Section 6.6. Are any of the following programs reentrant: **Heap\_Init**, **Heap\_Allocate**, **Heap\_Release**, **Fifo\_Init**, **Fifo\_Put**, and **Fifo\_Get**? How can the problem be fixed?

**11.4** Consider the pointer-based **Fifo\_Put** and **Fifo\_Get** functions shown in Program 11.5. There is one foreground thread that calls **Fifo\_Get**, and two interrupt threads that call **Fifo\_Put**. In particular, both the regular UART interrupt and a SysTick periodic ISR call **Fifo\_Put** to enter data into the **FIFO**. Is there a critical section?

**11.5** Consider the linked list **Fifo\_Put** and **Fifo\_Get** functions from Program 6.16 and Program 6.17. Assume exactly one is called from the foreground main program and one from the background ISR. Is there a critical section? I.e., do interrupts need to be disabled to use these two functions?

**11.6** Consider the situation in which a FIFO queue is used to buffer data between a main program (e.g., **UART\_OutChar** that calls **TxFifo\_Put**) and an output interrupt service routine (e.g., **UART0\_Handler** that calls **TxFifo\_Get** and writes to the hardware). Experimental observations show that this FIFO is usually empty, and at most contains 3 elements. What does it mean? Choose A-F.

- a) The system is I/O bound
- b) The system is CPU bound
- c) Bandwidth could be increased by increasing FIFO size
- d) The FIFO could be replaced by a global variable
- e) The latency is small and bounded
- f) Interrupts are not needed in this system

**11.7** Consider the situation in which a FIFO queue is used to buffer data between a main program (e.g., **UART\_OutChar** that calls **TxFifo\_Put**) and an output interrupt service routine (e.g., **UART0\_Handler** that calls **TxFifo\_Get** and writes to the hardware). Experimental observations show that this FIFO often becomes full, and usually having more than 5 elements. What does it mean? Choose A-F.

- a) The system is I/O bound
- b) The system is CPU bound
- c) Bandwidth could be increased by increasing FIFO size
- d) The FIFO could be replaced by a global variable
- e) The latency is small and bounded
- f) Interrupts are not needed in this system

**11.8** Consider the situation in which a FIFO queue is used to buffer data between a main program (e.g., **UART\_InChar** that calls **RxFifo\_Get**) and an inputinterrupt service routine (e.g., **UART0\_Handler** that reads the hardware and calls **RxFifo\_Put**). Experimental observations show that this FIFO is often becomes full, and usually having more than 5 elements. What does it mean? Choose A-F.

- a) The system is I/O bound
- b) The system is CPU bound
- c) Bandwidth could be increased by increasing FIFO size
- d) The FIFO could be replaced by a global variable
- e) The latency is small and bounded
- f) Interrupts are not needed in this system

**11.9** Consider the situation in which a FIFO queue is used to buffer data between a main program (e.g., **UART\_InChar** that calls **RxFifo\_Get**) and an inputinterrupt service routine (e.g., **UART0\_Handler** that reads the hardware and calls **RxFifo\_Put**). Experimental observations show that this FIFO is usually empty, and at most contains 3 elements. What does it mean? Choose A-F.

- a) The system is I/O bound
- b) The system is CPU bound
- c) Bandwidth could be increased by increasing FIFO size
- d) The FIFO could be replaced by a global variable
- e) The latency is small and bounded
- f) Interrupts are not needed in this system

**D11.10.** Design a UART driver that just outputs using interrupt synchronization. Use the UART1 serial port and run at 38400 bits/sec. Include user functions **UART1\_Init()** , **UART1\_OutChar()** , **UART1\_OutString()** and **UART1\_OutDec** .

**D11.11.** Assume you have two microcontrollers the UART1s connected. Implement interrupt synchronization with a baud rate of 78600 bits/sec. Implement a distributed mailbox as described in Section 9.3. The main program on one microcontroller produces data and puts it in the mailbox, and the main program on the other microcontroller consumes the data. The two threads are synchronized. This means they wait for each other.

**D11.12.** Write two functions that operate on variable length strings. The first is LRC generation that takes an ASCII string and calculates the exclusive or of all the data in a byte wise fashion. The input to the function is a string and the output is a single byte, which is the LRC. The second function assumes the string already has an LRC byte as the last character. There are two inputs and one output for this function. The inputs are a pointer to the string and the length (length is needed because the LRC might be 0). The output is a Boolean true if the LRC is OK or false if the LRC is incorrect.

---

# 11.7. Lab Assignments

**Lab 11.1** Distributed Data Acquisition System. Take one of the measurement labs (Lab 10.1, 10.2, 10.3, or 10.4 and split the sensor and display. Interface the sensor to one microcontroller and interface the display to the other. Collect data from the sensor, transmit it across the serial link and display it on the other microcontroller.

**Lab 11.2** Walkie-talkie. Interface microphones and speakers to two or more microcontrollers. Measure sound on from one microphone, transmit the sound to the other microcontroller and output it to the speaker.

**Lab 11.3** Ring Network. Take one of the measurement labs (Lab 10.1, 10.2, 10.3, or 10.4 and split the sensor and display. Implement 3 or more microcontrollers in a ring network. Interface a sensor and a display to each microcontroller. Collect data from the sensors, share it with the other microcontrollers. Create a display that shows both local and remote measurements.

# 11.8. Best Practices

- Consider debugging when defining, designing, implementing, building and deploying.
- Careful thought during design can save lots of time during implementation and debugging.
- Choose good variable names so the software is easier to understand.
- Divide large projects into modules and test each module separately.
- Separate hardware from software bugs by first testing the software on a simulator.
- When designing modules start with the interfaces, e.g., the header files.
- The second step when designing modules is pseudo code typed in as comments.
- Make the time to service an interrupt short compared to the time between interrupts.
- When developing a modular system, try not to change the header files.
- Use a consistent coding style so all your software is easy to read, change, and debug.
- Most of your time is spent changing or fixing existing code called maintenance.
- So, when designing code plan for testing and make it easy to change.
- Writing friendly code makes it easier to combine components into systems.
- Use quality connectors, because faulty connectors can be a difficult flaw to detect.
- It is your responsibility to debug your hardware and software.
- It is also your responsibility to debug other hardware/software you put into your system.
- A simple solution is often more powerful than a complex solution.
- Listen carefully to your customers so you can understand their needs.
- Draw wiring diagrams of electrical circuits before building.
- Double-check all the wiring before turning on the power.
- Double-check all signals in cables, don't assume red is power and black is ground.
- Be courageous enough to show your work to others.
- Be humble enough to allow others to show you how your system could be better.

# Appendix 1. Glossary

- 1/f noise** A fundamental noise in resistive devices arising from fluctuating conductivity. Same as pink noise.
- 2's complement** (see two's complement).
- 60 Hz noise** An added noise from electromagnetic fields caused by either magnetic field induction or capacitive coupling.
- accumulator** High-speed storage located in the processor used to perform arithmetic or logical functions. The accumulators on the ARM Cortex M are Register R0 through R12.
- accuracy** A measure of how close our instrument measures the desired parameter referred to the NIST.
- acknowledge** Clearing the interrupt flag bit that requested the interrupt.
- actuator** Electro-mechanical or electro-chemical device that allows computer commands to affect the external world. Examples include motors, relays, solenoids, and speakers.
- ADC** Analog to digital converter, an electronic device that converts analog signals (e.g., voltage) into digital form (i.e., integers). The ADC on the TM4C is 12 bits and can sample up to 1 M samples/sec.
- address bus** A set of digital signals that connect the CPU, memory and I/O devices, specifying the location to read or write for each bus cycle. See also control bus and data bus.
- aliasing** When digital values sampled at  $f_s$  contain frequency components above 0.5  $f_s$ , then the apparent frequency of the data is shifted into the 0 to 0.5  $f_s$  range. See Nyquist theory.
- alternatives** the total number of possibilities. E.g., an 8-bit number scheme can represent 256 different numbers. An 8-bit digital to analog converter (DAC) can generate 256 different analog outputs.
- arithmetic logic unit (ALU)** Component of the processor that performs arithmetic and logic operations.
- arm** Activate an individual trigger so that interrupts are requested when that trigger flag is set.
- ASCII** American Standard Code for Information Interchange, a code for representing characters, symbols, and synchronization messages as 7 bit, 8-bit or 16-bit binary values.
- assembler** System software that converts an assembly language program (human readable format) into object code (machine readable format).
- assembly directive** Operations included in the program that are not executed by the computer at run time, but rather are interpreted by the assembler during the assembly process. Same as pseudo-op.
- assembly listing** Information generated by the assembler in human readable format, typically showing the object code, the original source code, assembly errors, and the symbol table.
- asynchronous protocol** a protocol where the two devices have separate and distinct clocks
- atomic** Software execution that cannot be divided or interrupted. Once started an atomic operation will run to its completion without interruption. On most computers the assembly language instructions are atomic. All instructions on the ARM® Cortex™-Mprocessor are atomic except store and load multiple, **STM LDM**.
- availability** The portion of the total time that the system is working. MTBF is the mean time between failures, MTTR is the mean time to repair, and availability is  $MTBF/(MTBF+MTTR)$ .

**bandwidth** The information transfer rate, the amount of data transferred per second. Same as throughput.

**basis** Subset from which linear combinations can be used to reconstruct the entire set. The basis of the 8-bit unsigned number system is 1, 2, 4, 8, 16, 32, 64, and 128.

**baud rate** In general the baud rate is the total number of bits (information, overhead, and idle) per time that are transmitted, in a modem application it is the total number of sounds per time are transmitted.

**bi-directional** Digital signals that can be either input or output.

**biendian** The ability to process numbers in both big and little endian formats.

**big endian** Mechanism for storing multiple byte numbers such that the most significant byte exists first (in the smallest memory address). See also little endian.

**binary** A system that has two states, on and off.

**binary operation** A function that produces its result given two input parameters. For example, addition, subtraction, and multiplication are binary operations.

**binary recursion** A recursive technique that makes two calls to itself during the execution of the function. See also recursion, linear recursion, and tail recursion.

**bipolar stepper motor** A stepper motor where the current flows in both directions (in/out) along the interface wires; a stepper with 4 interface wires.

**bit** Basic unit of digital information taking on the value of either 0 or 1.

**bit time** The basic unit of time used in serial communication.

**blind cycle** A software/hardware synchronization method where the software waits a specified amount of time for the hardware operation to complete. The software has no direct information (blind) about the status of the hardware.

**Board Support Package (BSP)** A set of software routines that abstract the I/O hardware such that the same high-level code can run on multiple computers. Same as hardware abstraction layer (HAL).

**borrow** During subtraction, if the difference is too small, then we use a borrow to pass the excess information into the next higher place. For example, in decimal subtraction 36-27 requires a borrow from the ones to tens place because 6-7 is too small to fit into the 0 to 9 range of decimal numbers.

**break or trap** A break or a trap is an instrument that halts the processor. When encountered it will stop your program and jump into the debugger. Therefore, a break halts the software. The condition of being in this state is also referred to as a break.

**breakpoint** The place where a break is inserted, the time when a break is encountered, or the time period when a break is active.

**buffered I/O** A FIFO queue is placed in between the hardware and software in an attempt to increase bandwidth by allowing both hardware and software to run in parallel.

**burn** The process of programming a ROM, PROM or EEPROM.

**bus** A set of digital signals that connect the CPU, memory and I/O devices, consisting of address signals, data signals and control signals. See also address bus, control bus and data bus.

**bus interface unit (BIU)** Component of the processor that reads and writes data from the bus.

**busy wait** A software/hardware synchronization method where the software continuously reads the hardware status waiting for the hardware operation to complete. The software usually performs no work while waiting for the hardware. Same as gadfly. Same as polling.

**byte** Digital information containing 8 bits. In C, we use **char** or **unsigned char** to create a byte. In C99, we use **int8\_t** or **uint8\_t** to create a byte. In both C and C99, we use **char** to create an 8-bit ASCII character.

**call graph** A graphical way to define how the software/hardware modules interconnect. If a function in module A invokes a function in module B, then there is an arrow from A to B.

**carry** During addition, if the sum is too large, then we use a carry to pass the excess information into the next higher place. For example, in decimal addition  $36+27$  requires a carry from the ones to tens place because  $6+7$  is too big to fit into the 0 to 9 range of decimal numbers.

**ceiling** Establishing an upper bound on the result of an operation.

**checksum** The simple sum of the data, usually in finite precision (e.g., 8, 16, 24 bits).

**client** A programmer/engineer who will use our software and/or hardware. This is typically not the end-user of the final system rather it is another engineer who will integrate our software and/or hardware into a larger system.

**closed loop control system** A control system that includes sensors to measure the current state variables. These inputs are used to drive the system to the desired state.

**CMOS** A digital logic system called complementary metal oxide semiconductor. It has properties of low power and small size. Its power is a function of the number of transitions per second.

**cohesion** A cohesive module is one such that all parts of the module are related to each other to satisfy a common objective.

**compiler** System software that converts a high-level language program (human readable format) into object code (machine readable format).

**complex instruction set computer (CISC)** A computer with many instructions, instructions that have varying lengths, instructions that execute in varying times, many instructions can access memory, instructions that can read and write memory in the same bus cycle, fewer and more specialized registers, and many different types of addressing modes. Contrast to RISC.

**concurrent programming** A computer system that supports two or more software tasks that are simultaneously active. Typically one task executes at a time, and there are mechanisms to suspend one task and execute another task. Compare to parallel programming.

**control bus** A set of digital signals that connect the processor, memory and I/O devices, specifying when to read or write for each bus cycle. See also address bus and data bus.

**control unit (CU)** Component of the processor that determines the sequence of operations.

**CPU bound** A situation where the input or output device is faster than the software. In other words it takes less time for the I/O device to process data, than for the software to process data.

**critical section** Locations within a software module, which if an interrupt were to occur at one of these locations, then an error could occur (e.g., data lost, corrupted data, program crash, etc.) Same as vulnerable window.

**cross-assembler** An assembler that runs on one computer but creates object code for a different computer.

**cross-compiler** A compiler that runs on one computer but creates object code for a different computer.

**cyber-physical system** A system that performs a specific dedicated operation where the computer is hidden or embedded inside the machine. The system has intelligence in the software and physical connections to the real world. Same as embedded system.

**DAC** Digital to analog converter, an electronic device that converts digital signals (i.e., integers) to analog form (e.g., voltage).

**data acquisition system (DAS)** A system that collects information, same as instrument.

**data bus** A set of digital signals that connect the CPU, memory and I/O devices, specifying the value that is being read or written for each bus cycle. See also address bus and control bus.

**data flow graph** A block diagram of the system, showing the flow of information. Arrows represent the flow of data from one module to another.

**decibel** A measure of the relative amplitude of two voltages:  $dB = 20 \log_{10}(V_1/V_2)$ . It is also refers to the relative amplitude of two powers:  $dB = 10 \log_{10}(P_1/P_2)$ .

**denormalized** A denormalized number is an unnormalized floating-point number with an exponent of the smallest possible value. An unnormalized number has a mantissa value less than one. The mantissa of a normalized floating-point number is greater than or equal to 1, but strictly less than 2.

**desk-checking or dry run** We perform a desk check (or dry run) by determining in advance, either by analytical algorithm or explicit calculations, the expected outputs of strategic intermediate stages and final results for typical inputs. We then run our program and compare the actual outputs with this template of expected results.

**device driver** A collection of software routines that perform I/O functions.

**digital signal processing** Processing of data with digital hardware or software after the signal has been sampled by the ADC, e.g., filters, detection and compression/decompression.

**direction register** A bi-directional port configuration register that determines if the port will be an input or an output.

**disarm** Deactivate a trigger flag so that interrupts are not requested when that trigger flag is set.

**DMA** Direct Memory Access is a software/hardware synchronization method where the hardware itself causes a data transfer between the I/O device and memory at the appropriate time when data needs to be transferred. The software usually can perform other work while waiting for the hardware. No software action is required for each individual byte.

**double byte** Two bytes containing 16 bits. Same as halfword.

**double-pole switch** Two separate and complete switches that are activated together, same as two-pole. Contrast with single-pole.

**double-throw switch** A switch with three contact connections. The center contact will be connected exactly one of the other two contacts. Contrast with single-throw.

**download** The process of transferring object code from the host (e.g., the PC) to the target microcontroller.

**drop-out** An error that occurs after a right shift or a divide, and the consequence is that an intermediate result loses its ability to represent all of the values. E.g.,  $I=100*(N/51)$  can only result in the values 0, 100, or 200, whereas  $I=(100*N)/51$  properly calculates the desired result.

**duty cycle** For a periodic digital wave, it is the percentage of time the signal is high.

**dynamic efficiency** A measure of how fast the program executes.

**dynamic RAM** Volatile read/write storage built from a capacitor and a single transistor having a low cost, but requiring refresh. Contrast with static RAM.

**EEPROM** Electrically erasable programmable read only memory that is nonvolatile and easy to reprogram.

**effective address register (EAR)** A register that contains the address for the current memory cycle.

**embedded computer system** A system that performs a specific dedicated operation where the computer is hidden or embedded inside the machine. The system has intelligence in the software and physical connections to the real world. Same as cyber-physical system.

**emulator** An in-circuit emulator is an expensive debugging hardware tool that mimics the processor pin outs. To debug with an emulator, you would remove the processor chip and attach the emulator cable into the processor socket. The emulator would sense the processor input signals and recreate the processor outputs signals on the socket as if a real chip were actually there running at full speed. Inside the emulator you have internal read/write access to the registers and processor state. Most emulators allow you to visualize/record strategic information in real time without halting the program execution. You can also remove ROM chips and insert the connector of a ROM-emulator. This type of emulator is less expensive, and it allows you to debug ROM-based software systems.

**EPROM programmer** System hardware/software that burns the object code into the microcomputer's EPROM.

**EPROM** Same as PROM. Electrically programmable read only memory that is nonvolatile and requires external devices to erase and reprogram. It is usually erased using UV light.

**erase** The process of clearing the information in a PROM or EEPROM. The information bits are usually all set to logic 1.

**EVB** Evaluation Board, a product used to develop microcomputer software.

**even parity** A communication protocol where the number of ones in the data plus a parity bit is an even number. Contrast with odd parity.

**fan out** The number of inputs that a single output can drive if the devices are all in the same logic family.

**filter** In the debugging context, a filter is a Boolean function or conditional test used to make run-time decisions. For example, if we print information only if two variables  $x, y$  are equal, then the conditional ( $x==y$ ) is a filter. Filters can involve hardware status as well.

**Finite State Machine (FSM)** An abstract design method to build a machine with inputs and outputs. The machine can be in one of a finite number of states. Which state the system is in represents memory of previous inputs. The output and next state are a function of the input. There may be time delays as well.

**fixed point** A technique where calculations involving nonintegers are performed using a sequence of integer operations. E.g.,  $0.123*x$  is performed in decimal fixed point as  $(123*x)/1000$  or in binary fixed point as  $(126*x)>>10$ .

**flash EEPROM** Electrically erasable programmable read only memory that is nonvolatile and easy to reprogram. Flash EEPROMs are typically larger than regular EEPROM.

**floating** A logic state where the output device does not drive high or pull low. The outputs of open collector and tristate devices can be in the floating state. Same as HiZ.

**floor** Establishing a lower bound on the result of an operation.

**fork** Used in parallel programming to create additional software tasks that will run in parallel. See join.

**frame** A complete and distinct packet of bits occurring in a serial communication channel.

**framing error** An error when the receiver expects a stop bit (1) and the input is 0.

**friendly** Friendly software modifies just the bits that need to be modified, leaving the other bits unchanged.

**full-duplex channel** Hardware that allows bits (information, error checking, synchronization or overhead) to transfer simultaneously in both directions. Contrast with simplex and half-duplex channels.

**full-duplex communication** A system that allows information (data, characters) to transfer simultaneously in both directions.

**functional debugging** The process of detecting, locating, or correcting functional and logical errors in a program and the process of instrumenting a program for such purposes is called functional debugging or often simply debugging. Contrast with performance debugging.

**gadfly** A software/hardware synchronization method where the software continuously reads the hardware status waiting for the hardware operation to complete. The software usually performs no work while waiting for the hardware. Same as busy wait. Same as polling

**general purpose computer system** A system like the PC or Macintosh with a keyboard, disk and display that can be programmed for a wide variety of purposes.

**half-duplex channel** Hardware that allows bits (information, error checking, synchronization or overhead) to transfer in both directions, but in only one direction at a time. Contrast with simplex and full-duplex channels.

**half-duplex communication** a system that allows information to transfer in both directions, but in only one direction at a time.

**halfword** Two bytes containing 16 bits. Same as double byte. In C, we use **short** or **unsigned short** to create a halfword. In C99, we use **int16\_t** or **uint16\_t** to create a halfword.

**handshake** A software/hardware synchronization method where control and status signals go both directions between the transmitter and receiver. The communication is interlocked meaning each device will wait for the other.

**hard real-time system** as one that can guarantee that a process will complete a critical task within a certain specified range. In data acquisition system, hard real-time means there is an upper bound on the latency between when a sample is supposed to be taken (every 1/fs) and when the ADC conversion is actually started. Hard real-time also implies that no ADC samples are missed.

**heartbeat** A debugging monitor, such as a flashing LED, we add for the purpose of seeing if our program is running.

**hexadecimal** A number system that uses base 16.

**HiZ** A logic state where the output device does not drive high or pull low. The outputs of open collector and tristate devices can be in the HiZ state. Same as floating.

**hold time** When latching data into a device with a rising or falling edge of a clock, the hold time is the time after the active edge of the clock that the data must continue to be valid. Contrast with setup time.

**hysteresis** A condition when the output of a system depends not only on the input, but also on the previous outputs, e.g., a transducer that follows a different response curve when the input is increasing than when the input is decreasing.

**I/O bound** A situation where the input or output device is slower than the software. In other words it takes longer for the I/O device to process data, than for the software to process data.

**I/O device** A computer component capable of bringing information from the external environment into the computer (input device), or sending data out from the computer to the external environment (output device.)

**I/O port** A hardware device that connects the computer with external components.

**I<sub>IH</sub>** Input current when the signal is high.

**I<sub>IL</sub>** Input current when the signal is low.

**immediate** An addressing mode where the operand is a fixed data or address value.

**impedance** The ratio of the effort (voltage, force, pressure) divided by the flow (current, velocity, flow).

**incremental control system** A control system where the actuator has many possible states, and the system increments or decrements the actuator value depending on either in error is positive or negative.

**indexed** An addressing mode where the data or address value for the instruction is located in memory pointed to by an index register.

**infinite impulse response filter (IIR)** is a digital filter where the output is a function of an infinite number of past data samples, usually by making the filter output a function of previous filter outputs.

**input capture** A mechanism to set a flag and capture the current time (TCNT value) on the rising, falling or rising&falling edge of an external signal. The input capture event can also request an interrupt.

**input impedance** The input voltage divided by the input current. When a 3V input is applied to the TM4C123 ADC, a maximum of  $2\mu\text{A}$  will flow into the pin. Thus,  $Z_{\text{in}} = 3\text{V}/2\mu\text{A} = 1.5\text{M}\Omega$ .

**instruction register (IR)** Register in the control unit that contains the op code for the current instruction.

**instrument** An instrument is the code injected into a program for debugging or profiling. This code is usually extraneous to the normal function of a program and may be temporary or permanent.

Instruments injected during interactive sessions are considered to be temporary because these instruments can be removed simply by terminating a session. Instruments injected in source code are considered to be permanent because removal requires editing and recompiling the source. An example of a temporary instrument occurs when the debugger replaces a regular op code with a breakpoint instruction. This temporary instrument can be removed dynamically by restoring the original op code. A print statement added to your source code is an example of a permanent instrument, because removal requires editing and recompiling.

**instrument** A system that collects information, same as data acquisition system.

**instrumentation** The process of injecting or inserting an instrument.

**interrupt** A software/hardware synchronization method where the hardware causes a special software program (interrupt handler) to execute when its operation to complete. The software usually can perform other work while waiting for the hardware.

**interrupt flag** A status bit that is set by the hardware to signify an external event has occurred. Same as trigger flag.

**interrupt mask** A control bit that, if programmed to 1, will cause an interrupt request when the associated flag is set. Same as **arm**.

**interrupt polling** A software function to look and see which of the potential sources requested the interrupt.

**interrupt service routine (ISR)** Program that runs as a result of an interrupt.

**interrupt vector** 32-bit values at the beginning of memory specifying where the software should execute after an interrupt request. There is a unique interrupt vector for each type of interrupt.

**intrusive** A characteristic of a debugging instrument when the presence of the collection of information itself does significantly affect the parameters being measured.

**I<sub>OH</sub>** Output current when the signal is high. This is the maximum current that has a voltage above V<sub>OH</sub>.

**I<sub>OL</sub>** Output current when the signal is low. This is the maximum current that has a voltage below V<sub>OL</sub>.

**join** Used in parallel programming to combine two or more software tasks into one. Execution after a join will continue when all software tasks above the join are complete. See fork.

**kibibit** Stands for kilo-binary-bits, which is 1024 bits or 128 bytes, abbreviated Kibit.

**kibibyte** Stands for kilo-binary-bytes, which is 1024 bytes or 8192 bits, abbreviated KiB.

**latch** As a noun, it means a register. As a verb, it means to store data into the register.

**latched input port** An input port where the signals are latched (saved) on an edge of an associated strobe signal.

**latency** In this book latency usually refers to the response time of the computer to external events. For example, latency is the time between new input becoming available and the time the input is read by the computer. For example, the time between an output device becoming idle and the time the input is the computer writes new data to it. There can also be a latency for an I/O device, which is the response time of the external I/O device hardware to a software command. For a data acquisition system, the time between the time when the signal should be sampled and the time the ADC is actually started.

**LCD** Liquid Crystal Display, where the computer controls the reflectance or transmittance of the liquid crystal, characterized by its flexible display patterns, low power, low cost, and slow speed.

**LED** Light Emitting Diode, where the computer controls the electrical power to the diode, characterized by its simple display patterns, medium power, and high speed.

**linear recursion** A recursive technique that makes only one call to itself during the execution of the function. Linear recursive functions are easier to implement iteratively. We draw the execution pattern as a straight or linear path. See also recursion, binary recursion, and tail recursion.

**little endian** Mechanism for storing multiple byte numbers such that the least significant byte exists first (in the smallest memory address). Contrast with big endian.

**loader** System software that places the object code into the microcomputer's memory. If the object code is stored in EEPROM, the loader is also called an EEPROM programmer.

**logic analyzer** A hardware debugging tool that allows you to visualize many digital logic signals versus time. Real logic analyzers have at least 32 channels and can have up to 200 channels, with sophisticated techniques for triggering, saving and analyzing the real-time data.

**LSB** The least significant bit in a number system is the bit with the smallest significance, usually the right-most bit. With signed or unsigned integers the significance of the LSB is 1.

**maintenance** Process of verifying, changing, correcting, enhancing, and extending a system.

**mark** A digital value of true or logic 1 used in serial communication. Contrast with space.

**mask** As a verb, mask is the operation that selects certain bits out of many bits, using the logical and operation. The bits that are not being selected will be cleared to zero. When used as a noun, mask refers to the specific bits that are being selected.

**Mealy FSM** A FSM where both the output and next state are a function of the input and state.

**measurand** A signal measured by a data acquisition system.

**mebibit** Stands for mega-binary-bits, which is 1,048,576 bits, abbreviated Mibit.

**mebibyte** Stands for mega-binary-bytes, which is 1,048,576 bytes, abbreviated MiB.

**memory** A computer component capable of storing and recalling information.

**memory-mapped I/O** A configuration where the I/O devices are interfaced to the computer in a manner identical to the way memories are connected, from an interfacing perspective I/O devices and memory modules share the same bus signals, from a programmer's point of view the I/O devices exist as locations in the memory map, and I/O device access can be performed using any of the memory access instructions.

**Mibit** Stands for mega-binary-bits, which is 1,048,576 bits, same as mebibit.

**MiB** Stands for mega-binary-bytes, which is 1,048,576 bytes, same as mebibyte.

**microcomputer** An electronic device capable of performing input/output functions containing a microprocessor, memory, and I/O devices.

**microcontroller** A single chip microcomputer like the Texas Instruments TM4C123, Freescale 9S12, Intel 8051, Atmel ATmega328, Atmel SAM3X8E, PIC16, or the Texas Instruments MSP430.

**minimally intrusive** A characteristic of a debugging instrument when the presence of the collection of information itself has a small but insignificant effect on the parameters being measured.

**memonic** The symbolic name of an operation code, like **mov str push**.

**monitor or debugger window** A monitor is a debugger feature that allows us to passively view strategic software parameters during the real-time execution of our program. An effective monitor is one that has minimal effect on the performance of the system. When debugging software on a windows-based machine, we can often set up a debugger window that displays the current value of certain software variables.

**MSB** The most significant bit in a number system is the bit with the greatest significance, usually the left-most bit. If the number system is signed, then the MSB signifies positive (0) or negative (1).

**multiple access circular queue MACQ** A data structure used in data acquisition systems to hold the current sample and a finite number of previous samples.

**multi-threaded** A system with multiple threads (e.g., main program and interrupt service routines) that cooperate towards a common overall goal.

**negative logic** A signal where the true value has a lower voltage than the false value, in digital logic true is 0 and false is 1, in digital logic true is less than 0.7 volts and false is greater than 2 volts, in RS232 protocol true is -5.5 volts and false is +5.5 volts. Contrast with positive logic.

**nibble** 4 binary bits or 1 hexadecimal digit.

**nonatomic** Software execution that can be divided or interrupted. Most lines of C code require multiple assembly language instructions to execute, therefore an interrupt may occur in the middle of a line of C code. The instructions store and load multiple, **STM LDM**, are nonatomic.

**nonintrusive** A characteristic of a debugging instrument when the presence of the collection of information itself does not affect the parameters being measured. Nonintrusiveness is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in program performance by an instrument. For example, a print statement added to your source code and single-stepping are very intrusive because they significantly affect the real-time interaction of the hardware and software. When a program interacts with real-time events, the performance is significantly altered. On the other hand, an instrument with outputs strategic information on LEDs (that requires just 1  $\mu$ s to execute) is much less intrusive. A logic analyzer that passively monitors the address and data bus is completely nonintrusive. An in-circuit emulator is also nonintrusive because the software input/output relationships will be the same with and without the debugging tool.

**noninvasive/invasive** Noninvasiveness is the characteristic or quality of a debugger that makes the order of invocation immaterial. The debugger and the user program co-exist in the same global environment. On the other hand, an invasive debugger requires the user program to execute within an environment defined by the debugger. The debugger is invoked first and the program is then loaded either by the debugger or by the user from within the debugger. Invasiveness is also a measure of the degree of source code modification to debug or monitor a program. A resident debugger like the serial monitor is invasive because it exists first and then your program is loaded on top of it. This program development environment is invasive because the UART interrupts with the serial monitor is different from the eventual the single chip embedded application. An in-circuit emulator is non-invasive because it can coexist (be added or deleted) from our system without changing the way our system runs.

**nonreentrant** A software module that once started by one thread, cannot be interrupted and executed by a second thread. Nonreentrant modules usually involve nonatomic accesses to global variables or I/O ports: read modify write, write followed by read, or a multistep write.

**nonvolatile** A condition where information is not lost when power is removed. When power is restored, then the information is in the state that occurred when the power was removed.

**nonvolatile RAM** Read/write storage that achieves its long term storage ability because it includes a battery.

**normalized** The mantissa of a normalized floating-point number is greater than or equal to 1, but strictly less than 2.

**Nyquist Theorem** If a input signal is captured by an ADC at the regular rate of  $f_s$  samples/sec, then the digital sequence can accurately represent the 0 to  $\frac{1}{2}f_s$  frequency components of the original signal.

**object code** Programs in machine readable format created by the compiler or assembler.

**odd parity** A communication protocol where the number of ones in the data plus a parity bit is an odd number. Contrast with even parity.

**op code opcode or operation code** A specific instruction executed by the computer. The op code along with the operand completely specify the function to be performed. In assembly language programming, the op code is represented by its mnemonic, like **LDR**. During execution, the op code is stored as a machine code loaded in memory.

**open collector** A digital logic output that has two states low and HiZ. On CMOS circuits, it is sometimes called open drain.

**open drain** A CMOS digital logic output that has two states low and HiZ. Often used interchangeably with the term open collector.

**operand** The second part of an instruction that specifies either the data or the address for that instruction. An assembly instruction typically has an op code and an operand (e.g., #55). Instructions that use inherent addressing mode have no operand field.

**operating system** System software for managing computer resources and facilitating common functions like input/output, memory management, and file system.

**oscilloscope** A hardware debugging tool that allows you to visualize one or two analog signals versus time.

**output impedance** A specification of how strong an output signal is.  $Z_{out}$  is the open circuit output voltage divided by the short circuit output current. In the 2-bit DAC made with a  $10\text{k}\Omega$  and a  $20\text{k}\Omega$  resistor, if the both digital signals are high, the open circuit voltage is 3.3V. If the output of the DAC is shorted,  $3.3\text{V}/10\text{k}\Omega=0.33\text{mA}$  will flow through the  $10\text{k}\Omega$ , and  $3.3\text{V}/20\text{k}\Omega=0.165\text{mA}$  will flow through the  $20\text{k}\Omega$ , making a total short circuit current of about 0.5mA.  $Z_{out} = 3.3\text{V}/0.5\text{mA} = 6.6\text{k}\Omega$

**overflow** An error that occurs when the result of a calculation exceeds the range of the number system. For example, with 8-bit unsigned integers,  $200+57$  will yield the incorrect result of 1.

**overrun error** An error that occurs when the receiver gets a new frame but the data register and shift register already have information.

**parallel port** A port where all signals are available simultaneously. In this book the parallel ports are 8 bits wide. Some ports have less than 8 bits.

**parallel programming** A computer system that supports simultaneous execution of two or more software tasks. Compare to concurrent programming.

**PC-relative** An addressing mode where the effective address is calculated by its position relative to the current value of the program counter.

**performance debugging or profiling** The process of acquiring or modifying timing characteristics and execution patterns of a program and the process of instrumenting a program for such purposes is called performance debugging or profiling. Contrast with functional debugging.

**periodic polling** A software/hardware synchronization method that is a combination of interrupts and busy wait. An interrupt occurs at a regular rate (periodic) independent of the hardware status. The interrupt handler checks the hardware device (polls) to determine if its operation is complete. The software usually can perform other work while waiting for the hardware.

**personal computer system** A small general purpose computer system having a price low enough for individual people to afford and used for personal tasks.

**port** External pins through which the microcomputer can perform input/output. Same as I/O port.

**positive logic** a signal where the true value has a higher voltage than the false value, in digital logic true is 1 and false is 0, in digital logic true is greater than 2 volts and false is less than 0.7 volts, in RS232 protocol true is +5.5 volts and false is -5.5 volts. Contrast with negative logic.

**precision** For an input signal, it is the number of distinguishable input signals that can be reliably detected by the measurement. For an output signal, it is the number of different output parameters that can be produced by the system. For a number system, precision is the number of distinct or different values of a number system in units of “alternatives”. The precision of a number system is also the number of binary digits required to represent all its numbers in units of “bits”.

**priority** When two requests for service are made simultaneously, priority determines which order to process them. If we are processing a low priority task and a higher priority request is received, we will suspend the low priority task, execute the high priority task to completion, and then return to the lower priority task.

**private** Can be accessed only by software functions in that module. Contrast with public.

**private variable** A variable that is used by a single module, and not shared with other modules.

**process** The execution of software that does not necessarily cooperate with other processes. Contrast with thread. Processes generally do not share global memory or I/O devices.

**producer-consumer** A multi-threaded system where the producers generate new data, and the consumers process or output the data.

**program counter** (PC) A register in the processor that points to the memory containing the instruction to execute next.

**program status register** (PSR) Register in the processor that contains the status of the previous ALU operation, as well as some operating mode flags such as the interrupt enable bit.

**PROM** Same as EPROM. Programmable read only memory that is nonvolatile and requires external devices to erase and reprogram. It is usually erased using UV light. Contrast with EEPROM.

**promotion** Increasing the precision of a number for convenience or to avoid overflow errors during calculations.

**pseudo-code** A shorthand for describing a software algorithm. The exact format is not defined, but many programmers use their favorite high-level language syntax (like C) without paying rigorous attention to the punctuation.

**pseudo op** Operations included in the program that are not executed by the computer at run time, but rather are interpreted by the assembler during the assembly process. Same as assembly directive.

**public** Can be accessed by any software module. Contrast with private.

**public variable** A variable that is shared by multiple programs or threads.

**pulse width modulation** A technique to deliver a variable signal (voltage, power, and energy) using an on/off signal with a variable percentage of time the signal is on (duty cycle). Same as **variable duty cycle**.

**RAM** Random Access Memory, a type of memory where the information can be stored and retrieved easily and quickly. Since it is volatile the information is lost when power is removed.

**range** Includes both the smallest possible and the largest possible signal (input or output). The difference between the largest and smallest input that can be measured by the instrument. The units are in the units of the measurand. When precision is in alternatives, range=precision•resolution.

**real-time** A system that can guarantee an upper bound (worst case) on latency.

**real-time computer system** A system where time-critical operations occur when needed.

**recursion** A programming technique where a function calls itself. See also linear recursion, tail recursion, and binary recursion.

**reduced instruction set computer (RISC)** A computer with a few instructions, instructions with fixed lengths, instructions that execute in 1 or 2 bus cycles, only load and store can access memory, instructions that cannot read and write memory in the same bus cycle, many identical general purpose registers, and a limited number of addressing modes. Contrast to CISC.

**reentrant** A software module that can be started by one thread, interrupted and executed by a second thread. A reentrant module allows multiple threads to properly execute the desired function.

**registers** High-speed storage located in the processor. The registers in the ARM ® Cortex™-M processor include R0 through R15.

**reproducibility (or repeatability)** A parameter specifying how consistent over time the measurement is when the input remains fixed.

**requirements document** A formal description of what the system will do in a very complete way, but not including how it will be done. It should be unambiguous, complete, verifiable, and modifiable.

**reset vector** The 32-bit value at memory locations 4–7 specifying where the software should start after power is turned on or after a hardware reset.

**resolution** For an input signal, it is the smallest change in the input parameter that can be reliably detected by the measurement. For an output signal, it is the smallest change in the output parameter that can be produced by the system, range equals precision times resolution, where precision is given in alternatives.

**ritual** Software, usually executed once at the beginning of the program, that defines the operational modes of the I/O ports.

**ROM** Read Only Memory, a type of memory where the information is programmed into the device once, but can be accessed quickly. It is low cost, must be purchased in high volume, and can be programmed only once. See also PROM, EEPROM, and flash EEPROM.

**roundoff** The error that occurs in a fixed-point or floating-point calculation when the least significant bits of an intermediate calculation are discarded so the result can fit into the finite precision.

**sampling rate** The rate at which data is collected in a data acquisition system. Sampling rate applies to both the ADC while collecting data, and the DAC while outputting data.

**scan or scanpoint** Any instrument used to produce a side effect without causing a break (halt) is a scan. Therefore, a scan may be used to gather data passively or to modify functions of a program. Examples include software added to your source code that simply outputs or modifies a global variable without halting. A scanpoint is triggered in a manner similar to a breakpoint but a scanpoint simply records data at that time without halting execution.

**scope** A logic analyzer or an oscilloscope, hardware debugging tools that allows you to visualize multiple digital or analog signals versus time.

**semaphore** A system function with two operations (wait and signal) that provide for thread synchronization and resource sharing.

**sensitivity** The sensitivity of a transducer is the slope of the output versus input response. The sensitivity of a data acquisition system that detects events is the percentage of actual events that are properly recognized by the system.

**serial communication** A process where information is transmitted one bit at a time.

**serial peripheral interface (SPI)** device to transmit data with synchronous serial communication protocol. The clock is shared on both sides. Same as synchronous serial interface (SSI).

**serial port** An I/O port where the bits are input or output one at a time.

**setup time** When latching data into a device with a rising or falling edge of a clock, the setup time is the time before the active edge of the clock that the data must be valid. Contrast with hold time.

**signed two's complement binary** A mechanism to represent signed integers where 1 followed by all 0's is the most negative number, all 1's represents the value -1, all 0's represents the value 0, and 0 followed by all 1's is the largest positive number.

**sign-magnitude binary** A mechanism to represent signed integers where the most significant bit is set if the number is negative, and the remaining bits represent the magnitude as an unsigned binary.

**simplex channel** Hardware that allows bits (information, error checking, synchronization or overhead) to transfer only in one direction. Contrast with half-duplex and full-duplex channels.

**simplex communication** A system that allows information to transfer only in one direction.

**simulator** A simulator is a software application, which simulates or mimics the operation of a processor or computer system. Most simulators recreate only simple I/O ports and often do not effectively duplicate the real-time interactions of the software/hardware interface. On the other hand, they do provide a simple and interactive mechanism to test software. Simulators are especially useful when learning a new language, because they provide more control and access to the simulated machine, than one normally has with real hardware.

**single-pole switch** One switch that acts independent from other switches in the system. Contrast with double-pole.

**single-throw switch** A switch with two contact connections. The two contacts may be connected or disconnected. Contrast with double-throw.

**software interrupt vector** The 32-bit value at memory locations in low memory specifying where the software should go after executing a software interrupt instruction.

**software maintenance** Process of verifying, changing, correcting, enhancing, and extending software.

**source code** Programs in human readable format created with an editor.

**space** A digital value of false or logic 0 used in serial communication. Contrast with mark.

**specificity** The specificity of a transducer is the relative sensitivity of the device to the signal of interest versus the sensitivity of the device to other unwanted signals. The specificity of a data acquisition system that detects events is the percentage of events detected by the system that are actually true.

**stabilize** The process of stabilizing a software system involves specifying all its inputs. When a system is stabilized, the output results are consistently repeatable. Stabilizing a system with multiple real-time events, like input devices and time-dependent conditions, can be difficult to accomplish. It often involves replacing input hardware with sequential reads from an array or disk file.

**stack** Last in first out data structure located in RAM and used to temporarily save information.

**stack pointer (SP)** A register in the processor that points to the RAM location of the stack.

**start bit** An overhead bit(s) specifying the beginning of the frame, used in serial communication to synchronize the receiver shift register with the transmitter clock. See also stop bit, even parity and odd parity.

**static efficiency** A measure of program size, which is number of memory bytes required. In an embedded system we need to specify both RAM size for variables/stack and ROM size for programs/constants.

**static RAM** Volatile read/write storage built from three transistors having fast speed, and not requiring refresh. Contrast with dynamic RAM.

**stepper motor** A motor that moves in discrete steps.

**stop bit** An overhead bit(s) specifying the end of the frame, used in serial communication to separate one frame from the next. See also start bit, even parity and odd parity.

**string** A sequence of ASCII characters, usually terminated with a zero.

**symbol table** A mapping from a symbolic name to its corresponding 32-bit address, generated by the assembler in pass one and displayed in the listing file.

**synchronous protocol** a system where the two devices share the same clock.

**synchronous serial interface (SSI)** device to transmit data with synchronous serial communication protocol. The clock is shared on both sides. Same as serial peripheral interface (SPI).

**tachometer** a sensor that measures the revolutions per second of a rotating shaft.

**tail recursion** A technique where the recursive call occurs as the last action taken by the function. See also recursion, binary recursion, and linear recursion.

**thread** The execution of software that cooperates with other threads. A thread embodies the action of the software. One concept describes a thread as the sequence of operations including the input and output data. Contrast with process.

**throughput** The information transfer rate, the amount of data transferred per second. Same as bandwidth.

**time constant** The time to reach 63.2% of the final output after the input is instantaneously increased.

**time profile and execution profile** Time profile refers to the timing characteristic of a program and execution profile refers to the execution pattern of a program.

**toggle** Change 0 to 1 or 1 to 0. A toggle switch is one that if it is off when you push it, it will turn on. If it is on when you push it, it will turn off.

**transducer** A device that converts one type of signal into another type.

**trigger flag** A status bit that is set by the hardware to signify an external event has occurred. Same as interrupt flag.

**tristate** The state of a tristate logic output when HiZ or not driven.

**tristate logic** A digital logic device that has three output states low, high, and HiZ.

**truncation** The act of discarding bits as a number is converted from one format to another.

**two-pole switch** Two separate and complete switches, which are activated together, same as double-pole.

**two's complement** A number system used to define signed integers. The MSB defines whether the number is negative (1) or positive (0). To negate a two's complement number, one first complements (flip from 0 to 1 or from 1 to 0) each bit, then add 1 to the number.

**unary operation** A function that produces its result given a single input parameter. For example, negate, increment, and decrement are unary operations.

**unbuffered I/O** The hardware and software are tightly coupled so that both wait for each other during the transmission of data.

**unipolar stepper motor** A stepper motor where the current flows in only one direction (on/off) along the interface wires; a stepper with 5 or 6 interface wires.

**universal asynchronous receiver/transmitter (UART)** A device to transmit data with asynchronous serial communication protocol.

**unnormalized** An unnormalized floating-point number has a mantissa value less than one. The mantissa of a normalized floating-point number is greater than or equal to 1, but strictly less than 2.

**unsigned binary** A mechanism to represent unsigned integers where all 0's represents the value 0, and all 1's represents is the largest positive number.

**vector** An address at the end of memory containing the location of the interrupt service routines. See also reset vector and interrupt vector.

**V<sub>H</sub>** If the input voltage is above this value, the input is considered high.

**V<sub>L</sub>** If the input voltage is below this value, the input is considered low.

**V<sub>OH</sub>** The smallest possible output voltage when the signal is high, and the current is less than I<sub>OH</sub>.

**V<sub>OL</sub>** The largest possible output voltage when the signal is low, and the current is less than I<sub>OL</sub>.

**volatile** A condition where information is lost when power is removed. In C, **volatile** tells the compiler, the value may change beyond the control of the software itself.

**vulnerable window** Locations within a software module, which if an interrupt were to occur at one of these locations, then an error could occur (e.g., data lost, corrupted data, program crash, etc.)  
Same as critical section.

**white noise** A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as Johnson noise and thermal noise.

**word** Four bytes containing 32 bits. In C, we use **long** or **unsigned long** to create a word. In C99, we use **int32\_t** or **uint32\_t** to create a word.

**workstation** A powerful general purpose computer system having a price in the \$3K to 50K range and used for handling large amounts of data and performing many calculations.

**XON/XOFF** A protocol used by printers to feedback the printer status to the computer. XOFF is sent from the printer to the computer in order to stop data transfer, and XON is sent from the printer to the computer in order to resume data transfer.

## Appendix 2. Solutions to Checkpoints

**Checkpoint 1.1:** 2mA is 0.002A. Ohm's Law  $R=V/I = 1V/0.002A = 500\Omega$ .

**Checkpoint 1.2:** Ohm's Law  $I=V/R = 5V/100\Omega = 0.02A = 20mA$ .

**Checkpoint 1.3:** Theoretically, the current will be infinite, but practically there will be sparks.

**Checkpoint 1.4:** Resistance is effort over flow = Newtons/m<sup>2</sup>/ (m<sup>3</sup>/sec) = Newtons-sec/m<sup>5</sup>.

**Checkpoint 1.5:** Resistance is effort over flow = ° C/ watts.

**Checkpoint 1.6:** 2mA is 0.002A. Power is  $V*I = 1V*0.002A = 0.002W = 2\text{ mW}$ .

**Checkpoint 1.7:** Power is  $V^2/R = 5V*5V/100\Omega = 0.25W = 250\text{ mW}$ .

**Checkpoint 1.8:** Power is  $I^2*R = (0.09A)*(0.09A)*100\Omega = 0.81W$ .

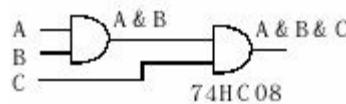
**Checkpoint 1.9:** Total resistance is  $1k\Omega + 2k\Omega = 3k\Omega$ . Ohm's Law  $V=I*R = 0.001A*3000\Omega = 3V$ .

**Checkpoint 1.10:** Total resistance is  $1k\Omega+2k\Omega=3k\Omega$ . I is  $6V/3k\Omega=2mA$ .  $V_2 = I*R_2 = 0.002A*2000\Omega = 4V$ .

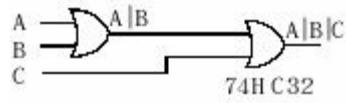
**Checkpoint 1.11:** Total resistance is  $2000*3000/(2000+3000)=1200\Omega$ .  $V=I*R = 0.001A*1200\Omega = 1.2\text{ V}$ .

**Checkpoint 1.12:** Ohm's Law  $I_2=V/R_2 = 6V/3000\Omega = 0.002A = 2\text{ mA}$

**Checkpoint 1.13:** Use 2 gates



**Checkpoint 1.14:** Use 2 gates



**Checkpoint 1.15:** Add the powers of 2 for each digit that is 1.

$$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 255$$

**Checkpoint 1.16:**  $15 \cdot 16^1 + 14 \cdot 16^0 = 254$

**Checkpoint 1.17:** First, divide the binary into 4-bit nibbles, then convert the two 4-bit nibbles:  $0100_2 = 0x4$  and  $0111_2 = 0x7$ . Third, combine the two hex digits into one number  $0x47$ .

**Checkpoint 1.18:** First, divide the binary into 4-bit nibbles, then convert the three 4-bit nibbles:  $1101_2 = 0xD$ ,  $1010_2 = 0xA$  and  $1011_2 = 0xB$ . Third, combine the three hex digits into one number  $0xDAB$

**Checkpoint 1.19:** First, convert the two 4-bit nibbles:  $0x4 = 0100_2$  and  $0x9 = 1001_2$ . Second, combine the 8 binary bits into one binary number  $01001001_2$

**Checkpoint 1.20:** First, convert the four 4-bit nibbles:  $0xB = 1011_2$ ,  $0xE = 1100_2$ ,  $0xE = 1100_2$  and  $0xF = 1111_2$ . Second, combine the 16 binary bits into one binary number  $1011110011001111_2$

**Checkpoint 1.21:** Four binary bits are required for each hex digit.  $4*5$  is 20 bits.

**Checkpoint 1.22:** There are 8 bits/byte, so 60 bits will take  $60/8 = 7.5$ , or 8 bytes of memory.

**Checkpoint 1.23:**  $3\frac{1}{2}$  decimal digits is about 2000 alternatives, which is about 11 bits.

**Checkpoint 1.24:** The rule of thumb says  $2^{60}$  is about  $10^{18}$ , which is 18 decimal digits.  $2^4$  is 16, which is about  $1\frac{1}{2}$  decimal digits. Together, we have  $19\frac{1}{2}$  decimal digits.

**Checkpoint 1.25:**  $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 64 + 32 + 8 + 2 + 1 = 107$

**Checkpoint 1.26:**  $4 * 16 + 6 = 64 + 6 = 70$

**Checkpoint 1.27:** We start by setting the running total to the number we wish to convert. We start with the basis element associated with the MSB and work towards the basis element for the LSB. We must also subtract basis elements from the running total as we determine they are needed. If the basis element in question is less than or equal to the running total, then we need that basis element.

**Checkpoint 1.28:** Combine binary basis elements to create the desired value.  $45 = 32 + 8 + 4 + 1$ , so  $45 = 00101101_2 = 0x2D$ .

**Checkpoint 1.29:** Combine binary basis elements to create the desired value.  $200 = 128 + 64 + 8$ , so  $200 = 11001000_2 = 0xC8$ .

**Checkpoint 1.30:** Combine signed binary basis elements to create the desired value.  
 $-128 + 64 + 32 + 8 + 2 = -22$ .

**Checkpoint 1.31:** They are the same, because bit 7 is zero.

**Checkpoint 1.32:** Combine signed binary basis elements to create the desired value.  $-45 = -128 + 64 + 16 + 2 + 1 = 11010011_2 = 0xD3$ .

**Checkpoint 1.33:** Because the range of 8-bit signed numbers is -128 to +127.

**Checkpoint 1.34:** Each four bits represent a single decimal digit,  $0x25 = 00100101_2$ .

**Checkpoint 1.35:**  $8192 + 64 + 32 + 8 + 2 = 8298$ .

**Checkpoint 1.36:**  $1 * 4096 + 2 * 256 + 3 * 16 + 4 = 4660$ .

**Checkpoint 1.37:**  $1234 = 4 * 256 + 13 * 16 + 2 = 0x04D2$ .

**Checkpoint 1.38:**  $10000 = 8192 + 1024 + 512 + 256 + 16 = 0010011100010000_2$ .

**Checkpoint 1.39:**  $1 * 4096 + 2 * 256 + 3 * 16 + 4 = 4660$ .

**Checkpoint 1.40:**  $-32768 + 2 * 4096 + 11 * 256 + 12 * 16 + 13 = -21555$ .

**Checkpoint 1.41:**  $1234 = 4 * 256 + 13 * 16 + 2 = 0x04D2$ .

**Checkpoint 1.42:**  $-10000 = -32768 + 16384 + 4096 + 2048 + 128 + 64 + 32 + 16 = 1101100011110000_2$ .

**Checkpoint 1.43:** Looking in the ASCII table we see ‘0’ is  $0x30$  (or 48).

**Checkpoint 1.44:** Let  $c$  be the character ‘0’ to ‘9’,  $n = c - 0x30$ .

**Checkpoint 1.45:** Look up each letter, concatenate, add 0 at end,  $0x48656C6C6F20576F726C6400$ .

**Checkpoint 1.46:** A microprocessor is a small processor. A microcomputer is a small computer that includes a processor, memory and I/O devices. A microcontroller is a single chip computer.

**Checkpoint 1.47:** Flash ROM is higher density because it requires few transistors compared to RAM.

**Checkpoint 2.1:** An embedded system is a microcomputer with mechanical, chemical, and electrical devices attached to it, programmed for a specific dedicated purpose, and packaged up as a complete system.

**Checkpoint 2.2:** A microcomputer is a small computer that includes a processor, memory and I/O devices.

**Checkpoint 2.3:** Typical input devices include the keys on the keyboard, mouse and its buttons, touch pad, DVD reader, and microphone. USB drives, Ethernet, and wireless can be used for input and output.

**Checkpoint 2.4:** Typical output devices include the LEDs on the keyboard, monitor, speaker, printer, DVD burner, and speaker. USB drives, Ethernet, and wireless can be used for input and output.

**Checkpoint 2.5:** The software in a digital watch must maintain time using a real-time clock, output the current time on the LCD, respond to button pushes updating parameters as required, check and see if the current time matches the alarm time.

**Checkpoint 2.6:** Both terms refer to parameters of a system, but the differences lie in the level of detail used to describe the parameter. A requirement is usually defined in general terms, whereas a specification entails detailed engineering rigor. A requirement often refers to an objective of the system, while a specification describes how well the actual device works.

**Checkpoint 2.7:** It failed because employees were rewarded for poor behavior. It is much better to punish poor behavior and reward good behavior.

**Checkpoint 2.8:** In general, the presence of a minimally intrusive debugging instrument itself only has minimal effect on the parameter being measured. One criterion is the total execution time required to perform the instrumentation is small compared to the execution times of the original target operation.

**Checkpoint 2.9:** Runtime debugging can be activated in final production systems. Runtime debugging is quicker to activate/deactivate because an edit/assemble/download cycle is not needed. Assembly-time debugging produces a final production system that runs faster and requires less memory.

**Checkpoint 2.10:** We are sure we debugged the exact system that is being manufactured. The debugging statements can be used to evaluate the proper operation of systems before they are shipped. The instruments can also be used to diagnose and repair systems.

**Checkpoint 2.11:** The  $V_{OL}$  of the LED driver is still 0.5V.  $R = (3.3 - 1.7 - 0.5) / 0.012 = 92 \square \Omega$ .

**Checkpoint 2.12:** Negative logic interface: The  $V_{OL}$  is still 0.4V.  $R = (3.3 - 1.7 - 0.4) / 0.002 = 600\Omega$ .

**Checkpoint 3.1:** The addressing mode defines the format for the effective address for that instruction. In other words, it defines how the instruction will access the data it needs.

**Checkpoint 3.2:** 0x2000.0008, R3 is not changed.

**Checkpoint 3.3:** 0x2000.0000, and 8 is added to R3, it becomes 0x2000.0008.

**Checkpoint 3.4:** Bit-wise AND.  $0x12345678 \& 0x87654321 = 0x02244220$ .

Bit-wise EOR.  $0001^{\wedge}1000=1001$ .  $0010^{\wedge}0111=0101$ .  $0011^{\wedge}0110=0101$ .  $0100^{\wedge}0101=0001$ .

So  $0x12345678 ^{\wedge} 0x87654321 = 0x95511559$ .

**Checkpoint 3.5:**



**Checkpoint 3.6:** Read N, shift, store into M

**LDR R2,=N ; R2 = &N**

**LDRSH R1,[R2] ; R1 = N (16-bit signed)**

**LSL R0,R1,#2 ; R0 = N<<2**

**LDR R2,=M ; R2 = &M**

**STRH R0,[R2] ; M = 4\*N**

**Checkpoint 3.7:** 32 bits plus 32 bits yields 33 bits

**Checkpoint 3.8:** 32 bits plus 32 bits yields 33 bits

**Checkpoint 3.9:**  $2^{32}-1$  to 0, or 4,294,967,295 to 0.

**Checkpoint 3.10:**  $0x7000.0000 + 0x2000.0000 = 0x9000.0000$ . N=1 because result is negative. Z=0 because result is not zero. V=1 because a positive number was added to a positive number yielding a negative result. C=0 because the unsigned sum is less than  $2^{32}$ .

**Checkpoint 3.11:**  $0x0000.0001 + 0xFFFF.FFFF = 0x0000.0000$ . N=0 because result is positive. We consider zero a positive number. Z=1 because result is zero. V=0 because a positive number was added to a negative number. C=1 because the unsigned sum is equal to  $2^{32}$  passing the discontinuity.

**Checkpoint 3.12:**  $100 - 200 = -100$ . N=1 because result is negative. Z=0 because result is not zero. V=0 because the result is correct. Remember, crossing the discontinuity clears the carry, not crossing sets the carry. C=0 because the difference crosses the discontinuity. Thinking another way,  $100 - 200$  causes an unsigned overflow, so C=0.

**Checkpoint 3.13:**  $100 - -200 = 300$ . N=0 because result is positive. Z=0 because result is not zero. V=0 because the result is correct. Remember, crossing the discontinuity clears the carry, not crossing sets the carry. C=0 because the difference crosses the discontinuity. -200 is the same number as 4294967096. Thinking another way,  $100 - 4294967096$  causes an unsigned overflow, so C=0.

**Checkpoint 3.14:** unsigned 16-bit means using LDRH and STRH

```
LDR R3, =N ; R3 = &N  
LDRH R1, [R3] ; R1 = N  
ADD R0, R1, #10 ; R0 = N+10  
LDR R2, =M ; R2 = &M  
STRH R0, [R2] ; M = N+10
```

**Checkpoint 3.15:** signed 8-bit means using LDRSB and STRB

```
LDR R3, =N ; R3 = &N  
LDRSB R1, [R3] ; R1 = N  
ADD R0, R1, #10 ; R0 = N+10  
LDR R2, =M ; R2 = &M  
STRB R0, [R2] ; M = N+10
```

**Checkpoint 3.16:** Change ADDS to SUBS , the rest is the same

**Checkpoint 3.17:** 32 bits times 32 bits yields 64 bits

**Checkpoint 3.18:** 32 bits times 32 bits yields 64 bits

**Checkpoint 3.19:** No, because the product is 64 bits

**Checkpoint 3.20:** dividend=quotient\*divisor+remainder.

**Checkpoint 4.1:** Nothing happens if the software writes to an input port.

**Checkpoint 4.2:** If the software reads this output port it gets the values last written to the port. For example, if the user mistakenly grounded the output pin (very bad thing to do), and the software writes a '1'; when it reads it will get '1'.

**Checkpoint 4.3:** Since there are as many bits in a port as there are bits in the direction register, each bit can be individually programmed as input or output.

**Checkpoint 4.4:** First, write a 0x00000002 to the clock register to activate Port B. Second, change all the labels PORTF to PORTB (change all 0x40025 to 0x40005). If it is an LM3S microcontroller, remove steps 2, 3, and 4\*\*\*\*fix\*\*\*\*.

**Checkpoint 4.5:** Nothing happens. Since none of the address bits are selected, none of the port bits are affected.

**Checkpoint 4.6:** The base address for Port A is 0x4000.4000.

```
#define PA71 ((volatile uint32_t *)0x40004208))
```

**PA71 = 0x82; // sets PA7 PA1, other 6 bits are not affected**

**Checkpoint 4.7:** The base address for Port B is 0x4000.5000.

**#define PB610 (\*((volatile uint32\_t \*)0x4000510C))**

**PB610 = 0x43; // sets PB6 PB1 PB0, other 5 bits are not affected**

**Checkpoint 4.8:** 16 MHz means 62.5ns per cycle. It takes 6 cycles to be high and 6 cycles to be low.

There are 14 cycles per period of the squarewave. So, the frequency is  $16 \text{ MHz}/12 = 1.33 \text{ MHz}$

**Checkpoint 4.9:** It will still operate according to specifications, but it may be more expensive to build or it may be harder to order components to build it.

**Checkpoint 4.10:** It will no longer operate according to specifications.

**Checkpoint 4.11:** Change the specification from 16 MHz to 8 MHz. Change the line

**SYSCTL\_RCC\_R += 0x00000540; // 10101, configure for 16 MHz crystal**

to

**SYSCTL\_RCC\_R += 0x00000380; // 01110, configure for 8 MHz crystal**

Change the specification from divide by 5 to divide by 8. Change the line

**SYSCTL\_RCC2\_R += (4<<22); // configure for 80 MHz clock**

to

**SYSCTL\_RCC2\_R += (7<<22); // configure for 50 MHz clock**

**Checkpoint 4.12:** Change SysTick\_Wait(500000); to SysTick\_Wait(120000); .

**Checkpoint 5.1:** The initialization in one module may modify the configuration needed by another module. To resolve, assign individual port pins to only one module, and make all the initializations friendly.

**Checkpoint 5.2:** Coupling can be parameters passed, shared globals, functions called, and shared I/O devices. It is measured in bytes/sec transferred from one module to the other.

**Checkpoint 5.3:** They are permanently allocated and can be accessed by any function.

**Checkpoint 5.4:** Use a “does-call” graph so only one module actually does access the I/O device.

**Checkpoint 5.5:** Use **LDRB** , because the number is 8 bits unsigned. Bring first number into a register, subtract second number. Since we want to call the subroutine if equal, we skip over if not equal.

**LDR R2, =N ; R2 = &N**

**LDRB R0, [R2] ; R0 = N (unsigned)**

**CMP R0, #25 ; is N == 25 ?**

**BNE next1 ; if not, skip**

**BL isEqual ; N == 25**

**next1**

**Checkpoint 5.6:** Use **LDRH** , because the number is 16 bits unsigned. Bring first number into a register, subtract second number. Since we want to call the subroutine if equal, we skip over if not equal.

**LDR R1, =H1 ; R1 = &H1**

**LDRH R1, [R1] ; R1 = H1 (unsigned)**

**LDR R2, =H2 ; R2 = &H2**

**LDRH R2, [R2] ; R2 = H2 (unsigned)**

**CMP R1, R2 ; is H1 == H2 ?**

**BNE next2 ; if not, skip**

**BL isEqual ; H1 == H2**

**Next2**

**Checkpoint 5.7:** It determines whether to use the **BLS** or **BLE** instruction.

**Checkpoint 5.8:** Use **LDRSH**, because the number is 16 bits signed.

**LDR R2, =M ; R2 = &M**

**LDRSH R0, [R2] ; R0 = M (signed)**

**CMP R0, #1000 ; is M > 1000 ?**

**BGT high ; if so, skip to high**

**low BL isLessEq ; M <= 1000**

**B next ; unconditional**

**high BL isGreater ; M > 1000**

**next**

**Checkpoint 5.9:** Use **LDRH**, because the number is 16 bits unsigned.

**LDR R4, =N ; R4 = &N (use R4 for AAPCS)**

**loop LDRH R0, [R4] ; R0 = N (unsigned)**

**CMP R0, #25 ; is N == 25?**

**BEQ next ; if so, skip to next**

**BL Body ; body of the loop**

**B loop**

**next**

**Checkpoint 5.10:** The macro runs faster. If the subroutine/macro is called/invoked from one or two locations in our software, then the macro will also require less storage.

**Checkpoint 5.11:** Recursive version requires 8 bytes, four bytes to save R0 and four bytes for the return address. It is called five times, so 40 bytes are required.

**Checkpoint 5.12:** Public functions have an underline. E.g., **UART\_OutString**. Private functions do not have an underline. E.g., **SetBaud**.

**Checkpoint 5.13:** Local variables begin with a lower case letter E.g., **myKey**. Global variables begin with an upper case letter E.g., **TheKey**.

**Checkpoint 5.14:** Each conditional branch creates two potential execution paths. Twenty conditional branches might create  $2^{20}$ , which is about a million, potential paths. In most cases, the actual number of paths will be much less, because taking one branch path usually prevents other conditional branches from being executed.

**Checkpoint 5.15:** The assembler determines the size of each instruction. Using the **AREA** statements it will create a symbol table mapping the symbols into physical addresses.

**Checkpoint 5.16:** The assembler determines the machine code for each instruction and creates the listing file.

**Checkpoint 6.1:** To make it easier to understand.

**Checkpoint 6.2:** Create it using an 80-byte size, and just waste the space when less than 80 bytes are requested.

**Checkpoint 7.1:** Define the variable within the scope of the function. E.g.,

```
void MyFunction(void){ int32_t myLocalVariable;
```

```
}
```

**Checkpoint 7.2:** Define the variable outside the scope of the function. E.g.,

```
int32_t myGlobalVariable; // accessible by all programs
```

```
void MyFunction(void){
```

```
}
```

**Checkpoint 7.3:** Look for this line in the startup.s file. It contains the stack size in bytes

```
Stack EQU 0x00000400
```

**Checkpoint 7.4:** Multiply R0 by 4 and subtract it from SP. **LSL R0,R0,#2** then **SUB SP,SP,R0**

**Checkpoint 7.5:** Three 32-bit variables is 12 bytes

Func **SUB SP,#12 ;1) allocate local variables**

**;2) body**

**ADD SP,#12 ;3) deallocate local variables**

**BX LR**

**Checkpoint 7.6:** In call by value, we pass a copy of the data (which may be result of an expression).

In call by reference, we pass a pointer to the data such that the calling program and the subroutine are accessing the same data.

**Checkpoint 7.7:**  $\pi \times 1000$  is about 3141.59, so the variable integer part is 3142.

**Checkpoint 7.8:**  $\pi \times 256$  is about 804.2477, so the variable integer part is 804.

**Checkpoint 7.9:**  $F = (461 \cdot C) / 256 + 32$ .

**Checkpoint 7.10:**  $y = (1000 \cdot x_1 - 53 \cdot x_1 + 1000 \cdot x_2 + 51 \cdot y_1 - 903 \cdot y_2) / 1000$ .

**Checkpoint 7.11:** Simply,  $R3 = (R1 \cdot R2) / (R1 + R2)$ , because the fixed constants factor out.

**Checkpoint 8.1:** There is 1 byte of data per 10 bits of transmission. So, there are 100 bytes/sec.

**Checkpoint 8.2:** divider = 0010.100000<sub>2</sub>, or 2 plus 32/64 = 2.5. The baud rate is 10MHz/2.5/16 which is 250 kHz.

**Checkpoint 8.3:**  $50,000,000 / 38400 / 16$  is 81.3802, which is similar to 81 and

24/64. **UART0\_IBRD\_R** is 81 **UART0\_FBRD\_R** is 24. The baud rate is  $50\text{MHz} / (81 + 24/64) / 16$  which is 38402 bits/sec.

**Checkpoint 8.4:** RXFE is set and cleared by hardware. It means receive FIFO empty. To make it 0 means to put data into the FIFO. Software cannot clear this flag. An incoming UART frame will clear RXFE.

**Checkpoint 8.5:** TXFF is set and cleared by hardware. It means transmit FIFO full. To make it 0 means to get data from the FIFO. Software cannot clear this flag. An outgoing UART frame will clear TXFF.

**Checkpoint 8.6:** The data will be received in error (values will not be correct). The receiver could appear to get two input frames for every one frame transmitted. It will probably cause framing errors (FE). It would cause parity errors if active.

**Checkpoint 8.7:** The data will be received in error (values will not be correct). The receiver will appear to get one input frame for every one frame transmitted. It will probably not cause framing errors (FE). It would cause parity errors if active.

**Checkpoint 8.8:**  $10,000,000/115200/16$  is 5.4253, which is similar to 5 and  $27/64$ . **UART0\_IBRD\_R** is 5 **UART0\_FBRD\_R** is 27. The baud rate is  $10\text{MHz}/(5+27/64)/16$  which is 115274 bits/sec.

**Checkpoint 8.9:** Setup time is the time before a clock input data must be valid. Hold time, the time after a clock input data must continue to be valid.

**Checkpoint 8.10:** Speed =  $(1 \text{ rotation}/36 \text{ steps}) * (1000\text{ms}/\text{s}) * (60\text{sec}/\text{min}) * (1\text{step}/50\text{ms}) = 33.3 \text{ RPM}$

**Checkpoint 8.11:** Change the 50ms to 10ms, and it will spin 5 times faster.

Speed =  $(1 \text{ rotation}/200 \text{ steps}) * (1000\text{ms}/\text{s}) * (60\text{sec}/\text{min}) * (1\text{step}/10\text{ms}) = 30 \text{ RPM}$

**Checkpoint 9.1:** Trigger flag set by hardware; the device is armed by software; the device is enabled for interrupts in the NVIC; the processor is enabled for interrupts (**PRIMASK** I bit is clear); the interrupt level must be less than the **BASEPRI**. The order of these conditions does not matter.

**Checkpoint 9.2:** The processor is enabled for interrupts by clearing the I bit in the **PRIMASK**. Execute

### CPSIE I

**Checkpoint 9.3:** Instruction is finished; registers R0–R3, R12, LR, PC, and PSR are pushed; LR is set to 0xFFFFFFFF9; **IPSR** is set to the interrupt number being processed; PC is set with interrupt vector address. The last three steps can occur in any order.

**Checkpoint 9.4:** From Program 9.1 or Table 9.1 we see the vector is 32 bits at 0x0000003C. The standard name of the interrupt handler is **SysTick\_Handler**.

**Checkpoint 9.5:** Negative logic means when we touch the switch the voltage goes to 0 (low). Formally, negative logic means the true voltage is lower than the false voltage. Positive logic means when we touch the switch the voltage goes to +3.3 (high). Formally, positive logic means the true voltage is higher than the false voltage.

**Checkpoint 9.6:** For PA2, we need input with pull-up. DIR bit 2 is low (input), AFSEL bit 2 is low (not alternate), PUE bit 2 high (pull-up) and PDE bit 2 low (not pull-down). For PA3, we need input with pull-down. DIR bit 3 is low (input), AFSEL bit 3 is low (not alternate), PUE bit 3 low (no pull-up) and PDE bit 3 high (pull-down).

**Checkpoint 9.7:** The timer counts down, and when it hits zero it reloads and continues to count. The TATORIS flag is set when the timer rolls over.

**Checkpoint 9.8:** This bit-specific address makes the access friendly

```
#define PB3 ((volatile uint32_t *)0x40005020)
```

The first instrument sets it high and the second sets it low

```
PB3 = 0x08; // high
```

```
PB3 = 0x00; // low
```

**Checkpoint 10.1:** Because the frequency components of the wiggles are higher than  $\frac{1}{2}$  the sampling rate. The Nyquist Theorem is violated.

**Checkpoint 10.2:** Because temperatures above  $31^{\circ}\text{C}$  are beyond the range, which is defined in this example as 0 to  $31^{\circ}\text{C}$ .

**Checkpoint 10.3:** If the sampling rate is 1 Hz, according to the Nyquist Theorem, the digital data can reliably represent frequencies from 0 to  $\frac{1}{2}$  Hz.

**Checkpoint 10.4:** According to the Nyquist Theorem we would have to output to the DAC at 2 kHz.

**Checkpoint 10.5:**  $2.5\text{V}/256$  is about 0.01 V or 10 mV.

**Checkpoint 10.6:**  $2\text{V}/1\text{mV}$  is 2000 alternatives. This is about 11 bits.

**Checkpoint 10.7:** Q2 is connected to  $10\text{k}\Omega$ , Q1 is connected to  $20\text{k}\Omega$  and Q0 is connected to  $40\text{k}\Omega$ , the other ends of the resistors are connected together. Any three resistors with a 1/2/4 ratio would be ok.

**Checkpoint 10.8:** Approximating the 10-bit ADC is linear, either  $D_{\text{out}} = 1024*\text{V}_{\text{in}}/3$  or  $1023*\text{V}_{\text{in}}/3 = 341$ .

**Checkpoint 10.9:** Approximating the 12-bit ADC is linear, either  $D_{\text{out}} = 4096*\text{V}_{\text{in}}/3.3$  or  $4095*\text{V}_{\text{in}}/3.3 = 1241$ .

**Checkpoint 10.10:** This is the expected result. This means it can resolve differences in input voltage at about 1 ADC value. Note the data in Figure 10.13 was collected with 64-sample hardware averaging.

**ADC0\_SAC\_R = 0x06;**

**Checkpoint 11.1:** At the end of the inner nested program, interrupts would be enabled. So, the last part of the outer section would be running with interrupts enabled. In this example **Stuff2B** runs with interrupts enabled.

**Critical1**

**Disable**

**Stuff1A**

**Call Critical2**

**Stuff1B**

**Enable**

**return**

**Critical2**

**Disable**

**Stuff2A**

**Enable**

**return**

**Checkpoint 11.2:** There are two ways a FIFO can get full. If the average rate at which data is put in the FIFO is larger than the average rate data is get from the FIFO, then the FIFO will always fill up. If the temporary rate at which data is put in the FIFO is larger than the temporary rate data is get from the FIFO, and the FIFO size is small, then the FIFO may fill up.

**Checkpoint 11.3:** The RxFifo is empty when there is no input data. Software is waiting for hardware.

**Checkpoint 11.4:** The TxFifo is empty when there is no output data. Hardware is waiting for software.

**Checkpoint 11.5:** No, if the average producer rate exceeds the average consumer rate, the FIFO will always fill regardless of size. However, if the average producer rate is less than the average consumer rate, the FIFO full errors can be eliminated by increasing size.

**Checkpoint 11.6:** With open collector outputs, the low will dominate over HiZ. The signal will be low.

**Checkpoint 11.7:** The minimum is 1 and the maximum is N-1. On average, it will take  $N/2$  transmissions for the message to go from one computer to another. There are 10 bits/frame, so there are 10,000 bytes/sec. Because there are 10 bytes/message, it takes 1ms to transmit a message. Because it has to be sent 5 times, it takes 5ms on average.

**Checkpoint 11.8:** The frame sent by a transmitter is echoed to its own receiver. If the data does not match, or if there are any framing or noise errors then a collision occurred.

**Checkpoint 11.9:** Parity could be used to detect collisions. Also the message could have checksum added. Framing or noise errors can also indicate a collision.

# Appendix 3. How to Convert Projects from Keil to CCS

Most of the examples in this book follow the Keil™ uVision® syntax. An equally powerful code development tool is the Texas Instruments Code Composer Studio™. The purpose of this Appendix is to illustrate how to convert files from Keil to CCS. Program A3.1 shows the equivalent code and order of use in an assembly file. The subroutine will input from Port A bit 5 and store the value into global variable **M** (0 or 0x20).

;Keil	;CCS
THUMB	.thumb ;1)
AREA DATA, ALIGN=2	.data ;2)
EXPORT M	.align 4 ;3)
M SPACE 4	.global M ;4)
AREA	M .field 32 ;5)
.text, CODE, READONLY, ALIGN=2	.align 2 ;6)
PORATA EQU 0x400043FC	.text ;7)
BIT5 EQU 0x20	PtM .field M,32 ;8)
EXPORT InputPA5	PORTA .field 0x400043FC,32
InputPA5	;8)
LDR R0,=PORATA ;R0 =	BIT5 .equ 0x20 ;9)
&PORTA	.global InputPA5 ;10)
LDR R1,[R0] ;R1 = PORTA	.thumbfunc InputPA5 ;11)
AND R1,R1,#BIT5 ;Mask	InputPA5: .asmfunc ;12)
LDR R2,=M ;R2 = &M	LDR R0,PORTA
STR R1,[R2] ;M = PA5	;13)
BX LR	LDR R1,[R0]
END	AND R1,R1,#BIT5
	LDR R2,PtM ;13)
	STR R1,[R2]
	BX LR
	.endasmfunc ;12)
	.end ;14)

Program A3.1. This illustrates the order and syntax of pseudo-ops in assembly files.

- 1) Use Thumb assembly language
- 2) This is a data section (variables typically go in RAM)
- 3) Align on 32-bit boundary
- 4) Declare the variable **M** globally visible to other files including to C programs
- 5) Define an uninitialized 32-bit object and call it **M**
- 6) Align on 16-bit boundary
- 7) This is a text section, which is executable code and callable from C (in ROM)
- 8) **.field** defines 32-bit objects and initialize them as pointers to **M** and to Port A
- 9) **.equ** defines a numerical constant

- 10) Declare it globally visible to other files including to C programs
- 11) There is a thumb function with this name
- 12) **.asmfunc** and **.endasmfunc** help with debugging, marking beginning and end
- 13) A pointer-constant is stored in ROM, and PC relative addressing is used
- 14) Marks the end of the file

One of the difficulties in translating Keil to CCS is that the Keil syntax of **LDR R#,=Label** is not supported in CCS. So, to access variables and I/O ports we need to define a 32-bit pointer-constant using the **.field** pseudo-op. The actual machine code created by these two assemblers is virtually identical. The only difference is where in ROM the pointer-constant resides. In CCS you explicitly position the pointer-constants, and in Keil, the assembly automatically positions them.

In CCS there MUST be a ‘main’ function, if you have to you can alias it using substitution of symbols

**.asg “main”, XXXXXXXX**

where **XXXXXXXX** is the function name you want to substitute for main

In Keil you could write these four invalid instructions

```
AND R0,R1,#0x00FFFFFF
MOV R1,#-1
ORR R2,#0x0FFFFFFF
CMP R3,#-100
```

and it would be automatically converted to equivalent valid instructions

```
BIC R0,R1,#0xFF000000
MVN R1,#0
ORN R2,#0xF0000000
CMN R3,#100
```

In CCS you have to do this manually.

Each compiler has its own syntax for handling inline assembly. The syntax for inline assembly in C is illustrated in Program A3.2. Both compilers follow the AAPCS convention for passing parameters and saving registers.

<pre>// Keil __asm void Delay(uint32_t ulCount){     subs r0, #1     bne Delay     bx lr }</pre>	<pre>// CCS void Delay(uint32_t ulCount){     __asm ( "    subs    r0, #1\n"             "    bne     Delay\n"             "    bx      lr\n"); }</pre>
--	---

Program A3.2. This illustrates inline assembly in C programs.

The CCS code requires the quotation marks with a new line character at the end of each assembly line. This is a clever hack around to enable multiple lines to be written as one line. In essence Keil allows straight inline assembly, whereas in CCS you have to specify it as a string that will then be inserted. If you have to use assembly it is better to place it in a separate file, because inline assembly can be difficult to debug and makes the code less portable.

The example files of this book are posted on the book's web site and have versions for both compilers. For help with CCS equivalents please reference the document spnu118j.pdf (which can be found on [www.TI.com](http://www.TI.com)).

# Appendix 4. Assembly Reference

My purpose in writing this book is not to provide a complete description of the ARM®Cortex M or any LM3S/TM4C microcontroller. Rather the book is a learning tool for first year college students majoring in engineering and science. As such, this appendix is not a complete list of all Thumb instructions. It gives details on the subset of instructions introduced in this book. Depending on exactly how you count, the Cortex M processor has over 150 instructions. However, I think this subset of 30 instructions will be sufficient to perform the homework and labs associated with the book. As the book evolves I will consider adding instructions to this subset that enhance the learning objectives without adding unnecessary complexity. Once you finish reading this book, I encourage you to propose to me Thumb instructions you feel every freshman engineer or scientist needs to know. On the other hand, these three are complete reference manuals for the Cortex-M processor. They are available as pdf files and are posted on the book web site.

CortexM_InstructionSet.pdf	Cortex-M3/M4 Instruction Set Technical User's Manual
CortexM4_TRM_r0p1.pdf	Cortex-M4 Technical Reference Manual
QuickReferenceCard.pdf Card	ARM® and Thumb-2 Instruction Set Quick Reference

Table 3.2, repeated here, shows the conditions **{cond}** that we will use for conditional branching. Conditional instructions, except for conditional branches, must be inside an If-Then (IT) instruction block. Depending on the vendor, the assembler might automatically insert an IT instruction if you have conditional instructions outside the IT block.

Suffix	Flags	Meaning
<b>EQ</b>	Z = 1	Equal
<b>NE</b>	Z = 0	Not equal
<b>CS or HS</b>	C = 1	Higher or same, unsigned $\geq$
<b>CC or LO</b>	C = 0	Lower, unsigned <
<b>MI</b>	N = 1	Negative
<b>PL</b>	N = 0	Positive or zero
<b>VS</b>	V = 1	Overflow
<b>VC</b>	V = 0	No overflow
<b>HI</b>	C = 1 and Z = 0	Higher, unsigned >
<b>LS</b>	C = 0 or Z = 1	Lower or same, unsigned $\leq$
<b>GE</b>	N = V	Greater than or equal, signed $\geq$
<b>LT</b>	N $\neq$ V	Less than, signed <
<b>GT</b>	Z = 0 and N = V	Greater than, signed >
<b>LE</b>	Z = 1 or N $\neq$ V	Less than or equal, signed $\leq$
<b>AL</b>	Can have any value	Always. This is the default when no suffix specified

**Table 3.2. Condition code suffixes used to optionally execution instruction.**



# ADR

Load PC-relative address

## Syntax

**ADR{cond} Rd, label**

where **{cond}** is an optional condition code. See Table 3.2. **Rd** Is the destination register. **label** is a PC-relative expression.

## Operation

ADR determines the address by adding an immediate value to the PC, and writes the address of the **label** to the destination register. ADR produces position-independent code, because the address is PC-relative. If you use ADR to generate a target address for a BX or BLX instruction, you must ensure that bit[0] of the address you generate is set to 1 for correct execution. Values of **label** must be within the range of -4095 to +4095 from the address in the PC. You might have to use the .W suffix to get the maximum offset range or to generate addresses that are not word-aligned. See the instruction manual for more information about .W width selection.

## Restrictions

- **Rd** must not be **SP** and must not be **PC**.

## Condition Flags

This instruction does not change the flags.

## Examples

Hello PUSH {R4,LR}

ADR R0, Name ;R0 is the address of the string at Name

BL OutString ;Print welcome

POP {R4,PC}

Name DCB "Hello world",0

AREA DATA

FuncPt SPACE 4

AREA CODE,READONLY,ALIGN=2

Set ADR R0, Hello ;R0 points to function Hello

ORR R0,R0,#1 ;set Thumb bit (this step IS necessary)

LDR R1,=FuncPt

STR R0,[R1] ;FuncPt points to Hello

BX LR

# ADD

## 32-bit Addition

### Syntax

**ADD{S}{cond} {Rd,} Rn, Op2**

**ADD{cond} {Rd,} Rn, #imm12**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. **imm12** is any value in the range 0–4095. The syntax of **Op2** is

**ADD Rd, Rn, Rm ; op2 = Rm**

**ADD Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**ADD Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**ADD Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**ADD Rd, Rn, #constant ; op2 = constant**, where **X** and **Y** are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits
- in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

### Operation

The ADD instruction adds the value of **Op2** or **imm12** to the value in **Rn** and stores the sum in **Rd**.

$$Rd = Rn + Op2$$

$$Rd = Rn + imm12$$

### Restrictions (for additional restrictions about PC see data sheet)

- **Op2** must not be **SP** and must not be **PC**.
- **Rd** can be **SP** only with the additional restrictions:
  - **Rn** must also be **SP**.
  - any shift in **Op2** must be limited to a maximum of 3 bits using **LSL**.

### Condition Flags

If **S** is specified, these instructions update the N, Z, C and V flags according to the result.  $R = X + M$ , where **X** is initial register value, **M** is the flexible second operand or the **#imm12** constant, and **R** is the final register value.

N: result is negative

$$N = R_{31}$$

$$Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$$

Z: result is zero

$$V = X_{31} \& M_{31} \& \overline{R_{31}} \quad | \quad \overline{X_{31}} \& \overline{M_{31}} \& R_{31}$$

V: signed overflow

$$C = X_{31} \& M_{31} \quad | \quad M_{31} \& \overline{R_{31}} \quad | \quad \overline{R_{31}} \& X_{31}$$

C: unsigned overflow

## Examples

ADD R2, R1, R3 ;R2=R1+R3

ADDS R4, R4, #100 ;R4=R4+100, set flags

ADDHI R11, R0, R3 ;R11=R0+R3, part of IT, execute if C=0 and Z=0

# AND

## 32-bit Logical AND

### Syntax

**AND{S}{cond} {Rd,} Rn, Op2**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. The syntax of **Op2** is

**AND Rd, Rn, Rm ; op2 = Rm**

**AND Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**AND Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**AND Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**AND Rd, Rn, #constant ; op2 = constant**, where X and Y are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

Rn	Op	AND
0	0	0
0	1	0
1	0	0
1	1	1

### Operation

The AND instruction performs a 32-bit bitwise AND operation on the values in **Rn** and **Op2** and places the results into **Rd**. The AND instruction is useful for selecting bits. You specify which bits to select in the **Op2**.

**Rd = Rn & Op2**

### Restrictions

- Do not use **SP** and do not use **PC**.

### Condition Flags

If **S** is specified, update the N and Z flags according to the result, **Rd**. It can also update the C flag during the calculation of **Op2**. It does not affect the V flag.

N: result is negative       $N = R_{31}$

Z: result is zero       $Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$

### Examples

**AND R9, R2, #0xFF00 ;R9=R2&0x0000FF00**

**AND R0, R0, R5 ;R0=R0&R5**

**AND R0, R0, R5, LSR #3 ;R0=R0&(R5<<3)**  
**ANDS R9, R8, #1 ;R9=R8&0x00000001, sets flags**  
**ANDHS R11, R0, R3 ;R11=R0&R3, part of IT, execute if C=0**

ASR

## 32-bit Arithmetic Shift Right

## Syntax

ASR{S}{cond} Rd, Rm, Rs

ASR{S}{cond} Rd, Rm, #n

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. **Rd** cannot be omitted. **Rm** is the register holding the value to be shifted. **Rs** is the register holding the shift length to apply to the value in **Rm**. Only the least significant byte of **Rs** is used and can be in the range 0 to 255. **n** is the shift length (1 to 32).

## Operation

ASR moves the bits in the register **Rm** to the right by the number of places specified by constant **n** or register **Rs**. Values are signed integers, so the sign bit in bit 31 is preserved. The result is written to **Rd**, and the value in register **Rm** remains unchanged.

$$R_d = R_m \gg R_s \quad (\text{signed})$$

$$Rd = Rm \gg n \quad (\text{signed})$$



## Restrictions

- Do not use SP and do not use PC.

## Condition Flags

If **S** is specified, these instructions update the N and Z flags according to the result in **Rd**. The C flag is updated to the last bit shifted out, except when the shift length is 0.

N: result is negative

$$N = R_{31}$$

Z: result is zero

$$Z = R_{31} \& R_{30} \& L \& R_0$$

## Examples

**ASR R7, R8, #9 ;R7 = R8>>9, signed, (similar to R7 = R8/512)**

**ASR R0, R1, R2 ;R0 = R1>>R2, signed (similar to R0 = R1/2<sup>R2</sup>)**

ASRS R1, R2, #3 ;R1 = R2>>3, signed, with flag update

ASREO R4, R5, #6 ;part of IT, if Z=1 then R4=R5>>6, signed

# B

## Branch instructions

### Syntax

**B label ; branch to label Always**

**BEQ label ; branch if Z == 1 Equal**

**BNE label ; branch if Z == 0 Not equal**

**BCS label ; branch if C == 1 Higher or same, unsigned  $\geq$**

**BHS label ; branch if C == 1 Higher or same, unsigned  $\geq$**

**BCC label ; branch if C == 0 Lower, unsigned <**

**BLO label ; branch if C == 0 Lower, unsigned <**

**BMI label ; branch if N == 1 Negative**

**BPL label ; branch if N == 0 Positive or zero**

**BVS label ; branch if V == 1 Overflow**

**BVC label ; branch if V == 0 No overflow**

**BHI label ; branch if C==1 and Z==0 Higher, unsigned >**

**BLS label ; branch if C==0 or Z==1 Lower or same, unsigned  $\leq$**

**BGE label ; branch if N == V Greater than or equal, signed  $\geq$**

**BLT label ; branch if N != V Less than, signed <**

**BGT label ; branch if Z==0 and N==V Greater than, signed >**

**BLE label ; branch if Z==1 or N!=V Less than or equal, signed  $\leq$**

where {cond} is an optional condition code. See Table 3.2. **label** is a PC-relative expression.

### Operation

These instructions cause a branch to **label**. These are the only conditional instructions that can be either inside or outside an IT block. All other conditional instructions must be inside an IT block.

**B label** -16 MB to +16 MB

**Bcond label** -1 MB to +1 MB (outside IT block)

**Bcond label** -16 MB to +16 MB (inside IT block)

You might have to use the .W suffix to get the maximum branch range. See the instruction manual for more information about .W width selection.

### Restrictions

- When inside an IT block, it must be the last instruction of the IT block.

### Condition Flags

These instructions do not change the flags.

### Examples

**B loopA ;Branch to loopA**

**BLE ng ;Conditionally branch to label ng**

**B.W** target ;Branch to target within 16MB range

**BEQ** target ;Conditionally branch to target

**BEQ.W** target ;Conditionally branch to target within 1MB

# BIC

## 32-bit Logical Bit Clear

### Syntax

**BIC{S}{cond} {Rd,} Rn, Op2**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. The syntax of **Op2** is

**BIC Rd, Rn, Rm ; op2 = Rm**

**BIC Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**BIC Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**BIC Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**BIC Rd, Rn, #constant ; op2 = constant**, where X and Y are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

Rn	O p	BIC
0	0	0
0	1	0
1	0	1
1	1	0

### Operation

The BIC instruction performs a 32-bit bitwise AND operation on the bits in **Rn** with the complements of the corresponding bits from **Op2** and places the results into **Rd**. This instruction is useful for clearing bits. You specify which bits to clear in the **Op2**.

**Rd = Rn & ~Op2**

### Restrictions

- Do not use **SP** and do not use **PC**.

### Condition Flags

If **S** is specified, update the N and Z flags according to the result, **Rd**. It can also update the C flag during the calculation of **Op2**. It does not affect the V flag.

N: result is negative

$N = R_{31}$

Z: result is zero

$Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$

### Examples

**BIC R9, R2, #0xFF00 ;R9 = R2&0xFFFF00FF, clear bits 15-8**

**BICS R0, R0, #2 ;R0 = R0&0xFFFFFFF0, clear bit1, set flags**

**BIC R1, R2, R3, LSL #2 ;R1 = R2&( $\sim$ (R3<<2))  
BICEQ R2, R0, R5 ;R2 = R0&( $\sim$ R5), part of IT execute if Z=1**

# BL

Branch link (call subroutine)

## Syntax

**BL{cond} label ; branch to subroutine at label**

where {cond} is an optional condition code. See Table 3.2. **label** is a PC-relative expression.

## Operation

BL is the call to subroutine instruction. The address of the subroutine is specified by the **label**. The BL instruction also saves the return address (the address of the next instruction) in the Link Register (LR), Register R14. The range of the BL instruction is -16 MB to +16 MB from the current instruction. You might have to use the .W suffix to get the maximum branch range. See the instruction manual for more information about .W width selection.

## Restrictions

- Conditional execution can only occur inside an IT block.
- When inside an IT block, it must be the last instruction of the IT block.

## Condition Flags

This instruction does not change the flags.

## Examples

**BL Func ;Call to Func, return address in LR**

MOV R0,#1234

MOV R1,#100

**BL.W Divide ;Call to Divide, return address in LR (-16 to +16MB)**

;example subroutine

Func PUSH {R4-R8,LR} ;save registers

;body of subroutine

POP {R4-R8,PC} ;restore registers and return

; find the unsigned quotient and remainder

; Inputs: dividend in R0

; divisor in R1

; Outputs: quotient in R2

; remainder in R3

;dividend = divisor\*quotient + remainder

Divide

UDIV R2,R0,R1 ;R2=R0/R1, R2 is quotient

MUL R3,R2,R1 ;R3=(R0/R1)\*R1

SUB R3,R0,R3 ;R3=R0%R1, R3 is remainder of R0/R1  
BX LR ;return

# BLX

Branch link indirect (call subroutine)

## Syntax

**BLX{cond} Rm ; branch to subroutine indirect specified by Rm**

where {cond} is an optional condition code. See Table 3.2. Rm is a register that indicates the address to which to branch. Bit[0] of the value in Rm must be 1, but the address to branch to is created by changing bit[0] to 0.

## Operation

BLX is an indirect call to subroutine instruction. The address of the subroutine is specified by the register **Rm**. The BLX instruction also saves the return address (the address of the next instruction) in the Link Register (LR), register R14.

## Restrictions

- Conditional execution can only occur inside an IT block.
- It must be unconditional outside the IT block.
- Do not use **PC** in the BLX instruction.
- Bit[0] of **Rm** must be 1, the target address is created by changing bit[0] to 0.
- When inside an IT block, it must be the last instruction of the IT block.

## Condition Flags

This instruction does not change the flags.

## Examples

FList DCD Fun0,Fun1,Fun2,Fun3 ;pointers to four functions

;Assume R2 contains a value from 0 to 3

LDR R1,=FList ;R1 points to list of functions

LDR R0,[R1,R2,LSL#2] ;R0 points to subroutine to execute

ORR R0,R0,#1 ;set thumb bit (this step may not be necessary)

**BLX R0** ;call subroutine, return address in LR

;end of example

Fun0 ;body of function 0

  BX LR

Fun1 ;body of function 1

  BX LR

Fun2 ;body of function 2

  BX LR

Fun3 ;body of function 3

  BX LR



# BX

## Branch indirect

### Syntax

**BX{cond} Rm ; branch indirect to location specified by Rm**

where {cond} is an optional condition code. See Table 3.2. Rm is a register that indicates the address to which to branch.

### Operation

This is a branch indirect instruction, with the branch address indicated in Rm. This instruction causes a UsageFault exception if bit[0] of Rm is 0. BX LR is often used as a return from subroutine.

### Restrictions

- Conditional execution can only occur inside an IT block.
- It must be unconditional outside the IT block.
- Bit[0] of Rm must be 1, the target address is created by changing bit[0] to 0.
- When inside an IT block, it must be the last instruction of the IT block.

### Condition Flags

This instruction does not change the flags.

### Examples

; Inputs: dividend in R0

; divisor in R1

; Outputs: quotient in R2

; remainder in R3

;dividend = divisor\*quotient + remainder

Divide

UDIV R2,R0,R1 ;R2=R0/R1, R2 is quotient

MUL R3,R2,R1 ;R3=(R0/R1)\*R1

SUB R3,R0,R3 ;R3=R0%R1, R3 is remainder of R0/R1

**BX LR ;return**

List DCD Place0,Place1,Place2,Place3 ;pointers to four places

;Assume R2 contains a value from 0 to 3

LDR R1,=List ;R1 points to list of assembly labels

LDR R0,[R1,R2,LSL#2] ;R0 points to code to execute

ORR R0,R0,#1 ;make sure thumb bit set (not needed on Keil)

**BX R0 ;jump to place specified by R0**

;end of example

Place0 ;code 0  
Place1 ;code 1  
Place2 ;code 2  
Place3 ;code 3

# CMN

## 32-bit Compare Negative

### Syntax

**CMN{cond} Rn, Op2**

where {cond} is an optional condition code. See Table 3.2. There is no destination register. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. The syntax of **Op2** is

**CMN Rn, Rm ; op2 = Rm**

**CMN Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**CMN Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**CMN Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**CMN Rn, #constant ; op2 = constant**, where X and Y are hexadecimal digits:

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

### Operation

This instruction compares the value in a register with **Op2**. They update the condition flags on the result, but do not write the result to a register. The CMN instruction adds the value of **Op2** to the value in Rn. The condition codes are set as if a subtraction occurred. This instruction can be followed by a conditional branch.

**Rn - (-Op2)**

**(result is calculated but not stored)**

This instruction is useful for extending the range of the immediate field of a CMP instruction. E.g.,

**CMP R0,#-2 ;not a valid instruction**

can be executed as

**CMN R0,#2 ;a valid instruction comparing R0 to -2**

### Restrictions

- Do not use PC.
- Operand2 must not be SP.

### Condition Flags

These instructions update the N, Z, C and V flags according to the result, **Rn - (-Op2)**. R=X-(-M), where X is initial register value, M is the flexible second operand, and R is the final subtracted value.

N: result is negative

$$N = R_{31}$$

$$Z = \overline{R_{31}} \& \overline{R_{30}} \& L \& \overline{R_0}$$

Z: result is zero

$$V = X_{31} \& M_{31} \& (\sim R_{31}) \mid (\sim X_{31}) \& (\sim M_{31}) \& R_{31}$$

V: signed overflow

$$C = \sim((\sim X_{31}) \& (\sim M_{31})) \mid (\sim M_{31}) \& R_{31} \mid R_{31} \& (\sim X_{31})$$

C: unsigned overflow

## Examples

**CMN R2, #25 ; compare R2 to -25**

BEQ gothere ;branch to gothere if R2 equals -25

# CMP

## 32-bit Compare

### Syntax

**CMP{cond} Rn, Op2**

where {cond} is an optional condition code. See Table 3.2. There is no destination register. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. The syntax of **Op2** is

**CMP Rn, Rm ; op2 = Rm**

**CMP Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**CMP Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**CMP Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**CMP Rn, #constant ; op2 = constant**, where X and Y are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

### Operation

These instructions compare the value in a register with **Op2**. They update the condition flags on the result, but do not write the result to a register. The CMP instruction subtracts the value of **Op2** from the value in Rn. This is the same as a SUBS instruction, except that the result is discarded. This instruction can be followed by a conditional branch.

**Rn – Op2**

(result is calculated but not stored)

### Restrictions

- Do not use PC.
- Operand2 must not be SP.

### Condition Flags

These instructions update the N, Z, C and V flags according to the result, **Rn – (Op2)**. R=X-M, where X is initial register value, M is the flexible second operand, and R is the final subtracted value.

N: result is negative

$$N = R_{31}$$

Z: result is zero

$$Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$$

V: signed overflow

$$V = X_{31} \& \overline{M_{31}} \& \overline{R_{31}} \quad | \quad \overline{X_{31}} \& M_{31} \& R_{31}$$

C: unsigned overflow

$$C = \overline{X_{31}} \& M_{31} \quad | \quad M_{31} \& R_{31} \quad | \quad R_{31} \& \overline{X_{31}}$$

### Examples

**CMP R2, #6400**

BGT gothere ;branch to gothere if R2>6400 (signed)

## CMP R2, R3, LSR #1

BLO gothere ;branch to gothere if R2<(R3>>1) (unsigned)

# CPS

## Change Processor State

### Syntax

**CPSIE I ;Clears the Priority Mask Register (PRIMASK)**

**CPSIE F ;Clears the Fault Mask Register (FAULTMASK)**

**CPSID I ;Sets the Priority Mask Register (PRIMASK)**

**CPSID F ;Sets the Fault Mask Register (FAULTMASK)**

### Operation

CPS changes the PRIMASK and FAULTMASK special register values. See the Data Sheet for more information about these registers.

### Restrictions

- Use CPS only from privileged software; it has no effect if used in unprivileged software.
- CPS cannot be conditional and so must not be used inside an IT block.

### Condition Flags

This instruction does not change the flags.

### Examples

**CPSID I ; Set I, disable interrupts and configurable fault handlers**

**CPSIE I ; Clear I, enable interrupts and configurable fault handlers**

# EOR

## 32-bit Logical Exclusive OR

### Syntax

**EOR{S}{cond} {Rd,} Rn, Op2**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. The syntax of **Op2** is

**EOR Rd, Rn, Rm ; op2 = Rm**

**EOR Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**EOR Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**EOR Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**EOR Rd, Rn, #constant ; op2 = constant**, where X and Y are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

Rn	Op	EO R
0	0	0
0	1	1
1	0	1
1	1	0

### Operation

The EOR instruction performs a 32-bit bitwise Exclusive OR operation on the values in **Rn** and **Op2** and places the results into **Rd**. The EOR instruction is useful for toggling bits. You specify which bits to toggle in the **Op2**.

$$Rd = Rn \wedge Op2$$

### Restrictions

- Do not use **SP** and do not use **PC**.

### Condition Flags

If **S** is specified, update the N and Z flags according to the result, **Rd**. It can also update the C flag during the calculation of **Op2**. It does not affect the V flag.

N: result is negative

$$N = R_{31}$$

$$Z = \overline{R_{31}} \wedge \overline{R_{30}} \wedge \dots \wedge \overline{R_0}$$

Z: result is zero

### Example

#### TogglePG2

LDR R1, =GPIO\_PORTG\_DATA\_R ;R1 = &GPIO\_PORTG\_DATA\_R

LDR R2, [R1] ;R2 = [R1] (read all data on PG)  
**EOR R2, R2, #0x04** ;**R2 = R2 ^ ~0x04 (toggle bit 2)**  
STR R2, [R1] ;[R1] = R2  
BX LR ;return

# LDR

Load from memory into a register

## Syntax

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset EA=Rn+offset
LDR{type}{cond} Rt, [Rn, #offset]!   ; pre-indexed, EA=Rn+offset
LDR{type}{cond} Rt, [Rn], #offset    ; post-indexed, EA=Rn
LDR{type}{cond} Rt, [Rn, Rm {, LSL #n}] ; register offset, EA=Rn+(Rm<<n)
LDR{type}{cond} Rt, label          ; PC relative, EA=PC+relative
```

where **type** is one of

- **B** Unsigned byte, zero extended to 32 bits.
- **SB** Signed byte, sign extended to 32 bits.
- **H** Unsigned halfword, zero extended to 32 bits.
- **SH** Signed halfword, sign extended to 32 bits.
- Omit, for word.

{**cond**} is an optional condition code. See Table 3.2. **Rt** is the register to load. **Rn** is the register on which the memory address is based. **offset** is an offset from **Rn**. If **offset** is omitted, the effective address is the contents of **Rn**. **Rm** is a register containing a value to be used as the offset. **n** is a number 0 to 3.

## Operation

LDR instructions copy values from memory into registers. **Immediate offset** addressing adds the offset value (-255 to 4095) to the value of **Rn** to get the effective address. The register **Rn** is unaltered. **Pre-indexed addressing** first adds the offset (-255 to 255) to **Rn** to create the effective address. This mode changes **Rn**. **Post-indexed addressing** uses the original value of **Rn** as the effective memory address. After memory is accessed, the offset (-255 to 255) is added to **Rn**. This mode also changes **Rn**. **Register offset addressing**, first shifts left **Rm** by 0 to 3 bits and adds it to the value of **Rn** to get the effective address. This mode does not alter **Rn** or **Rm**. When using **PC relative addressing**, the **label** must be within -4095 to +4095.

## Restrictions

- **Rt** can be **SP** or **PC** for word loads only.
- **Rm** must not be **SP** and must not be **PC**.
- When **Rt** is **PC** in a word load instruction, Bit[0] must be 1. If conditional in IT block, it must be last
- **Rn** must not be **PC** for register offset addressing.

## Condition Flags

These instructions do not change the flags.

## Examples

LDR R8, [R10] ;Load 32-bit from R10 address to R8

LDRB R0, [R1] ;Load 8-bit unsigned from R1 address to R0

LDRSB R1, [R2,#5] ;Load 8-bit signed from (R2+5) address to R1

LDRH R2, [R5,#2]! ;R5=R5+2, load 16-bit unsigned from R5 address to R2

LDR R3, [R7],#4 ;Load 32-bit from R7 address to R3, R7=R7+4

LDR R0, Pi ;R0=314159

Pi DCD 314159

LSL

# 32-bit Logical Shift Left

## Syntax

LSL{S}{cond} Rd, Rm, Rs

**LSL{S}{cond} Rd, Rm, #n**

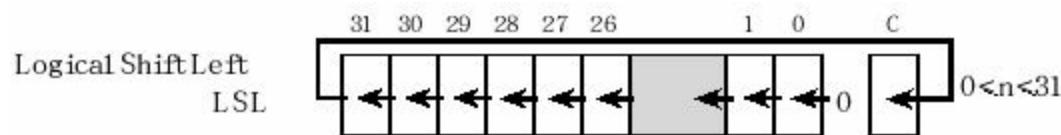
where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. **Rd** can not be omitted. **Rm** is the register holding the value to be shifted. **Rs** is the register holding the shift length to apply to the value in **Rm**. Only the least significant byte of **Rs** is used and can be in the range 0 to 255. **n** is the shift length (0 to 31).

# Operation

LSL moves the bits in the register **Rm** to the left by the number of places specified by constant **n** or register **Rs**. This instruction can be used for signed and unsigned integers. Shift left is similar to multiplication by a power of 2. The result is written to **Rd**, and the value in register **Rm** remains unchanged.

(signed or unsigned)

(signed or unsigned)



## Restrictions

- Do not use SP and do not use PC.

## Condition Flags

If **S** is specified, these instructions update the N and Z flags according to the result in **Rd**. The C flag is updated to the last bit shifted out, except when the shift length is 0.

N: result is negative

$$N = R_{31}$$

Z: result is zero

$$Z = \overline{R_{31}} \text{ & } \overline{R_{30}} \text{ & L } \text{ & } \overline{R_0}$$

## Examples

**LSL R7, R8, #9 ;R7 = R8<<9 (similar to R7=R8\*512)**

**LSLS R1, R2, #3 ;R1 = R2<<3 (similar to R1=R2\*8) with flag update**

**LSL R4, R5, R6 :R4 = R5<<R6** (similar to  $R4=R5*2^{R6}$ )

# LSR

## 32-bit Logical Shift Right

### Syntax

**LSR{S}{cond} Rd, Rm, Rs**

**LSR{S}{cond} Rd, Rm, #n**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. **Rd** cannot be omitted. **Rm** is the register holding the value to be shifted. **Rs** is the register holding the shift length to apply to the value in **Rm**. Only the least significant byte of **Rs** is used and can be in the range 0 to 255. **n** is the shift length (1 to 32).

### Operation

LSR moves the bits in the register **Rm** to the right by the number of places specified by constant **n** or register **Rs**. Values are unsigned integers, so zeros are shifted into bit 31. Shift right is similar to unsigned division by a power of 2. The result is written to **Rd**, and the value in register **Rm** remains unchanged.

$$\begin{array}{ll} \mathbf{Rd} = \mathbf{Rm} \gg \mathbf{Rs} & \text{(unsigned)} \\ \mathbf{Rd} = \mathbf{Rm} \gg \mathbf{n} & \text{(unsigned)} \end{array}$$



### Restrictions

- Do not use SP and do not use PC.

### Condition Flags

If **S** is specified, these instructions update the N and Z flags according to the result in **Rd**. The C flag is updated to the last bit shifted out, except when the shift length is 0.

N: result is negative       $N = R_{31}$

Z: result is zero       $Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$

### Examples

**LSR R7, R8, #9 ;R7 = R8>>9 (similar to R7=R8/512)**

**LSRS R1, R2, #3 ;R1 = R2>>3 (similar to R1=R2/8) with flag update**

**LSR R4, R5, R6 ;R4 = R5>>R6 (similar to R4=R5/2<sup>R6</sup>)**

# MLA

## 32-bit Multiplication with Accumulation

### Syntax

**MLA{cond} Rd, Rn, Rm, Ra ; Multiply with accumulate**

where {cond} is an optional condition code. See Table 3.2. **Rd** is the destination register. **Rd** cannot be omitted. **Rn**, **Rm** are registers holding the values to be multiplied. **Ra** is a register holding the value to be added.

### Operation

The MLA instruction multiplies the values from **Rn** and **Rm**, adds the value from **Ra**, and places the least-significant 32 bits of the result in **Rd**. The result does not depend on whether the operands are signed or unsigned. This instruction is useful for implementing digital filters and other digital signal processing.

$$\mathbf{Rd} = \mathbf{Ra} + (\mathbf{Rn} * \mathbf{Rm})$$

### Restrictions

- Do not use **SP** and do not use **PC**.

### Condition Flags

This instruction does not change the flags.

### Examples

**MLA R9, R2, R1, R5 ;R9 = R5 + (R2 \* R1)**

**MLA R0, R2, R3, R0 ;R0 = R0 + (R2 \* R3)**

# MLS

## 32-bit Multiplication with Subtraction

### Syntax

**MLS{cond} Rd, Rn, Rm, Ra ; Multiply with subtract**

where **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. **Rd** cannot be omitted. **Rn**, **Rm** are registers holding the values to be multiplied. **Ra** is a register holding the value to be subtracted from.

### Operation

The MLS instruction multiplies the values from **Rn** and **Rm**, subtracts the product from **Ra**, and places the least-significant 32 bits of the result in **Rd**. The result does not depend on whether the operands are signed or unsigned. This instruction is useful for implementing digital filters and other digital signal processing.

$$Rd = Ra - (Rn * Rm)$$

### Restrictions

- Do not use **SP** and do not use **PC**.

### Condition Flags

This instruction does not change the flags.

### Examples

**MLS R9, R2, R1, R5 ;R9 = R5 - (R2 \* R1)**

**MLS R0, R2, R3, R0 ;R0 = R0 - (R2 \* R3)**

# MOV

## 32-bit Move

### Syntax

**MOV{S}{cond} Rd, Op2**

**MOV{cond} Rd, #imm16**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. **Op2** is a flexible second operand. **imm16** is any value in the range 0–65535. The syntax of **Op2** is

**MOV Rd, Rm ; op2 = Rm**

**MOV Rd, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**MOV Rd, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**MOV Rd, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**MOV Rd, #constant ; op2 = constant**, where X and Y are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

### Operation

The MOV instruction copies the value of **Op2** into **Rd**. This instruction is useful for moving values from one register to another, and for initializing registers to a constant value.

**Rd = Op2**

### Restrictions

You can use SP and PC, with the following restrictions:

- The second operand must be a register without shift
- You must not specify the S suffix.

Though it is possible to use MOV as a branch instruction, it is strongly recommended the use BX or BLX.

### Condition Flags

If **S** is specified, update the N and Z flags according to the result, Rd. It can also update the C flag during the calculation of **Op2**. It does not affect the V flag.

N: result is negative

$N = R_{31}$

Z: result is zero

$Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$

### Examples

**MOVS R11, #10 ; R11=10, N and Z flags get updated**

**MOV R1, #0xFA05 ; R1 = 0xFA05**

**MOVS R10, R12 ; R10 = R12, N and Z flags get updated**  
**MOVS R0, R0 ; N and Z flags set according to R0**  
**MOV R11, SP ; Place a copy of stack pointer in R11**  
**MOVEQ R0, #0 ; Part of IT, if Z=1, set R0=0**

# MUL

## 32-bit Multiplication

### Syntax

**MUL{S}{cond} {Rd,} Rn, Rm ; Multiply**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn, Rm** are registers holding the values to be multiplied.

### Operation

The MUL instruction multiplies the values from **Rn** and **Rm**, and places the least-significant 32 bits of the result in **Rd**. The result does not depend on whether the operands are signed or unsigned. This instruction is useful for implementing digital filters and other digital signal processing.

$$Rd = Rn * Rm$$

### Restrictions

In these instructions, do not use **SP** and do not use **PC**. If you use the **S** suffix:

- **Rd, Rn, and Rm** must all be in the range R0 to R7.
- **Rd** must be the same as **Rm**.
- You must not use the **cond** suffix.

### Condition Flags

If **S** is specified, update the N and Z flags according to the result, **Rd**. It does not affect the C and V flags.

N: result is negative

$$N = R_{31}$$

Z: result is zero

$$Z = \overline{R_{31}} \& \overline{R_{30}} \& L \& \overline{R_0}$$

### Examples

**MUL R10, R2, R5 ;R10 = R2 \* R5**

**MUL R0, R1 ;R0 = R0 \* R1**

**MULS R0, R1 ;R0 = R0 \* R1, sets N and Z flags**

**MULCC R2, R3, R2 ;part of IT, if C=0, R2 = R3 x R2**

# MVN

## 32-bit Move NOT

### Syntax

**MVN{S}{cond} Rd, Op2**

where {S} is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation. {cond} is an optional condition code. See Table 3.2. Rd is the destination register. Op2 is a flexible second operand. The syntax of Op2 is

**MVN Rd, Rm ; op2 = Rm**

**MVN Rd, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**MVN Rd, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**MVN Rd, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**MVN Rd, #constant ; op2 = constant**, where X and Y are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

### Operation

The MVN instruction takes the value of Op2, performs a bitwise logical NOT operation on the value, and places the result into Rd.

**Rd = ~Op2**

This instruction is useful for extending the range of the immediate field of a MOV instruction. For example

**MOV R0,#-2 ;not a valid instruction (-2 = 0xFFFF.FFFFE)**

can be executed as

**MVN R0,#1 ;R0 = -2**

### Restrictions

- You cannot use SP and PC

### Condition Flags

If S is specified, update the N and Z flags according to the result, Rd. It can also update the C flag during the calculation of Op2. It does not affect the V flag.

N: result is negative

$N = R_{31}$

Z: result is zero

$Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$

### Examples

**MVN R0, #10 ;R0 = -11**

**MVN R1, R2 ;R1 = ~R2**

**MVN R1, R2, LSL #2 ;R1 = ~(R2<<2)**

# ORN

## 32-bit Logical OR NOT

### Syntax

**ORN{S}{cond} {Rd,} Rn, Op2**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. The syntax of **Op2** is

**ORN Rd, Rn, Rm ; op2 = Rm**

**ORN Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**ORN Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**ORN Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**ORN Rd, Rn, #constant ; op2 = constant**, where X and Y are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

Rn	Op	ORN
0	0	1
0	1	0
1	0	1
1	1	1

### Operation

The ORN instruction performs a 32-bit bitwise OR operation on the bits in **Rn** with the complements of the corresponding bits in **Op2** and places the results into **Rd**. The ORN extends the useful range of the OR.

$$Rd = Rn \mid \sim Op2$$

### Restrictions

- Do not use **SP** and do not use **PC**.

### Condition Flags

If **S** is specified, update the N and Z flags according to the result, **Rd**. It can also update the C flag during the calculation of **Op2**. It does not affect the V flag.

N: result is negative

$$N = R_{31}$$

Z: result is zero

$$Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$$

### Examples

**ORN R9, R2, #0x00FF ; R9 = R2 | 0xFFFFF00**

**ORN R7, R11, R12, LSR #4 ; R7 = R11 | (~R12>>2)), R12 is unsigned**



# ORR

## 32-bit Logical OR

### Syntax

**ORR{S}{cond} {Rd,} Rn, Op2**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. The syntax of **Op2** is

**ORR Rd, Rn, Rm ; op2 = Rm**

**ORR Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**ORR Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**ORR Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**ORR Rd, Rn, #constant ; op2 = constant**, where X and Y are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

Rn	Op	ORR
0	0	0
0	1	1
1	0	1
1	1	1

### Operation

The OR instruction performs a 32-bit bitwise OR operation on the values in **Rn** and **Op2** and places the results into **Rd**. The OR instruction is useful for setting bits. You specify which bits to set in the **Op2**.

$$Rd = Rn \mid Op2$$

### Restrictions

- Do not use **SP** and do not use **PC**.

### Condition Flags

If **S** is specified, update the N and Z flags according to the result, **Rd**. It can also update the C flag during the calculation of **Op2**. It does not affect the V flag.

N: result is negative       $N = R_{31}$

Z: result is zero       $Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$

### Examples

**ORR R9, R2, #0xFF00 ;R9 = R2 | 0xFF00 (set bits 15-8)**

**ORREQ R2, R0, R5 ;part of IT, if Z=1, R2=R0|R5**

**ORRS R7, R11, #0x18181818 ;R7=R11|0x18181818**

# POP

Pop registers off a full-descending stack

## Syntax

**POP{cond} reglist**

where **{cond}** is an optional condition code. See Table 3.2. **reglist** is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range. POP is a synonym for LDM with the memory addresses for the access based on **SP**, and with the final address for the access written back to the **SP**. According to AAPCS we need to push and pop an even number of registers to maintain an 8-byte alignment on the stack.

## Operation

POP loads registers from the stack in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address.

## Restrictions

- **reglist** must not contain SP.
- **reglist** must not contain PC if it contains LR.
- When PC is in **reglist** in a POP instruction:
  - Bit[0] of the value loaded to the PC must be 1 for correct execution,
  - A branch occurs to this halfword-aligned address.
- If the instruction is conditional, it must be the last instruction in the IT block.

## Condition Flags

This instruction does not change the flags.

## Examples

**POP {R5} ;pop 32 bits from stack and place it in R5**

**POP {R0,R4-R7} ;pop 5 words from stack and place into R0,R4,R5,R6,R7**

**POP {R2,LR} ;pop 2 words from stack and place into R2, R14**

**POP {R0,R10,PC} ;pop 3 words from stack and place into R0,R10,PC**

;example subroutine

Func PUSH {R4-R8,LR} ;save registers

;body of subroutine

**POP {R4-R8,PC} ;restore registers and return**

# PUSH

Push registers onto a full-descending stack

## Syntax

**PUSH{cond} reglist**

where **{cond}** is an optional condition code. See Table 3.2. **reglist** is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range. PUSH is a synonym for STMDB with the memory addresses for the access based on **SP**, and with the final address for the access written back to the **SP**. According to AAPCS we need to push and pop an even number of registers to maintain an 8-byte alignment on the stack.

## Operation

PUSH stores registers on the stack in order of decreasing register numbers, with the highest numbered register using the highest memory address and the lowest numbered register using the lowest memory address.

## Restrictions

- **reglist** must not contain SP or PC
- If the instruction is conditional, it must be the last instruction in the IT block.

## Condition Flags

This instruction does not change the flags.

## Examples

**PUSH {R0}** ;push the 32-bit contents of R0 on stack

**PUSH {R0,R4-R7}** ;push R7,R6,R5,R4,R0 on stack (R0 on top)

**PUSH {R2,LR}** ;push LR,R2 on stack (R2 on top)

;example subroutine, with local variable

**Func PUSH {R4-R8,LR} ;save registers**

sum EQU 0 ;32-bit local variable, stored on the stack

MOV R0,#0

**PUSH {R0,R1} ;allocate and initialize 2 local variables**

;body of subroutine

LDR R1,[SP,#sum] ;R1=sum

ADD R1,R0 ;R1=R0+sum

STR R1,[SP,#sum] ;sum=R0+sum

;end of subroutine

ADD SP,#8 ;deallocate sum

POP {R4-R8,PC} ;restore registers and return

# RSB

## 32-bit Reverse Subtraction

### Syntax

**RSB{S}{cond} {Rd,} Rn, Op2**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. The syntax of **Op2** is

**RSB Rd, Rn, Rm ; op2 = Rm**

**RSB Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**RSB Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**RSB Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**RSB Rd, Rn, #constant ; op2 = constant**, where **X** and **Y** are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

### Operation

The RSB instruction subtracts the value in **Rn** from the value of **Op2** and stores the sum in **Rd**.

**Rd = Op2 - Rn**

This is useful because of the wide range of options for **Op2**.

### Restrictions (for additional restrictions about PC see data sheet)

- **Op2** must not be **SP** and must not be **PC**.
- **Rn** cannot be **SP**.

### Condition Flags

If **S** is specified, these instructions update the N, Z, C and V flags according to the result.  $R=M-X$ , where X is initial register value, M is the flexible second operand, and R is the final register value.

N: result is negative

$$N = R_{31}$$

$$Z = \overline{R_{31}} \& \overline{R_{30}} \& \dots \& \overline{R_0}$$

Z: result is zero

$$V = M_{31} \& (\sim X_{31}) \& (\sim R_{31}) \mid (\sim M_{31}) \& X_{31} \& R_{31}$$

V: signed overflow

$$C = \sim( (\sim M_{31} \& X_{31}) \mid X_{31} \& R_{31} \mid R_{31} \& (\sim M_{31}) )$$

### Examples

**RSB R2, R1, R3 ;R2=R3-R1**

**RSBS R8, R6, #240 ;R8=240-R6, sets the flags**

**RSB R4, R4, #1280 ;R4=1280-R4**



# **SDIV**

## 32-bit Signed Division

### Syntax

**SDIV{cond} {Rd,} Rn, Rm**

where **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the value to be dividend. **Rm** is a register holding the divisor.

### Operation

SDIV performs a signed integer division of the value in **Rn** by the value in **Rm**. If the value in **Rn** is not divisible by the value in **Rm**, the result is rounded towards zero.

**Rd = Rn / Rm**

### Restrictions

- Do not use **SP** and do not use **PC**.

### Condition Flags

This instruction does not change the flags.

### Examples

**SDIV R0, R2, R4 ;Signed divide, R0 = R2/R4.**

**SDIV R8, R1 ;Signed divide, R8 = R8/R1.**

; find the signed quotient and remainder of R0 divided by R1

; Inputs: dividend in R0

; divisor in R1

; Outputs: quotient in R2

; remainder in R3

;dividend = divisor\*quotient + remainder

;remainder has the same sign as dividend

Divide

**SDIV R2,R0,R1 ;R2=R0/R1, R2 is quotient**

**MUL R3,R2,R1 ;R3=(R0/R1)\*R1**

**SUB R3,R0,R3 ;R3=R0%R1, R3 is remainder of R0/R1**

**BX LR ;return**

# STR

Store from register into memory

## Syntax

**STR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset EA=Rn+offset**

**STR{type}{cond} Rt, [Rn, #offset]! ; pre-indexed, EA=Rn+offset**

**STR{type}{cond} Rt, [Rn], #offset ; post-indexed, EA=Rn**

**STR{type}{cond} Rt, [Rn, Rm {, LSL #n}] ; register offset, EA=Rn+(Rm<<n)**

where **type** is one of

- **B** Unsigned byte, save bits 7-0 into memory.
- **H** Unsigned halfword, save bits 15-0 into memory.
- Omit, for word.

{**cond**} is an optional condition code. See Table 3.2. **Rt** is the register to load or store. **Rn** is the register on which the memory address is based. **offset** is an offset from **Rn**. If **offset** is omitted, the effective address is the contents of **Rn**. **Rm** is a register containing a value to be used as the offset. **n** is a number 0 to 3.

## Operation

STR instructions copy 8-bit 16-bit or 32-bit values from registers to memory. **Immediate offset** addressing adds the offset value (-255 to 4095) to the value of **Rn** to get the effective address. The register **Rn** is unaltered. **Pre-indexed addressing** first adds the offset (-255 to 255) to **Rn** to create the effective address. This mode changes **Rn**. **Post-indexed addressing** uses the original value of **Rn** as the effective memory address. After memory is accessed, the offset (-255 to 255) is added to **Rn**. This mode also changes **Rn**. **Register offset addressing**, first shifts left **Rm** by 0 to 3 bits and adds it to the value of **Rn** to get the effective address. This mode does not alter **Rn** or **Rm**.

## Restrictions

- **Rt** can be **SP** for word stores only.
- **Rt** must not be **PC**.
- **Rm** must not be **SP** and must not be **PC**.
- **Rn** must not be **PC**.

## Condition Flags

These instructions do not change the flags.

## Examples

**STR R2, [R9,#4] ;32-bit store value of R2 into address R9+4**

**STRH R3, [R4], #2 ;16-bit store value of R3 into address R4, R4=R4+2**

**STRB R0, [R5, R1] ;8-bit store value of R0 into address R5+R1**

**STRH R0, [R5, R1, LSL #1] ;16-bit store R0 into address R5+2\*R1**

**STR R0, [R1, R2, LSL #2] ;32-bit store R0 into address R1+4\*R2**  
**STRB R0, [R5, #1]! ;R5=R5+1,8-bit store value of R0 into address R5**

# SUB

## 32-bit Subtraction

### Syntax

**SUB{S}{cond} {Rd,} Rn, Op2**

**SUB{cond} {Rd,} Rn, #imm12**

where **{S}** is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation. **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the first operand. **Op2** is a flexible second operand. **imm12** is any value in the range 0–4095. The syntax of **Op2** is

**SUB Rd, Rn, Rm ; op2 = Rm**

**SUB Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**SUB Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

**SUB Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**SUB Rd, Rn, #constant ; op2 = constant**, where **X** and **Y** are hexadecimal digits :

- produced by shifting an 8-bit unsigned value left by any number of bits
- in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

### Operation

The SUB instruction subtracts the value of **Op2** or **imm12** from the value in **Rn** and stores the sum in **Rd**. This instruction can be followed by a conditional branch.

**Rd = Rn - Op2**

**Rd = Rn - imm12**

### Restrictions (for additional restrictions about PC see data sheet)

- **Op2** must not be **SP** and must not be **PC**.
- **Rd** can be **SP** only with the additional restrictions:
  - **Rn** must also be **SP**.
  - any shift in **Op2** must be limited to a maximum of 3 bits using **LSL**.

### Condition Flags

If **S** is specified, these instructions update the N, Z, C and V flags according to the result.  $R = X + M$ , where **X** is initial register value, **M** is the flexible second operand or the **#imm12** constant, and **R** is the final register value.

N: result is negative

$$N = R_{31}$$

$$Z = \overline{R_{31}} \& \overline{R_{30}} \& L \& \overline{R_0}$$

Z: result is zero

$$V = X_{31} \& \overline{M_{31}} \& \overline{R_{31}} \quad | \quad \overline{X_{31}} \& M_{31} \& R_{31}$$

V: signed overflow

C: unsigned overflow

$$C = \overline{X_{31} \& M_{31}} \quad | \quad M_{31} \& R_{31} \quad | \quad R_{31} \& \overline{X_{31}}$$

## Examples

**SUB R2, R1, R3 ;R2=R1-R3**

**SUBS R6, #240 ;R6=R6-240, sets the flags on the result**

# **UDIV**

## 32-bit Unsigned Division

### Syntax

**UDIV{cond} {Rd,} Rn, Rm**

where **{cond}** is an optional condition code. See Table 3.2. **Rd** is the destination register. If **Rd** is omitted, the destination register is **Rn**. **Rn** is the register holding the value to be dividend. **Rm** is a register holding the divisor.

### Operation

UDIV performs an unsigned integer division of the value in **Rn** by the value in **Rm**. If the value in **Rn** is not divisible by the value in **Rm**, the result is rounded towards zero.

**Rd = Rn / Rm**

### Restrictions

- Do not use **SP** and do not use **PC**.

### Condition Flags

This instruction does not change the flags.

### Examples

**UDIV R0, R2, R4 ;Signed divide, R0 = R2/R4.**

**UDIV R8, R1 ;Unsigned divide, R8 = R8/R1.**

; find the unsigned quotient and remainder

; Inputs: dividend in R0

; divisor in R1

; Outputs: quotient in R2

; remainder in R3

;dividend = divisor\*quotient + remainder

Divide

**UDIV R2,R0,R1 ;R2=R0/R1, R2 is quotient**

**MUL R3,R2,R1 ;R3=(R0/R1)\*R1**

**SUB R3,R0,R3 ;R3=R0%R1, R3 is remainder of R0/R1**

**BX LR ;return**

# Reference Material

Vector address	Number	IRQ	ISR name in Startup.s	NVIC	Priority bits
0x000000038	14	-2	PendSV_Handler	NVIC_SYS_PRI3_R	23 – 21
0x00000003C	15	-1	SysTick_Handler	NVIC_SYS_PRI3_R	31 – 29
0x000000040	16	0	GPIOPortA_Handler	NVIC_PRI0_R	7 – 5
0x000000044	17	1	GPIOPortB_Handler	NVIC_PRI0_R	15 – 13
0x000000048	18	2	GPIOPortC_Handler	NVIC_PRI0_R	23 – 21
0x00000004C	19	3	GPIOPortD_Handler	NVIC_PRI0_R	31 – 29
0x000000050	20	4	GPIOPortE_Handler	NVIC_PRI1_R	7 – 5
0x000000054	21	5	UART0_Handler	NVIC_PRI1_R	15 – 13
0x000000058	22	6	UART1_Handler	NVIC_PRI1_R	23 – 21
0x00000005C	23	7	SSI0_Handler	NVIC_PRI1_R	31 – 29
0x000000060	24	8	I2C0_Handler	NVIC_PRI2_R	7 – 5
0x000000064	25	9	PWM0Fault_Handler	NVIC_PRI2_R	15 – 13
0x000000068	26	10	PWM0_Handler	NVIC_PRI2_R	23 – 21
0x00000006C	27	11	PWM1_Handler	NVIC_PRI2_R	31 – 29
0x000000070	28	12	PWM2_Handler	NVIC_PRI3_R	7 – 5
0x000000074	29	13	Quadrature0_Handler	NVIC_PRI3_R	15 – 13
0x000000078	30	14	ADC0_Handler	NVIC_PRI3_R	23 – 21
0x00000007C	31	15	ADC1_Handler	NVIC_PRI3_R	31 – 29
0x000000080	32	16	ADC2_Handler	NVIC_PRI4_R	7 – 5
0x000000084	33	17	ADC3_Handler	NVIC_PRI4_R	15 – 13
0x000000088	34	18	WDT_Handler	NVIC_PRI4_R	23 – 21
0x00000008C	35	19	Timer0A_Handler	NVIC_PRI4_R	31 – 29
0x000000090	36	20	Timer0B_Handler	NVIC_PRI5_R	7 – 5
0x000000094	37	21	Timer1A_Handler	NVIC_PRI5_R	15 – 13
0x000000098	38	22	Timer1B_Handler	NVIC_PRI5_R	23 – 21
0x00000009C	39	23	Timer2A_Handler	NVIC_PRI5_R	31 – 29
0x0000000A0	40	24	Timer2B_Handler	NVIC_PRI6_R	7 – 5
0x0000000A4	41	25	Comp0_Handler	NVIC_PRI6_R	15 – 13
0x0000000A8	42	26	Comp1_Handler	NVIC_PRI6_R	23 – 21
0x0000000AC	43	27	Comp2_Handler	NVIC_PRI6_R	31 – 29
0x0000000B0	44	28	SysCtl_Handler	NVIC_PRI7_R	7 – 5
0x0000000B4	45	29	FlashCtl_Handler	NVIC_PRI7_R	15 – 13
0x0000000B8	46	30	GPIOPortF_Handler	NVIC_PRI7_R	23 – 21
0x0000000BC	47	31	GPIOPortG_Handler	NVIC_PRI7_R	31 – 29
0x0000000C0	48	32	GPIOPortH_Handler	NVIC_PRI8_R	7 – 5
0x0000000C4	49	33	UART2_Handler	NVIC_PRI8_R	15 – 13
0x0000000C8	50	34	SSI1_Handler	NVIC_PRI8_R	23 – 21
0x0000000CC	51	35	Timer3A_Handler	NVIC_PRI8_R	31 – 29
0x0000000D0	52	36	Timer3B_Handler	NVIC_PRI9_R	7 – 5
0x0000000D4	53	37	I2C1_Handler	NVIC_PRI9_R	15 – 13
0x0000000D8	54	38	Quadrature1_Handler	NVIC_PRI9_R	23 – 21
0x0000000DC	55	39	CAN0_Handler	NVIC_PRI9_R	31 – 29
0x0000000E0	56	40	CAN1_Handler	NVIC_PRI10_R	7 – 5
0x0000000E4	57	41	CAN2_Handler	NVIC_PRI10_R	15 – 13
0x0000000E8	58	42	Ethernet_Handler	NVIC_PRI10_R	23 – 21

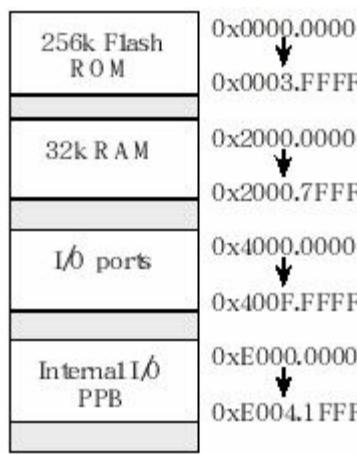
0x0000000EC	59	43	Hibernate_Handler	NVIC_PRI10_R	31 – 29
0x0000000F0	60	44	USB0_Handler	NVIC_PRI11_R	7 – 5
0x0000000F4	61	45	PWM3_Handler	NVIC_PRI11_R	15 – 13
0x0000000F8	62	46	uDMA_Handler	NVIC_PRI11_R	23 – 21
0x0000000FC	63	47	uDMA_Error	NVIC_PRI11_R	31 – 29

Some of the interrupt vectors for the LM3S/TM4C.

C Data type	C99 Data type	Precision	Range
<b>unsigned char</b>	<b>uint8_t</b>	8-bit unsigned	0 to +255
<b>signed char</b>	<b>int8_t</b>	8-bit signed	-128 to +127
<b>char</b>	<b>char</b>	8-bit	ASCII characters
<b>unsigned int</b>	<b>unsigned int</b>	compiler-dependent	
<b>int</b>	<b>int</b>	compiler-dependent	
<b>unsigned short</b>	<b>uint16_t</b>	16-bit unsigned	0 to +65535
<b>short</b>	<b>int16_t</b>	16-bit signed	-32768 to +32767
<b>unsigned long</b>	<b>uint32_t</b>	unsigned 32-bit	0 to 4294967295L
<b>long</b>	<b>int32_t</b>	signed 32-bit	-2147483648L to 2147483647L
<b>float</b>	<b>float</b>	32-bit float	$\pm 10^{-38}$ to $\pm 10^{+38}$
<b>double</b>	<b>double</b>	64-bit float	$\pm 10^{-308}$ to $\pm 10^{+308}$

### BITS 4 to 6

	0	1	2	3	4	5	6	7
0	NUL DLE	SP	0	@	P	`	p	
B 1	SOH XON	!	1	A	Q	a	q	
I 2	STX DC2	"	2	B	R	b	r	
T 3	ETX XOFF	#	3	C	S	c	s	
S 4	EOT DC4	\$	4	D	T	d	t	
5	ENQ NAK	%	5	E	U	e	u	
0 6	ACK SYN	&	6	F	V	f	v	
7	BEL ETB	'	7	G	W	g	w	
T 8	BS CAN	(	8	H	X	h	x	
O 9	HT EM	)	9	I	Y	i	y	
A	LF SUB	*	:	J	Z	j	z	
3 B	VT ESC	+	;	K	[	k	{	
C	FF FS	,	<	L	\	l		
D	CR GS	-	=	M	]	m	}	
E SO	RS	.	>	N	^	n	~	

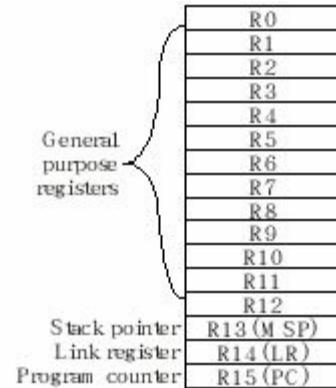


TM4C123

### Standard 7-bit ASCII.

#### Condition code bits

N negative  
Z zero  
V signed overflow  
C carry or  
unsigned overflow



## Memory access instructions

**LDR Rd, [Rn]** ; load 32-bit number at [Rn] to Rd  
**LDR Rd, [Rn,#off]** ; load 32-bit number at [Rn+off] to Rd  
**LDR Rd, =value** ; set Rd equal to any 32-bit value (PC rel)  
**LDRH Rd, [Rn]** ; load unsigned 16-bit at [Rn] to Rd  
**LDRH Rd, [Rn,#off]** ; load unsigned 16-bit at [Rn+off] to Rd  
**LDRSH Rd, [Rn]** ; load signed 16-bit at [Rn] to Rd  
**LDRSH Rd, [Rn,#off]** ; load signed 16-bit at [Rn+off] to Rd  
**LDRB Rd, [Rn]** ; load unsigned 8-bit at [Rn] to Rd  
**LDRB Rd, [Rn,#off]** ; load unsigned 8-bit at [Rn+off] to Rd  
**LDRSB Rd, [Rn]** ; load signed 8-bit at [Rn] to Rd  
**LDRSB Rd, [Rn,#off]** ; load signed 8-bit at [Rn+off] to Rd  
**STR Rt, [Rn]** ; store 32-bit Rt to [Rn]  
**STR Rt, [Rn,#off]** ; store 32-bit Rt to [Rn+off]  
**STRH Rt, [Rn]** ; store least sig. 16-bit Rt to [Rn]  
**STRH Rt, [Rn,#off]** ; store least sig. 16-bit Rt to [Rn+off]  
**STRB Rt, [Rn]** ; store least sig. 8-bit Rt to [Rn]  
**STRB Rt, [Rn,#off]** ; store least sig. 8-bit Rt to [Rn+off]  
**PUSH {Rt}** ; push 32-bit Rt onto stack  
**POP {Rd}** ; pop 32-bit number from stack into Rd  
**ADR Rd, label** ; set Rd equal to the address at label  
**MOV{S} Rd, <op2>** ; set Rd equal to op2

**MOV Rd, #im16 ; set Rd equal to im16, im16 is 0 to 65535**

**MVN{S} Rd, <op2> ; set Rd equal to -op2**

## Branch instructions

**B label ; branch to label Always**

**BEQ label ; branch if Z == 1 Equal**

**BNE label ; branch if Z == 0 Not equal**

**BCS label ; branch if C == 1 Higher or same, unsigned  $\geq$**

**BHS label ; branch if C == 1 Higher or same, unsigned  $\geq$**

**BCC label ; branch if C == 0 Lower, unsigned <**

**BLO label ; branch if C == 0 Lower, unsigned <**

**BMI label ; branch if N == 1 Negative**

**BPL label ; branch if N == 0 Positive or zero**

**BVS label ; branch if V == 1 Overflow**

**BVC label ; branch if V == 0 No overflow**

**BHI label ; branch if C==1 and Z==0 Higher, unsigned >**

**BLS label ; branch if C==0 or Z==1 Lower or same, unsigned  $\leq$**

**BGE label ; branch if N == V Greater than or equal, signed  $\geq$**

**BLT label ; branch if N != V Less than, signed <**

**BGT label ; branch if Z==0 and N==V Greater than, signed >**

**BLE label ; branch if Z==1 or N!=V Less than or equal, signed  $\leq$**

**BX Rm ; branch indirect to location specified by Rm**

**BL label ; branch to subroutine at label**

**BLX Rm ; branch to subroutine indirect specified by Rm**

## Interrupt instructions

**CPSIE I ; enable interrupts (I=0)**

**CPSID I ; disable interrupts (I=1)**

## Logical instructions

**AND{S} {Rd,} Rn, <op2> ; Rd=Rn&op2 (op2 is 32 bits)**

**ORR{S} {Rd,} Rn, <op2> ; Rd=Rn|op2 (op2 is 32 bits)**

**EOR{S} {Rd,} Rn, <op2> ; Rd=Rn^op2 (op2 is 32 bits)**

**BIC{S} {Rd,} Rn, <op2> ; Rd=Rn&(~op2) (op2 is 32 bits)**

**ORN{S} {Rd,} Rn, <op2> ; Rd=Rn|(~op2) (op2 is 32 bits)**

**LSR{S} Rd, Rm, Rs ; logical shift right Rd=Rm>>Rs (unsigned)**

**LSR{S} Rd, Rm, #n ; logical shift right Rd=Rm>>n (unsigned)**

**ASR{S} Rd, Rm, Rs ; arithmetic shift right Rd=Rm>>Rs (signed)**

**ASR{S} Rd, Rm, #n ; arithmetic shift right Rd=Rm>>n (signed)**

**LSL{S} Rd, Rm, Rs ; shift left Rd=Rm<<Rs (signed, unsigned)**

**LSL{S} Rd, Rm, #n ; shift left Rd=Rm<<n (signed, unsigned)**

## Arithmetic instructions

**ADD{S} {Rd,} Rn, <op2> ; Rd = Rn + op2**

**ADD{S} {Rd,} Rn, #im12 ; Rd = Rn + im12, im12 is 0 to 4095**

**SUB{S} {Rd,} Rn, <op2> ; Rd = Rn - op2**

**SUB{S} {Rd,} Rn, #im12 ; Rd = Rn - im12, im12 is 0 to 4095**

**RSB{S} {Rd,} Rn, <op2> ; Rd = op2 - Rn**

**RSB{S} {Rd,} Rn, #im12 ; Rd = im12 – Rn**

**CMP Rn, <op2> ; Rn – op2 sets the NZVC bits**

**CMN Rn, <op2> ; Rn - (-op2) sets the NZVC bits**

**MUL{S} {Rd,} Rn, Rm ; Rd = Rn \* Rm signed or unsigned**

**MLA Rd, Rn, Rm, Ra ; Rd = Ra + Rn\*Rm signed or unsigned**

**MLS Rd, Rn, Rm, Ra ; Rd = Ra - Rn\*Rm signed or unsigned**

**UDIV {Rd,} Rn, Rm ; Rd = Rn/Rm unsigned**

**SDIV {Rd,} Rn, Rm ; Rd = Rn/Rm signed**

**Notes Ra Rd Rm Rn Rt represent 32-bit registers**

**value any 32-bit value: signed, unsigned, or address**

**{S} if S is present, instruction will set condition codes**

**#im12 any value from 0 to 4095**

**#im16 any value from 0 to 65535**

**{Rd,} if Rd is present Rd is destination, otherwise Rn**

**#n any value from 0 to 31**

**#off any value from -255 to 4095**

**label any address within the ROM of the microcontroller**

**op2 the value generated by <op2>**

Examples of flexible operand <op2> creating the 32-bit number. E.g., **Rd = Rn+op2**

**ADD Rd, Rn, Rm ; op2 = Rm**

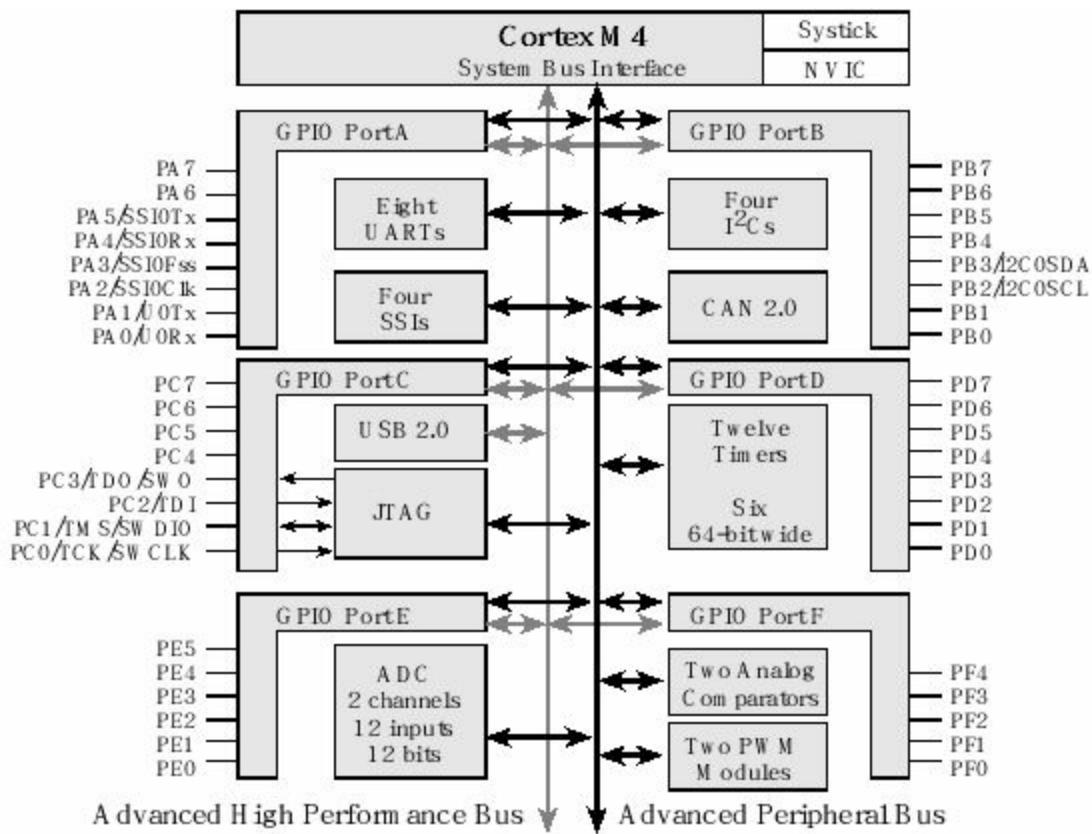
**ADD Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**

**ADD Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**

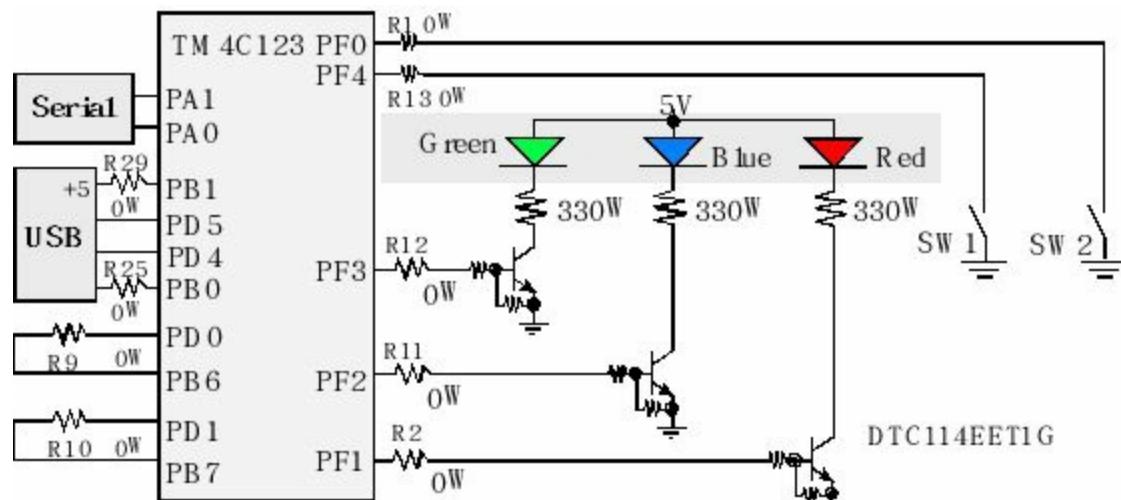
**ADD Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**

**ADD Rd, Rn, #constant ; op2 = constant , where X and Y are hexadecimal digits:**

- produced by shifting an 8-bit unsigned value left by any number of bits in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**



I/O port pins for the TM4C123 microcontroller (LM4F120 is the same except no PWM).



TM4C123 LaunchPad (LM4F120 is the same except no R25 and R29)

IO	Ain	0	1	2	3	4	5	6	7	8	9	14
PA0											CAN1Rx	
PA1											CAN1Tx	
PA2						SSI0Clk						
PA3						SSI0Fss						
PA4						SSI0Rx						
PA5						SSI0Tx						
PA6							I <sub>2</sub> C1SCL			M1PWM2		
PA7							I <sub>2</sub> C1SDA			M1PWM3		
PB0										T2CCP0		
PB1										T2CCP1		

PB2	Port	I <sub>2</sub> C0SCL	T3CCP0	
PB3	Port	I <sub>2</sub> C0SDA	T3CCP1	
PB4	Ain10	Port SSI2Clk	M0PWM2	T1CCP0 CAN0Rx
PB5	Ain11	Port SSI2Fss	M0PWM3	T1CCP1 CAN0Tx
PB6		Port SSI2Rx	M0PWM0	T0CCP0
PB7		Port SSI2Tx	M0PWM1	T0CCP1
PC4	C1-	Port U4Rx	U1Rx	M0PWM6 IDX1 WT0CCP0 U1RTS
PC5	C1+	Port U4Tx	U1Tx	M0PWM7 PhA1 WT0CCP1 U1CTS
PC6	C0+	Port U3Rx		PhB1 WT1CCP0
PC7	C0-	Port U3Tx		WT1CCP1
PD0	Ain7	Port SSI3Clk	SSI1Clk I <sub>2</sub> C3SCL	M0PWM6 M1PWM0 WT2CCP0
PD1	Ain6	Port SSI3Fss	SSI1Fss I <sub>2</sub> C3SDA	M0PWM7 M1PWM1 WT2CCP1
PD2	Ain5	Port SSI3Rx	SSI1Rx	M0Fault0 WT3CCP0
PD3	Ain4	Port SSI3Tx	SSI1Tx	IDX0 WT3CCP1
PD4	USB0DM	Port U6Rx		WT4CCP0
PD5	USB0DP	Port U6Tx		WT4CCP1
PD6		Port U2Rx		M0Fault0 PhA0 WT5CCP0
PD7		Port U2Tx		PhB0 WT5CCP1 NMI
PE0	Ain3	Port U7Rx		
PE1	Ain2	Port U7Tx		
PE2	Ain1	Port		
PE3	Ain0	Port		
PE4	Ain9	Port U5Rx	I <sub>2</sub> C2SCL	M0PWM4 M1PWM2 CAN0Rx
PE5	Ain8	Port U5Tx	I <sub>2</sub> C2SDA	M0PWM5 M1PWM3 CAN0Tx
PF0		Port U1RTS	SSI1Rx CAN0Rx	M1PWM4 PhA0 T0CCP0 NMI C0o
PF1		Port U1CTS	SSI1Tx	M1PWM5 PhB0 T0CCP1 C1o TRD1
PF2		Port	SSI1Clk	M0Fault0 M1PWM6 T1CCP0 TRD0
PF3		Port	SSI1Fss CAN0Tx	M1PWM7 T1CCP1 TRCLK
PF4		Port		M1Fault0 IDX0 T2CCP0 USB0epen

**Table 4.3. PMCx bits in the GPIOCTL register on the LM4F/TM4C specify alternate functions. PD4 and PD5 are hardwired to the USB device. PA0 and PA1 are hardwired to the serial port. PWM not on LM4F120.**