MORGAN & CLAYPOOL PUBLISHERS

Bad to the Bone

Crafting Electronic Systems with BeagleBone Black

SECOND EDITION

Steven F. Barrett
Jason Kridner

Synthesis Lectures on Digital Circuits and Systems

Mitchell A. Thornton, *Series Editor*

Bad to the Bone

Crafting Electronic Systems with BeagleBone Black
Second Edition

Synthesis Lectures on Digital Circuits and Systems

Editor
Mitchell A. Thornton, ***Southern Methodist University***

The *Synthesis Lectures on Digital Circuits and Systems* series is comprised of 50- to 100-page books targeted for audience members with a wide-ranging background. The *Lectures* include topics that are of interest to students, professionals, and researchers in the area of design and analysis of digital circuits and systems. Each *Lecture* is self-contained and focuses on the background information required to understand the subject matter and practical case studies that illustrate applications. The format of a *Lecture* is structured such that each will be devoted to a specific topic in digital circuits and systems rather than a larger overview of several topics such as that found in a comprehensive handbook. The *Lectures* cover both well-established areas as well as newly developed or emerging material in digital circuits and systems design and analysis.

Bad to the Bone: Crafting Electronic Systems with BeagleBone Black, Second Edition
Steven Barrett and Jason Kridner
2016

Fundamentals of Electronics: Book 1 Electronic Devices and Circuit Applications
Thomas F. Schubert and Ernest M. Kim
2015

Applications of Zero-Suppressed Decision Diagrams
Tsutomu Sasao and Jon T. Butler
2014

Modeling Digital Switching Circuits with Linear Algebra
Mitchell A. Thornton
2014

Arduino Microcontroller Processing for Everyone! Third Edition
Steven F. Barrett
2013

Boolean Differential Equations
Bernd Steinbach and Christian Posthoff
2013

Bad to the Bone: Crafting Electronic Systems with BeagleBone and BeagleBone Black
Steven F. Barrett and Jason Kridner
2016

Introduction to Noise-Resilient Computing
S.N. Yanushkevich, S. Kasai, G. Tangim, A.H. Tran, T. Mohamed, and V.P. Shmerko
2013

Atmel AVR Microcontroller Primer: Programming and Interfacing, Second Edition
Steven F. Barrett and Daniel J. Pack
2012

Representation of Multiple-Valued Logic Functions
Radomir S. Stankovic, Jaakko T. Astola, and Claudio Moraga
2012

Arduino Microcontroller: Processing for Everyone! Second Edition
Steven F. Barrett
2012

Advanced Circuit Simulation Using Multisim Workbench
David Báez-López, Félix E. Guerrero-Castro, and Ofelia Delfina Cervantes-Villagómez
2012

Circuit Analysis with Multisim
David Báez-López and Félix E. Guerrero-Castro
2011

Microcontroller Programming and Interfacing Texas Instruments MSP430, Part I
Steven F. Barrett and Daniel J. Pack
2011

Microcontroller Programming and Interfacing Texas Instruments MSP430, Part II
Steven F. Barrett and Daniel J. Pack
2011

Pragmatic Electrical Engineering: Systems and Instruments
William Eccles
2011

Pragmatic Electrical Engineering: Fundamentals
William Eccles
2011

Introduction to Embedded Systems: Using ANSI C and the Arduino Development Environment
David J. Russell
2010

Arduino Microcontroller: Processing for Everyone! Part II
Steven F. Barrett
2010

Arduino Microcontroller Processing for Everyone! Part I
Steven F. Barrett
2010

Digital System Verification: A Combined Formal Methods and Simulation Framework
Lun Li and Mitchell A. Thornton
2010

Progress in Applications of Boolean Functions
Tsutomu Sasao and Jon T. Butler
2009

Embedded Systems Design with the Atmel AVR Microcontroller: Part II
Steven F. Barrett
2009

Embedded Systems Design with the Atmel AVR Microcontroller: Part I
Steven F. Barrett
2009

Embedded Systems Interfacing for Engineers using the Freescale HCS08 Microcontroller II: Digital and Analog Hardware Interfacing
Douglas H. Summerville
2009

Designing Asynchronous Circuits using NULL Convention Logic (NCL)
Scott C. Smith and JiaDi
2009

Embedded Systems Interfacing for Engineers using the Freescale HCS08 Microcontroller I: Assembly Language Programming
Douglas H.Summerville
2009

Developing Embedded Software using DaVinci & OMAP Technology
B.I. (Raj) Pawate
2009

Mismatch and Noise in Modern IC Processes
Andrew Marshall
2009

Asynchronous Sequential Machine Design and Analysis: A Comprehensive Development of the Design and Analysis of Clock-Independent State Machines and Systems
Richard F. Tinder
2009

An Introduction to Logic Circuit Testing
Parag K. Lala
2008

Pragmatic Power
William J. Eccles
2008

Multiple Valued Logic: Concepts and Representations
D. Michael Miller and Mitchell A. Thornton
2007

Finite State Machine Datapath Design, Optimization, and Implementation
Justin Davis and Robert Reese
2007

Atmel AVR Microcontroller Primer: Programming and Interfacing
Steven F. Barrett and Daniel J. Pack
2007

Pragmatic Logic
William J. Eccles
2007

PSpice for Filters and Transmission Lines
Paul Tobin
2007

PSpice for Digital Signal Processing
Paul Tobin
2007

PSpice for Analog Communications Engineering
Paul Tobin
2007

PSpice for Digital Communications Engineering
Paul Tobin
2007

PSpice for Circuit Theory and Electronic Devices
Paul Tobin
2007

Pragmatic Circuits: DC and Time Domain
William J. Eccles
2006

Pragmatic Circuits: Frequency Domain
William J. Eccles
2006

Pragmatic Circuits: Signals and Filters
William J. Eccles
2006

High-Speed Digital System Design
Justin Davis
2006

Introduction to Logic Synthesis using Verilog HDL
Robert B.Reese and Mitchell A.Thornton
2006

Microcontrollers Fundamentals for Engineers and Scientists
Steven F. Barrett and Daniel J. Pack
2006

Copyright © 2016 by Morgan & Claypool

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews, without the prior permission of the publisher.

Bad to the Bone: Crafting Electronic Systems with BeagleBone Black, Second Edition

Steven Barrett and Jason Kridner

www.morganclaypool.com

ISBN: 9781627055116 paperback
ISBN: 9781627055123 ebook
ISBN: 9781627058308 ePub

DOI 10.2200/S00675ED1V01Y201509DCS046

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS

Lecture #47
Series Editor: Mitchell A. Thornton, *Southern Methodist University*
Series ISSN
Synthesis Lectures on Digital Circuits and Systems
Print 1932-3166 Electronic 1932-3174

Bad to the Bone

Crafting Electronic Systems with BeagleBone Black

Second Edition

Steven Barrett
University of Wyoming

Jason Kridner
Texas Instruments

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #47

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

BeagleBone Black is a low-cost, open hardware computer uniquely suited to interact with sensors and actuators directly and over the Web. Introduced in April 2013 by BeagleBoard.org, a community of developers first established in early 2008, BeagleBone Black is used frequently to build vision-enabled robots, home automation systems, artistic lighting systems, and countless other do-it-yourself and professional projects. BeagleBone variants include the original BeagleBone and the newer BeagleBone Black, both hosting a powerful 32-bit, super-scalar ARM Cortex A8 processor capable of running numerous mobile and desktop-capable operating systems, typically variants of Linux including Debian, Android, and Ubuntu. Yet, BeagleBone is small enough to fit in a small mint tin box. The "Bone" may be used in a wide variety of projects from middle school science fair projects to senior design projects to first prototypes of very complex systems. Novice users may access the power of the Bone through the user-friendly BoneScript software, experienced through a Web browser in most major operating systems, including Microsoft Windows, Apple Mac OS X, or the Linux operating systems. Seasoned users may take full advantage of the Bone's power using the underlying Linux-based operating system, a host of feature extension boards (Capes) and a wide variety of Linux community open source libraries. This book provides an introduction to this powerful computer and has been designed for a wide variety of users including the first time novice through the seasoned embedded system design professional. The book contains background theory on system operation coupled with many well-documented, illustrative examples. Examples for novice users are centered on motivational, fun robot projects while advanced projects follow the theme of assistive technology and image-processing applications.

KEYWORDS

BeagleBone, Linux, microcontroller interfacing, embedded systems design, Bonescript, ARM, open source computing

For the students!

Contents

Preface . . . . **xxi**

Acknowledgments . . . . **xxv**

1 Getting Started **1**

1.1 Welcome! 1
1.2 Overview 2
1.3 A Brief Beagle History 3
1.4 BeagleBoard.org Community 4
1.5 BeagleBone Hardware 4
1.5.1 Open-Source Hardware 6
1.6 Developing with Bonescript 6
1.7 BeagleBone Capes 6
1.8 Power Requirements and Capabilities 8
1.9 Getting Started—Success Out of the Box 8
1.9.1 Exercise 1: Accessing Bonescript through your Browser 8
1.9.2 Exercise 2: Blinking an LED with Bonescript 9
1.9.3 Executing the blinkled.js Program 12
1.9.4 Exercise 3: Developing your Own Boneyard—AROO! 12
1.10 Summary 13
1.11 References 14
1.12 Chapter Exercises 15

2 Bonescript **17**

2.1 Overview 17
2.2 Application 1: Bonescript Tour 21
2.3 Application 2: Robot IR Sensor 28
2.4 Application 3: Art Piece Illumination System 33
2.5 Summary 35
2.6 References 35
2.7 Chapter Exercises 35

3 Programming **37**

3.1 An Overview of the Design Process 37
3.2 Overview 37
3.3 Anatomy of a Program 38
3.3.1 Comments 40
3.3.2 Include Files 41
3.3.3 Functions 42
3.3.4 Interrupt Handler Definitions 45
3.3.5 Program Constants 45
3.3.6 Variables 45
3.3.7 Main Function 46
3.4 Fundamental Programming Concepts 47
3.4.1 Operators 47
3.4.2 Programming Constructs 51
3.4.3 Decision Processing 54
3.5 Programming in JavaScript Using Node.js 59
3.5.1 JavaScript 60
3.5.2 Event-driven Programming 60
3.5.3 Node.js 62
3.6 Application: Dagu Magician Autonomous Maze Navigating Robot 66
3.6.1 Dagu Magician Robot 66
3.6.2 Requirements 68
3.6.3 Circuit Diagram 68
3.6.4 Structure Chart 70
3.6.5 UML Activity Diagrams 70
3.6.6 Bonescript Code 70
3.7 Summary 75
3.8 References 76
3.9 Chapter Exercises 76

4 BeagleBone Operating Parameters and Interfacing **79**

4.1 Overview 79
4.2 Operating Parameters 80
4.2.1 BeagleBone 3.3 VDC Operation 80
4.2.2 Compatible 3.3 VDC Logic Families 81
4.2.3 Input/output Operation at 5.0 VDC 82

4.2.4 Interfacing 3.3 VDC Logic Families to 5.0 VDC Logic Families 83
4.3 Input Devices 84
4.3.1 Switches 84
4.3.2 Switch Debouncing 86
4.3.3 Keypads 86
4.3.4 Sensors 88
4.3.5 Transducer Interface Design (TID) Circuit 91
4.3.6 Operational Amplifiers 96
4.4 Output Devices 98
4.4.1 Light-Emitting Diodes (LEDs) 98
4.4.2 Seven-Segment LED Displays 103
4.4.3 Tri-state LED Indicator 103
4.4.4 Dot Matrix Display 106
4.4.5 Liquid Crystal Display (LCD) 106
4.5 High-Power Interfaces 108
4.5.1 High-Power DC Devices 108
4.5.2 DC Motor Speed and Direction Control 109
4.5.3 DC Motor Operating Parameters 110
4.5.4 H-bridge Direction Control 110
4.5.5 DC Solenoid Control 111
4.5.6 Stepper Motor Control 112
4.5.7 Optical Isolation 112
4.6 Interfacing to Miscellaneous Devices 113
4.6.1 Sonalerts, Beepers, Buzzers 113
4.6.2 Vibrating Motor 114
4.6.3 DC Fan 116
4.6.4 Bilge Pump 116
4.7 AC Devices 116
4.8 Application 1: Equipping the Dagu Magician Robot with a LCD 117
4.9 Application 2: the Dagu Magician Interface on a Custom Cape 132
4.10 Application 3: Special Effects LED Cube 137
4.10.1 Construction Hints 139
4.10.2 LED Cube Bonescript Code 139
4.11 Summary 152
4.12 References 152
4.13 Chapter Exercises 153

5 BeagleBone Systems Design 155

5.1 Overview 155
5.2 What Is an Embedded System? 156
5.3 Embedded System Design Process 156
5.3.1 Project Description 156
5.3.2 Background Research 156
5.3.3 Pre-Design 158
5.3.4 Design 158
5.3.5 Implement Prototype 160
5.3.6 Preliminary Testing 161
5.3.7 Complete and Accurate Documentation 161
5.4 Submersible Robot 162
5.4.1 Approach 162
5.4.2 Requirements 162
5.4.3 ROV Structure 164
5.4.4 Structure Chart 166
5.4.5 Circuit Diagram 166
5.4.6 UML Activity Diagram 167
5.4.7 BeagleBone Code 168
5.4.8 Control Housing Layout 182
5.4.9 Final Assembly Testing 182
5.4.10 Final Assembly 184
5.4.11 Project Extensions 184
5.5 Mountain Maze Navigating Robot 184
5.5.1 Description 184
5.5.2 Requirements 186
5.5.3 Circuit diagram 186
5.5.4 Structure Chart 186
5.5.5 UML Activity Diagrams 186
5.5.6 Bonescript Code 186
5.5.7 Mountain Maze 192
5.5.8 Project Extensions 192
5.6 Summary 194
5.7 References 195
5.8 Chapter Exercises 195

6 BeagleBone Features and Subsystems 199

6.1 Overview 199
6.2 Beagling in Linux 200
6.2.1 Communication with BeagleBone Black 201
6.3 Updating your eMMC 206
6.3.1 Updating Your eMMC in MS Windows 207
6.4 A Brief Introduction to Linux 208
6.5 Programming in C using the Linux Toolchain 210
6.6 BeagleBone Features and Subsystems 212
6.6.1 Exposed Function Access 214
6.6.2 Expansion Interface 216
6.7 BeagleBone Black Device Tree and Overlays 216
6.7.1 Overview 216
6.7.2 Binary Tree 217
6.7.3 Device Tree Format 220
6.7.4 Device Tree Related Files 221
6.7.5 BeagleBone Black Device Tree 221
6.7.6 Universal Device Tree Overlay 226
6.8 Programming in C with BeagleBone Black 229
6.8.1 Linux GPIO Files 229
6.8.2 Configuring the GPIO Files 230
6.8.3 Accessing the GPIO Files in C 232
6.9 Analog-to-Digital Converters (ADC) 239
6.9.1 ADC Process: Sampling, Quantization, and Encoding 239
6.9.2 Resolution and Data Rate 241
6.9.3 ADC Conversion Technologies 241
6.9.4 BeagleBone Black ADC system 242
6.9.5 ADC conversion 243
6.9.6 ADC Support Functions in Bonescript 243
6.9.7 Accessing the ADC System in Linux 247
6.9.8 ADC Support Functions in C 248
6.10 Serial Communications 251
6.10.1 Serial Communication Terminology 251
6.10.2 Serial UART 254
6.10.3 Serial Peripheral Interface (SPI) 259
6.11 Precision Timing 267

6.11.1 Timing-Related Terminology 267
6.11.2 BeagleBone Timing Capability 269
6.12 Pulse Width Modulation (PWM) 271
6.12.1 BeagleBone PWM Subsystem (PWMSS) Description 272
6.12.2 Bonescript PWM Support 273
6.12.3 PWM Device Tree Overlay and C Support Functions 273
6.13 Internet of Things—Networking 276
6.13.1 Inter-Integrated Circuit (I2C) Bus 276
6.13.2 Controller Area Network (CAN) Bus 278
6.13.3 Ethernet 278
6.13.4 Internet 280
6.14 Liquid Crystal Display (LCD) Interface 281
6.14.1 C Support Functions 281
6.15 Interrupts 281
6.15.1 Bonescript Interrupt Support 281
6.16 Programmable Real-Time Units 285
6.16.1 Architecture Overview 287
6.16.2 PRU Memory Map 287
6.16.3 PRU Interrupt System 288
6.16.4 PRU Pin Mapping to BeagleBone Black 288
6.16.5 PRU Assembly Program (PASM) 288
6.16.6 Development Process 292
6.17 Summary 304
6.18 References 304
6.19 Chapter Exercises 306

7 BeagleBone "Off the Leash" 309

7.1 Overview 309
7.2 Boneyard II: a Portable Linux Platform-BeagleBone Unleashed 310
7.3 Boneyard III: a Low-Cost Desktop Linux Platform 310
7.3.1 Accessing Bonescript 315
7.3.2 Accessing the Internet 315
7.4 Application 1: Inexpensive Laser Light Show 315
7.5 Application 2: Arbitrary Waveform Generator 320
7.6 Application 3: Robot Arm 320
7.7 Application 4: Weather Station in Bonescript 325

7.7.1 Requirements 325
7.7.2 Structure Chart 325
7.7.3 Circuit Diagram 326
7.7.4 UML Activity Diagrams 326
7.7.5 Bonescript Code 327
7.8 Application 5: Speak & Spell in C 336
7.8.1 BeagleBone C Code 337
7.9 Application 6: Dagu Rover 5 Treaded Robot 342
7.9.1 Description 342
7.9.2 Requirements 342
7.9.3 Circuit Diagram 342
7.9.4 Structure Chart 342
7.9.5 UML Activity Diagrams 342
7.9.6 BeagleBone C Code 346
7.10 Application 7: Portable Image Processing Engine 351
7.10.1 Brief Introduction to Image Processing 351
7.10.2 Image Processing Tasks 353
7.10.3 OpenCV Computer Vision Library 353
7.10.4 Stache Cam 355
7.11 Summary 360
7.12 References 360
7.13 Chapter Exercises 361

8 Where to from Here? 363

8.1 Overview 363
8.2 Software Libraries 363
8.2.1 OpenCV 363
8.2.2 Qt 363
8.2.3 Kinect 364
8.3 Additional Resources 364
8.3.1 OpenROV 364
8.3.2 Ninja Blocks 364
8.3.3 Related Books 365
8.3.4 BeagleBoard.org Resources 367
8.3.5 Contributing to Bonescript 367
8.4 Summary 367
8.5 References 367

8.6 Chapter Exercises 368

A Bonescript functions 369

B LCD interface for BeagleBone in C 373

B.1 BeagleBone LCD Interface 373

B.2 BeagleBone Black LCD C code 373

C Parts List for Projects 387

Authors' Biographies 391

Preface

BeagleBone Black is a low-cost, open hardware computer uniquely suited to interact with sensors and actuators directly and over the Web. Introduced in April 2013 by BeagleBoard.org, a community of developers first established in early 2008, BeagleBone Black is used frequently to build vision-enabled robots, home automation systems, artistic lighting systems and countless other do-it-yourself and professional projects. BeagleBone variants include the original BeagleBone and the newer BeagleBone Black, both hosting a powerful 32-bit, super-scalar ARM Cortex A8 processor capable of running numerous mobile and desktop-capable operating systems, typically variants of Linux including Debian, Android, and Ubuntu. Yet, BeagleBone is small enough to fit in a small mint tin box. The "Bone" may be used in a wide variety of projects from middle school science fair projects to senior design projects to first prototypes of very complex systems. Novice users may access the power of the Bone through the user-friendly BoneScript software, experienced through a Web browser in most major operating systems, including Microsoft Windows, Apple Mac OS X or the Linux operating systems. Seasoned users may take full advantage of the Bone's power using the underlying Linux-based operating system, a host of feature extension boards (Capes) and a wide variety of Linux community open source libraries. This book provides an introduction to this powerful computer and has been designed for a wide variety of users including the first time novice through the seasoned embedded system design professional. The book contains background theory on system operation coupled with many well-documented, illustrative examples. Examples for novice users are centered on motivational, fun robot projects while advanced projects follow the theme of assistive technology and image processing applications.

Texas Instruments has a long history of educating the next generation of STEM (Science, Technology, Engineering, and Mathematics) professionals. BeagleBone is the latest educational innovation in a long legacy which includes the Speak & Spell and Texas Instruments handheld calculators. The goal of the BeagleBone project is to place a powerful, expandable computer in the hands of young innovators. Novice users can unleash the power of the Bone through the user-friendly, browser-based Bonescript environment. As the knowledge and skill of the user develops and matures, BeagleBone provides more sophisticated interfaces including a complement of C-based functions to access the hardware systems aboard the ARM Cortex A8 processor. These features will prove useful for college-level design projects including capstone design projects. The full power of the processor may be unleashed using the underlying onboard Linux-based operating system. A wide variety of hardware extension features are also available using daughter card "Capes" and Linux community open source software libraries. These features allow for the rapid prototyping of complex, expandable embedded systems.

A BRIEF BEAGLE HISTORY

The Beagle family of computer products include BeagleBoard, BeagleBoard xM, the original BeagleBone, and the newest member BeagleBone Black. All have been designed by Gerald Coley, a long-time expert Hardware Applications Engineer, of Texas Instruments. He has been primarily responsible for all hardware design, manufacturing planning, and scheduling for the product line. The goal of the entire product line is to provide users a powerful computer at low cost with no restrictions. As Gerald describes it, his role has been to "get the computers out there and then get out of the way" to allow users to do whatever they want. The computers are intended for everyone and the goal is to provide "out of the box success." He indicated it is also important to provide users a full suite of open source hardware details and to continually improve the product line by listening to customers' desires and keeping the line fresh with new innovations. In addition to the computers, a full line of Capes to extend processor features is available. There are more than 40 different Capes currently available with many more on the way.

BeagleBone was first conceptualized in early Fall 2011. The goal was to provide a full-featured, expandable computer employing the Sitara ARM processor in a small profile case. Jason Kridner told Gerald the goal was to design a processor board that would fit in a small mint tin box. After multiple board revisions, Gerald was able to meet the desired form factor. BeagleBone enjoyed a fast development cycle. From a blank sheet of paper as a starting point, to full production was accomplished, in 90 days.

New designs at Texas Instruments are given a code name during product development. During the early development of the Beagle line, the movie "Underdog" was showing in local theaters. Also, Gerald had a strong affinity to Beagles based on his long-time friend and pet "Jake." Gerald dubbed the new project "Beagle" with the intent of changing the name later. However, the name stuck and the rest is history. Recently, an acronym was developed to highlight the Beagle features:

- **B**ring your own peripherals
- **E**ntry-level costs
- **A**RM Cortex–A8 superscalar processor
- **G**raphics accelerated
- **L**inux and open source community
- **E**nvironment for innovators

APPROACH OF THE BOOK

Concepts will be introduced throughout the book with the underlying theory of operation, related concepts and many illustrative examples. Concepts early in the book will be illustrated using motivational robot examples including an autonomous robot navigating about a two-dimensional maze

based on the low-cost Dagu Magician Chassis (#ROB-10825) robot platform, an autonomous four-wheel drive (4WD) robot (DFROBOT ROBOT0003) in a three-dimensional mountain maze, and a SeaPerch Remotely Operated Vehicle (ROV).

Advanced examples follow a common theme of assistive technology. Assistive technology provides those with challenges the ability to meet their potential using assistive devices. One example covered in this book is a BeagleBone based Speak & Spell. Speak & Spell is an educational toy developed by Texas Instruments in the mid-1970's. It was developed by the engineering team of Gene Franz, Richard Wiggins, Paul Breedlove, and Larry Branntingham.

This book is organized as follows. Chapter 1 provides an introduction to the BeagleBone processor. Chapter 2 provides a brief introduction to the Bonescript programming environment. Chapter 3 provides an introduction to programming and concludes with an extended example of the Dagu Magician robot project. Chapter 4 provides electrical interfacing fundamentals to properly connect BeagleBone to peripheral components. Chapter 5 provides an introduction to system-level design, the tools used to develop a system and two advanced robot system examples.

Chapter 6 begins an advanced treatment of BeagleBone and the ARM Cortex A8 processor and an in-depth review of exposed functions and how to access them using both Bonescript and C functions. Chapter 6 provides an introduction to the universal cape and also the programmable real-time units (PRUs). In Chapter 7 the full power of BeagleBone is unleashed via several system level examples. Chapter 8 briefly reviews additional resources available to the BeagleBone user.

A LITTLE BEAGLE LORE

The beagle dog has been around for centuries and was quite popular in Europe. They were known for their sense of smell and were used to track hare. They are also known for their musical, bugling voice. In *The Beagle Handbook,* Dan Rice, Doctor of Veterinary Medicine (DVM), indicates the Beagle was probably named for their size or voice. A combination of the French words for "open wide" and "throat" results in Begueule and might refer to the Beagle's distinctive and jubilant bugle while on the hunt. Rice also notes "the word beagling isn't found in Webster's dictionary, but it's well recognized in canine fancy. Used by the Beagle community for practically all endeavors that involve their beautiful dogs [Rice, 2000]." We too will use the term "beagling" to describe the fun sessions of working with BeagleBone that are ahead. It is also worth mentioning that Ian Dunbar, Member of the Royal College of Veterinary Surgeons (MRCVS) in *The Essential Beagle* indicates the "overall temperament of the Beagle is bold and friendly." This seems like an appropriate description for the BeagleBone computer running Bonescript software.

THE BEAGLEBOARD.ORG COMMUNITY

The `www.BeagleBoard.org` community has many members. What we all have in common is the desire to put processing power in the hands of the next generation of users. BeagleBoard.org, with Texas Instruments' support, embraced the open source concept with the development and release of BeagleBone in late 2011. Their support will insure the BeagleBone project will be sustainable.

BeagleBoard.org partnered with Circuitco (www.Circuitco.com) to produce BeagleBone and its associated Capes. The majority of the Capes have been designed and fabricated by Circuitco. Clint Cooley, President of Circuitco, is most interested in helping users develop and produce their own ideas for BeagleBone Capes. BeagleBone Black is now manufactured by Circuitco and Element 14. Texas Instruments has also supported the www.BeagleBoard.org community by giving Jason Kridner the latitude to serve as the open platform technologist and evangelist for the BeagleBoard.org community. The most important members of the community are the BeagleBoard and Bone users. Our ultimate goal is for the entire community to openly share their successes and to encourage the next generation of STEM practitioners.

Gerald Coley and Jason Kridner of Texas Instruments and Clint Cooley of CircuitCo are the core team of the BeagleBoard.org Foundation founded in 2008. The Foundation is a US-based 501(c) non-profit corporation existing to provide education in and promotion of the design and use of open-source software and hardware in embedded computing.

Steven Barrett and Jason Kridner
September 2015

Acknowledgments

The authors would like to thank Cathy Wicks and Larissa Swanland of Texas Instruments who proposed this collaboration. We also thank Gerald Coley of Texas Instruments who was interviewed for this book. We also thank Clint Cooley, President of Circuitco, for hosting a tour of his company where BeagleBone and its associated Capes are produced. We would also like to thank Joel Claypool of Morgan & Claypool Publishers for his support of this project and his permission to use selected portions from previous M&C projects. We also thank Professor Walter Schilling, Ph.D. of the Milwaukee School of Engineering who provided a thorough, expert review of the book. We also thank Sara Kreisman for her careful and thorough copy edit of the final manuscript. Also, a special thank you to Jonathan Barrett of Closer to the Sun International for photography support throughout this book (www.closertothesuninternational.com).

Steven Barrett and Jason Kridner
September 2015

CHAPTER 1

Getting Started

Objectives: After reading this chapter, the reader should be able to do the following.

- Provide a brief history of the Beagle computer line.
- Outline the different members of the BeagleBoard.org community.
- Appreciate the role of the BeagleBoard.org community.
- Describe BeagleBone concept of open source hardware.
- Diagram the layout and features of the BeagleBone Black computer.
- Describe BeagleBone Cape concept and available Capes.
- Define the power requirements for BeagleBone computer.
- Download, configure, and successfully execute a test program using the Cloud9 Integrated Development Environment (IDE) and the Bonescript software.
- Design and implement a BeagleBone Boneyard prototype area to conduct laboratory exercises.

1.1 WELCOME!

Welcome to the wonderful world of BeagleBone! Whether you are a first-time BeagleBone user or are seasoned at "Beagling," this book should prove useful. Chapter 1 is an introduction to BeagleBone, its environment and the Beagle community. BeagleBone hosts the Linux operating system; however, the user-friendly Bonescript programming environment may be used in a wide variety of browser environments including: Microsoft Windows, MAC OS X, and Linux. We provide instructions on how to rapidly get up-and-operating right out of the box! Also, an overview of BeagleBone features and subsystems is provided. The chapter concludes with several examples to get you started.

Note: If you can't wait to get started, skip to Section 1.9 "Getting Started—Success Out of the Box." Happy Beagling!

Figure 1.1: BeagleBone. (Figures adapted and used with permission of www.adafruit.com.)

1.2 OVERVIEW

BeagleBone is a low-cost, open-hardware, expandable computer first introduced in November 2011 by www.BeagleBoard.org, a community of developers started by Beagle enthusiasts at Texas Instruments. Various BeagleBone variants host a powerful 32-bit, super-scalar ARM® Cortex^{TM}—A8 processor operating up to 1 GHz. This allows the "Bone" to be used in a wide variety of applications usually reserved for powerful, desktop systems. The Bone is a full-featured computer small enough to fit in a small mint tin box. The combination of computing power and small form factor allows the Bone to be used in a wide variety of projects from middle school science fair projects to senior design projects to first prototypes of very complex systems.

Novice users may access the power of the Bone through the user-friendly, browser-based Bonescript environment in MS Windows, Mac OS X, and Linux. Seasoned users may take full advantage of the Bone's power using the underlying Linux-based operating system, a host of feature extension boards (Capes) and a wide variety of open source libraries.

Texas Instruments has supported BeagleBone development and has a long history of educating the next generation of STEM (Science, Technology, Engineering, and Mathematics) professionals. BeagleBone is the latest education innovation in a long legacy which includes the Speak & Spell and Texas Instruments handheld calculators. The goal of BeagleBone project is

to place a powerful, expandable computer in the hands of young innovators through the user-friendly, browser-based Bonescript environment. As the knowledge and skill of the user develops and matures, BeagleBone provides more sophisticated interfaces including a complement of C-based functions to access the hardware systems aboard the ARM Cortex A8 processor. These features will prove useful for college-level design projects including capstone design projects. The full power of the processor may be unleashed using the underlying onboard Linux-based operating system.

To assemble a custom system, the Bone may be coupled with a wide variety of daughter board "Capes" and open source libraries. These features allow for the rapid prototyping of complex, expandable embedded systems. A full line of Capes to extend processor features is available. There are over 80 different Capes currently available with at least as many more on the way. The Cape system is discussed later in this chapter. Open source libraries are discussed later in the book.

1.3 A BRIEF BEAGLE HISTORY

The Beagle family of computer products include BeagleBoard, BeagleBoard-xM, the original BeagleBone, and the newest member BeagleBone Black. All have been designed by Gerald Coley, a long-time expert Hardware Applications Engineer, of Texas Instruments. He has been primarily responsible for all hardware design, manufacturing planning, and scheduling for the product lines. The goal of the entire platform line is to provide users a powerful computer at a low cost with no restrictions. As Gerald describes it, his role has been to "get the computers out there and then get out of the way" to allow users to do whatever they want. The computers are intended for everyone and the goal is to provide out-of-box success." He indicated it is also important to provide users a full suite of open source hardware details and to continually improve the product line by listening to customers' desires and keeping the line fresh with new innovations.

BeagleBone was first conceptualized in early Fall 2011. The goal was to provide a full-featured, expandable computer employing the Sitara^{TM} AM335x ARM® processor in a small profile case. Jason Kridner, a longtime expert Software Applications Engineer responsible for all Beagle software and co-author of this textbook, of Texas Instruments told Gerald the goal was to design a processor board that would fit in a small mint tin box. After multiple board revisions, Gerald was able to meet the desired form factor. BeagleBone enjoyed a fast development cycle. From a blank sheet of paper starting point to full production was accomplished in 90 days.

New designs at Texas Instruments are given a code name during product development. During the early development of the Beagle line, the movie "Underdog" was showing in local theaters. Also, Gerald had a strong affinity to Beagles based on his long-time friend and pet, "Jake." Gerald dubbed the new project "Beagle" with the intent of changing the name later. However, the name stuck and the rest is history.

The "Underdog" movie was adapted from the cartoon series of the same name. The series debuted in 1964 and featured the hero "Underdog"—who was humble and loveable "Shoeshine

Boy" until danger called. Shoeshine Boy was then transformed into the invincible flying Underdog complete with a cape that allowed him to fly. His famous catch phrase was "Have no fear, Underdog is here!" This series is readily available on Amazon and is a lot of good fun. BeagleBone is a bit like Underdog. It can fit into an unassuming, mint tin box, yet is a full-featured, "fire-breathing," computer equipped with the Linux operating system whose features are extended with Capes.

1.4 BEAGLEBOARD.ORG COMMUNITY

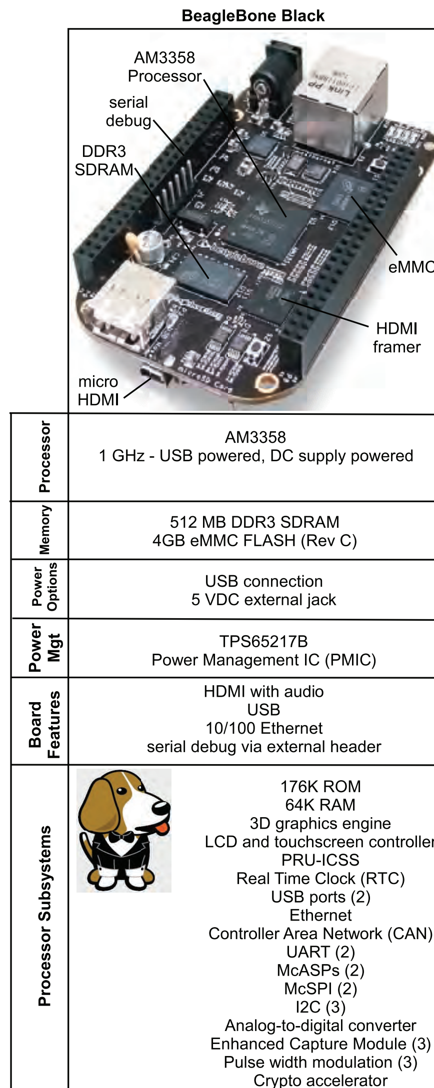
The BeagleBoard.org community has thousands of members. What we all have in common is the desire to put processing power in the hands of the next generation of users. BeagleBoard.org with Texas Instruments support embraced the open-source concept with the development and release of BeagleBone in late 2011. Their support will ensure BeagleBone project remains sustained. BeagleBoard.org partnered with Circuitco (a contract manufacturer) to produce BeagleBone and many of its associated Capes. The majority of the Capes have been designed and fabricated by Circuitco. Clint Cooley, President of Circuitco, is most interested in helping users develop and produce their own ideas for BeagleBone Capes. BeagleBone Black is now manufactured by Circuitco and Element 14. Texas Instruments has also supported the BeagleBoard.org community by giving Jason Kridner the latitude to serve as the open platform technologist and evangelist for the BeagleBoard.org community. The most important members of the community are the BeagleBoard and BeagleBone users (you). Our ultimate goal is for the entire community to openly share their successes and to encourage the next generation of STEM practitioners.

In the next several sections, an overview of BeagleBone hardware, system configuration, and software is discussed.

1.5 BEAGLEBONE HARDWARE

There are two different variants of BeagleBone: The original BeagleBone, released in late 2011, and BeagleBone Black released in early 2013. This new book edition concentrates on the BeagleBone Black; however, all Bonescript examples provided in the book will execute on both BeagleBone variants. Also, the original BeagleBone may be uploaded with the Debian Linux release shipped on BeagleBone Black. The features of BeagleBone Black is summarized in Figure 1.2.

The BeagleBone Black fits inside a small mint box. It hosts the Sitara^{TM} AM3358 ARM® processor which operates at a maximum frequency of 1 GHz. BeagleBone Black is equipped with a 4 GB eMMC (embedded MultiMedia Card). This provides for non-volatile mass storage in an integrated circuit package. The eMMC acts as the "hard drive" for BeagleBone Black board and hosts the Linux operating system, Cloud9 Integrated Development Environment (IDE) and Bonescript. A micro SD card connector is available for expanding storage capability. The BeagleBone Black may also be booted from the SD card. BeagleBone Black is equipped with an HDMI (High-Definition Multimedia Interface) framer and micro connector. HDMI is a compact digi-



[source: BeagleBone System Reference Manual]

Figure 1.2: BeagleBone Black features [Coley, 2014].(Figures adapted and used with permission of `www.beagleboard.org`.)

tal audio/interface which is commonly used in commercial entertainment products such as DVD players, video game consoles, and mobile phones. BeagleBone Black costs approximately US $55 [Coley, 2014].

1.5.1 OPEN-SOURCE HARDWARE

To spur the development and sharing of new developments and features among the BeagleBoard community, all hardware details of BeagleBone boards are readily available as open source. This includes detailed schematics, bill-of-materials and printed circuit board (PCB) layout. This allows for custom designs to take full advantage of BeagleBone-based products at the lowest possible cost. Hardware details are readily available at `www.beagleboard.org`.

1.6 DEVELOPING WITH BONESCRIPT

Bonescript helps to provide a user-friendly, browser-based programming environment for BeagleBone. Bonescript consists of a JavaScript library of functions to rapidly develop a variety of physical computing applications. Bonescript is accessed via a built-in web server that is pre-installed on the BeagleBone board. Because Bonescript programming is accessed over your browser, it may be developed from any of MS Windows, Mac OS X, and Linux. Exercise 1 provided later in the chapter provides easy-to-use instructions on how to quickly get Bonescript-based program up and operating. Although Bonescript is user-friendly, it may also be used to rapidly prototype complex embedded systems.

1.7 BEAGLEBONE CAPES

A wide variety of peripheral components may be easily connected to BeagleBone. A series of Capes have been developed to give BeagleBone "super powers" (actually, using these plug-in boards provide BeagleBone with additional functionality). Currently, there are over 80 different capes available with many more on the way. See `www.beaglebonecapes.com` for the latest information on available capes. BeagleBone may be equipped with up to four Capes at once. Each Cape is equipped with an onboard EEPROM to store Cape configuration data. For example, Capes are available to provide BeagleBone:

- a 7-in, 800 x 400 pixel TFT LCD (liquid crystal display) (LCD7 Cape);
- a battery pack (Battery Cape);
- a Controller Area Network (CAN) interface (CANbus Cape);
- separate Capes for interface to a wide variety of devices including DVI-D, Profibus, RS-232, RS-485 and VGA standards;
- an interface for two stepper motors;

- a Global Positioning System (GPS) cape;
- a high-definition camera cape;
- an inertial navigation system cape;
- DC motor control cape;
- weather and environmental sensor cape; and
- radio frequency (RF) link cape.

There is also a Cape equipped with a solderless breadboard area to provide a straightforward interface to external components. This Cape is illustrated in Figure 1.3 along with a representative sample of other Capes.

Figure 1.3: BeagleBone Capes: (Top left) prototype Cape; (top right) Liquid Crystal Display LCD7 Cape—Figures used with permission of Circuitco (www.circuitco.com); (bottom left) Controller Area Network Cape; (bottom right) motor control Cape—Figures used with permission of Circuitco (www.circuitco.com).

Most of these Capes have been designed by www.BeagleBoard.org and are manufactured by Circuitco in Richardson, Texas. Both BeagleBoard.org and Circuitco are interested in manufacturing Cape designs submitted by the BeagleBoard.org community.

1.8 POWER REQUIREMENTS AND CAPABILITIES

Figure 1.4: BeagleBone power source. (Photo courtesy of J. Barrett [2015], Closer to the Sun International).

BeagleBone may be powered from the USB cable, an external 5 VDC power supply or via the Battery Cape. An external 5 VDC supply may be connected to the external power supply connector. The 5 VDC supply requires a minimum 1 amp current rating and must be equipped with a 2.1 mm center positive with a 5.5 mm outer barrel [Coley, 2014]. A BeagleBone-compatible 5 VDC, 2A power supply is available from Adafruit (`www.adafruit.com`).

1.9 GETTING STARTED—SUCCESS OUT OF THE BOX

It is important to the BeagleBoard designers that a novice user experience out-of-box success on first time use. In this section we provide a series of exercises to quickly get up and operating with BeagleBone and Bonescript.

1.9.1 EXERCISE 1: ACCESSING BONESCRIPT THROUGH YOUR BROWSER

For the novice user, the quickest way to get up and operating with Bonescript is through your web browser. Detailed step-by-step, quick-start instructions are provided in the online "BeagleBone

Quick-Start Guide" available at www.beagleboard.org. An abbreviated "Quick-Start Guide" ships with BeagleBone Black. The guide consists of four easy-to-follow the steps summarized below. They may be accomplished in less than 10 min to allow rapid out-of-box operation [www.beagleboard.org].

1. Plug BeagleBone into the host computer via the mini USB cape and open the START.htm file.

2. Install the proper drivers for your system. MS Windows users need to first determine if the host computer is running 32- or 64-bit Windows. Instructions to do this are provided at http://support.microsoft.com/kb/827218.

3. Browse "Information on BeagleBoard" information using Chrome or Firefox by going to http://192.168.7.2. (Internet Explorer is not recommended for use with BeagleBone Black.)

4. Explore the Cloud9 IDE develop environment by navigating to http://192.168.7.2:3000/ using Chrome or Firefox.

1.9.2 EXERCISE 2: BLINKING AN LED WITH BONESCRIPT

In this exercise we blink a light-emitting diode (LED) onboard BeagleBone and also an external LED. Bonescript will be used along with the blinkled.js program. The LED interface to the BeagleBone is shown in Figure 1.5 and also Figure 1.6. The interface circuit consists of a 270 ohm resistor in series with the LED. This limits the current from BeagleBone to a safe value below 8 mA. The LED has two terminals: the anode (+) and cathode (–). The cathode is the lead closest to the flattened portion of the LED when viewed from above. When connecting the LED circuit, pay close attention to the orientation of the BeagleBone computer. With the 5 VDC receptacle in the upper-left portion of the computer, header P9 is on the left and header P8 is on the right. The pins of each header are numbered 1–46, as shown in Figure 1.5.

The blinkled.js program is provided in the code listing below. The purpose of this program is to blink two different LEDs: BeagleBone (LED USR3) and an external LED connected to header P8 pin 13. A detailed description of Bonescript is provided in the next chapter. We provide a brief description here of code operation. In the first line of the code, the Bonescript library must be included. The library contains function definitions used within the program. The pin designators are then assigned to variable names. Use of variable names makes to code more readable. The pins are then configured as general purpose outputs. The initial state of the LEDs are set to logic LOW turning the LEDs off. A JavaScript timer is then configured that runs the function "toggle" once every second (1000-ms) via the function "SetInterval." The SetInterval function executes a function (in this example the "toggle" function) at intervals specified by the delay in milliseconds. Within the function "toggle," the variable "state" is toggled between the LOW and HIGH logic states and used to set the state of the pins driving the LEDs.

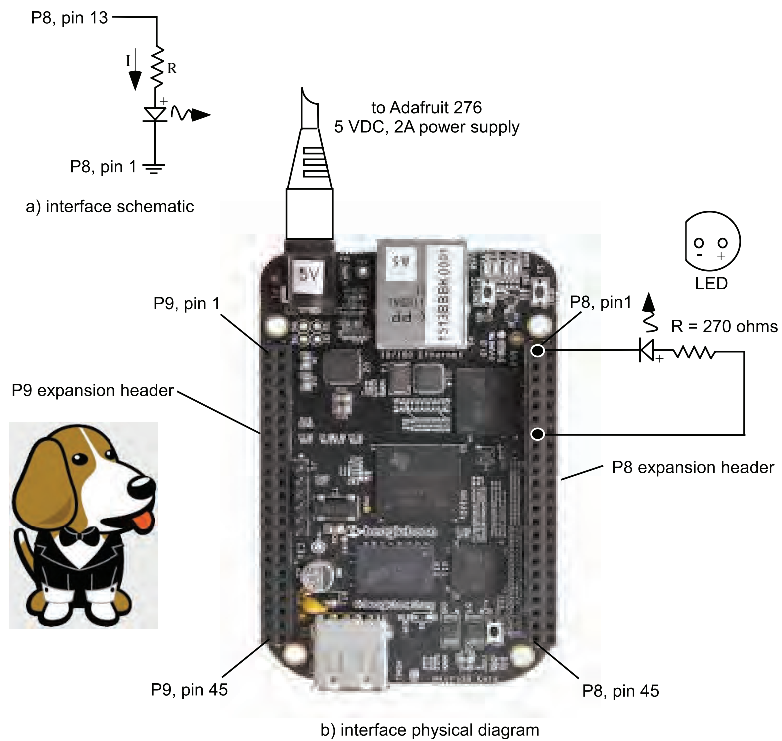


Figure 1.5: Interfacing an LED to BeagleBone. **Notes:** The current being supplied by the BeagleBone pin should not exceed 8 mA. (Illustrations used with permission of Texas Instruments (www.TI.com).)

```
1  //********************************************************************
2  var b = require('bonescript');    //include Bonescript library
3                                    //contains function definitions
4
5  var ledPin  = ''P8_13'';          //Set variable ledPin to P8.13
6  var ledPin2 = ''USR3'';           //Set variable ledPin2 to onboard
7                                    //LED USR3
8
9  b.pinMode(ledPin, b.OUTPUT);      //Configure pin as output
10 b.pinMode(ledPin2, b.OUTPUT);     //Configure pin as output
11
12 var state = b.LOW;
13 b.digitalWrite(ledPin, state);    //set ledPin  logic LOW
14 b.digitalWrite(ledPin2, state);   //set ledPin2 logic LOW
15
16 setInterval(toggle, 1000);        //call function toggle at 1s
17                                   //interval
```

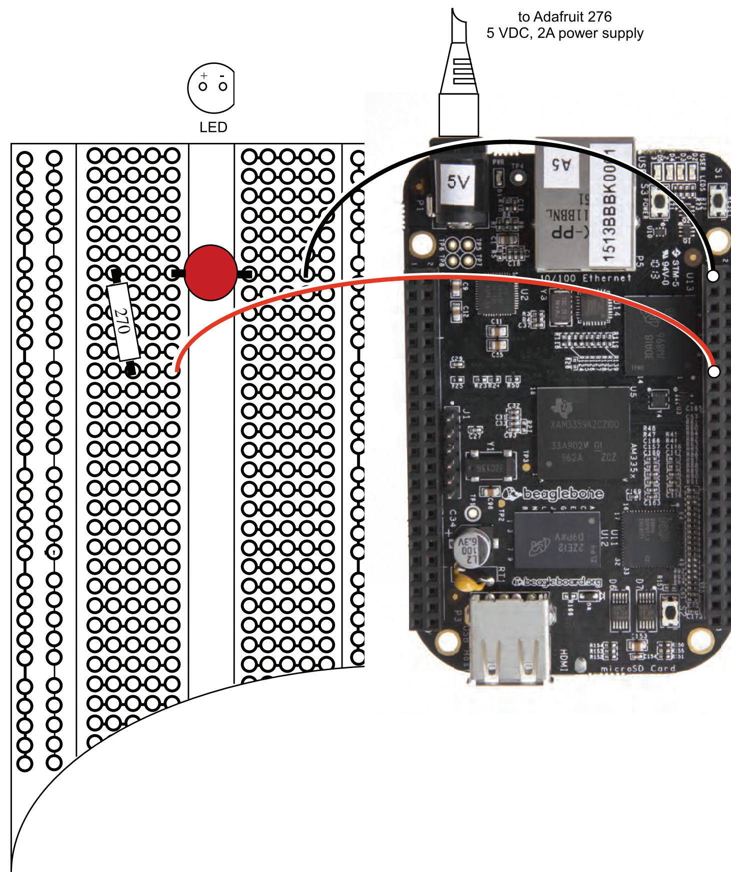


Figure 1.6: Interfacing an LED to BeagleBone using a prototype board. **Notes:** The current being supplied by the BeagleBone pin should not exceed 8 mA. (Illustrations used with permission of Texas Instruments (www.TI.com).)

```
18
19 //*****************************************************************
20 //toggle: toggles (flips) LED state
21 //*****************************************************************
22
23 function toggle()
24   {
25   if(state == b.LOW)
26     {
27     state = b.HIGH;
28     }
```

```
29      else
30        {
31        state = b.LOW;
32        }
33      b.digitalWrite(ledPin, state);        //update LED value
34      b.digitalWrite(ledPin2, state);       //update LED value
35      }
36
37  /*****************************************************************
```

1.9.3 EXECUTING THE BLINKLED.JS PROGRAM

To execute the program, Cloud9 IDE is started as described earlier in the chapter by navigating to http://192.168.7.2:3000 via Chrome or Firefox. The blinkled.js program is selected and the "run" button is depressed. The LEDs will commence blinking!

The program may also be launched from the Linux command line at the bottom of the Cloud9 screen. Insure you are in the demo directory (where the program resides) using the following command:

```
>cd /var/lib/cloud9/demo/
```

Once in the demo directory, the program is launched using:

```
/var/lib/cloud9/demo> node blinkled.js
```

The program may be stopped by asserting a [Ctrl]-C at the Linux command line.

1.9.4 EXERCISE 3: DEVELOPING YOUR OWN BONEYARD—AROO!

In this exercise you develop a prototype board to exercise BeagleBone. We have dubbed this the "Boneyard." In the spirit of "Do-it-yourself (DIY)," we encourage you to develop your own Boneyard by using your creativity and sense of adventure.

If your schedule does not permit time to design and build your own Boneyard, "way cool" prototype boards and cases are available from Adafruit and built-to-spec.com. Adafruit (www.adafruit.com) provides a plethora of Beagle-related products including an Adafruit proto plate and Bone Box pictured in Figure 1.7. Built-to-spec also offers a variety of BeagleBone products including prototype boards and enclosures. Several are shown in Figure 1.8.

For our version of the Boneyard, we have used available off-the-shelf products as shown in Figure 1.9. The Boneyard includes the following:

- a black Pelican Micro Case #1040;
- a BeagleBone evaluation board;
- two Jameco JE21 3.3 x 2.1 inch solderless breadboards;

Figure 1.7: Sample of Adafruit BeagleBone products [www.adafruit.com].

Figure 1.8: Sample of built-to-spec BeagleBone products [www.built-to-spec.com].

- a Jameco #106551 circuit board hardware mounting kit; and
- one piece of black plexiglass acrylic sheet.

To construct the Boneyard, the shape of the Pelican Micro Case interior lid is traced on to the protective paper on the black plexiglass acrylic sheet. The plexiglass platform is then cut out using a scroll saw. Rough edges may then be smoothed using small grit sandpaper. The BeagleBone evaluation board is mounted to the plexiglass using the circuit board hardware mounting kit. The solderless breadboards are mounted using the peel and stick backing.

We encourage you to use your own imagination to develop and construct your own BeagleBone Boneyard. Also, a variety of Beagle related stickers are available from Café Press www.cafepress.com to customize your boneyard.

1.10 SUMMARY

This chapter has provided an introduction to BeagleBone Black, its environment, and Beagle community. Also, an overview of BeagleBone Black features and subsystems was provided. The chapter concluded with several examples to get you started. In the next chapter a detailed introduction to programming and programming tools is provided.

Figure 1.9: BeagleBone Boneyard. (Photo courtesy of Barrett, J. [2015] , Closer to the Sun International).

1.11 REFERENCES

- Barret, J. "Closer to the Sun International." 2015; www.closertothesungallery.com.
- Coley, G. *BeagleBone Black Rev C Systems Reference Manual.* May 22, 2014; www.beagleboard.org.
- CircuitCo—Printed Circuit Board Solutions, 2015; www.circuitco.com.
- Dulaney, E. *Linux All-In-One for Dummies.* Hoboken, NJ: Wiley Publishing, Inc., 2010.
- Octavio. "First steps with BeagleBone." 2015; www.borderhack.com.
- "Adafruit Industries." 2015; www.adafruit.com.
- "Built-to-Spec." 2015; www.built-to-spec.com

1.12 CHAPTER EXERCISES

1. What processor is hosted aboard the BeagleBone Black? Summarize the features of the processor.
2. What is the difference between the BeagleBone single board computer and a microcontroller based hobbyist board?
3. What is the operating system aboard BeagleBone? What are the advantages of using this operating system?
4. Describe the Beagle community.
5. Where is the operating system stored on BeagleBone Black?
6. What is the Cloud9 IDE?
7. Summarize the features of Bonescript.
8. Describe the concept of open-source hardware.
9. What is a BeagleBone Cape? How many can be used simultaneously? How are conflicts prevented between Capes?
10. Modify the blinkled.js program such that one LED blinks at three times the rate of the other.
11. Modify the blinkled.js program with two external LEDs (red and green). The red LED should be on while the other is off and vice versa.

CHAPTER 2

Bonescript

Objectives: After reading this chapter, the reader should be able to do the following.

- Describe the key features of the Bonescript library, Node.js interpreter, and Cloud9 Integrated Development Environment (IDE).
- Describe what features of the Bonescript library, Node.js, and Cloud9 IDE ease the program development process.
- Write programs for use on BeagleBone using Bonescript.

2.1 OVERVIEW

This brief chapter provides an introduction to the Bonescript programming environment. Bonescript is a user-friendly, easy-to-use library and environment to harness the power and features of BeagleBone. It was designed to allow one to quickly develop electronic systems. Bonescript consists of a library of event driven functions to use different systems aboard BeagleBone and a set of services running at boot-up to expose many of the functions to a web browser using the socket.io Node.js library. A list of available functions is provided in Figure 2.1. A detailed description of each function is provided in Appendix A. We introduce each function on an as-needed basis in upcoming examples.

The Bonescript functions are easily used within the Cloud9 IDE programming environment illustrated in Figure 2.2. As can be seen, the Cloud9 IDE environment provides familiar file handling features.

Programs for BeagleBone are often written within the Cloud9 IDE programming environment using the Bonescript functions. The program is then executed simply by pushing the "Run." Provided below is the basic formats of Bonescript programs. We then illustrate program writing with a series of examples.

```
1 //***********************************************************
2 //format 1
3 //***********************************************************
4
5 var b = require('bonescript');
6
7 //define variables used in program
8 var
9   :
```

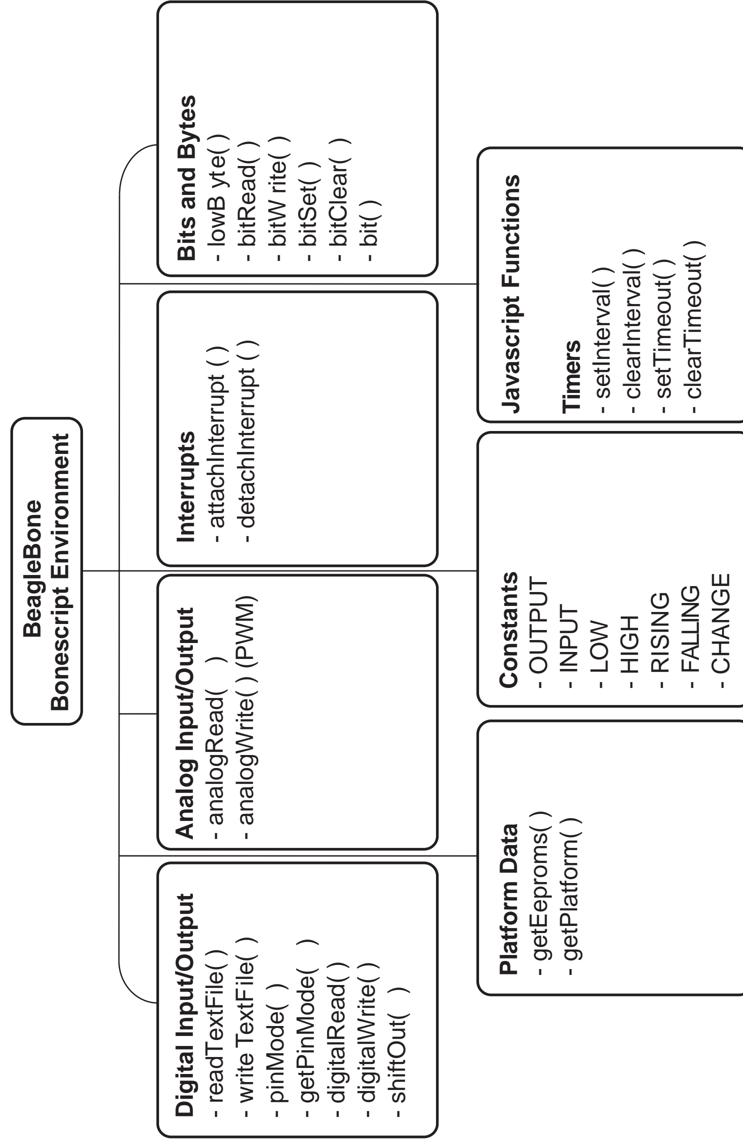


Figure 2.1: Bonescript development environment.

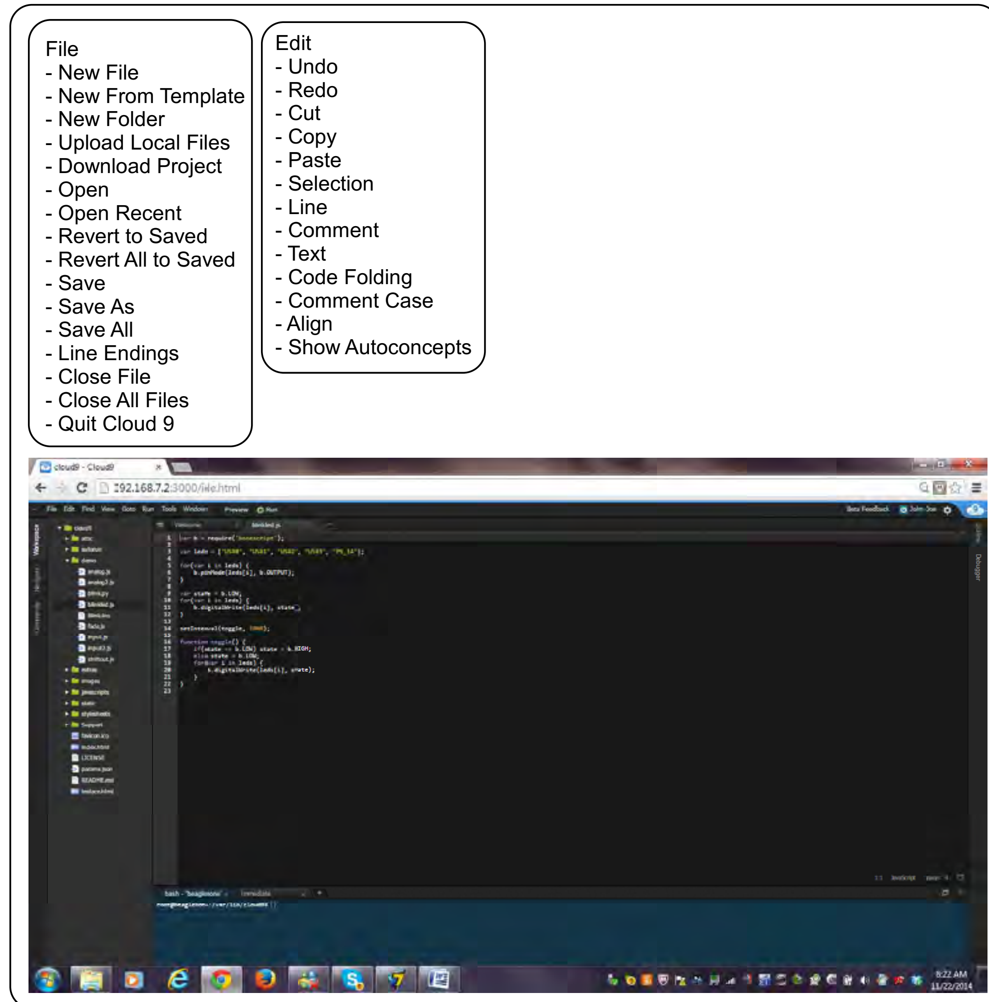


Figure 2.2: Cloud9 IDE programming environment. A close–up of the File and Edit drop down menus are provided for reference.

```
10    :
11
12 //set input and output pin configurations
13 //initialize required systems
14 b.pinMode(output_pin, b.OUTPUT);
15 b.pinMode(input_pin, b.INPUT);
16
17 loop();
18
19 //function loop repeats continuously
20
21 function loop()
22 {
23 function1(argument 1, argument 2);
24 setTimeout(loop, 1);
25 }
26
27 //*******************************************************
28 //function1
29 //*******************************************************
30
31 function function1(variable 1, variable 2)
32 {
33 //function body
34    :
35    :
36 }
37
38 //*******************************************************
```

```
 1 //*******************************************************
 2 //format 2
 3 //*******************************************************
 4
 5 var b = require('bonescript');
 6
 7 //define variables used in program
 8 var
 9    :
10    :
11
12 //set input and output pin configurations
13 //initialize required systems
14 b.pinMode(output_pin, b.OUTPUT);
15 b.pinMode(input_pin, b.INPUT);
16
17 setInterval(do_something, 1000);
18
19 function do_something()
```

```
20 {
21
22 }
23
24 //************************************************************************
```

2.2 APPLICATION 1: BONESCRIPT TOUR

In this first application exercise, we investigate some of the onboard Bonescript examples provided in the Cloud9 IDE. The first example was the "blinkled.js" program completed in Chapter 1, Exercise 2. If you have not completed this exercise, go back and do so before proceeding.

We now extend this example to include a switch. The switch is connected as shown in Figure 2.3. When the switch is pressed the LED is illuminated. When the switch is released the LED goes off.

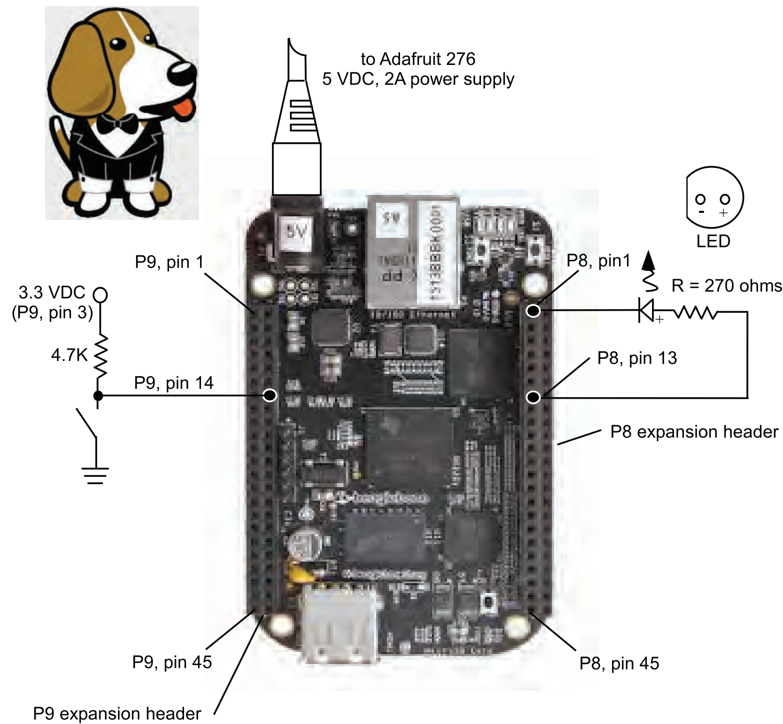


Figure 2.3: Interfacing an LED and switch to BeagleBone. **Notes:** The current being supplied by the BeagleBone pin should not exceed 8 mA. (Illustrations used with permission of Texas Instruments (www.TI.com).)

```
//***************************************************************
//switch_input.js
//***************************************************************

var b = require('bonescript');

var outputPin = "P9_14";
var inputPin = "P8_13";
var state = b.LOW;

b.pinMode(inputPin, b.INPUT);
b.pinMode(outputPin, b.OUTPUT);
b.digitalWrite(outputPin, b.HIGH);


setInterval(check_switch, 1000);   //check switch status every 1s

function check_switch()
  {
  state = b.digitalRead(inputPin);
  if ((state) == b.LOW)
    {
    b.digitalWrite(outputPin, b.HIGH);
    }
  else
    {
    b.digitalWrite(outputPin, b.LOW);
    }

  }

//*****************************************************************
```

In the next example, program "analog2.js" is used to measure the voltage from a potentiometer. The wiper arm of the potentiometer is connected to header P9, pin 36. The voltage at the wiper arm is reported on the console display within the Cloud9 IDE.

A potentiometer is a variable resistor. It has a fixed value between two of its terminals. A third terminal, called the wiper arm, provides a resistance that is related to the wiper arm position. For example, if the potentiometer has a fixed resistance value of 10 kOhms and the wiper arm is in the mid-position, the resistance reading at the wiper arm with respect to ground will be 5 kOhms.

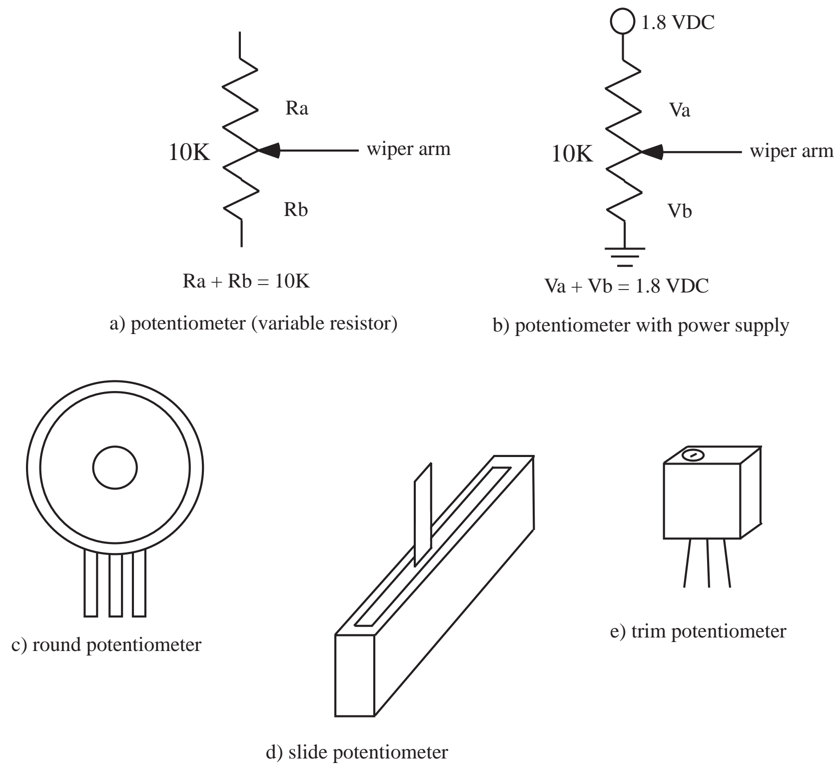


Figure 2.4: Potentiometer.

If the potentiometer is connected to a voltage source and ground as shown in Figure 2.4b), the output voltage will be proportional to the wiper arm position. Various styles of potentiometers are shown in Figure 2.4c–e.

Note: Do not exceed 1.8 VDC on any of the BeagleBone Black's analog input pins.[1]

In this example, a potentiometer is connected between 1.8 VDC supply and ground. The wiper arm is connected to header P9, pin 36 as shown in Figure 2.5. As the potentiometer position is changed, the analog value read is reported to the Cloud9 IDE console. Depending on the position of the potentiometer, a value between 0 (0 VDC) and 1 (1.8 VDC) will be reported on the Console screen within Bonescript.

```
1 //*******************************************************************
2 //analog2.js
3 //*******************************************************************
4
```

[1]This note will be repeated throughout the text every time the ADC system is used. Failure to comply with the note could potentially damage the BeagleBone Black.

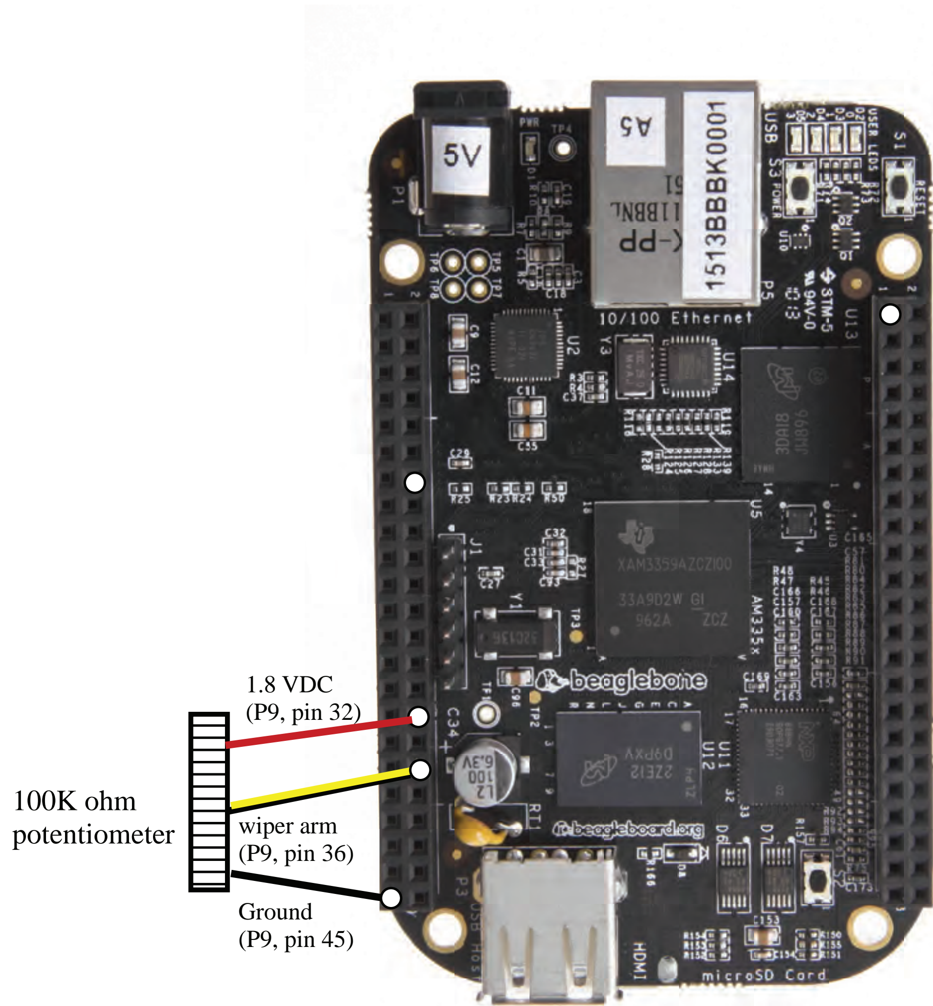


Figure 2.5: Potentiometer connected to header P9, pin 36. (Illustrations used with permission of Texas Instruments (www.TI.com).)

```
5 var b = require('bonescript');
6
7 inputPin = ``P9_36'';
8
9 loop();
10
11 function loop() {
12     var value = b.analogRead(inputPin);
13     console.log(value);
14     setTimeout(loop, 1);
15 }
16 //****************************************************************
```

In this next example, an LED is connected to header P8, pin 13 as shown in Figure 2.6. As the potentiometer value is changed, the corresponding intensity of the LED is changed. This is accomplished using program "analog.js" by reading in the analog value from the potentiometer on header P9, pin 36 and reading out the corresponding value on header P8, pin 13 using the Bonescript "b.analogWrite" function.

```
1 //****************************************************************
2 //analog.js
3 //****************************************************************
4
5 var b = require('bonescript');
6
7 inputPin = ``P9_36'';
8 outputPin = ``P8_13'';
9
10 b.pinMode(outputPin, b.OUTPUT);
11 loop();
12
13 function loop() {
14     var value = b.analogRead(inputPin);
15     b.analogWrite(outputPin, value);
16     setTimeout(loop, 1);
17 };
18 //****************************************************************
```

In the next example, an LED is connected to header P8, pin 13, as shown in Figure 2.7. When the "fade.js" program is executed the intensity of the LED gradually becomes brighter, gets to a maximum value, goes down to a minimum value, and repeats. This is accomplished by setting an initial value of LED intensity and incrementally increasing the value every 10 ms. When the maximum value of intensity is reached, the intensity value is incrementally decreased. The UML activity diagram for fade.js is provided in Figure 2.8.

```
1 //****************************************************************
2 //fade.js
3 //****************************************************************
```

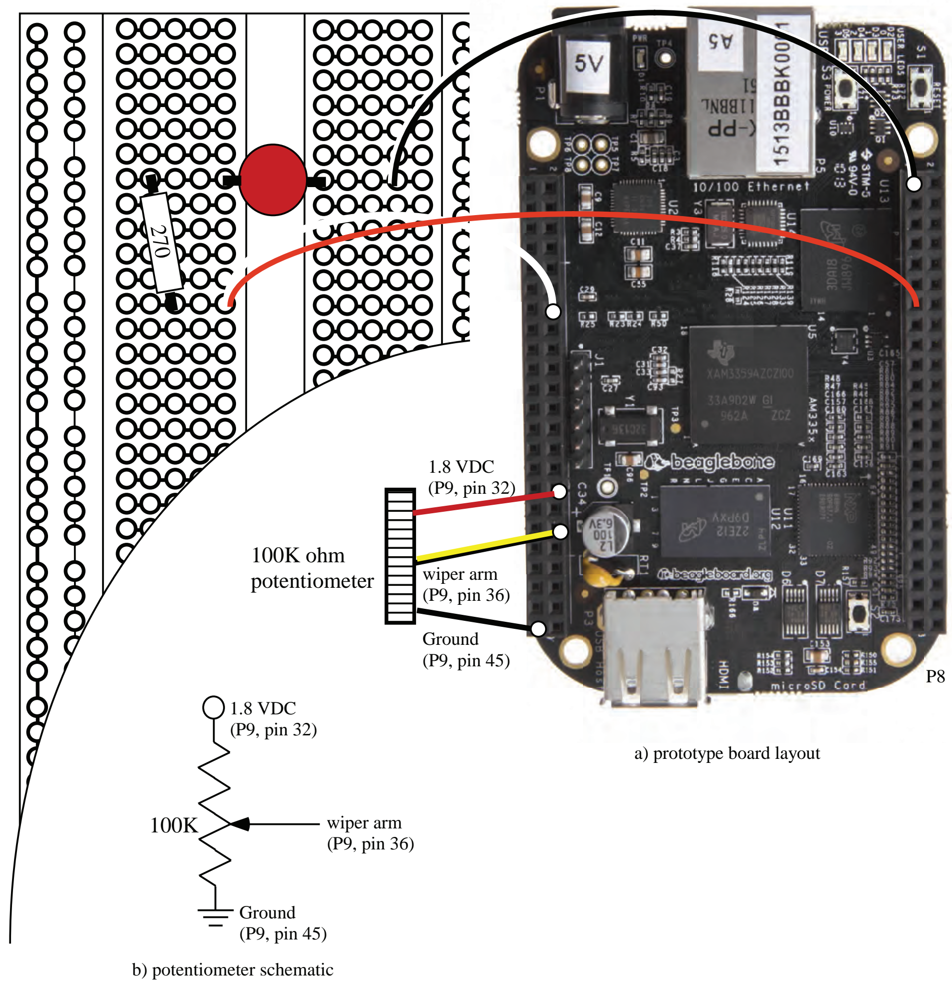


Figure 2.6: Potentiometer connected to header P9, pin 36. As the potentiometer value is changed, the corresponding intensity of the LED is changed. (Illustrations used with permission of Texas Instruments (`www.TI.com`).)

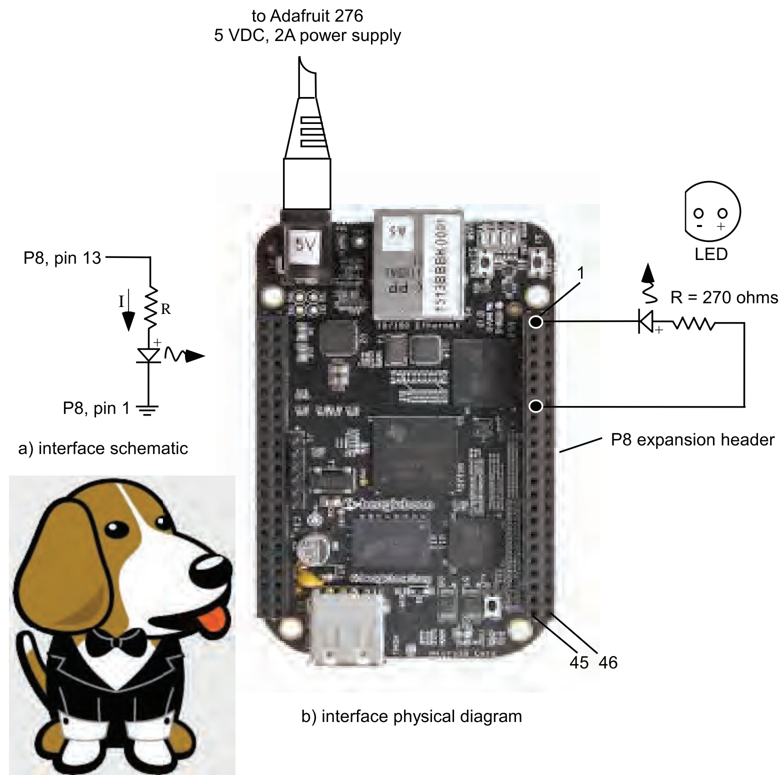


Figure 2.7: Interfacing an LED to BeagleBone. **Notes:** The current being supplied by the BeagleBone pin should not exceed 8 mA. (Illustrations used with permission of Texas Instruments (www.TI.com).)

```
4
5 var b = require('bonescript');
6
7 // setup starting conditions
8 var awValue = 0.01;              //LED intensity increment value
9 var awDirection = 1;
10 var awPin = ``P8_13'';
11
12 // configure pin
13 b.pinMode(awPin, b.OUTPUT);
14
15 // call function to update brightness every 10ms
16 setInterval(fade, 10);
17
18 // function to update brightness
19 function fade()
```

```
20 {
21                                      //Write analog value to LED
22 b.analogWrite(awPin, awValue);
23                                      //Increment LED intensity value
24 awValue = awValue + (awDirection*0.01);
25
26 if(awValue > 1.0)
27   {
28   awValue = 1.0;
29   awDirection = -1;                  //Switch intensity ramp direction
30   }
31 else if(awValue <= 0.01)
32   {
33   awValue = 0.01;
34   awDirection = 1;                   //Switch intensity ramp direction
35   }
36 }
37
38 //*******************************************************************
```

2.3 APPLICATION 2: ROBOT IR SENSOR

As mentioned earlier, we use autonomous, maze-navigating robots several times throughout the book as electronic systems examples. An autonomous, maze-navigating robot is equipped with sensors to detect the presence of maze walls and navigate about the maze. The robot has no prior knowledge about the maze configuration. It uses the sensors and an onboard algorithm to determine the robot's next move. The overall goal is to navigate from the starting point of the maze to the end point as quickly as possible without bumping into maze walls as shown in Figure 2.9. Maze walls are usually painted white to provide a good, light reflective surface, whereas the maze floor is painted matte black to minimize light reflections.

We equip BeagleBone with a single Sharp GP2Y0A21YKOF IR sensor. This is the same sensor used in the robot. The goal is to become acquainted with the sensor profile and use the sensor to assert a digital output. In the next exercise we use an IR sensor to control a pulse width modulated (PWM) output. This is a good stepping stone to the maze navigating robot example provided in the next chapter.

Next, we link a BeagleBone analog input to a digital output. One of the Sharp IR sensors will be connected to an analog input. The IR sensor provides a voltage output as shown in Figure 2.10. The output voltage from the sensor increases and peaks at a specific range and then falls as the range increases.

A software threshold is adjusted such that a digital output will go to logic high when a maze wall is at a desired range. In this application we assume the robot will not be closer than 5 cm to a maze wall. Therefore, we use the portion of the curve beyond the peak where the sensor

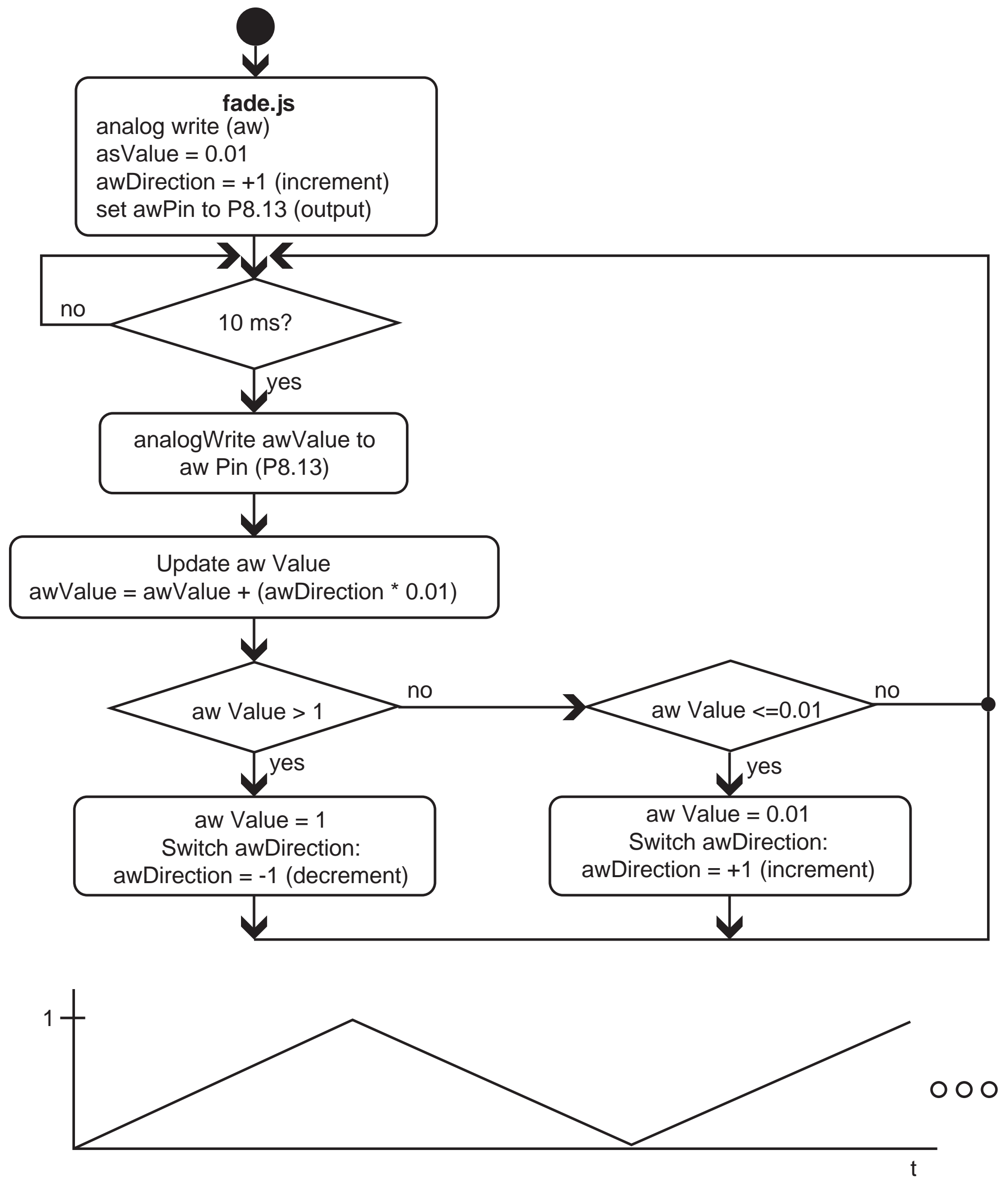


Figure 2.8: UML activity diagram for fade.js.

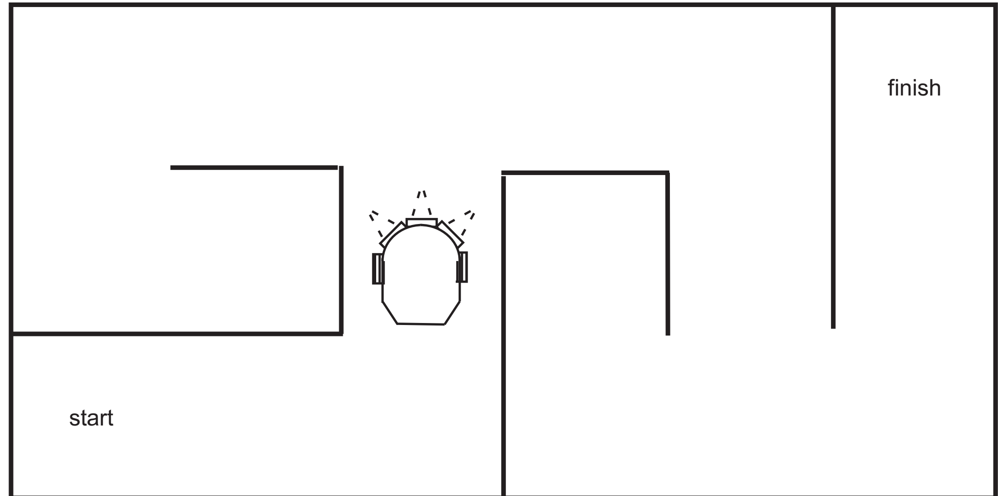


Figure 2.9: Autonomous robot within maze.

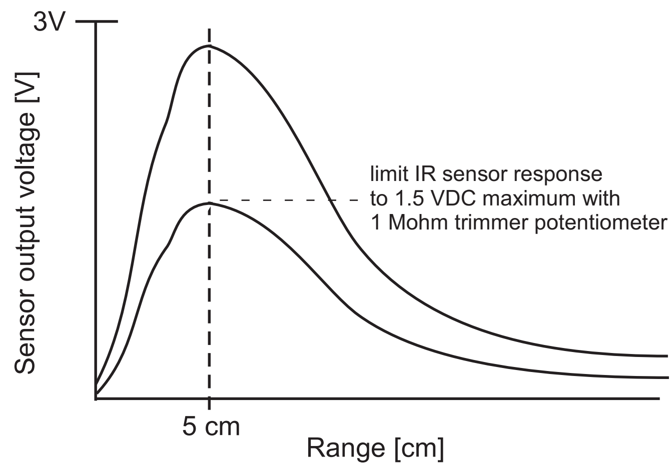


Figure 2.10: Sharp GP2Y0A21YKOF IR sensor profile.

output is inversely proportional to range. That is, the sensor output voltage decreases as the range from the sensor to maze wall increases.

Note: Remember, it is important to note the input to BeagleBone's analog-to-digital converter may not exceed 1.8 VDC. We use a voltage division circuit between the IR sensor and BeagleBone to stay beneath 1.8 VDC. The interface between the IR sensor and the BeagleBone computer is provided in Figure 2.11. The voltage divider network can be formed using two fixed resistors or a small potentiometer. In the figure a 1MΩ trimmer potentiometer is used to set the maximum value of the IR sensor at 1.75 VDC. The LED has an improved interface circuit using a 2N2222 transistor, a 10 Kohm resistor, and a 220 ohm resistor. This interface circuit will be discussed in Chapter 4.

The IR sensor's power (red wire) and ground (black wire) connections are connected to the 5 VDC and ground pins on BeagleBone computer, respectively. The IR sensor's output connection (yellow wire) is connected to an analog input (P9, pin 39) via the trimmer potentiometer on BeagleBone. The LED circuit shown in the top-right corner of the diagram is connected to a digital input/output pin (P8, pin 13) on BeagleBone. We discuss the operation of the LED interface circuit later in Chapter 4.

It is desired to illuminate the LED if the robot is within 10 cm of the maze wall. Let's assume the IR sensor interface circuit provides a voltage of 1.25 VDC when a maze wall is 10 cm from the sensor. The code to illuminate the LED at a 10 cm range is provided below.

```
1  //*****************************************************************
2
3  var b=require('bonescript');
4
5  var ledPin = ``P8_13'';               //digital pin for LED interface
6  var ainPin = ``P9_39'';               //analog input pin for IR sensor
7  var IR_sensor_value;
8
9  b.pinMode(ledPin, b.OUTPUT);          //set pin to digital output
10
11 while(1)
12   {
13                                        //read analog output from IR sensor
14                                        //normalized value ranges from 0..1
15   IR_sensor_value = b.analogRead(ainPin);
16                                        //assumes desired threshold at
17                                        //1.25 VDC with max value of 1.75
                                             VDC
18   if(IR_sensor_value > 0.714)
19     {
20     b.digitalWrite(ledPin, b.HIGH);   //turn LED on
21     }
22   else
23     {
24     b.digitalWrite(ledPin, b.LOW);    //turn LED off
```

```
25          }
26      }
27
28 //***************************************************************
```

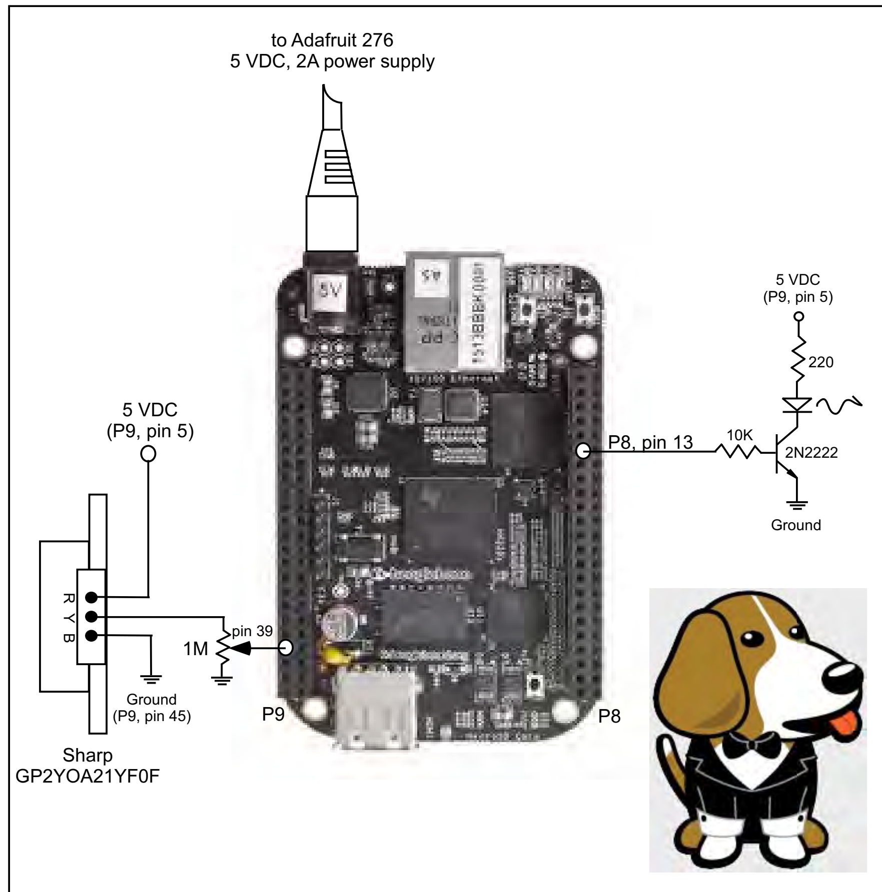


Figure 2.11: IR sensor interface. A transistor circuit is used to interface the BeagleBone Black to the LED. (Illustrations used with permission of Texas Instruments (www.TI.com).)

The program begins by providing names for the two BeagleBone board pins that are used in the program. After providing the names for pins, the next step is to declare the pin the LED circuit will use as output within the "setup" function. In this example, the output from the IR sensor will be converted from an analog to a digital value using the built-in Bonescript "analogRead" function. The "analogRead" function requires the pin for analog conversion variable passed to

it and returns a normalized value from 0 to 1. The value is normalized to a maximum value of 1.8 VDC.

Within the while loop, the present value of the analog value on P9, pin 39 is read. If the reading is above 0.714 (1.25 VDC), the LED on P8, pin 13 is illuminated, or else it is turned off. In the next example, we adapt the IR sensor project to provide custom lighting for an art piece.

2.4 APPLICATION 3: ART PIECE ILLUMINATION SYSTEM

Steven Barrett's (the first author) oldest son Jonathan Barrett (Jonny) is a gifted artist (`www.closertothesuninternational.com`). Although he (Steven Barrett) owns several of Jonny's pieces, his favorite one is a painting he did during his early student days. The assignment was to paint your favorite place. Jonny painted Lac Laronge, Saskatchewan as viewed through the window of a pontoon plane. An image of the painting is provided in Figure 2.12.

Figure 2.12: Lac Laronge, Saskatchewan. Image used with permission, Jonny Barrett, Closer to the Sun Fine Art and Design. (`www.closertothesungallery.com`)

The circuit provided in the previous example may be slightly modified to provide custom lighting for an art piece. The IR sensor is used to detect the presence and position of those viewing the piece. Custom lighting such as a white LED is then activated by BeagleBone via one of the pulse width modulated (PWM) pins (e.g., P9, pin 14), as shown in Figure 2.13. In the Lac Laronge piece, lighting could be provided from behind the painting using high intensity white LEDs. The intensity of the LEDs could be adjusted by changing the value of PWM duty cycle based on the distance the viewer is from the painting. The PWM concept will be discussed in a later chapter. Briefly, the PWM is a digital signal whose duty cycle (percent on time) may be varied to change the average analog value of the signal. The "analogWrite" function generates a 1 kHz pulse width modulated signal at the specified pin. The duty cycle (the percentage of time the 1 kHz signal is a logic high within a period) is set using a value from 0–1. We discuss the PWM concept in greater detail in Chapter 6.

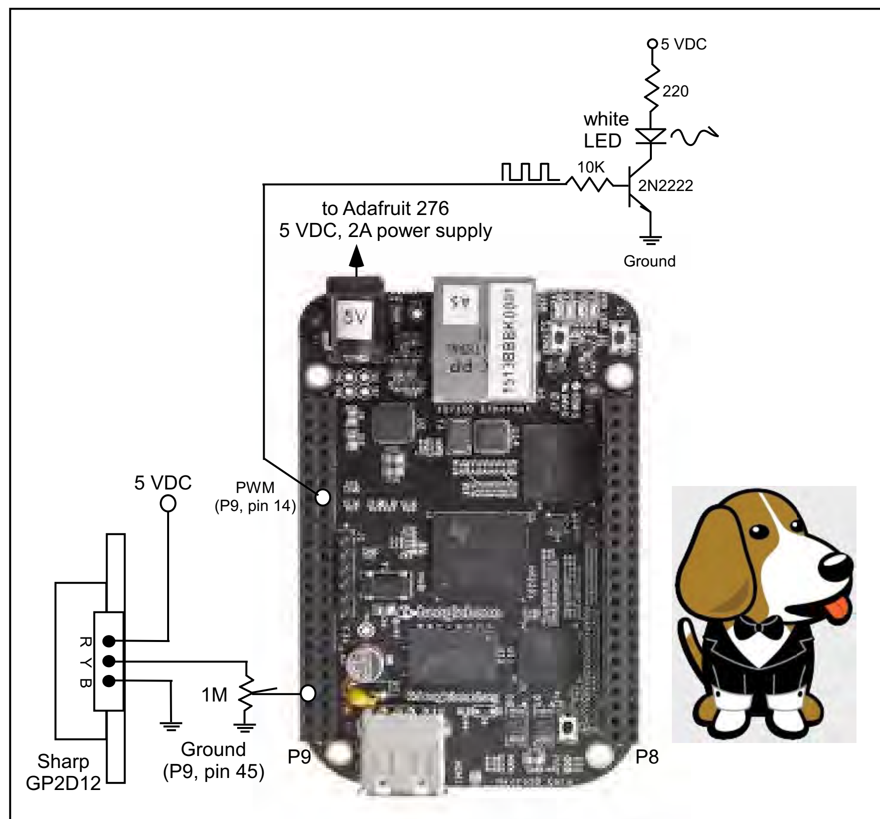


Figure 2.13: IR sensor interface for art illumination. (Illustrations used with permission of Texas Instruments (`www.TI.com`).)

```
1
2 //*****************************************************************
3
4 var b = require('bonescript');
5
6 var ledPin = ''P9_14'';            //PWM pin for LED interface
7 var ainPin = ''P9_39'';            //analog input pin for IR sensor
8 var IR_sensor_value;
9
10 b.pinMode(ledPin, b.OUTPUT);       //set pin to digital output
11
12 while(1)
13   {
14                                     //read analog output from IR sensor
15                                     //normalized value ranges from 0..1
16   IR_sensor_value = b.analogRead(ainPin);
17   b.analogWrite(ledPin, IR_sensor_value);
18   }
19
20 //*****************************************************************
```

2.5 SUMMARY

The chapter provided a tutorial on the Bonescript Development Environment, including the Bonescript library, Cloud9 IDE, and the Node.js JavaScript interpreter, and how it may be used to develop a program for BeagleBone.

2.6 REFERENCES

- Barrett, J. "Closer to the Sun International;" www.closertothesungallery.com.
- "SparkFun Electronics;" www.sparkfun.com.

2.7 CHAPTER EXERCISES

1. Describe what variables are required and returned and the basic function of the following built–in Bonescript functions: Blink, Analog Input.
2. Develop a glossary of Bonescript functions introduced in this chapter.
3. How will you handle the complex profile of the IR sensor?
4. Design a lighting system for an art piece which incorporates three IR sensors and three white LEDs.

5. Develop a rain gage indicator for the BeagleBone using three LEDs and a potentiometer. When the potentiometer is set for one-third full scale a single LED will illuminate, two--thirds full-scale two LEDs will illuminate, and three LEDs for the potentiometer at full scale.

6. Develop a Bonescript program that will illuminate a green LED when a switch is pushed and a red LED when the switch is released.

7. Develop a Bonescript program that will sequentially illuminate 5 LEDs for 1 s each when a switch is depressed.

CHAPTER 3

Programming

Objectives: After reading this chapter, the reader should be able to do the following.

- Provide an overview of the system design process and related programming challenges.
- Describe the key components of a program.
- Specify the size of different variables within the C programming language.
- Define the purpose of the main program.
- Explain the importance of using functions within a program.
- Write functions that pass parameters and return variables.
- Describe the function of a header file.
- Discuss different programming constructs used for program control and decision processing.
- Outline the key features of programming in JavaScript.
- Run BeagleBone "off the leash"— without a host PC connection.
- Apply lessons learned to design and development of an autonomous robot system.

3.1 AN OVERVIEW OF THE DESIGN PROCESS

This chapter provides an introduction to JavaScript and the C languages via comparison and contrast. We begin with a brief review of programming basics. Readers who are familiar with JavaScript and C programming may want to skip ahead although a review of the fundamentals may be helpful.

3.2 OVERVIEW

To the novice, programming a processor may appear mysterious, complicated, overwhelming, and difficult. When faced with a new task, one often does not know where to start. The goal of this section is to provide a tutorial on how to begin programming. We will use a top-down design approach. We begin with the "big picture" of the program followed by an overview of the major

pieces of a program. We then discuss the basics of the JavaScript and C programming languages. Only the most fundamental programming concepts are covered.

Portions of the programming overview were adapted with permission from earlier Morgan and Claypool projects. Throughout the chapter, we provide examples and also provide pointers to a number of excellent references. The chapter concludes with the development of an autonomous maze navigating robot based on the Dagu Magician platform.

3.3 ANATOMY OF A PROGRAM

Programs have a fairly repeatable format. Slight variations exist but many follow the format provided. We first provide a template for a C program followed by a Node.js program template.

C program template:

```
1 //*****************************************************************
2 //Comments containing program information
3 // - file name:
4 // - author:
5 // - revision history:
6 // - compiler setting information:
7 // - hardware connection description to processor pins
8 // - program description
9 //*****************************************************************
10
11 //include files
12 #include <file_name.h>
13
14 //function prototypes
15 //Note: these might be in a separate header file for readability.
16 A list of functions and their format used within the program.
17
18 //program constants
19 #define    TRUE    1
20 #define    FALSE   0
21 #define    ON      1
22 #define    OFF     0
23
24 //interrupt handler definitions
25 //Note: these might be in a separate header file for readability.
26 Used to link the software to hardware interrupt features
27
28 //global variables
29 Listing of variables used throughout the program
30
31 //main program
32
33 void main(void)
34 {
```

```
35
36 body of the main program
37
38 }
39
40 //function definitions
41 A detailed function body and definition for each function
42 used within the program.
43 The functions may also include local variables and parameters. //
      *****************************************************************
```

Node.js program template:

```
 1 //*****************************************************************
 2 //Comments containing program information
 3 // - file name:
 4 // - author:
 5 // - revision history:
 6 // - interpreter and library requirements:
 7 // - hardware connection description to processor pins
 8 // - program description
 9 //*****************************************************************
10
11 //include files
12 var x=require('x');
13
14 //function prototypes not required for JavaScript
15 //because dynamic typing converts variables between types
16
17 //no pre-processor means that constants are the same as variables
18 var TRUE  = 1;
19 var FALSE = 0;
20 var ON    = 1;
21 var OFF   = 0;
22
23 //event handler definitions
24 Used to link the software to asynchronous events
25
26 //global variables
27 Listing of variables used throughout the program
28
29 //main program
30
31 Unlike C, JavaScript statements don't need to be inside of a
32 function and will be executed as they are interpreted
33
34 body of the main program
35
36 //function definitions
```

```
//*****************************************************************************
//C program template
//Comments containing program information
// - file name:
// - author:
// - revision history:
// - compiler setting information
// - hardware connection description to processor pins
// - program description
//*****************************************************************************

//include files
#include<file_name.h>

//function prototypes
A list of functions and their format used within the program

//program constants
#define TRUE     1
#define FALSE    0
#define ON       1
#define OFF      0

//interrupt handler definitions
Used to link the software to hardware interrupt features

//global variables
Listing of variables used throughout the program

//main program

void main (void)
{

body of the main program

}

//function definitions
A detalied function body and definition for each function
used within the program. The functions may also include local variables.
//*****************************************************************************
```

```
//*****************************************************************************
//Node.js program template
Comments containing program information
// - file name:
// - author:
// - revision history:
// - compiler setting information
// - hardware connection description to processor pins
// - program description
//*****************************************************************************

//include files
var x=require('x');

//function prototypes not required for JavaScript
//because dynamic typing converts variables between types

//no pre-processor means that constants are the same as variables
var TRUE = 1;
var FALSE = 0;
var ON   = 1;
var OFF = 0;

//event handler definitions
Used to link the software to asynchronous events

//global variables
Listing of variables used throughout the program

//main program

Unlike C, JavaScript statements don't need to be inside of a
function and will be exectuted as they are interpreted
{

body of the main program

//function definitions
A detalied function body and definition for each function
used within the program. The functions may also include local variables.
//*****************************************************************************
```

Figure 3.1: C and Node.js program templates.

```
37 A detailed function body and definition for each function
38 used within the program.
39 The functions may also include local variables.
40 //*****************************************************************
```

For convenience the C and Node.js program templates are shown side-by-side in Figure 3.1. Let's take a closer look at each piece.

3.3.1 COMMENTS

Comments are used throughout the program to document what, how, and why things were accomplished within a program. The comments help you reconstruct your work at a later time. Imagine that you wrote a program a year ago for a project. You now want to modify that program for a new project. The comments help you remember the key details of the program.

Comments are not compiled into machine code for loading into the processor. Therefore, the comments will not fill up the memory of your processor. Comments are indicated using double

slashes (//). Anything from the double slashes to the end of a line is then considered a comment. A multi-line comment can be constructed using a /∗ at the beginning of the comment and a ∗/ at the end of the comment. These are handy to block out portions of code during troubleshooting or providing multi–line comments. Comment syntax is largely the same between JavaScript and C.

At the beginning of the program, comments may be extensive. Comments may include some of the following information:

- file name;
- program author;
- revision history or a listing of the key changes made to the program;
- instructions on how to compile the program or specific version of interpreter and libraries required;
- hardware connection description to processor pins; and
- program description.

3.3.2 INCLUDE FILES

Often you need to add extra files to your project besides the main program. In JavaScript, this is typically where you actually pull in your library code used to perform advanced tasks, such as Bonescript. In C, include files provide a "personality file" on the specific processor that you are using. This file is provided with the compiler and provides the name of each register used within the processor. It also provides the link between a specific register's name within software and the actual register location within hardware. These files are typically called header files and their name ends with a ".h." You may need to include other header files in your program such as the "math.h" file when programming with advanced math functions.

To include header files within a program, the following syntax is used:

```
//C programming: include files
#include<file_name1.h>   //searches for file in a standard list
#include<file_name2.h>
#include ''file_name3.h''//searches for file in current directory


//Node.js included libraries
var library1 = require('library1');
var library2 = require('library2');
```

In JavaScript, variables, including functions, declared by a library are typically returned from the call to "require." These variables are typically accessed later using variable member operations.

```
//Node.js included libraries
var library1 = require('library1');

//Node.js access library provided variable memberA using one syntax
var memberA = library1.memberA;

//Node.js access library provided variable memberA using another syntax
var alsoMemberA = library1["memberA"];
```

3.3.3 FUNCTIONS

In Chapter 4, we discuss in detail the top down design, bottom up implementation approach to designing processor based systems. In this approach, a processor based project including both hardware and software is partitioned into systems, subsystems, etc. The idea is to take a complex project and break it into doable pieces with a defined action.

We use the same approach when writing computer programs. At the highest level is the main program which calls functions that have a defined action. When a function is called, program control is released from the main program to the function. Once the function is complete, program control reverts back to the main program.

Functions may in turn call other functions as shown in Figure 3.2. This approach results in a collection of functions that may be reused over and over again in various projects. Most importantly, the program is now subdivided into doable pieces, each with a defined action. This makes writing the program easier but also makes it much easier to modify the program since every action is in a known location.

In C, there are four different pieces of code required to properly configure and call the function:

- the function prototype;
- the function call;
- the function parameter or argument; and
- the function body.

In JavaScript, all of the same pieces of code are required except for the function prototype.

Function prototypes are provided early in the program as previously shown in the program template. The function prototype provides the name of the function and any variables or parameters required by the function and any variable returned by the function.

The function prototype follows this format:

```
return_variable  function_name(required_variable1, required_variable2);
```

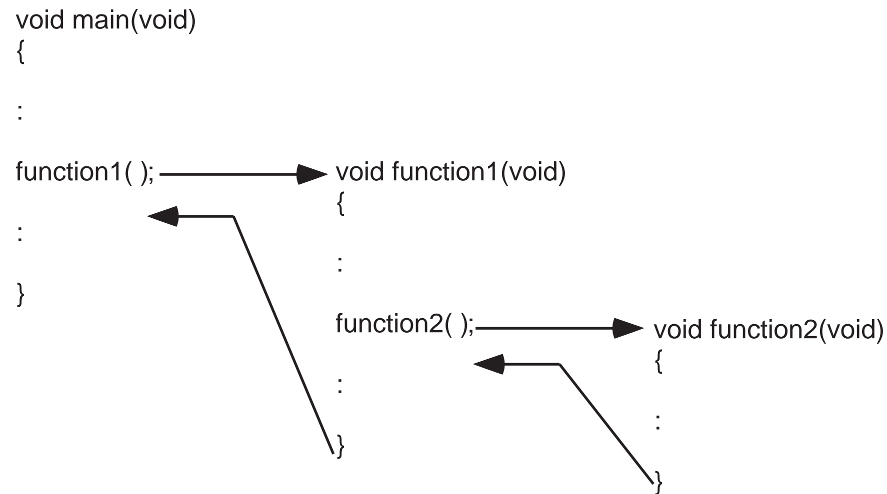


Figure 3.2: Function calling.

If the function does not require variables or does not send back a variable, the word "void" is placed in the variable's position.

The primary purpose of a function prototype is to tell the compiler what types of variables to expect when making or receiving the function call. JavaScript never declares the type of a variable ahead of it being assigned or used. Further, if a JavaScript variable is assigned with one type of variable, say an integer, and then consumed by a function that expects another type of variable, say a string, then JavaScript attempts to automatically convert the variable between the two types. This provides a lot of simplicity to programming as you are learning, but be warned it can get you into trouble as your programs start to get more complex.

The **function call** is the code statement used within a program to execute the function. The function call consists of the function name and the actual arguments required by the function. If the function does not require arguments to be delivered to it for processing, the parenthesis containing the variable list is left empty.

The function call follows this format:

```
function_name(required_variable1, required_variable2);
```

A function that requires no variables follows this format:

```
function_name( );
```

When the function call is executed by the program, program control is transferred to the function, the function is executed, and program control is then returned to the portion of the program that called it.

The syntax for function calls is largely the same between JavaScript and C.

The **function body** is a self-contained "mini-program." In C, the first line of the function body contains the same information as the function prototype: the name of the function, any variables required by the function, and any variable returned by the function. In JavaScript, the return variable type is not provided. Typically, the last line of the function contains a "return" statement. Here a variable may be sent back to the portion of the program that called the function. The processing action of the function is contained within the open ({) and close brackets (}). If the function requires any variables within the confines of the function, they are declared next. These variables are referred to as local variables. A local variable is known only within the scope of a specific function. The local variable is temporarily declared when the function is called and disappears when then function is exited. The actions required by the function follow.

In JavaScript, it is also allowed to have other functions declared as local variables. These functions are special because they utilize the same local "scope" as the other variables declared locally and are not seen directly by outside functions. Because they have access to local variables, it is possible to play some interesting tricks we'll discuss later. It is important to learn about "variable scope" in your development to avoid creating undesired results.

Because JavaScript uses dynamic typing, it is also possible to provide optional arguments to your function and test to see if they are provided within your function body.

The function body in C follows this format:

```
1  return_type  function_name(required_variable1, required_variable2)
2  {
3  //local variables required by the function
4   unsigned int  variable1;
5   unsigned char variable2;
6
7  //program statements required by the function
8
9
10 //return variable
11    return return_variable;
12 }
```

In JavaScript, a function body would look like:

```
1  function_name(required_variable1, required_variable2,
       optional_variable3)
2  {
3  //local variables required by the function
4  var variable1;
5  var variable2;
6
7  //program statements required by the function
8
9
10 //return variable
```

```
11 return return_variable;
12 }
```

3.3.4 INTERRUPT HANDLER DEFINITIONS

Interrupt service routines are functions that are written by the programmer but usually called by a specific hardware event during system operation. In C, interrupt service routines are handled specially by the compiler because they run outside of the context of the rest of your program. It is not possible to declare actual interrupt service routines in JavaScript, but much of the basic functionality is managed with event handlers. We discuss interrupts and how to properly configure them in Chapter 5.

3.3.5 PROGRAM CONSTANTS

In C, the #define statement is used by the compiler pre-processor to associate a constant name with a numerical value in a program. It can be used to define common constants such as pi. It may also be used to give terms used within a program a numerical value. This makes the code easier to read. For example, the following constants may be defined within a program:

```
1 //program constants
2 #define TRUE 1
3 #define FALSE 0
4 #define ON 1
5 #define OFF 0
```

JavaScript does not typically make use of any pre-processor and therefore constants must be treated the same as other variables.

3.3.6 VARIABLES

There are two types of variables used within a program: global variables and local variables. A global variable is available and accessible to all portions of the program, whereas, a local variable is only known and accessible within the context where it is declared.

When declaring a variable in C, the number of bits used to store the operator is also specified. In Figure 3.3, we provide a list of common C variable sizes. The size of other variables such as pointers, shorts, longs, etc. are contained in the compiler documentation.

When programming processors, it is important to know the number of bits used to store the variable and correctly assign values to variables. For example, assigning the contents of an unsigned char variable, which is stored in 8-bits, to an 8-bit output port will have a predictable result. However, assigning an unsigned int variable, which is stored in 16-bits, to an 8-bit output port does not produce predictable results. It is wise to insure your assignment statements are balanced for accurate and predictable results. The modifier "unsigned" indicates all bits will be used to specify the magnitude of the argument. Signed variables will use the left most bit to indicate the polarity ($\pm$) of the argument.

Type	Size (bytes)	Range
unsigned char	1	0..255
signed char	1	-128..127
unsigned int	4*	2^{32}
signed int	4*	2^{32} - (2^{32} - 1)
float	4	+/-(1.18e-38 to 3.40e +38)
double	8*	+/-(2.23e-308 to 1.780e +308)

* compiler dependent

Figure 3.3: Common C variable sizes in bytes.

A variable is declared using the following format. The type of the variable is specified, followed by its name, and an initial value if desired.

```
//global variables
unsigned int  loop_iterations = 6;
```

3.3.7 MAIN FUNCTION

The main function is the hub of activity for the entire program. The main function typically consists of program steps and function calls to initialize the processor followed by program steps to collect data from the environment external to the processor, process the data and make decisions, and provide external control signals back to the environment based on the data collected.

In JavaScript, the main function simply lives in the body of the JavaScript file specified when you invoke the interpreter. For libraries, it is important to avoid having statements outside of variable and function declarations to avoid having those statements invoked when reading in your library.

3.4 FUNDAMENTAL PROGRAMMING CONCEPTS

In the previous section, we covered many fundamental concepts. In this section we discuss operators, programming constructs, and decision-processing constructs to complete our fundamental overview of programming concepts.

3.4.1 OPERATORS

There is a wide variety of operators provided in the JavaScript and C languages. An abbreviated list of common operators are provided in Figures 3.4 and 3.5. The operators have been grouped by general category. The symbol, precedence, and brief description of each operator are provided. The precedence column indicates the priority of the operator in a program statement containing multiple operators. Only the fundamental operators are provided. For more information on this topic, see Barrett and Pack [2005]. JavaScript largely mimics C in terms of operations and precedence.

General Operations

Within the general operations category are brackets, parentheses, and the assignment operator. We have seen in an earlier example how bracket pairs are used to indicate the beginning and end of the main program or a function. They are also used to group statements in programming constructs and decision processing constructs. This is discussed in the next several sections.

The parentheses are used to boost the priority of an operator. For example, in the mathematical expression $7 \times 3 + 10$, the multiplication operation is performed before the addition since it has a higher precedence. Parenthesis may be used to boost the precedence of the addition operation. If we contain the addition operation within parentheses $7 \times (3 + 10)$, the addition will be performed before the multiplication operation and yield a different result than the earlier expression.

The assignment operator (=) is used to assign the argument(s) on the right-hand side of an equation to the left-hand side variable. It is important to insure that the left- and the right-hand side of the equation have the same type of arguments. If not, unpredictable results may occur.

Arithmetic Operations

The arithmetic operations provide for basic math operations using the variables described in the previous section. As described in the previous section, the assignment operator (=) is used to assign the expression on the right-hand side of an equation to the left-hand side variable.

Example: In this example, a function returns the sum of two unsigned int variables which are passed as arguments to the function "sum_two."

```
unsigned int  sum_two(unsigned int variable1, unsigned int variable2)
{
unsigned int  sum;
```

General		
Symbol	Precedence	Description
{ }	1	Brackets, used to group program statements
( )	1	Parenthesis, used to establish precedence
=	12	Assignment

Arithmetic Operations		
Symbol	Precedence	Description
*	3	Multiplication
/	3	Division
+	4	Addition
-	4	Subtraction

<table>
<tr><th colspan="3">Logical Operations</th></tr>
<tr><th>Symbol</th><th>Precedence</th><th>Description</th></tr>
<tr><td><</td><td>6</td><td>Less than</td></tr>
<tr><td><=</td><td>6</td><td>Less than or equal to</td></tr>
<tr><td>></td><td>6</td><td>Greater</td></tr>
<tr><td>>=</td><td>6</td><td>Greater than or equal to</td></tr>
<tr><td>==</td><td>7</td><td>Equal to</td></tr>
<tr><td>!=</td><td>7</td><td>Not equal to</td></tr>
<tr><td>&&</td><td>9</td><td>Logical AND</td></tr>
<tr><td>||</td><td>10</td><td>Logical OR</td></tr>
</table>

Figure 3.4: C operators. (Adapted from Barrett and Pack [2005]).

Bit Manipulation Operations		
Symbol	Precedence	Description
<<	5	Shift left
>>	5	Shift right
&	8	Bitwise AND
^	8	Bitwise exclusive OR
|	8	Bitwise OR

Unary Operations		
Symbol	Precedence	Description
-	2	Unary negative
~	2	One's complement (bit-by-bit inversion)
++	2	Increment
--	2	Decrement
type(argument)	2	Casting operator (data type conversion)

Figure 3.5: C operators (continued). (Adapted from Barrett and Pack [2005]).

```
sum = variable1 + variable2;

return sum;
}
```

Logical Operations

The logical operators provide Boolean logic operations and are useful in comparing two variables. One argument is compared against another using the logical operator provided. The result is returned as a logic value of one (1, true, high) or zero (0 false, low). The logical operators are used extensively in program constructs and decision processing operations to be discussed later.

Bit Manipulation Operations

There are two general types of operations in the bit manipulation category: shifting operations and bitwise operations. Let's examine several examples:

Example: Given the following code segment, what will be the value of variable2 after execution?

```
unsigned char   variable1 = 0x73;
unsigned char   variable2;

variable2 = variable1 << 2;
```

Answer: Variable "variable1" is declared as an eight bit unsigned char and assigned the hexadecimal value of $(73)_{16}$. In binary this is $(0111_0011)_2$. The $<<$ 2 operator provides a left shift of the argument by two places. After two left shifts of $(73)_{16}$, the result is $(cc)_{16}$ and will be assigned to the variable "variable2."

Note that the left- and right-shift operations are equivalent to multiplying and dividing the variable by a power of two.

The bitwise operators perform the desired operation on a bit–by–bit basis. That is, the least significant bit of the first argument is bit–wise operated with the least significant bit of the second argument and so on.

Example: Given the following code segment, what will be the value of variable3 after execution?

```
unsigned char   variable1 = 0x73;
unsigned char   variable2 = 0xfa;
unsigned char   variable3;

variable3 = variable1 & variable2;
```

Answer: Variable "variable1" is declared as an eight bit unsigned char and assigned the hexadecimal value of $(73)_{16}$. In binary, this is $(0111_0011)_2$. Variable "variable2" is declared as an eight bit unsigned char and assigned the hexadecimal value of $(fa)_{16}$. In binary, this is $(1111_1010)_2$. The bitwise AND operator is specified. After execution variable "variable3," declared as an eight bit unsigned char, contains the hexadecimal value of $(72)_{16}$.

Unary Operations

The unary operators, as their name implies, require only a single argument.

For example, in the following code segment, the value of the variable "i" is incremented. This is a shorthand method of executing the operation "$i = i + 1;$"

```
unsigned int    i = 0;

i++;
```

Example: It is not uncommon in embedded system design projects to have every pin on a processor employed. Furthermore, it is not uncommon to have multiple inputs and outputs assigned to the same port but on different port input/output pins. Some compilers support specific pin reference. Another technique that is not compiler specific is **bit twiddling**. Figure 3.6 provides bit twiddling examples on how individual bits may be manipulated without affecting other bits using bitwise and unary operators [ImageCraft]. Examples are provided for an eight bit register (Reg_A). This concept may be easily extended to registers of other sizes (e.g., 32 bit registers).

Syntax	Description	Example	
a | b	bitwise or	Reg_A |= 0x80;	// turn on bit 7 (msb)
a & b	bitwise and	if ((Reg_A & 0x81) == 0)	// check if both bit 7 and bit 0 are zero
a ^b	bitwise exclusive or	Reg_A ^= 0x80;	// flip bit 7
~a	bitwise complement	Reg_A &= ~0x80;	// turn off bit 7

Figure 3.6: Bit twiddling [ImageCraft].

3.4.2 PROGRAMMING CONSTRUCTS

In this section, we discuss several methods of looping through a set of statements. We examine the "for" and the "while" looping constructs.

The **for** loop provides a mechanism for looping through the same portion of code a fixed number of times or while a certain condition is present. The for loop consists of three main parts:

- initialize variables such as the loop counter;
- loop termination testing; and
- update the loop counter, e.g., increment the loop counter.

In the following code fragment, the for loop is executed 10 times.

```
//In C
unsigned int  loop_ctr;

for(loop_ctr = 0; loop_ctr < 10; loop_ctr++)
  {
                      //loop body
```

```
  }

//In Javascript
var loop_ctr;

for(loop_ctr = 0; loop_ctr < 10; loop_ctr++)
  {
                          //loop body

  }
```

The for loop begins with the variable "loop_ctr" equal to 0. During the first pass through the loop, the variable retains this value. During the next pass, the variable "loop_ctr" is incremented by one. This action continues until the "loop_ctr" variable reaches the value of 10. Since the argument to continue the loop is no longer true, program execution continues beyond the closing braces of the for loop.

In the previous example, the for loop counter was incremented at the beginning of each loop pass. The "loop_ctr" variable can be updated by any amount. For example, in the following code fragment the "loop_ctr" variable is increased by three for every pass of the loop.

```
//In C
unsigned int  loop_ctr;

for(loop_ctr = 0; loop_ctr < 10; loop_ctr=loop_ctr+3)
  {
                          //loop body

  }

//In Javascript
var loop_ctr;

for(loop_ctr = 0; loop_ctr < 10; loop_ctr=loop_ctr+3)
  {
                          //loop body
  }
```

The "loop_ctr" variable may also be initialized at a high value and then decremented at the beginning of each pass of the loop.

```
//In C
unsigned int  loop_ctr;
```

```
for(loop_ctr = 10; loop_ctr > 0; loop_ctr--)
  {
                          //loop body

  }

//In Javascript
var loop_ctr;

for(loop_ctr = 10; loop_ctr > 0; loop_ctr--)
  {
                          //loop body

  }
```

As before, the "loop_ctr" variable may be decreased by any numerical value as appropriate for the application at hand.

In JavaScript, it is also possible to loop over the indexes of an array or keys of an object as shown in the example below.

```
//In Javascript
var index;
var myarray = [1, 2, 3];

for(index in myarray)
  {
     myarray[index]++;   //loop body
  }
```

The value in "index" can be used to index into the array or object. Note that it doesn't contain the indexed value itself, but the value used to perform the index.

The **while** loop is another programming construct that allows multiple passes through a portion of code. The while loop will continue to execute the statements within the open and close braces as long as the condition accompanying the while statement remains logically true. The code snapshot below will implement a ten iteration loop. Note how the "loop_ctr" variable is initialized outside of the loop and incremented within the body of the loop. As before, the variable may be initialized to a greater value and then decremented within the loop body.

```
//In C
unsigned int  loop_ctr;

loop_ctr = 0;
```

```
while(loop_ctr < 10)
  {
                          //loop body
  loop_ctr++;
  }

//In Javascript
var loop_ctr;

loop_ctr = 0;
while(loop_ctr < 10)
  {
                          //loop body
  loop_ctr++;
  }
```

Frequently, within a processor application, the program begins with system initialization actions. Once initialization activities are complete, the processor enters a continuous loop. This may be accomplished using the following code fragment.

```
while(1)
  {



  }
```

In JavaScript, this is sort of synchronous programming is typically avoided and often replaced with use of the JavaScript timers. For example, it is possible to use a JavaScript timer to call a function once every second.

```
var mytimer = setInterval(myfunction, 1000);
```

Here "myfunction" is passed to setInterval as the first argument and the second argument is the number of milliseconds between when the timer goes off to call the function. "myfunction" will be called without any arguments provided to it. "mytimer" contains a reference to the timer that can be used to disable the timer using clearInterval. We'll discuss JavaScript timers in a bit more detail later in this chapter.

3.4.3 DECISION PROCESSING

There are a variety of constructs that allow decision making. These include the following:

- the **if** statement;

- the **if–else** construct;
- the **if–else if–else** construct, and the
- **switch** statement.

The **if** statement will execute the code between an open and close bracket set should the condition within the if statement be logically true.

Example: We use autonomous, maze navigating robots several times throughout the book as electronic systems examples. An autonomous, maze navigating robot is equipped with sensors to detect the presence of maze walls and navigate about the maze. The robot has no prior knowledge about the maze configuration. It uses the sensors and an onboard algorithm to determine the robot's next move. The overall goal is to navigate from the starting point of the maze to the end point as quickly as possible without bumping into maze walls, as shown in Figure 3.7. Maze walls are usually painted white to provide a good, light reflective surface, whereas the maze floor is painted matte black to minimize light reflections.

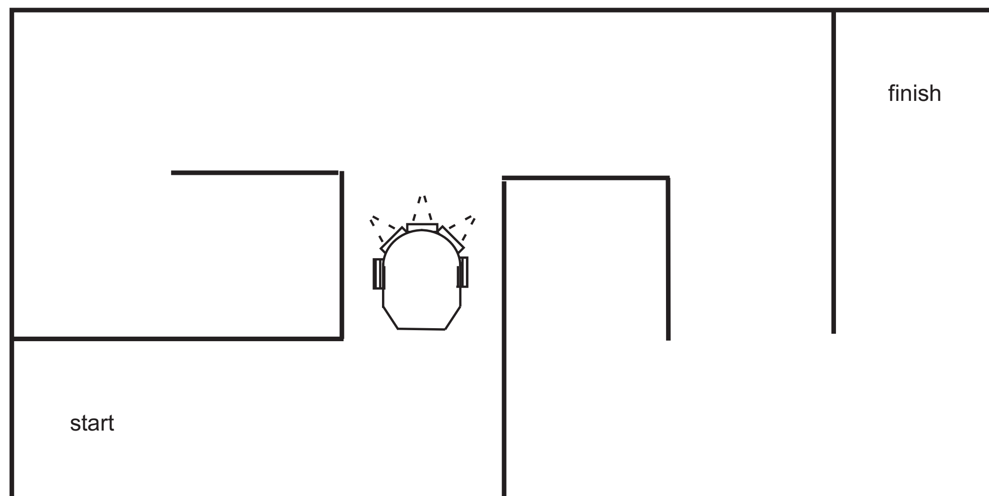


Figure 3.7: Autonomous robot within maze.

In several examples we use the Dagu Magician robot. We refer to it as the "Magician." Suppose our goal is to have the Magician find a path through a maze as shown in Figure 2.9. The Magician is equipped with two wheels driven by DC motors. When moving through the maze, the Magician must safeguard itself from bumping against the walls. For this purpose, we equip the Magician with three IR sensors. The IR sensors provide a voltage output which depends on the distance of the sensor from the reflecting surface.

Later in the chapter we develop the algorithm to allow the Magician to navigate the maze. To help develop the algorithm, a light emitting diode (LED) is connected to a digital input/output pin on BeagleBone. The robot's center infrared (IR) sensor is connected to an analog-to-digital converter (ADC) input pin on BeagleBone. The IR sensor provides a voltage output that is inversely proportional to distance of the sensor from the maze wall. It is desired to illuminate the LED if the robot is within 10 cm of the maze wall. The sensor's output is too large to be directly applied to BeagleBone. We place a voltage divided network between the sensor and BeagleBone to remedy this situation. With the voltage divider network, the sensor provides an output voltage of 1.25 VDC at the 10 cm range. The Bonescript library's analogRead function provides a normalized analog voltage reading from 0–1. A reading of 1 corresponds to the maximum ADC system voltage of 1.8 VDC. Therefore, the ADC will report a reading of 0.694 ($1.25\ VDC/1.80\ VDC$) when the maze wall is 10 cm from the robot.

The following **if** statement construct will implement this LED indicator. We use pseudocode to illustrate the concept. We provide the actual code to do this later in the chapter.

```
if (center_sensor_output > 0.694) //Center IR sensor voltage
                                  //greater than 1.25 VDC
   {
   led_pin = LOGIC_HIGH;         //illuminate LED connected
                                 //to led_pin
   }
```

In the example provided, there is no method to turn off the LED once it is turned on. This will require the **else** portion of the construct as shown in the next code fragment.

```
if (center_sensor_output > 0.694) //Center IR sensor voltage
                                  //greater than 1.25 VDC
   {
   led_pin = LOGIC_HIGH;          //illuminate LED connected
                                  //to led_pin
   }
else
   {
   led_pin = LOGIC_LOW;           //turn LED off
   }
```

The **if-else if-else** construct may be used to implement a three LED system. In this example, the left, center, and right IR sensors are connected to three different analog-to-digital converter pins on BeagleBone. Also, three different LEDs are connected to digital output pins on BeagleBone. The following pseudocode fragment implements this LED system.

```
if (left_sensor_output > 0.694)       //Left IR sensor voltage
                                      //greater than 1.25 VDC
   {
   left_led_pin = LOGIC_HIGH;         //illuminate LED connected
                                      //to left_led_pin
   }

else if (center_sensor_output > 0.694) //Center IR sensor voltage
                                      //greater than 1.25 VDC
   {
   center_led_pin = LOGIC_HIGH;       //illuminate LED connected
                                      //to center_led_pin
   }

else if (right_sensor_output > 0.694)  //Right IR sensor voltage
                                      //greater than 1.25 VDC
   {
   right_led_pin = LOGIC_HIGH;        //illuminate LED connected
                                      //to right_led_pin
   }



else
   {
   left_led_pin = LOGIC_LOW;           //turn LEDs off
   center_led_pin = LOGIC_LOW;
   right_led_pin = LOGIC_LOW;
   }
```

The **switch** statement is used when multiple if-else conditions exist. Each possible condition is specified by a case statement. When a match is found between the switch variable and a specific case entry, the statements associated with the case are executed until a **break** statement is encountered. In general, the break terminates the execution of the nearest enclosing do, switch or while statement and program control passes to the next program statement following the break. In the switch statement, this ensures only a single case of the switch statement is executed.

Example: Suppose an eight bit variable "robot_status" is periodically updated to reflect the current status of the robot (e.g., low battery power, maze walls in the robot's path, etc.). Each bit in the register represents a different robot status item. In response to a change the status the robot must complete status related items. A switch statement may be used to process the multiple possible actions in an orderly manner.

```
//*********************************************************************
if(new_robot_status != old_robot_status)   //check for status change
  {
  switch(new_robot_status)
    {                                       //process change in status
    case 0x01:                              //new_robot_status bit 0
      :                                     //related actions
    break;


    case 0x02:                               //new_robot_status bit 1
       :                                     //related actions
    break;


    case 0x04:                                //new_robot_status bit 2
       :                                      //related actions
    break;


    case 0x08:                                //new_robot_status bit 3
       :                                      //related actions
     break;


     case 0x10:                               //new_robot_status bit 4
       :                                      //related actions
     break;


     case 0x20:                               //new_robot_status bit 5
       :                                      //related actions
     break;
```

```
    case 0x40:                                //new_robot_status bit 6
       :                                      //related actions
    break;

    case 0x80:                                //new_robot_status bit 7
       :                                      //related actions
    break;

    default:;                                 //all other cases
    }                                         //end switch
  }                                           //end if new_robot_status
old_robot_status = new_robot_status;          //update old_robot_status
//*******************************************************************
```

That completes our brief overview of the JavaScript and C programming languages. In the next section, we provide a bit more detail on the Bonescript library and the environment in which we will be using it. You will see how this development environment provides a user-friendly method of quickly developing code applications for BeagleBone. But first, let's discuss some of the basics of programming using a JavaScript interpreter.

3.5 PROGRAMMING IN JAVASCRIPT USING NODE.JS

In this section we provide an admittedly brief introduction to Node.js. However, we provide pointers to several excellent sources on these topics at the end of the chapter. As you will recall, Bonescript is the user-friendly "website" interface to write application programs for BeagleBone. Since JavaScript was specifically developed to quickly implement websites, it was a natural choice to implement Bonescript. JavaScript is an interpreted language. This means the code is not compiled. Instead a JavaScript program is a script or listing of pre-built functions to quickly implement dynamic user interactions within a website.

Node.js is an implementation of a JavaScript interpreter running on the web host, rather than within your web browser. It was developed to implement event driven programming techniques. In our discussion of Node.js we use a restaurant example to gain a general understanding of event driven programming. Having a fundamental understanding of JavaScript and Node.js will allow you to extend the features and capabilities of Bonescript. It is important to emphasize that Bonescript is an open source library. We are counting on the user community to expand the features of Bonescript. If there is a feature you need, please develop it and share it with the BeagleBoard.org community.

3.5.1 JAVASCRIPT

As previously illustrated, JavaScript is very similar in syntax to the C programming language. It consists of two basic parts: the syntax and the object model. The syntax can be divided into six basic areas:

- comments
- conditionals
- loops
- operators
- functions
- variables

The first four items are virtually identical to the C programming language previously discussed. We discussed function writing and the declaration of variables in the section on Bonescript. It should be noted that strong typed variables are available in JavaScript 2.0 [Vander Veer, 2005; Pollock, 2010; Kiessling, 2012; Hughes-Crocher and Wilson, 2012].

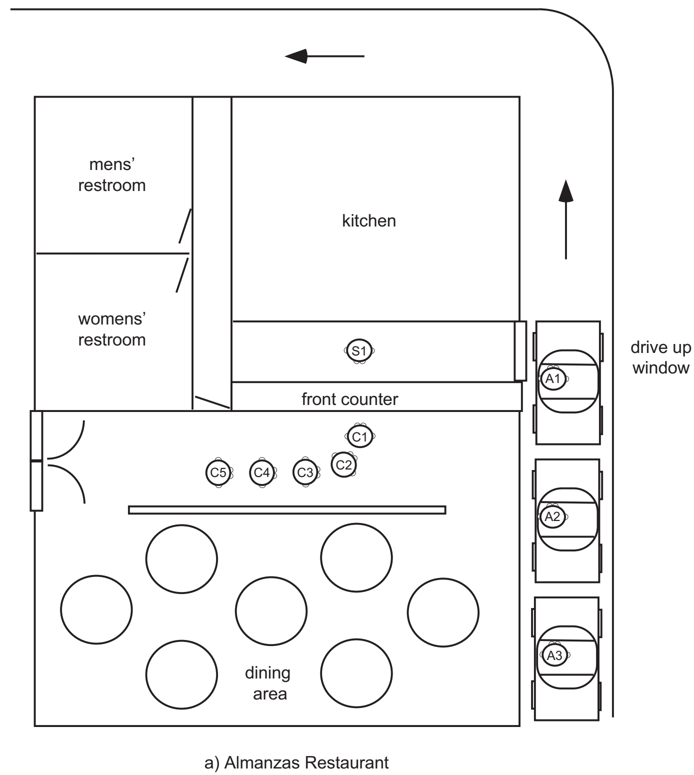
The object model for JavaScript within web browsers is the document object module (DOM). The components of a webpage are referred to as self-contained or encapsulated modules. Module encapsulation includes the object's data, properties, methods and event handlers. Predefined modules are available in HTML, web browsers and JavaScript. Also, custom modules may be implemented using JavaScript [Vander Veer, 2005].

3.5.2 EVENT-DRIVEN PROGRAMMING

Node.js uses the same syntax as other JavaScript interpreters, but provides a different object model for running outside of a browser environment. It was developed to execute event-driven programming. Before discussing event-driven programming, let's take an aside and discuss breakfast burritos!

Aside. Steven Barrett is an aficionado of breakfast burritos! The best I've had was from a small family-owned restaurant in Colorado Springs, Colorado, called Rita's. I left this beloved city when I retired from military service and moved to Laramie, Wyoming. Laramie is a wonderful place to live and has a wonderful restaurant named Almanza's that serves awesome breakfast burritos. I usually go on Saturdays for this breakfast treat. Provided in Figure 3.8 is a floor plan of the Almanza's restaurant.

On Saturday mornings there is a single server working at Alamanza's. The server is designated at "S1" in the diagram. The server is very efficient and is able to keep up with the steady stream of customers (C1, C2, . . .) that come to the front counter to place their orders and those at the drive-up window (A1, A2, . . .). I really admire those that serve in the restaurant industry. I attempted this earlier in my life and was horrible at it.



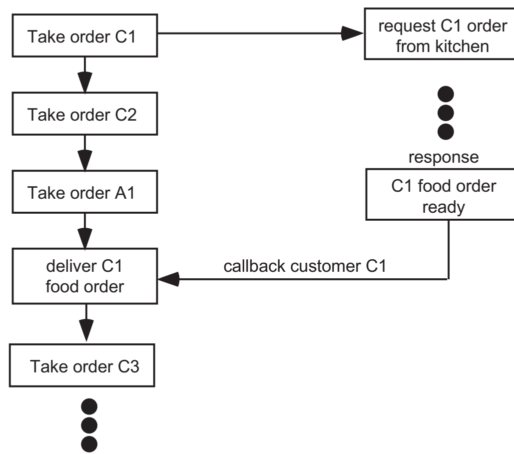


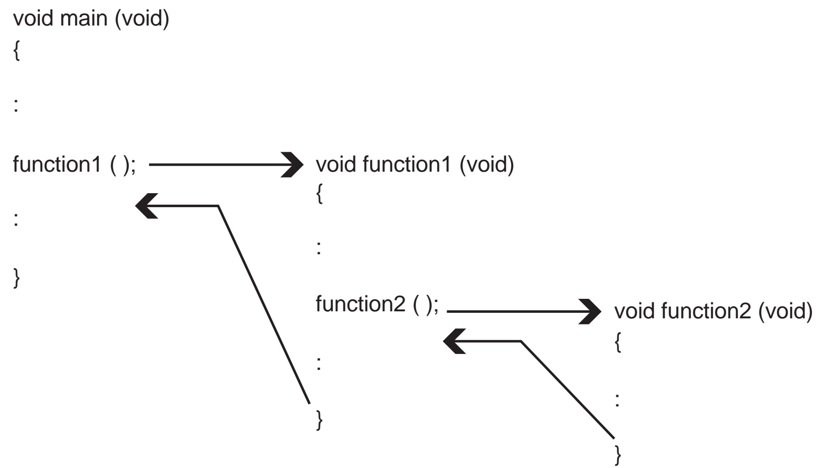
Figure 3.8: Alamanza's restaurant.

Customers would be lost if the server would take the order from counter customer C1 and not serve any other customers until C1's order was complete and delivered. Instead, the server quickly and efficiently takes customer C1's order and passes the order back to the kitchen. Customer C1 is then asked to step aside and wait for their order to be completed. In the meantime the server (S1) takes orders from counter customer C2 and drive up window customer A1. The kitchen then notifies the server S1 that customer C1 order is ready. The server then delivers the food to customer C1.

3.5.3 NODE.JS

The Node.js object model is specifically focused on event driven programming. In contrast, procedural programming is illustrated in Figure 3.9a. In procedural programming a program executes in an orderly fashion as prescribed by the program steps. In event driven programming, the program processes through an event loop as shown in Figure 3.9b. When an asynchronous event occurs, a **function** is sent to service the event request. In response to the request, event related tasks are performed. If some time is required to complete the event related tasks, the system does not wait for the task to be completed. Instead, the tasks are initiated. When the tasks are complete, the function sent to the activity, the response function termed the callback, is executed. This allows for the efficient processing of multiple activities. The literature describes this as "everything runs in parallel, except your code." That is, the processor is sequentially executing operations. However, event level techniques allow efficient execution of multiple events [Vander Veer, 2005; Pollock, 2010; Kiessling, 2012; Hughes-Crocher and Wilson, 2012]. If this sounds familiar, this is because it is exactly what the server does at Almanza's restaurant. Provided below is an example to better illustrate event driven programming using the Almanza's scenario.

```
1 //*******************************************************************
2 // almanzas.js
3 //*******************************************************************
4
5 var counter = 1;
6 var driveup = 1;
7
8 setTimeout(counterOrder, 1000);//Counter customer at second 1
9 setTimeout(counterOrder, 2000);//Counter customer at second 2
10 setTimeout(driveupOrder, 3000);//Driveup customer at second 3
11 setTimeout(counterOrder, 4000);//Counter customer at second 4
12 setTimeout(counterOrder, 5000);//Counter customer at second 5
13 setTimeout(driveupOrder, 6000);//Driveup customer at second 6
14 setTimeout(driveupOrder, 7000);//Driveup customer at second 7
15 setTimeout(counterOrder, 8000);//Counter customer at second 8
16
17 //*******************************************************************
18 //server(task)
19 //
20 //The server takes the orders from the customers and gives them
```



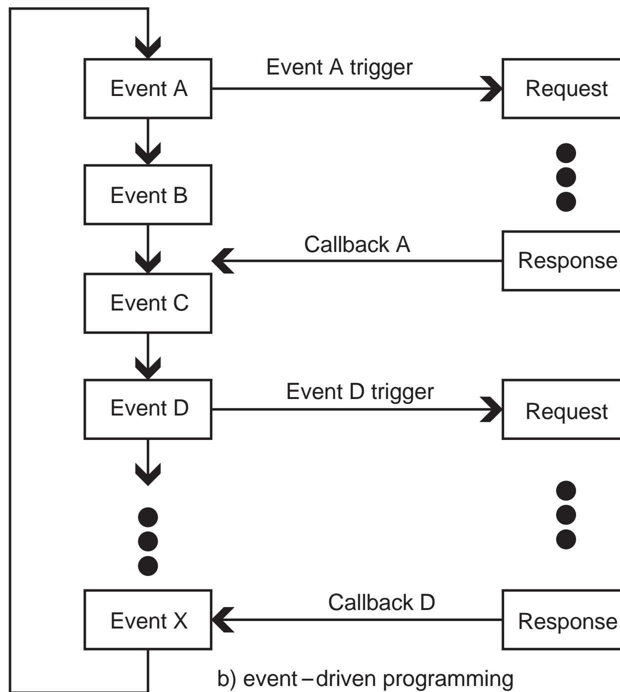


Figure 3.9: Event-driven programming.

```
21 //to the kitchen, as well as delivering orders from the kitchen
22 //*******************************************************************
23
24 function server(customer)
25 {
26     console.log("Take order " + customer);
27     kitchen(deliver);
28
29     function deliver()
30     {
31         console.log("Deliver " + customer + " food order");
32     }
33 }
34
35 //*******************************************************************
36 //kitchen(callback)
37 //
38 //The kitchen takes tasks and completes them at a random time
39 //between 1 and 5 seconds
40 //*******************************************************************
41
42 function kitchen(callback)
43 {
44     var delay = 1000 + Math.round(Math.random() * 4000);
45     setTimeout(callback, delay);
46 }
47
48 //*******************************************************************
49 //counterOrder()
50 //
51 //Give order to server
52 //*******************************************************************
53
54 function counterOrder()
55 {
56     server("C" + counter);
57     counter++;
58 }
59
60 //*******************************************************************
61 //driveupOrder()
62 //
63 //Give order to server
64 //*******************************************************************
65
66 function driveupOrder()
67 {
68     server("A" + driveup);
```

```
69      driveup++;
70 }
71 //**************************************************************
```

Now that you've seen the way JavaScript handles asynchronous tasks, what if you want to run a series of tasks sequentially that have long delays between them without blocking all of your event handlers? Unlike many other programming environments, Node.JS highly discourages using operating system threads that would allow other tasks to run while one thread is blocked. In Node.JS, if we decided to have a function pause and watch the clock, no other function can run until that function returns. Performing sequential tasks is made a bit complicated by this, but here's a pattern that you can use. The series of tasks provided are executed sequentially with delays between them and other event handlers are able to run during those delays.

```
1 //**************************************************************
2 // next.js - A pattern for calling sequential functions in node.js
3 //**************************************************************
4
5 mysteps(done);
6
7 //**************************************************************
8 //done
9 //**************************************************************
10
11 function done() {
12   console.log("done");
13 }
14
15 //**************************************************************
16 //mysteps
17 //**************************************************************
18
19 function mysteps(callback)
20 {
21 //Provide a list of functions to call
22 var steps =
23 [
24 function(){ console.log("i = " + i); setTimeout(next, 15); }, //delay
     15 ms
25 function(){ console.log("i = " + i); setTimeout(next, 5);  }, //delay 5
     ms
26 function(){ callback(); }
27 ];
28
29 //Start at 0
30 var i = 0;
31
32 console.log("i = " + i);
33 next(); //Call the first function
```

```
34
35 //Nested helper function to call the next function in 'steps'
36 function next()
37   {
38   i++
39   steps[i-1]();
40   }
41 }
42
43 //*******************************************************************
```

3.6 APPLICATION: DAGU MAGICIAN AUTONOMOUS MAZE NAVIGATING ROBOT

In the next several chapters we investigate different autonomous navigating robot designs. Before delving into these designs, it would be helpful to review the fundamentals of robot steering and motor control. Figure 3.10 illustrates the fundamental concepts. Robot steering is dependent upon the number of powered wheels and whether the wheels are equipped with unidirectional or bidirectional control. Additional robot steering configurations are possible. An H-bridge is typically required for bidirectional control of a DC motor. We discuss the H-bridge in greater detail in the next chapter.

3.6.1 DAGU MAGICIAN ROBOT

In this application project we equip the Dagu Magician robot for control by BeagleBone as a maze navigating robot. Reference Figure 3.11. The Magician kit may be purchased from SparkFun Electronics (`www.sparkfun.com`). The robot is controlled by two 7.2 VDC motors which independently drive a left and right wheel. A third non-powered drag ball provides tripod stability for the robot.

We equip the Dagu Magician robot platform with three Sharp GP2Y0A21YKOF IR sensors as shown in Figure 3.12. The sensors are available from SparkFun Electronics (`www.sparkfun.com`). We mount the sensors on a bracket constructed from thin aluminum. Dimensions for the bracket are provided in the figure. Alternatively, the IR sensors may be mounted to the robot platform using "L" brackets available from a local hardware store. The characteristics of the sensor were provided earlier in Figure 2.10. The robot is placed in a maze with reflective walls. The project goal is for the robot to detect wall placement and navigate through the maze. It is important to note the robot is not provided any information about the maze. The control algorithm for the robot is hosted on BeagleBone.

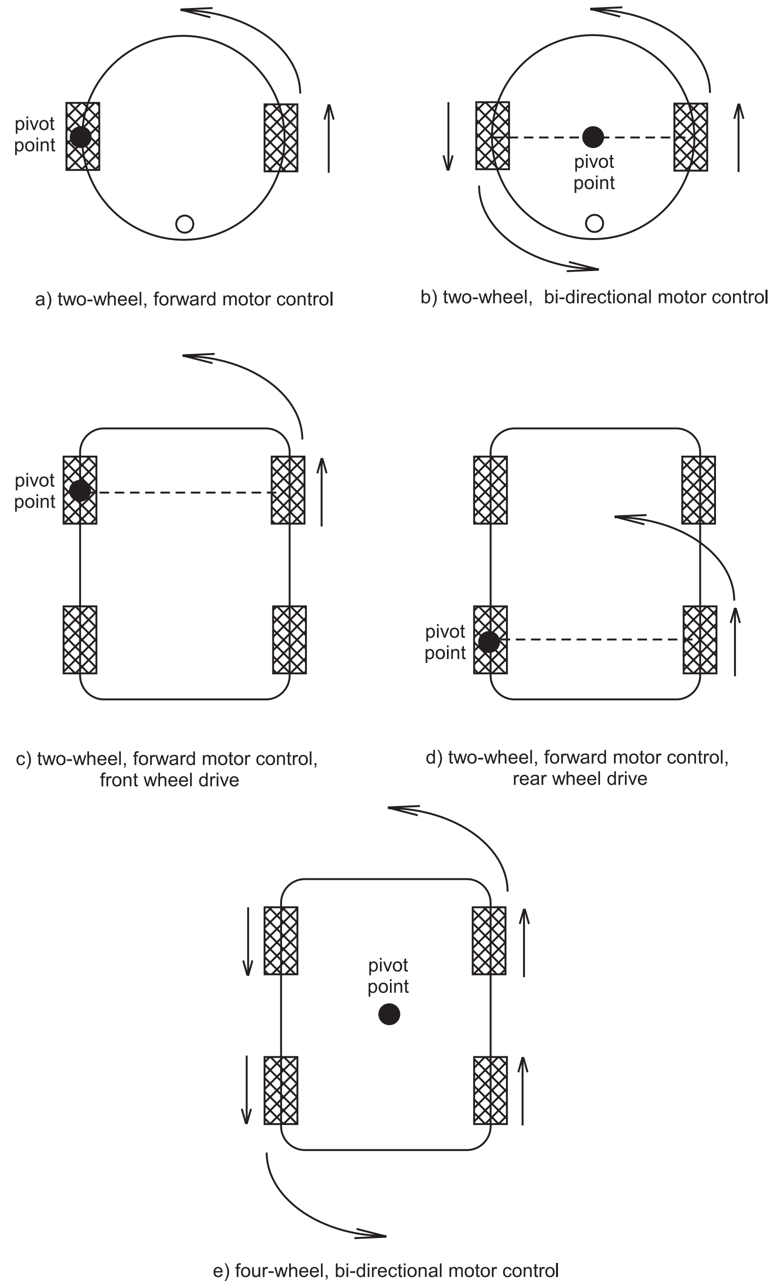


Figure 3.10: Robot control configurations.

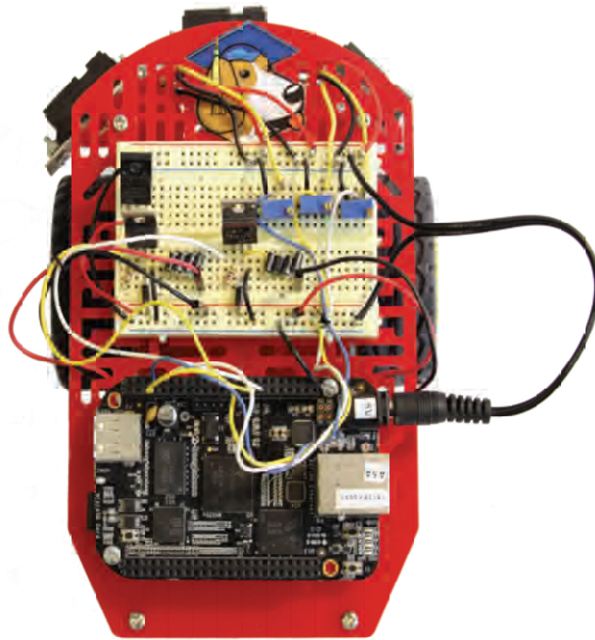


Figure 3.11: Dagu Magician robot. (Photo courtesy of Barrett [2015].)

3.6.2 REQUIREMENTS

The requirements for this project are simple, the robot must autonomously navigate through the maze as quickly as possible without touching maze walls. The requirements for this project are purposely stated flexibly to allow for creativity and competition in developing a final design.

3.6.3 CIRCUIT DIAGRAM

The circuit diagram for the robot is provided in Figure 3.14. The three IR sensors (left, center, and right) are mounted on the leading edge of the robot to detect maze walls. Recall the maximum allowable voltage that may be presented to the BeagleBone Black analog-to-digital system is 1.8 VDC. The output from each IR sensor is provided to a 1M Ohm potentiometer set to one-half full scale. This insures the input voltage to the ADC channels do not exceed the 1.8 VDC limit. The output from the potentiometers is fed to three separate ADC channels: left - P9, pin 39, center - P9, pin 40, and right - P9, pin 37.

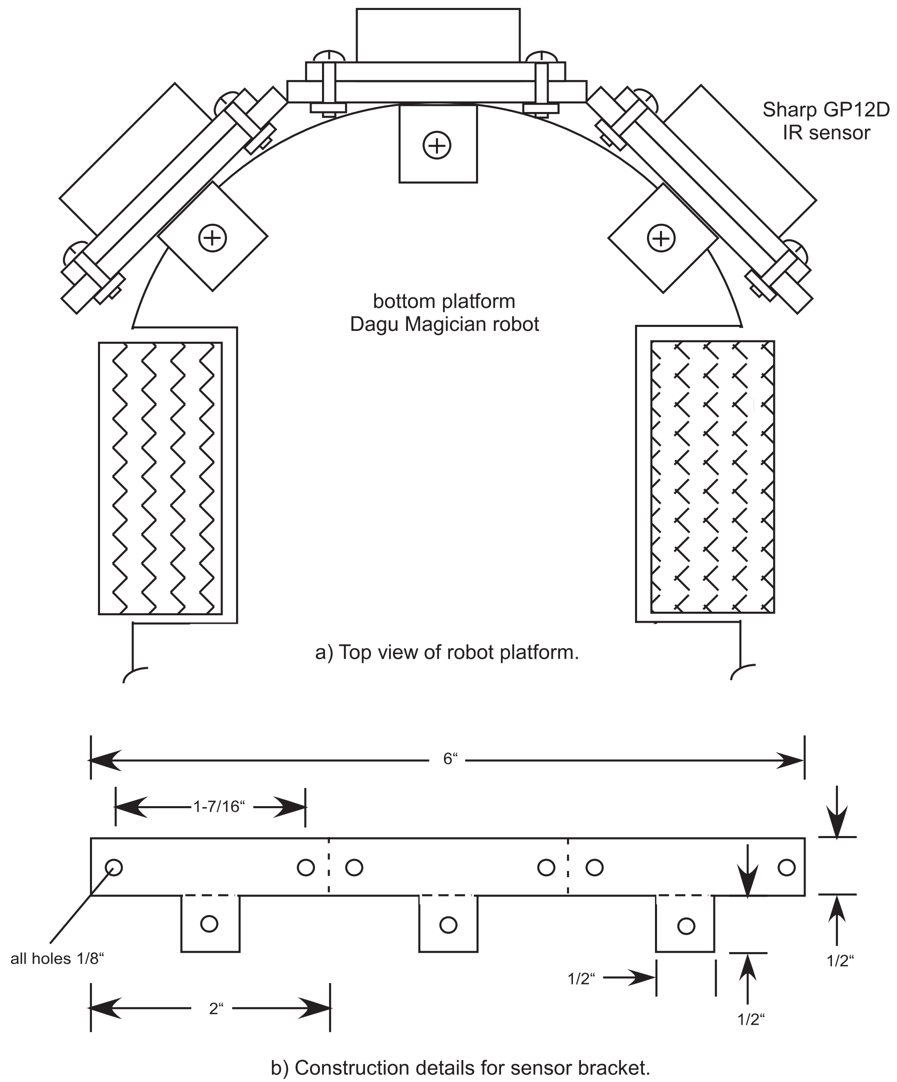


Figure 3.12: Dagu Magician robot platform modified with three IR sensors.

The robot motors are driven by PWM channels: left motor - P9, pin 14 and right motor - P9, pin 16. The PWM channels are interfaced to the to the motors via a Darlington transistor (TIP 130, NPN) with enough drive capability to handle the maximum current requirement of the motor. The Darlington transistor configuration provides high current gain. The PWM output from BeagleBone Black is fed to the base of the TIP130 via a 330 ohm resistor. The motor is connected to the collector of the TIP130 transistor. It is connected in series with three 1N4001 diodes to step down the supply voltage of 9 VDC to the motor voltage. A reversed biased diode is provided across the motor and the diode string to eliminate flyback. When the voltage to an inductive load (motor, solenoid, etc.) is removed, a flyback spike voltage may occur. The reversed biased diode allows energy in the inductor to safely dissipate [O'Berto, 2015]. A prototype diagram of the circuit is provided in Figure 3.13.

The robot requires a 9 VDC power supply. This may be provided by an onboard battery pack (6 AA batteries) or a 9 VDC regulated supply provided by an external power supply. The 9 VDC source is fed to a 5 VDC voltage regulator to power BeagleBone. The 9 VDC power supply should be rated at several amps (e.g., Adafruit 276 or Jameco #1952847). The supply may be connected to the robot via a flexible umbilical cable. A 2.1-mm center positive with a 5.5-mm outer barrel plug may be used to connect the 5 VDC power supply output to the BeagleBone Black power jack. The center positive connection may be verified with a voltmeter.

3.6.4 STRUCTURE CHART

The structure chart for the robot project is provided in Figure 3.15. The structure chart shows the hierarchy of how system hardware and software components will interact and interface with one another. It will be discussed in some detail in Chapter 5.

3.6.5 UML ACTIVITY DIAGRAMS

The UML (Unified Modeling Language) activity diagram for the robot is provided in Figure 3.16. The activity diagram is simply a UML compliant flow chart. UML is a standardized method of documenting systems. The activity diagram will be discussed in more detail in Chapter 5.

3.6.6 BONESCRIPT CODE

Provided below is the basic framework for the Bonescript code. As illustrated in the Robot UML activity diagram, the control algorithm initializes pins, senses wall locations, and issues motor control signals to avoid walls.

It is helpful to characterize the infrared sensor response to the maze walls. This allows a threshold to be determined indicating the presence of a wall. In this example we assume that a threshold of 1.25 VDC has been experimentally determined and the potentiometer has been set for a maximum possible IR senor reading of 1.75 VDC. This equates to threshold reading from the "b.analogRead function of 0.714."

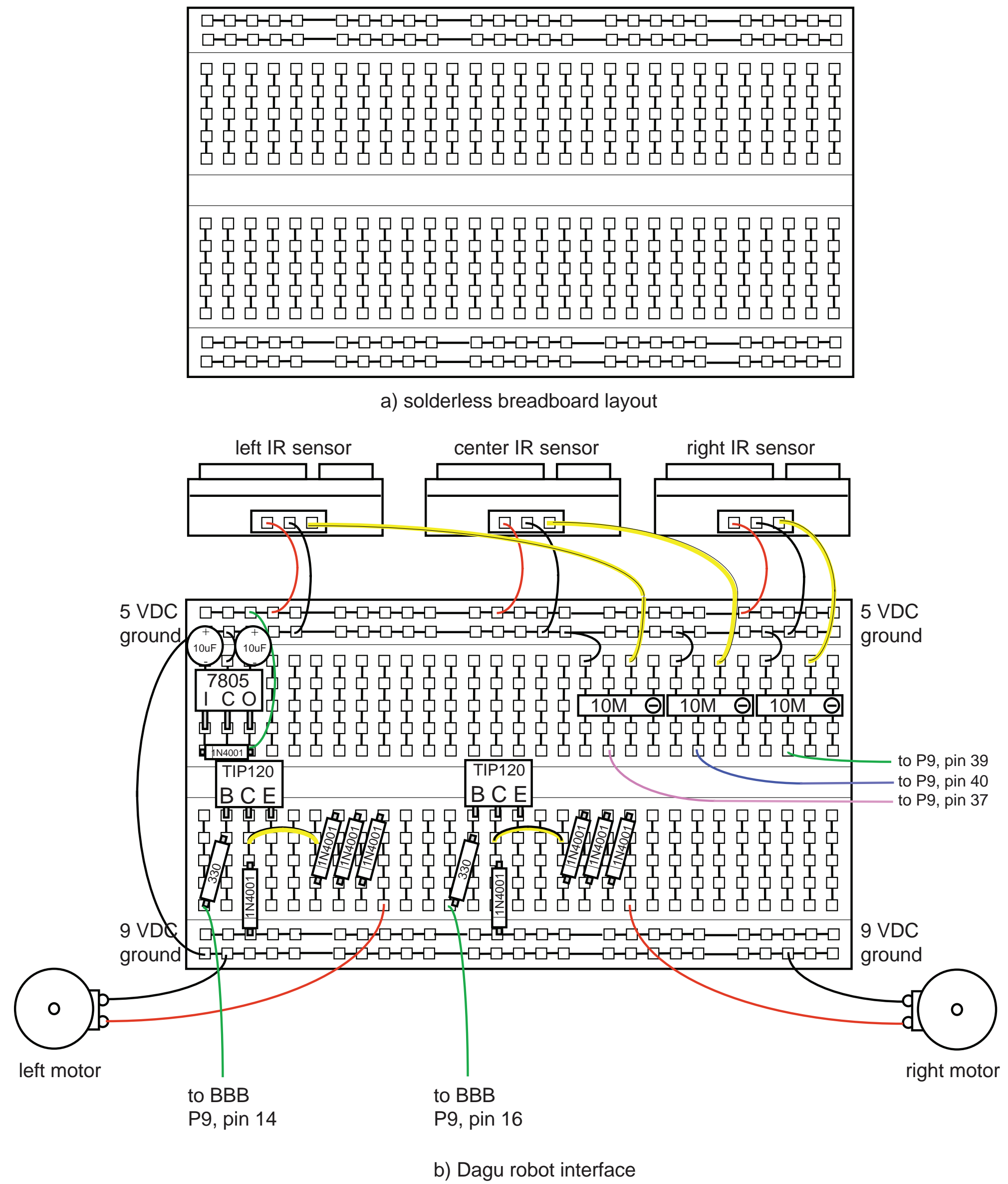
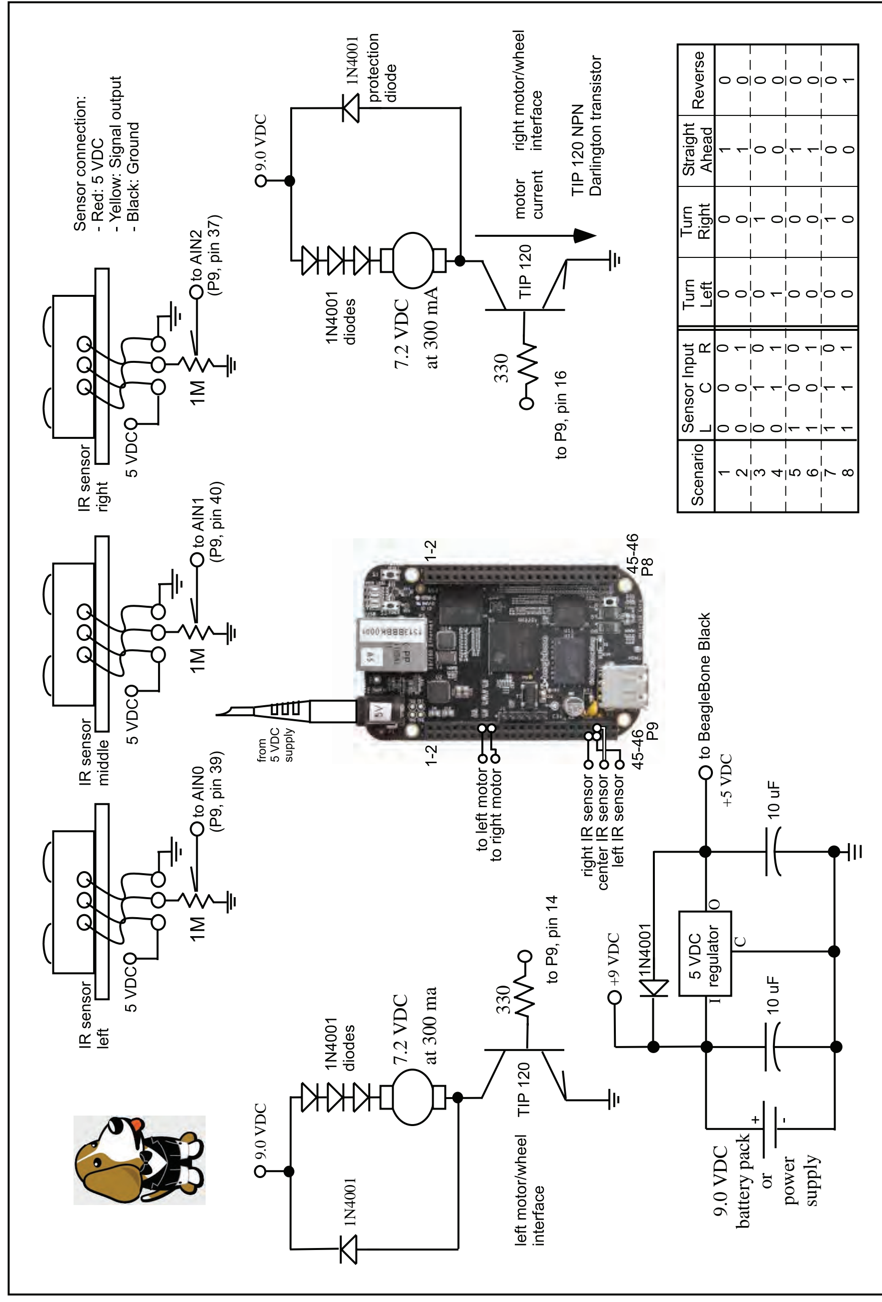


Figure 3.13: Dagu Magician robot interface circuit layout.



Scenario	Sensor Input L	C	R	Turn Left	Turn Right	Straight Ahead	Reverse
1	0	0	0	0	0	1	0
2	0	0	1	0	0	1	0
3	0	1	0	0	1	0	0
4	0	1	1	1	0	0	0
5	1	0	0	0	0	1	0
6	1	0	1	0	0	1	0
7	1	1	0	0	1	0	0
8	1	1	1	0	0	0	1

Figure 3.14: Robot circuit diagram. (Illustrations used with permission of Texas Instruments (`www.TI.com`).)

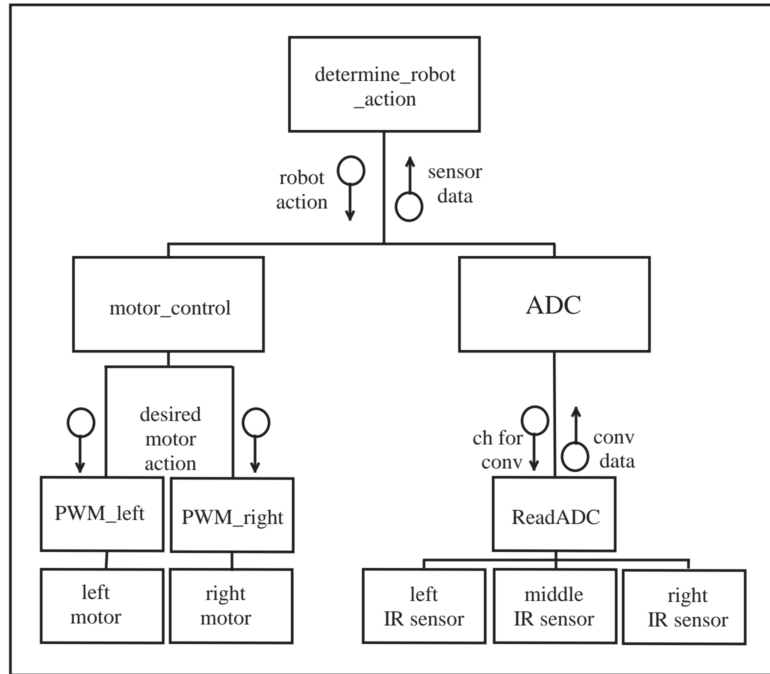


Figure 3.15: Robot structure diagram.

It is important to note that the amount of robot turn is determined by the PWM duty cycle (motor speed) and the length of time the turn is executed. For motors without optical tachometers, the appropriate values for duty cycle and motor on time must be experimentally determined. In the example functions provided the motor PWM and on time are fixed.

```
1 //****************************************************************
2 //robot.js
3 //****************************************************************
4
5 var b = require('bonescript');
6
7 var left_IR_sensor = ``P9_39'';      //analog input for left IR sensor
8 var center_IR_sensor = ``P9_40'';    //analog input for center IR
      sensor
9 var right_IR_sensor = ``P9_37'';     //analog input for right IR sensor
10
11 var left_motor_pin = ``P9_14'';     //PWM pin for left motor
12 var right_motor_pin = ``P9_16'';    //PWM pin for right motor
13
14 var left_sensor_value;
```

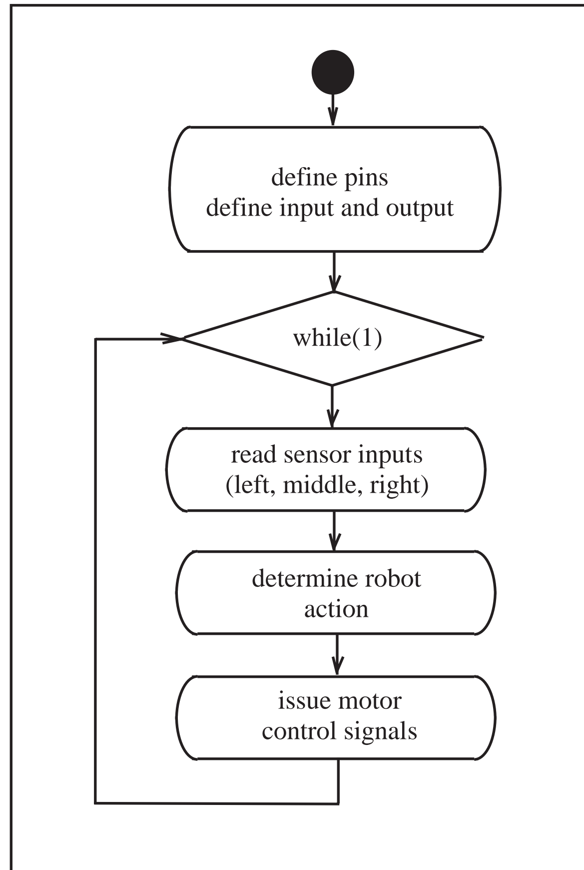


Figure 3.16: Robot UML activity diagram.

```
15 var center_sensor_value;
16 var right_sensor_value;
17
18 b.pinMode(left_motor_pin , b.OUTPUT); //left motor pin
19 b.pinMode(right_motor_pin , b.OUTPUT);//right motor pin
20
21 while(1)
22   {
23                                         //read analog output from IR sensors
24                                         //normalized value ranges from 0..1
25   left_sensor_value   = b.analogRead(left_IR_sensor);
26   center_sensor_value = b.analogRead(center_IR_sensor);
27   right_sensor_value  = b.analogRead(right_IR_sensor);
```

```
28
29                                          //assumes desired threshold at
30                                          //1.25 VDC with max value of 1.75
                                              VDC
31    if((left_sensor_value > 0.714)&&
32       (center_sensor_value <= 0.714)&&
33       (right_sensor_value > 0.714))
34      {                                   //robot continues straight ahead
35      b.analogWrite(left_motor_pin, 0.7);
36      b.analogWrite(right_motor_pin, 0.7);
37      }
38    else if
39      {
40      :
41      :
42      :
43
44      //insert other cases
45
46      }
47    }
48
49 //*******************************************************************
```

The robot may be tested in a maze. Once the robot control algorithm is downloaded and running, BeagleBone may be disconnected from the host computer. The design provided is very basic. The end of chapter homework assignments extend the design to include the following.

- Modify the PWM turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as arguments to the function.
- Equip the motor with a fourth IR sensor that looks down to the maze floor for "land mines." A land mine consists of a paper strip placed on the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must "deactivate" it by rotating three times and flashing a large LED while rotating.
- Develop a function for reversing the direction of robot movement.

3.7 SUMMARY

The goal of this chapter was to provide a tutorial on how to begin programming. We used a top--down design approach. We began with an overview of programming and explained the major parts of JavaScript and C programs for an embedded application. We examined some of the most esse ntial constructs of JavaScript and C programs in greater detail. Throughout the chapter, we provided examples and also provided pointers to a number of excellent references.

3.8 REFERENCES

- "ImageCraft Embedded Systems C Development Tools." 2015; `www.imagecraft.com`.
- Barrett, S. and Pack, D. *Embedded Systems Design and Applications with the 68HC12 and HCS12*. Upper Saddle River, NJ: Pearson Prentice Hall, 2005.
- Barrett, J. "Closer to the Sun International." 2015; `www.closertothesuninternational.com`.
- Barrett, S. and Pack, D. *Processors Fundamentals for Engineers and Scientists*. San Rafael, CA: Morgan & Claypool Publishers, 2006; `www.morganclaypool.com`.
- Barrett, S. and Pack, D. *Atmel AVR Processor Primer Programming and Interfacing*. San Rafael, CA: Morgan & Claypool Publishers, 2008; `www.morganclaypool.com`.
- Barrett, S. *Embedded Systems Design with the Atmel AVR Processor*. San Rafael, CA: Morgan and Claypool Publishers, 2010; `www.morganclaypool.com`.
- "SparkFun Electronics." 2015; `www.sparkfun.com`.
- Vander Veer, E. *JavaScript for Dummies*. 4th ed., Hoboken, NJ: Wiley Publishing, Inc., 2005.
- Pollock, J. *JavaScript*. 3rd ed., New York: McGraw Hill, 2010.
- Kiessling, M. *The Node Beginner Guide: A Comprehensive Node.js Tutorial*. 2012.
- Hughes-Croucher, T. and Wilson, M. *Node Up and Running*. Sebastopol, CA: O'Reilly Media, Inc., 2012.
- O'Berto, G. Falcon Electronic Technologies, 2015, Colorado Springs, CO.

3.9 CHAPTER EXERCISES

1. Describe the key portions of a C program.
2. What is an include file?
3. What are the three pieces of code required for a program function?
4. Describe how to define a program constant.
5. What is the difference between a "for" and "while" loop?
6. When should a switch statement be used versus the if-then statement construct?
7. Complete the Dagu Magician robot control algorithm.

8. Develop a series of functions to turn the Dagu Magician robot right and left. The motor duty cycle and the motor on time should be passed to the function as a variable.

9. Equip the motor with a fourth IR sensor that looks down to the maze floor for “land mines.” A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate it by rotating three times and flashing a large LED while rotating.

10. Develop a function for reversing the robot.

CHAPTER 4

BeagleBone Operating Parameters and Interfacing

Objectives: After reading this chapter, the reader should be able to do the following.

- Describe the voltage and current parameters for BeagleBone.
- Apply the voltage and current parameters toward properly interfacing input and output devices to BeagleBone.
- Interface BeagleBone operating at 3.3 VDC with a peripheral device operating at 5.0 VDC.
- Interface a wide variety of input and output devices to BeagleBone.
- Describe the special concerns that must be followed when BeagleBone is used to interface to a high-power DC or AC device.
- Describe how to control the speed and direction of a DC motor.
- Describe how to control several types of AC loads.
- Summarize the layout of the Prototype Cape for BeagleBone.
- Equip the Dagu Magician robot with a Liquid Crystal Display (LCD).

4.1 OVERVIEW

In this chapter, we introduce the extremely important concept of the operating envelope for a processor. Basically, if we use a processor within its operating envelope, life is good. However, if the processor is used outside of its operating envelope, unpredictable behavior or processor damage may result. It is important to use the processor within its defined operating envelope of voltage and current parameters.

We begin by reviewing the voltage and current electrical parameters for BeagleBone. We use this information to properly interface input and output devices to BeagleBone. BeagleBone operates at 3.3 VDC. There are many compatible peripheral devices. However, many peripheral devices operate at 5.0 VDC. We discuss how to interface a 3.3 VDC microcontroller to 5.0 VDC peripherals. We then discuss the special considerations for controlling a high-power DC or AC

load such as a motor. The overview of interface techniques was adapted with permission from other Morgan and Claypool projects [Pack, 2005 and Barrett, 2006]. Throughout this chapter, we provide a number of detailed examples to illustrate concepts. If this is your first exposure to electronics, please consider reading *Getting Started in Electronics* by Forrest M. Mims III [2000]. This book provides a well-written, comprehensive introduction to the fascinating world of electronics.

4.2 OPERATING PARAMETERS

A processor is an electronic device which has precisely defined operating conditions. As long as the processor is used within its defined operating parameter limits, it should continue to operate correctly. However, if the allowable conditions are violated, spurious results may arise.

4.2.1 BEAGLEBONE 3.3 VDC OPERATION

Any time a device is connected to a processor, careful interface analysis must be performed. BeagleBone digital input/output signals operate at 3.3 VDC. To perform interface analysis, there are eight different electrical specifications we must consider. The electrical parameters are defined below and illustrated in Figure 4.1:

- V_{OH}: the lowest guaranteed output voltage for a logic high;
- V_{OL}: the highest guaranteed output voltage for a logic low;
- I_{OH}: the output current for a V_{OH} logic high;
- I_{OL}: the output current for a V_{OL} logic low;
- V_{IH}: the lowest input voltage guaranteed to be recognized as a logic high;
- V_{IL}: the highest input voltage guaranteed to be recognized as a logic low;
- I_{IH}: the input current for a V_{IH} logic high; and
- I_{IL}: the input current for a V_{IL} logic low.

To properly interface a peripheral device to BeagleBone, the parameters provided in Figure 4.2 must be used. It is important to realize that these are static values taken under very specific operating conditions. If external circuitry is connected such that the processor acts as a current source (current leaving processor) or current sink (current entering processor), the voltage parameters listed above will also be affected.

In the current source case, an output voltage V_{OH} is provided at the output pin of the processor when the load connected to this pin draws a current of I_{OH}. If a load draws more current from the output pin than the I_{OH} specification, the value of V_{OH} is reduced. If the load current becomes too high, the value of V_{OH} falls below the value of V_{IH} for the subsequent logic

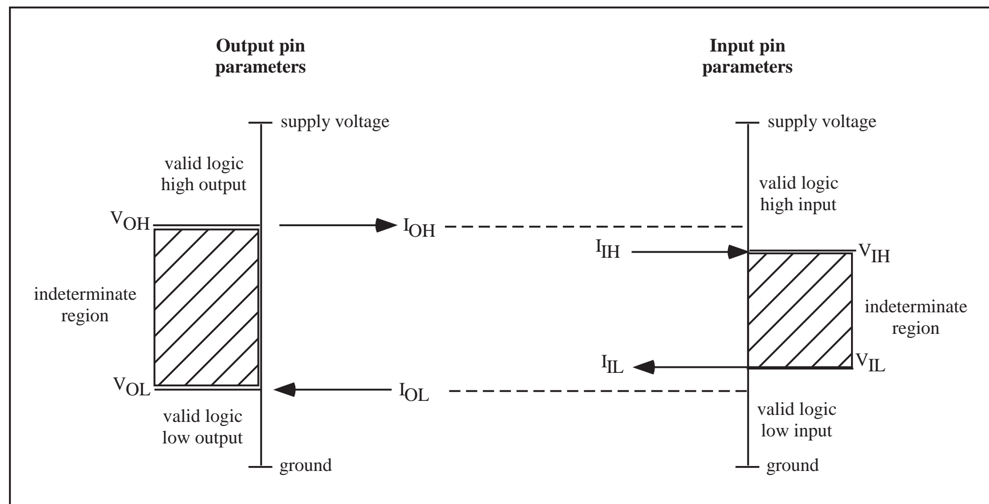


Figure 4.1: Parameters definitions.

circuit stage, and it will not be recognized as an acceptable logic high signal. When this situation occurs, erratic and unpredictable circuit behavior results.

In the sink case, an output voltage V_{OL} is provided at the output pin of the processor when the load connected to this pin delivers a current of I_{OL} to this logic pin. If a load delivers more current to the output pin of the processor than the I_{OL} specification, the value of V_{OL} increases. If the load current becomes too high, the value of V_{OL} rises above the value of V_{IL} for the subsequent logic circuit stage, and it will not be recognized as an acceptable logic low signal. As before, when this situation occurs, erratic and unpredictable circuit behavior results.

4.2.2 COMPATIBLE 3.3 VDC LOGIC FAMILIES

There are several compatible logic families that operate at 3.3 VDC. These families include the LVC, LVA, and the LVT logic families. Key parameters for the low-voltage compatible families are provided in Figure 4.3.

Prototyping Aids

Many of the 3.3 VDC logic families are only available in surface mount technology. To aid in developing a first prototype there are a number of products available to convert surface mount integrated circuits to a dual inline package style [`www.jameco.com`].

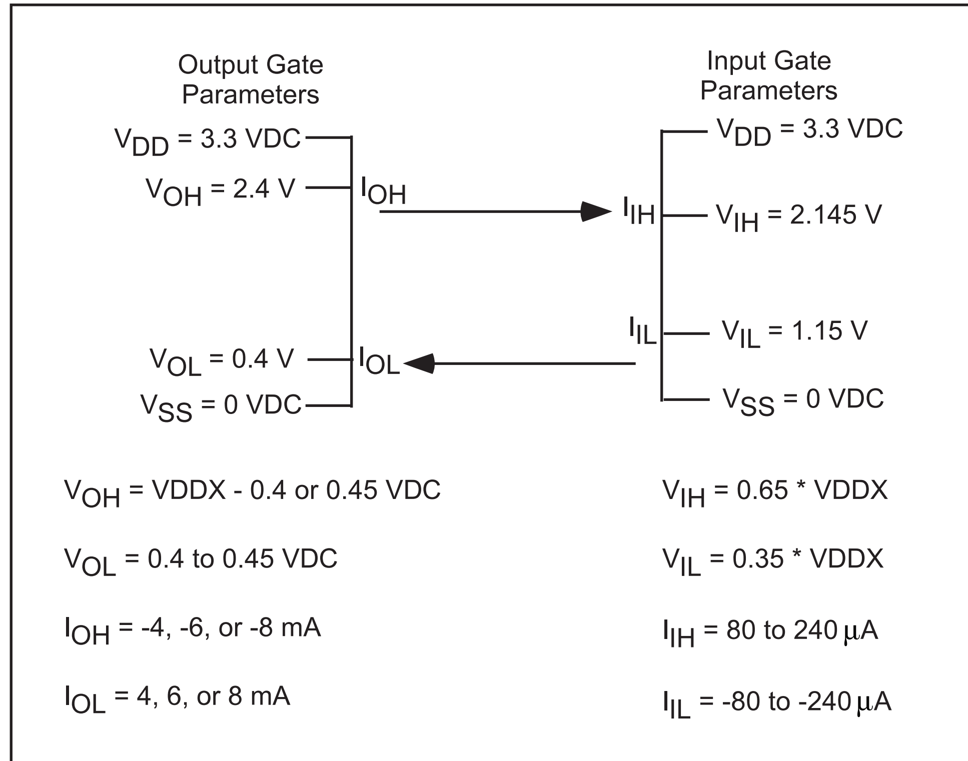


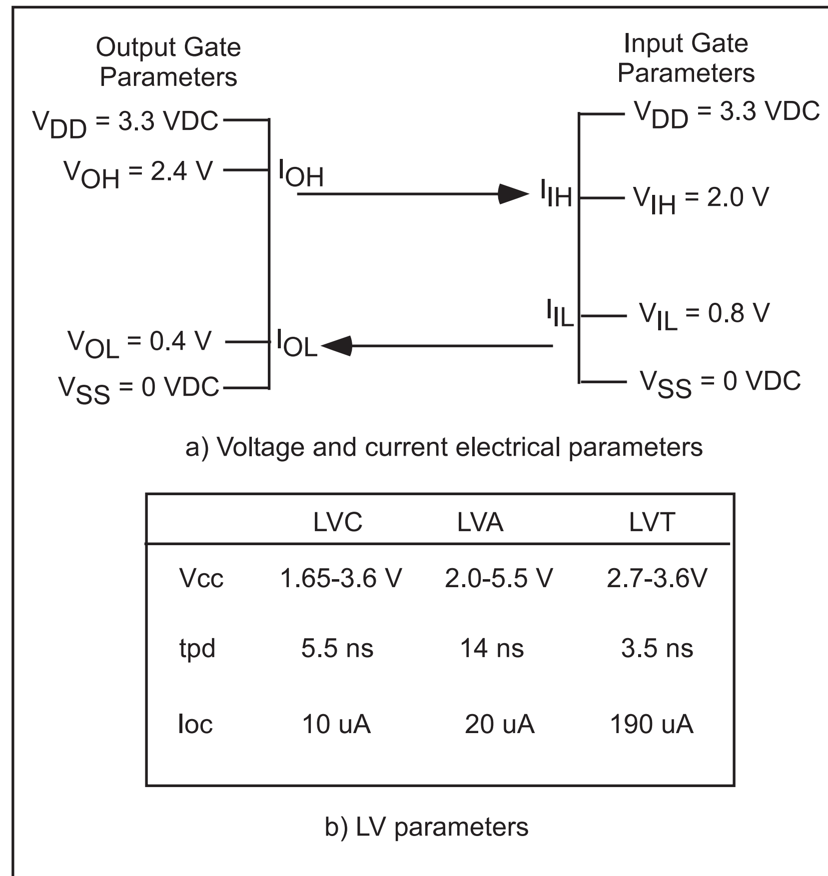
Figure 4.2: BeagleBone interface parameters.

4.2.3 INPUT/OUTPUT OPERATION AT 5.0 VDC

BeagleBone operates at 3.3 VDC. However, many HC CMOS microcontroller families and peripherals operate at a supply voltage of 5.0 VDC. For completeness, we provide operating parameters for these types of devices. This information is essential should BeagleBone be interfaced to a 5 VDC CMOS device or peripheral.

Typical values for a microcontroller in the HC CMOS family, assuming V_{DD} = 5.0 volts and V_{SS} = 0 volts, are provided below. The minus sign on several of the currents indicates a current flow out of the device. A positive current indicates current flow into the device:

- V_{OH} = 4.2 volts;
- V_{OL} = 0.4 volts;
- I_{OH} = -0.8 milliamps;
- I_{OL} = 1.6 milliamps;



	LVC	LVA	LVT
Vcc	1.65-3.6 V	2.0-5.5 V	2.7-3.6V
tpd	5.5 ns	14 ns	3.5 ns
Ioc	10 uA	20 uA	190 uA

Figure 4.3: Low-voltage compatible logic families.

- V_{IH} = 3.5 volts;
- V_{IL} = 1.0 volt;
- I_{IH} = 10 microamps; and
- I_{IL} = -10 microamps.

4.2.4 INTERFACING 3.3 VDC LOGIC FAMILIES TO 5.0 VDC LOGIC FAMILIES

Although there are a wide variety of available 3.3 VDC peripheral devices available for BeagleBone, some useful peripheral devices are not available for a 3.3 VDC system. If bidirectional information exchange is required between the processor and the peripheral device, a bidirectional

level shifter is required between the processor and the peripheral device. The level shifter translates the 3.3 VDC signal to 5 VDC for the peripheral device and back down to 3.3 VDC for the processor. There are a wide variety of unidirectional and bidirectional level shifting devices available. For example, Sparkfun Electronics has the BOB-08745 logic level shifter breakout board (`www.sparkfun.com`).

4.3 INPUT DEVICES

In this section, we discuss how to properly interface input devices to a processor. We start with the most basic input component, a simple on/off switch.

4.3.1 SWITCHES

Switches come in many varieties. As a system designer it is up to you to choose the appropriate switch for a specific application. Switch varieties commonly used in processor applications are illustrated in Figure 4.4a). Here is a brief summary of the different types.

- **Slide switch:** A slide switch has two different positions: on and off. The switch is manually moved to one position or the other. For processor applications, slide switches are available that fit in the profile of a common integrated circuit size dual inline package (DIP). A bank of four or eight DIP switches in a single package is commonly available.

- **Momentary contact pushbutton switch:** A momentary contact pushbutton switch comes in two varieties: normally closed (NC) and normally open (NO). A normally open switch, as its name implies, does not normally provide an electrical connection between its contacts. When the pushbutton portion of the switch is depressed, the connection between the two switch contacts is made. The connection is held as long as the switch is depressed. When the switch is released, the connection is opened. The converse is true for a normally closed switch. For processor applications, pushbutton switches are available in a small tactile (tact) type switch configuration.

- **Push on/push off switches:** These type of switches are also available in a normally open or normally closed configuration. For the normally open configuration, the switch is depressed to make connection between the two switch contacts. The pushbutton must be depressed again to release the connection.

- **Hexadecimal rotary switches:** Small profile rotary switches are available for processor applications. These switches commonly have sixteen rotary switch positions. As the switch is rotated to each position, a unique four bit binary code is provided at the switch contacts.

Common switch interfaces are shown in Figure 4.4a and b. This interface allows a logic one or zero to be properly introduced to a processor input port pin. The basic interface consists of the switch in series with a current limiting resistor. The node between the switch and the resistor is

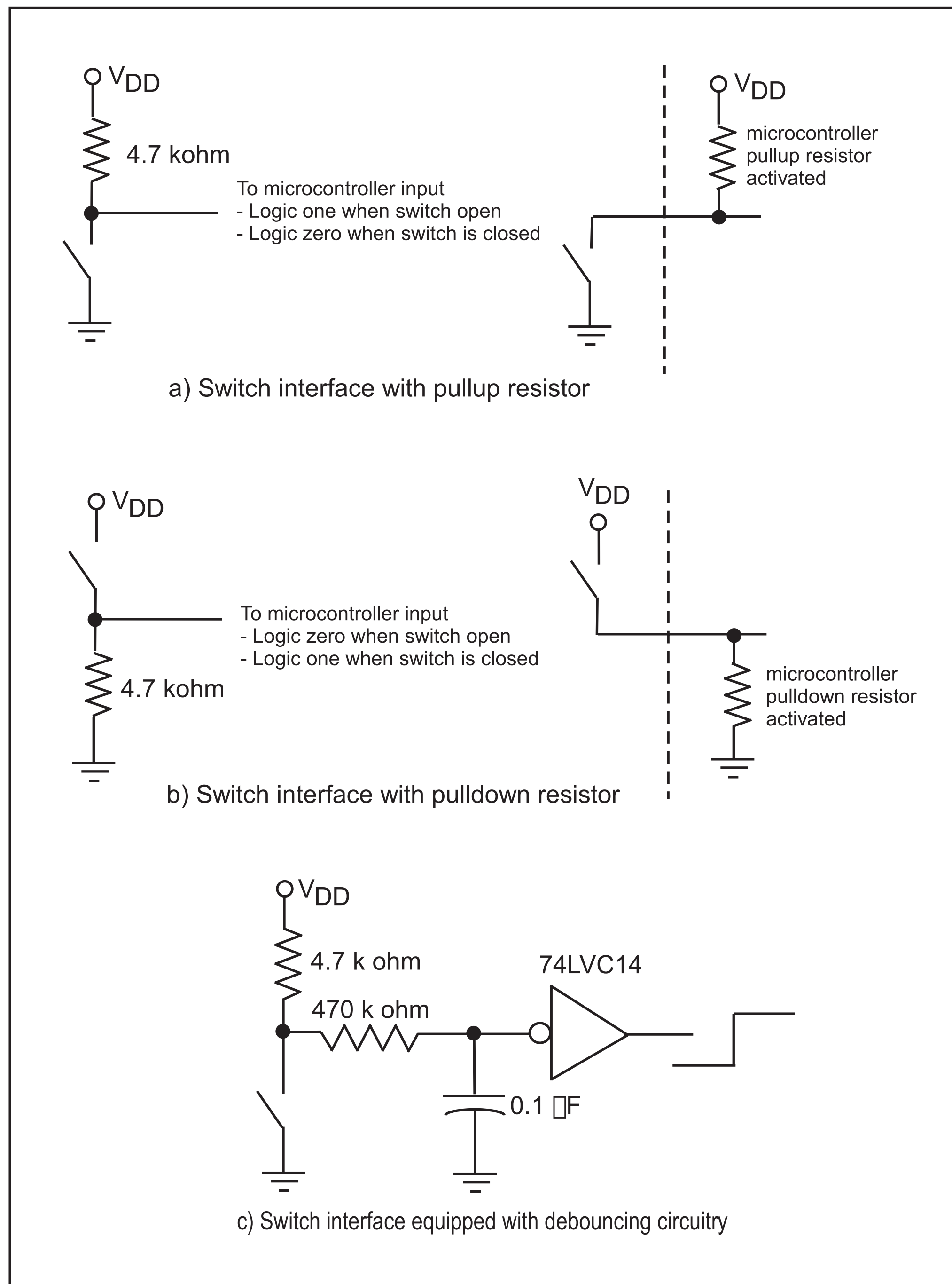


Figure 4.4: Switch interface.

provided to the processor input pin. In the configuration shown, the resistor pulls the processor input up to the supply voltage V_{DD}. When the switch is closed, the node is grounded and a logic zero is detected by the processor input pin. To reverse the logic of the switch configuration, the position of the resistor and the switch is simply reversed. In Chapter 6, we discuss the built-in pull up resistor features on BeagleBone Black.

4.3.2 SWITCH DEBOUNCING

Unfortunately, mechanical switches do not make a clean transition from one position (on) to another (off). When a switch is moved from one position to another, it makes and breaks contact multiple times. This activity may go on for tens of milliseconds. A processor is relatively fast as compared to the action of the switch. Therefore, the processor is able to recognize each switch bounce as a separate and erroneous transition.

To correct the switch bounce phenomenon, additional external hardware components may be used or software techniques may be employed. A hardware debounce circuit is illustrated in Figure 4.4c. The node between the switch and the limiting resistor of the basic switch circuit is fed to a low pass filter (LPF), formed by the 470 k ohm resistor and the capacitor. The LPF prevents abrupt changes (bounces) in the input signal from the processor. The LPF is followed by a 74LVC14 Schmitt Trigger which is simply an inverter equipped with hysteresis. This further limits the switch bouncing.

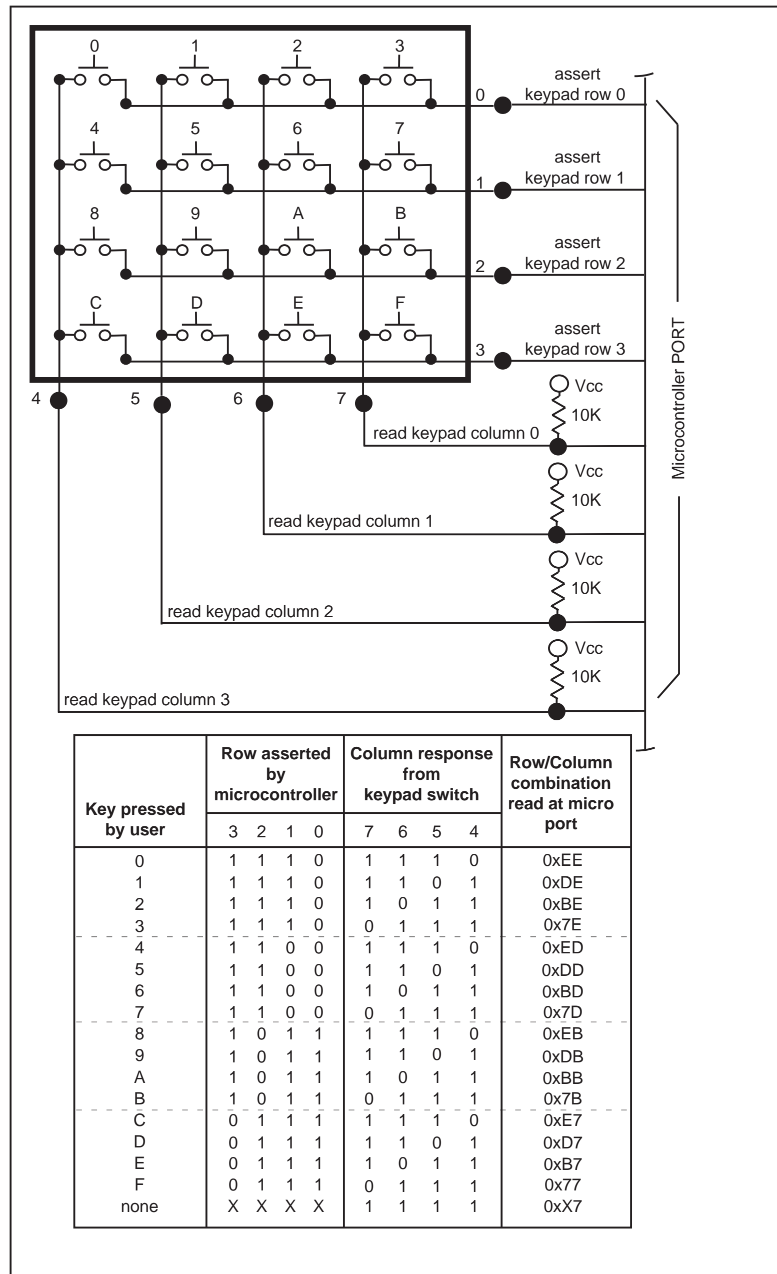
Switches may also be debounced using software techniques. This is accomplished by inserting a 30–50 ms lockout delay in the function responding to port pin changes. The delay prevents the processor from responding to the multiple switch transitions due to bouncing.

You must carefully analyze a given design to determine if hardware or software switch debouncing techniques should be used. It is important to remember that all switches exhibit bounce phenomena and therefore must be debounced.

4.3.3 KEYPADS

A keypad is simply an extension of the simple switch configuration. A typical keypad configuration and interface are shown in Figure 4.5. As you can see, the keypad simply consists of multiple switches in the same package. A hexadecimal keypad is shown in the figure. A single row of keypad switches is asserted by the processor, and then the host keypad port is immediately read. If a switch in this row has been depressed, the keypad pin corresponding to the column the switch is in will also be asserted. The combination of a row and a column assertion can be decoded to determine which key has been pressed as illustrated in the table. Keypad rows are continually asserted one after the other in sequence. Since the keypad is a collection of switches, debounce techniques must also be employed.

The keypad may be used to introduce user requests to a processor. A standard keypad with alphanumeric characters may be used to provide alphanumeric values to the processor such as providing your personal identification number (PIN) for a financial transaction. However, some



Key pressed by user	Row asserted by microcontroller				Column response from keypad switch				Row/Column combination read at micro port
	3	2	1	0	7	6	5	4	
0	1	1	1	0	1	1	1	0	0xEE
1	1	1	1	0	1	1	0	1	0xDE
2	1	1	1	0	1	0	1	1	0xBE
3	1	1	1	0	0	1	1	1	0x7E
4	1	1	0	0	1	1	1	0	0xED
5	1	1	0	0	1	1	0	1	0xDD
6	1	1	0	0	1	0	1	1	0xBD
7	1	1	0	0	0	1	1	1	0x7D
8	1	0	1	1	1	1	1	0	0xEB
9	1	0	1	1	1	1	0	1	0xDB
A	1	0	1	1	1	0	1	1	0xBB
B	1	0	1	1	0	1	1	1	0x7B
C	0	1	1	1	1	1	1	0	0xE7
D	0	1	1	1	1	1	0	1	0xD7
E	0	1	1	1	1	0	1	1	0xB7
F	0	1	1	1	0	1	1	1	0x77
none	X	X	X	X	1	1	1	1	0xX7

Figure 4.5: Keypad interface.

keypads are equipped with removable switch covers such that any activity can be associated with a key press.

4.3.4 SENSORS

A processor is typically used in control applications where data is collected, the data is assimilated and processed by the host algorithm, and a control decision and accompanying signals are provided by the processor. Input data for the processor is collected by a complement of input sensors. These sensors may be digital or analog in nature.

Digital Sensors

Digital sensors provide a series of digital logic pulses with sensor data encoded. The sensor data may be encoded in any of the parameters associated with the digital pulse train such as duty cycle, frequency, period, or pulse rate. The input portion of the timing system may be configured to measure these parameters.

An example of a digital sensor is an optical encoder. An optical encoder consists of a small transparent, plastic disk with opaque lines etched on the disk surface. A stationary optical emitter and detector source are placed on either side of the disk. As the disk rotates, the opaque lines break the continuity between the optical source and detector. The signal from the optical detector is monitored to determine disk rotation, as shown in Figure 4.6. The optical encoder configuration provides an optical tachometer.

There are two major types of optical encoders: incremental encoders and absolute encoders. An absolute encoder is used when it is required to retain position information when power is lost. For example, if you were using an optical encoder in a security gate control system, an absolute encoder would be used to monitor the gate position. An incremental encoder is used in applications where a velocity or a velocity and direction information is required.

The incremental encoder types may be further subdivided into tachometers and quadrature encoders. An incremental tachometer encoder consists of a single track of etched opaque lines as shown in Figure 4.6a. It is used when the velocity of a rotating device is required. To calculate velocity, the number of detector pulses is counted in a fixed amount of time. Since the number of pulses per encoder revolution is known, velocity may be calculated.

The quadrature encoder contains two tracks shifted in relationship to one another by 90°. This allows the calculation of both velocity and direction. To determine direction, one would monitor the phase relationship between Channel A and Channel B as shown in Figure 4.6b. The absolute encoder is equipped with multiple data tracks to determine the precise location of the encoder disk [Sick Stegmann].

Analog Sensors and Transducers

Analog sensors or transducers provide a DC voltage that is proportional to the physical parameter being measured. The analog signal may be first preprocessed by external analog hardware such that

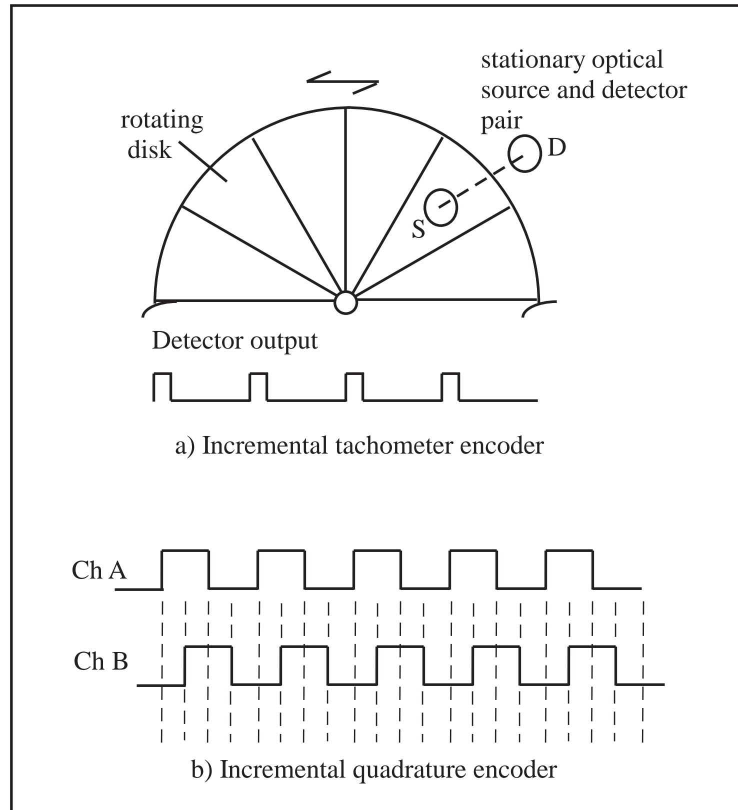


Figure 4.6: Optical encoder.

it falls within the voltage references of the conversion subsystem. In the case of BeagleBone, the transducer output must fall between 0 and 1.8 VDC. The analog voltage is then converted to a corresponding binary representation. **Note: As previously mentioned, the analog input to the analog-to-digital converter must not exceed 1.8 VDC.**

Example 1: Flex Sensor

An example of an analog sensor is the flex sensor shown in Figure 4.7a. The flex sensor provides a change in resistance for a change in sensor flexure. At 0° flex, the sensor provides 10 k Ohms of resistance. For 90° flex, the sensor provides 30–40 k Ohms of resistance. Since the processor can not measure resistance directly, the change in flex sensor resistance must be converted to a change in a DC voltage. This is accomplished using the voltage divider network shown in Figure 4.7c. For increased flex, the DC voltage will increase. The voltage can be measured using the BeagleBone's analog-to-digital converter subsystem.

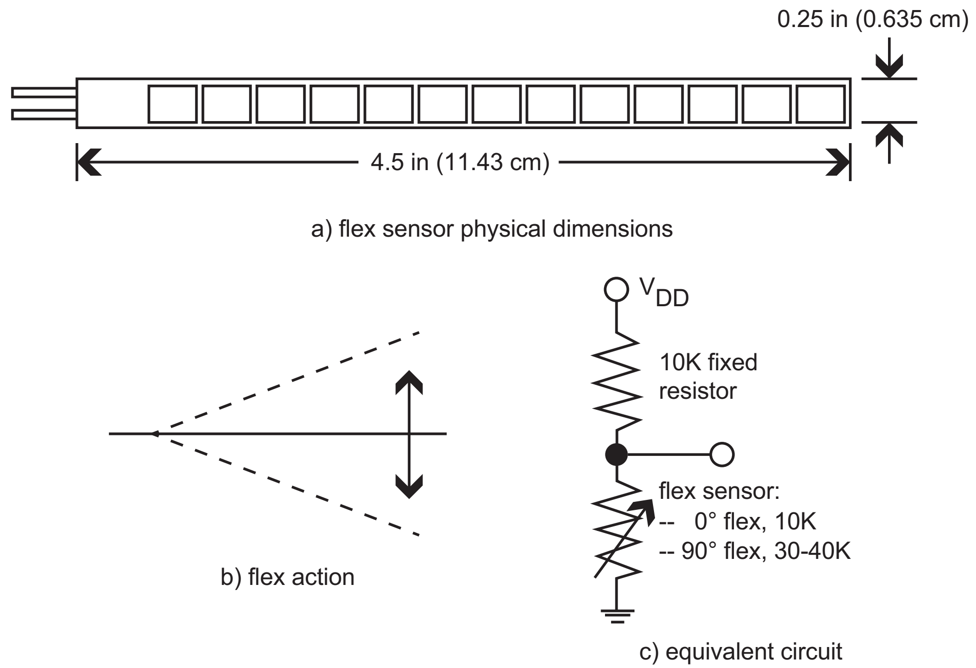


Figure 4.7: Flex sensor.

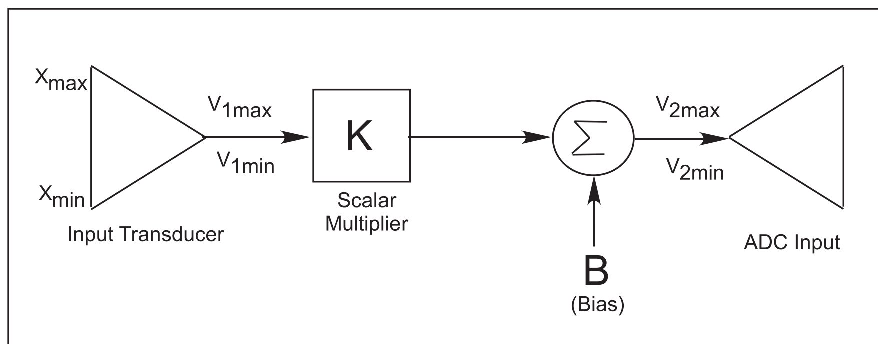


Figure 4.8: A block diagram of the signal conditioning for an analog-to-digital converter. The range of the sensor voltage output is mapped to the analog-to-digital converter input voltage range. The scalar multiplier maps the magnitudes of the two ranges and the bias voltage is used to align two limits.

The flex sensor may be used in applications such as virtual reality data gloves, robotic sensors, biometric sensors, and in science and engineering experiments [Images Company]. One of the co-authors (Steven Barret) used the circuit provided in Figure 4.7 to help a colleague in Zoology monitor the movement of a newt salamander during a scientific experiment.

Example 2: Ultrasonic Sensor

The ultrasonic sensor pictured in Figure 4.9 is an example of an analog-based sensor. The sensor is based on the concept of ultrasound or sound waves that are at a frequency above the human range of hearing (20 Hz to 20 kHz). The ultrasonic sensor pictured in Figure 4.9c emits a sound wave at 42 kHz. The sound wave reflects from a solid surface and returns back to the sensor. The amount of time for the sound wave to transit from the surface and back to the sensor may be used to determine the range from the sensor to the wall. Pictured in Figure 4.9c and d is an ultrasonic sensor manufactured by Maxbotix (LV-EZ3). The sensor provides an output that is linearly related to range in three different formats: (a) a serial RS-232 compatible output at 9600 bits per second, (b) a pulse output which corresponds to 147 us/inch width, and (c) an analog output at a resolution of 10 mV/inch. The sensor is powered from a 2.5–5.5 VDC source [`www.sparkfun.com`].

Example 3: Inertial Measurement Unit

Pictured in Figure 4.10 is an inertial measurement unit (IMU) combination which consists of an IDG5000 dual-axis gyroscope and an ADXL335 triple axis accelerometer. This sensor may be used in unmanned aerial vehicles (UAVs), autonomous helicopters, and robots. For robotic applications the robot tilt may be measured in the X and Y directions, as shown in Figure 4.10c and d [`www.sparkfun.com`].

Example 4: Level Sensor

Milone Technologies manufacture a line of continuous fluid level sensors. The sensor resembles a ruler and provides a near liner response as shown in Figure 4.11. The sensor reports a change in resistance to indicate the distance from sensor top to the fluid surface. A wide resistance change occurs from 700 Ohms at a one inch fluid level to 50 Ohms at a 12.5-in fluid level [`www.milonetech.com`].

To covert the resistance change to a voltage change measurable by BeagleBone Black, a voltage divider circuit as shown in Figure 4.11 may be used. With a supply voltage (V^{DD})of 3.3 VDC, a V_{TAPE} voltage of 0.855 VDC results for a one inch fluid level, whereas a fluid of 12.5-in provides a V_{TAPE} voltage level of 0.080 VDC.

4.3.5 TRANSDUCER INTERFACE DESIGN (TID) CIRCUIT

In addition to transducers, we also need a signal conditioning circuitry before we can apply the signal for analog-to-digital conversion. The signal conditioning circuitry is called the transducer

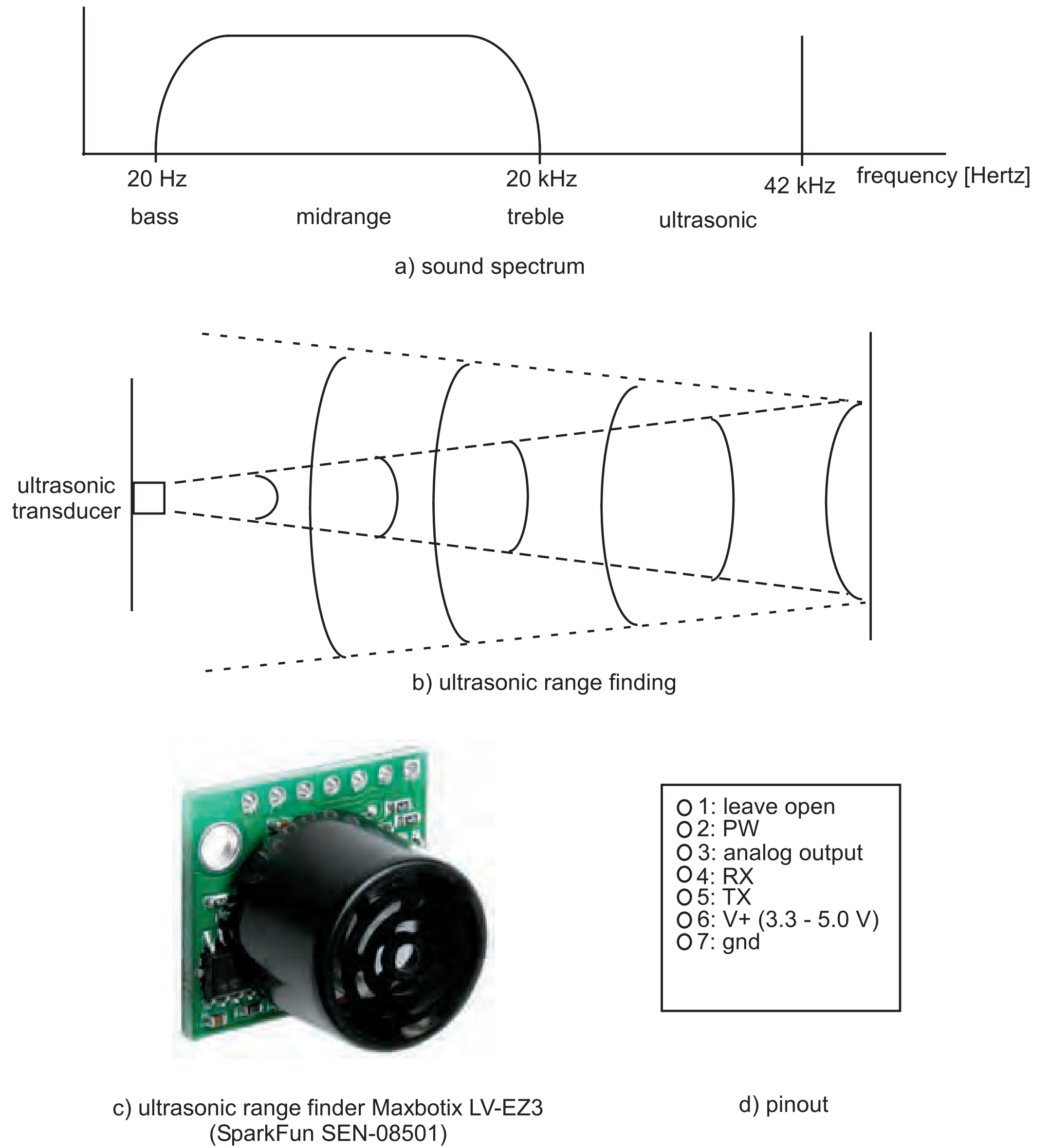
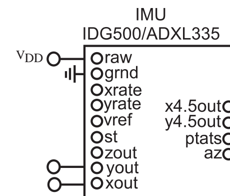
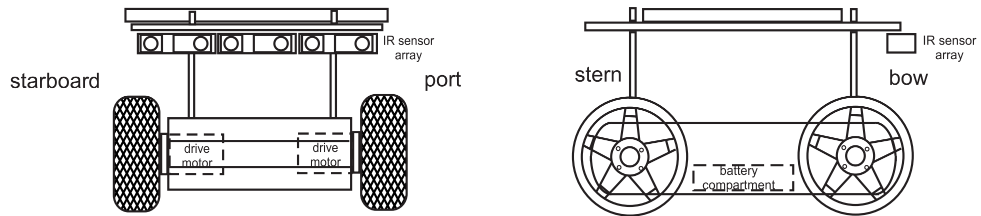


Figure 4.9: Ultrasonic sensor. (Sensor image used courtesy of SparkFun, Electronics.)

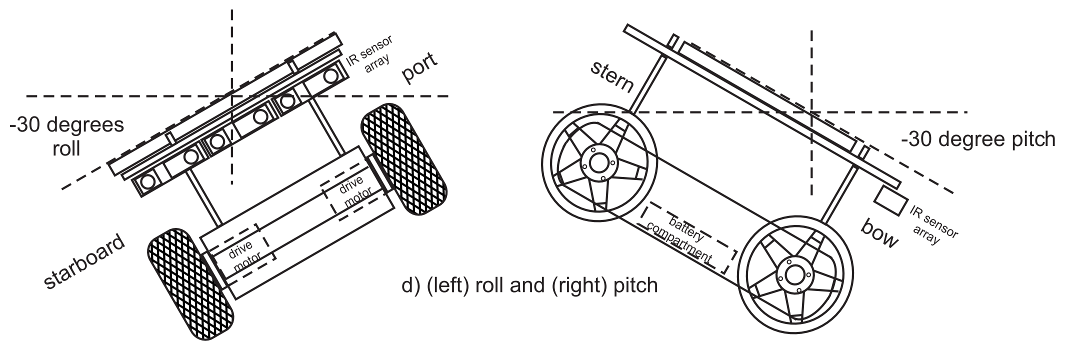
a) SparkFun IMU Analog Combo Board
5 Degrees of Freedom IDG500/ADXL335 SEN



b) IDG500/ADXL335 pinout

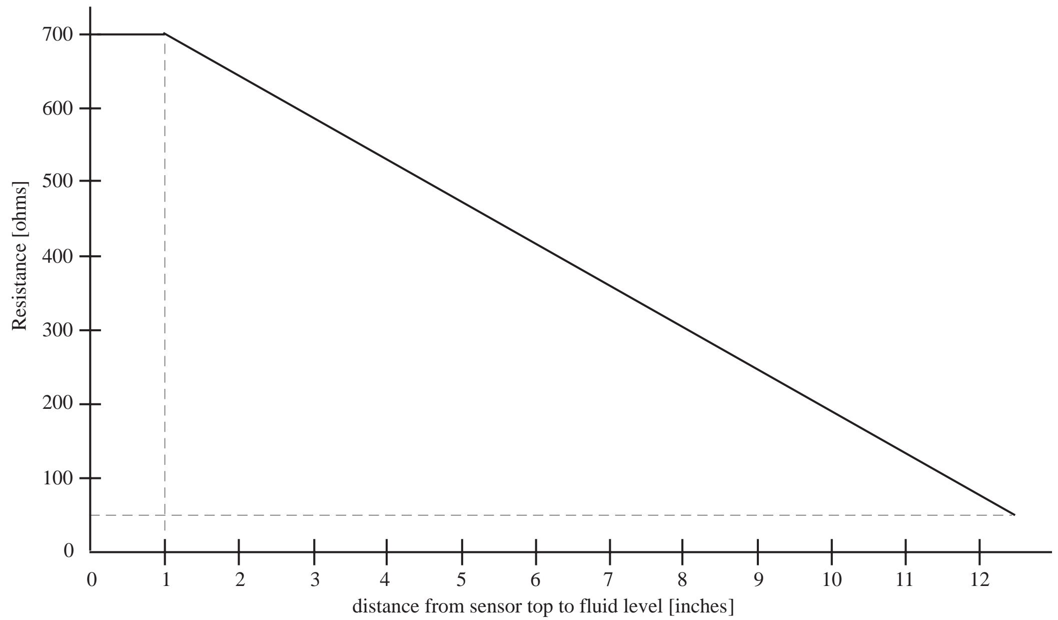


c) (left) robot front view and (right) side view

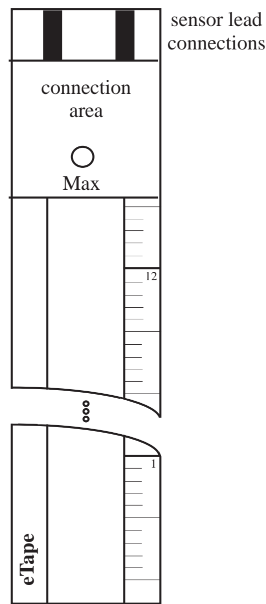


d) (left) roll and (right) pitch

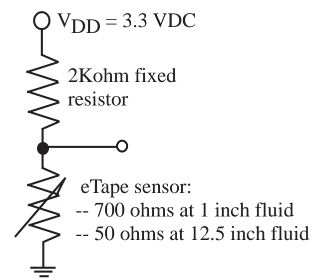
Figure 4.10: Inertial measurement unit. (IMU image used courtesy of SparkFun, Electronics.)



a) characteristics for Milone Technologies eTapeTM fluid level sensor.



b) eTape sensor



c) equivalent circuit

Figure 4.11: Milone Technologies fluid level sensor. (www.milonetech.com)

interface. The objective of the transducer interface circuit is to scale and shift the electrical signal range to map the output of the transducer to the input range of the analog-to-digital converter, which is typically 0–3.3 VDC. Figure 4.8 shows the transducer interface circuit using an input transducer.

The transducer interface consists of two steps: scaling and shifting via a DC bias. The scale step allows the span of the transducer output to match the span of the analog-to-digital conversion (ADC) system input range. The bias step shifts the output of the scale step to align with the input of the ADC system. In general, the scaling and bias process may be described by two equations:

$$V_{2max} = (V_{1max} \times K) + B$$

$$V_{2min} = (V_{1min} \times K) + B$$

The variable V_{1max} represents the maximum output voltage from the input transducer. This voltage occurs when the maximum value of the physical variable (X_{max}) is presented to the input transducer. This voltage must be scaled by the scalar multiplier (K) and then have a DC offset bias voltage (B) added to provide the voltage V_{2max} to the input of the ADC converter [USAFA].

Similarly, the variable V_{1min} represents the minimum output voltage from the input transducer. This voltage occurs when the minimum physical variable (X_{min}) is presented to the input transducer. This voltage must be scaled by the scalar multiplier (K) and then have a DC offset bias voltage (B) added to produce voltage V_{2min}, the input of the ADC converter.

Usually, the values of V_{1max} and V_{1min}, are provided with the documentation for the transducer. Also, the values of V_{2max} and V_{2min} are known. They are the high- and low-reference voltages for the ADC system (1.8 VDC and 0 VDC for BeagleBone). We thus have two equations and two unknowns to solve for K and B. The circuits to scale by K and add the offset B are usually implemented with operational amplifiers.

Example: A photodiode is a semiconductor device that provides an output current, corresponding to the light impinging on its active surface. The photodiode is used with a transimpedance amplifier to convert the output current to an output voltage. A photodiode/transimpedance amplifier provides an output voltage of 0 volts for maximum rated light intensity and −2.50 VDC output voltage for the minimum rated light intensity. Calculate the required values of K and B for this light transducer, so it may be interfaced to BeagleBone's ADC system.

$$V_{2max} = (V_{1max} \times K) + B$$

$$V_{2min} = (V_{1min} \times K) + B$$

$$1.8\ V = (0\ V \times K) + B$$

$$0\,V = (-2.50\,V \times K) + B$$

The values of K and B may then be determined to be 0.72 and 1.8 VDC, respectively.

4.3.6 OPERATIONAL AMPLIFIERS

In the previous section, we discussed the transducer interface design (TID) process. Going through this design process yields a required value of gain (K) and DC bias (B). Operational amplifiers (op amps) are typically used to implement a TID interface. In this section, we briefly introduce operational amplifiers including ideal op amp characteristics, classic op amp circuit configurations, and an example to illustrate how to implement a TID with op amps. Op amps are also used in a wide variety of other applications, including analog computing, analog filter design, and a myriad of other applications. Interested readers are referred to Section 4.12 at the end of this chapter for pointers to some excellent texts on this topic.

The Ideal Operational Amplifier

An ideal operational amplifier is shown in Figure 4.12. An ideal operational amplifier does not exist in the real world. However, it is a good first approximation for use in developing op amp application circuits.

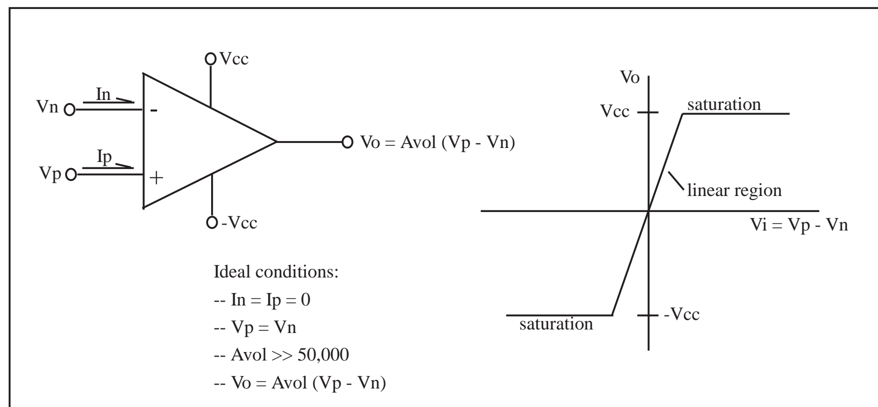


Figure 4.12: Ideal operational amplifier characteristics.

The op amp is an active device (requires power supplies) equipped with two inputs, a single output, and several voltage source inputs. The two inputs are labeled Vp, or the non-inverting input, and Vn, the inverting input. The output of the op amp is determined by taking the difference between Vp and Vn and multiplying the difference by the open loop gain (A_{vol}) of the op amp, which is typically 10,000. Due to the large value of A_{vol}, it does not take much of a difference between Vp and Vn before the op amp will saturate. When an op amp saturates, it

does not damage the op amp, but the output is limited to $\pm V_{cc}$. This will clip the output, and hence distort the signal, at levels slightly less than $\pm V_{cc}$. Op amps are typically used in a closed loop, negative feedback configuration. A sample of classic operational amplifier configurations with negative feedback are provided in Figure 4.13 [Faulkenberry, 1977].

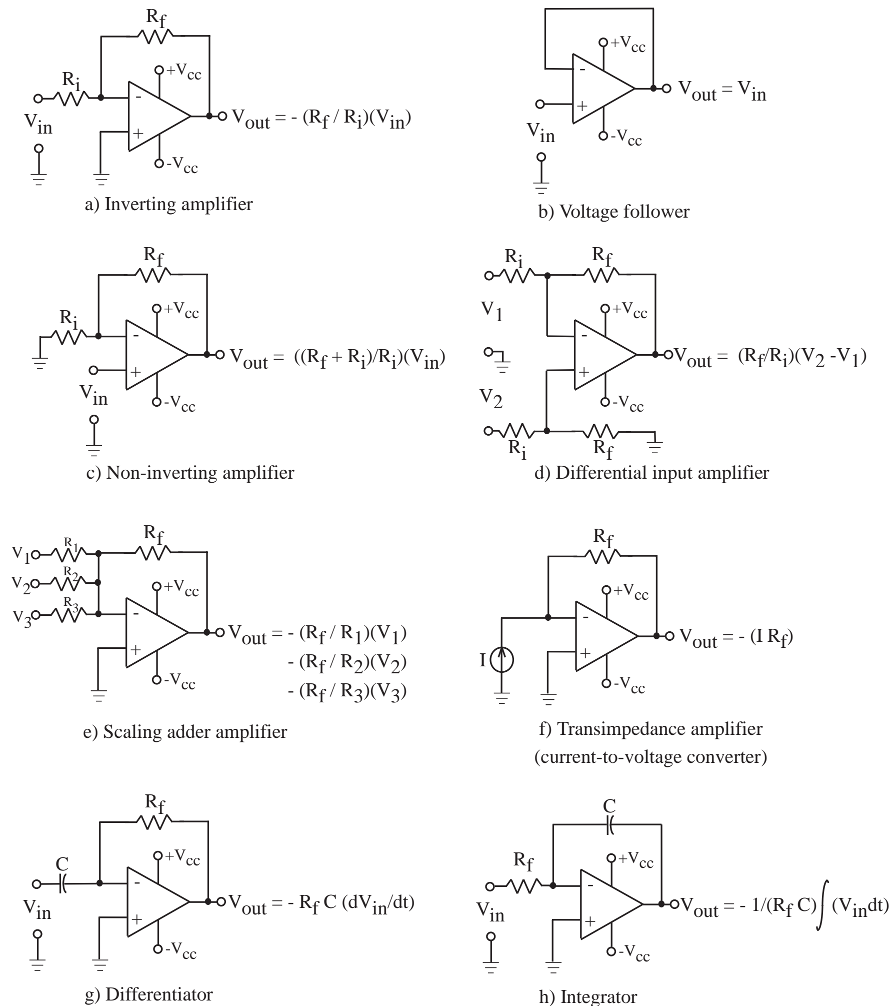


Figure 4.13: Classic operational amplifier configurations. (Adapted from Faulkenberry, 1977.)

It should be emphasized that the equations provided with each operational amplifier circuit are only valid if the circuit configurations are identical to those shown. Even a slight variation in the circuit configuration may have a dramatic effect on circuit operation. To analyze each operational amplifier circuit, use the following steps.

- Write the node equation at the inverting input.
- Apply ideal op amp characteristics to the node equation.
- Solve the node equation for Vo.

As an example, we provide the analysis of the non-inverting amplifier circuit in Figure 4.14. This same analysis technique may be applied to all of the circuits in Figure 4.13 to arrive at the equations for Vout provided.

Example: In the previous section, it was determined that the values of K and B were 0.72 and 1.8 VDC, respectively. The two-stage op amp circuitry in Figure 4.15 implements these values of K and B. The first stage provides an amplification of -0.72 due to the use of the inverting amplifier configuration. In the second stage, a summing amplifier is used to add the output of the first stage with a bias of 1.8 VDC. Since this stage also introduces a minus sign to the result, the overall result of a gain of 0.72 and a bias of $+1.8$ VDC is achieved.

Low-voltage operational amplifiers, operating in the ± 2.7 to ± 5 VDC range, are readily available from Texas Instruments.

4.4 OUTPUT DEVICES

An external device should not be connected to a processor without first performing careful interface analysis to ensure the voltage, current, and timing requirements of the processor and the external device are met. In this section, we describe interface considerations for a wide variety of external devices. We begin with the interface for a single light-emitting diode.

4.4.1 LIGHT-EMITTING DIODES (LEDS)

LEDs may be used to indicate the logic state at a specific pin on a processor. Most LEDs have two leads: the anode or positive lead and the cathode or negative lead. When taking a "bird's eye" view of a round LED from above, one side has a slight flattening. The cathode is the lead nearest the flat portion. Also, you can hold an LED up to a light source and distinguish the cathode from the anode by its characteristic shape as shown in Figure 4.16b.

To properly bias an LED, the anode lead must be biased at a level approximately 1.7–2.2 volts higher than the cathode lead. This specification is known as the forward voltage (V_f) of the LED. The LED current must also be limited to a safe current level known as the forward current (I_f). The forward diode voltage and current specifications are usually provided by the manufacturer.

A processor normally represents a logic one with a logic high voltage. In the processor documentation this is referred to as V_{OH} or the voltage when an output pin is at logic high. When at a logic high, a processor pin delivers (sources) current to the external circuit connected

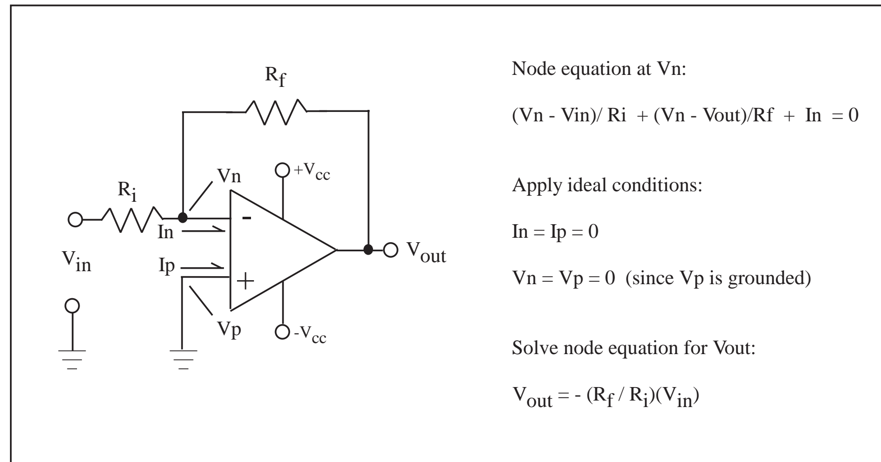


Figure 4.14: Operational amplifier analysis for the non-inverting amplifier. (Adapted from Faulkenberry, 1977.)

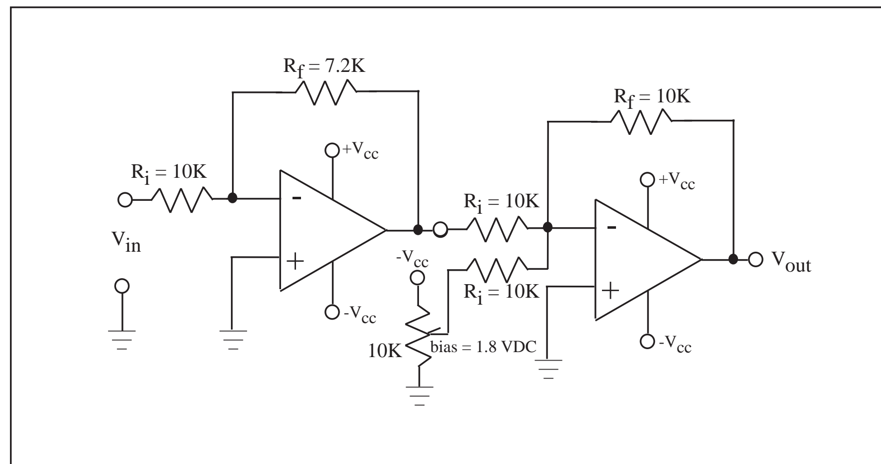


Figure 4.15: Operational amplifier implementation of the transducer interface design (TID) example circuit.

to it. The pin acts as a small DC power supply with an output voltage of V_{OH} with a maximum current rating of I_{OH}.

To properly interface external components to the processor, the V_{OH} and I_{OH} values of a specific pin must be determined. In this example, we interface a LED to BeagleBone header P8 pin 13. It is important to note BeagleBone documentation uses different pin names than the host

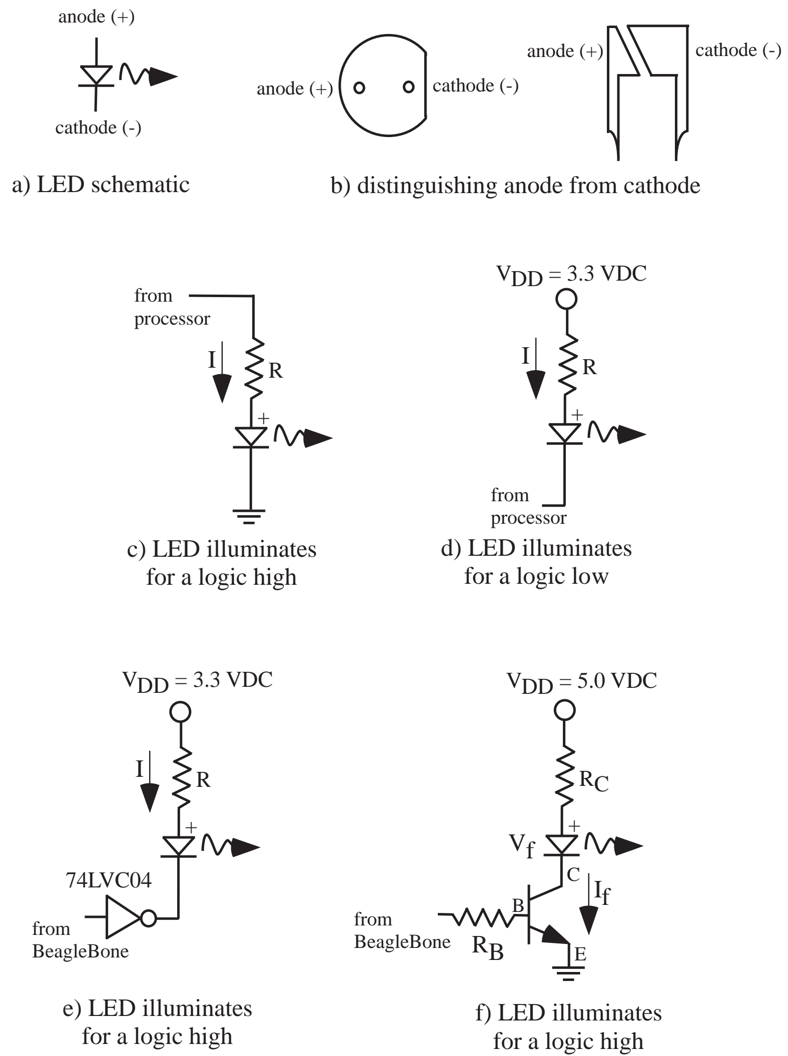


Figure 4.16: Interfacing an LED.

processor aboard. To resolve the names and find the V_{OH} and I_{OH} values for a pin, the following steps are required.

- In the *BeagleBone Black System Reference Manual,* Revision A5.2 [Coley, 2014], determine the processor ball associated with the header pin. This information is provided on Table 10. Expansion Header P8 Pinout for pin 13. The signal name associated with pin 13 is "EHRPWM2B" and the associated processor pin is "T10."

- Looking at the processor datasheet for the pin will provide us with additional information useful for interfacing to the pin. Recall BeagleBone uses the AM3358/9 ARM Cortex--A8 microprocessor. BeagleBone uses the ZCZ package. The datasheet for this processor may be downloaded from the Texas Instruments website. Tables 2–7 (Ball Characteristics) of the AM3358/9 datasheet provides the initial details. The datasheet uses the pin name "GPMC_AD9" and it is associated with ZCE and ZCZ ball pins W16 and T10, respectively. Notice that each pin has multiple functions as denoted by the mode number. This is covered in this book.

- As a final step the V_{OH} and I_{OH} values for the pin may be located in Tables 3–11 of the datasheet. DC Electrical Characteristics. Note the pin names used in the table are for multiplexer mode 0. Tables 3–11 is then scanned from the beginning looking for a specific pin name. If it is not found the last page of the table is used to obtain the V_{OH} and I_{OH} values. In this example, the last page of the table is used to determine the values. For BeagleBone the $VDDSHVx = 3.3\ V$ portion of the table is used. We find the value of V_{OH} is specified as $VDDSHVx - 0.2\ V$ or 3.1 volts. The value of I_{OH} is given as 6 mA.

With the values of V_{OH} and I_{OH} determined, an interface circuit can be designed.

In Figure 4.16c–f we provide several methods of properly interfacing an LED to BeagleBone Black. Several alternatives are provided for flexibility in design. All variations will be used throughout the textbook.

An LED biasing circuit is provided in Figure 4.16c. In Figure 4.16c a logic one provided by the processor provides the voltage to forward bias the LED. The processor also acts as the source for the forward current through the LED. To properly bias the LED the value of the limit resistor (R) is chosen.

Example: A red (635 nm) LED is rated at 1.8 VDC with a forward operating current of 6 mA. Design a proper bias for the LED using the configuration of Figure 4.16c.

Answer: In the configuration of Figure 4.16c the processor pin can be viewed as an unregulated power supply. That is, the pin's output voltage is determined by the current supplied by the pin. The current flows out of the processor pin through the LED and resistor combination to

ground (current source). The value of R may be calculated using Ohm's Law. The voltage drop across the resistor is the difference between the 3.1 VDC supplied by the processor pin and the LED forward voltage of 1.8 VDC. The current flowing through the resistor is the LED's forward current (6 mA). This renders a resistance value of approximately 220 Ohms:

$$(V_{DD} - V_{LED}/I_{LED} = \\ (3.1 - 1.8)/.006 = 216.7$$

For the LED interface provided in Figure 4.16d, the LED is illuminated when the processor provides a logic low. In this case, the current flows from the power supply back into the processor pin (current sink).

If LEDs with higher forward voltages and currents are used, alternative interface circuits may be employed. Figures 4.16e and 4.16f provide two more LED interface circuits. In Figure 4.16e, a logic one is provided by BeagleBone to the input of the inverter. The inverter provides a logic zero at its output, which provides a virtual ground at the cathode of the LED. Therefore, the proper voltage biasing for the LED is provided. The resistor (R) limits the current through the LED. A proper resistor value can be calculated using $R = (V_{DD} - V_{DIODE})/I_{DIODE}$. It is important to note that the inverter used must have sufficient current sink capability (I_{OL}) to safely handle the forward current requirements of the LED. As in previous examples, the characteristic curves of the inverter must be carefully analyzed.

An NPN transistor such as a 2N2222 (PN2222 or MPQ2222) may be used in place of the inverter, as shown in Figure 4.16f. In this configuration, the transistor is used as a switch. When a logic low is provided by the processor, the transistor is in the cutoff region. When a logic one is provided by the microcontroller, the transistor is driven into the saturation region. To properly interface the processor to the LED, resistor values R_B and R_C must be chosen. The resistor R_B is chosen to limit the base current.

Example: Using the interface configuration of Figure 4.16f, design an interface for an LED with V_f of 2.2 VDC and I_f of 20 mA.

Answer: In this example, we use a BeagleBone digital output pin with an I_{OH} value of 4 mA and a V_{OH} value 2.4 VDC. A loop equation, which includes these parameters, may be written as:

$$V_{OH} = (I_B \times R_B) + V_{BE}$$

Also, transistor collector current (I_c) is related to transistor base current (I_b) by the value of beta (b).

$$I_c = If = b \times Ib$$

The transistor V_{BE} is typically 0.7 VDC. Therefore, all parameters are known except R_B. Solving for R_B yields a value of 8500 Ohms. We use a value of 10 k Ohms. In this interface configuration, resistor R_C is chosen as in previous examples to safely limit the forward LED current to prescribed values. A loop equation may be written that includes R_C:

$$V_{CC} - (I_f \times R_C) - V_f - V_{CE(sat)} = 0$$

A typical value for $V_{CE(sat)}$ is 0.2 VDC. All equation values are now known except R_C. The equation may be solved yielding an R_C value of 130 Ohms.

4.4.2 SEVEN-SEGMENT LED DISPLAYS

To display numeric data, seven-segment LED displays are available, as shown in Figure 4.17b. Different numerals can be displayed by asserting the proper LED segments. For example, to display the number five, segments a, c, d, f, and g would be illuminated; see Figure 4.17a. Seven-segment displays are available in common cathode (CC) and common anode (CA) configurations. As the CC designation implies, all seven individual LED cathodes on the display are tied together.

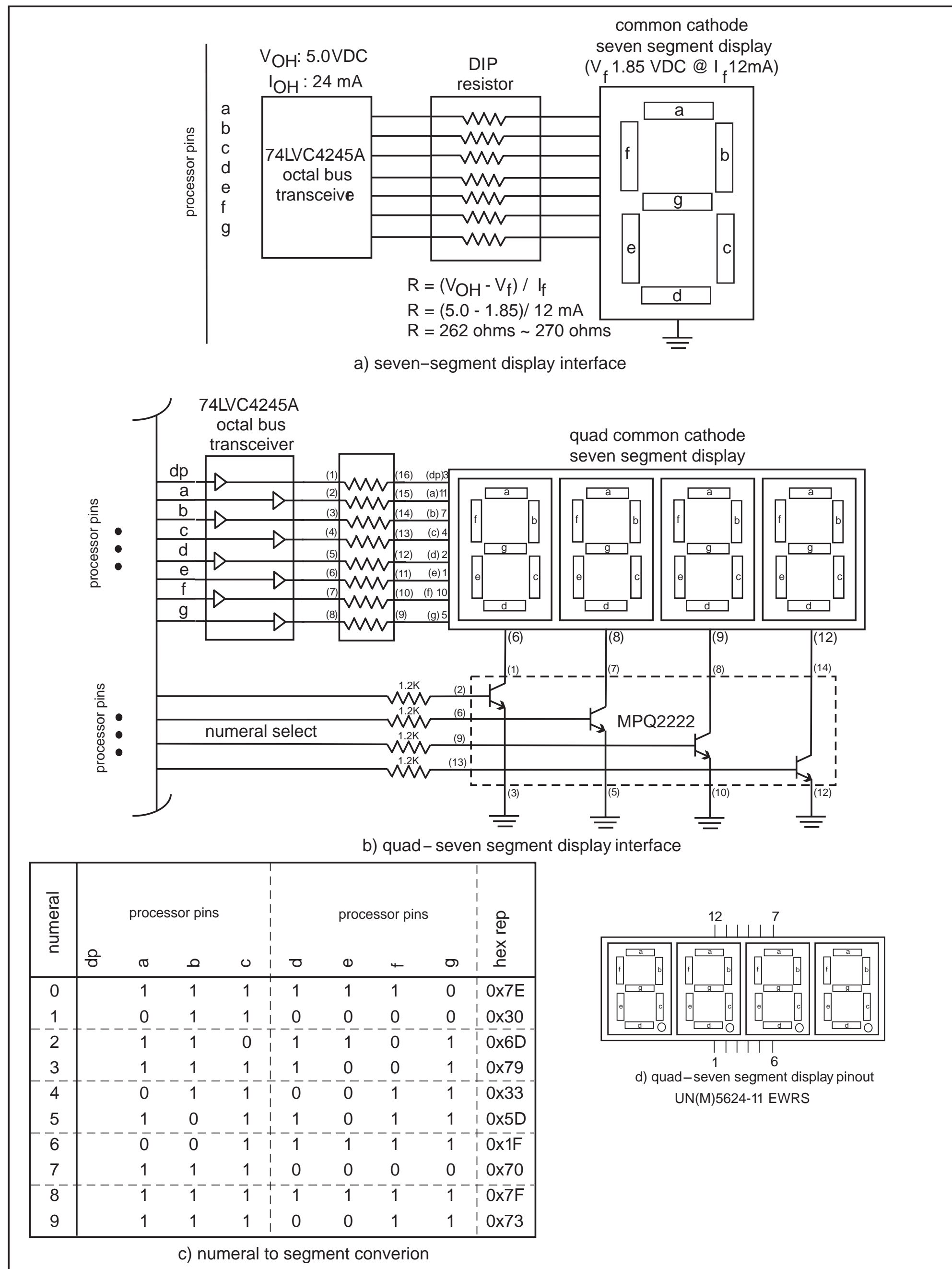
As shown in Figure 4.17b, an interface circuit is required between the processor and the seven-segment LED. We use a 74LVC4245A octal bus transceiver circuit to translate the 3.3 VDC output from the processor up to 5 VDC and also provide a maximum I_{OH} value of 24 mA. A limiting resistor is required for each segment to limit the current to a safe value for the LED. Conveniently, resistors are available in DIP packages of eight for this type of application.

Seven-segment displays are available in multi-character panels. In this case, separate processor pins are not used to provide data to each seven-segment character. Instead, a single port equivalent (8 pins) is used to provide character data. Several other pins are used to sequence through each of the characters, as shown in Figure 4.17b. An NPN (for a CC display) transistor is connected to the common cathode connection of each individual character. As the base contact of each transistor is sequentially asserted, the specific character is illuminated. If the processor sequences through the display characters at a rate greater than 30 Hz, the display will appear to be steady and will not flicker.

4.4.3 TRI-STATE LED INDICATOR

A tri-state LED indicator is shown in Figure 4.18. It may be used to provide the status of many processor pins simultaneously. The indicator bank consists of eight green and eight red LEDs. When an individual processor pin is logic high the green LED is illuminated. When at logic low, the red LED is illuminated. If the port pin is at a tri-state, high impedance state, no LED is illuminated. Tri-state logic is used to connect a number of devices to a common bus. When a digital circuit is placed in the Hi-z (high impedance) state it is electrically isolated from the bus.

The NPN/PNP transistor pair at the bottom of the figure provides a 2.5 VDC voltage reference for the LEDs. When a specific processor pin is logic high, the green LED will be forward biased since its anode will be at a higher potential than its cathode. The 47 Ohm resistor



numeral	dp	a	b	c	d	e	f	g	hex rep
		processor pins			processor pins				
0		1	1	1	1	1	1	0	0x7E
1		0	1	1	0	0	0	0	0x30
2		1	1	0	1	1	0	1	0x6D
3		1	1	1	1	0	0	1	0x79
4		0	1	1	0	0	1	1	0x33
5		1	0	1	1	0	1	1	0x5D
6		0	0	1	1	1	1	1	0x1F
7		1	1	1	0	0	0	0	0x70
8		1	1	1	1	1	1	1	0x7F
9		1	1	1	0	0	1	1	0x73

c) numeral to segment converion

Figure 4.17: LED display devices.

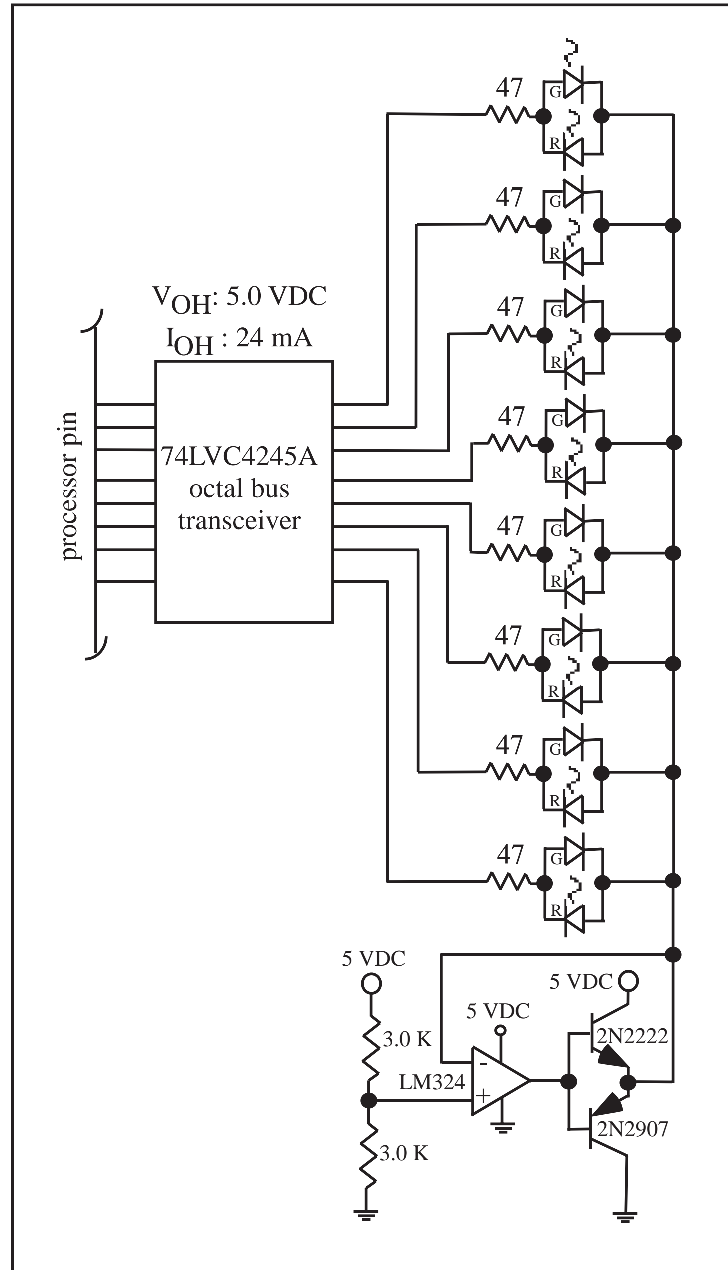


Figure 4.18: Tri-state LED display.

limits current to a safe value for the LED. Conversely, when a specific processor pin is at a logic low (0 VDC), the red LED will be forward biased and illuminate. For clarity, the red and green LEDs are shown as being separate devices. LEDs are available that have both LEDs in the same device. The 74LVC4245A octal bus transceiver translates the output voltage of the processor from 3.3–5.0 VDC.

4.4.4 DOT MATRIX DISPLAY

The dot matrix display consists of a large number of LEDs configured in a single package. A typical 5 × 7 LED arrangement is a matrix of five columns of LEDs with seven LEDs per row, as shown in Figure 4.19. Display data for a single matrix column [R6-R0] is provided by the processor. That specific row is then asserted by the processor using the column select lines [C2-C0]. The entire display is sequentially illuminated a column at a time. If the processor sequences through each column fast enough (greater than 30 Hz), the matrix display appears to be stationary to a viewer.

In Figure 4.19, we have provided the basic configuration for the dot matrix display for a single display device. However, this basic idea can be expanded in both dimensions to provide a multi-character, multi-line display. A larger display does not require a significant number of processor pins for the interface. The dot matrix display may be used to display alphanumeric data as well as graphics data. Several manufacturers provide 3.3 VDC compatible dot matrix displays with integrated interface and control circuitry.

4.4.5 LIQUID CRYSTAL DISPLAY (LCD)

An LCD is an output device to display text information, as shown in Figure 4.20. LCDs come in a wide variety of configurations including multi-character, multi-line format. A 16 × 2 LCD format is common. That is, it has the capability of displaying two lines of 16 characters each. The characters are sent to the LCD via American Standard Code for Information Interchange (ASCII) format a single character at a time. For a parallel-configured LCD, an eight-bit data path and two lines are required between the processor and the LCD, as shown in Figure 4.20a. Many parallel configured LCDs may also be configured for a four bit data path this saving several processor pins. A small microcontroller mounted to the back panel of the LCD translates the ASCII data characters and control signals to properly display the characters. Several manufacturers provide 3.3 VDC compatible displays.

To conserve processor input/output pins, a serial configured LCD may be used. A serial LCD reduces the number of required processor pins for interface, from ten down to one, as shown in Figure 4.20b. Display data and control information is sent to the LCD via an asynchronous serial communication link (8 bit, 1 stop bit, no parity, 9600 Baud). A serial configured LCD costs slightly more than a similarly configured parallel LCD.

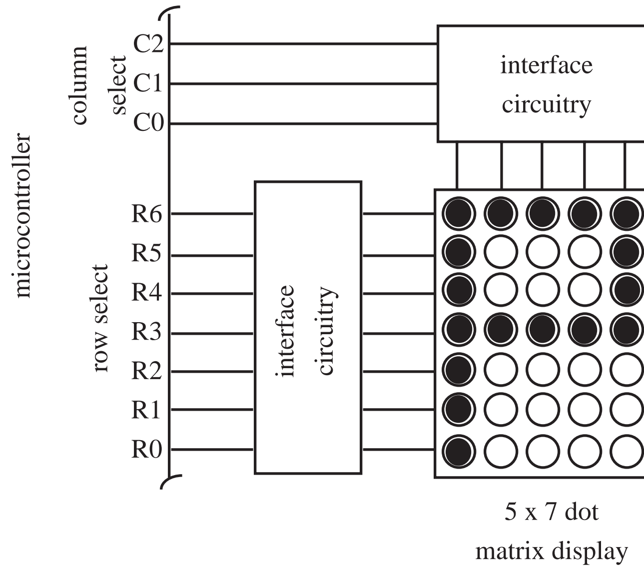


Figure 4.19: Dot matrix display.

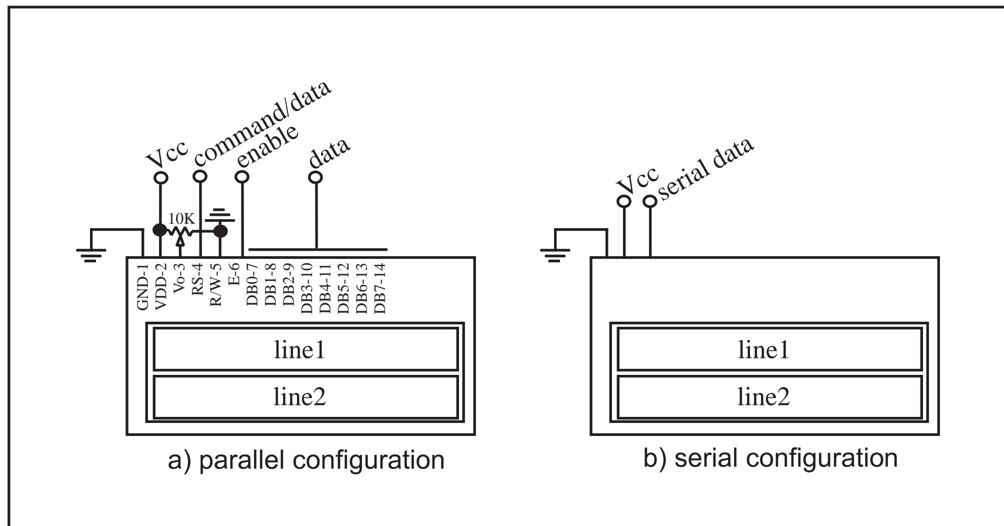


Figure 4.20: LCD display with a) parallel interface and b) serial interface.

4.5 HIGH-POWER INTERFACES

Processors are frequently used to control a variety of high power AC and DC devices. In this section, we discuss interface techniques for a wide variety of these devices.

4.5.1 HIGH-POWER DC DEVICES

A number of direct current devices may be controlled with an electronic switching device such as a MOSFET (metal oxide semiconductor field effect transistor). Specifically, an N-channel enhancement MOSFET may be used to switch a high current load (such as a motor) on or off using a low current control signal from a processor, as shown in Figure 4.21. The low current control signal from the processor is connected to the gate of the MOSFET via a MOSFET driver. As shown in Figure 4.21, an LTC 1157 MOSFET driver is used to boost the control signal from the processor to be compatible with an IRLR024 power MOSFET. The IRLR024 is rated at 60 VDC V_{DS} and a continuous drain current I_D of 14 amps. The IRLR024 MOSFET switches the high current load on and off consistent with the control signal. In a low-side connection, the high current load is connected between the MOSFET source and ground.

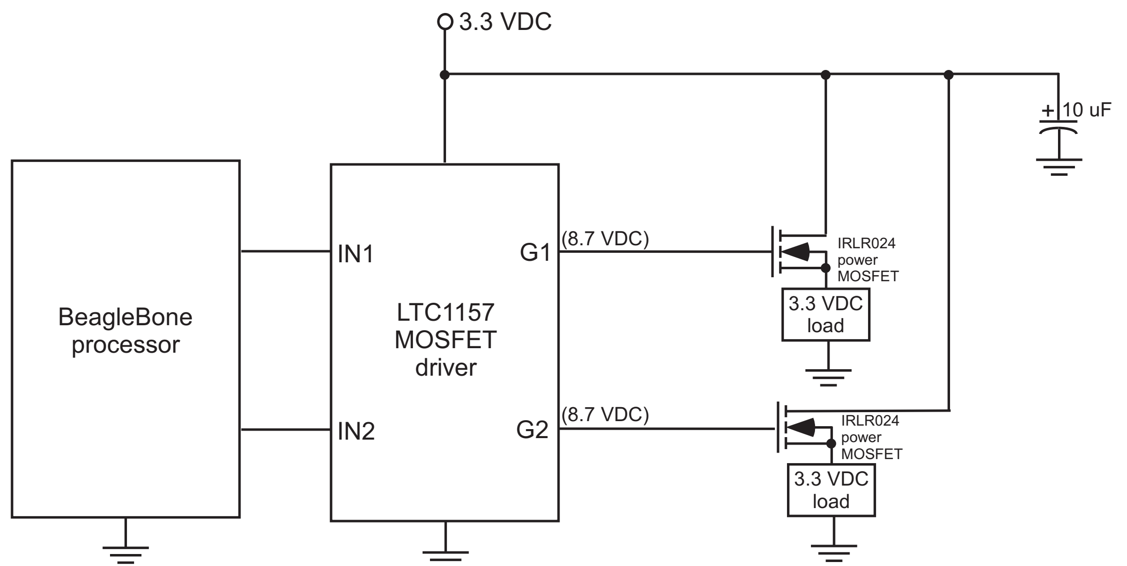


Figure 4.21: MOSFET drive circuit. (Adapted from Linear Technology.)

4.5.2 DC MOTOR SPEED AND DIRECTION CONTROL

There are a wide variety of DC motor types that can be controlled by a processor. To properly interface a motor to the processor, we must be familiar with the different types of motor technologies. Motor types are illustrated in Figure 4.22.

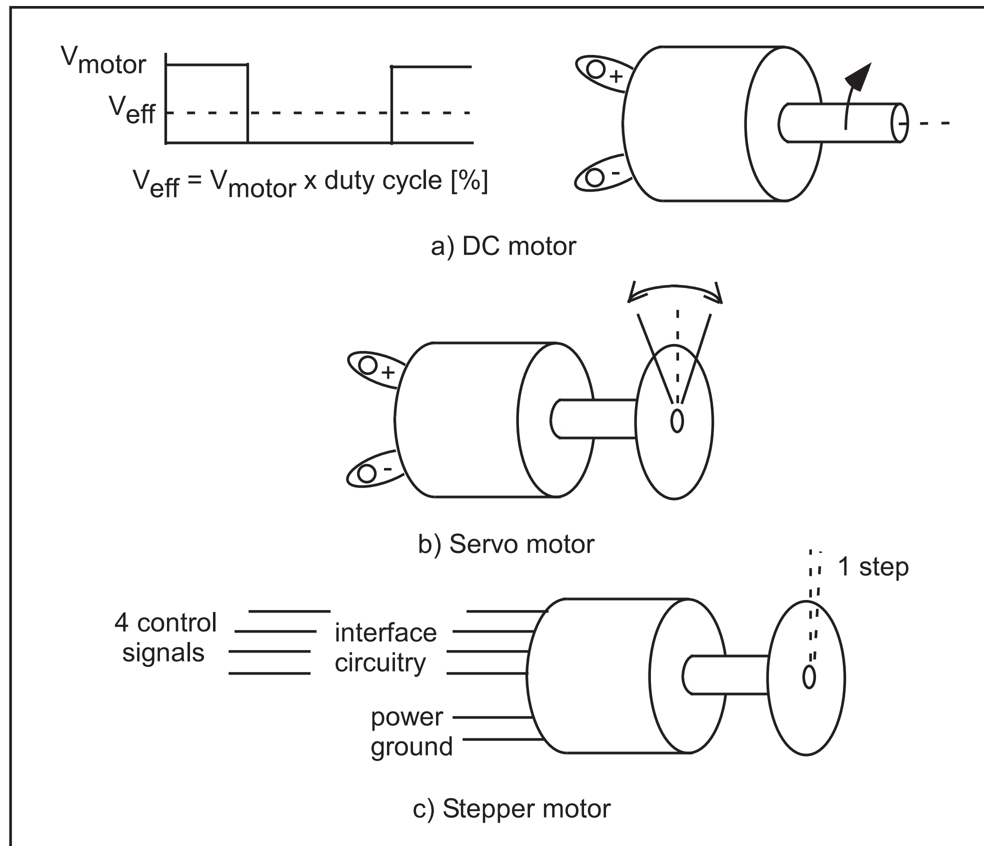


Figure 4.22: Motor types.

General categories of DC motor types include the following.

- **DC motor:** A DC motor has a positive and negative terminal. When a DC power supply of suitable current rating is applied to the motor, it will rotate. If the polarity of the supply is switched with reference to the motor terminals, the motor will rotate in the opposite direction. The speed of the motor is roughly proportional to the applied voltage up to the rated voltage of the motor.

- **Servo motor:** A servo motor provides a precision angular rotation for an applied pulse width modulation (PWM) duty cycle. As the duty cycle of the applied signal is varied, the angular displacement of the motor also varies. This type of motor is used to change mechanical positions such as the steering angle of a wheel.

- **Stepper motor:** A stepper motor, as its name implies, provides an incremental step change in rotation (typically 2.5° per step) for a step change in control signal sequence. Stepper motors are available with either a two- or four-wire interface. For the four-wire stepper motor, the processor provides a four bit control sequence to rotate the motor clockwise. To turn the motor counterclockwise, the control sequence is reversed. The low-power control signals are interfaced to the motor via MOSFETs or power transistors to provide for the proper voltage and current requirements of the pulse sequence. The stepper motor is used for precise positioning of mechanical components.

4.5.3 DC MOTOR OPERATING PARAMETERS

As previously mentioned, DC motor speed may be varied by changing the applied voltage. This is difficult to do with a digital control signal. However, pulse width modulation (PWM) techniques combined with a MOSFET interface circuit may be used to precisely control motor speed. The duty cycle of the PWM signal governs the percentage of the power supply voltage applied to the motor and hence the percentage of rated full speed at which the motor will rotate. The interface circuit to accomplish this type of control is shown in Figure 4.23. It is a slight variation of the control circuit provided in Figure 4.21. In this configuration, the motor supply voltage may be different than the processor's 3.3 VDC supply. For an inductive load, a reverse biased protection diode is provided across the load. The interface circuit shown allows the motor to rotate in a given direction.

4.5.4 H-BRIDGE DIRECTION CONTROL

For a DC motor to operate in both the clockwise and counter clockwise directions, the polarity of the DC motor supplied must be changed. To operate the motor in the forward direction, the positive battery terminal must be connected to the positive motor terminal while the negative battery terminal must be attached to the negative motor terminal. To reverse the motor direction, the motor supply polarity must be reversed. An H-bridge is a circuit employed to perform this polarity switch.

An H-bridge may be constructed from discrete components, as shown in Figure 4.24. The transistors Q1, Q2, Q3, and Q4 form an H-bridge. When transistors Q1 and Q4 are on, current flows from the positive terminal to the negative terminal of the motor winding. When transistors Q2 and Q3 are on, the polarity of the current is reversed, causing the motor to rotate in the opposite direction. When transistors Q3 and Q4 are on, the motor does not rotate and is in the brake state.

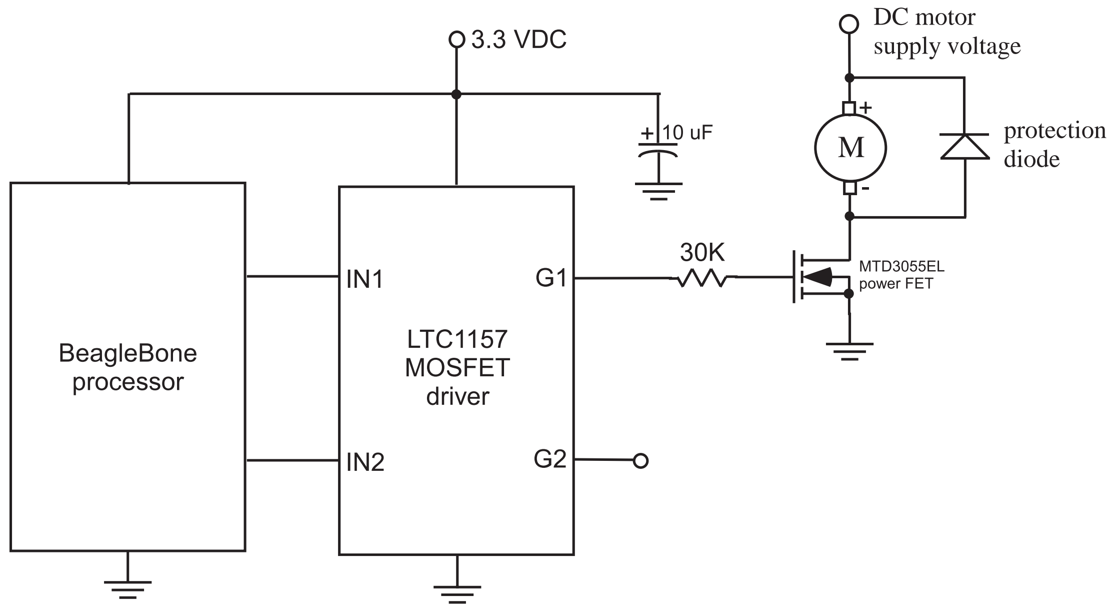


Figure 4.23: DC motor interface.

If PWM signals are used to drive the base of the transistors, both motor speed and direction may be controlled by the circuit. The transistors used in the circuit must have a current rating sufficient to handle the current requirements of the motor during start and stall conditions.

Texas Instruments provides a self-contained H-bridge motor controller integrated circuit, the DRV8829. Within the DRV8829 package is a single H-bridge driver. The driver may control DC loads with supply voltages from 8–45 VDC with a peak current rating of 5 amps. The single H-bridge driver may be used to control a DC motor or one winding of a bipolar stepper motor [DRV8829].

4.5.5 DC SOLENOID CONTROL

The interface circuit for a DC solenoid is shown in Figure 4.25. A solenoid is used to activate a mechanical insertion (or extraction). As in previous examples, we employ the LTC1157 MOSFET driver between the processor and the power MOSFET used to activate the solenoid. A reverse biased diode is placed across the solenoid. Both the solenoid power supply and the MOSFET must have the appropriate voltage and current rating to support the solenoid requirements.

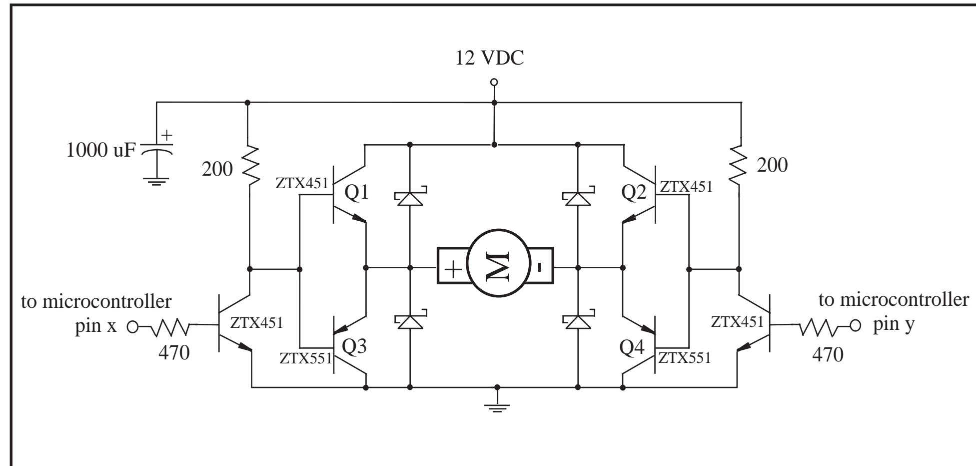


Figure 4.24: H-bridge control circuit.

4.5.6 STEPPER MOTOR CONTROL

Stepper motors are used to provide a discrete angular displacement in response to a control signal step. There is a wide variety of stepper motors including bipolar and unipolar types with different configurations of motor coil wiring. Due to space limitations we only discuss the unipolar, five wire stepper motor. The internal coil configuration for this motor is shown in Figure 4.26b.

Often, a wiring diagram is not available for the stepper motor. Based on the wiring configuration (see Figure 4.26b), one can find out the common line for both coils. It has a resistance that is one-half of all of the other coils. Once the common connection is found, one can connect the stepper motor into the interface circuit. By changing the other connections, one can determine the correct connections for the step sequence. To rotate the motor either clockwise or counter clockwise, a specific step sequence must be sent to the motor control wires, as shown in Figure 4.26b.

The processor does not have sufficient capability to drive the motor directly. Therefore, an interface circuit is required, as shown in Figure 4.27. The speed of motor rotation is determined by how fast the control sequence is completed. Stepper motor interfaces are available from a number of sources. An open source hardware line of stepper motor drivers, called "EasyDriver," is available from `www.schmalzhaus.com/EasyDriver/`.

4.5.7 OPTICAL ISOLATION

When designing the interface between a controller and a motor it is a good design practice to provide optical isolation. An optical isolator (e.g., 4N25) consists of an LED and optical transistor in a common package. The LED is driven by the controller, whereas the optical transistor provides

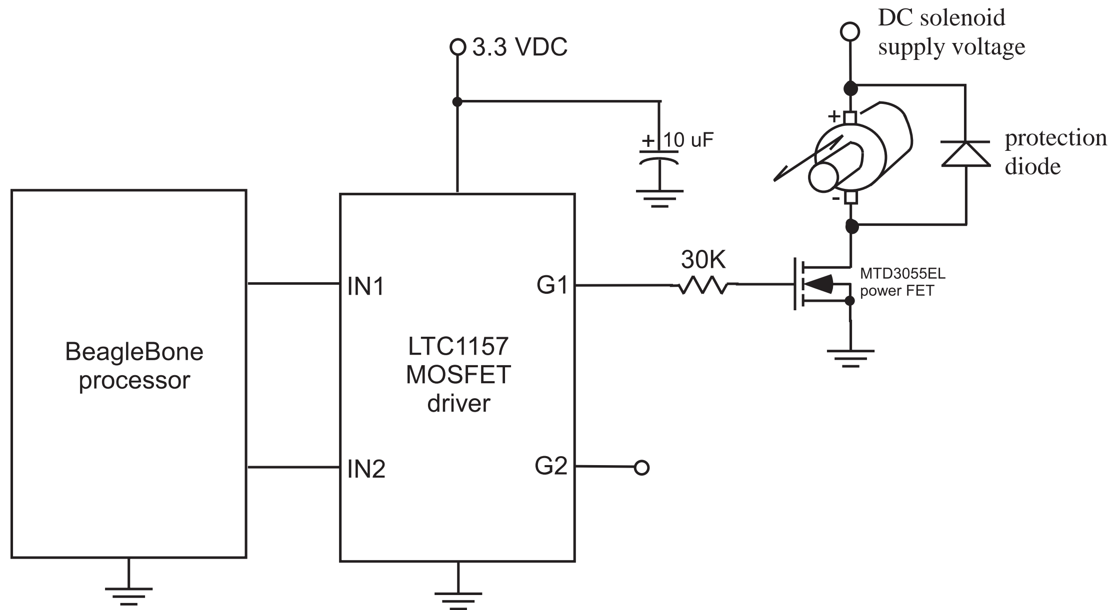


Figure 4.25: Solenoid interface circuit.

the control signal to the motor interface circuit. The link between the processor to the motor interface circuit is now provided by light rather than an electrical link. This provides a high level of noise isolation between the processor and the motor interface circuit. Many optical isolators provide a signal inversion.

4.6 INTERFACING TO MISCELLANEOUS DEVICES

In this section, we present a potpourri of interface circuits to connect a processor to a wide variety of peripheral devices.

4.6.1 SONALERTS, BEEPERS, BUZZERS

In Figure 4.28, we provide several circuits used to interface a processor to a buzzer, beeper or other types of annunciator devices such as a sonalert. It is important that the interface transistor and the supply voltage are matched to the requirements of the sound producing device.

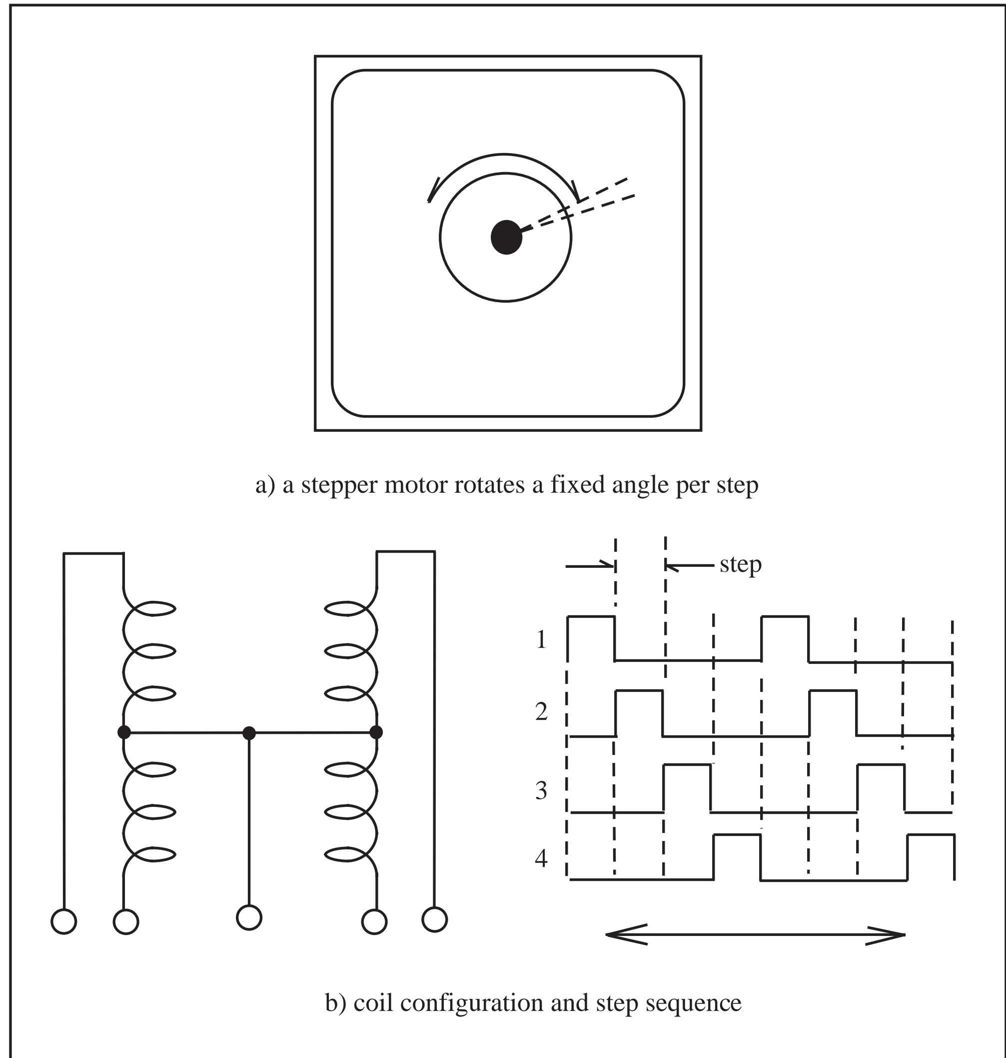


Figure 4.26: Unipolar stepper motor.

4.6.2 VIBRATING MOTOR

A vibrating motor is often used to gain one's attention as in a cell phone. These motors are typically rated at 3 VDC and a high current. The interface circuit shown in Figure 4.21 is used to drive the low-voltage motor.

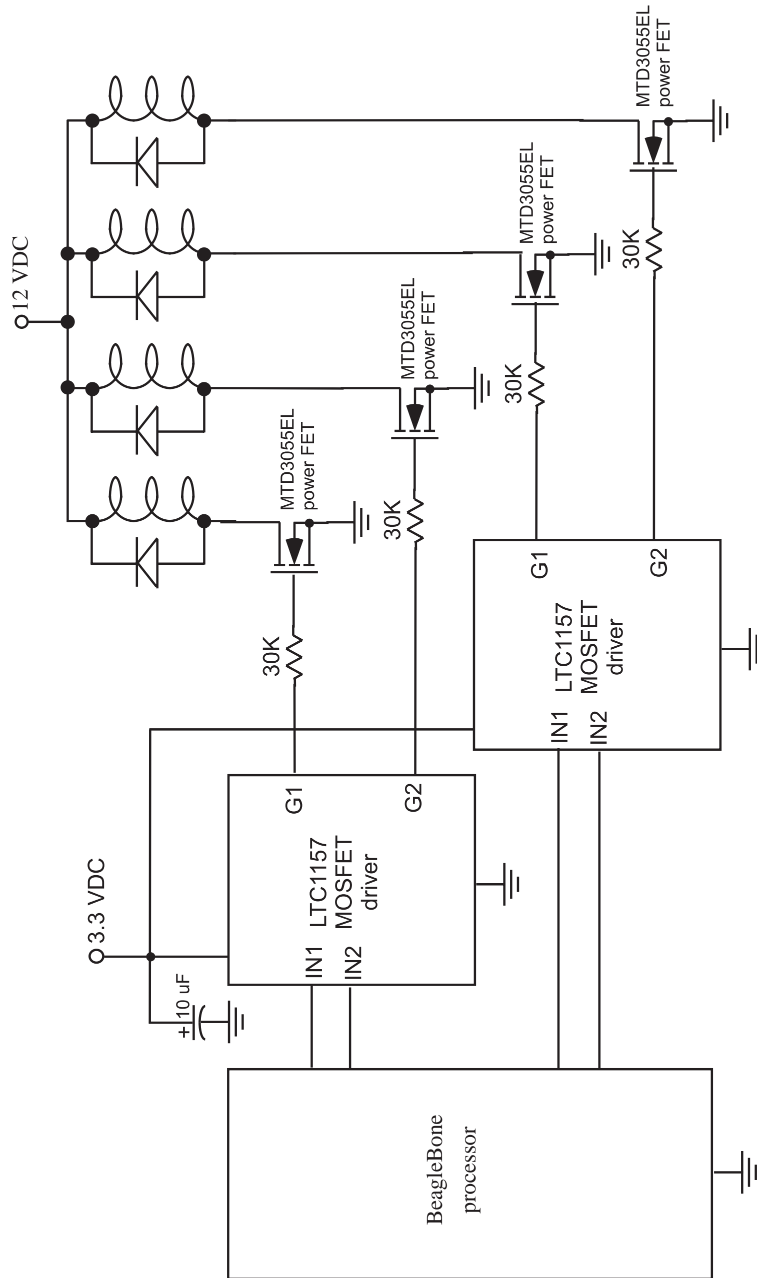


Figure 4.27: Unipolar stepper motor interface circuit.

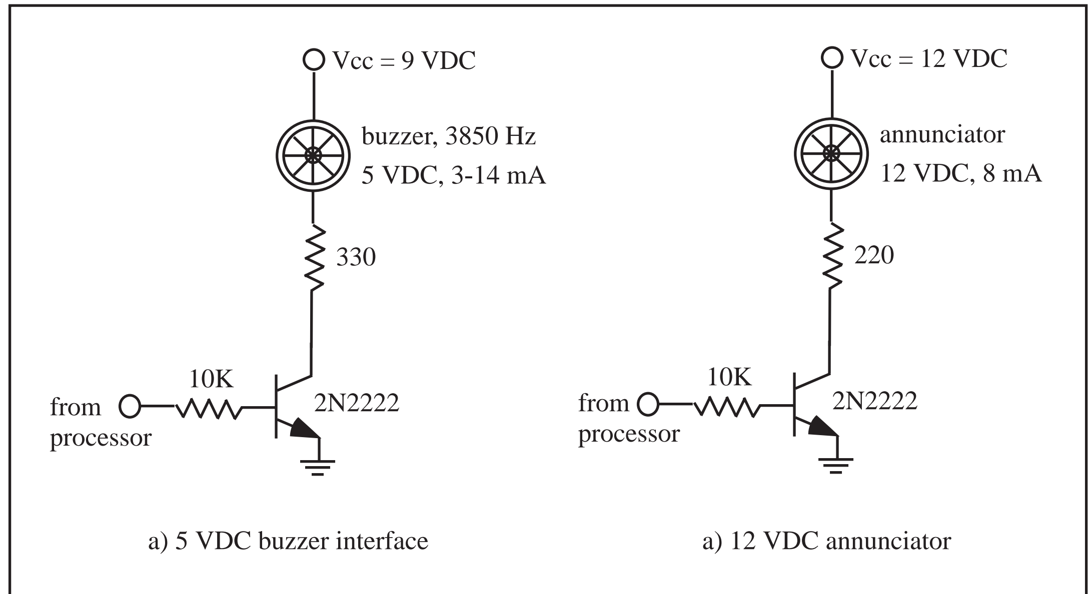


Figure 4.28: Sonalert, beepers, buzzers.

4.6.3 DC FAN

The interface circuit provided in Figure 4.23 may also be used to control a DC fan. As before, a reverse biased diode is placed across the DC fan motor.

4.6.4 BILGE PUMP

A bilge pump is a pump specifically designed to remove water from the inside of a boat. The pumps are powered from a 12 VDC source and have typical flow rates from 360 to over 3,500 gallons per minute. They range in price from US $20–80 [`www.shorelinemarinedevelopment.com`]. An interface circuit to control a bilge pump from BeagleBone Black is provided in Figure 4.29. The interface circuit consists of a 470 ohm resistor, a power NPN Darlington transistor (TIP 120) and a 1N4001 diode. The 12 VDC supply should have sufficient current capability to supply the needs of the bilge pump.

4.7 AC DEVICES

A high-power alternating current (AC) load may be switched on and off using a low-power control signal from the processor. In this case, a Solid State Relay is used as the switching device. Solid state relays are available to switch a high-power DC or AC load [Crydom, 2015]. For example, the Crydom 558-CX240D5R is a printed circuit board mounted, air-cooled, single-pole single-throw

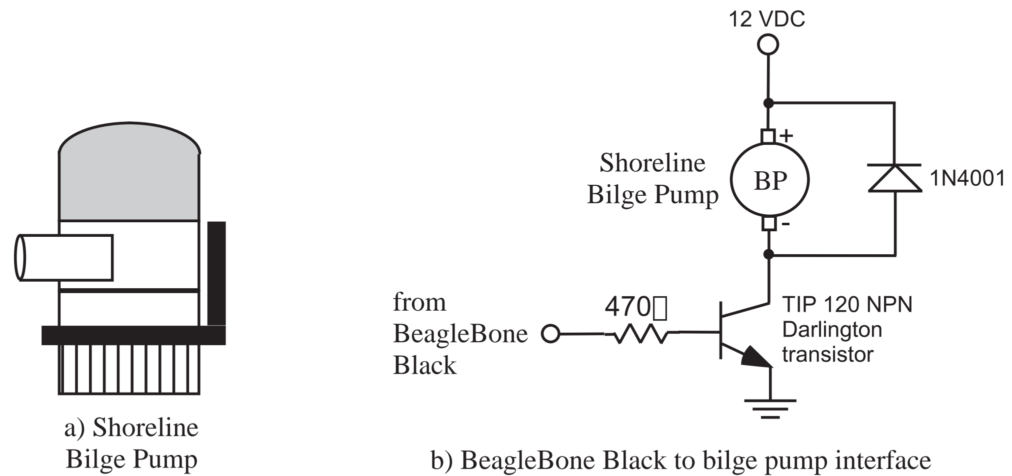


Figure 4.29: Bilge pump interface.

(SPST), normally open (NO) solid state relay. It requires a DC control voltage of 3–15 VDC at 15 mA. However, this small processor compatible DC control signal is used to switch 12–280 VAC loads rated from 0.06–5 amps [Crydom, 2015].

To vary the direction of an AC motor, you must use a bi-directional AC motor. A bi--directional motor is equipped with three terminals: common, clockwise, and counterclockwise. To turn the motor clockwise, an AC source is applied to the common and clockwise connections. In like manner, to turn the motor counterclockwise, an AC source is applied to the common and counterclockwise connections. This may be accomplished using two of the Crydom SSRs.

PowerSwitch manufacturers an easy-to-use AC interface the PowerSwitch Tail II. The device consists of a control module with attached AC connections rated at 120 VAC, 15 A. The device to be controlled is simply plugged inline with the PowerSwitch Tail II. A digital control signal from BeagleBone Black (3 VDC at 3 mA) serves as the on/off control signal for the controlled AC device. The controlled signal is connected to the PowerSwitch Tail II via a terminal block connection. The PowerSwitch II is available as either normally closed (NC) or normally open (NO) [`www.powerswitchtail.com`].

4.8 APPLICATION 1: EQUIPPING THE DAGU MAGICIAN ROBOT WITH A LCD

An LCD is a useful development and diagnostic tool during project development. In this section we add an LCD to the Dagu Magician robot. A low-cost, 3.3 VDC, 16 character by two line LCD with a parallel connection (Sparkfun LCD-09025) is interfaced to the robot. The LCD support functions and hardware interface is provided in Figure 4.30.

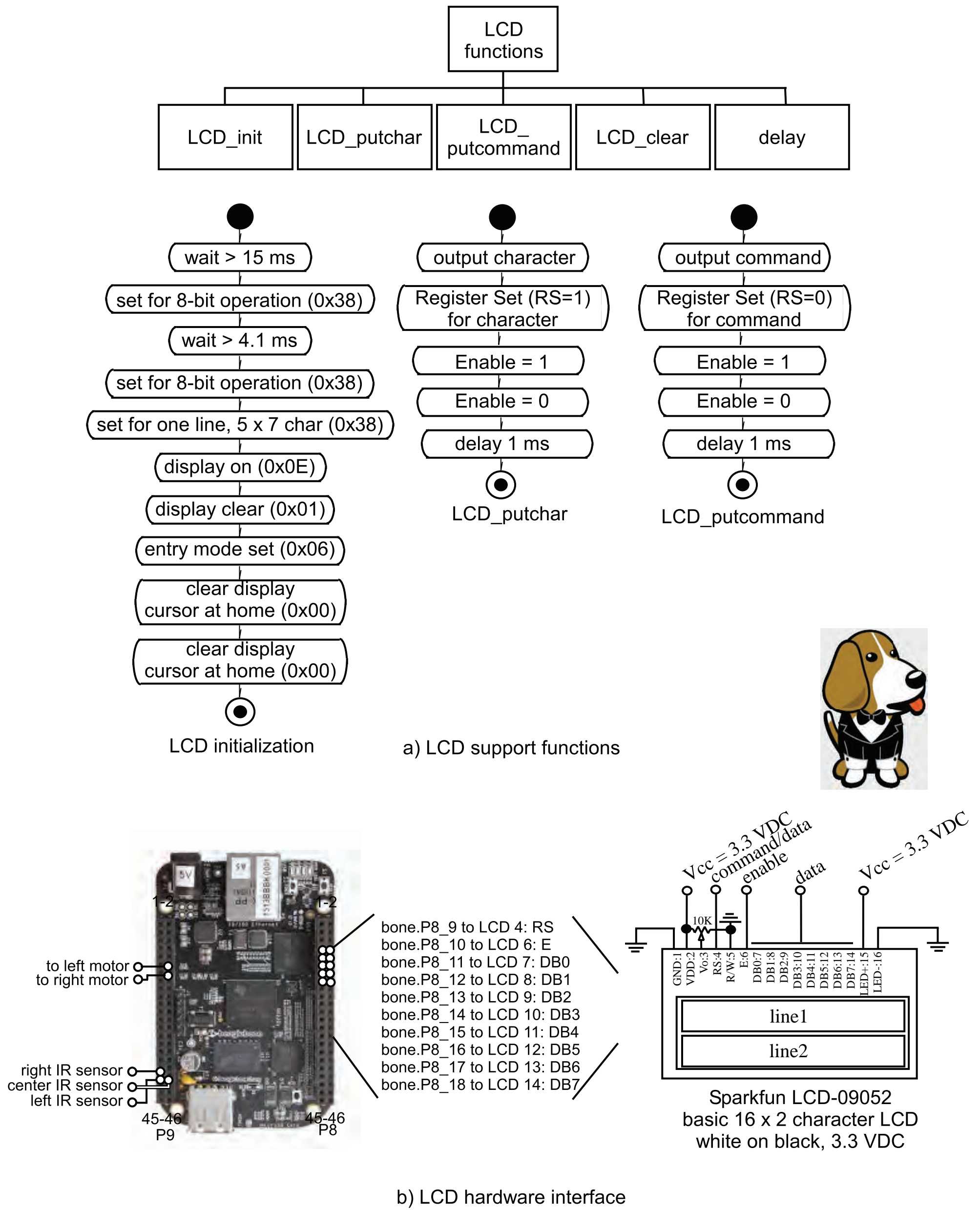


Figure 4.30: LCD for Dagu Magician robot. (Illustrations used with permission of Texas Instruments (www.TI.com).)

There are several functions required to use the LCD including:

- LCD initialization (LCD_init): initializes LCD to startup state as specified by LCD technical data;
- LCD put character (LCD_putchar): sends a specified ASCII character to the LCD display;
- LCD print string (LCD_print): sends a string of ASCII characters to the LCD display until the null character is received; and
- LCD put command (LCD_putcommand): sends a command to the LCD display.

The UML activity diagrams (UML compliant flow charts) for these functions are provided in Figure 4.30a.

The ASCII characters and commands are sent to the LCD via an 8-bit parallel data connection and two control lines (Register Set (RS) and Enable (E)), as shown in Figure 4.30b.

Provided below is the Bonescript code to configure the LCD and the code for the support functions.

```
1 //******************************************************************
2 //******************************************************************
3
4 var b = require ('bonescript');
5
6 var LCD_RS   = ''P8_9'';               //LCD Register Set (RS) control
7 var LCD_E    = ''P8_10'';              //LCD Enable (E) control
8 var LCD_DB0 = ''P8_11'';               //LCD Data line DB0
9 var LCD_DB1 = ''P8_12'';               //LCD Data line DB1
10 var LCD_DB2 = ''P8_13'';              //LCD Data line DB2
11 var LCD_DB3 = ''P8_14'';              //LCD Data line DB3
12 var LCD_DB4 = ''P8_15'';              //LCD Data line DB4
13 var LCD_DB5 = ''P8_16'';              //LCD Data line DB5
14 var LCD_DB6 = ''P8_17'';              //LCD Data line DB6
15 var LCD_DB7 = ''P8_18'';              //LCD Data line DB7
16
17 var counter = 1;                      //counter to be displayed
18
19 b.pinMode(LCD_RS,  b.OUTPUT);         //set pin to digital output
20 b.pinMode(LCD_E,   b.OUTPUT);         //set pin to digital output
21 b.pinMode(LCD_DB0, b.OUTPUT);         //set pin to digital output
22 b.pinMode(LCD_DB1, b.OUTPUT);         //set pin to digital output
23 b.pinMode(LCD_DB2, b.OUTPUT);         //set pin to digital output
24 b.pinMode(LCD_DB3, b.OUTPUT);         //set pin to digital output
25 b.pinMode(LCD_DB4, b.OUTPUT);         //set pin to digital output
26 b.pinMode(LCD_DB5, b.OUTPUT);         //set pin to digital output
27 b.pinMode(LCD_DB6, b.OUTPUT);         //set pin to digital output
28 b.pinMode(LCD_DB7, b.OUTPUT);         //set pin to digital output
29 LCD_init(LCD_update);                 //call LCD initialize
```

```
30
31 //*********************************************************************
32 //LCD_update
33 //*********************************************************************
34
35 function LCD_update()
36 {
37 LCD_putchar(counter, next);          //write 'counter' value to LCD
38
39 //When LCD_putchar completes, schedule the next run of it
40 function next()
41   {
42   counter++;                         //update counter
43   if(counter > 9)                    //Re-init after 9
44     {
45     counter = 1;
46     LCD_init(LCD_update);
47     }
48   else
49     setTimeout(LCD_update, 500);     //update again in 500 ms
50   }
51 }
52
53 //*********************************************************************
54 //LCD_init
55 //*********************************************************************
56
57 function LCD_init(callback)
58 {
59 //LCD Enable (E) pin low
60 b.digitalWrite(LCD_E, b.LOW);
61
62 //Start at the beginning of the list of steps to perform
63 var i = 0;
64
65 //List of steps to perform
66 var steps =
67   [
68   function(){ setTimeout(next, 15); },               //delay 15ms
69   function(){ LCD_putcommand(0x38, next); },      //set for 8-bit
        operation
70   function(){ setTimeout(next, 5); },                //delay 5ms
71   function(){ LCD_putcommand(0x38, next); },      //set for 8-bit
        operation
72   function(){ LCD_putcommand(0x38, next); },      //set for 5 x 7
        character
73   function(){ LCD_putcommand(0x0E, next); },      //display on
74   function(){ LCD_putcommand(0x01, next); },      //display clear
```

```
75    function(){ LCD_putcommand(0x06, next); },      //entry mode set
76    function(){ LCD_putcommand(0x00, next); },      //clear display, cursor
          home
77    function(){ LCD_putcommand(0x00, callback); } //clear display, cursor
          home
78    ];
79
80 next(); //Execute the first step
81
82 //Function for executing the next step
83 function next()
84    {
85    i++;
86    steps[i-1]();
87    }
88 }
89
90 //*****************************************************************
91 //LCD_putcommand
92 //*****************************************************************
93
94 function LCD_putcommand(cmd, callback)
95 {
96 //parse command variable into individual bits for output
97 //to LCD
98 if((cmd & 0x0080)== 0x0080) b.digitalWrite(LCD_DB7, b.HIGH);
99    else b.digitalWrite(LCD_DB7, b.LOW);
100 if((cmd & 0x0040)== 0x0040) b.digitalWrite(LCD_DB6, b.HIGH);
101    else b.digitalWrite(LCD_DB6, b.LOW);
102 if((cmd & 0x0020)== 0x0020) b.digitalWrite(LCD_DB5, b.HIGH);
103    else b.digitalWrite(LCD_DB5, b.LOW);
104 if((cmd & 0x0010)== 0x0010) b.digitalWrite(LCD_DB4, b.HIGH);
105    else b.digitalWrite(LCD_DB4, b.LOW);
106 if((cmd & 0x0008)== 0x0008) b.digitalWrite(LCD_DB3, b.HIGH);
107    else b.digitalWrite(LCD_DB3, b.LOW);
108 if((cmd & 0x0004)== 0x0004) b.digitalWrite(LCD_DB2, b.HIGH);
109    else b.digitalWrite(LCD_DB2, b.LOW);
110 if((cmd & 0x0002)== 0x0002) b.digitalWrite(LCD_DB1, b.HIGH);
111    else b.digitalWrite(LCD_DB1, b.LOW);
112 if((cmd & 0x0001)== 0x0001) b.digitalWrite(LCD_DB0, b.HIGH);
113    else b.digitalWrite(LCD_DB0, b.LOW);
114
115 //LCD Register Set (RS) to logic zero for command input
116 b.digitalWrite(LCD_RS, b.LOW);
117 //LCD Enable (E) pin high
118 b.digitalWrite(LCD_E, b.HIGH);
119
120 //End the write after 1ms
```

```
121 setTimeout(endWrite, 1);
122
123 function endWrite()
124   {
125   //LCD Enable (E) pin low
126   b.digitalWrite(LCD_E, b.LOW);
127   //delay 1ms before calling 'callback'
128   setTimeout(callback, 1);
129   }
130 }
131
132 //**********************************************************************
133 //LCD_putchar
134 //**********************************************************************
135
136 function LCD_putchar(chr1, callback)
137 {
138 //Convert chr1 variable to UNICODE (ASCII)
139 var chr = chr1.toString().charCodeAt(0);
140
141 //parse character variable into individual bits for output
142 //to LCD
143 if((chr & 0x0080)== 0x0080) b.digitalWrite(LCD_DB7, b.HIGH);
144   else b.digitalWrite(LCD_DB7, b.LOW);
145 if((chr & 0x0040)== 0x0040) b.digitalWrite(LCD_DB6, b.HIGH);
146   else b.digitalWrite(LCD_DB6, b.LOW);
147 if((chr & 0x0020)== 0x0020) b.digitalWrite(LCD_DB5, b.HIGH);
148   else b.digitalWrite(LCD_DB5, b.LOW);
149 if((chr & 0x0010)== 0x0010) b.digitalWrite(LCD_DB4, b.HIGH);
150   else b.digitalWrite(LCD_DB4, b.LOW);
151 if((chr & 0x0008)== 0x0008) b.digitalWrite(LCD_DB3, b.HIGH);
152   else b.digitalWrite(LCD_DB3, b.LOW);
153 if((chr & 0x0004)== 0x0004) b.digitalWrite(LCD_DB2, b.HIGH);
154   else b.digitalWrite(LCD_DB2, b.LOW);
155 if((chr & 0x0002)== 0x0002) b.digitalWrite(LCD_DB1, b.HIGH);
156   else b.digitalWrite(LCD_DB1, b.LOW);
157 if((chr & 0x0001)== 0x0001) b.digitalWrite(LCD_DB0, b.HIGH);
158   else b.digitalWrite(LCD_DB0, b.LOW);
159
160 //LCD Register Set (RS) to logic one for character input
161 b.digitalWrite(LCD_RS, b.HIGH);
162 //LCD Enable (E) pin high
163 b.digitalWrite(LCD_E, b.HIGH);
164
165 //End the write after 1ms
166 setTimeout(endWrite, 1);
167
168 function endWrite()
```

```
169   {
170   //LCD Enable (E) pin low and call scheduleCallback when done
171   b.digitalWrite(LCD_E, b.LOW);
172   //delay 1ms before calling 'callback'
173   setTimeout(callback, 1);
174   }
175 }
176
177 //***********************************************************************
```

The following function allows a message to be sent to the LCD display. The function is called by indicating the LCD line to display the message on and the message.

```
 1 //***********************************************************************
 2
 3 var b = require('bonescript');
 4
 5 var LCD_RS  = ``P8_9'';                  //LCD Register Set (RS) control
 6 var LCD_E   = ``P8_10'';                 //LCD Enable (E) control
 7 var LCD_DB0 = ``P8_11'';                 //LCD Data line DB0
 8 var LCD_DB1 = ``P8_12'';                 //LCD Data line DB1
 9 var LCD_DB2 = ``P8_13'';                 //LCD Data line DB2
10 var LCD_DB3 = ``P8_14'';                 //LCD Data line DB3
11 var LCD_DB4 = ``P8_15'';                 //LCD Data line DB4
12 var LCD_DB5 = ``P8_16'';                 //LCD Data line DB5
13 var LCD_DB6 = ``P8_17'';                 //LCD Data line DB6
14 var LCD_DB7 = ``P8_18'';                 //LCD Data line DB7
15
16 b.pinMode(LCD_RS,  b.OUTPUT);            //set pin to digital output
17 b.pinMode(LCD_E,   b.OUTPUT);            //set pin to digital output
18 b.pinMode(LCD_DB0, b.OUTPUT);            //set pin to digital output
19 b.pinMode(LCD_DB1, b.OUTPUT);            //set pin to digital output
20 b.pinMode(LCD_DB2, b.OUTPUT);            //set pin to digital output
21 b.pinMode(LCD_DB3, b.OUTPUT);            //set pin to digital output
22 b.pinMode(LCD_DB4, b.OUTPUT);            //set pin to digital output
23 b.pinMode(LCD_DB5, b.OUTPUT);            //set pin to digital output
24 b.pinMode(LCD_DB6, b.OUTPUT);            //set pin to digital output
25 b.pinMode(LCD_DB7, b.OUTPUT);            //set pin to digital output
26 LCD_init(firstLine);                     //call LCD initialize
27 function firstLine()
28 {
29 LCD_print(1, "BeagleBone", nextLine);
30 }
31 function nextLine()
32 {
33 LCD_print(2, "Bonescript");
34 }
35
36 //***********************************************************************
```

```
37 //LCD_print
38 //***************************************************************
39
40 function LCD_print(line, message, callback)
41 {
42 var i = 0;
43
44 if(line == 1)
45   {
46   LCD_putcommand(0x80, writeNextCharacter);//print to LCD line 1
47   }
48 else
49   {
50   LCD_putcommand(0xc0, writeNextCharacter);//print to LCD line 2
51   }
52
53 function writeNextCharacter()
54   {
55   //if we already printed the last character, stop and callback
56   if(i == message.length)
57     {
58     if(callback) callback();
59     return;
60     }
61
62   //get the next character to print
63   var chr = message.substring(i, i+1);
64   i++;
65
66   //print it using LCD_putchar and come back again when done
67   LCD_putchar(chr, writeNextCharacter);
68   }
69 }
70
71 //***************************************************************
72 //LCD_init
73 //***************************************************************
74
75 function LCD_init(callback)
76 {
77 //LCD Enable (E) pin low
78 b.digitalWrite(LCD_E, b.LOW);
79
80 //Start at the beginning of the list of steps to perform
81 var i = 0;
82
83 //List of steps to perform
84 var steps =
```

```
85    [
86    function(){ setTimeout(next, 15); },              //delay 15ms
87    function(){ LCD_putcommand(0x38, next); },        //set for 8-bit
          operation
88    function(){ setTimeout(next, 5); },               //delay 5ms
89    function(){ LCD_putcommand(0x38, next); },        //set for 8-bit
          operation
90    function(){ LCD_putcommand(0x38, next); },        //set for 5 x 7
          character
91    function(){ LCD_putcommand(0x0E, next); },        //display on
92    function(){ LCD_putcommand(0x01, next); },        //display clear
93    function(){ LCD_putcommand(0x06, next); },        //entry mode set
94    function(){ LCD_putcommand(0x00, next); },        //clear display, cursor
           home
95    function(){ LCD_putcommand(0x00, callback); } //clear display, cursor
           home
96    ];
97
98  next(); //Execute the first step
99
100 //Function for executing the next step
101 function next()
102   {
103   i++;
104   steps[i-1]();
105   }
106 }
107
108 //*********************************************************************
109 //LCD_putcommand
110 //*********************************************************************
111
112 function LCD_putcommand(cmd, callback)
113 {
114 //parse command variable into individual bits for output
115 //to LCD
116 if((cmd & 0x0080)== 0x0080) b.digitalWrite(LCD_DB7, b.HIGH);
117   else b.digitalWrite(LCD_DB7, b.LOW);
118 if((cmd & 0x0040)== 0x0040) b.digitalWrite(LCD_DB6, b.HIGH);
119   else b.digitalWrite(LCD_DB6, b.LOW);
120 if((cmd & 0x0020)== 0x0020) b.digitalWrite(LCD_DB5, b.HIGH);
121   else b.digitalWrite(LCD_DB5, b.LOW);
122 if((cmd & 0x0010)== 0x0010) b.digitalWrite(LCD_DB4, b.HIGH);
123   else b.digitalWrite(LCD_DB4, b.LOW);
124 if((cmd & 0x0008)== 0x0008) b.digitalWrite(LCD_DB3, b.HIGH);
125   else b.digitalWrite(LCD_DB3, b.LOW);
126 if((cmd & 0x0004)== 0x0004) b.digitalWrite(LCD_DB2, b.HIGH);
127   else b.digitalWrite(LCD_DB2, b.LOW);
```

```
128 if((cmd & 0x0002)== 0x0002) b.digitalWrite(LCD_DB1, b.HIGH);
129   else b.digitalWrite(LCD_DB1, b.LOW);
130 if((cmd & 0x0001)== 0x0001) b.digitalWrite(LCD_DB0, b.HIGH);
131   else b.digitalWrite(LCD_DB0, b.LOW);
132
133 //LCD Register Set (RS) to logic zero for command input
134 b.digitalWrite(LCD_RS, b.LOW);
135 //LCD Enable (E) pin high
136 b.digitalWrite(LCD_E, b.HIGH);
137
138 //End the write after 1ms
139 setTimeout(endWrite, 1);
140
141 function endWrite()
142   {
143   //LCD Enable (E) pin low
144   b.digitalWrite(LCD_E, b.LOW);
145   //delay 1ms before calling 'callback'
146   setTimeout(callback, 1);
147   }
148 }
149
150 //*******************************************************************
151 //LCD_putchar
152 //*******************************************************************
153
154 function LCD_putchar(chr1, callback)
155 {
156 //Convert chr1 variable to UNICODE (ASCII)
157 var chr = chr1.toString().charCodeAt(0);
158
159 //parse character variable into individual bits for output
160 //to LCD
161 if((chr & 0x0080)== 0x0080) b.digitalWrite(LCD_DB7, b.HIGH);
162   else b.digitalWrite(LCD_DB7, b.LOW);
163 if((chr & 0x0040)== 0x0040) b.digitalWrite(LCD_DB6, b.HIGH);
164   else b.digitalWrite(LCD_DB6, b.LOW);
165 if((chr & 0x0020)== 0x0020) b.digitalWrite(LCD_DB5, b.HIGH);
166   else b.digitalWrite(LCD_DB5, b.LOW);
167 if((chr & 0x0010)== 0x0010) b.digitalWrite(LCD_DB4, b.HIGH);
168   else b.digitalWrite(LCD_DB4, b.LOW);
169 if((chr & 0x0008)== 0x0008) b.digitalWrite(LCD_DB3, b.HIGH);
170   else b.digitalWrite(LCD_DB3, b.LOW);
171 if((chr & 0x0004)== 0x0004) b.digitalWrite(LCD_DB2, b.HIGH);
172   else b.digitalWrite(LCD_DB2, b.LOW);
173 if((chr & 0x0002)== 0x0002) b.digitalWrite(LCD_DB1, b.HIGH);
174   else b.digitalWrite(LCD_DB1, b.LOW);
175 if((chr & 0x0001)== 0x0001) b.digitalWrite(LCD_DB0, b.HIGH);
```

```
176    else b.digitalWrite(LCD_DB0, b.LOW);
177
178 //LCD Register Set (RS) to logic one for character input
179 b.digitalWrite(LCD_RS, b.HIGH);
180 //LCD Enable (E) pin high
181 b.digitalWrite(LCD_E, b.HIGH);
182
183 //End the write after 1ms
184 setTimeout(endWrite, 1);
185
186 function endWrite()
187    {
188    //LCD Enable (E) pin low and call scheduleCallback when done
189    b.digitalWrite(LCD_E, b.LOW);
190    //delay 1ms before calling 'callback'
191    setTimeout(callback, 1);
192    }
193 }
194
195 //*****************************************************************
```

This next example provides load average for running processes or waiting on input/output averaged over 1, 5, and 15 min. The Linux kernel provides these numbers via reading /proc/loadavg. We then use the LCD writing routines already discussed to perform the display.

```
 1 //*****************************************************************
 2
 3 var b = require('bonescript');
 4 var fs = require('fs');
 5
 6 var LCD_RS  = ''P8_9'';            //LCD Register Set (RS) control
 7 var LCD_E   = ''P8_10'';           //LCD Enable (E) control
 8 var LCD_DB0 = ''P8_11'';           //LCD Data line DB0
 9 var LCD_DB1 = ''P8_12'';           //LCD Data line DB1
10 var LCD_DB2 = ''P8_13'';           //LCD Data line DB2
11 var LCD_DB3 = ''P8_14'';           //LCD Data line DB3
12 var LCD_DB4 = ''P8_15'';           //LCD Data line DB4
13 var LCD_DB5 = ''P8_16'';           //LCD Data line DB5
14 var LCD_DB6 = ''P8_17'';           //LCD Data line DB6
15 var LCD_DB7 = ''P8_18'';           //LCD Data line DB7
16
17 b.pinMode(LCD_RS,  b.OUTPUT);      //set pin to digital output
18 b.pinMode(LCD_E,   b.OUTPUT);      //set pin to digital output
19 b.pinMode(LCD_DB0, b.OUTPUT);      //set pin to digital output
20 b.pinMode(LCD_DB1, b.OUTPUT);      //set pin to digital output
21 b.pinMode(LCD_DB2, b.OUTPUT);      //set pin to digital output
22 b.pinMode(LCD_DB3, b.OUTPUT);      //set pin to digital output
23 b.pinMode(LCD_DB4, b.OUTPUT);      //set pin to digital output
24 b.pinMode(LCD_DB5, b.OUTPUT);      //set pin to digital output
```

```
25 b.pinMode(LCD_DB6, b.OUTPUT);          //set pin to digital output
26 b.pinMode(LCD_DB7, b.OUTPUT);          //set pin to digital output
27 LCD_init(firstLine);                   //call LCD initialize
28 function firstLine()
29 {
30 LCD_print(1, "BeagleBone Load", doUpdate);
31 }
32 function doUpdate()
33 {
34 fs.readFile("/proc/loadavg", writeUpdate);
35 }
36 function writeUpdate(err, data)
37 {
38 LCD_print(2, data.toString().substring(0, 14), onUpdate);
39 }
40 function onUpdate()
41 {
42 setTimeout(doUpdate, 1000);
43 }
44
45 //******************************************************************
46 //LCD_print
47 //******************************************************************
48
49 function LCD_print(line, message, callback)
50 {
51 var i = 0;
52
53 if(line == 1)
54   {
55   LCD_putcommand(0x80, writeNextCharacter);//print to LCD line 1
56   }
57 else
58   {
59   LCD_putcommand(0xc0, writeNextCharacter);//print to LCD line 2
60   }
61
62 function writeNextCharacter()
63   {
64   //if we already printed the last character, stop and callback
65   if(i == message.length)
66     {
67     if(callback) callback();
68     return;
69     }
70
71   //get the next character to print
72   var chr = message.substring(i, i+1);
```

```
73    i++;
74
75    //print it using LCD_putchar and come back again when done
76    LCD_putchar(chr, writeNextCharacter);
77    }
78 }
79
80 //******************************************************************
81 //LCD_init
82 //******************************************************************
83
84 function LCD_init(callback)
85 {
86 //LCD Enable (E) pin low
87 b.digitalWrite(LCD_E, b.LOW);
88
89 //Start at the beginning of the list of steps to perform
90 var i = 0;
91
92 //List of steps to perform
93 var steps =
94   [
95   function(){ setTimeout(next, 15); },                 //delay 15ms
96   function(){ LCD_putcommand(0x38, next); },     //set for 8-bit
       operation
97   function(){ setTimeout(next, 5); },                   //delay 5ms
98   function(){ LCD_putcommand(0x38, next); },     //set for 8-bit
       operation
99   function(){ LCD_putcommand(0x38, next); },     //set for 5 x 7
       character
100  function(){ LCD_putcommand(0x0E, next); },     //display on
101  function(){ LCD_putcommand(0x01, next); },     //display clear
102  function(){ LCD_putcommand(0x06, next); },     //entry mode set
103  function(){ LCD_putcommand(0x00, next); },     //clear display, cursor
        at home
104  function(){ LCD_putcommand(0x00, callback); } //clear display, cursor
        at home
105  ];
106
107 next(); //Execute the first step
108
109 //Function for executing the next step
110 function next()
111   {
112   i++;
113   steps[i-1]();
114   }
115 }
```

```
116
117 //************************************************************************
118 //LCD_putcommand
119 //************************************************************************
120
121 function LCD_putcommand(cmd, callback)
122 {
123 var chr = cmd;
124
125 //Start at the beginning of the list of steps to perform
126 var i = 0;
127
128 //List of steps to perform
129 var steps =
130   [
131 function() {
132 //parse character variable into individual bits for output
133 //to LCD
134 if((chr & 0x0080)== 0x0080) b.digitalWrite(LCD_DB7, b.HIGH, next);
135   else b.digitalWrite(LCD_DB7, b.LOW, next);
136 },function() {
137 if((chr & 0x0040)== 0x0040) b.digitalWrite(LCD_DB6, b.HIGH, next);
138   else b.digitalWrite(LCD_DB6, b.LOW, next);
139 },function() {
140 if((chr & 0x0020)== 0x0020) b.digitalWrite(LCD_DB5, b.HIGH, next);
141   else b.digitalWrite(LCD_DB5, b.LOW, next);
142 },function() {
143 if((chr & 0x0010)== 0x0010) b.digitalWrite(LCD_DB4, b.HIGH, next);
144   else b.digitalWrite(LCD_DB4, b.LOW, next);
145 },function() {
146 if((chr & 0x0008)== 0x0008) b.digitalWrite(LCD_DB3, b.HIGH, next);
147   else b.digitalWrite(LCD_DB3, b.LOW, next);
148 },function() {
149 if((chr & 0x0004)== 0x0004) b.digitalWrite(LCD_DB2, b.HIGH, next);
150   else b.digitalWrite(LCD_DB2, b.LOW, next);
151 },function() {
152 if((chr & 0x0002)== 0x0002) b.digitalWrite(LCD_DB1, b.HIGH, next);
153   else b.digitalWrite(LCD_DB1, b.LOW, next);
154 },function() {
155 if((chr & 0x0001)== 0x0001) b.digitalWrite(LCD_DB0, b.HIGH, next);
156   else b.digitalWrite(LCD_DB0, b.LOW, next);
157 },function() {
158 //LCD Register Set (RS) to logic zero for command input
159 b.digitalWrite(LCD_RS, b.LOW, next);
160 },function() {
161 //LCD Enable (E) pin high
162 b.digitalWrite(LCD_E, b.HIGH, next);
163 },function() {
```

```
164 //End the write after 1ms
165 setTimeout(next, 1);
166 },function() {
167   //LCD Enable (E) pin low and call scheduleCallback when done
168   b.digitalWrite(LCD_E, b.LOW, next);
169 },function() {
170   //delay 1ms before calling 'callback'
171   setTimeout(callback, 1);
172 }];
173
174 next(); //Execute the first step
175
176 //Function for executing the next step
177 function next()
178   {
179   i++;
180   steps[i-1]();
181   }
182 }
183
184 //******************************************************************
185 //LCD_putchar
186 //******************************************************************
187
188 function LCD_putchar(chr1, callback)
189 {
190 //Convert chr1 variable to UNICODE (ASCII)
191 var chr = chr1.toString().charCodeAt(0);
192
193 //Start at the beginning of the list of steps to perform
194 var i = 0;
195
196 //List of steps to perform
197 var steps =
198   [
199 function() {
200 //parse character variable into individual bits for output
201 //to LCD
202 if((chr & 0x0080)== 0x0080) b.digitalWrite(LCD_DB7, b.HIGH, next);
203   else b.digitalWrite(LCD_DB7, b.LOW, next);
204 },function() {
205 if((chr & 0x0040)== 0x0040) b.digitalWrite(LCD_DB6, b.HIGH, next);
206   else b.digitalWrite(LCD_DB6, b.LOW, next);
207 },function() {
208 if((chr & 0x0020)== 0x0020) b.digitalWrite(LCD_DB5, b.HIGH, next);
209   else b.digitalWrite(LCD_DB5, b.LOW, next);
210 },function() {
211 if((chr & 0x0010)== 0x0010) b.digitalWrite(LCD_DB4, b.HIGH, next);
```

```
212     else b.digitalWrite(LCD_DB4, b.LOW, next);
213 },function() {
214 if((chr & 0x0008)== 0x0008) b.digitalWrite(LCD_DB3, b.HIGH, next);
215     else b.digitalWrite(LCD_DB3, b.LOW, next);
216 },function() {
217 if((chr & 0x0004)== 0x0004) b.digitalWrite(LCD_DB2, b.HIGH, next);
218     else b.digitalWrite(LCD_DB2, b.LOW, next);
219 },function() {
220 if((chr & 0x0002)== 0x0002) b.digitalWrite(LCD_DB1, b.HIGH, next);
221     else b.digitalWrite(LCD_DB1, b.LOW, next);
222 },function() {
223 if((chr & 0x0001)== 0x0001) b.digitalWrite(LCD_DB0, b.HIGH, next);
224     else b.digitalWrite(LCD_DB0, b.LOW, next);
225 },function() {
226 //LCD Register Set (RS) to logic one for character input
227 b.digitalWrite(LCD_RS, b.HIGH, next);
228 },function() {
229 //LCD Enable (E) pin high
230 b.digitalWrite(LCD_E, b.HIGH, next);
231 },function() {
232 //End the write after 1ms
233 setTimeout(next, 1);
234 },function() {
235     //LCD Enable (E) pin low and call scheduleCallback when done
236     b.digitalWrite(LCD_E, b.LOW, next);
237 },function() {
238     //delay 1ms before calling 'callback'
239     setTimeout(callback, 1);
240 }];
241
242 next(); //Execute the first step
243
244 //Function for executing the next step
245 function next()
246     {
247     i++;
248     steps[i-1]();
249     }
250 }
251 //**********************************************************************
```

4.9 APPLICATION 2: THE DAGU MAGICIAN INTERFACE ON A CUSTOM CAPE

As mentioned earlier in the book, BeagleBone designers developed an elegant Cape system to interface hardware. A variety of BeagleBone Capes are available from Circuitco, Inc. In this section, we develop a Dagu Magician Cape to host the IR sensor, motor, and LCD interface. We

employ the Adafruit Proto Cape Kit. We also provide a layout for the Dagu Magician power regulator board.

Provided in Figure 4.31 is the layout and the connection diagram for the Adafruit Proto Cape for BeagleBone. The Proto Cape comes as a kit with edge connectors. The Cape may be equipped with stacking header connectors so several Capes may be stacked together. The layout provided for the connectors are also connected to corresponding nearby holes to allow for ease of hardware interface.

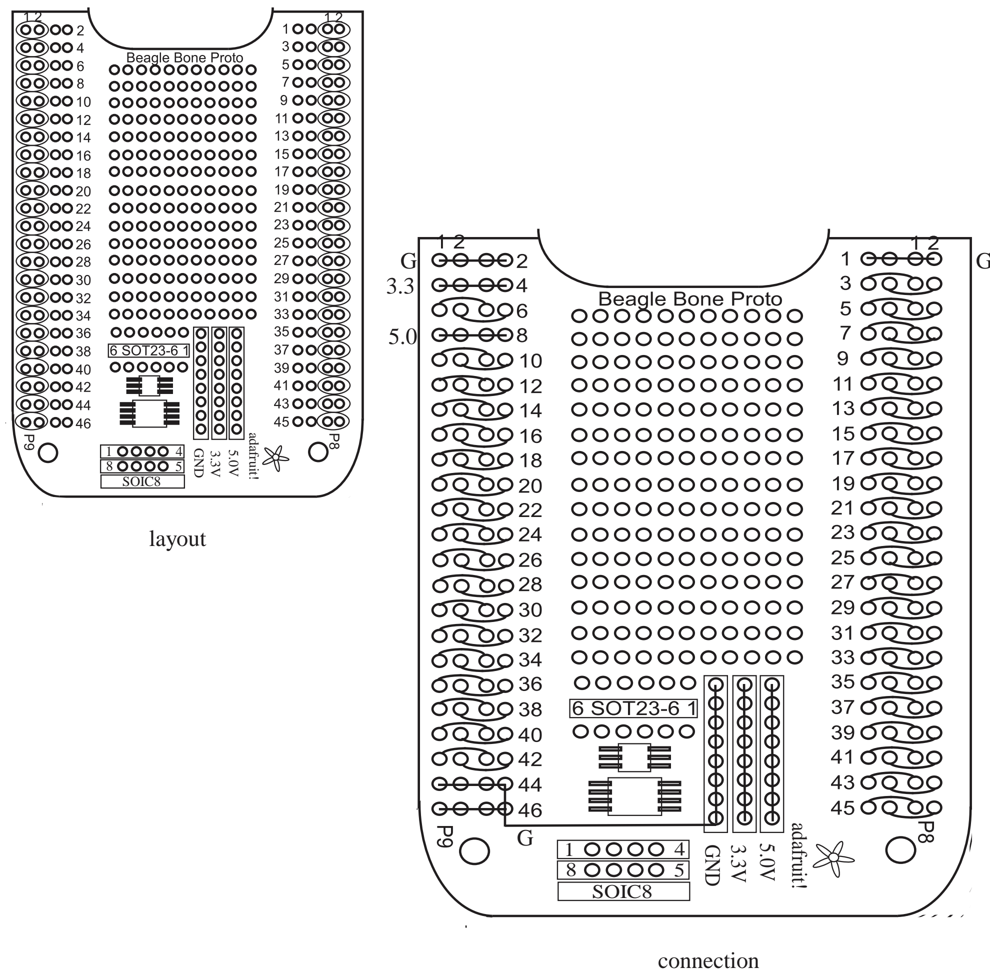


Figure 4.31: Adafruit BeagleBone prototype Cape.

In Figure 4.33 the Adafruit Cape has been used to interface the Dagu Magician robot IR sensors and motor interface drivers. The schematic for this circuitry was provided in Chapter 2.

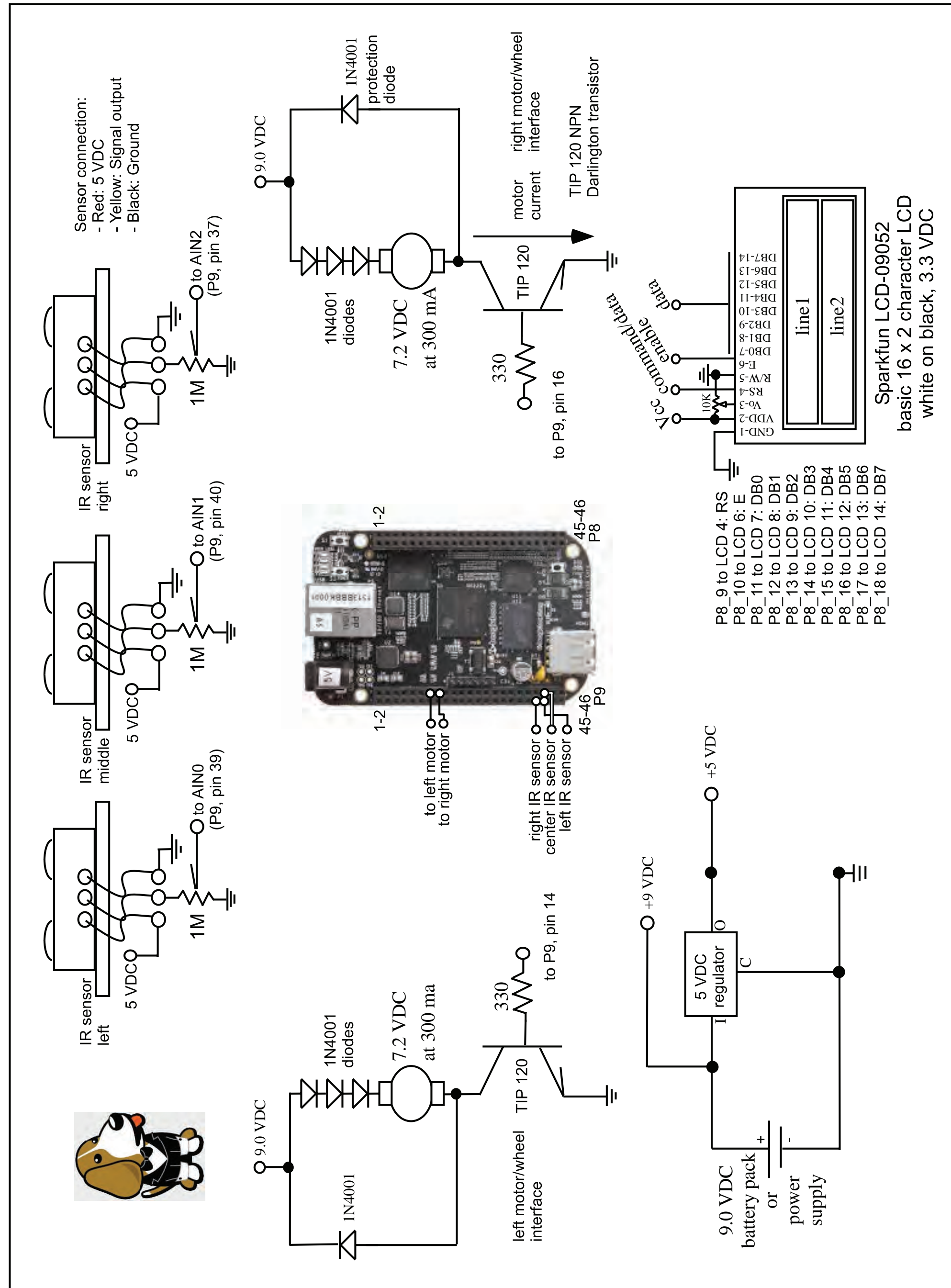
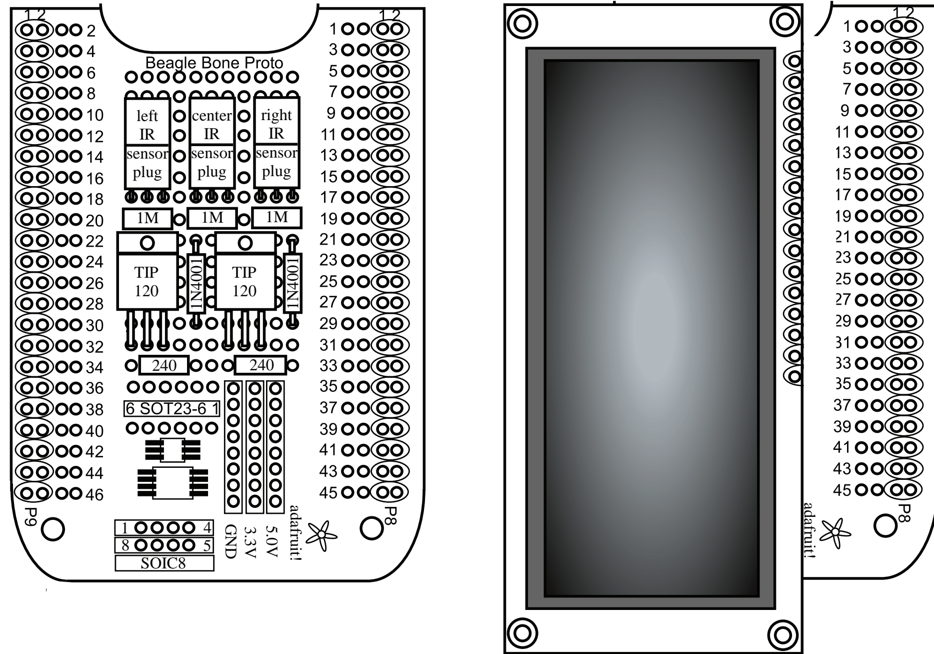
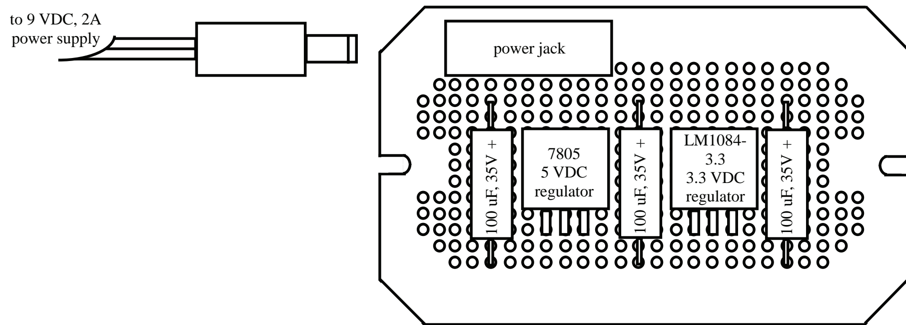


Figure 4.32: Dagu Magician interface circuit diagram. (Illustrations used with permission of Texas Instruments (www.TI.com).)



a) Dagu Magician IR sensor and motor interface custom Cape

b) LCD interface custom Cape



c) Dagu Magician power regulator

Figure 4.33: Dagu Magician interface. The interface hardware consists of a) the custom cape, b) the LCD, and c) the onboard voltage regulator board.

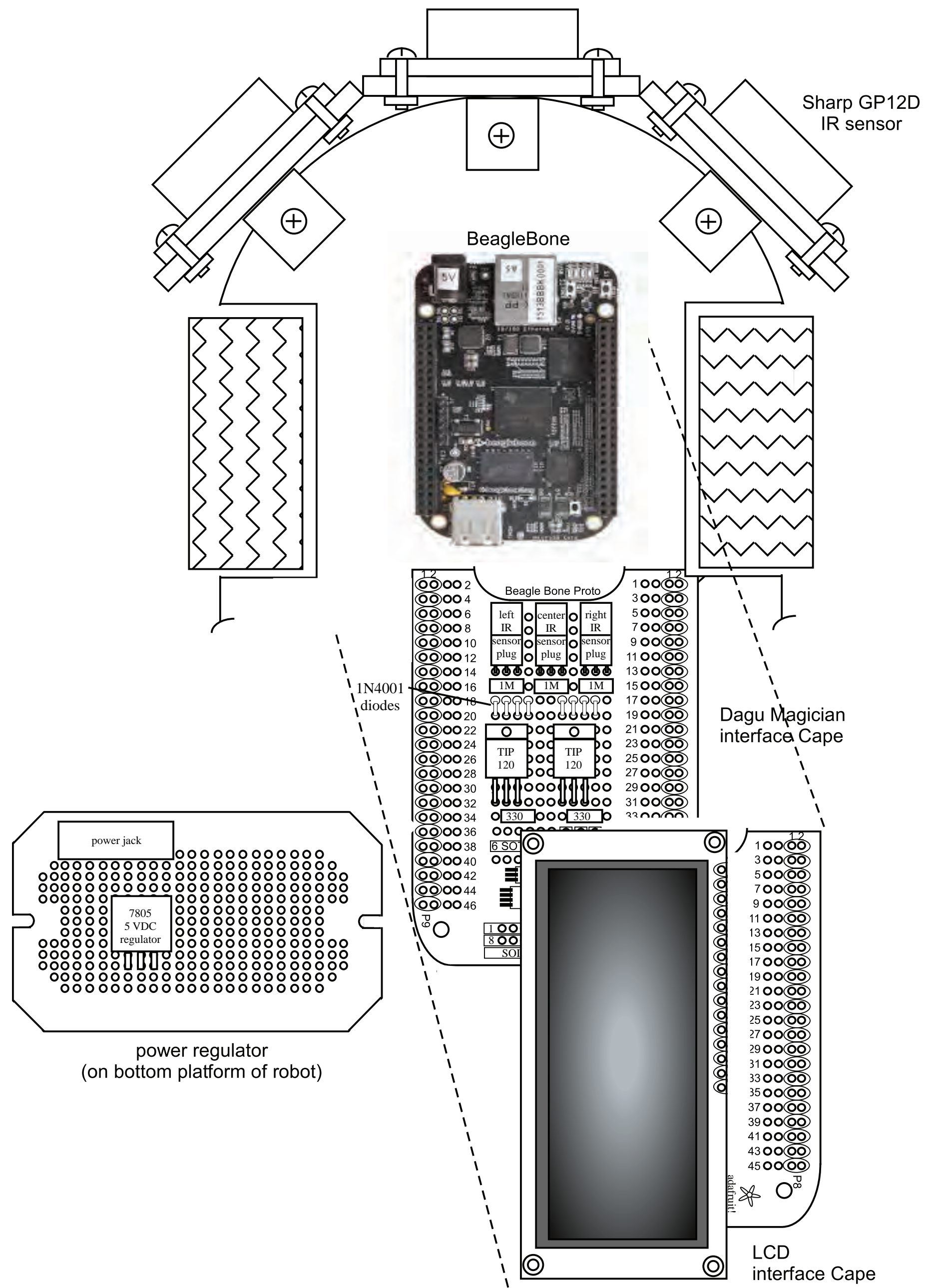


Figure 4.34: Interface hardware placement on the Dagu Magician robot.

For convenience we have provided a complete circuit diagram in Figure 4.32. Also, a Jameco general purpose prototyping board (#105100) is used for layout of the Dagu Magician power regulator. In Figure 4.34 the Dagu Magician robot has been equipped with the Capes and the power regulator board. A description of the software used to control the robot was provided in Chapter 3.

4.10 APPLICATION 3: SPECIAL EFFECTS LED CUBE

To illustrate some of the fundamentals of BeagleBone interfacing, we construct a three–dimensional LED cube. This design was inspired by an LED cube kit available from Jameco (`www.jameco.com`). This application originally appeared in the third edition of "Arduino Microcontroller Processing for Everyone!" The LED cube example has been adapted with permission for compatibility with the BeagleBone [Barrett, 2006].

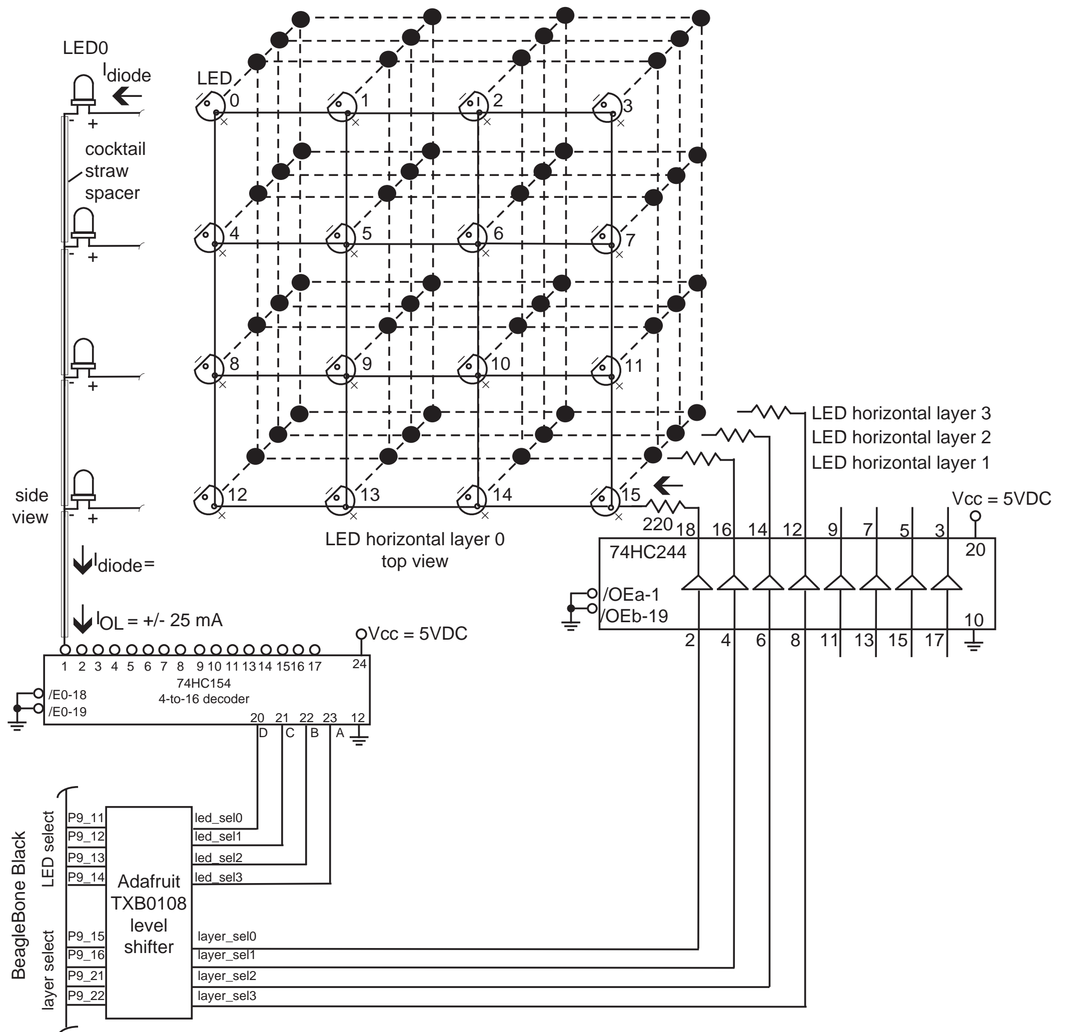
The BeagleBone Black is a 3.3 VDC system. With this is mind, we take two different design approaches.

1. Interface the 3.3 VDC BeagleBone Black to an LED cube designed for 5 VDC operation via a 3.3–5.0 VDC level shifter.
2. Modify the design of the LED cube to operate at 3.3 VDC.

We explore each design approach in turn.

Approach 1: 5 VDC LED cube: The LED cube consists of 4 layers of LEDs with 16 LEDs per layer. Only a single LED is illuminated at a given time. However, different effects may be achieved by how long a specific LED is left illuminated and the pattern of LED sequence followed. A specific LED layer is asserted using the layer select pins on the microcontroller using a one-hot-code (a single line asserted while the others are de-asserted). The asserted line is fed through a 74HC244 (three state, octal buffer, line driver) which provides an I_{OH}/I_{OL} current of ± 35 mA, as shown in Figure 4.35. A given output from the 74HC244 is fed to a common anode connection for all 16 LEDs in a layer. All four LEDs in a specific LED position, each in a different layer, share a common cathode connection. That is, an LED in a specific location within a layer shares a common cathode connection with three other LEDs that share the same position in the other three layers. The common cathode connection from each LED location is fed to a specific output of the 74HC154 4–16 decoder. The decoder has a one-cold-code output (one output at logic low while the others are at logic high). To illuminate a specific LED, the appropriate layer select and LED select line are asserted using the layer_sel[3:0] and led_sel[3:0] lines, respectively. This basic design may be easily expanded to a larger LED cube.

To interface the 5 VDC LED cube to the 3.3 VDC BeagleBone Black, a 3.3–5 VDC level shifter is required for each of the control signals (layer_sel and led_sel). In this example, a TXB0108 (low-voltage octal bidirectional transceiver) is employed to shift the 3.3 VDC signals of the BeagleBone to 5 VDC levels. Adafruit provides a breakout board for the level shifter (#TXB0108) [`www.adafruit.com`].



Notes:
1. LED cube consists of 4 layers of 16 LEDs each.
2. Each LED is individually addressed by asserting the appropriate cathode signal (0–15) and asserting a specific LED layer.
3. All LEDs in a given layer share a common anode connection.
4. All LEDs in a given position (0–15) share a common cathode connection.

Figure 4.35: 5 VDC LED special effects cube.

Approach 2: 3.3 VDC LED cube: A 3.3 VDC LED cube design is provided in Figure 4.36. The 74HC154 1-of-16 decoder has been replaced by two 3.3 VDC 74LVX138 1-of-8 decoders. The two 74LVX138 decoders form a single 1-of-16 decoder. The led_sel3 is used to select between the first decoder via enable pin /E2 or the second decoder via enable pin E3. Also, the 74HC244 has been replaced by a 3.3 VDC 74LVX244.

4.10.1 CONSTRUCTION HINTS

To limit project costs, low-cost red LEDs (Jameco #333973) were used. This LED has a forward voltage drop (V_f) of approximately 1.8 VDC and a nominal forward current (I_f) of 20 mA. The project requires a total of 64 LEDs (4 layers of 16 LEDs each). A LED template pattern was constructed from a 5" by 5" piece of pine wood. A 4-by-4 pattern of holes were drilled into the wood. Holes were spaced 3/4" apart. The hole diameter was slightly smaller than the diameter of the LEDs to allow for a snug LED fit.

The LED array was constructed a layer at a time using the wood template. Each LED was tested before inclusion in the array. A 5 VDC power supply with a series 220 ohm resistor was used to insure each LED was fully operational. The LED anodes in a given LED row were then soldered together. A fine-tip soldering iron and a small bit of solder were used for each interconnect as shown in Figure 4.37. Cross wires were then used to connect the cathodes of adjacent rows. A 22 gage bare wire was used. Again, a small bit of solder was used for the interconnect points. Four separate LED layers (4 by 4 array of LEDs) were completed.

To assemble the individual layers into a cube, cocktail straw segments were used as spacers between the layers. The straw segments provided spacing between the layers and also offered improved structural stability. The anodes for a given LED position were soldered together. For example, all LEDs in position 0 for all four layers shared a common anode connection.

The completed LED cube was mounted on a perforated printed circuit board (perfboard) to provide a stable base. LED sockets for the 74LS244 and the 74HC154 were also mounted to the perfboard. Connections were routed to a 16 pin ribbon cable connector. The other end of the ribbon cable was interfaced to the appropriate pins of the BeagleBone via the level shifter. The entire LED cube was mounted within a 4" plexiglass cube. The cube is available from the Container Store (`www.containerstore.com`). A construction diagram is provided in Figure 4.37. A picture of the LED cube is provided in Figure 4.38.

4.10.2 LED CUBE BONESCRIPT CODE

Provided below is the basic code template to illuminate a single LED (LED 0, layer 0). This basic template may be used to generate a number of special effects (e.g., tornado, black hole, etc.). Pin numbers are provided for the BeagleBone Black.

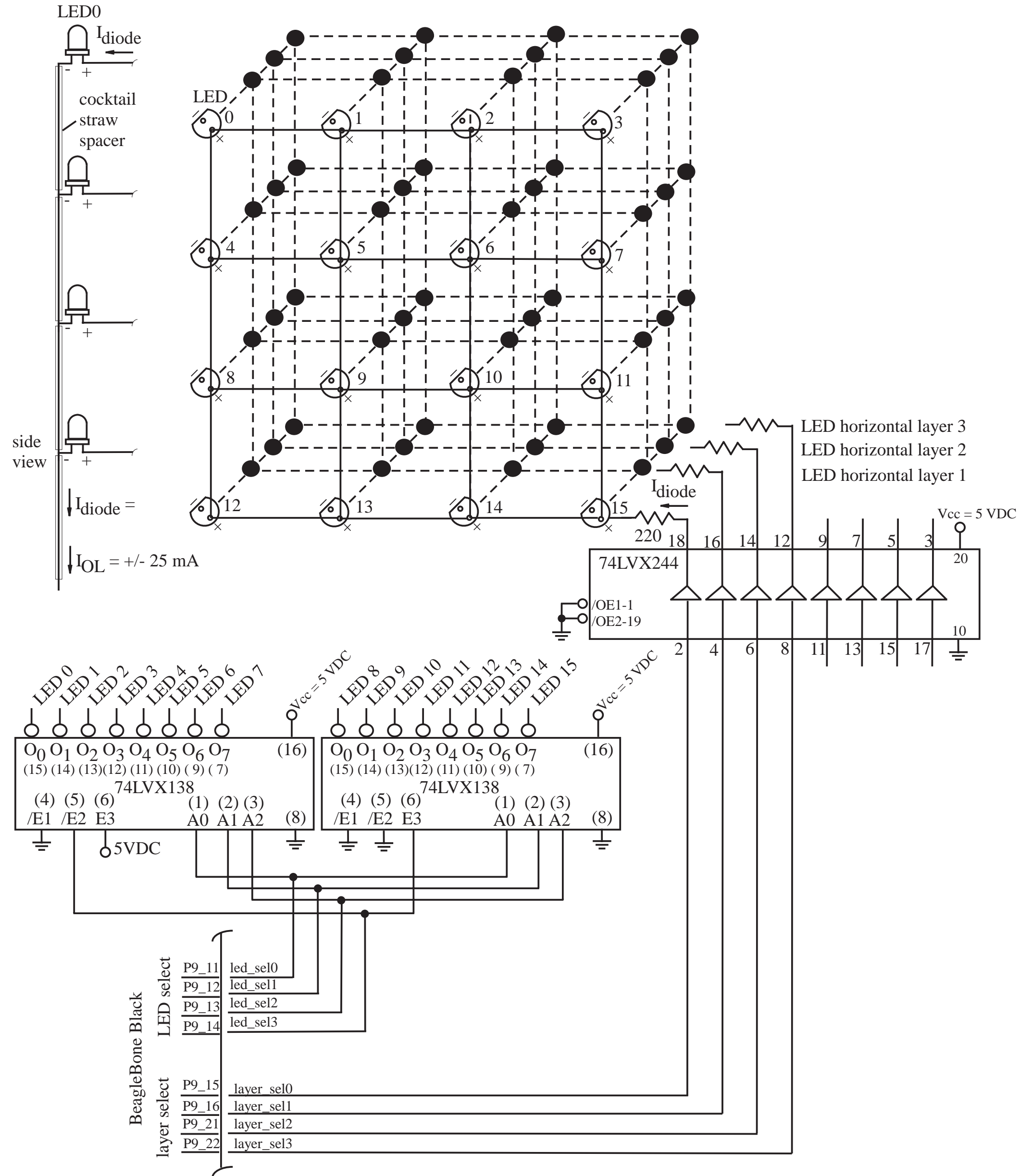
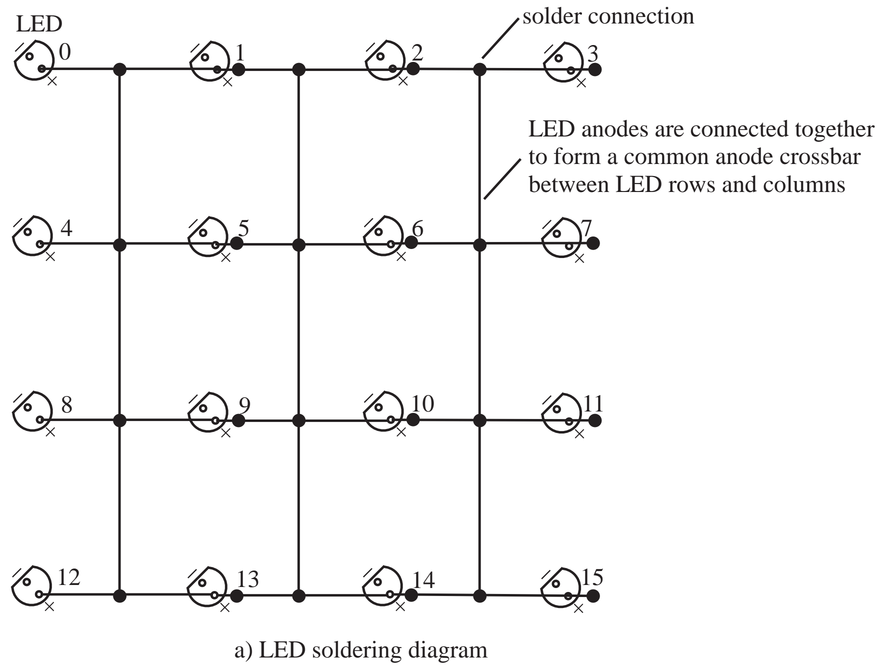
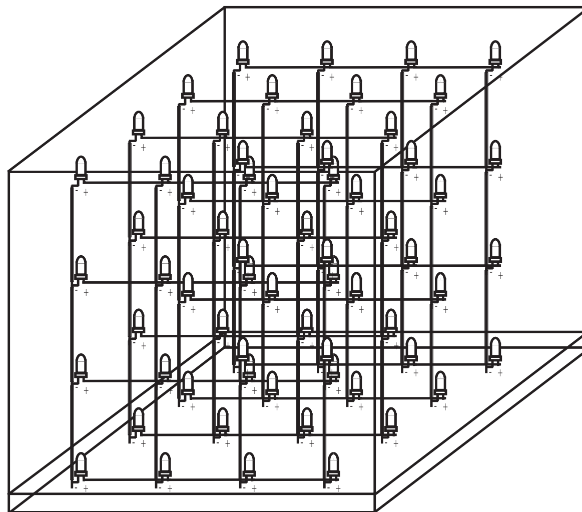


Figure 4.36: LED special effects cube.



a) LED soldering diagram



b) 3D LED array mounted within plexiglass cube

Figure 4.37: LED cube construction.

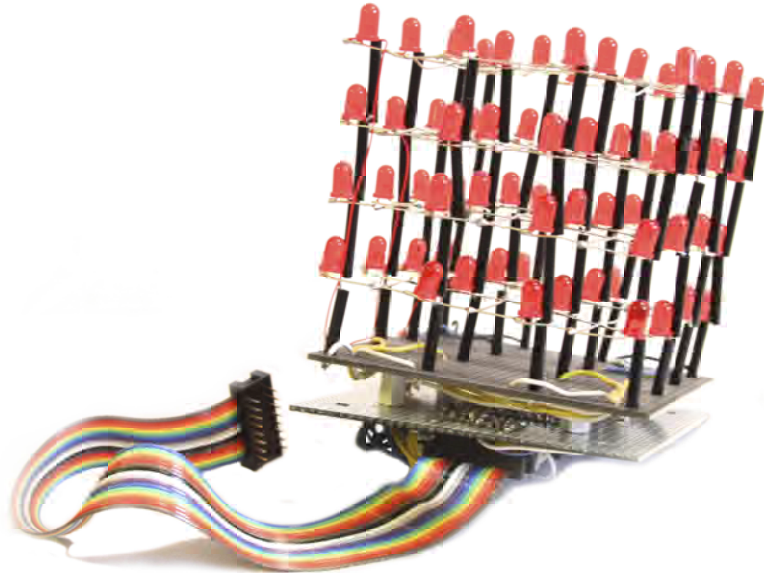


Figure 4.38: LED Cube. (Photo courtesy of Barrett [2015].)

```
 1 //************************************************
 2 //led_cube1.js
 3 //************************************************
 4
 5 var b = require('bonescript');
 6                                //led select pins
 7 var led_sel0 = ``P9_11'';      //BB Black header P9, pin 11
 8 var led_sel1 = ``P9_12'';      //BB Black header P9, pin 12
 9 var led_sel2 = ``P9_13'';      //BB Black header P9, pin 13
10 var led_sel3 = ``P9_14'';      //BB Black header P9, pin 14
11
12                                //layer select pins
13 var layer_sel0 = ``P9_15'';    //BB Black header P9, pin 15
14 var layer_sel1 = ``P9_16'';    //BB Black header P9, pin 16
15 var layer_sel2 = ``P9_21'';    //BB Black header P9, pin 21
16 var layer_sel3 = ``P9_22'';    //BB Black header P9, pin 22
17
18 b.pinMode(led_sel0, b.OUTPUT);
19 b.pinMode(led_sel1, b.OUTPUT);
20 b.pinMode(led_sel2, b.OUTPUT);
```

```
21 b.pinMode(led_sel3 , b.OUTPUT);
22
23 b.pinMode(layer_sel0 , b.OUTPUT);
24 b.pinMode(layer_sel1 , b.OUTPUT);
25 b.pinMode(layer_sel2 , b.OUTPUT);
26 b.pinMode(layer_sel3 , b.OUTPUT);
27
28 loop();
29
30 function loop()
31 {
32                                  //illuminate LED 0, layer 0
33                                  //led select
34 b.digitalWrite(led_sel0 , b.LOW);
35 b.digitalWrite(led_sel1 , b.LOW);
36 b.digitalWrite(led_sel2 , b.LOW);
37 b.digitalWrite(led_sel3 , b.LOW);
38                                  //layer select
39 b.digitalWrite(layer_sel0 , b.HIGH);
40 b.digitalWrite(layer_sel1 , b.LOW);
41 b.digitalWrite(layer_sel2 , b.LOW);
42 b.digitalWrite(layer_sel3 , b.LOW);
43 }
44
45 //*********************************************
```

In the next example, a function "illuminate_LED" has been added. To illuminate a specific LED, the LED position (0–15), the LED layer (0–3), and the length of time to illuminate the LED in milliseconds is specified using the setInterval function. In this short example, LED 0 is sequentially illuminated in each layer. An LED grid map is provided in Figure 4.39. It is useful for planning special effects.

```
 1 //**********************************************
 2 //led_cube2.js
 3 //**********************************************
 4
 5 var b = require('bonescript');
 6
 7 var i = 0;
 8
 9                                  //led select pins
10 var led_sel0 = ``P9_11'';        //BB Black header P9, pin 11
11 var led_sel1 = ``P9_12'';        //BB Black header P9, pin 12
12 var led_sel2 = ``P9_13'';        //BB Black header P9, pin 13
13 var led_sel3 = ``P9_14'';        //BB Black header P9, pin 14
14
15                                  //layer select pins
16 var layer_sel0 = ``P9_15'';      //BB Black header P9, pin 15
```

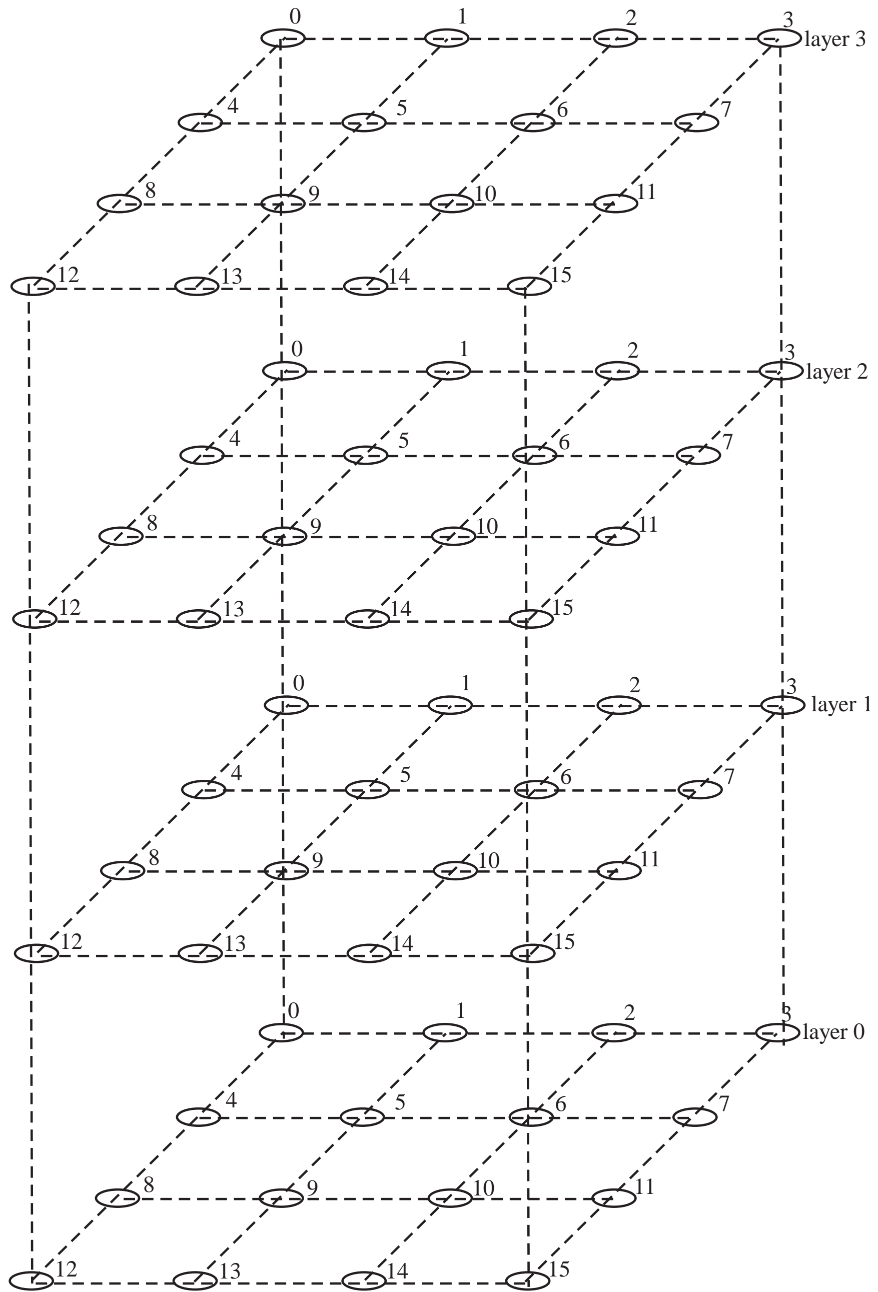


Figure 4.39: LED grid map.

```
17 var layer_sel1 = ``P9_16'';   //BB Black header P9, pin 16
18 var layer_sel2 = ``P9_21'';   //BB Black header P9, pin 21
19 var layer_sel3 = ``P9_22'';   //BB Black header P9, pin 22
20
21 b.pinMode(led_sel0 , b.OUTPUT);
22 b.pinMode(led_sel1 , b.OUTPUT);
23 b.pinMode(led_sel2 , b.OUTPUT);
24 b.pinMode(led_sel3 , b.OUTPUT);
25
26 b.pinMode(layer_sel0 , b.OUTPUT);
27 b.pinMode(layer_sel1 , b.OUTPUT);
28 b.pinMode(layer_sel2 , b.OUTPUT);
29 b.pinMode(layer_sel3 , b.OUTPUT);
30
31 setInterval(loop, 500);       //illuminate each LED for 500ms
32
33 function loop()
34 {
35 if(i==0){illuminate_LED(0,0);}
36 if(i==1){illuminate_LED(0,1);}
37 if(i==2){illuminate_LED(0,2);}
38 if(i==3){illuminate_LED(0,3);}
39 i = i+1;
40 if(i==4){i=0;}
41 }
42
43 function illuminate_LED(led, layer)
44 {
45
46 //select LED
47 if(led==0)
48   {
49   //illuminate LED 0
50   b.digitalWrite(led_sel0 , b.LOW);
51   b.digitalWrite(led_sel1 , b.LOW);
52   b.digitalWrite(led_sel2 , b.LOW);
53   b.digitalWrite(led_sel3 , b.LOW);
54   }
55
56 else if(led==1)
57   {
58   //illuminate LED 1
59   b.digitalWrite(led_sel0 , b.HIGH);
60   b.digitalWrite(led_sel1 , b.LOW);
61   b.digitalWrite(led_sel2 , b.LOW);
62   b.digitalWrite(led_sel3 , b.LOW);
63   }
64
```

```
65 else if(led==2)
66   {
67   //illuminate LED 2
68   b.digitalWrite(led_sel0, b.LOW);
69   b.digitalWrite(led_sel1, b.HIGH);
70   b.digitalWrite(led_sel2, b.LOW);
71   b.digitalWrite(led_sel3, b.LOW);
72   }
73
74 else if(led==3)
75   {
76   //illuminate LED 0
77   b.digitalWrite(led_sel0, b.HIGH);
78   b.digitalWrite(led_sel1, b.HIGH);
79   b.digitalWrite(led_sel2, b.LOW);
80   b.digitalWrite(led_sel3, b.LOW);
81   }
82
83 else if(led==4)
84   {
85   //illuminate LED 4
86   b.digitalWrite(led_sel0, b.LOW);
87   b.digitalWrite(led_sel1, b.LOW);
88   b.digitalWrite(led_sel2, b.HIGH);
89   b.digitalWrite(led_sel3, b.LOW);
90   }
91
92 else if(led==5)
93   {
94   //illuminate LED 5
95   b.digitalWrite(led_sel0, b.HIGH);
96   b.digitalWrite(led_sel1, b.LOW);
97   b.digitalWrite(led_sel2, b.HIGH);
98   b.digitalWrite(led_sel3, b.LOW);
99   }
100
101 else if(led==6)
102   {
103   //illuminate LED 6
104   b.digitalWrite(led_sel0, b.LOW);
105   b.digitalWrite(led_sel1, b.HIGH);
106   b.digitalWrite(led_sel2, b.HIGH);
107   b.digitalWrite(led_sel3, b.LOW);
108   }
109
110 else if(led==7)
111   {
112   //illuminate LED 7
```

```
113     b.digitalWrite(led_sel0, b.HIGH);
114     b.digitalWrite(led_sel1, b.HIGH);
115     b.digitalWrite(led_sel2, b.HIGH);
116     b.digitalWrite(led_sel3, b.LOW);
117     }
118
119  else if(led==8)
120     {
121     //illuminate LED 8
122     b.digitalWrite(led_sel0, b.LOW);
123     b.digitalWrite(led_sel1, b.LOW);
124     b.digitalWrite(led_sel2, b.LOW);
125     b.digitalWrite(led_sel3, b.HIGH);
126     }
127
128  else if(led==9)
129     {
130     //illuminate LED 9
131     b.digitalWrite(led_sel0, b.HIGH);
132     b.digitalWrite(led_sel1, b.LOW);
133     b.digitalWrite(led_sel2, b.LOW);
134     b.digitalWrite(led_sel3, b.HIGH);
135     }
136
137  else if(led==10)
138     {
139     //illuminate LED 10
140     b.digitalWrite(led_sel0, b.LOW);
141     b.digitalWrite(led_sel1, b.HIGH);
142     b.digitalWrite(led_sel2, b.LOW);
143     b.digitalWrite(led_sel3, b.HIGH);
144     }
145
146  else if(led==11)
147     {
148     //illuminate LED 11
149     b.digitalWrite(led_sel0, b.HIGH);
150     b.digitalWrite(led_sel1, b.HIGH);
151     b.digitalWrite(led_sel2, b.LOW);
152     b.digitalWrite(led_sel3, b.HIGH);
153     }
154
155  else if(led==12)
156     {
157     //illuminate LED 12
158     b.digitalWrite(led_sel0, b.LOW);
159     b.digitalWrite(led_sel1, b.LOW);
160     b.digitalWrite(led_sel2, b.HIGH);
```

```
161   b.digitalWrite(led_sel3, b.HIGH);
162   }
163
164 else if(led==13)
165   {
166   //illuminate LED 13
167   b.digitalWrite(led_sel0, b.HIGH);
168   b.digitalWrite(led_sel1, b.LOW);
169   b.digitalWrite(led_sel2, b.HIGH);
170   b.digitalWrite(led_sel3, b.HIGH);
171   }
172
173 else if(led==14)
174   {
175   //illuminate LED 14
176   b.digitalWrite(led_sel0, b.LOW);
177   b.digitalWrite(led_sel1, b.HIGH);
178   b.digitalWrite(led_sel2, b.HIGH);
179   b.digitalWrite(led_sel3, b.HIGH);
180   }
181
182 else //(led==15)
183   {
184   //illuminate LED 14
185   b.digitalWrite(led_sel0, b.HIGH);
186   b.digitalWrite(led_sel1, b.HIGH);
187   b.digitalWrite(led_sel2, b.HIGH);
188   b.digitalWrite(led_sel3, b.HIGH);
189   }
190
191 //led layer select
192 if(layer==0)
193   {
194   //LED layer 0
195   b.digitalWrite(layer_sel0, b.HIGH);
196   b.digitalWrite(layer_sel1, b.LOW);
197   b.digitalWrite(layer_sel2, b.LOW);
198   b.digitalWrite(layer_sel3, b.LOW);
199   }
200
201 else if(layer==1)
202   {
203   //LED layer 1
204   b.digitalWrite(layer_sel0, b.LOW);
205   b.digitalWrite(layer_sel1, b.HIGH);
206   b.digitalWrite(layer_sel2, b.LOW);
207   b.digitalWrite(layer_sel3, b.LOW);
208   }
```

```
209
210 else if(layer==2)
211   {
212   //LED layer 2
213   b.digitalWrite(layer_sel0 , b.LOW);
214   b.digitalWrite(layer_sel1 , b.LOW);
215   b.digitalWrite(layer_sel2 , b.HIGH);
216   b.digitalWrite(layer_sel3 , b.LOW);
217   }
218
219  else //(layer==3)
220   {
221   //LED layer 3
222   b.digitalWrite(layer_sel0 , b.LOW);
223   b.digitalWrite(layer_sel1 , b.LOW);
224   b.digitalWrite(layer_sel2 , b.LOW);
225   b.digitalWrite(layer_sel3 , b.HIGH);
226   }
227
228 }
229 //****************************************
```

In the next example, a "tornado" special effect is produced. The effect starts with a small swirl at the bottom of the array. The swirl grows larger as it proceeds to the top of the array, as shown in Figure 4.40. It is useful for planning special effects.

```
 1 //****************************************
 2 //led_cube3.js
 3 //****************************************
 4
 5 var b = require('bonescript');
 6
 7 var i = 0;
 8
 9                              //led select pins
10 var led_sel0 = ``P9_11'';    //BB Black header P9, pin 11
11 var led_sel1 = ``P9_12'';    //BB Black header P9, pin 12
12 var led_sel2 = ``P9_13'';    //BB Black header P9, pin 13
13 var led_sel3 = ``P9_14'';    //BB Black header P9, pin 14
14
15                              //layer select pins
16 var layer_sel0 = ``P9_15'';  //BB Black header P9, pin 15
17 var layer_sel1 = ``P9_16'';  //BB Black header P9, pin 16
18 var layer_sel2 = ``P9_21'';  //BB Black header P9, pin 21
19 var layer_sel3 = ``P9_22'';  //BB Black header P9, pin 22
20
21 b.pinMode(led_sel0 , b.OUTPUT);
22 b.pinMode(led_sel1 , b.OUTPUT);
23 b.pinMode(led_sel2 , b.OUTPUT);
```

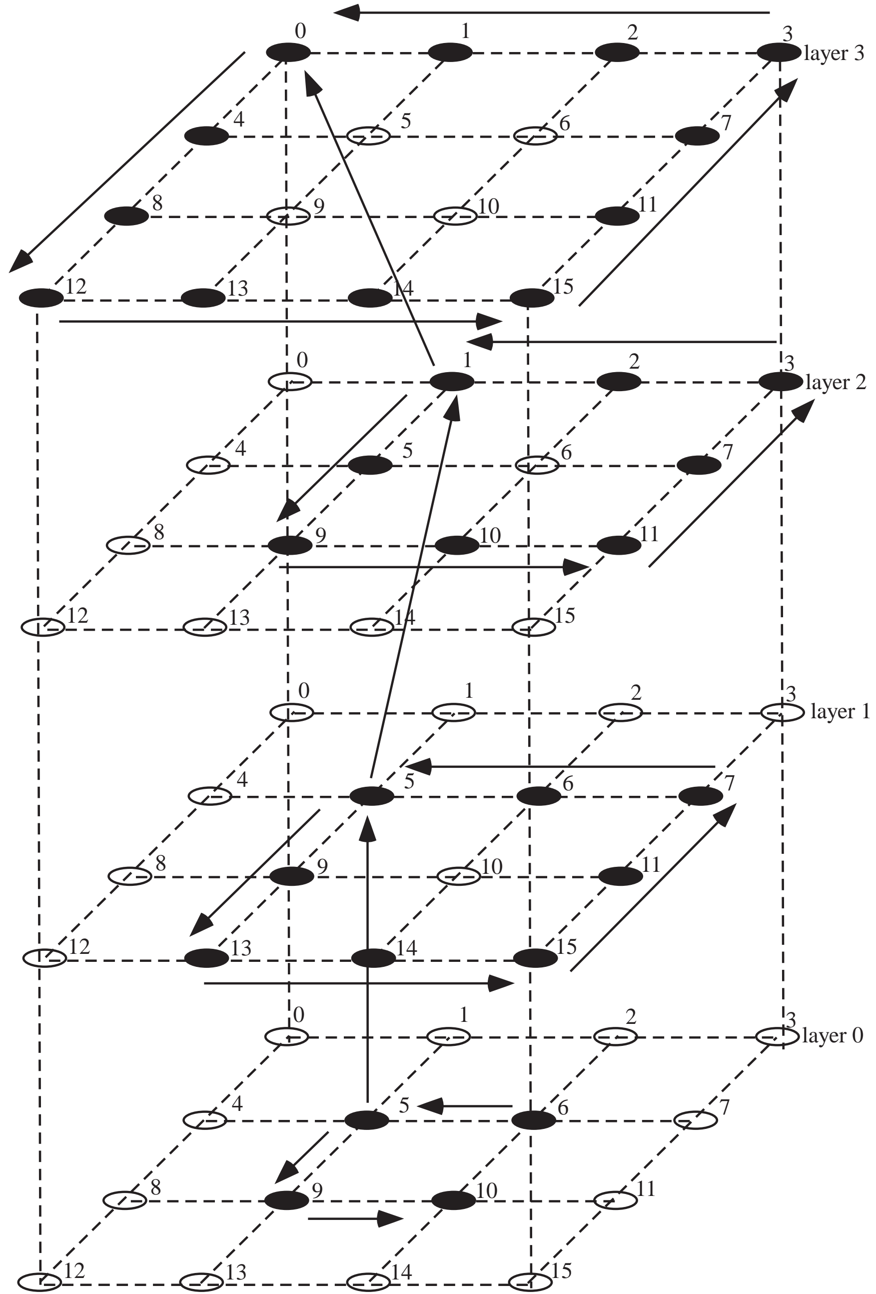


Figure 4.40: LED grid map for a tornado.

```
24 b.pinMode(led_sel3 , b.OUTPUT);
25
26 b.pinMode(layer_sel0 , b.OUTPUT);
27 b.pinMode(layer_sel1 , b.OUTPUT);
28 b.pinMode(layer_sel2 , b.OUTPUT);
29 b.pinMode(layer_sel3 , b.OUTPUT);
30
31 setInterval(loop, 500);
32
33 function loop()
34 {
35 if(i==0){illuminate_LED(5,0);}
36 if(i==1){illuminate_LED(6,0);}
37 if(i==2){illuminate_LED(9,0);}
38 if(i==3){illuminate_LED(10,0);}
39
40 if(i==4){illuminate_LED(5,1);}
41 if(i==5){illuminate_LED(9,1);}
42 if(i==6){illuminate_LED(13,1);}
43 if(i==7){illuminate_LED(14,1);}
44 if(i==8){illuminate_LED(15,1);}
45 if(i==9){illuminate_LED(11,1);}
46 if(i==10){illuminate_LED(7,1);}
47 if(i==11){illuminate_LED(6,1);}
48
49 if(i==12){illuminate_LED(1,2);}
50 if(i==13){illuminate_LED(5,2);}
51 if(i==14){illuminate_LED(9,2);}
52 if(i==15){illuminate_LED(10,2);}
53 if(i==16){illuminate_LED(11,2);}
54 if(i==17){illuminate_LED(7,2);}
55 if(i==18){illuminate_LED(3,2);}
56 if(i==19){illuminate_LED(2,2);}
57
58 if(i==20){illuminate_LED(0,3);}
59 if(i==21){illuminate_LED(4,3);}
60 if(i==22){illuminate_LED(8,3);}
61 if(i==23){illuminate_LED(12,3);}
62 if(i==24){illuminate_LED(13,3);}
63 if(i==25){illuminate_LED(14,3);}
64 if(i==26){illuminate_LED(15,3);}
65 if(i==27){illuminate_LED(11,3);}
66 if(i==28){illuminate_LED(7,3);}
67 if(i==29){illuminate_LED(3,3);}
68 if(i==30){illuminate_LED(2,3);}
69 if(i==31){illuminate_LED(1,3);}
70
71 i = i+1;
```

```
72 if(i==32){i=0;}
73 }
74
75 function illuminate_LED(led, layer)
76 {
77
78   :
79   :
80   :
81
82 }
83 //*************************************
```

4.11 SUMMARY

In this chapter, we presented the voltage and current operating parameters for BeagleBone. We discussed how this information may be applied to properly design an interface for common input and output circuits. It must be emphasized a carefully and properly designed interface allows the processor to operate properly within its parameter envelope. If, due to a poor interface design, a processor is used outside its prescribed operating parameter values, spurious and incorrect logic values will result. We provided interface information for a wide range of input and output devices. We also discussed the concept of interfacing a motor to the processor using PWM techniques coupled with high power MOSFET or SSR switching devices.

4.12 REFERENCES

- Barret, J. "Closer to the Sun International." www.closertothesungallery.com.
- Barrett, J. 2006. *Arduino Processing for Everyone,* 3rd. ed.
- Barrett, S. and Pack, D. 2005. *Embedded Systems Design and Applications with the 68HC12 and HCS12.* Upper Saddle River, NJ: Pearson Prentice Hall.
- Barrett, S. and Pack, D. 2006. *Processors Fundamentals for Engineers and Scientists.* San Rafael, CA: Morgan & Claypool Publishers; www.morganclaypool.com.
- Barrett, S. and Pack, D. 2008. *Atmel AVR Processor Primer Programming and Interfacing.* Morgan & Claypool Publishers; www.morganclaypool.com.
- Coley, G. 2014. *BeagleBone Rev C.1 Systems Reference Manual*; www.beaglebord.org.
- "Crydom Corporation." 2015. www.crydom.com.
- *Electrical Signals and Systems.* Primis Custom Publishing, McGraw-Hill Higher Education, Department of Electrical Engineering, United States Air Force Academy, USAF Academy, CO.

- *eTape Continuous Fluid Level Sensor.* 2015. Milone Technologies; www.milonetech.com
- Faulkenberry, L. 1977. *An Introduction to Operational Amplifiers,* New York: John Wiley & Sons.
- Faulkenberry, L. 1982. *Introduction to Operational Amplifiers with Linear Integrated Circuit Applications,* New York: John Wiley & Sons.
- Hughes-Croucher, T. and Wilson, M. 2012. *Node Up and Running.* Sebastopol, CA: O'Reilly Media, Inc.
- Images Company. Staten Island, NY.
- Kiessling, M. 2012. *The Node Beginner Guide: A Comprehensive Node.js Tutorial.*
- *Linear Technology, LTC1157 3.3 Dual Micropower High-Side/Low-Side MOSFET Driver.* Linear Technology Corporation, Milpitas, CA, 1993.
- Mims III, F.M. 2000. *Getting Started in Electronics.* Niles, IL: Master Publishing.
- Pollock, J. 2010. *JavaScript.* 3rd ed. New York: McGraw Hill.
- "Sick/Stegmann Incorporated." 2015. www.stegmann.com.
- *Texas Instruments H-bridge Motor Controller IC,* SLVSA74A, 2010, Texas Instruments Incorporated.
- Vander Veer, E. 2005. *JavaScript for Dummies.* 4th ed. Hoboken, NJ: Wiley Publishing, Inc.

4.13 CHAPTER EXERCISES

1. What will happen if a processor is used outside of its prescribed operating envelope?
2. Discuss the difference between the terms "sink" and "source" as related to current loading of a processor.
3. Can an LED with a series limiting resistor be directly driven by an output pin on the BeagleBone? Explain.
4. In your own words, provide a brief description of each of the electrical parameters of the processor.
5. What is switch bounce? Describe two techniques to minimize switch bounce.
6. Describe a method of debouncing a keypad.

7. What is the difference between an incremental encoder and an absolute encoder? Describe applications for each type.

8. What must be the current rating of the 2N2222 and 2N2907 transistors used in the tri-state LED circuit? Support your answer.

9. Draw the circuit for a six-character, seven-segment display. Fully specify all components.

10. Repeat the problem above for a dot matrix display.

11. Repeat the problem above for an LCD display.

12. BeagleBone has been connected to a JRP 42BYG016 unipolar, 1.8° per step, 12 VDC at 160 mA stepper motor. The interface circuit is shown in Figure 4.27. A 1 s delay is used between the steps to control motor speed. Pushbutton switches SW1 and SW2 are used to assert CW and CCW stepper motion. Write the code to support this application.

CHAPTER 5

BeagleBone Systems Design

Objectives: After reading this chapter, the reader should be able to do the following.

- Define an embedded system.
- List all aspects related to the design of an embedded system.
- Provide a step–by–step approach to design an embedded system.
- Discuss design tools and practices related to embedded systems design.
- Discuss the importance of system testing.
- Apply embedded system design practices in the prototype of a BeagleBone-based system with several subsystems.
- Provide a detailed design for a submersible remotely operated vehicle (ROV) including hardware layout and interface, structure chart, UML activity diagrams, and an algorithm coded in Bonescript.
- Provide a detailed design for a four-wheel drive (4WD) mountain maze navigating robot including hardware layout and interface, structure chart, UML activity diagrams, and an algorithm coded in BoneScript.

5.1 OVERVIEW

In the first three chapters of the book, we introduced BeagleBone, the Bonescript programming environment, and hardware interface techniques. We pull these three topics together in this chapter. This chapter provides a step-by-step, methodical approach towards designing advanced embedded systems. In this chapter, we begin with a definition of an embedded system. We then explore the process of how to successfully (and with low stress) develop an embedded system prototype that meets established requirements. The overview of embedded system design techniques was adapted with permission from earlier Morgan & Claypool projects. We also emphasize good testing techniques. We conclude the chapter with several extended examples. The examples illustrate the embedded system design process in the development and prototype of a submersible remotely operated vehicle (ROV) and a 4WD mountain maze navigating robot.

5.2 WHAT IS AN EMBEDDED SYSTEM?

An embedded system is typically designed for a specific task. It contains a processor to collect system inputs and generate system outputs. The link between system inputs and outputs is provided by a coded algorithm stored within the processor's resident memory. What makes embedded systems design so challenging and interesting is the design must also account for proper electrical interface for the input and output devices, potentially limited on-chip resources, human interface concepts, the operating environment of the system, cost analysis, related standards, and manufacturing aspects [Anderson, 2008]. Through careful application of this material you will be able to design and prototype embedded systems based on BeagleBone.

5.3 EMBEDDED SYSTEM DESIGN PROCESS

In this section, we provide a step-by-step approach to develop the first prototype of an embedded system that will meet established requirements. There are many formal design processes that we could study. We concentrate on the steps that are common to most. We purposefully avoid formal terminology of a specific approach and instead concentrate on the activities that are accomplished during the development of a system prototype. The design process we describe is illustrated in Figure 5.1 using a Unified Modeling Language (UML) activity diagram. We discuss the UML activity diagrams later in this section.

5.3.1 PROJECT DESCRIPTION

The goal of the project description step is to determine what the system is ultimately supposed to do. Questions to raise and answer during this step include, but are not limited to, the following.

- What is the system supposed to do?
- Where will it be operating and under what conditions?
- Are there any restrictions placed on the system design?

To answer these questions, the designer interacts with the client to ensure clear agreement on what is to be done. The establishment of clear, definable system requirements may require considerable interaction between the designer and the client. It is essential that both parties agree on system requirements before proceeding further in the design process. The final result of this step is a detailed listing of system requirements and related specifications. If you are completing this project for yourself, you must still carefully and thoughtfully complete this step.

5.3.2 BACKGROUND RESEARCH

Once a detailed list of requirements has been established, the next step is to perform background research related to the design. In this step, the designer will ensure they understand all requirements and features required by the project. This will again involve interaction between the designer

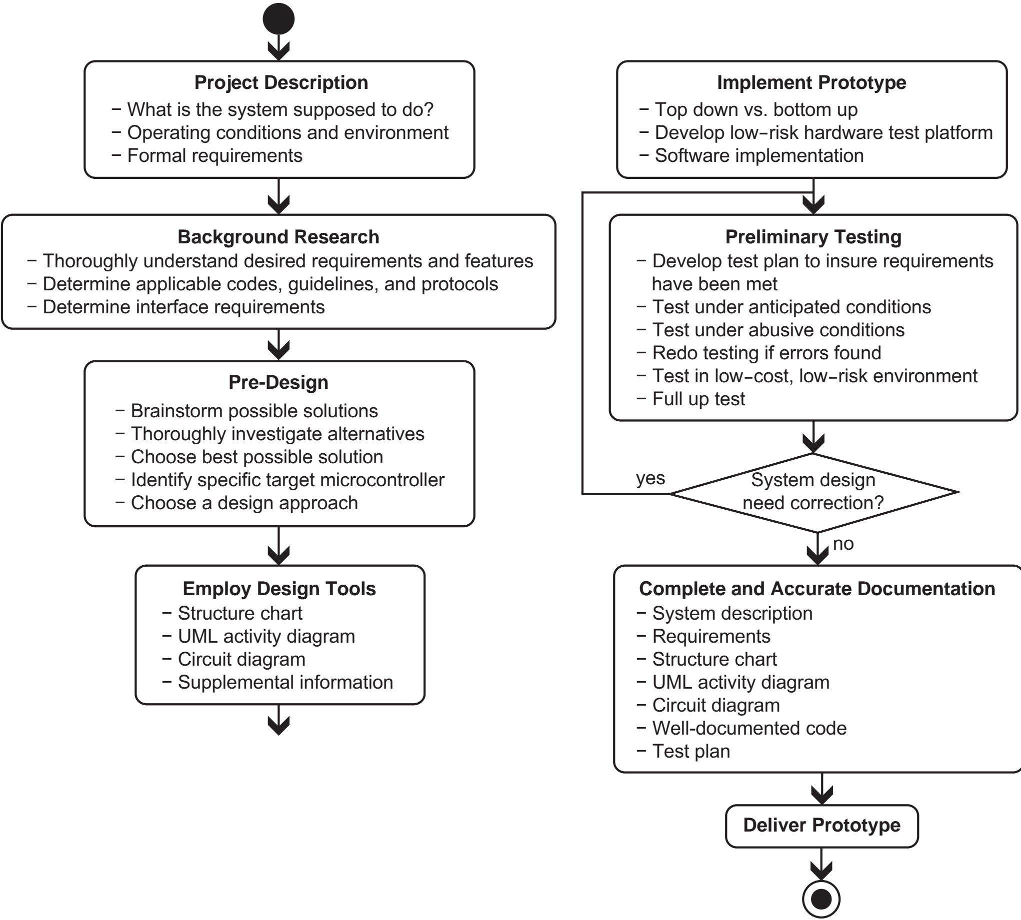


Figure 5.1: Embedded system design process.

and the client. The designer will also investigate applicable codes, guidelines, protocols, and standards related to the project. This is also a good time to start thinking about the interface between different portions of the input and output devices peripherally connected to the processor. The ultimate objective of this step is to have a thorough understanding of the project requirements, related project aspects, and any interface challenges within the project.

5.3.3 PRE-DESIGN

The goal of the pre-design step is to convert a thorough understanding of the project into possible design alternatives. Brainstorming is an effective tool in this step. Here, a list of alternatives is developed. Since an embedded system involves hardware and/or software, the designer can investigate whether requirements could be met with a hardware only solution or some combination of hardware and software. Generally speaking, a hardware only solution executes faster; however, the design is fixed once fielded. On the other hand, a software implementation provides flexibility but a slower execution speed. Most embedded design solutions will use a combination of both hardware and software to capitalize on the inherent advantages of each.

Once a design alternative has been selected, the general partition between hardware and software can be determined. It is also an appropriate time to select a specific hardware device to implement the prototype design. If a technology has been chosen, it is now time to select a specific processor. This is accomplished by answering the following questions.

- What processor systems or features (i.e., ADC, PWM, timer, etc.) are required by the design?
- How many input and output pins are required by the design?
- What type of memory components are required?
- What is the maximum anticipated operating speed of the processor expected to be?

Due to the variety of onboard systems, clock speed, and low cost; BeagleBone may be used in a wide array of applications typically held by microcontrollers and advanced processors.

5.3.4 DESIGN

With a clear view of system requirements and features, a general partition determined between hardware and software, and a specific processor chosen, it is now time to tackle the actual design. It is important to follow a systematic and disciplined approach to design. This will allow for low stress development of a documented design solution that meets requirements. In the design step, several tools are employed to ease the design process. They include the following:

- employing a top-down design, bottom up implementation approach;
- using a structure chart to assist in partitioning the system;

- using a Unified Modeling Language (UML) activity diagram to work out program flow, and
- developing a detailed circuit diagram of the entire system.

Let's take a closer look at each of these. The information provided here is an abbreviated version of the one provided in "Microcontrollers Fundamentals for Engineers and Scientists." The interested reader is referred there for additional details and an in-depth example [Barrett and Pack, 2006].

Top-down design, bottom-up implementation. An effective tool to start partitioning the design is based on the techniques of top-down design, bottom-up implementation. In this approach, you start with the overall system and begin to partition it into subsystems. At this point of the design, you are not concerned with how the design will be accomplished but how the different pieces of the project will fit together. A handy tool to use at this design stage is the structure chart. The structure chart shows how the hierarchy of system hardware and software components will interact and interface with one another. You should continue partitioning system activity until each subsystem in the structure chart has a single definable function. Directional arrows are used to indicate data flow in and out of a function.

UML activity diagram. Once the system has been partitioned into pieces, the next step is to work out the details of the operation of each subsystem previously identified. Rather than beginning to code each subsystem as a function, work out the information and control flow of each subsystem using another design tool: the Unified Modeling Language (UML) activity diagram. The activity diagram is simply a UML compliant flow chart. UML is a standardized method of documenting systems. The activity diagram is one of the many tools available from UML to document system design and operation. The basic symbols used in a UML activity diagram for a processor based system are provided in Figure 5.2 [Fowler and Scott, 2000].

To develop the UML activity diagram for the system, we can use a top-down, bottom-up, or a hybrid approach. In the top-down approach, we begin by modeling the overall flow of the algorithm from a high level. If we choose to use the bottom-up approach, we would begin at the bottom of the structure chart and choose a subsystem for flow modeling. The specific course of action chosen depends on project specifics. Often, a combination of both techniques, a hybrid approach, is used. You should work out all algorithm details at the UML activity diagram level prior to coding any software. If you can not explain system operation at this higher level first, you have no business being down in the detail of developing the code. Therefore, the UML activity diagram should be of sufficient detail so you can code the algorithm directly from it [Dale and Lilly, 1995].

In the design step, a detailed circuit diagram of the entire system is developed. It will serve as a roadmap to implement the system. It is also a good idea at this point to investigate available design information relative to the project. This would include hardware design examples, software code examples, and application notes available from manufacturers. As before, use a subsystem

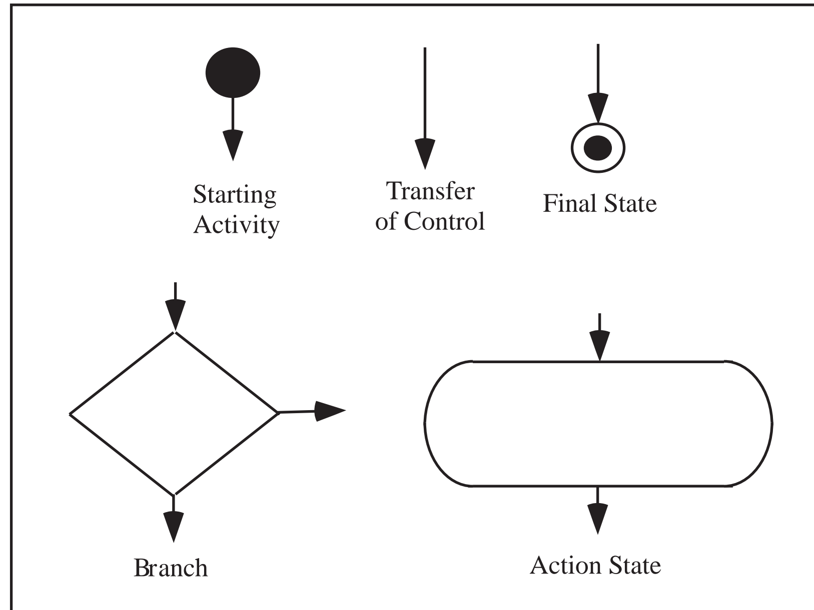


Figure 5.2: UML activity diagram symbols. Adapted from Fowler and Scott [2000].

approach to assemble the entire circuit. The basic building block interface circuits discussed in the previous chapter may be used to assemble the complete circuit.

At the completion of this step, the prototype design is ready for implementation and testing.

5.3.5 IMPLEMENT PROTOTYPE

To successfully implement a prototype, an incremental approach should be followed. Again, the top-down design, bottom-up implementation provides a solid guide for system implementation. In an embedded system design involving both hardware and software, the hardware system including the processor should be assembled first. This provides the software the required signals to interact with. As the hardware prototype is assembled on a prototype board, each component is tested for proper operation as it is brought online. This allows the designer to pinpoint malfunctions as they occur.

Once the hardware prototype is assembled, coding may commence. It is important to note that on larger projects software and hardware may be developed concurrently. As before, software should be incrementally brought online. You may use a top down, bottom up, or hybrid approach depending on the nature of the software. The important point is to bring the software online incrementally such that issues can be identified and corrected early on.

It is highly recommended that low-cost stand-in components be used when testing the software with the hardware components. For example, push buttons, potentiometers, and LEDs may be used as low-cost stand-in component simulators for expensive input instrumentation devices and expensive output devices such as motors. This allows you to insure the software is properly operating before using it to control the actual components.

5.3.6 PRELIMINARY TESTING

To test the system, a detailed test plan must be developed. Tests should be developed to verify that the system meets all of its requirements and also intended system performance in an operational environment. The test plan should also include scenarios in which the system is used in an unintended manner. As before, a top-down, bottom-up, or hybrid approach can be used to test the system. In a bottom-up approach individual units are tested first.

Once the test plan is completed, actual testing may commence. The results of each test should be carefully documented. As you go through the test plan, you will probably uncover a number of run-time errors in your algorithm. After you correct a run-time error, the entire test plan must be repeated. This ensures that the new fix does not have an unintended effect on another part of the system. Also, as you process through the test plan, you will probably think of other tests that were not included in the original test document. These tests should be added to the test plan. As you go through testing, realize your final system is only as good as the test plan that supports it!

Once testing is complete, you should accomplish another level of testing where you intentionally try to "jam up" the system. In other words, try to get your system to fail by trying combinations of inputs that were not part of the original design. A robust system should continue to operate correctly in this type of an abusive environment. It is imperative that you design robustness into your system. When testing on a low cost simulator is complete, the entire test plan should be performed again with the actual system hardware. Once this is completed you should have a system that meets its requirements!

5.3.7 COMPLETE AND ACCURATE DOCUMENTATION

With testing complete, the system design should be thoroughly documented. Much of the documentation will have already been accomplished during system development. Documentation will include the system description, system requirements, the structure chart, the UML activity diagrams documenting program flow, the test plan, results of the test plan, system schematics, and properly documented code. To properly document code, you should carefully comment all functions describing their operation, inputs, and outputs. Also, comments should be included within the body of the function describing key portions of the code. Enough detail should be provided such that code operation is obvious. It is also extremely helpful to provide variables and functions within your code names that describe their intended use.

You might think that a comprehensive system documentation is not worth the time or effort to complete it. Complete documentation pays rich dividends when it is time to modify, repair, or update an existing system. Also, well–documented code may be often reused in other projects: a method for efficient and timely development of new systems.

In the next two sections we provide detailed examples of the system design process: a submersible robot and a 4WD robot capable of navigating through a mountainous maze.

5.4 SUBMERSIBLE ROBOT

The area of submersible robots is fascinating and challenging. To design a robot is quite complex (yet fun). To add the additional requirement of waterproofing key components provides an additional level of challenge. (Water and electricity do not mix!) In this section we provide the construction details and a control system for a remotely operated vehicle, an ROV. Specifically, we develop the structure and control system for the SeaPerch style ROV, as shown in Figure 5.3. By definition, an ROV is equipped with a tether umbilical cable that provides power and control signals from a surface support platform. An Autonomous Underwater Vehicle (AUV) carries its own power and control equipment and does not require surface support [seaperch].

Details on the construction and waterproofing of an ROV are provided in the excellent and fascinating *Build Your Own Underwater Robot and Other Wet Projects* by Harry Bohm and Vickie Jensen. For an advanced treatment, please see *The ROV Manual—A User Guide for Remotely Operated Vehicles* by Robert Crist and Robert Wernli, Sr. There is a national-level competition for students based on the SeaPerch ROV. The goal of the program is to stimulate interest in the next generation of marine related engineering specialties [seaperch].

5.4.1 APPROACH

This is a challenging project; however, we take a methodical, step–by–step approach to successful design and construction of the ROV. We complete the design tasks in the following order:

- determine requirements;
- design and construct ROV structure;
- design and fabricate control electronics;
- design and implement control software using Bonescript;
- construct and assemble a prototype; and
- test the prototype.

5.4.2 REQUIREMENTS

The requirements for the ROV system include the following.

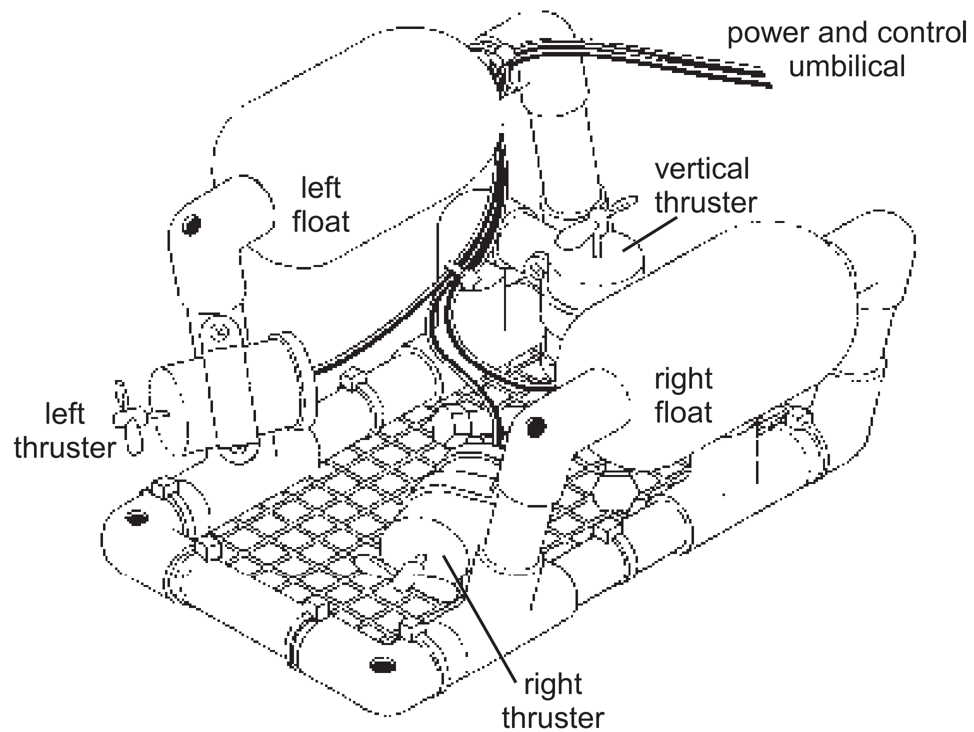


Figure 5.3: SeaPerch ROV. (Adapted and used with permission of Bohm and Jensen, 2012.)

- Develop a control system to allow a three-thruster (motor or bilge pump) ROV to move forward, left (port) and right (starboard).
- The ROV will be pushed down to a shallow depth via a vertical thruster and return to surface based on its own, slightly positive buoyancy.
- ROV movement will be under joystick control.
- Light-emitting diodes (LEDs) are used to indicate thruster assertion.
- All power and control circuitry will be maintained in a surface support platform, as shown in Figure 5.4.
- An umbilical cable connects the support platform to the ROV.

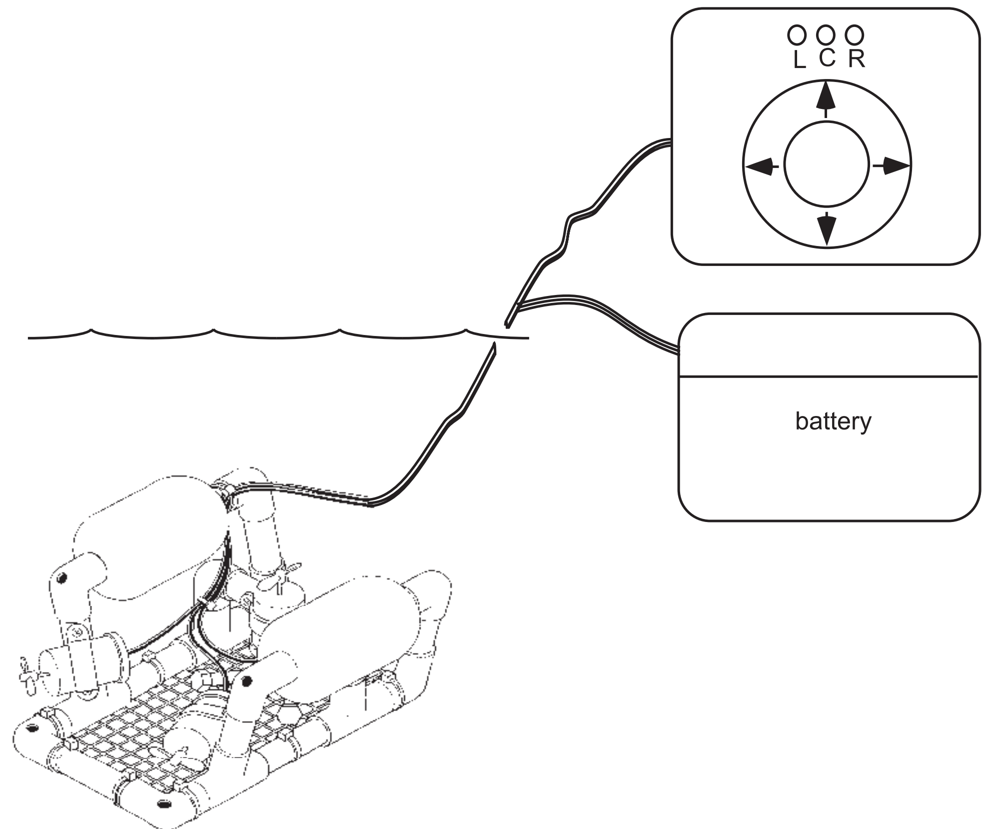


Figure 5.4: Power and control are provided remotely to the SeaPerch ROV. (Adapted and used with permission of Bohm and Jensen, 2012.)

5.4.3 ROV STRUCTURE

The ROV structure is shown in Figure 5.5. The structure is constructed with 0.75 in PVC piping. The structure is assembled quickly using "T" and corner connectors. The pieces are connected using PVC glue or machine screws. The PVC pipe and connectors are readily available in hardware and home improvement stores.

The fore or bow portion of the structure is equipped with plexiglass panels to serve as mounting bulkheads for the thrusters. The panels are mounted to the PVC structure using ring clamps. Either waterproofed electric motors or submersible bilge pumps are used as thrusters. A bilge pump is a pump specifically designed to remove water from the inside of a boat. The pumps are powered from a 12 VDC source and have typical flow rates from 360 to over 3,500 gallons per minute. They range in price from US $20–US $80 [www.shorelinemarinedevelopment.com].

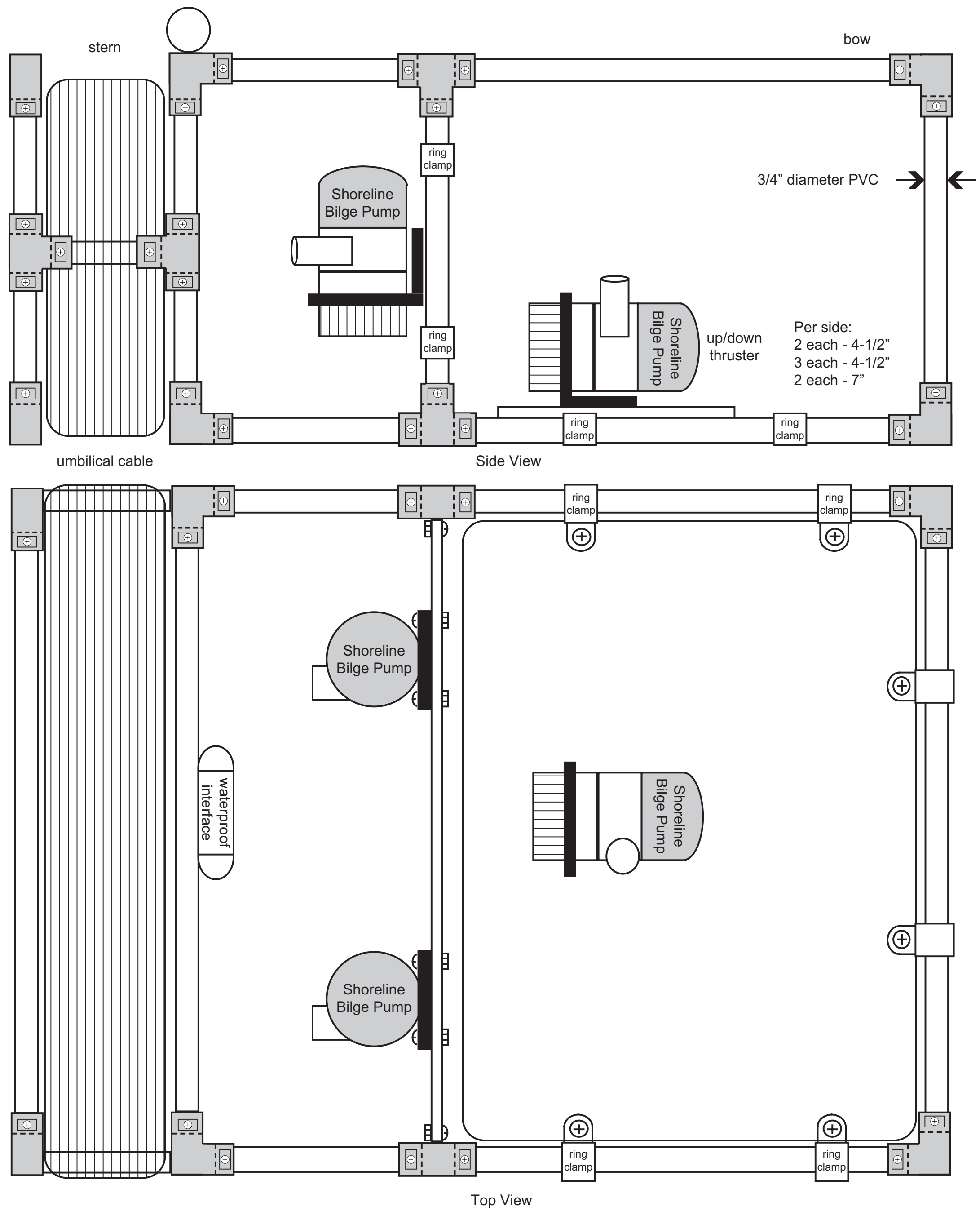


Figure 5.5: SeaPerch ROV structure.

Details on waterproofing electric motors are provided in *Build Your Own Underwater Robot and Other Wet Projects*, [Bohm and Jensen, 2012]. We use three Shoreline Bilge Pumps rated at 600 gallons per minute (GPM). They are available online from `www.walmart.com`.

The aft or stern portion of the structure is designed to hold the flexible umbilical cable. The cable provides a link between the BeagleBone Black based control system and the thrusters. Each thruster may require up to 1–2 amps of current. Therefore, a four-conductor, 16 AWG, braided (for flexibility) conductor cable is recommended. The cable is interfaced to the bilge pump leads using soldered connections or Butt connectors. The interface should be thoroughly waterproofed using caulk. For this design the interface was placed within a section of PVC pipe equipped with end caps. The resulting container is filled with waterproof caulk.

Once the ROV structure is complete its buoyancy is tested. This is accomplished by placing the ROV structure in water. The goal is to achieve a slightly positive buoyancy. With positive buoyancy the structure floats. With neutral buoyancy the structures hovers beneath the surface. With negative buoyancy the structure sinks. A more positive buoyancy way be achieved by attaching floats or foam to the structure tubing. A more negative buoyancy may be achieved by adding weights to the structure [Bohm and Jensen, 2012].

5.4.4 STRUCTURE CHART

The SeaPerch structure chart is provided in Figure 5.6. As can be seen in the figure, the SeaPerch control system will accept input from the five- position joystick (left, right, select, up, and down). We use the Sparkfun thumb joystick (Sparkfun COM–09032) mounted to an Adafruit Proto Cape Kit for BeagleBone (Adafruit 572), as shown in Figure 5.7. The joystick schematic and connections to BeagleBone are provided in Figures 5.8 and 5.9.

In response to user joystick input, the SeaPerch control algorithm will issue a control command indicating desired ROV direction. In response to this desired direction command, the motor control algorithm will issue control signals to assert the appropriate thrusters and LEDs.

5.4.5 CIRCUIT DIAGRAM

The circuit diagram for the SeaPerch control system is provided in Figure 5.8. The thumb joystick is used to select desired ROV direction. The thumb joystick contains two built-in potentiometers (horizontal and vertical). A reference voltage of 1.8 VDC is applied to the VCC input of the joystick. **A 1.8 VDC reference is required since this is the maximum allowable voltage to the BeagleBone Black's analog-to-digital conversion system.** The 1.8 VDC reference is provided by the LM317 positive, adjustable regulator circuit. As the joystick is moved, the horizontal (HORZ) and vertical (VERT) analog output voltages will change to indicate the joystick position. The joystick is also equipped with a digital select (SEL) button. The SEL button is used to activate an ROV dive. The joystick is interfaced to BeagleBone, as shown in Figure 5.8.

There are three LED interface circuits connected to BeagleBone header pins P8 pins 7–9. The LEDs illuminate to indicate the left, vertical, and right thrusters have been asserted. As

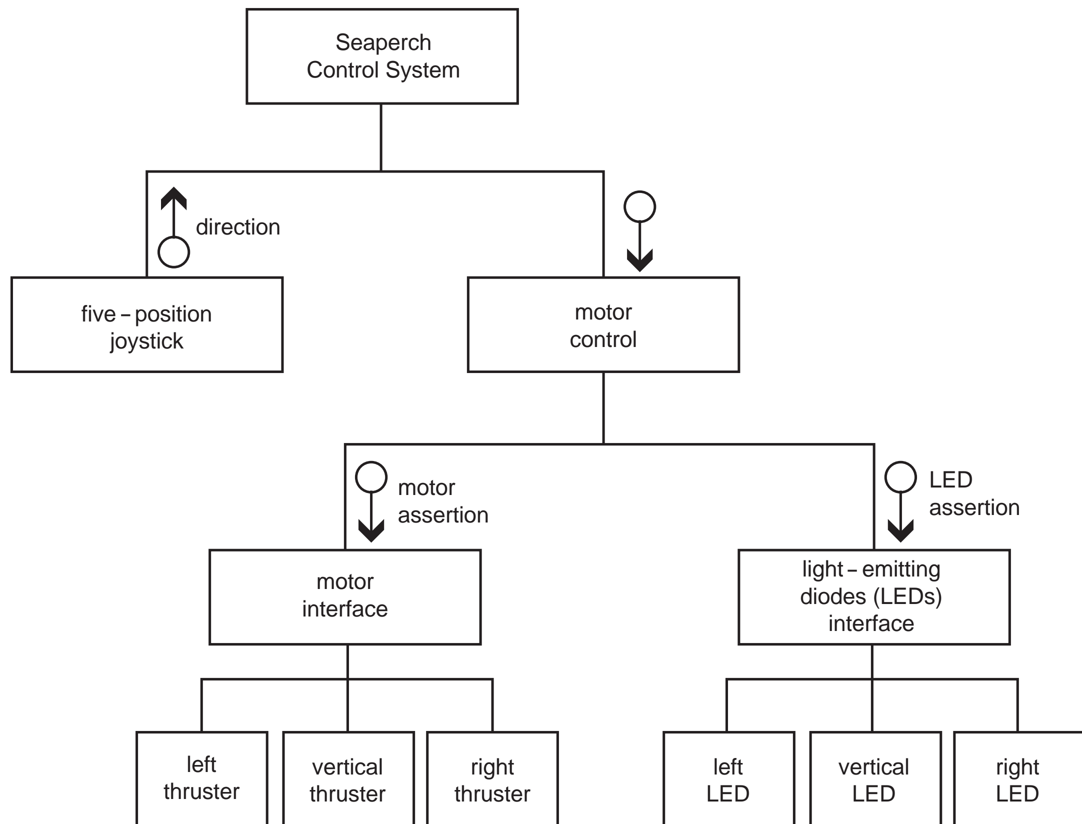


Figure 5.6: SeaPerch ROV structure chart.

previously mentioned, the prime mover for the ROV are three bilge pumps. The left and right bilge pumps are driven by pulse width modulation channels (BeagleBone P9 pins 13, 14, and 16) via power NPN Darlington transistors (TIP 120), as shown in Figure 5.8. The vertical thrust is under digital pin control P8 pin 13 equipped with NPN Darlington transistor (TIP 120) interface. Both the LED and the pump interfaces were discussed in the previous chapter.

The interface circuitry between BeagleBone Black and the bilge pumps is mounted on a printed circuit board (PCB) within the control housing. The interface between BeagleBone Black, the PCB, and the umbilical cable is provided in Figure 5.9.

5.4.6 UML ACTIVITY DIAGRAM

The SeaPerch control system UML activity diagram is provided in Figure 5.10. After initializing the BeagleBone Black pins the control algorithm is placed in a continuous loop awaiting user

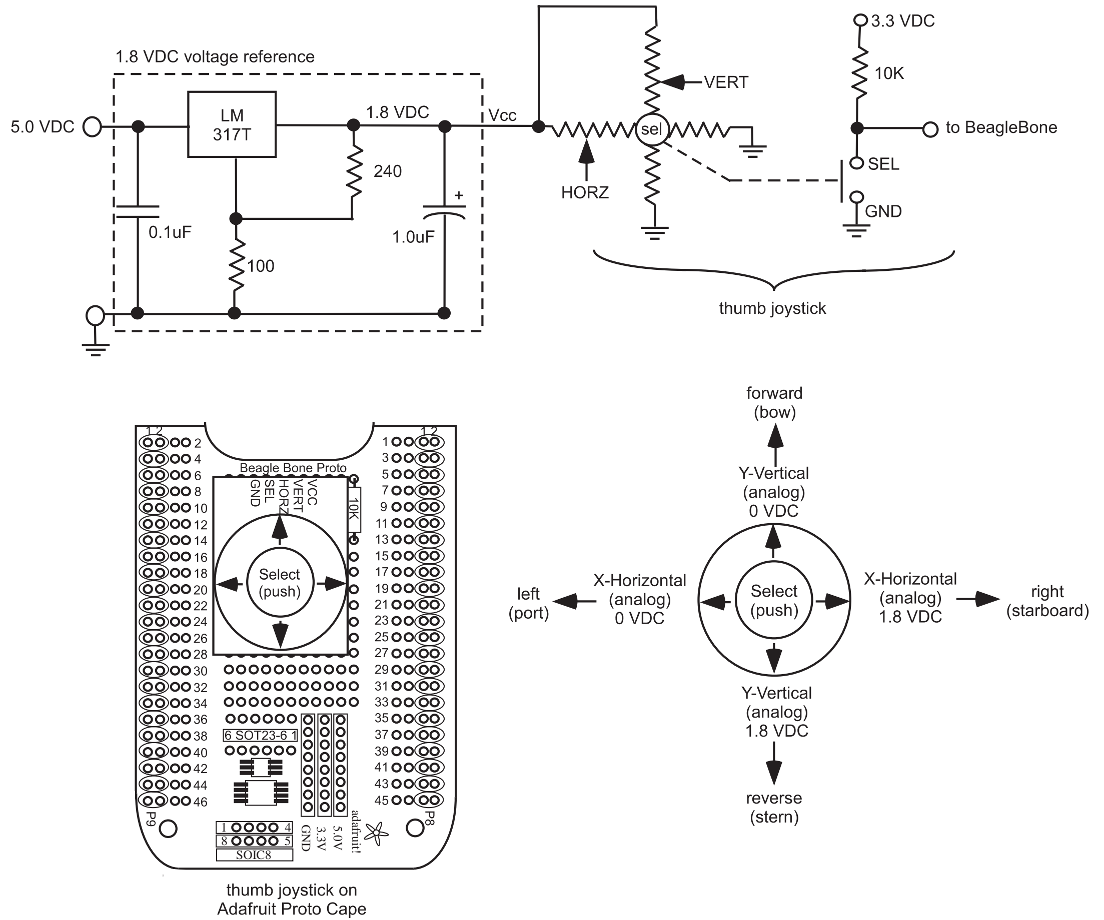


Figure 5.7: Thumb joystick mounted on an Adafruit Proto Cape Kit.

input. In response to user input, the algorithm determines desired direction of ROV travel and asserts appropriate control signals for the LED and motors.

5.4.7 BEAGLEBONE CODE

In this example we use the thumb joystick to control the left and right thruster (motor or bilge pump). The joystick provides a separate voltage from 0–1.8 VDC for the horizontal (HORZ) and vertical (VERT) position of the joystick. We use this voltage to set the duty cycle of the pulse width modulated (PWM) signals sent to the left and right thrusters. The select pushbutton (SEL) on the joystick is used to assert the vertical thruster. The Bonescript analog read function (b.analogRead) is used to read the X and Y position of the joystick. A value from 0–1 is reported

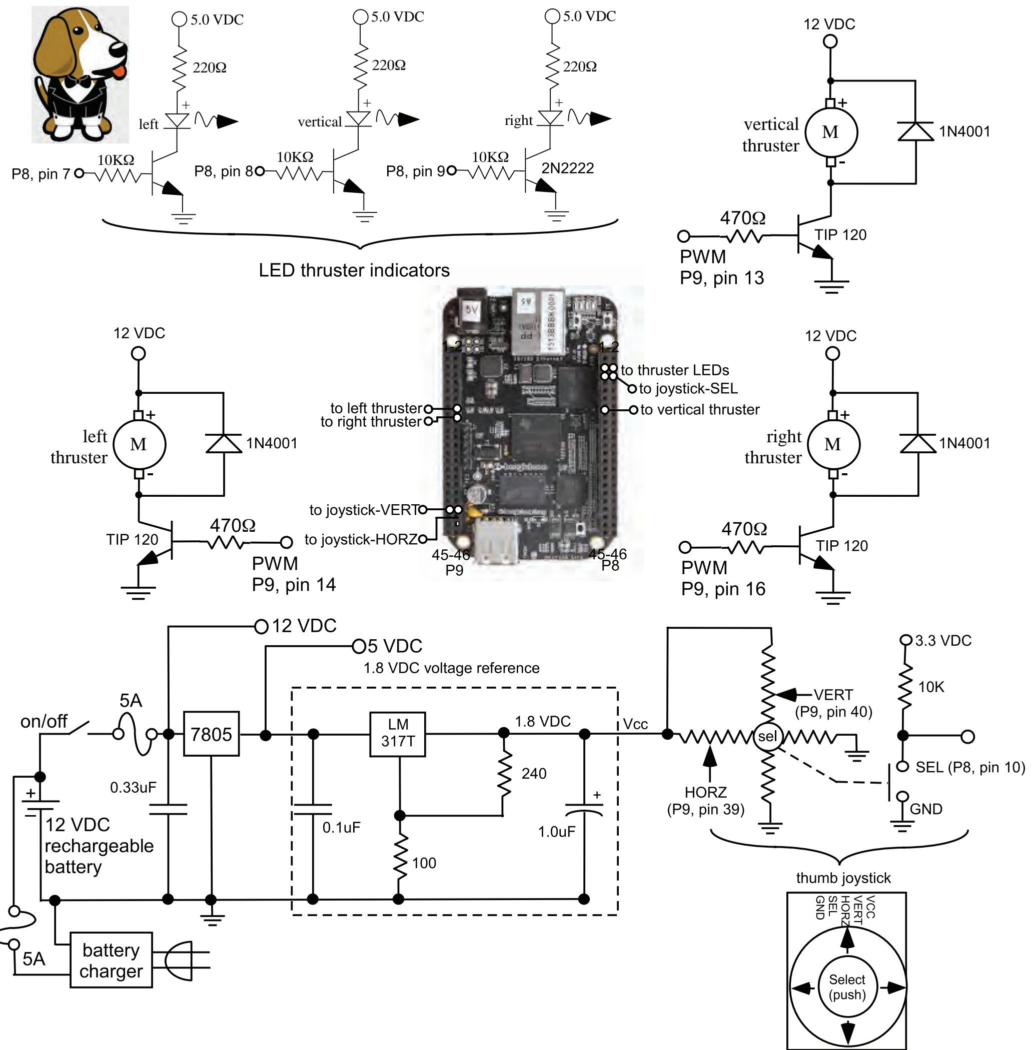


Figure 5.8: SeaPerch ROV interface control. (Illustrations used with permission of Texas Instruments (www.TI.com).)

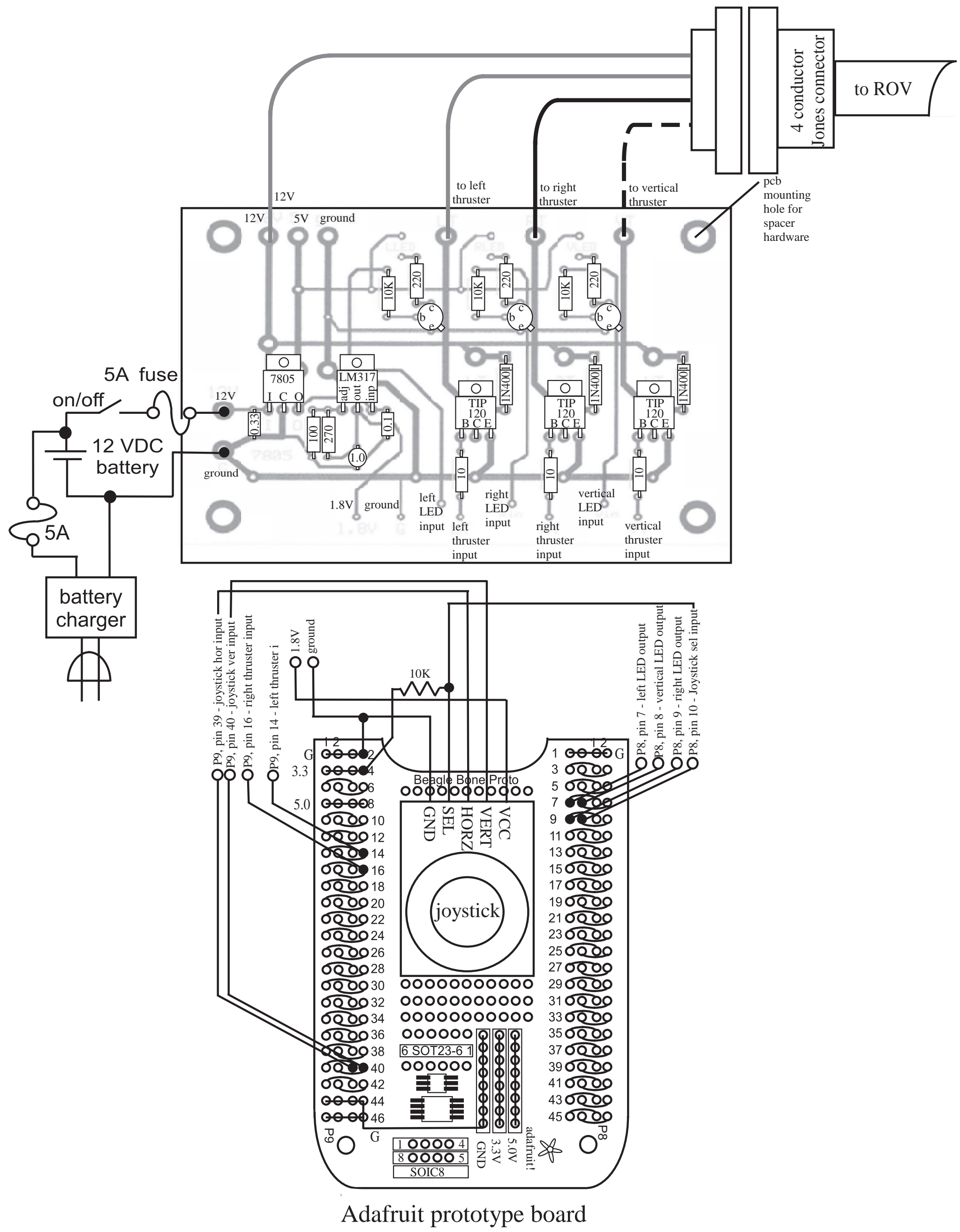


Figure 5.9: SeaPerch ROV printed circuit board interface.

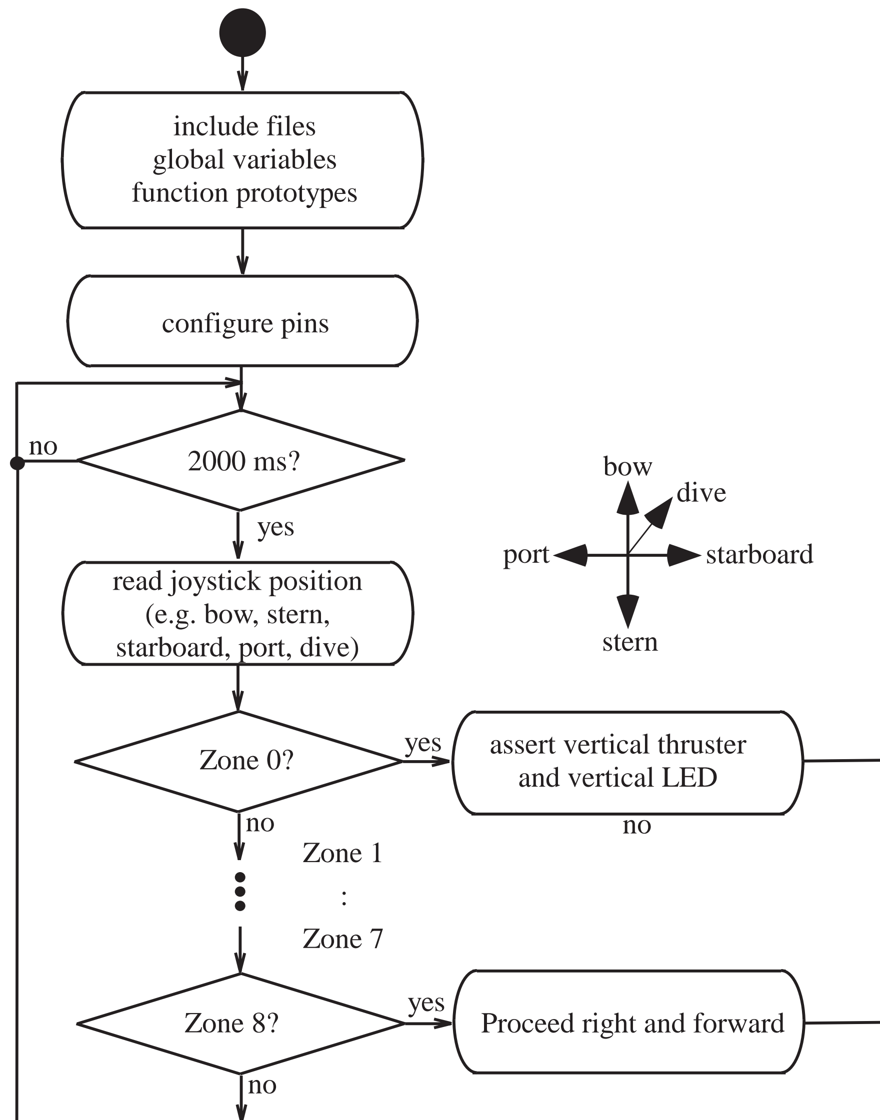


Figure 5.10: SeaPerch ROV UML activity diagram.

from the analog read function corresponding to 0–1.8 VDC. After the voltage readings are taken they are scaled to 1.8 VDC for further processing. Joystick activity is divided into multiple zones (0–8) as shown in Figure 5.11. The joystick signal is further processed consistent with the joystick zone selected.

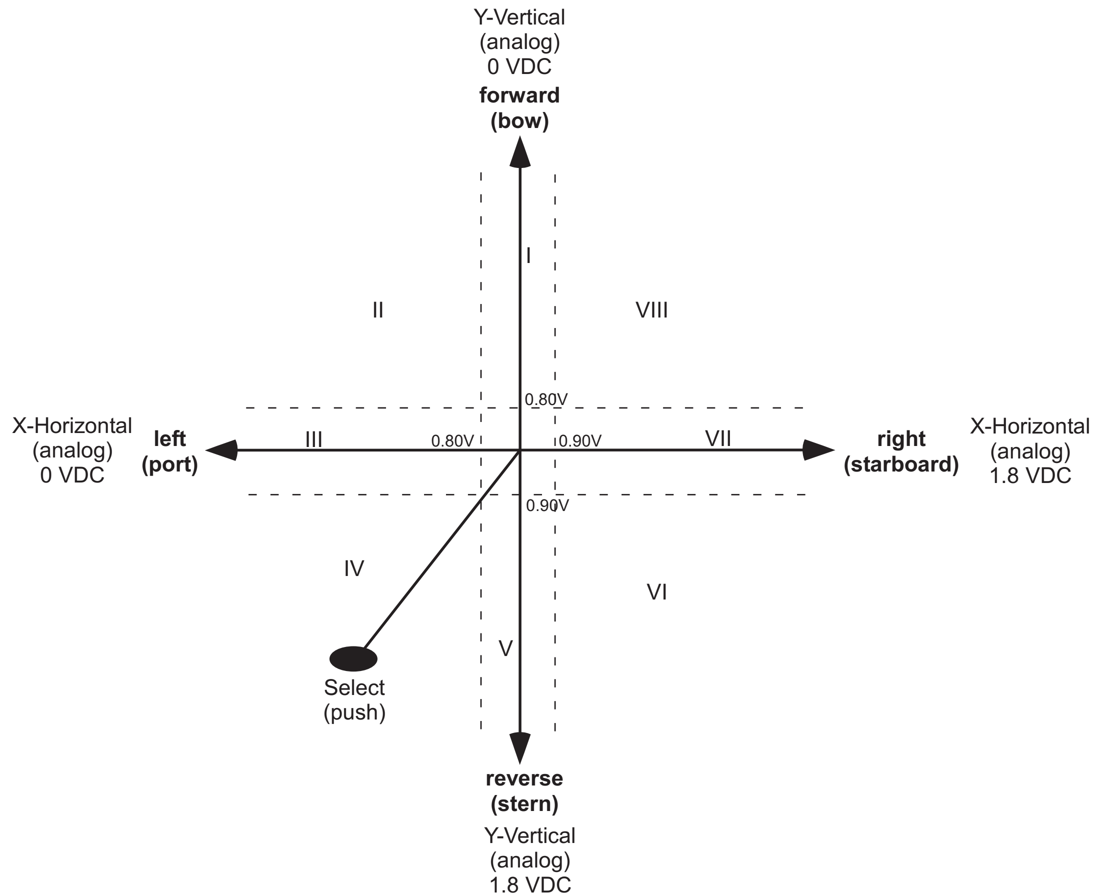


Figure 5.11: Joystick position as related to thruster activity.

```
1 //*****************************************************************
2 //rov.js
3 //*****************************************************************
4
5 var b = require('bonescript');
6
7 //define inputs and outputs
8 var joystick_hor = ``P9_39'';              //joystick horizontal input
```

```
 9 var joystick_ver = ``P9_40'';             //joystick vertical input
10 var joystick_sel = ``P8_10'';             //joystick select input
11
12 var left_thrust_pin  = ``P9_14'';         //PWM output left thruster
13 var right_thrust_pin = ``P9_16'';         //PWM output right thruster
14 var vert_thrust_pin  = ``P8_13'';         //Output output vertical
       thruster
15
16 var left_LED_pin  = ``P8_7'';             //left LED output
17 var vert_LED_pin  = ``P8_8'';             //vertical LED output
18 var right_LED_pin = ``P8_9'';             //right LED output
19
20 //configure input and output pins
21 b.pinMode(joystick_sel, b.INPUT);
22 b.pinMode(left_thrust_pin, b.OUTPUT);
23 b.pinMode(right_thrust_pin, b.OUTPUT);
24 b.pinMode(vert_thrust_pin, b.OUTPUT);
25 b.pinMode(left_LED_pin, b.OUTPUT);
26 b.pinMode(vert_LED_pin, b.OUTPUT);
27 b.pinMode(right_LED_pin, b.OUTPUT);
28
29 //define variables
30 var joystick_horizontal;
31 var joystick_vertical;
32 var joystick_thrust_on;
33 var comments_on = 1;
34
35
36 //Read the joystick and process input
37 setInterval(loop, 2000);
38
39 function loop()
40 {                                            //reset LEDs for new status
41 b.digitalWrite(vert_LED_pin, b.LOW);         //de-assert vertical LED
42 b.digitalWrite(left_LED_pin, b.LOW);         //de-assert left LED
43 b.digitalWrite(right_LED_pin, b.LOW);        //de-assert right LED
44
45 //read horizontal joystick value and scale by 1.8 VDC
46 joystick_horizontal = b.analogRead(joystick_hor);  //read joystick horz
        position
47 joystick_horizontal = joystick_horizontal * 1.8;
48
49 //read vertical joystick value and scale by 1.8 VDC
50 joystick_vertical   = b.analogRead(joystick_ver);  //read joystick ver
       position
51 joystick_vertical = joystick_vertical * 1.8;
52
53 //check for active low assertion of vertical thrust
```

```
54 joystick_thrust_on   = b.digitalRead(joystick_sel); //vertical thrust?
55
56 if(comments_on == 1)
57   {
58   console.log(joystick_horizontal);
59   console.log(joystick_vertical);
60   console.log(joystick_thrust_on);
61   }
62
63 //*************************************************************
64 //*************************************************************
65 //vertical thrust - active low pushbutton on joystick
66 //*************************************************************
67 if(joystick_thrust_on == 0)
68   {
69   b.digitalWrite(vert_thrust_pin, b.HIGH);
70   b.digitalWrite(vert_LED_pin, b.HIGH);
71   if(comments_on ==1) console.log("Thrust is on!")
72   }
73 else
74   {
75   b.digitalWrite(vert_thrust_pin, b.LOW);
76   b.digitalWrite(vert_LED_pin, b.LOW);
77   if(comments_on ==1) console.log("Thrust is off!")
78   }
79
80 //************************************************************
81 //************************************************************
82 //process different joystick zones
83 //************************************************************
84 //Case 0: Joystick in null position
85 //Inputs:
86 //   X channel between 0.80 to 0.90 VDC - null zone
87 //   Y channel between 0.80 to 0.90 VDC - null zone
88 //Output:
89 //   Shut off thrusters
90 //************************************************************
91
92 if((joystick_horizontal > 0.80)&&(joystick_horizontal < 0.90)&&
93    (joystick_vertical > 0.80)&&(joystick_vertical < 0.90))
94
95   {
96   if(comments_on == 1) console.log("Zone 0");
97
98   if(comments_on == 1)
99   {
100  console.log(joystick_horizontal);
101  console.log(joystick_vertical);
```

```
102     console.log(joystick_thrust_on);
103     }
104
105     //assert thrusters to move forward
106     b.analogWrite(left_thrust_pin, 0);
107     b.analogWrite(right_thrust_pin, 0);
108
109     //assert LEDs
110     b.digitalWrite(left_LED_pin, b.LOW);        //de-assert left LED
111     b.digitalWrite(right_LED_pin, b.LOW);       //de-assert right LED
112     }
113
114 //***************************************************************
115 //***************************************************************
116 //process different joystick zones
117 //***************************************************************
118 //Case 1:
119 //Inputs:
120 //   X channel between 0.80 to 0.90 VDC - null zone
121 //   Y channel <= 0.80 VDC
122 //Output:
123 //   Move forward - provide same voltage to left and right thrusters
124 //***************************************************************
125
126 if((joystick_horizontal > 0.80)&&(joystick_horizontal < 0.90)&&
127     (joystick_vertical <= 0.80))
128     {
129     if(comments_on == 1) console.log("Zone 1");
130
131     //scale joystick vertical to value from 0 to 1
132     joystick_vertical = 0.80 - joystick_vertical;
133
134     if(comments_on == 1)
135     {
136     console.log(joystick_horizontal);
137     console.log(joystick_vertical);
138     console.log(joystick_thrust_on);
139     }
140
141     //assert thrusters to move forward
142     b.analogWrite(left_thrust_pin, joystick_vertical);
143     b.analogWrite(right_thrust_pin, joystick_vertical);
144
145     //assert LEDs
146     b.digitalWrite(left_LED_pin, b.HIGH);        //assert left LED
147     b.digitalWrite(right_LED_pin, b.HIGH);       //assert right LED
148     }
149
```

```
150 //*****************************************************************
151 //*****************************************************************
152 //Case 2:
153 //Inputs:
154 //   X channel <= 0.80 VDC
155 //   Y channel <= 0.80 VDC
156 //Output:
157 //   Move forward and bare left
158 //   - Which joystick direction is asserted more?
159 //   - Scale PWM voltage to left and right thruster accordingly
160 //*****************************************************************
161
162 if((joystick_horizontal <= 0.80)&&(joystick_vertical <= 0.80))
163   {
164   if(comments_on == 1) console.log("Zone 2");
165
166   //scale joystick horizontal and vertical to value from 0 to 1
167   joystick_horizontal = 0.80 - joystick_horizontal;
168   joystick_vertical = 0.80 - joystick_vertical;
169
170   if(comments_on == 1)
171   {
172   console.log(joystick_horizontal);
173   console.log(joystick_vertical);
174   console.log(joystick_thrust_on);
175   }
176
177   //assert thrusters and LEDs
178   if(joystick_horizontal > joystick_vertical)
179     {
180     b.analogWrite(left_thrust_pin, (joystick_horizontal -
            joystick_vertical));
181     b.analogWrite(right_thrust_pin, joystick_horizontal);
182
183     //assert LEDs
184     b.digitalWrite(left_LED_pin, b.HIGH);        //assert left LED
185     b.digitalWrite(right_LED_pin, b.HIGH);       //assert right LED
186     }
187   else
188     {
189     b.analogWrite(left_thrust_pin, joystick_vertical);
190     b.analogWrite(right_thrust_pin, (joystick_vertical -
            joystick_horizontal));
191
192     //assert LEDs
193     b.digitalWrite(left_LED_pin, b.HIGH);        //assert left LED
194     b.digitalWrite(right_LED_pin, b.HIGH);       //assert right LED
195     }
```

```
196   }
197
198 //***************************************************************
199 //***************************************************************
200 //Case 3:
201 //Inputs:
202 //   X channel <= 0.80 VDC
203 //   Y channel between 0.80 to 0.90 VDC - null zone
204 //Output:
205 //   Bare left
206 //***************************************************************
207
208 if((joystick_horizontal <= 0.80)&&(joystick_vertical > 0.80)&&
209     (joystick_vertical < 0.90))
210   {
211   if(comments_on == 1) console.log("Zone 3");
212
213   //scale joystick vertical to value from 0 to 1
214   joystick_horizontal = 0.80 - joystick_horizontal;
215
216  if(comments_on == 1)
217   {
218   console.log(joystick_horizontal);
219   console.log(joystick_vertical);
220   console.log(joystick_thrust_on);
221   }
222
223   //assert thrusters
224   b.analogWrite(left_thrust_pin, 0);
225   b.analogWrite(right_thrust_pin, joystick_horizontal);
226
227   //assert LEDs
228   b.digitalWrite(left_LED_pin, b.LOW);        //de-assert left LED
229   b.digitalWrite(right_LED_pin, b.HIGH);      //assert right LED
230   }
231
232 //*****************************************************************
233 //*****************************************************************
234 //Case 4:
235 //Inputs:
236 //   X channel <= 0.80 VDC
237 //   Y channel >= 0.90 VDC
238 //Output:
239 //   Bare left to turn around
240 //*****************************************************************
241
242 if((joystick_horizontal <= 0.80)&&(joystick_vertical >= 0.90))
243   {
```

```
244    if(comments_on == 1) console.log("Zone 4");
245
246    //scale joystick horizontal and vertical to value from 0 to 1
247    joystick_horizontal = 0.80 - joystick_horizontal;
248    joystick_vertical = joystick_vertical - 0.90;
249
250    if(comments_on == 1)
251    {
252    console.log(joystick_horizontal);
253    console.log(joystick_vertical);
254    console.log(joystick_thrust_on);
255    }
256
257     //assert thrusters and LEDs
258    if(joystick_horizontal > joystick_vertical)
259      {
260      b.analogWrite(left_thrust_pin, 0);
261      b.analogWrite(right_thrust_pin, (joystick_horizontal-
           joystick_vertical));
262
263      //assert LEDs
264      b.digitalWrite(left_LED_pin, b.LOW);         //de-assert left LED
265      b.digitalWrite(right_LED_pin, b.HIGH);       //assert right LED
266      }
267    else
268      {
269      b.analogWrite(left_thrust_pin, 0);
270      b.analogWrite(right_thrust_pin, (joystick_vertical-
           joystick_horizontal));
271
272      //assert LEDs
273      b.digitalWrite(left_LED_pin, b.LOW);         //de-assert left LED
274      b.digitalWrite(right_LED_pin, b.HIGH);       //assert right LED
275      }
276     }
277
278 //***************************************************************
279 //***************************************************************
280 //Case 5:
281 //Inputs:
282 //  X channel between 0.80 to 0.90 VDC - null zone
283 //  Y channel >= 0.90 VDC
284 //Output:
285 //  Move backward - provide same voltage to left and right thrusters
286 //***************************************************************
287
288 if((joystick_horizontal > 0.80)&&(joystick_horizontal < 0.90)&&
289     (joystick_vertical >= 0.90))
```

```
290    {
291    if(comments_on ==1) console.log("Zone 5");
292
293    //scale joystick vertical to value from 0 to 1
294    joystick_vertical = joystick_vertical - 0.90;
295
296    if(comments_on == 1)
297    {
298    console.log(joystick_horizontal);
299    console.log(joystick_vertical);
300    console.log(joystick_thrust_on);
301    }
302
303    //assert thrusters
304    b.analogWrite(left_thrust_pin, 0);
305    b.analogWrite(right_thrust_pin, joystick_vertical);
306
307    //assert LEDs
308    b.digitalWrite(left_LED_pin, b.LOW);          //de-assert left LED
309    b.digitalWrite(right_LED_pin, b.HIGH);        //assert right LED
310    }
311
312 //*************************************************************
313 //*************************************************************
314 //Case 6:
315 //Inputs:
316 //  X channel >= 0.90 VDC
317 //  Y channel >= 0.90 VDC
318 //Output:
319 //  Bare left to turn around
320 //*************************************************************
321
322 if((joystick_horizontal >= 0.90)&&(joystick_vertical >= 0.90))
323    {
324    if(comments_on == 1) console.log("Zone 6");
325
326    //scale joystick horizontal and vertical to value from 0 to 1
327    joystick_horizontal = joystick_horizontal - 0.90;
328    joystick_vertical = joystick_vertical - 0.90;
329
330   if(comments_on == 1)
331    {
332    console.log(joystick_horizontal);
333    console.log(joystick_vertical);
334    console.log(joystick_thrust_on);
335    }
336
337     //assert thrusters and LEDs
```

```
338     if(joystick_horizontal > joystick_vertical)
339       {
340       b.analogWrite(left_thrust_pin, (joystick_horizontal-
            joystick_vertical));
341       b.analogWrite(right_thrust_pin, 0);
342
343       //assert LEDs
344       b.digitalWrite(left_LED_pin, b.HIGH);        //assert left LED
345       b.digitalWrite(right_LED_pin, b.LOW);        //de-assert right LED
346       }
347     else
348       {
349       b.analogWrite(left_thrust_pin, (joystick_vertical-
            joystick_horizontal));
350       b.analogWrite(right_thrust_pin, 0);
351
352       //assert LEDs
353       b.digitalWrite(left_LED_pin, b.HIGH);       //assert left LED
354       b.digitalWrite(right_LED_pin, b.LOW);       //de-assert right LED
355       }
356      }
357
358 //*****************************************************************
359 //*****************************************************************
360 //Case 7:
361 //Inputs:
362 //   X channel >= 0.90 VDC
363 //   Y channel between 0.80 to 0.90 VDC - null zone
364 //Output:
365 //   Bare right
366 //*****************************************************************
367
368 if((joystick_horizontal >= 0.90)&&(joystick_vertical > 0.80)&&
369     (joystick_vertical < 0.90))
370   {
371   if(comments_on == 1) console.log("Zone 7");
372
373   //scale joystick vertical to value from 0 to 1
374   joystick_horizontal = joystick_horizontal - 0.90;
375
376   if(comments_on == 1)
377   {
378   console.log(joystick_horizontal);
379   console.log(joystick_vertical);
380   console.log(joystick_thrust_on);
381   }
382
383   //assert thrusters
```

```
384    b.analogWrite(left_thrust_pin, joystick_horizontal);
385    b.analogWrite(right_thrust_pin, 0);
386
387    //assert LEDs
388    b.digitalWrite(left_LED_pin, b.HIGH);          //assert left LED
389    b.digitalWrite(right_LED_pin, b.LOW);         //de-assert right LED
390    }
391
392 //****************************************************************
393 //****************************************************************
394 //Case 8:
395 //Inputs:
396 //  X channel >= 0.90 VDC
397 //  Y channel <= 0.80 VDC
398 //Output:
399 //  Move forward and bare right
400 //  - Which joystick direction is asserted more?
401 //  - Scale PWM voltage to left and right thruster accordingly
402 //****************************************************************
403
404 if((joystick_horizontal >= 0.90)&&(joystick_vertical <= 0.80))
405    {
406    if(comments_on == 1) console.log("Zone 8");
407
408    //scale joystick horizontal and vertical to value from 0 to 1
409    joystick_horizontal = joystick_horizontal - 0.90;
410    joystick_vertical = 0.80 - joystick_vertical;
411
412    if(comments_on == 1)
413    {
414    console.log(joystick_horizontal);
415    console.log(joystick_vertical);
416    console.log(joystick_thrust_on);
417    }
418
419    //assert thrusters and LEDs
420    if(joystick_horizontal > joystick_vertical)
421      {
422      b.analogWrite(left_thrust_pin, joystick_horizontal);
423      b.analogWrite(right_thrust_pin, (joystick_horizontal-
           joystick_vertical));
424
425      //assert LEDs
426      b.digitalWrite(left_LED_pin, b.HIGH);          //assert left LED
427      b.digitalWrite(right_LED_pin, b.HIGH);         //assert right LED
428      }
429    else
430      {
```

```
431        b.analogWrite(left_thrust_pin, (joystick_vertical-
             joystick_horizontal));
432        b.analogWrite(right_thrust_pin, joystick_vertical);
433
434        //assert LEDs
435        b.digitalWrite(left_LED_pin, b.HIGH);         //assert left LED
436        b.digitalWrite(right_LED_pin, b.HIGH);        //assert right LED
437        }
438    }
439 }
440
441 //*************************************************************
442 //*************************************************************
```

5.4.8 CONTROL HOUSING LAYOUT

A Plano Model 1312-00 water-resistant field box is used to house the control circuitry and rechargeable battery. The battery is a rechargeable, sealed, lead-acid battery, 12 VDC, with an 8.5 amp-hour capacity. It is available from McMaster-Carr (#7448K82). A battery charger (12 VDC, 4–8 amp-hour rating) is also available (#7448K67). The layout for the ROV control housing is provided in Figure 5.12.

The control circuitry consists of two connected plastic panels as shown in Figure 5.12. The top panel has the on/off switch, the LED thruster indicators (left, dive, and right), an access hole for the joystick, and a 1/4 in jack for the battery recharge cable.

The lower panel is connected to the top panel using aluminum spacers, screws, and corresponding washers, lock washers, and nuts. The lower panel contains BeagleBone Black equipped with the Adafruit Cape configured with the thumb joystick. The BeagleBone Black is connected to the lower panel using a Jameco stand off kit (#106551). The BeagleBone Black is interfaced to the thrusters via interface circuitry described in Figures 5.8 and 5.9. The interface printed circuit board is connected to the four conductor thruster cable via a 4-conductor Jones connector.

5.4.9 FINAL ASSEMBLY TESTING

The final system is tested a subassembly at a time. The following sequence is suggested.

- Recheck all waterproofed connections. Reapply waterproof caulk as necessary.
- With the Adafruit Cape disconnected from BeagleBone Black, test each LED indicator (left, dive, and right). This is accomplished by applying a 3.3 VDC signal in turn to Adafruit Cape pins P8, pin 7; P8, pin 8; and P8, pin 9.
- In a similar manner each thruster (left, right, and vertical) may be tested. If available, a signal generator may be used to generate a pulse width modulated signal to test each thruster.

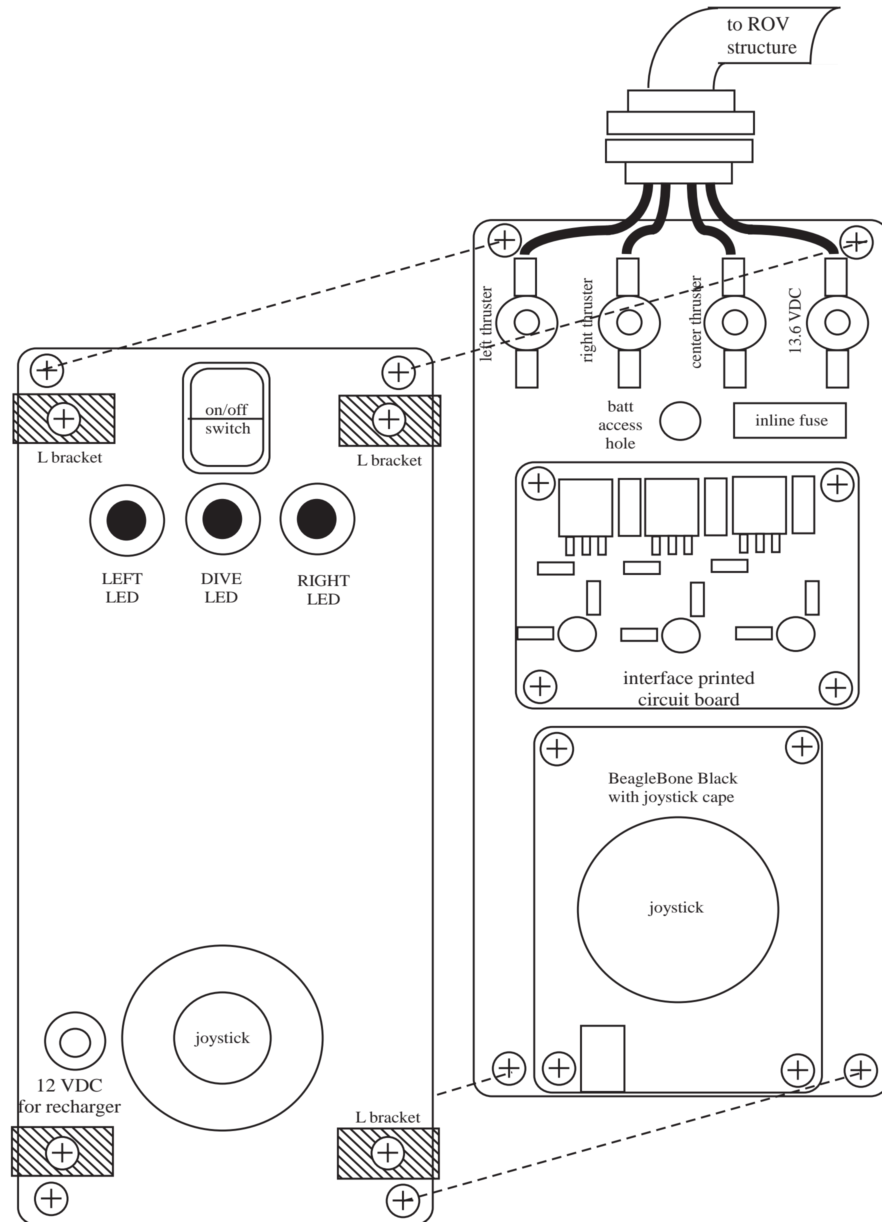


Figure 5.12: ROV control housing layout.

- With power applied, the voltage regulators aboard the printed circuit board should be tested for proper voltages.
- The output voltages from the thumb joystick may be verified at header P9, pin 39 and P9, pin 40 and also the select pushbutton at header P8, pin 10.
- With the software fully functional, the Adafruit Cape may be connected to BeagleBone Black for end-to-end testing.

5.4.10 FINAL ASSEMBLY

The fully assembled ROV is shown in Figure 5.13.

5.4.11 PROJECT EXTENSIONS

The control system provided above has a set of very basic features. Here are some possible extensions for the system.

- Provide a powered dive and surface thruster. To provide for a powered dive and surface capability, the ROV must be equipped with a vertical thruster equipped with an H-bridge to allow for motor forward and reversal. This modification is given as an assignment at the end of the chapter.
- Left and right thruster reverse. Currently, the left and right thrusters may only be powered in one direction. To provide additional maneuverability, the left and right thrusters could be equipped with an H-bridge to allow bi-directional motor control. This modification is given as an assignment at the end of the chapter.
- Proportional speed control with bi-directional motor control. Both of these advanced features may be provided by driving the H-bridge circuit with PWM signals. This modification is given as an assignment at the end of the chapter.

5.5 MOUNTAIN MAZE NAVIGATING ROBOT

In this project we extend the Dagu Magician maze navigating project described in Chapter 3 to a three-dimensional mountain pass. Also, we use a robot equipped with four motorized wheels. Each of the wheels is equipped with an H-bridge to allow bidirectional motor control. In this example we will only control two wheels. We leave the development of a 4WD robot as an end of chapter homework assignment.

5.5.1 DESCRIPTION

For this project, a DF Robot 4WD mobile platform kit was used (DFROBOT ROB0003, Jameco #2124285). The robot kit is equipped with four powered wheels. As in the Dagu Magician project,

Figure 5.13: ROV fully assembled. (Photo courtesy of Barrett [2015].)

we equipped the DF Robot with three Sharp GP12D IR sensors, as shown in Figure 5.14. The robot will be placed in a three-dimensional maze with reflective walls modeled after a mountain pass. The goal of the project is for the robot to detect wall placement and navigate through the maze. The robot will not be provided any information about the maze. The control algorithm for the robot is hosted on BeagleBone.

5.5.2 REQUIREMENTS

The requirements for this project are simple, the robot must autonomously navigate through the maze without touching maze walls as quickly as possible. Furthermore, the robot must be able to safely navigate through the rugged maze without becoming "stuck" on maze features.

5.5.3 CIRCUIT DIAGRAM

The circuit diagram for the robot is provided in Figure 5.15. The three IR sensors (left, middle, and right) are mounted on the leading edge of the robot to detect maze walls. The sensors' outputs are fed to three separate analog-to-digital (ADC) channels. The robot motors will be driven by PWM channels via an H-bridge. The robot is powered by a 7.5 VDC battery pack (5 AA batteries) which is fed to a 3.3 and 5 VDC voltage regulator. Alternatively, the robot may be powered by a 7.5 VDC power supply rated at several amps. In this case, the power is delivered to the robot by a flexible umbilical cable. The circuit diagram includes the inertial measurement unit (IMU) to measure vehicle tilt and a liquid crystal display. Both were discussed in Chapter 3.

5.5.4 STRUCTURE CHART

The structure chart for the robot project is provided in Figure 5.16.

5.5.5 UML ACTIVITY DIAGRAMS

The UML activity diagram for the robot is provided in Figure 5.17.

5.5.6 BONESCRIPT CODE

The code for the robot may be adapted from that for the Blinky602A robot. Since the motors are equipped with an H-bridge, slight modifications are required to the robot turning code. These modifications include an additional signal ($forward/\overline{reverse}$) for each H-bridge configuration to provide forward and reverse capability. For example, when forward is desired a PWM signal is delivered to one side of the H-bridge and a logic zero to the other side. A level shifter (Texas Instruments PCA9306) is used to adapt the 3.3 VDC signal output from BeagleBone to 5.0 VDC levels.

We only provide the basic framework for the Bonescript code here. Throughout Chapters 1–4 we have provided re-useable Bonescript code to meet system requirements.

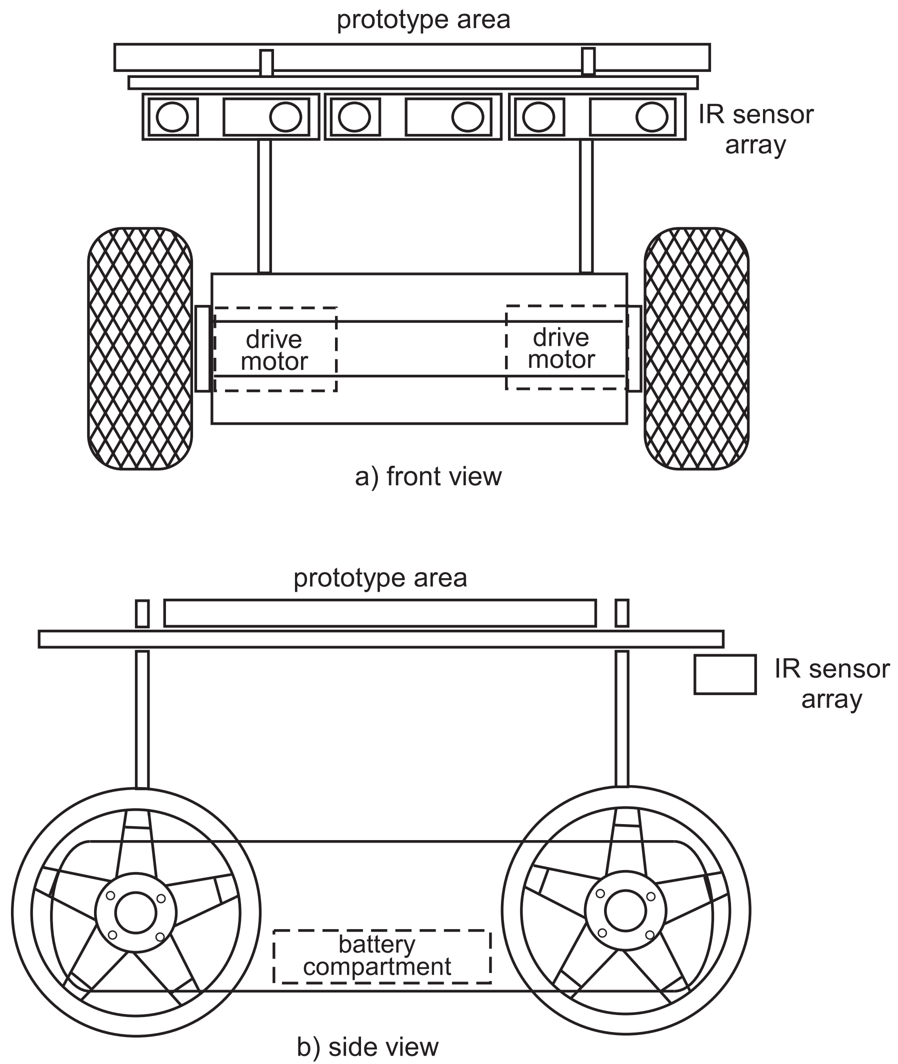


Figure 5.14: Robot layout.

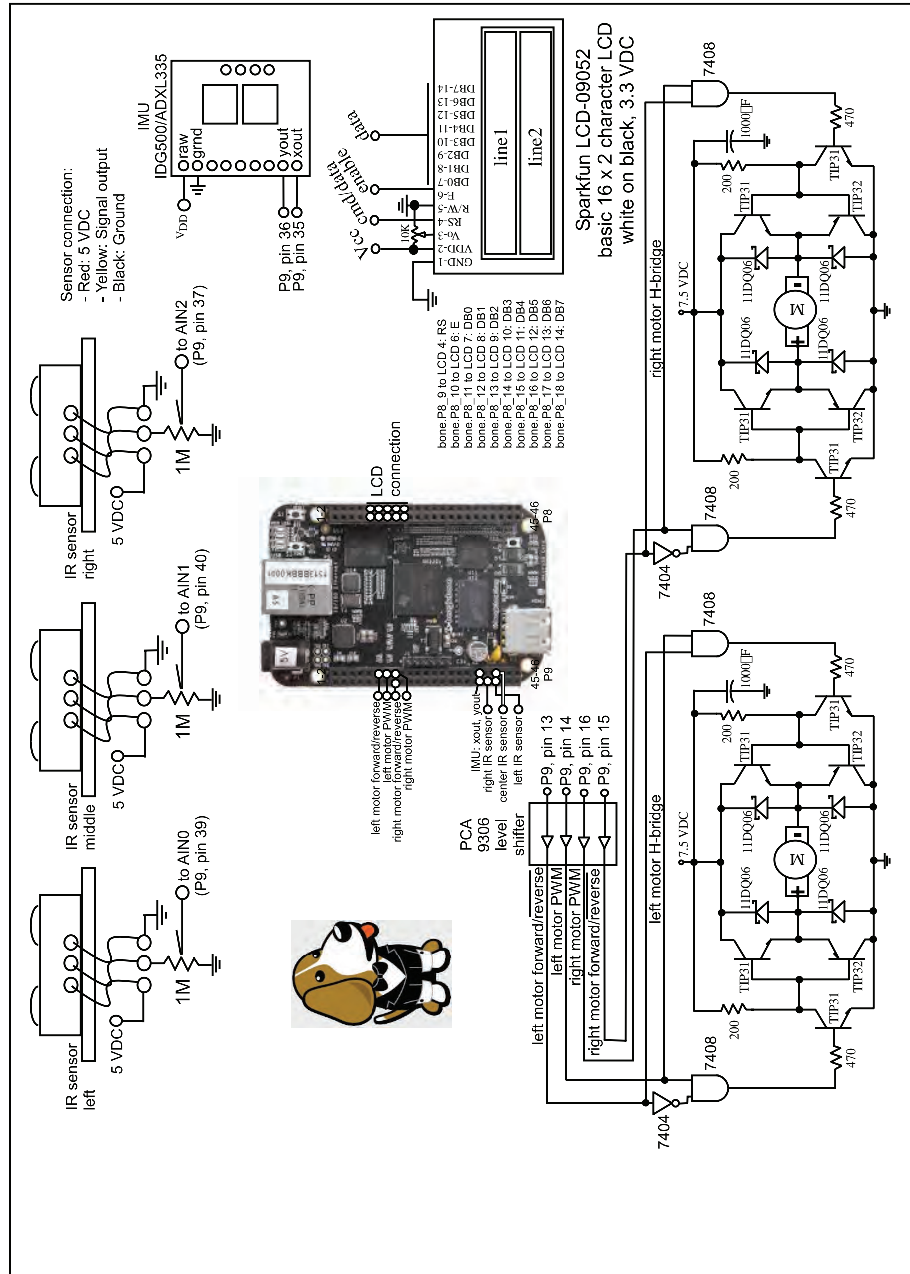


Figure 5.15: Robot circuit diagram. (Illustrations used with permission of Texas Instruments (www.TI.com).)

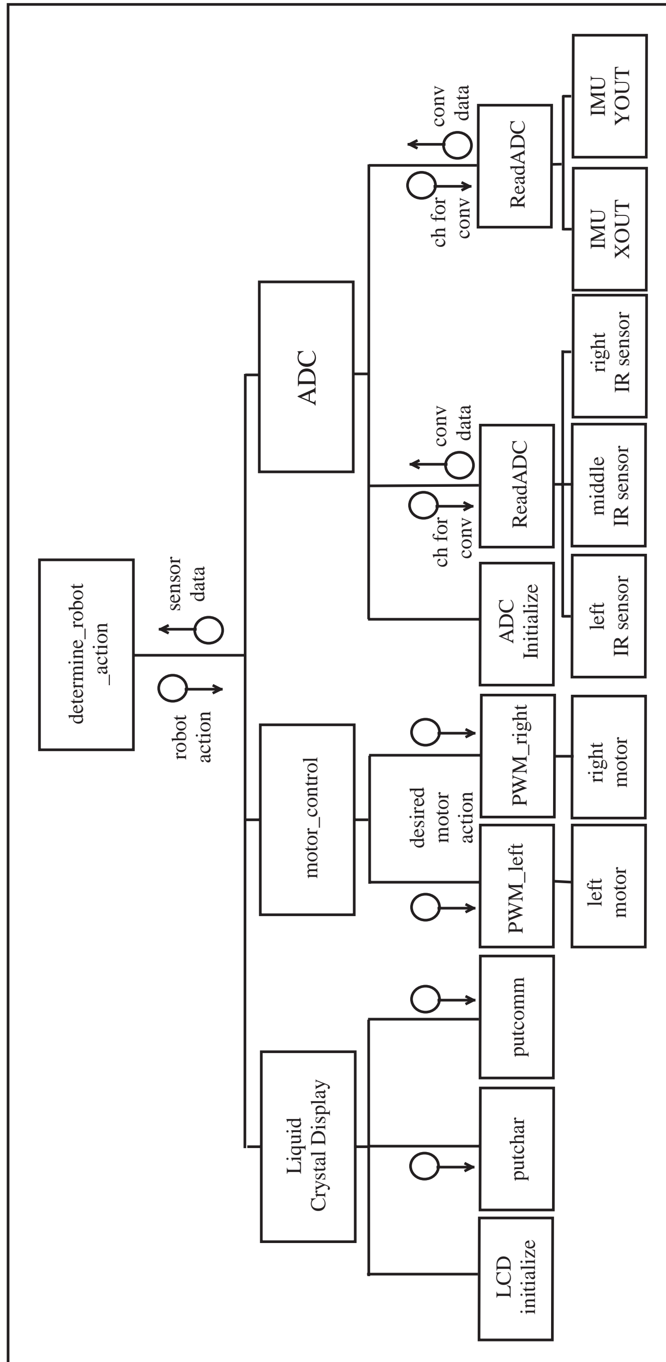


Figure 5.16: Robot structure diagram.

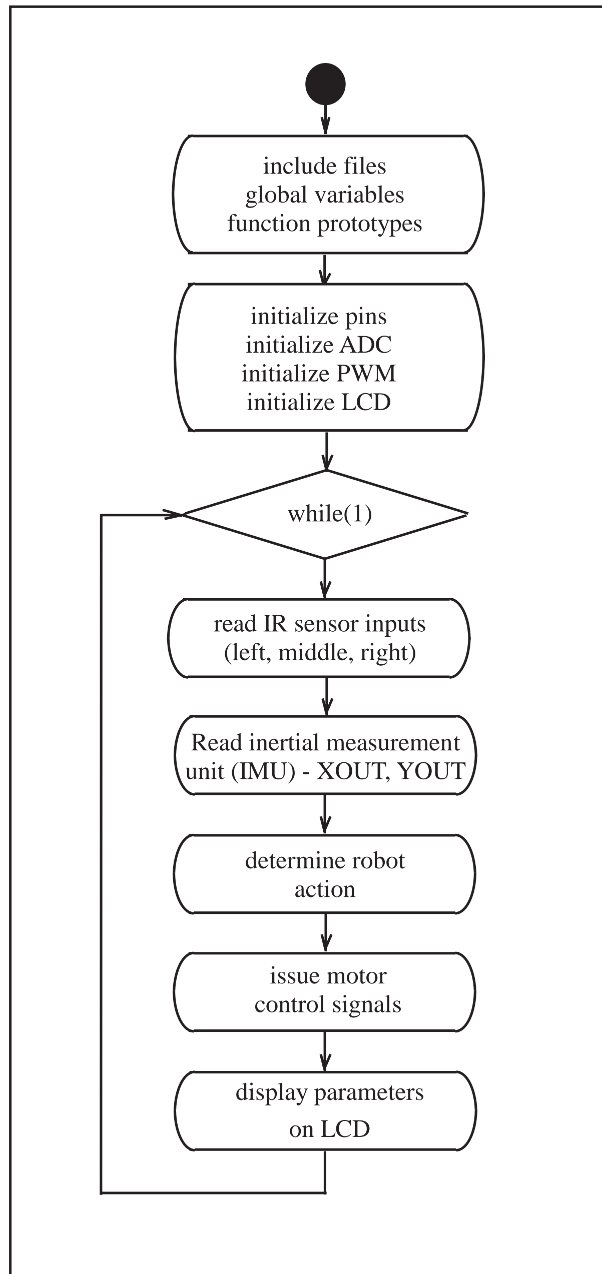


Figure 5.17: Abbreviated robot UML activity diagram. The "determine robot action" consists of multiple decision statements.

```
1 //*************************************************************
2 var b = require('bonescript');
3
4
5 var left_IR_sensor   = ``P9_39'';    //analog input for left IR sensor
6 var center_IR_sensor = ``P9_40'';    //analog input for center IR
      sensor
7 var right_IR_sensor  = ``P9_37'';    //analog input for right IR sensor
8
9 var IMU_xout = ``P9_35'';            //IMU xout analog signal
10 var IMU_yout = ``P9_36'';           //IMU yout analog signal
11
12 var left_motor_pin = ``P9_14'';     //PWM pin for left motor
13 var right_motor_pin = ``P9_16'';    //PWM pin for right motor
14 var left_motor_for_rev = ``P9_13''; //left motor forward/reverse
15 var right_motor_for_rev= ``P9_15''; //right motor forward/reverse
16
17 var left_sensor_value;
18 var center_sensor_value;
19 var right_sensor_value;
20
21
22 b.pinMode(left_motor_pin, b.OUTPUT);          //set pin to digital output
23 b.pinMode(right_motor_pin, b.OUTPUT);         //set pin to digital output
24 b.pinMode(left_motor_for_rev, b.OUTPUT);      //set pin to digital output
25 b.pinMode(right_motor_for_rev, b.OUTPUT);     //set pin to digital output
26
27 while(1)
28   {
29                                   //read analog output from IR sensors
30                                   //normalized value ranges from 0..1
31   left_sensor_value = b.analogRead(left_IR_sensor);
32   center_sensor_value = b.analogRead(center_IR_sensor);
33   right_sensor_value = b.analogRead(right_IR_sensor);
34
35                                   //assumes desired threshold at
36                                   //1.25 VDC with max value of 1.75
                                       VDC
37     if((left_sensor_value > 0.714)&&
38        (center_sensor_value <= 0.714)&&
39        (right_sensor_value > 0.714))
40       {                                         //robot straight ahead
41       b.digitalWrite(left_motor_for_rev, b.HIGH); //left motor forward
42       b.analogWrite(left_motor_pin, 0.7);         //left motor RPM
43       b.digitalWrite(right_motor_for_rev, b.HIGH);//right motor forward
44       b.analogWrite(right_motor_pin, 0.7);        //right motor RPM
45       }
46 /*
```

```
47      else if(...)
48        {
49        :
50        :
51        :
52        }
53 */
54   }
55
56 //*****************************************************************
```

5.5.7 MOUNTAIN MAZE

The mountain maze was constructed from plywood, chicken wire, expandable foam, plaster cloth, and Bondo. A rough sketch of the desired maze path was first constructed. Care was taken to insure the pass was wide enough to accommodate the robot. The maze platform was constructed from 3/8 in plywood on 2 by 4 in framing material. Maze walls were also constructed from the plywood and supported with steel L brackets.

With the basic structure complete, the maze walls were covered with chicken wire. The chicken wire was secured to the plywood with staples. The chicken wire was then covered with plaster cloth (Creative Mark Artist Products #15006). To provide additional stability, expandable foam was sprayed under the chicken wire (Guardian Energy Technologies, Inc. Foam It Green 12). The mountain scene was then covered with a layer of Bondo for additional structural stability. Bondo is a two-part putty that hardens into a strong resin. Mountain pass construction steps are illustrated in Figure 5.18. The robot is shown in the maze in Figure 5.19.

5.5.8 PROJECT EXTENSIONS

- Modify the turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.
- Develop a function for reversing the robot.
- Equip the motor with another IR sensor that looks down toward the maze floor for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate the maze by moving slowly back and forth for 3 s and flashing a large LED.
- The current design is a two-wheel, front-wheel drive system. Modify the design for a two-wheel, rear-wheel drive system.
- The current design is a two-wheel, front-wheel drive system. Modify the design for a 4WD system.

Figure 5.18: Mountain maze.

Figure 5.19: Robot in maze. (Photo courtesy of Barrett [2013]).

- Develop a 4WD system which includes a tilt sensor. The robot should increase motor RPM (duty cycle) for positive inclines and reduce motor RPM (duty cycle) for negatives inclines.
- Equip the robot with an analog inertial measurement unit (IMU) to measure vehicle tilt. Use the information provided by the IMU to optimize robot speed going up and down steep grades.

5.6 SUMMARY

In this chapter, we discussed the design process, related tools, and applied the process to a real-world design. As previously mentioned, this design example will be periodically revisited throughout the text. It is essential to follow a systematic, disciplined approach to embedded systems design to successfully develop a prototype that meets established requirements.

5.7 REFERENCES

- Anderson, M. "Help Wanted: Embedded Engineers Why the United States is losing its edge in embedded systems." *IEEE–USA Today's Engineer*, Feb 2008.
- Barret, J. 2015. "Closer to the Sun Internationl." www.closertothesungallery.com.
- Barrett, S. and Pack, D. 2008. *Atmel AVR Processor Primer Programming and Interfacing.* Morgan & Claypool Publishers; www.morganclaypool.com.
- Barrett, S. and Pack, P. 2005. *Embedded Systems Design and Applications with the 68HC12 and HCS12.* Upper Saddle River, NJ: Pearson Prentice Hall.
- Barrett, S. and Pack, D. 2006. *Processors Fundamentals for Engineers and Scientists.* Morgan & Claypool Publishers; www.morganclaypool.com.
- Bohm, H. and Jensen, V. 2012. *Build Your Own Underwater Robot and Other Wet Projects.* 11th ed., Vancouver, B.C. Canada: Westcoast Words.
- Christ, R. and Wernli, Sr., R. 2014. *The ROV Manual–A User Guide for Remotely Operated Vehicle.* 2nd ed., Oxford. U.K.: Butterworth-Heinemann imprint of Elsevier.
- Dale, N. and Lilly, S.C. 1995. *Pascal Plus Data Structures.* 4th ed. Englewood Cliffs, NJ: Jones and Bartlett.
- Fowler, M. and Scott, K. 2000, *UML Distilled A Brief Guide to the Standard Object Modeling Language.* 2nd ed. Boston, MA: Addison-Wesley.
- *Seaperch.* 2015; www.seaperch.com.

5.8 CHAPTER EXERCISES

1. What is an embedded system?
2. What aspects must be considered in the design of an embedded system?
3. What is the purpose of the structure chart, UML activity diagram, and circuit diagram?
4. Why is a system design only as good as the test plan that supports it?
5. During the testing process, when an error is found and corrected, what should now be accomplished?
6. Discuss the top-down design, bottom-up implementation concept.
7. Describe the value of accurate documentation.

8. What is required to fully document an embedded systems design?
9. For the Dagu Magician robot, modify the PWM turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.
10. For the Dagu Magician robot, equip the motor with another IR sensor that looks down for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate it by rotating about its center axis three times and flashing a large LED while rotating.
11. For the Dagu Magician robot, develop a function for reversing the robot.
12. Provide a powered dive and surface thruster for the SeaPerch ROV. To provide for a powered dive and surface capability, the ROV must be equipped with a vertical thruster equipped with an H-bridge to allow for motor forward and reversal. This modification is given as an assignment at the end of the chapter.
13. Provide a left- and right-thruster reverse for the SeaPerch ROV. Currently, the left and right thrusters may only be powered in one direction. To provide additional maneuverability, the left and right thrusters could be equipped with an H-bridge to allow bi-directional motor control. This modification is given as an assignment at the end of the chapter.
14. Provide proportional speed control with bi-directional motor control for the SeaPerch ROV. Both of these advanced features may be provided by driving the H-bridge circuit with PWM signals. This modification is given as an assignment at the end of the chapter.
15. For the 4WD robot, modify the PWM turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.
16. For the 4WD robot, equip the motor with another IR sensor that looks down for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate it by rotating about its center axis three times and flashing a large LED while rotating.
17. For the 4WD robot, develop a function for reversing the robot.
18. For the 4WD robot, the current design is a two-wheel, front-wheel drive system. Modify the design for a two-wheel, rear wheel drive system.
19. For the 4WD robot, the current design is a two-wheel, front-wheel drive system. Modify the design for a 4WD system.
20. For the 4WD robot, develop a 4WD system which includes a tilt sensor. The robot should increase motor RPM (duty cycle) for positive inclines and reduce motor RPM (duty cycle) for negatives inclines.

21. Equip the robot with an inertial measurement unit (IMU) to measure vehicle tilt. Use the information provided by the IMU to optimize robot speed going up and down steep grades.

22. Develop an embedded system controlled dirigible/blimp (`www.microflight.com`, `www.rctoys.com`).

23. Develop a trip odometer for your bicycle (hint: use a Hall Effect sensor to detect tire rotation).

24. Develop a timing system for a four-lane Pinewood Derby track.

25. Develop a playing board and control system for your favorite game (Yahtzee, Connect Four, Battleship, etc.).

26. You have a very enthusiastic dog that loves to chase balls. Develop a system to launch balls for the dog.

CHAPTER 6

BeagleBone Features and Subsystems

Objectives: After reading this chapter, the reader should be able to do the following.

- Use the Linux operating system to communicate and interact with BeagleBone.
- Describe how the host Linux personal computer (PC) interacts with BeagleBone.
- Employ the Linux tool chain aboard BeagleBone to write, compile, and execute a C and C++ program.
- Describe the features and subsystems of the ARM Cortex A8 processor.
- Describe the operation of BeagleBone exposed functions available via the P8 and P9 expansion headers.
- Interact with BeagleBone exposed functions using device tree overlays and the Linux operating system.
- Program BeagleBone exposed functions using the Linux tool chain.
- Employ the Programmable Real-Time Unit (PRU) Subsystem for real time applications.

6.1 OVERVIEW

In the first five chapters of this book we employed BeagleBone as a user-friendly computer. We accessed its features and subsystems via the browser-based Bonescript programming environment. In the remainder of the book, we shift focus and "unleash" the power of BeagleBone as a Linux-based, 32-bit, super-scalar ARM Cortex A8 computer. We begin with an overview of methods to communicate with BeagleBone Black and a brief review of the C and C++ tool programming chain. This is followed by examples on how to interact with the digital and analog pins aboard the processor. We then take a closer look at the features and subsystems available aboard BeagleBone. We spend a good part of the chapter describing the exposed functions of BeagleBone. These are the functions accessible to the user via the P8 and P9 extension headers. We conclude the chapter with an introduction to the onboard PRUs. Throughout the chapter we provide sample programs on how to interact with and program the exposed functions.

6.2 BEAGLING IN LINUX

For system development BeagleBone is usually connected to a host desktop PC or laptop via a USB cable. The cable is provided with BeagleBone. An Ethernet cable may also be used to connect the host with BeagleBone in certain applications. It is not provided with BeagleBone but may be purchased at a local retail outlet, electronics supply store, or office supply store. The configuration forms a "mini-network" between the host computer and BeagleBone via the USB cable. One may wonder why a configuration like this is used. It must be emphasized that BeagleBone is a not a microcontroller-based project board. It is instead a fully functional, powerful, Linux-based, expandable computer. The network configuration allows BeagleBone to share some of the useful peripheral features of the host computer during project development (e.g., keyboard, display, etc.). In Chapter 7 we describe how to implement a standalone BeagleBone based expandable computer with its own peripheral devices.

It is important to note for ease of development the host PC or laptop must be equipped with the Linux operating system. Details on how to load Linux are provided in the next section. No prior use or familiarity of Linux is expected of the reader. Linux is a powerful operating system originally developed by Linus Torvald of Finland in the early 1990's. Since its initial release, a variety of Linux distributions have been developed. A Linux distribution consists of the Linux operating system kernel, a collection of useful software applications and also installation software. Linux distributions are given distinct names. As an example, we install Ubuntu Linux on the host computer [Dulaney, 2010].

As shown in Figure 6.1, the host computer uses Linux release Ubuntu. There are several options on running Linux. Linux may be run in a Live mode where it is executed from a companion CD or a USB drive. In this case, the Linux operating system is not loaded on the host computer's hard drive. Alternatively, Linux may be installed on the host computer's hard drive. Linux may be installed as the only operating system for the computer or Linux may share the hard drive with another operating system such as Windows or OS X. If Linux will share the hard drive space, the hard drive must first be partitioned and space set aside for the Linux operating system. In the next section we discuss how to partition the hard drive and install Ubuntu Linux alongside Microsoft Windows on the same PC hard drive.

BeagleBone Black originally shipped with the Ångström Linux Distribution. Starting in early 2014, BeagleBone Black has the Debian Linux operating system pre-installed on the eMMC Flash drive. This Linux distribution also includes a wide variety of useful applications. BeagleBone also arrives pre-installed with the Cloud9 IDE and Bonescript. Bonescript is undergoing rapid development. It is essential that the most recent version of Bonescript is loaded on BeagleBone. Instructions on updating the onboard Bonescript is provided in an upcoming section. The latest software is available from `www.BeagleBoard.org`.

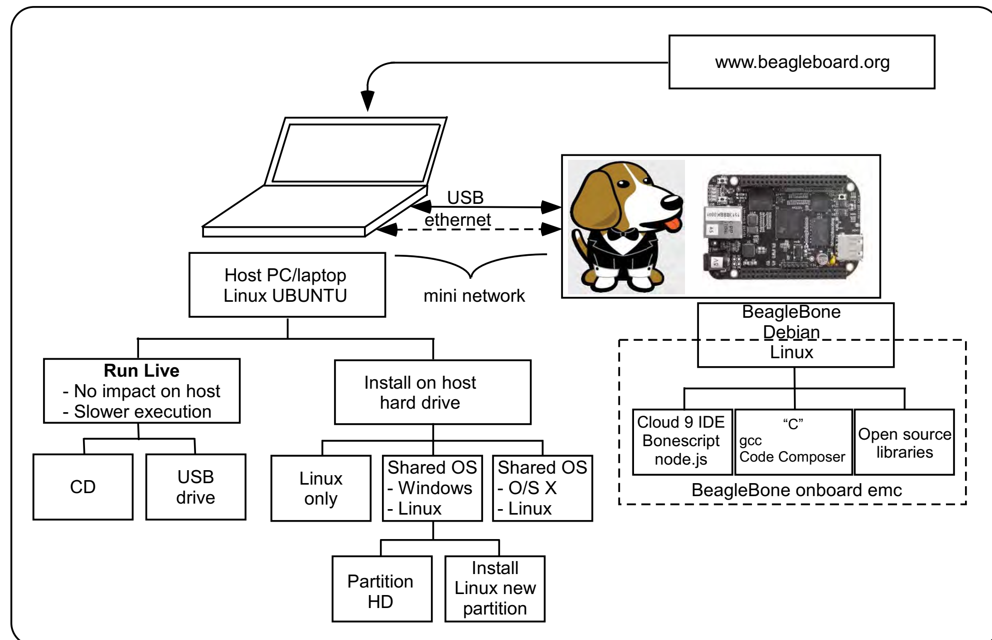


Figure 6.1: BoneScript processing environment in Linux.

6.2.1 COMMUNICATION WITH BEAGLEBONE BLACK

BeagleBone designers provided considerable flexibility in connecting the host computer to BeagleBone. In this section we provide a brief overview of the four methods of connecting BeagleBone to a host computer. Bill Traynor provides an excellent step-by-step instructions at `www.elinux.org/Beagleboard:TerminalShells` on various methods to connect BeagleBone Black to a host PC.

Here is a summary of several methods to connect BeagleBone Black to a host computer [Traynor, 2005]:

- Access BeagleBone Black via its IP address (`192.168.7.2`) using the USB cable. This technique is recommended for Bonescript processing. This technique was discussed in Chapter 1. Once within Bonescript, a Linux operating system prompt is available to access BeagleBone Black. As a friendly reminder, Chrome and Firefox are the preferred web browsers for use with BeagleBone Black.

- Access BeagleBone Black via a secure shell (SSH) using an Ethernet connection. This is the preferred method when the BeagleBone is in a remote location and accessible via an Ethernet connection or via a WiFi dongle. We cover this technique later in the chapter under networking techniques.

- Serial connection via a USB cable. BeagleBone is accessed via PuTTy, Terra Term, or a Google Chrome add-on secure shell. These are all terminal emulators which allow the host computer to interact with the BeagleBone remotely over the USB cable.
- Access BeagleBone Black via UART Channel 0 using a USB to TTL serial cable. We cover this technique later in the chapter when discussing the UART system aboard BeagleBone Black.

In the next several examples, we provide step-by-step instructions to connect BeagleBone Black to the host computer in MS Windows and then in Linux.

Example 1: Accessing Linux via Bonescript through your browser. For the novice user, the quickest way to get access to Linux is via Bonescript through your web browser. This is accomplished using the "Quick-Start Guide" that ships with BeagleBone Black [www.beagleboard.org].

1. Plug BeagleBone into the host computer via the mini-USB capable and open the START.htm file.
2. Install the proper drivers for your system. MS Windows users need to first determine if the host computer is running 32- or 64-bit Windows. Instructions to do this are provided at http://support.microsoft.com/kb/827218.
3. Browse to "Information on BeagleBoard" information using Chrome or Firefox by going to http://192.168.7.2.
4. Explore the Cloud9 IDE develop environment by navigating to http://192.168.7.2:3000/ using Chrome or Firefox.
5. A Linux prompt is provided in the lower window of the Cloud9 IDE.

Example 2: MS Windows connection to BeagleBone Black using Google Chrome add-on secure shell. In this example, we establish connection between the host computer hosting MS Windows and BeagleBone Black. These instructions are adapted from Bill Traynor's instructions at www.elinux.org/Beagleboard:TerminalShells.

- If not currently equipped, download and install the Google Chrome web browser. It is available for free download at www.google.com/chrome.
- Connect the host computer to BeagleBone Black using the USB cable.
- Download the secure shell (SSH) application (Secure Shell 0.8.19) from https://chrome.google.com/webstore.
- You will need to open an account with the webstore (please note this is a free download).

- Once downloaded and installed, access the installed Google Chrome Applications (Apps) using (Cntl+n). Press the "Secure Shell" button to launch the application.
- In the "username@hostname or free form text" insert the root location for BeagleBone Black (root@192.168.7.2) and press [Enter].
- The SSH application establishes a connection between the host computer and BeagleBone Black.
- The BeagleBone prompt will appear in the SSH application window.
- When prompted, answer "yes" to continue and [Enter] when prompted for a password.

Example 3: Linux Ubuntu connection to BeagleBone Black. In this example, Ubuntu Linux will be loaded on a host computer that already has Microsoft Windows installed. A connection is then established between the host computer and BeagleBone. The steps required to establish the network connection include the following.

- Partition the hard drive on the host computer using the GParted utility provided within the Linux Ubuntu Distribution.
- Install Ubuntu Linux on the established partition.
- Configure the network between the host computer and BeagleBone.
- Start the Could9 Integrated Development Environment (IDE) resident on the BeagleBone eMMC flash drive).

A detailed diagram of the startup steps is provided in Figure 6.2. Additional notes on each startup step is provided for the novice user. Please refer to Figure 6.2 as you reference the notes below.

- If you are a novice user, you need to find a source for Ubuntu Linux. It is highly recommended to obtain a supplementary text on Linux to provide additional background information on each step. A representative sample of available texts is provided in Section 6.18. Often a textbook companion CD/DVD will contain the operating system. The operating system may also be downloaded from `www.ubuntu.com` or purchased from a number of sources online.
- To partition the host computer, configure Linux to operate in the Live mode. In this mode Linux is run from a CD, DVD, or USB drive and the hard drive is not modified.
- The hard drive is then partitioned using the GParted application available on the Ubuntu desktop.

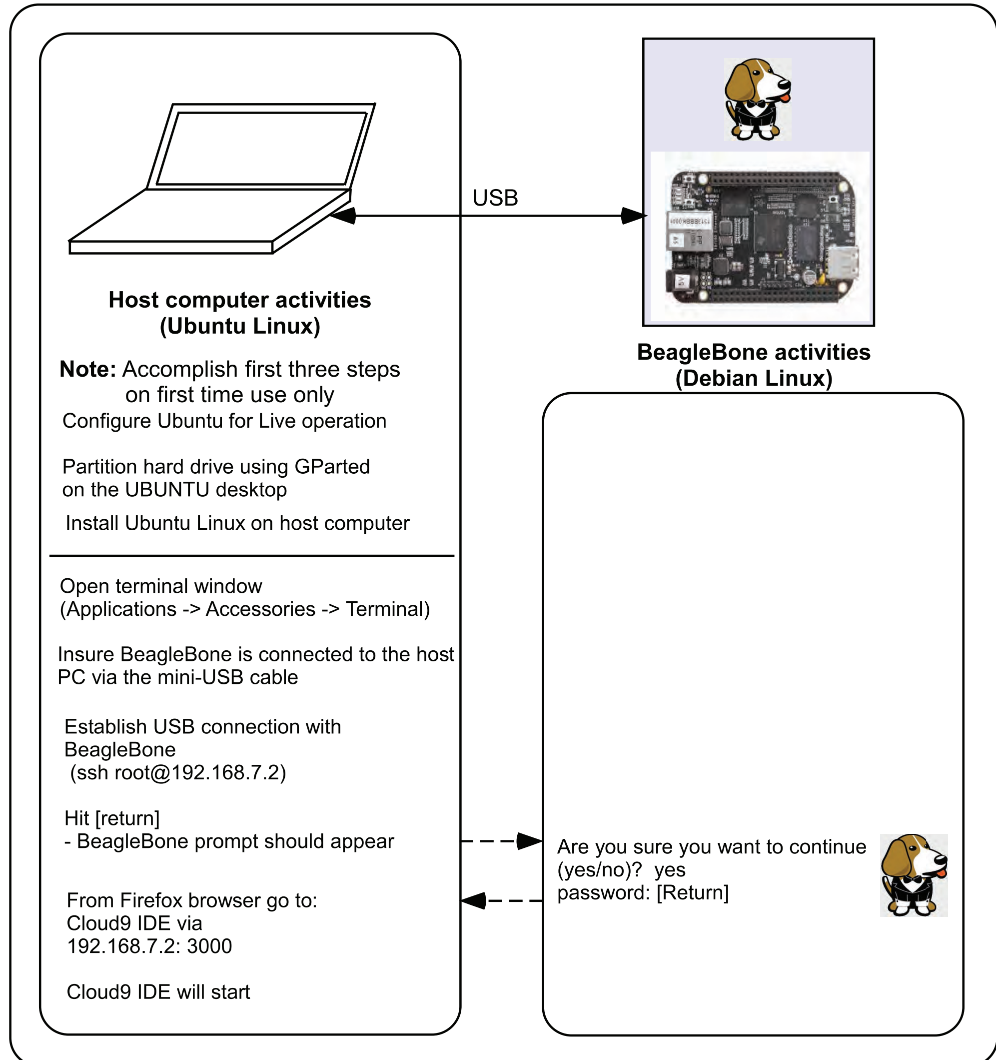


Figure 6.2: BeagleBone system startup in Linux.

- Once suitable space has been partitioned for Ubuntu Linux, Linux may be installed on the host computer hard drive.
- Every time the host computer is next started, you will be prompted for which operating system to boot. When working with BeagleBone, you may choose between Ubuntu Linux or MS Windows.
- Once Ubuntu Linux is successfully loaded on the hard drive and started, open a terminal window using Applications – > Accessories – > Terminal.
- Insure the host computer is connected to BeagleBone via the USB cable.
- From the host computer Ubuntu terminal prompt, establish a connection with BeagleBone (ssh root@192.168.7.2).
- Hit [Return] on the host computer.
- The BeagleBone Black prompt should appear.
- If prompted to continue, respond with "yes."
- When prompted for a password, hit [Return].
- Open the Mozilla Firefox web browser on the host computer and start Cloud9 IDE by going to 192.168.7.2:3000

A screenshot of the Cloud9 IDE is provided in Figure 6.3.

Example 4: Live Linux from USB. Instead of installing Linux to the host computer, Linux may also be configured to boot from a USB drive during host computer start-up. In this example we again use Ubuntu for exposure to another Linux release other than Debian. The live USB is configured using the following steps [`www.ubunto.com`]:

- Go to `www.ubunto.com` to download the ISO file for the desired operating system.
- Insert a USB drive to the host computer. The USB drive needs to be 2 Gbyte capacity or greater.
- Download Pen Drive's Linux installer software and follow the three configurations steps: (1) select Ubuntu, (2) browse to the location of the ISO file, and (3) select "Create."
- With the USB drive installed, Ubuntu will start upon host computer boot. Connection with BeagleBone Black is accomplished using the steps described above.

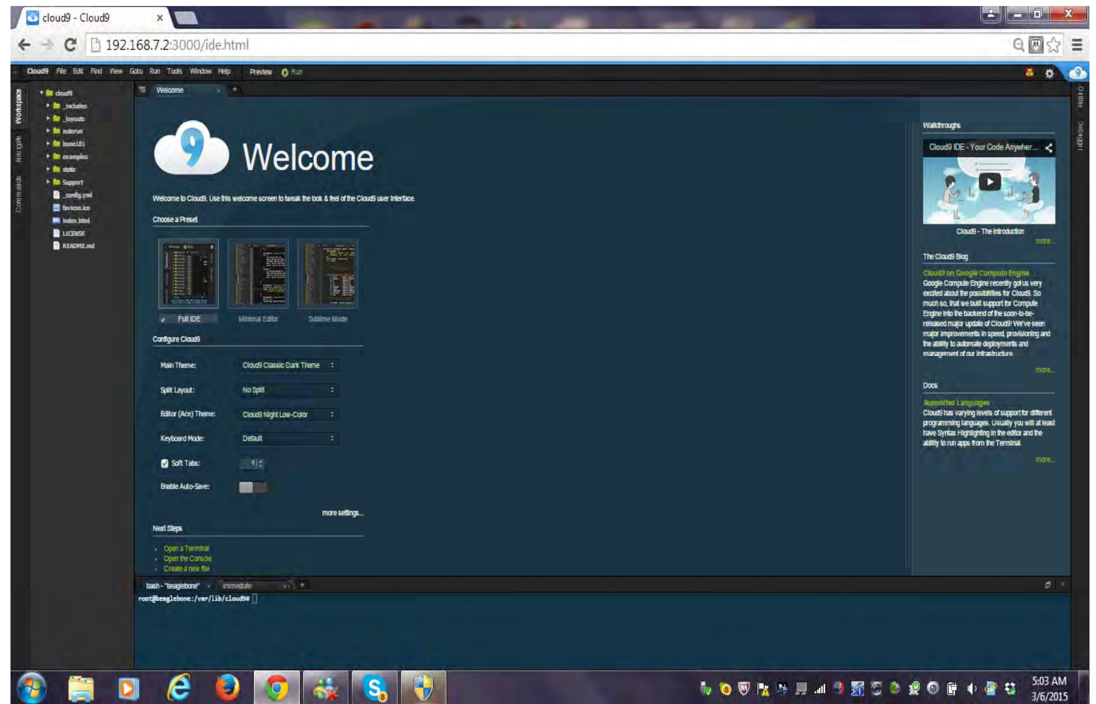


Figure 6.3: BeagleBone Cloud9 IDE screenshot.

Example 5: Linux within a virtual machine. The Linux operating system may also be executed from within a virtual machine on the host computer. There are several programs available to create a virtual computer system. This allows a different operating system (e.g., Linux) to be run within the safety of a virtual machine. For example, Virtual Box may be downloaded and installed from `www.virtualbox.org`. Once installed a virtual machine may be created and launched using the ISO file for the desired operating system.

Example 6: BeagleBone Black computer. In Chapter 7 we provide instructions on developing several versions of a stand-alone BeagleBone Black computer. Once configured the Linux prompt may be accessed by choosing "Accessories −> LXTerminal."

6.3 UPDATING YOUR EMMC

BeagleBone Bonescript and the Cloud9 IDE is a rapidly evolving, browser-based development environment. It is essential you regularly update the BeagleBone eMMC with the latest revision of Bonescript. In this section we describe how to update to latest software releases in Windows.

6.3.1 UPDATING YOUR EMMC IN MS WINDOWS

A step-by-step procedure to update to the latest software release is provided at www.beagleboard.org. In general, the latest compressed software image is downloaded, decompressed, and then written to a micro SD card. The BeagleBone Black is then booted from the micro SD card. Provided below is a brief summary of the steps [Molloy, 2015; Richardson, 2014]. Prior to updating the eMMC, user-developed programs should be backed up to a safe location. Procedures for doing this are provided later in the section.

- Download and install the required software tools to accomplish the software update.
  - Download and install "7-zip." This software is used to decompress the software image file.
  - Download and install "Image Writer for Windows." This software is used to write the decompressed software image to the micro SD card.
- Download the latest software image from http://beagleboard.org/latest-images. The file will have an ".img.xz" extension.
- Decompress the ".img.xz" file.
- Write the software image to the micro SD card using "Image Writer for Windows." Note: An adaptor may be required to interface the micro SD card to the host computer.
- When complete, eject the micro SD card from the host computer.
- Boot BeagleBone Black from the micro SD card.
  - Power down the BeagleBone Black computer.
  - Insert micro SD card into the BeagleBone Black micro SD card slot.
  - Hold down "USER/BOOT" button.
  - Apply power to the BeagleBone Black computer.
  - Let go of the "USER/BOOT" button when you see some of the blue LEDs on the BeagleBone illuminate.
- Happy Beagling!
- When your session is complete, power down the BeagleBone Black by holding the power button for 8–10 s.

Updating the eMMC Flash

To boot from the eMMC rather than the micro SD card, follow the steps above to obtain an updated image. However, instead download the "eMMC Flasher" version. Boot from the micro SD card as described above. It will take approximately an hour to update the eMMC. The four blue LEDs will illuminate to indicate completion of the flashing sequence. In the next section we provide a brief introduction to Linux.

File Backup and Transfer with WinSCP

Windows Secure CoPy (WinSCP) is a software package that provides for file transfer between a host Windows-based computer and BeagleBone Black. As mentioned previously, it is a good idea to back up user-written files before updating the eMMC Flash.

WinSCP may be downloaded from `inspire.logicsupply.com`. The site also provides file transfer instructions summarized here.

- Download WinSCP to a Windows-based host computer.
- Connect BeagleBone Black to the host computer with a mini-USB cable and allow BeagleBone to startup.
- Launch WinSCP on the host computer with the WinSCP icon.
- Configure WinSCP for file transfer with the following settings.
  - Select file transfer protocol SCP.
  - Enter BeagleBone Black IP address 192.168.7.2.
  - Provide user–name "root."
  - Save the settings.
- Depress the Login button. The login sequence will commence.
- Files may now be dragged in either direction between the host computer and BeagleBone Black.

6.4 A BRIEF INTRODUCTION TO LINUX

As mentioned earlier in the chapter, Linux is a powerful operating system originally developed by Linus Torvald of Finland in the early 1990's. Since its initial release, a variety of Linux distributions have been developed. BeagleBone Black Rev C shipped with the Debian release starting in early 2014. In this section we provide a very brief introduction to Linux. Additional Linux information will be provided throughout the text when needed.

A Linux distribution consists of the Linux operating system kernel, a collection of useful software applications and also installation software. Linux also includes a command interpreter

called the shell. The Debian release employs the BASH (Bourne-Again Shell). The basic format for a Linux command is:

```
command option1 option2 . . . optionX
```

A Linux command consists of the command name followed by different options. A space is placed between the command and each option. Provided below is a brief list of common Linux commands [Dulaney, 2010].

- **Help commands**
  - **info:** display information on a specific command
  - **man:** display online help on a specific command
- **File and Directory management commands**
  - **cd:** change directory
  - **cp:** copy files
  - **ls:** display contents of current directory
  - **mkdir:** create a new directory
  - **rm:** remove or delete a file
  - **rmdir:** remove or delete a directory
  - **pwd:** display the current directory
- **File processing commands**
  - **cat:** display file on standard output
  - **echo:** write arguments to standard output or specified file
  - **grep:** search for an expression in a file
  - **less:** display text file one page at a time
  - **more:** display text file one page at a time
  - **tail:** display the last few lines of a file
  - **zcat:** display a compressed file
- **Compression commands**
  - **compress:** compress specified file
  - **uncompress:** uncompress specified file

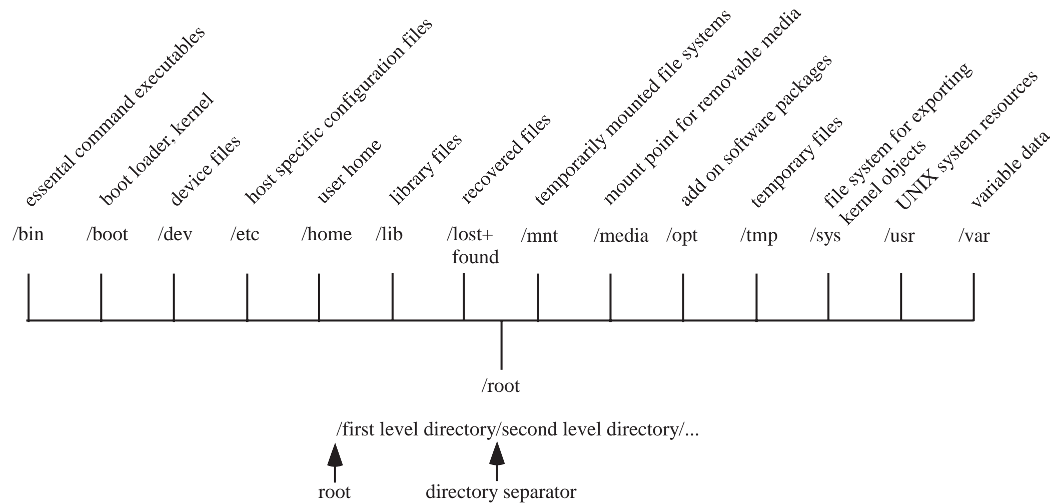


Figure 6.4: Linux file system [wiki.debian.org].

Shown in Figure 6.4 is a typical Linux file structure. The file structure system in Linux is standardized via the Filesystem Hierarchy Standard (FHS) [wiki.debian.org; Dulaney, 2010]. The file system is constructed like a tree. The foundation is the root directory. The other directories may be accessed via the root.

Example: Use the commands provided above to navigate about the Linux file structure aboard BeagleBone Black. Sketch a map of the file structure. As an alternative, using a BeagleBone Black stand-alone computer (Chapter 7), use the Debian File Manager to map the file structure. **Hint:** From the root prompt, use the "ls/" command the display the next level of file structure.

6.5 PROGRAMMING IN C USING THE LINUX TOOLCHAIN

The Linux distribution resident on BeagleBone has built-in C and C++ programming tools. The tools of interest include the vi text editor and the gcc and g++ compilers. The vi text editor is resident within the Linux distribution onboard BeagleBone. In this section we learn how to edit, compile, and execute a small sample program. The Linux distribution also includes a "what-you--see-is-what-you-get" (WYSIWYG) text editor called "nano." An excellent tutorial to nano is available at www.howtogeek.com.

The vi text editor is accessed from the Linux command line by typing: >vi <filename>

For our programming example, use the following to launch the editor and establish a new file "test" for editing:

```
# vi test.c
```

To enter the vi text editor insert mode, type an "a." The program may now be typed and edited using commands from table in Figure 6.5. When the program is complete, the program may be saved using "[Esc] : wq."

Common vi commands are provided in Figure 6.5.

Table 6.1. Common vi commands.	
a	insert text after cursor
A	insert text at end of current line
dd	delete current line
D	delete up to the end of current line
h or ←	move char to left
j or ↓	move down one line
k or ↑	move up one line
l or →	move char to right
yy	yank (copy) current line to buffer
nyy	yank n lines to buffer
P	put yanked line above current line
p	put yanked line below current line
x	delete character under cursor
/	finds a word going forward
:q	quit editor
:q!	quit without save
:r file	read file and insert after current line
:w file	write buffer to file
:wq	save changes and exit
Esc	end input mode, enter command mode

Figure 6.5: Common vi commands [Dulaney, 2010].

The next step is to install the g++ and gcc compiler on BeagleBone. This may be accomplished using the Linux command:

```
# gcc -v
```

This command will check to see if the latest version of this compiler is installed. If the most current version is not present, steps are provided to install it.

Example: Use the Linux toolchain to execute a "Hello World" or in the BeagleBone case a "Aroo from BeagleBone!" The steps to accomplish this are illustrated in Figure 6.6.

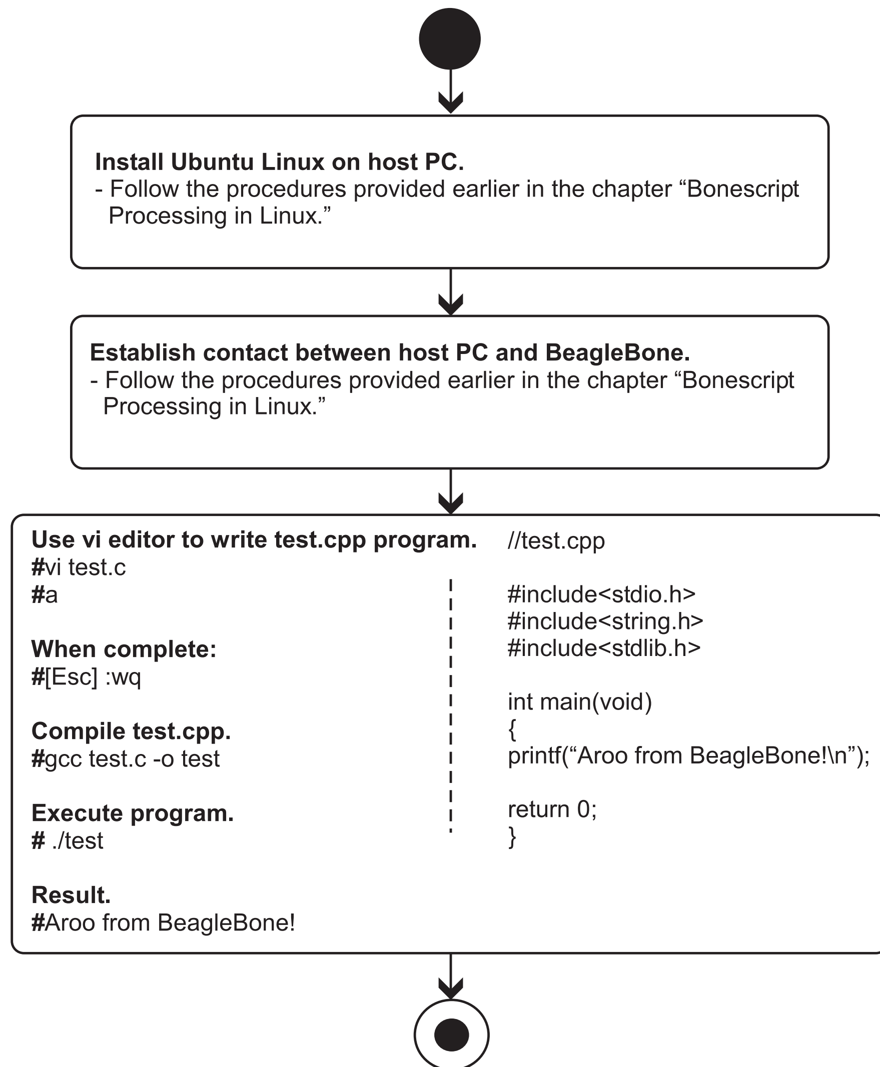


Figure 6.6: BeagleBone “Aroo!”

6.6 BEAGLEBONE FEATURES AND SUBSYSTEMS

In this section we provide a detailed overview of the features accessible via BeagleBone header pins P8 and P9. The reader is encouraged to thoroughly review the *BeagleBone Black Rev C.1 Systems Reference Manual* by Gerald Coley, 2014. The features of the “Bone” is summarized in Figure 6.7.

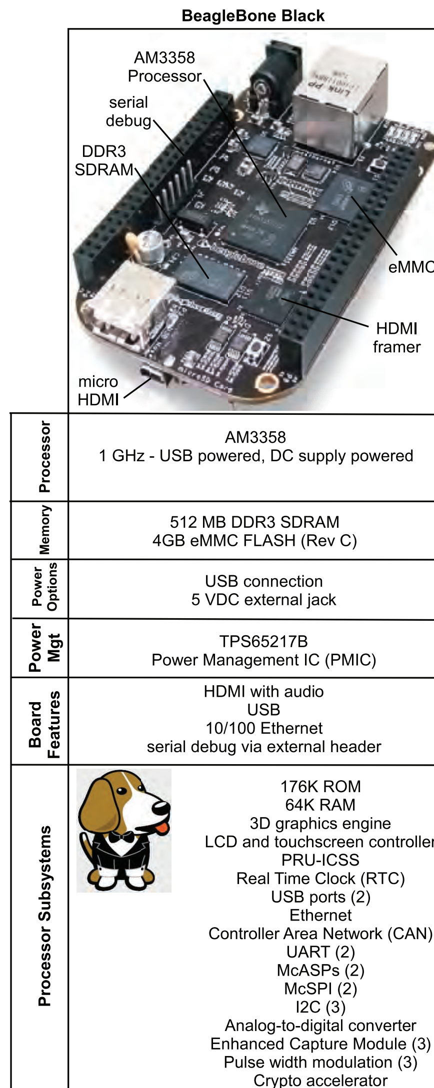


Figure 6.7: BeagleBone Black features [Coley, 2014]. (Figures adapted and used with permission from www.beagleboard.org.)

BeagleBone Black hosts the ARM AM3359 processor that operates at a frequency of 1 GHz when powered from an external 5 VDC source or USB. (Black may move to the AM3358 in the future as the Ethernet feature isn't required for the product, but it is initially shipping with AM3359). The BeagleBone Black Rev C is equipped with a 4 GB eMMC (embedded MultiMedia Card). This provides for non-volatile mass storage in an integrated circuit package. The eMMC acts as the "hard drive" for BeagleBone Black and hosts the Linux operating system, Cloud9 Integrated Development Environment (IDE) and Bonescript. BeagleBone Black is also equipped with a HDMI (High-Definition Multimedia Interface) framer and micro connector. HDMI is a compact digital audio/interface which is commonly used in commercial entertainment products such as DVD players, video game consoles and mobile phones. BeagleBone Black costs approximately US $55 [Coley, 2014].

BeagleBone Black is equipped with a double data rate synchronous dynamic random access memory (DDR3) with a capacity of 512 Mbytes configured as a 256 Mbyte by 16-bit memory. BeagleBone is also equipped with a 3 Kbyte memory EEPROM (electrically erasable programmable read only memory) which is a non-volatile memory used to store board configuration information [Coley, 2014].

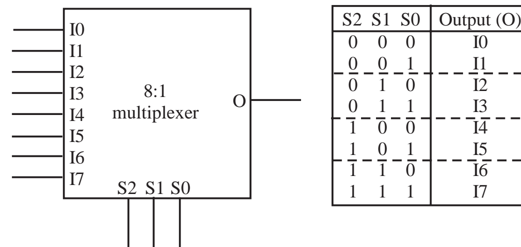
6.6.1 EXPOSED FUNCTION ACCESS

The ARM AM3358/3359 processor hosted aboard BeagleBone has a wide variety of systems and features. Many of these features are accessible to the user via BeagleBone's expansion interface via the P8 and P9 header connectors. Each pin on the headers supports multiple functions. The multiple functions of a given pin are controlled by a multiplexer, as shown in Figure 6.8a. The multiplexer acts as a multi-position switch. In the figure, an 8–1 multiplexer is shown. This multiplexer has eight inputs (I[0] through I[7]) and one output (O). Only one input may be connected to the output at any given time. The input is selected via the selected switches (S[2:0]). This has important implications in BeagleBone based systems. When designing a system, you will not be able to simultaneously use systems that share the same expansion header pins without careful precaution.

Tables 12 and 13 for the P9 header and Tables 9 and 10 for the P8 header of the *BeagleBone Black Rev C.1 Systems Reference Manual* provide all the different modes (Mode 0 through Mode 7). The modes may be viewed as the multiplexer inputs and the processor pin as the multiplexer output [Coley, 2014]. These tables also provide the:

- **PIN:** pin number on the expansion header (P9 or P8);
- **PROC:** pin number of the processor; and
- **Signal name:** the signal name of the pin.

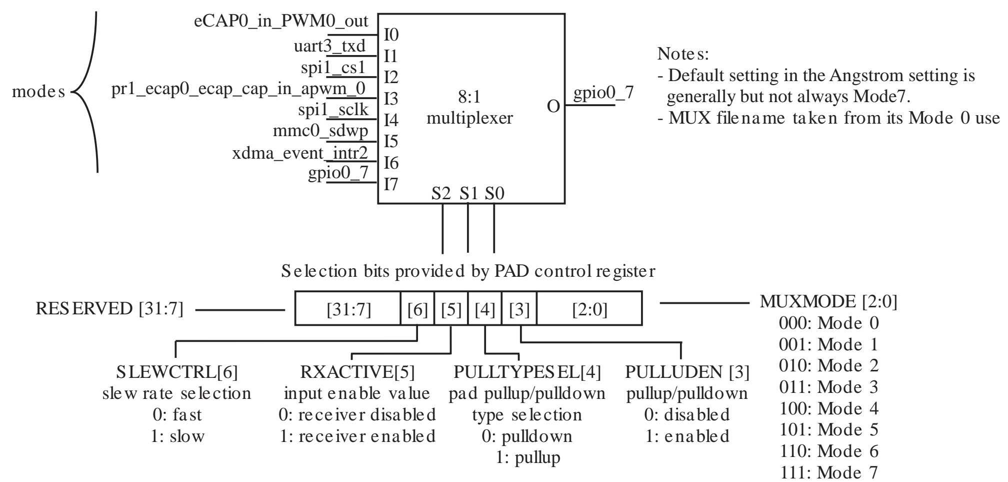
As an example, Figure 6.8b provides a detailed illustration of the GPIO0_7 pin. An extract of Tables 12 and 13 provide the important information for the pin:



a) 8:1 multiplexer

from Table 12, 13: Connector P9 Signal Pin Mux Options [Coley]

Pin	Proc	Signal Name	MODE0	MODE1	MODE2	MODE3	MODE4	MODE5	MODE6	MODE7
42	C18	GPIO0_7	eCAP0_in_ PWM0_out	uart3_txd	spi1_cs1	pr1_ecap0_ ecap_...	spi1_sclk	mmc0_sdwp	xdma_event _intr2	gpio0_7



b) Pin: 42, Signal name: GPIO0_7

Figure 6.8: Processor pin [Coley, 2014].

- **PIN:** 42
- **PROC:** C18
- **Signal name:** GPIO0_7
- **MODE[0] to MODE[7]:** the modes for this pin are shown as inputs to the multiplexer in Figure 6.8b.

Each processor pin has a corresponding PAD Control Register that contains the select pins for the multiplexer to select the pin mode and also other pin attributes. As shown in Figure 6.8b the PAD control register is a 32-bit register that contains settings for:

- **Slew rate selection (SLEWCTRL):** 0: fast and 1: slow;
- **Receiver input enabled (RXACTIVE):** 0: disabled and 1: enabled;
- **Pad pullup type selection (PULLTYPESEL):** 0: pulldown and 1: pullup;
- **Pullup/pulldown selection (PULLEDEN):** 0: disabled and 1: enabled; and
- **Multiplexer mode selection (MUXMODE):** 000 (Mode 0) through 111 (Mode 7).

The PAD control settings are set by configuring appropriate bits as shown in Figure 6.8. Let's take a closer look at pullup and pulldown resistors. As mentioned in Chapter 3 (see Figure 4.4), a pullup or pulldown resistor is required with various switch configurations. As shown in Figure 6.9, switches may be configured in different ways to provide various logic transitions. Pullup or pulldown resistors may be activated on the processor pin to aid in switch interfacing.

6.6.2 EXPANSION INTERFACE

The expansion interface is accessible via header pins P8 and P9. Many of BeagleBone Black's subsystems and features are accessible via the headers. The expansion interface is illustrated in Figures 6.10 and 6.11 [www.beagleboard.org].

6.7 BEAGLEBONE BLACK DEVICE TREE AND OVERLAYS

In this section we provide a brief introduction to the device tree overlay concept employed with BeagleBone Black. We discuss the motivation for the device tree overlay approach, some related general concepts, the basic format of a device tree overlay, review the overlay for BeagleBone Black, and conclude with an example. Information for this section was provided by a series of articles listed in Section 6.18. The reader is encouraged to review these articles in detail: [Devicetree, 2015; Likely, 2015; Hipster, 2013].

6.7.1 OVERVIEW

The device tree is a software data structure used to describe the hardware configuration of a specific processor to Linux. It allows the Linux kernel to remain the same even when used on a wide variety of processor configurations. Hardware specific details are passed to the Linux operating system via a device tree written specifically for the hardware device. The device tree is read via Linux during the boot operation. It may also be modified during run time by an application executing on the processor [Devicetree, 2015; Likely, 2015; Hipster, 2013].

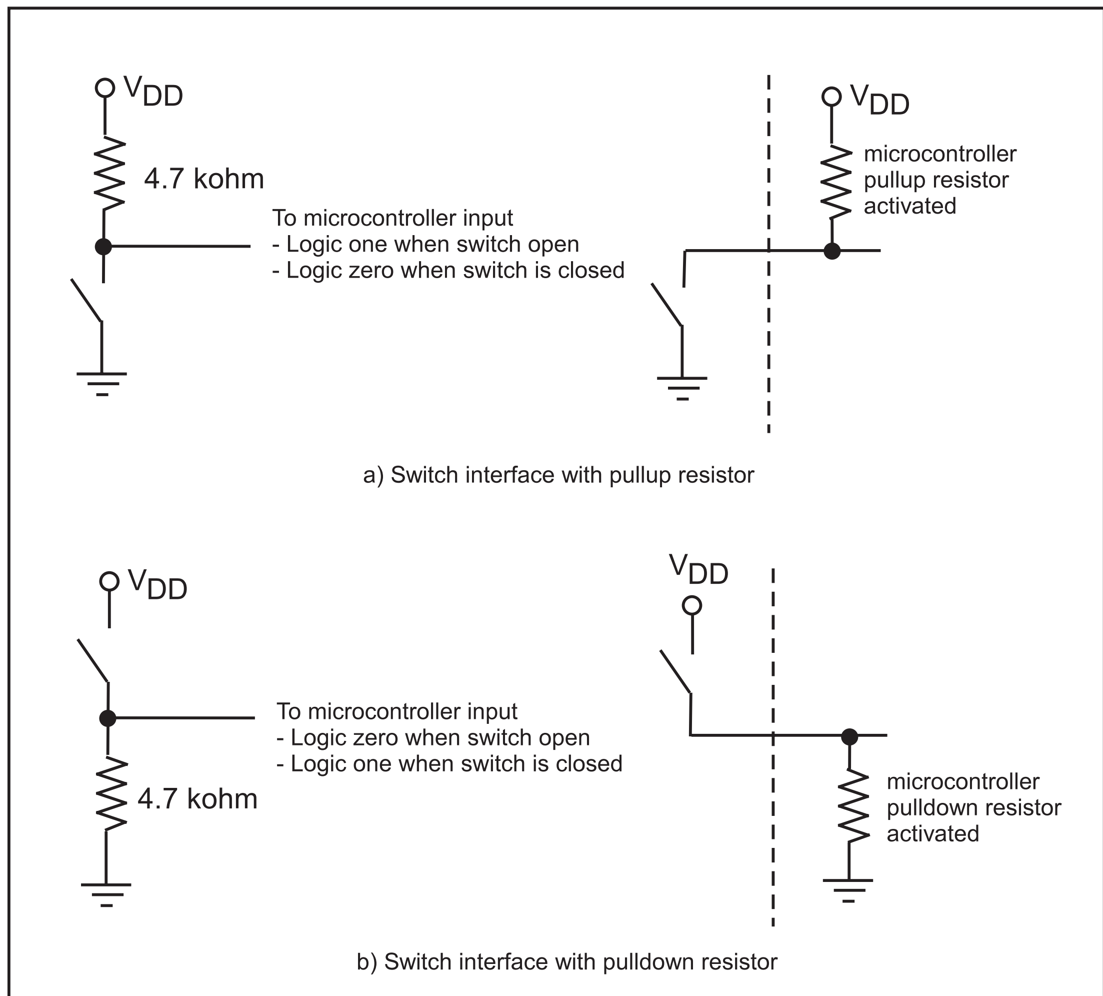


Figure 6.9: Switch interface with pullup and pulldown resistors.

6.7.2 BINARY TREE

The device tree is a software data structure similar to a binary tree. A binary tree consists of a collection of nodes as shown in Figure 6.12 [Korsch and Garrett, 1988]. Each node may hold a variety of data and also two links to its successor or child nodes. For example, Node 3 is the child or successor of Node 1. While Nodes 6 and 7 are child nodes of Node 3. From Node 3's point of view, Node 1 is its predecessor or parent node.

PIN 1,2	PROC	NAME	MODE0	MODE1	MODE2	MODE3 GND	MODE4	MODE5	MODE6	MODE7
3	R9	GPIO1_6	gpmc_ad6	mmc1_dat6						gpio1[6]
4	T9	GPIO1_7	gpmc_ad7	mmc1_dat7						gpio1[7]
5	R8	GPIO1_2	gpmc_ad2	mmc1_dat2						gpio1[2]
6	T8	GPIO1_3	gpmc_ad3	mmc1_dat3						gpio1[3]
7	R7	TIMER4	gpmc_advn_ale		timer4					gpio2[2]
8	T7	TIMER7	gpmc_oen_ren		timer7					gpio2[3]
9	T6	TIMER5	gpmc_be0n_cle		timer5					gpio2[5]
10	U6	TIMER6	gpmc_wen		timer6					gpio2[4]
11	R12	GPIO1_13	gpmc_ad13	lcd_data18	mmc1_dat5	mmc2_dat1	eQEP2B_in		pr1_pru0_pru_r30_15	gpio1[13]
12	T12	GPIO1_12	gpmc_ad12	Lcd_data19	mmc1_dat4	Mmc2_dat0	Eqep2a_in		pr1_pru0_pru_r30_14	gpio1[12]
13	T10	EHRPWM2B	gpmc_ad9	lcd_data22	mmc1_dat1	mmc2_dat5	ehrpwm2B			gpio0[23]
14	T11	GPIO0_26	gpmc_ad10	lcd_data21	mmc1_dat2	mmc2_dat6	ehrpwm2_tripzone_in			gpio0[26]
15	U13	GPIO1_15	gpmc_ad15	lcd_data16	mmc1_dat7	mmc2_dat3	eQEP2_strobe		pr1_pru0_pru_r31_15	gpio1[15]
16	V13	GPIO1_14	gpmc_ad14	lcd_data17	mmc1_dat6	mmc2_dat2	eQEP2_index		pr1_pru0_pru_r31_14	gpio1[14]
17	U12	GPIO0_27	gpmc_ad11	lcd_data20	mmc1_dat3	mmc2_dat7	ehrpwm0_synco			gpio0[27]
18	V12	GPIO2_1	gpmc_clk_mux0	lcd_memory_clk	gpmc_wait1	mmc2_clk			mcasp0_fsr	gpio2[1]
19	U10	EHRPWM2A	gpmc_ad8	lcd_data23	mmc1_dat0	mmc2_dat4	ehrpwm2A			gpio0[22]
20	V9	GPIO1_31	gpmc_csn2	gpmc_be1n	mmc1_cmd			pr1_pru1_pru_r30_13	pr1_pru1_pru_r31_13	gpio1[31]
21	U9	GPIO1_30	gpmc_csn1	gpmc_clk	mmc1_clk			pr1_pru1_pru_r30_12	pr1_pru1_pru_r31_12	gpio1[30]
22	V8	GPIO1_5	gpmc_ad5	mmc1_dat5						gpio1[5]
23	U8	GPIO1_4	gpmc_ad4	mmc1_dat4						gpio1[4]
24	V7	GPIO1_1	gpmc_ad1	mmc1_dat1						gpio1[1]
25	U7	GPIO1_0	gpmc_ad0	mmc1_dat0						gpio1[0]
26	V6	GPIO1_29	gpmc_csn0							gpio1[29]
27	U5	GPIO2_22	lcd_vsync	gpmc_a8				pr1_pru1_pru_r30_8	pr1_pru1_pru_r31_8	gpio2[22]
28	V5	GPIO2_24	lcd_pclk	gpmc_a10				pr1_pru1_pru_r30_10	pr1_pru1_pru_r31_10	gpio2[24]
29	R5	GPIO2_23	lcd_hsync	gpmc_a9				pr1_pru1_pru_r30_9	pr1_pru1_pru_r31_9	gpio2[23]
30	R6	GPIO2_25	lcd_ac_bias_en	gpmc_a11						gpio2[25]
31	V4	UART5_CTSN	lcd_data14	gpmc_a18	eQEP1_index	mcasp0_axr1	uart5_rxd		uart5_ctsn	gpio0[10]
32	T5	UART5_RTSN	lcd_data15	gpmc_a19	eQEP1_strobe	mcasp0_ahclkx	mcasp0_axr3		uart5_rtsn	gpio0[11]
33	V3	UART4_RTSN	lcd_data13	gpmc_a17	eQEP1B_in	mcasp0_fsr	mcasp0_axr3		uart4_rtsn	gpio0[9]
34	U4	UART3_RTSN	lcd_data11	gpmc_a15	ehrpwm1B	mcasp0_ahclkr	mcasp0_axr2		uart3_rtsn	gpio2[17]
35	V2	UART4_CTSN	lcd_data12	gpmc_a16	eQEP1A_in	mcasp0_aclkr	mcasp0_axr2		uart4_ctsn	gpio0[8]
36	U3	UART3_CTSN	lcd_data10	gpmc_a14	ehrpwm1A	mcasp0_axr0			uart3_ctsn	gpio2[16]
37	U1	UART5_TXD	lcd_data8	gpmc_a12	ehrpwm1_tripzone_in	mcasp0_aclkx	uart5_txd		uart2_ctsn	gpio2[14]
38	U2	UART5_RXD	lcd_data9	gpmc_a13	ehrpwm0_synco	mcasp0_fsx	uart5_rxd		uart2_rtsn	gpio2[15]
39	T3	GPIO2_12	lcd_data6	gpmc_a6		eQEP2_index		pr1_pru1_pru_r30_6	pr1_pru1_pru_r31_6	gpio2[12]
40	T4	GPIO2_13	lcd_data7	gpmc_a7		eQEP2_strobe	pr1_edio_data_out7	pr1_pru1_pru_r30_7	pr1_pru1_pru_r31_7	gpio2[13]
41	T1	GPIO2_10	lcd_data4	gpmc_a4		eQEP2A_in		pr1_pru1_pru_r30_4	pr1_pru1_pru_r31_4	gpio2[10]
42	T2	GPIO2_11	lcd_data5	gpmc_a5		eQEP2B_in		pr1_pru1_pru_r30_5	pr1_pru1_pru_r31_5	gpio2[11]
43	R3	GPIO2_8	lcd_data2	gpmc_a2		ehrpwm2_tripzone_in		pr1_pru1_pru_r30_2	pr1_pru1_pru_r31_2	gpio2[8]
44	R4	GPIO2_9	lcd_data3	gpmc_a3		ehrpwm0_synco		pr1_pru1_pru_r30_3	pr1_pru1_pru_r31_3	gpio2[9]
45	R1	GPIO2_6	lcd_data0	gpmc_a0		ehrpwm2A		pr1_pru1_pru_r30_0	pr1_pru1_pru_r31_0	gpio2[6]
46	R2	GPIO2_7	lcd_data1	gpmc_a1		ehrpwm2B		pr1_pru1_pru_r30_1	pr1_pru1_pru_r31_1	gpio2[7]

Figure 6.10: BeagleBone Black expansion interface header P8 [www.beagleboard.org].

PIN	PROC	NAME	MODE0	MODE1	MODE2	MODE3	MODE4	MODE5	MODE6	MODE7
1,2						GND				
3,4						DC_3.3V				
5,6						VDD_5V				
7,8						SYS_5V				
9						PWR_BUT				
10	A10					SYS_RESETn				
11	T17	UART4_RXD	gpmc_wait0	mii2_crs	gpmc_csn4	rmii2_crs_dv	mmc1_sdcd		uart4_rxd_mux2	gpio0[30]
12	U18	GPIO1_28	gpmc_be1n	mii2_col	gpmc_csn6	mmc2_dat3	gpmc_dir		mcasp0_aclkr_mux3	gpio1[28]
13	U17	UART4_TXD	gpmc_wpn	mii2_rxerr	gpmc_csn5	rmii2_rxerr	mmc2_sdcd		uart4_txd_mux2	gpio0[31]
14	U14	EHRPWM1A	gpmc_a2	mii2_txd3	rgmii2_td3	mmc2_dat1	gpmc_a18		ehrpwm1A_mux1	gpio1[18]
15	R13	GPIO1_16	gpmc_a0	gmii2_txen	rmii2_tctl	mii2_txen	gpmc_a16		ehrpwm1_tripzone_input	gpio1[16]
16	T14	EHRPWM1B	gpmc_a3	mii2_txd2	rgmii2_td2	mmc2_dat2	gpmc_a19		ehrpwm1B_mux1	gpio1[19]
17	A16	I2C1_SCL	spi0_cs0	mmc2_sdwp	I2C1_SCL	ehrpwm0_synci	pr1_uart0_txd			gpio0[5]
18	B16	I2C1_SDA	spi0_d1	mmc1_sdwp	I2C1_SDA	ehrpwm0_tripzone	pr1_uart0_rxd			gpio0[4]
19	D17	I2C2_SCL	uart1_rtsn	timer5	dcan0_rx	I2C2_SCL	spi1_cs1	pr1_uart0_rts_n		gpio0[13]
20	D18	I2C2_SDA	uart1_ctsn	timer6	dcan0_tx	I2C2_SDA	spi1_cs0	pr1_uart0_cts_n		gpio0[12]
21	B17	UART2_TXD	spi0_d0	uart2_txd	I2C2_SCL	ehrpwm0B	pr1_uart0_rts_n		EMU3_mux1	gpio0[3]
22	A17	UART2_RXD	spi0_sclk	uart2_rxd	I2C2_SDA	ehrpwm0A	pr1_uart0_cts_n		EMU2_mux1	gpio0[2]
23	V14	GPIO1_17	gpmc_a1	gmii2_rxdv	rgmii2_rxdv	mmc2_dat0	gpmc_a17		ehrpwm0_synco	gpio1[17]
24	D15	UART1_TXD	uart1_txd	mmc2_sdwp	dcan1_rx	I2C1_SCL		pr1_uart0_txd	pr1_pru0_pru_r31_16	gpio0[15]
25	A14	GPIO3_21*	mcasp0_ahclkx	eQEP0_strobe	mcasp0_axr3	mcasp1_axr1	EMU4_mux2	pr1_pru0_pru_r30_7	pr1_pru0_pru_r31_7	gpio3[21]
26	D16	UART1_RXD	uart1_rxd	mmc1_sdwp	dcan1_tx	I2C1_SDA		pr1_uart0_rxd	pr1_pru1_pru_r31_16	gpio0[14]
27	C13	GPIO3_19	mcasp0_fsr	eQEP0B_in	mcasp0_axr3	mcasp1_fsx	EMU2_mux2	pr1_pru0_pru_r30_5	pr1_pru0_pru_r31_5	gpio3[19]
28	C12	SPI1_CS0	mcasp0_ahclkr	ehrpwm0_synci	mcasp0_axr2	spi1_cs0	eCAP2_in_PWM2_out	pr1_pru0_pru_r30_3	pr1_pru0_pru_r31_3	gpio3[17]
29	B13	SPI1_D0	mcasp0_fsx	ehrpwm0B		spi1_d0	mmc1_sdcd_mux1	pr1_pru0_pru_r30_1	pr1_pru0_pru_r31_1	gpio3[15]
30	D12	SPI1_D1	mcasp0_axr0	ehrpwm0_tripzone		spi1_d1	mmc2_sdcd_mux1	pr1_pru0_pru_r30_2	pr1_pru0_pru_r31_2	gpio3[16]
31	A13	SPI1_SCLK	mcasp0_aclkx	ehrpwm0A		spi1_sclk	mmc0_sdcd_mux1	pr1_pru0_pru_r30_0	pr1_pru0_pru_r31_0	gpio3[14]
32						VADC				
33	C8					AIN4				
34						AGND				
35	A8					AIN6				
36	B8					AIN5				
37	B7					AIN2				
38	A7					AIN3				
39	B6					AIN0				
40	C7					AIN1				
41#	D14	CLKOUT2	xdma_event_intr1		tclkin	clkout2	timer7_mux1	pr1_pru0_pru_r31_16	EMU3_mux0	gpio0[20]
	D13	GPIO3_20	mcasp0_axr1	eQEP0_index		Mcasp1_axr0	emu3	pr1_pru0_pru_r30_6	pr1_pru0_pru_r31_6	gpio3[20]
42@	C18	GPIO0_7	eCAP0_in_PWM0_out	uart3_txd	spi1_cs1	pr1_ecap0_ecap_capin_apwm_o	spi1_sclk	mmc0_sdwp	xdma_event_intr2	gpio0[7]
	B12	GPIO3_18	Mcasp0_aclkr	eQEP0A_in	Mcaspo_axr2	Mcasp1_aclkx		pr1_pru0_pru_r30_4	pr1_pru0_pru_r31_4	gpio3[18]
43-46						GND				

Figure 6.11: BeagleBone Black expansion interface header P9 [www.beagleboard.org].

Access is gained to the binary tree via its root node. The tree may then be traversed to obtain the information at each node as required. It should be emphasized the binary tree and the related device tree are both abstract data types. That is, they are a custom collection of more fundamental data types with associated related operations to more easily accomplish a specific task. As previously mentioned, the device tree's task is to provide hardware details to the Linux operating system at boot or during run time.

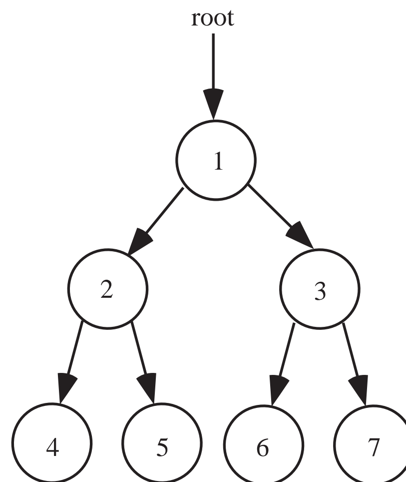


Figure 6.12: Binary tree. (Adapted from Korsch and Garrett, 1988.)

6.7.3 DEVICE TREE FORMAT

In this section we provide a brief introduction to the device tree format. An excellent, detailed discussion of this topic is provided at `www.devicetree.org`. The device tree uses a binary tree format to contain device hardware properties at each node along with connections to its child nodes. The device tree root is designated with a forward slash (/) symbol. It then has a compatibility statement to indicate the hardware processor in use. Following the compatibility statement is a listing of nodes. Each node contains information about specific hardware devices aboard the processor. The tree hierarchy also represents how the devices are interconnected within the processor.

Each of the individual nodes contains a compatibility statement and device addressing parameters. The device addressing parameters include the base address for its associated registers and the number of registers used. Provided in the code snapshot below is the basic format of the device tree [devicetree, 2015].

```
1 //*****************************************************************
2 /{
```

```
3   compatible = ''<manufacturer>,<model>";
4
5
6   <node name>[@<device address>]{
7       compatible = ''<manufacturer>,<model>";
8       reg = <start address   length>;
9       };
10
11 //Additional device node descriptions
12
13 :
14 :
15 :
16
17  };
18
19 //*****************************************************************
```

The device tree is loaded during system boot. However, portions of the device tree may be modified by using a fragment modifier during the boot process or during program execution.

6.7.4 DEVICE TREE RELATED FILES

When a device tree overlay is created or modified it must be recompiled. Device tree source files have a <filename>.dts prefix. These files are compiled by the device tree compiler (dtc). It is initiated from the command line using "dtc." When a $< filename >$.dts file is compiled it becomes a $< filename >$.dtbo object file.

6.7.5 BEAGLEBONE BLACK DEVICE TREE

Pantellis has provided a device tree overlay for BeagleBone users and also introduced a BeagleBone cape manager (capemgr) to assist with updating BeagleBone device tree information during boot time or during program run time [Pantellis, 2015]. The following example illustrates how the device tree may be modified using a fragment statement.

Example 1: We start the example by defining the two key sysfs entries we'll use to check status. The sysfs is a virtual file system used by the Linux operating system to pass information to user space. User space is virtual memory space outside the Linux operating system kernel to execute programs.

An interface to the capemgr is provided by "slots." It notifies capemgr to load additional device tree overlay fragments and also reports on what has already been loaded. It uses the EEPROMs on the cape plug-in boards to identify the board. **Note:** In the example, the bone_capemgr may be designated with either an ".8" or ".9."

Using one of the techniques described earlier in the chapter, access the BeagleBone root directory. For example, the Cloud 9 IDE may be accessed via a website browser (e.g., Google

Chrome) by navigating to 192.168.7.2:3000. Access to the Linux is provided in the lower pane of the Cloud 9 IDE. When starting the Cloud 9 IDE, access is usually provided to:

```
root@beaglebone:/var/lib/cloud9#
```

You can switch to the root directory using:

```
root@beaglebone:/var/lib/cloud9# cd
```

yielding:

```
root@beaglebone:~#
```

Navigate to the */sys/devices* directory and examine the directory contents with the "ls" command to determine the cape manager number (8 or 9).

```
root@beaglebone:~# cd /sys/devices
root@beaglebone:/sys/devices#
root@beaglebone:/sys/devices# ls
```

Load the "slots" and "pins" environment variables to use the cape manager [Kridner, Molloy]:

```
# export SLOTS=/sys/devices/bone_capemgr.9/slots
# export PINS=/sys/kernel/debug/pinctrl/44e10800.pinmux/pins
```

Note: The root prompt is designated with the shorthand "#" notation.
You can verify loading of "slots" with:

```
# cat $SLOTS
```

Also, you can verify loading of the "pins" by examining the */sys/kernel/debug/pinctrl/*44e10800.*pinmux/pins* directory.

Note: It must be emphasized that "slots" and "pins" must be loaded upon startup to interact with the BeagleBone Black device tree.

Example 2: Provided below is a device tree overlay to update the multiplexer setting of header P9 pin 42 to mode 7.

```
1 //**********************************************************************
2 //devicetree overlay (pinmux-test-7.dts):
3 //
4 //Copyright (C) 2012 Texas Instruments Incorporated - http://www.ti.com
    /
5 //
6 //This program is free software; you can redistribute it and/or modify
```

```
7 //it under the terms of the GNU General Public License version 2 as
8 //published by the Free Software Foundation.
9 //***********************************************************************
10
11 /dts-v1/;
12 /plugin/;
13
14 / {
15     compatible = "ti,beaglebone", "ti,beaglebone-black";
16
17     /* identification */
18     part-number = "pinctrl-test-7";
19
20     fragment@0 {
21         target = <&am33xx_pinmux>;
22         __overlay__ {
23             pinctrl_test: pinctrl_test_7_pins {
24                 pinctrl-single,pins = <
25                     0x164 0x07  /* P9_42 muxRegOffset, OUTPUT | MODE7
                        */
26                 >;
27             };
28         };
29     };
30
31     fragment@1 {
32         target = <&ocp>;
33         __overlay__ {
34             test_helper: helper {
35                 compatible = "bone-pinmux-helper";
36                 pinctrl-names = "default";
37                 pinctrl-0 = <&pinctrl_test>;
38                 status = "okay";
39             };
40         };
41     };
42 };
43 //*********************************************************************
```

The device tree fragments are now compiled and installed in /lib/firmware. In this example it is performed natively on the BeagleBone.

```
1 dtc -O dtb -o pinctrl-test-7-00A0.dtbo -b 0 -@ pinctrl-test-7.dts
2 dtc -O dtb -o pinctrl-test-0-00A0.dtbo -b 0 -@ pinctrl-test-0.dts
3 cp pinctrl-test-7-00A0.dtbo /lib/firmware/
4 cp pinctrl-test-0-00A0.dtbo /lib/firmware/
```

Before continuing with the example, let's check out the starting point state.

```
1 # cat $SLOTS
```

```
2   0: 54:PF---
3   1: 55:PF---
4   2: 56:PF---
5   3: 57:PF---
6   4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
7   5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
```

As can be seen from the report, slots 0–3 get assigned by EEPROM IDs on the capes. There are four possible addresses for the EEPROMs (typically determined by switches on the boards) enabling up to four boards to be stacked, depending on what functions they use. Additional slots are "virtual," added incrementally and are triggered in other ways. In the above report you can see that no capes are currently installed. There are two "virtual" capes installed, one for the eMMC and one for the HDMI. It makes sense to manage these as capes because both interfaces consume pins on the cape bus. These two "virtual" capes are triggered on BeagleBone Black that includes the eMMC and HDMI on the board. Disabling these capes would enable other capes to make use of their pins.

In the next step, let's tell the capemgr to load our device tree overlay fragment that configures the target pin's pinmux. Carefully take a look at the messages that are produced by the kernel and review the capemgr status and the status of the pinmux.

```
1  # echo pinctrl-test-7 > $SLOTS
2  # dmesg | tail
3  [   65.323606] bone-capemgr bone_capemgr.8:
4      part_number 'pinctrl-test-7', version 'N/A'
5  [   65.323744] bone-capemgr bone_capemgr.8:
6      slot #6: generic override
7  [   65.323794] bone-capemgr bone_capemgr.8:
8      bone: Using override eeprom data at slot 6
9  [   65.323845] bone-capemgr bone_capemgr.8:
10     slot #6: 'Override Board Name,00A0,Override Manuf,pinctrl-test-7'
11 [   65.324201] bone-capemgr bone_capemgr.8:
12     slot #6: Requesting part number/version based
13     'pinctrl-test-7-00A0.dtbo
14 [   65.325712] bone-capemgr bone_capemgr.8:
15     slot #6: Requesting firmware 'pinctrl-test-7-00A0.dtbo'
16     for board-name 'Override Board Name', version '00A0'
17 [   65.326239] bone-capemgr bone_capemgr.8:
18     slot #6: dtbo 'pinctrl-test-7-00A0.dtbo' loaded;
19     converting to live tree
20 [   65.327973] bone-capemgr bone_capemgr.8: slot #6: #2 overlays
21 [   65.338533] bone-capemgr bone_capemgr.8: slot #6: Applied #2
    overlays.
22 # cat $SLOTS
23  0: 54:PF---
24  1: 55:PF---
25  2: 56:PF---
26  3: 57:PF---
```

```
27  4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
28  5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
29  6: ff:P-O-L Override Board Name,00A0,Override Manuf,pinctrl-test-7
```

The $PINS file is a debug entry for the pinctrl kernel module. It provides the status of the pinmux. To examine a specific pin state, grep for the lower bits of the address where the pinmux control register is located. The Linux grep command searches the given file for lines containing a match. As you can see, the mux mode is now 7.

```
1 # cat $PINS | grep 964
2 pin 89 (44e10964) 00000007 pinctrl-single
```

Now we"ll have the capemgr unload the overlay so that a different one can be loaded.

```
1 # A=`perl -pe 's/^.*(\d+):.*/$1/' $SLOTS | tail -1`
2 # echo "-$A"
3 -6
4 # echo "-$A" > $SLOTS
5 # dmesg | tail
6 [   73.517002] bone-capemgr bone_capemgr.8: Removed slot #6
7 [   73.517002] bone-capemgr bone_capemgr.8: Removed slot #6
8 And then tell capemgr to load an alternative overlay.
9 # echo pinctrl-test-0 > $SLOTS
10 # dmesg | tail
11 [   73.663144] bone-capemgr bone_capemgr.8:
12     part_number 'pinctrl-test-0', version 'N/A'
13 [   73.663207] bone-capemgr bone_capemgr.8:
14     slot #7: generic override
15 [   73.663226] bone-capemgr bone_capemgr.8:
16     bone: Using override eeprom data at slot 7
17 [   73.663244] bone-capemgr bone_capemgr.8:
18     slot #7: 'Override Board Name,00A0,Override Manuf,pinctrl-test-0'
19 [   73.663340] bone-capemgr bone_capemgr.8:
20     slot #7: Requesting part number/version based
21     'pinctrl-test-0-00A0.dtbo
22 [   73.663357] bone-capemgr bone_capemgr.8:
23     slot #7: Requesting firmware 'pinctrl-test-0-00A0.dtbo'
24     for board-name 'Override Board Name', version '00A0'
25 [   73.663602] bone-capemgr bone_capemgr.8: slot #7:
26     dtbo 'pinctrl-test-0-00A0.dtbo' loaded; converting to live tree
27 [   73.663857] bone-capemgr bone_capemgr.8:
28     slot #7: #2 overlays
29 [   73.674682] bone-capemgr bone_capemgr.8:
30     slot #7: Applied #2 overlays.
```

And what does the pinmux look like now?

```
1 # cat $PINS | grep 964
2 pin 89 (44e10964) 00000000 pinctrl-single
```

Fortunately, there are existing devicetree fragments you can load to avoid creating these fragments yourself. There are many fragments available for examination in directory $/lib/firmware$.

6.7.6 UNIVERSAL DEVICE TREE OVERLAY

Charles Steinkuehler developed a series of helpful BeagleBone Black universal device tree overlays for common application configurations along with a "config-pin" utility. The overlays are available in common anticipated BeagleBone Black configurations. If an overlay is chosen with a specific system disabled (e.g., HDMI, eMMC), pins normally allocated to support the system may be used for other applications. When a specific overlay is chosen and loaded, various selected systems are loaded and the general purpose input/output (GPIO) pins are provided in a reset configuration. The "config-pin" utility allows the user to determine the state of a specific BeagleBone Black pin or change its multiplexer configuration. It is important to note the universal overlays are included with the March 1, 2015 (and beyond) BeagleBone Black Debian release [Steinkuehler, 2015].

The following universal overlays (with a brief description) are available for loading [Steinkuehler, 2015]:

- cape-universal: overlay exports pins not used by HDMI and eMMC (with audio support);
- cape-universaln: overlay exports pins not used by HDMI and eMMC (no audio support);
- cape-univ-emmc: overlay exports pins used by the eMMC. This overlay may be used if the eMMC is disabled;
- cape-univ-hdmi: overlay exports pins used by the HDMI video system. This overlay may be used if the HDMI is disabled; and
- cape-univ-audio: overlay exports pins used by the HDMI audio system.

The contents of each universal overlay may be examined at https://github.com/cdsteinkuehler.

A specific overlay may be loaded using the command [Steinkuehler, 2015]:

```
#echo cape-universaln > /sys/devices/bone_capemgr.*/slots
```

Once the overlay is loaded, accompanying support files related to the overlay may be examined at: "/sys/devices/ocp.*" The following command sequence may be used:

```
#cd /sys/devices/ocp.*
/sys/devices/ocp.3# ls
```

The "config-pin" utility allows the user to determine the state of a specific BeagleBone Black pin or change its multiplexer configuration. To obtain a list of pin settings for the loaded universal cape, the following command may be used [Steinkuehler, 2015]:

```
#config-pin -l
```

Additional details are available using:

```
#config-pin -h
```

The configuration options for a specific pin may be examined using [Steinkuehler]:

```
#config-pin -l <pin>
```

To access pins via the C programming language, it is important to know how to access them in the Linux file system. Each pin exported during a universal cape load has a corresponding entry in "/sys/class/gpio/gpioXX." The "XX" designates the GPIO number for the pin. The pin numbering system is discussed in the next section.

Provided below is a partial list of device tree overlays available [Steinkuehler]:

- BB–ADC–00A0.dts
- BB–I2C1–00A0.dts
- BB–I2C1A1–00A0.dts
- BB–SPIDEV0–00A0.dts
- BB–SPIDEV1–00A0.dts
- BB–SPIDEV1A1–00A0.dts
- BB–UART1–00A0.dts
- BB–UART2–00A0.dts
- BB–UART4–00A0.dts
- BB–UART5–00A0.dts
- am33xx_pwm–00A0.dts
- bone_pwm_P8_13–00A0.dts
- bone_pwm_P8_19–00A0.dts
- bone_pwm_P8_34–00A0.dts
- bone_pwm_P8_36–00A0.dts
- bone_pwm_P8_45–00A0.dts
- bone_pwm_P8_46–00A0.dts

- bone_pwm_P9_14–00A0.dts
- bone_pwm_P9_16–00A0.dts
- bone_pwm_P9_21–00A0.dts
- bone_pwm_P9_22–00A0.dts
- bone_pwm_P9_28–00A0.dts
- bone_pwm_P9_29–00A0.dts
- bone_pwm_P9_31–00A0.dts
- bone_pwm_P9_42–00A0.dts
- cape–bone–pinmux–test–00A0.dts
- cape–boneblack–hdmi–00A0.dts
- cape–boneblack–hdmin–00A0.dts
- cape–univ–audio–00A0.dts
- cape–univ–emmc–00A0.dts
- cape–univ–hdmi–00A0.dts
- cape–universal–00A0.dts
- cape–universaln–00A0.dts

The following example illustrates how to use the universal cape features. Additional examples will be provided throughout the chapter.

Example: In this example the analog-to-digital conversion overlay is loaded. The analog value is then read from header P9, pin 36 (ADC channel 5).

```
#config-pin overlay BB-ADC
Loading BB-ADC overlay
#config-pin -q p9.36
Pin is not modifyable: P9_36 AIN5
#cd /sys/bus/iio/devices/iio\:device0
#/sys/bus/iio/devices/iio:device0# cat in_voltage5_raw
1850
```

6.8 PROGRAMMING IN C WITH BEAGLEBONE BLACK

General purpose input and output (GPIO) pins may be accessed during run time. In this section we review how to configure pins and set their logic value from within an executing C program.

6.8.1 LINUX GPIO FILES

The Linux operating system uses a file system to access various pins and set their logic values. That is, each pin is accessed as though it were a file. The files may be found in the following BeagleBone Black directory:

```
/sys/class/gpio
```

There are several files associated with each general purpose input/output (GPIO) pin. They are:

- the GPIO "direction" file (input (in) or output (out)) and
- the GPIO "value" file (logic high (1) or low (0)).

These files must be created when access to the pin is required. To create the file the pin's number is sent to an export file. This process is illustrated with an example.

Example: To determine the file to access the GPIO1_12 pin (header P8, pin 12), we must translate its name to its corresponding pin number. The general form of the pin name is:

$$GPIO < bank\ number > _ < pin\ number\ within\ bank >$$

The general purpose pins are divided into three banks: 0, 1, 2, and 3. Each bank has a corresponding offset:

- bank: 0, offset: 0
- bank: 1, offset: 32
- bank: 2, offset: 64
- bank: 3, offset: 96

Also, the pin's number within the bank must be taken into account. The final pin number is determined using:

$$pin\ number = (bank\ number * 32) + pin\ number\ within\ bank$$

For the GPIO1_12 pin, the overall pin number is:

$$pin\ number = (1 * 32) + 12 = 44$$

The two files associated with this pin: "direction" and "value" can now be created. To create these files, the pin number is sent to the export file:

```
# echo 44 > /sys/class/gpio/export
```

Note: The root prompt is designated with the shorthand "#" notation.

The two files created are then located in the gpio44 directory:

```
/sys/class/gpio/gpio44/direction
/sys/class/gpio/gpio44/value
```

6.8.2 CONFIGURING THE GPIO FILES

Since the processor pins are controlled via a file system, they can be accessed from the Linux command line using the "cat" and the "echo" commands.

The cat command is used to concatenate and display files. That is, it displays a file's contents to the computer screen.

The echo command may be used to write a value to a file. The general format of the command is:

```
# echo <some value>  > file_name
```

For example, to set the direction of the GPIO1_12 pin, the following commands may be used:

```
# echo out > /sys/class/gpio/gpio44/direction
# echo 1 > /sys/class/gpio/gpio44/value
```

Example 1: Illuminate an LED connected to the GPIO1_12 pin using Linux commands. An LED is connected to the GPIO1_12 pin (P8, pin 12) using the circuit illustrated in Figure 6.13.

Use the following commands to configure the GPIO1_12 pin and turn the LED on and off.

Establish the GPIO1_12 direction and output files:

```
# echo 44 > /sys/class/gpio/export
```

Set GPIO1_12 for output:

```
# echo out > /sys/class/gpio/gpio44/direction
```

Set GPIO1_12 logic high (1):

```
# echo 1 > /sys/class/gpio/gpio44/value
```

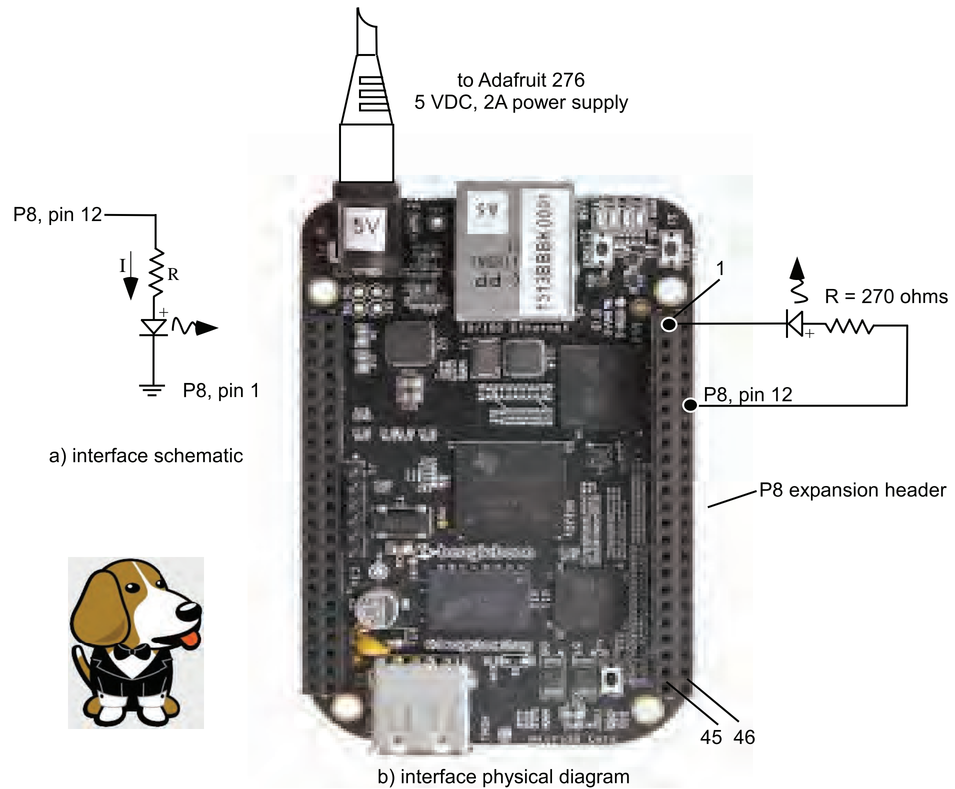


Figure 6.13: Interfacing an LED to BeagleBone. (Illustrations used with permission of Texas Instruments (www.TI.com).)

Set GPIO1_12 logic low (0):

```
# echo 0 > /sys/class/gpio/gpio44/value
```

Example 2: Illuminate an LED connected to the GPIO1_12 pin using the universal overlay and the "config-pin" utility. An LED is connected to the GPIO1_12 pin (P8, pin 12) using the circuit illustrated in Figure 6.13.

```
# config-pin overlay cape-universal
# config-pin -l P8.12
default  gpio  gpio_pu  gpio_pd  pruout  qep
# config-pin P8.12 gpio
# config-pin P8.12 out
# config-pin P8.12 hi
# config-pin P8.12 low
```

6.8.3 ACCESSING THE GPIO FILES IN C

To access the general purpose input/output pins and features from C, we use a similar technique to the Linux command technique. We access the pins and features via the file system described via C file commands.

The C programming language has several helpful functions to open and establish a C file handle (fopen) , set the file position indicator (fseek), write to a file (fprintf), read from a file (fscanf), insure data has been written to a file (fflush), and close a file (fclose) [Kelley and Pohl, 1998]. The provided file function descriptions below are admittedly brief. We only provide the information necessary to configure and use BeagleBone digital input and output pins. Descriptions were adapted from Kelley and Pohl's seminal book on the C programming language *A Book on C—Programming in C* [Kelley and Pohl, 1998].

- **fopen:** This function opens a file for use and attaches the file to a pointer. The file may be opened for reading ("r") or writing ("w"). If a file is used for reading, an infile pointer (ifp) should be attached to it. If used for writing an outfile (ofp) should be attached.
- **fseek:** This function is used with BeagleBone to set the file pointer to the beginning of the file.
- **fprintf:** This function writes variables of a specified type to a file. A wide variety of types may be written to the file: integers("%d"), characters ("%c"), strings ("%s"), and floating point ("%f").
- **fscanf:** This function reads variables of a specified type from a file. A wide variety of types may be read from the file: integers("%d"), characters ("%c"), strings ("%s"), and floating point ("%f").
- **fflush:** This function is used following a write operation (fprintf) to insure unwritten data has been written to a file.
- **fclose:** This function is used to close the specified file.

Several examples are provided to illustrate how these functions are used to configure BeagleBone's general purpose input and output (gpio) pins.

In the examples, note how pins are configured by writing to files written within the Linux structure.

Example 1: In this example we redo the previous example of illuminating an LED from the Linux command line. An LED and a series connected resistor is connected to expansion header P8, pin 12 (GPIO1_12 designated as gpio44), as shown in Figure 6.13. The code may be compiled and executed using:

```
#gcc led1.c -o led1
#./led1
```

```
1 //*****************************************************************
2 //led1.c: illuminates an LED connected to expansion header P8, pin 12
3 //(GPIO1_12 designated as gpio44)
4 //*****************************************************************
5
6 #include <stdio.h>
7 #include <stddef.h>
8 #include <time.h>
9
10 #define OUTPUT "out"
11 #define INPUT  "in"
12
13 int main (void)
14 {
15 //define file handles
16 FILE *ofp_export, *ofp_gpio44_value, *ofp_gpio44_direction;
17
18 //define pin variables
19 int pin_number = 44, logic_status = 1;
20 char* pin_direction = OUTPUT;
21
22 //establish a direction and value file within export for gpio44
23 ofp_export = fopen("/sys/class/gpio/export", "w");
24 if(ofp_export == NULL) {printf("Unable to open export.\n");}
25 fseek(ofp_export, 0, SEEK_SET);
26 fprintf(ofp_export, "%d", pin_number);
27 fflush(ofp_export);
28
29 //configure gpio44 for writing
30 ofp_gpio44_direction = fopen("/sys/class/gpio/gpio44/direction", "w");
31 if(ofp_gpio44_direction==NULL){printf("Unable to open gpio44_direction
     .\n");}
32 fseek(ofp_gpio44_direction, 0, SEEK_SET);
33 fprintf(ofp_gpio44_direction, "%s",  pin_direction);
34 fflush(ofp_gpio44_direction);
35
36 //write a logic 1 to gpio44 to illuminate the LED
37 ofp_gpio44_value = fopen("/sys/class/gpio/gpio44/value", "w");
38 if(ofp_gpio44_value == NULL) {printf("Unable to open gpio44_value.\n")
     ;}
39 fseek(ofp_gpio44_value, 0, SEEK_SET);
40 fprintf(ofp_gpio44_value, "%d", logic_status);
41 fflush(ofp_gpio44_value);
42
43 //close all files
44 fclose(ofp_export);
45 fclose(ofp_gpio44_direction);
46 fclose(ofp_gpio44_value);
```

```
47 return 1;
48 }
49 //***********************************************************************
```

Example 2: In this example we read the logic value of a pushbutton switch connected to expansion header P8, pin 7 (GPIO2_2 designated as gpio66). If the switch is logic high (1) an LED connected to expansion header P8, pin 12 is illuminated. A circuit diagram is provided in Figure 6.14.

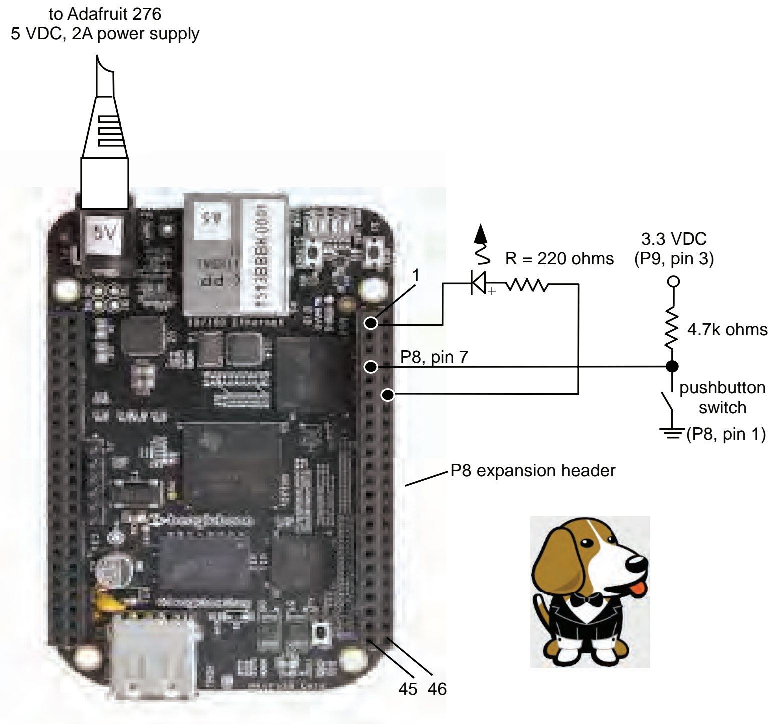


Figure 6.14: Interfacing an LED and switch to BeagleBone. (Illustrations used with permission of Texas Instruments (www.TI.com).)

```
1 //***********************************************************************
2 //led2.c: This program reads the logic value of a pushbutton switch
3 //connected to header P8, pin 7(GPIO2_2 designated as gpio66).
4 //If the switch is logic high (1) an LED connected to expansion header
5 //P8, pin 12 (GPIO1_12 designated as gpio44) is illuminated.
6 //***********************************************************************
7
8 #include <stdio.h>
9 #include <stddef.h>
```

```
10 #include <time.h>
11
12 #define OUTPUT "out"
13 #define input "in"
14
15 int main (void)
16 {
17 //define file handles for gpio44 (P8, pin 12, GPIO1_12)
18 FILE *ofp_export_44, *ofp_gpio44_value, *ofp_gpio44_direction;
19
20 //define file handles for gpio66 (P8, pin 7, GPIO2_2)
21 FILE *ofp_export_66, *ifp_gpio66_value, *ofp_gpio66_direction;
22
23 //define pin variables for gpio44
24 int pin_number_44 = 44, logic_status_44 = 1;
25 char* pin_direction_44 = OUTPUT;
26
27 //define pin variables for gpio66
28 int pin_number_66 = 66, logic_status_66;
29 char* pin_direction_66 = INPUT;
30
31 //create direction and value file for gpio44
32 ofp_export_44 = fopen("/sys/class/gpio/export", "w");
33 if(ofp_export_44 == NULL) {printf("Unable to open export.\n");}
34 fseek(ofp_export_44, 0, SEEK_SET);
35 fprintf(ofp_export_44, "%d", pin_number_44);
36 fflush(ofp_export_44);
37
38 //create direction and value file for gpio66
39 ofp_export_66 = fopen("/sys/class/gpio/export", "w");
40 if(ofp_export_66 == NULL) {printf("Unable to open export.\n");}
41 fseek(ofp_export_66, 0, SEEK_SET);
42 fprintf(ofp_export_66, "%d", pin_number_66);
43 fflush(ofp_export_66);
44
45 //configure gpio44 direction
46 ofp_gpio44_direction = fopen("/sys/class/gpio/gpio44/direction", "w");
47 if(ofp_gpio44_direction == NULL) {printf("Unable to open
       gpio44_direction.\n");}
48 fseek(ofp_gpio44_direction, 0, SEEK_SET);
49 fprintf(ofp_gpio44_direction, "%s", pin_direction_44);
50 fflush(ofp_gpio44_direction);
51
52 //configure gpio66 direction
53 ofp_gpio66_direction = fopen("/sys/class/gpio/gpio66/direction", "w");
54 if(ofp_gpio66_direction == NULL) {printf("Unable to open
       gpio66_direction.\n");}
55 fseek(ofp_gpio66_direction, 0, SEEK_SET);
```

```
56 fprintf(ofp_gpio66_direction, "%s",  pin_direction_66);
57 fflush(ofp_gpio66_direction);
58
59 //configure gpio44 value initially set logic high
60 ofp_gpio44_value = fopen("/sys/class/gpio/gpio44/value", "w");
61 if(ofp_gpio44_value == NULL) {printf("Unable to open gpio44_value.\n")
     ;}
62 fseek(ofp_gpio44_value, 0, SEEK_SET);
63 fprintf(ofp_gpio44_value, "%d", logic_status_44);
64 fflush(ofp_gpio44_value);
65
66 while(1)
67   {
68   //configure gpio66 value and read the gpio66 pin
69   ifp_gpio66_value = fopen("/sys/class/gpio/gpio66/value", "r");
70   if(ifp_gpio66_value == NULL) {printf("Unable to open gpio66_value.\n"
       );}
71   fseek(ifp_gpio66_value, 0, SEEK_SET);
72   fscanf(ifp_gpio66_value, "%d", &logic_status_66);
73   fclose(ifp_gpio66_value);
74   printf("%d", logic_status_66);
75   if(logic_status_66 == 1)
76     {
77     //set gpio44 logic high
78     fseek(ofp_gpio44_value, 0, SEEK_SET);
79     logic_status_44 = 1;
80     fprintf(ofp_gpio44_value, "%d", logic_status_44);
81     fflush(ofp_gpio44_value);
82     printf(" High\n");
83     }
84   else
85     {
86     //set gpio44 logic low
87     fseek(ofp_gpio44_value, 0, SEEK_SET);
88     logic_status_44 = 0;
89     fprintf(ofp_gpio44_value, "%d", logic_status_44);
90     fflush(ofp_gpio44_value);
91     printf(" Low\n");
92     }
93   }
94
95 //close files
96 fclose(ofp_export_44);
97 fclose(ofp_gpio44_direction);
98 fclose(ofp_gpio44_value);
99
100 fclose(ofp_export_66);
101 fclose(ofp_gpio66_direction);
```

```
102 fclose(ifp_gpio66_value);
103
104 return 1;
105 }
106 //*************************************************************************
```

Example 3: In this example we toggle an LED connected to expansion header P8, pin 12 (GPIO1_12 designated as gpio44) at five second intervals.

The program uses the "difftime" function. The definition for this function is included in the "time.h" header file. The "difftime" function prototype is [Kelley and Pohl, 1998].

```
double difftime(time_t t0, time_t t1);
```

The "difftime" function computes the amount of elapsed time (t1-t0) in seconds and returns it as a double. A time hack may be obtained using:

```
1 //define time variable
2 time_t    now;
3
4 //get a time hack
5 now = time(NULL);
```

```
1 //*************************************************************************
2 //led3.c: this programs toggles (flashes) an LED connected to
3 //expansion header P8, pin 12 (GPIO1_12 designated as gpio44) at five
4 //second intervals.
5 //*************************************************************************
6
7 #include <stdio.h>
8 #include <stddef.h>
9 #include <time.h>
10
11 #define  OUTPUT "out"
12 #define  INPUT  "in"
13
14 int  main (void)
15 {
16 //define file handles
17 FILE *ofp_export, *ofp_gpio44_value, *ofp_gpio44_direction;
18
19 //define pin variables
20 int pin_number = 44, logic_status = 1;
21 char* pin_direction = OUTPUT;
22
23 //time parameters
24 time_t  now, later;
25
26 ofp_export = fopen("/sys/class/gpio/export", "w");
27 if(ofp_export == NULL) {printf("Unable to open export.\n");}
```

```
28 fseek(ofp_export, 0, SEEK_SET);
29 fprintf(ofp_export, "%d", pin_number);
30 fflush(ofp_export);
31
32 ofp_gpio44_direction = fopen("/sys/class/gpio/gpio44/direction", "w");
33 if(ofp_gpio44_direction == NULL) {printf("Unable to open
       gpio44_direction.\n");}
34 fseek(ofp_gpio44_direction, 0, SEEK_SET);
35 fprintf(ofp_gpio44_direction, "%s",  pin_direction);
36 fflush(ofp_gpio44_direction);
37
38 ofp_gpio44_value = fopen("/sys/class/gpio/gpio44/value", "w");
39 if(ofp_gpio44_value == NULL) {printf("Unable to open gpio44_value.\n")
       ;}
40 fseek(ofp_gpio44_value, 0, SEEK_SET);
41 logic_status = 1;
42 fprintf(ofp_gpio44_value, "%d", logic_status);
43 fflush(ofp_gpio44_value);
44
45 now = time(NULL);
46 later = time(NULL);
47
48 while(1)
49   {
50   while(difftime(later, now) < 5.0)
51     {
52     later = time(NULL); //keep checking time
53     }
54   if(logic_status == 1) logic_status = 0;
55     else logic_status = 1;
56   //write to gpio44
57   fprintf(ofp_gpio44_value, "%d", logic_status);
58   fflush(ofp_gpio44_value);
59   now=time(NULL);
60   later=time(NULL);
61   }
62 fclose(ofp_export);
63 fclose(ofp_gpio44_direction);
64 fclose(ofp_gpio44_value);
65 return 1;
66 }
67
68 //*********************************************************************
```

In this section we have learned how to configure and use BeagleBone's general purpose input/output (gpio) features. The pins must be systematically configured for proper use.

For the remainder of the chapter we review exposed functions aboard the BeagleBone Black. For each function we provide a bit of theory and then provide examples for BeagleBone Black beginning with the analog-to-digital converter system.

6.9 ANALOG-TO-DIGITAL CONVERTERS (ADC)

A processor may be used to capture analog information from the natural world, determine a course of action based on the information collected and the resident algorithm, and issue control signals to implement the decision. Information from the natural world, is analog or continuous in nature; whereas, a processor is digital. A subsystem to convert an analog signal to a digital form is required. An ADC system performs this task while a digital-to-analog converter (DAC) performs the conversion in the opposite direction.

In this section we discuss the ADC conversion process followed by a discussion of the successive-approximation ADC technique used aboard BeagleBone. We then review the basic features of the BeagleBone ADC system. We conclude our ADC discussion with several illustrative code examples.

6.9.1 ADC PROCESS: SAMPLING, QUANTIZATION, AND ENCODING

In this section, we provide an abbreviated discussion of the ADC process. This discussion was condensed from *Atmel AVR Microcontroller Primer Programming and Interfacing*. The interested reader is referred to this text for additional details and examples [Barrett and Pack, 2008]. We begin with three important stages associated with the ADC: sampling, quantization, and encoding.

Sampling. Sampling is the process of taking "snap shots" of a signal over time. When we sample a signal, we want to minimize the number of samples taken while retaining the ability to reconstruct the original signal from the samples. Intuitively, the rate of change of the signal determines the sample interval required to faithfully reconstruct the signal. We must use the appropriate sampling rate to capture the analog signal for a faithful representation in digital systems.

Harry Nyquist from Bell Laboratory studied the sampling process and derived a criterion that determines the minimum sampling rate for a continuous analog signal. The minimum sampling rate derived is known as the Nyquist sampling rate. It states a signal must be sampled at least twice as fast as the highest frequency content of the signal of interest. For example, the human voice signal contains frequency components that span from approximately 20 Hz to 4 kHz. The Nyquist sample theorem requires sampling the signal at least at 8 kHz, 8,000 "snap shots" every second. Also, when a signal is sampled, a low-pass anti-aliasing filter must be used to insure the Nyquist sampling rate is not violated. In the human voice example, a low-pass filter with a cutoff frequency of 4 kHz would be used before the sampling circuitry for this purpose.

Quantization. In many digital systems, the incoming signals are voltage signals. The voltage signals are obtained from physical variables (pressure, temperature, etc.) via transducers, such as microphones, angle sensors, and infrared sensors. The voltage signals are then conditioned to

map their range with the input range of the digital system. In the case of BeagleBone, the analog signal must be conditioned such that it does not exceed 1.8 VDC.

When an analog signal is sampled, the digital system needs a means to represent the captured samples. The quantization of a sampled analog signal is represented as one of the digital quantization levels. For example, if you have three bits available for quantization, you can represent 8-different levels: 000, 001, 010, 011, 100, 101, 110, and 111. In general, given n bits, we have 2^n unique numbers or quantization levels in our system. Figure 6.15 shows how n bits are used to quantize a range of values. As the number of bits used for the quantization levels increase for a given input range, the "distance" between two adjacent levels decreases accordingly. Intuitively, the more quantization levels means the better mapping of an incoming signal to its true value.

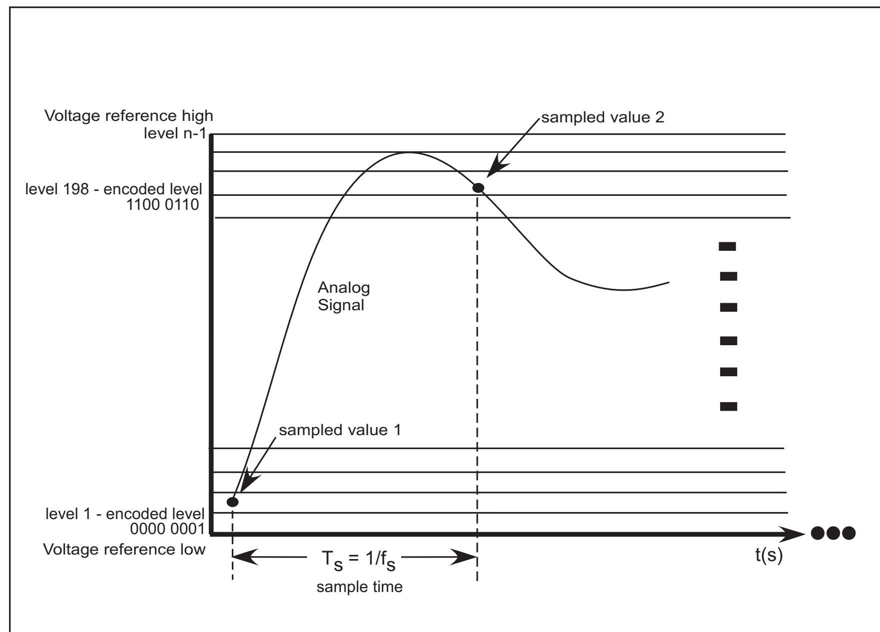


Figure 6.15: Sampling, quantization, and encoding [Barrett and Pack, 2006].

Encoding. The encoding process converts a quantized signal into a digital binary number. Suppose again we are using eight bits to quantize a sampled analog signal. The quantization levels are determined by the eight bits and each sampled signal is quantized as one of 256 quantization levels. Consider the two sampled signals shown in Figure 6.15. The first sample is mapped to quantization level 2 and the second one is mapped to quantization level 198. Note the amount of quantization error introduced for both samples. The quantization error is inversely proportional to the number of bits used to quantize the signal.

6.9.2 RESOLUTION AND DATA RATE

Resolution. Resolution is a measure used to quantize an analog signal. It is the voltage "distance" between two adjacent quantization levels. As we increase the available number of quantization levels within a fixed voltage range, the distance between adjacent levels decreases, reducing the quantization error of a sampled signal. As the number of quantization levels increase, the error decreases, making the representation of a sampled analog signal more accurate in the corresponding digital form. The number of bits used for the quantization is directly proportional to the resolution of a system. In general, resolution may be defined as:

$$resolution = (voltage\ span)/2^b = (V_{ref\ high} - V_{ref\ low})/2^b$$

for BeagleBone, the resolution is:

$$resolution = (1.8 - 0)/2^{12} = 1.8/4096 = 439.45\ \mu V$$

Data rate. Data rate is the amount of data generated by a system per unit time. Typically, the number of bits or the number of bytes per second is used as the data rate of a system. In the previous section, we observed the more bits we use for the quantization levels, the more accurate we can represent a sampled analog signal. So why not use the maximum number of bits when we convert analog signals to digital counterparts? For example, suppose you are working for a telephone company and your switching system must accommodate 100,000 customers. For each individual phone conversation, suppose the company uses an 8 kHz sampling rate (f_s) and 10 bits for the quantization levels for each sampled signal.[1] This means the voice conversation will be sampled every 125 μs (T_s) due to the reciprocal relationship between (f_s) and (T_s). If all customers are making out of town calls, what is the number of bits your switching system must process to accommodate all calls? The answer will be 100,000 × 8,000 × 10 or eight billion bits every second! For these reasons, when making decisions on the number of bits used for the quantization levels and the sampling rate, you must consider the computational burden the selection will produce on the computational capabilities of a digital system versus the required system resolution.

Dynamic range. Dynamic range describes the signal to noise ratio. The unit used for measurement is the Decibel (dB), which is the strength of a signal with respect to a reference signal. The greater the dB number, the stronger the signal is compared to a noise signal. The definition of the dynamic range is $20\ log\ 2^b$ where b is the number of bits used to convert analog signals to digital signals. Typically, you will find 8–12 bits used in commercial analog-to-digital converters, translating the dynamic range from $20\ log\ 2^8$ dB to $20\ log\ 2^{12}$ dB.

6.9.3 ADC CONVERSION TECHNOLOGIES

The ARM processor aboard BeagleBone Black uses a successive approximation converter technique to convert an analog sample into a 12-bit digital representation. The digital value is typically

[1]We ignore overhead involved in processing a phone call such as multiplexing, de-multiplexing, and serial-to-parallel conversion.

represented as an integer value between 0 and 4095. In this section, we discuss this type of conversion process. For a review of other converter techniques, the interested reader is referred to *Atmel AVR Microcontroller Primer: Programming and Interfacing*[Barret and Peck, 2008].

The successive-approximation technique uses a digital-to-analog converter, a controller, and a comparator to perform the ADC process. Starting from the most significant bit down to the least significant bit, the controller turns on each bit one at a time and generates an analog signal, with the help of the digital-to-analog converter, to be compared with the original input analog signal. Based on the result of the comparison, the controller changes or leaves the current bit and turns on the next most significant bit. The process continues until decisions are made for all available bits. Figure 6.16 shows the architecture of this type of converter. The advantage of this technique is that the conversion time is uniform for any input, but the disadvantage of the technology is the use of complex hardware for implementation.

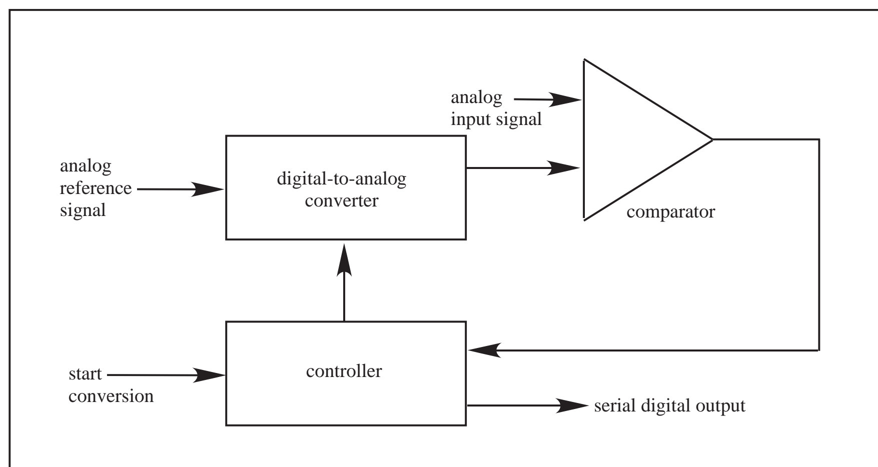


Figure 6.16: Successive-approximation ADC [Barrett and Pack, 2008].

6.9.4 BEAGLEBONE BLACK ADC SYSTEM

BeagleBone Black is equipped with an eight channel ADC system. **The maximum input voltage for each of the channels is 1.8 VDC.** The onboard ADC system uses the successive approximation conversion technique to convert an analog sample into a 12-bit digital value.

Access to the eight analog channels is provided via the following expansion header P9 pins:

- AIN0 pin 39
- AIN1 pin 40
- AIN2 pin 37

- AIN3 pin 38
- AIN4 pin 33
- AIN5 pin 36
- AIN6 pin 35
- Analog ground, GNDA_ADC, pin 34

6.9.5 ADC CONVERSION

Provided in Figure 6.17 is a series of helpful test configurations for the ADC system. Provided in Figure 6.17a is an analog input test configuration. A trimmer potentiometer is set for 1.5 VDC. The trimmer's wiper connection may be connected to ADC input AIN0 (P9, pin 39).

In Figure 6.17b a test configuration is provided to characterize the response of a Sharp GP2YOA21YK0F IR sensor used on the Dagu Magician robot. Results of characterizing the IR sensor are provided in Figure 6.18. The maximum output from the IR sensor may approach 3.0 VDC. Note how a 1 Mohm potentiometer is used to scale the IR sensor output by one-half. This insures the input to the BeagleBone Black does not exceed 1.8 VDC.

Provided in Figure 6.17c is a test configuration for an LM34 precision Fahrenheit temperature sensor. The LM34 provides an output voltage that is linearly related to Fahrenheit temperature. The LM34 is rated over a $-50°$ to $+300°$F range. The LM34 provides an output of $+10.0$ mV/°F.

6.9.6 ADC SUPPORT FUNCTIONS IN BONESCRIPT

In Section 2.2, we introduced the analogRead and analogWrite functions available within Bonescript. The analogRead function performs an ADC conversion on the voltage at a specified pin. The analog voltage must not exceed 1.8 VDC. The analogRead function returns a normalized value from 0 to 1 corresponding to 0 and 1.8 VDC.

The analogWrite function delivers an analog level to a specified pin via a 1 kHz pulse width modulated signal. The analog value is specified as an argument between 0 to 1 corresponding to an analog value between 0 and 1.8 VDC.

Example: An LED is connected to header P8, pin 13, as shown in Figure 6.19. As the potentiometer value is changed, the corresponding intensity of the LED is changed. This is accomplished using program "analog.js" by reading in the analog value from the potentiometer on header P9, pin 36 and reading out the corresponding value on header P8, pin 13 using the Bonescript "b.analogWrite" function.

```
1 //*****************************************************************
2 //analog.js
3 //*****************************************************************
4
```

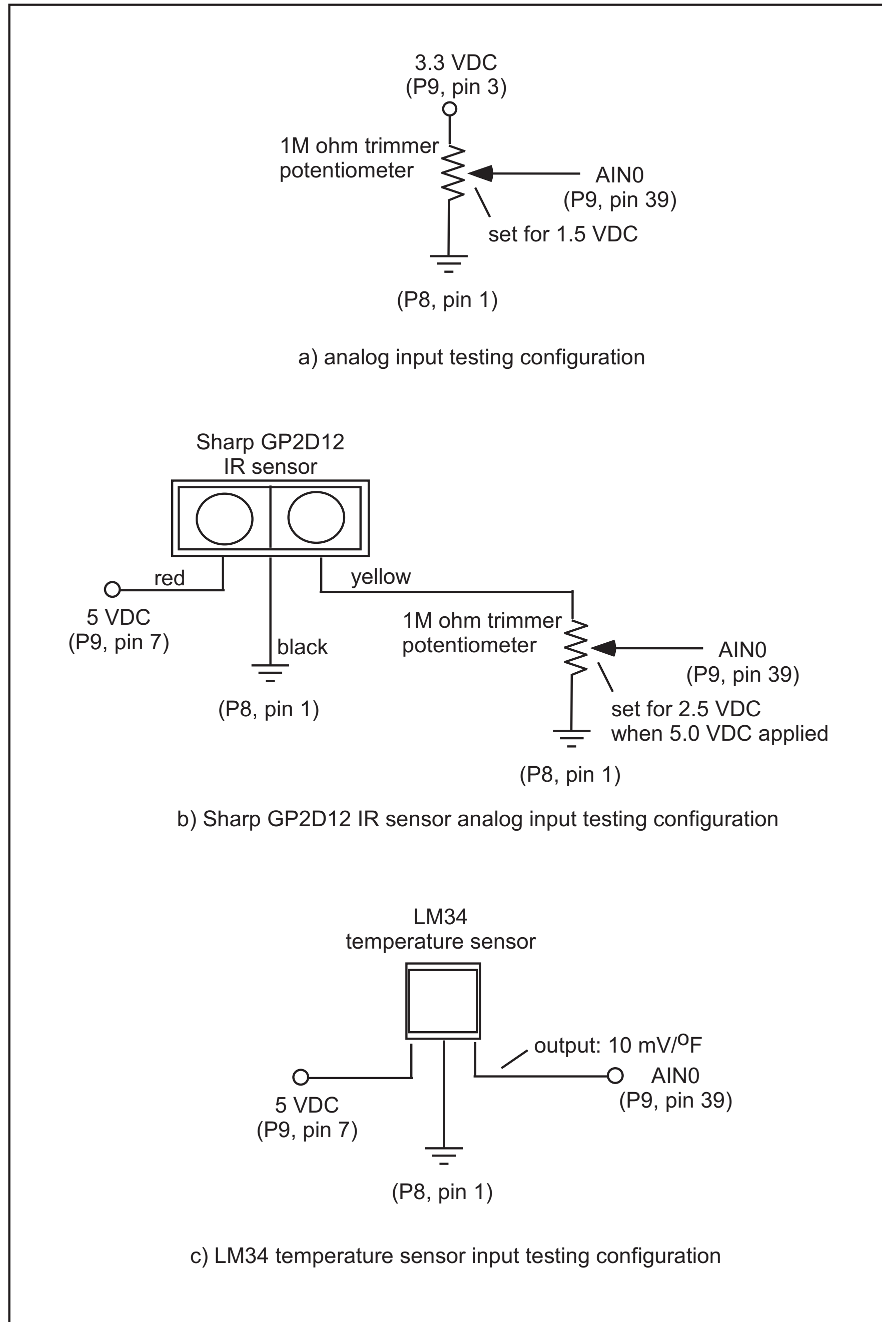
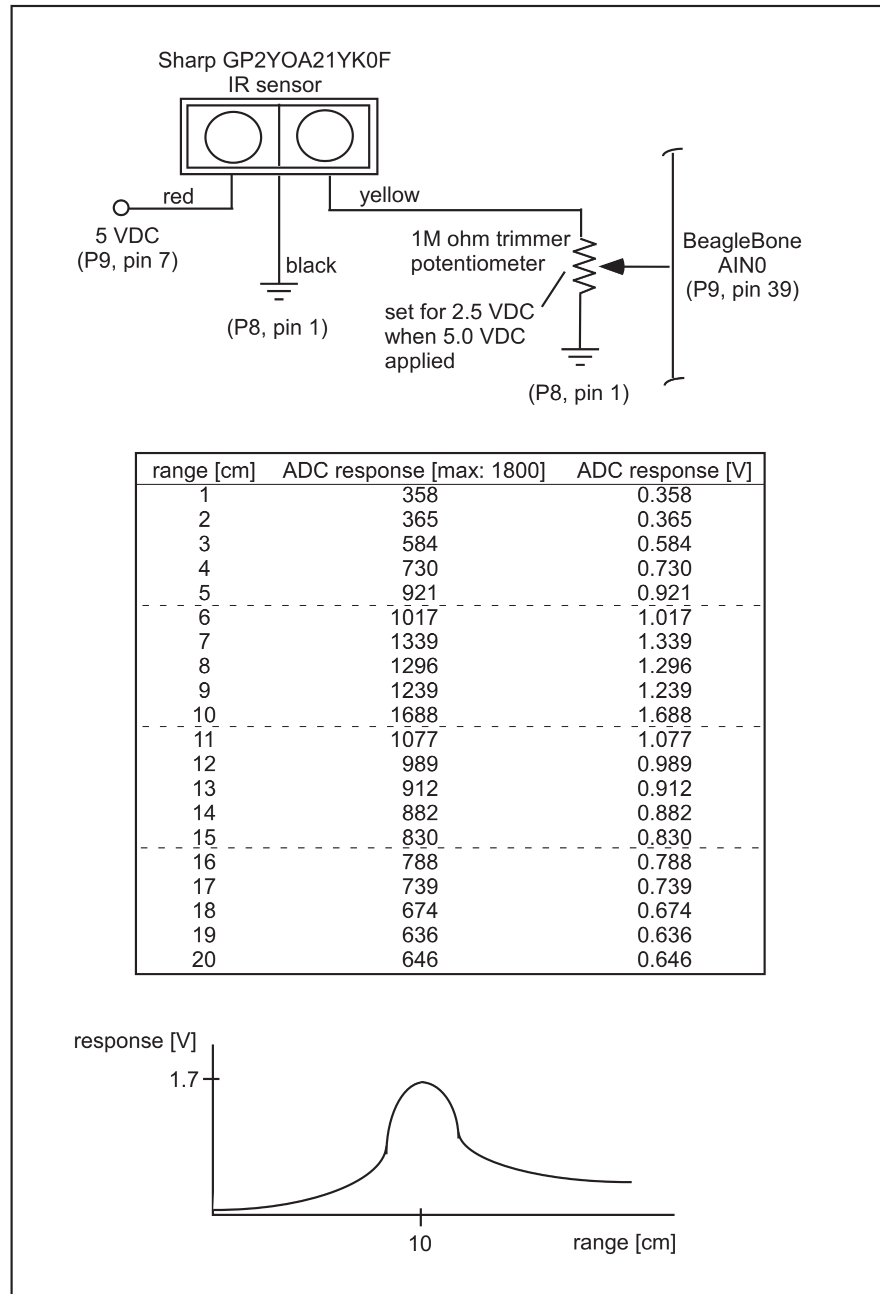


Figure 6.17: ADC test configurations.



range [cm]	ADC response [max: 1800]	ADC response [V]
1	358	0.358
2	365	0.365
3	584	0.584
4	730	0.730
5	921	0.921
6	1017	1.017
7	1339	1.339
8	1296	1.296
9	1239	1.239
10	1688	1.688
11	1077	1.077
12	989	0.989
13	912	0.912
14	882	0.882
15	830	0.830
16	788	0.788
17	739	0.739
18	674	0.674
19	636	0.636
20	646	0.646

Figure 6.18: IR sensor characterization.

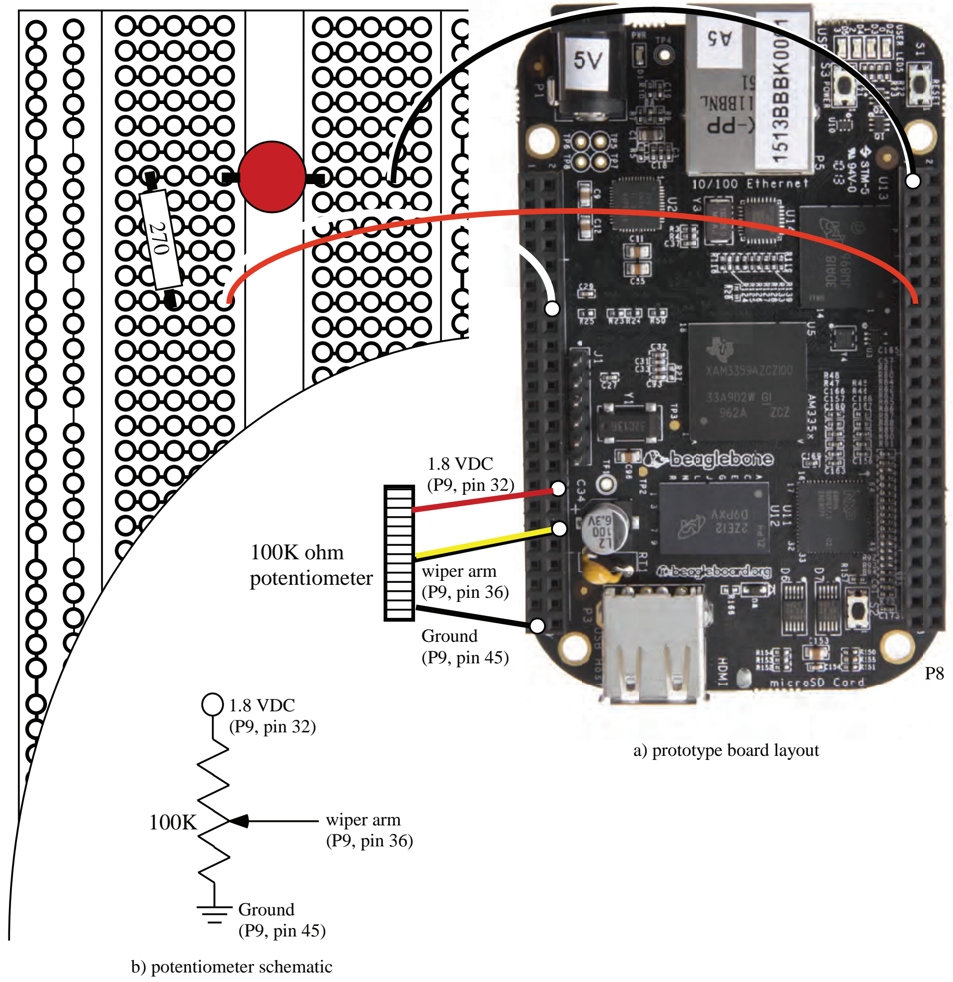


Figure 6.19: Potentiometer connected to header P9, pin 36. As the potentiometer value is changed, the corresponding intensity of the LED is changed. (Illustrations used with permission of Texas Instruments (www.TI.com).)

```
5  var b = require('bonescript');
6
7  inputPin = "P9_36";
8  outputPin = "P8_13";
9
10 b.pinMode(outputPin, b.OUTPUT);
11 loop();
12
13 function loop() {
14     var value = b.analogRead(inputPin);
15     b.analogWrite(outputPin, value);
16     setTimeout(loop, 1);
17 };
18 //*****************************************************************
```

6.9.7 ACCESSING THE ADC SYSTEM IN LINUX

In general, to use a system aboard BeagleBone Black the corresponding device tree overlay must be loaded. The overlay typically enables the system and properly configures the system for use. Alternatively, one of the universal capes, discussed earlier, may be used. Also, if another system conflicts (e.g., HDMI) with the desired system, it must be first disabled.

Note: It must be emphasized that "slots" and "pins" must be loaded upon startup to interact with the BeagleBone Black device tree as described earlier in this chapter.

To access the ADC channels, they must first be enabled using the related ADC device tree overlay (BB-ADC). This may be accomplished by going to the */sys/bus/iio/devices* directory and loading the BB-ADC device tree overlay to SLOTS [Molloy, 2015].

```
# cd /sys/bus/iio/devices
/sys/bus/iio/devices# sudo sh -c ``echo BB-ADC > $SLOTS''
```

The "sudo" command (super user do) as it's name implies provides super user access to the Linux operating system. The "su" command allows switching from user account to another within Linux. If no argument is specified, Linux root access is provided. The "-c" option provides for passing a command to the Linux shell. To verify the overlay was properly load, perform "cat $SLOTS". There should be a new entry containing BB-ADC.

The device tree overlay loads the "iiodevice0" directory which provides access points to read the ADC channels (in_voltage0_raw to in_voltage7_raw). A specific channel value may be read from the Linux command line using:

```
# cd iio:device0
# cat in_voltagen_raw
```

Where "n" specifies the channel of interest. When the command is executed a value between 0 and 4095 is returned. A reading of 0 corresponds to 0 VDC; whereas, 4095 corresponds to 1.8 VDC. Values in between are linearly mapped to appropriate values.

Example 1: Figure 6.17 provides a configuration for testing the ADC system. In this example we connect the potentiometer wiper arm to the input for ADC channel AIN0 (P9, pin 39). Set the potentiometer wiper arm voltage for a value less than 1.8 VDC (e.g., 1.25 VDC). Read the analog value using:

```
>cat in_voltage0_raw
```

Example 2: Another method of accessing the ADC resources from the Linux command line is to install a universal cape. The procedures for doing this were provided earlier in the chapter. Once installed, the ADC may be enabled and a channel voltage accessed using [Steinkuehler]:

```
#config-pin overlay BB-ADC
Loading BB-ADC overlay
#config-pin -q p9.36
Pin is not modifyable: P9_36 AIN5
#cd /sys/bus/iio/devices/iio\:device0
#/sys/bus/iio/devices/iio:device0# cat in_voltage5_raw
1850
```

6.9.8 ADC SUPPORT FUNCTIONS IN C

In this section we provide two examples using C to access the ADC system of BeagleBone Black. As a friendly reminder, before executing the sample code insure the appropriate device tree overlays have been loaded using one of the techniques described above.

Example 1: In this example we read the analog value on AIN0 (P9, pin 39) provided by the LM34 temperature sensor. The connection for the LM34 to BeagleBone is provided in Figure 6.17c. The file used to read AIN0 is first opened (fopen) and then read (fscanf). The value from the LM34 is printed to the terminal (printf). The temperature may be increased by holding the LM34 between your fingers or decreased using compressed air. The readings are accomplished continuously until [Control][C] is used to stop the program.

The LM34 is a precision Fahrenheit temperature sensor. It provides an output voltage that is linearly related to Fahrenheit temperature. It is rated over a $-50°$ to $+300°$F range. The LM34 provides an output of $+10.0$ mV/°F.

The LM34 may be operated from 5–30 VDC. A 5 VDC reference is available from BeagleBone Black at P9, pins 5, 6.

```
1 //*****************************************************************
2 //adc1.c: the analog value on AIN0 (P9, pin 39) provided by
3 //a LM34 temperature sensor is read continuously until
4 //[Control][C] is used to stop the program.
5 //
6 // Note: before executing the sample code insure the SLOTS, PINS,
7 //and the appropriate device tree overlays have been loaded.
8 //*****************************************************************
```

```
 9
10 #include <stdio.h>
11 #include <stddef.h>
12 #include <time.h>
13 #include <math.h>
14
15 #define  OUTPUT "out"
16 #define  INPUT  "in"
17
18 int  main (void)
19 {
20 //define file handles
21 FILE *ifp_ain0;
22 float ain0_value;
23
24 ifp_ain0 = fopen("/sys/bus/iio/devices/iio:device0/in_voltage0_raw", "r
      ");
25 if (ifp_ain0 == NULL) {printf("Unable to AIN0.\n");}
26
27 while(1)
28   {
29   fseek(ifp_ain0, 0, SEEK_SET);
30   fscanf(ifp_ain0, "%f", &ain0_value);
31   printf("%f\n", ain0_value);
32   fflush(ifp_ain0);
33   }
34
35 fclose(ifp_ain0);
36 return 1;
37 }
38 //*************************************************************
```

Example 2: In this example readings of the LM34 are taken at two second intervals. The value from AIN0 is read and printed to the terminal. Each reading is converted to a voltage and also a temperature. A #define statement is used to link the file path for ain1 to a more convenient (and shorter) name. The #define statements could be moved to a header file.

```
 1 //*************************************************************
 2 //adc2.c  the analog value on AIN0 (P9, pin 39) provided by
 3 //a LM34 temperature sensor is read at 2 second intervals until
 4 //[Control][C] is used to stop the program.
 5 //
 6 // Note: before executing the sample code insure the SLOTS, PINS,
 7 //and the appropriate device tree overlays have been loaded.
 8 //*************************************************************
 9 //*************************************************************
10
11 //include files
12 #include <stdio.h>
```

```
13 #include <stddef.h>
14 #include <time.h>
15 #include <math.h>
16
17 //define
18 #define OUTPUT "out"
19 #define INPUT "in"
20 #define ain0_in "/sys/bus/iio/devices/iio:device0/in_voltage0_raw"
21
22 //function prototypes
23 void delay_sec(float delay_value);
24
25 int main (void)
26 {
27 //define file handles
28 FILE *ifp_ain0;
29 float ain0_value;
30 float ain0_voltage;
31 float ain0_temp;
32
33 ifp_ain0=fopen(ain0_in, "r");
34 if(ifp_ain0==NULL){printf("Unable to ain0.\n");}
35
36 while(1)
37   {
38   fseek(ifp_ain0, 0, SEEK_SET);
39   fscanf(ifp_ain0, "%f", &ain0_value);
40   printf("AIN0 reading [of 4095]: %f\n", ain0_value);
41   ain0_voltage = ((ain0_value/4095.0) * 1.8);
42   printf("AIN0 voltage [V]: %f\n", ain0_voltage);
43   ain0_temp = (ain0_voltage/.010);
44   printf("AIN0 temperature [F]: %f\n\n", ain0_temp);
45   delay_sec(2.0);
46   fflush(ifp_ain0);
47   }
48 fclose(ifp_ain0);
49 return 1;
50 }
51
52 //*****************************************************************
53 //function definitions
54 //*****************************************************************
55
56 void delay_sec(float delay_value)
57 {
58 time_t now, later;
59
60 now = time(NULL);
```

```
61 later = time (NULL);
62
63 while(difftime(later, now) < delay_value)
64   {
65   later = time(NULL);              //keep checking time
66   }
67 }
68
69 //*************************************************************
```

Linux has several built-in delay functions including:

- **sleep(delay):** delay is specified in seconds as an unsigned integer. The file "unistd.h" must be included;
- **usleep(delay):** delay is specified in microseconds as an unsigned integer. The file "unistd.h" must be included; and
- **nanosleep(delay):** delay is specified in nanoseconds as an unsigned integer. The file "unistd.h" must be included.

6.10 SERIAL COMMUNICATIONS

Processors must often exchange data with peripheral devices. Data may be exchanged by using parallel or serial techniques. With parallel techniques, an entire byte or word of data is sent simultaneously from the transmitting device to the receiving device. While this is efficient from a time point of view, it requires multiple, parallel lines for data transfer which impacts system cost.

In serial transmission, a byte of data is sent a single bit at a time. Once eight bits have been received at the receiver, the data byte is reconstructed. While this is inefficient from a time point of view, it only requires a line (or two) to transmit the data.

Serial communication techniques provide a vital link between BeagleBone and input devices and output devices. In this section, we investigate the serial communication features beginning with a review of serial communication concepts and terminology. We then investigate serial communication systems available on BeagleBone: the Universal Asynchronous Receiver and Transmitter (UART), the Serial Peripheral Interface (SPI), and networking features. Before discussing the different serial communication features aboard BeagleBone, we review serial communication terminology.

6.10.1 SERIAL COMMUNICATION TERMINOLOGY

In this section, we review common terminology associated with serial communication.

Asynchronous vs. Synchronous Serial Transmission: In serial communications, the transmitting and receiving device must be synchronized to one another and use a common data rate

and protocol. Synchronization allows both the transmitter and receiver to be expecting data transmission/reception at the same time. There are two basic methods of maintaining "sync" between the transmitter and receiver: asynchronous and synchronous.

In an asynchronous serial communication system, such as the UART, framing bits are used at the beginning and end of a data byte. These framing bits alert the receiver that an incoming data byte has arrived and also signals the completion of the data byte reception. The data rate for an asynchronous serial system is typically much slower than the synchronous system, but it only requires a single wire between the transmitter and receiver.

A synchronous serial communication system maintains "sync" between the transmitter and receiver by employing a common clock between the two devices. Data bits are sent and received on the edge of the clock. This allows data transfer rates higher than with asynchronous techniques but requires two lines, data and clock, to connect the receiver and transmitter.

Baud Rate: Data transmission rates are typically specified as a Baud rate or bits per second rate. For example, 9600 Baud indicates the data is being transferred at 9600 bits per second.

Full Duplex: Often serial communication systems must both transmit and receive data. To perform transmission and reception simultaneously requires separate hardware for transmission and reception. A single duplex system has a single complement of hardware that must be switched from transmission to reception configuration. A full duplex serial communication system has separate hardware for transmission and reception.

Non-return to Zero (NRZ) Coding Format: There are many different coding standards used within serial communications. The important point is the transmitter and receiver must use a common coding standard so data may be interpreted correctly at the receiving end. In NRZ coding a logic one is signaled by a logic high during the entire time slot allocated for a single bit; whereas, a logic zero is signaled by a logic low during the entire time slot allocated for a single bit.

The RS-232 Communication Protocol: When serial transmission occurs over a long distance, additional techniques may be used to insure data integrity. Over long distances logic levels degrade and may be corrupted by noise. At the receiving end, it is difficult to discern a logic high from a logic low. The RS-232 standard has been around for some time. With the RS-232 standard (EIA-232), a logic one is represented with a −12 VDC level while a logic zero is represented by a +12 VDC level. Chips are commonly available (e.g., MAX232) that convert the 5 and 0 V output levels from a transmitter to RS-232 compatible levels and convert back to 5 V and 0 V levels at the receiver. For BeagleBone Black a MAX3232 may be used to convert 3.3 VDC signals to RS-232 compatible levels and back [MAXIM]. The RS-232 standard also specifies other features for this communication protocol. The standard specifies several signals including the following [Horowitz and Hill, 1992].

- **TX:** transmit
- **RX:** receive

- **CTS:** clear to send
- **RTS:** request to send

Depending on the specific peripheral device connected, some or all of these pins will be used in a given application.

The serial communication logic levels from BeagleBone are at 3.3 VDC for logic one and 0 VDC for logic zero. To communicate with an RS-232 device a series of level shifters are required. As shown in Figure 6.20a, the signals from BeagleBone must be first shifted to TTL compatible levels (0 and 5 VDC) and then to RS-232 compatible levels.

Figure 6.20: Serial communication level shifting.

Parity: To further enhance data integrity during transmission, parity techniques may be used. Parity is an additional bit (or bits) that may be transmitted with the data byte. With a single parity bit, a single bit error may be detected. Parity may be even or odd. In even parity, the parity

bit is set to one or zero such that the number of ones in the data byte including the parity bit is even. In odd parity, the parity bit is set to one or zero such that the number of ones in the data byte including the parity bit is odd. At the receiver, the number of bits within a data byte including the parity bit are counted to insure that parity has not changed, indicating an error, during transmission.

ASCII: The American Standard Code for Information Interchange or ASCII is a standardized, 7-bit method of encoding alphanumeric data. It has been in use for many decades, so some of the characters and actions listed in the ASCII table are not in common use today. However, ASCII is still the most common method of encoding alphanumeric data. The ASCII code is provided in Figure 6.21. For example, the capital letter "G" is encoded in ASCII as 0 × 47. The "0x" symbol indicates the hexadecimal number representation. Unicode is the international counterpart of ASCII. It provides standardized 16-bit encoding format for the written languages of the world. ASCII is a subset of Unicode. The interested reader is referred to the Unicode home page website, `www.unicode.org`, for additional information on this standardized encoding format.

6.10.2 SERIAL UART

The serial UART (or Universal Asynchronous Receiver and Transmitter) provides for full duplex (two way) communication between a receiver and transmitter. This is accomplished by equipping the processor with independent hardware for the transmitter and receiver. The UART is typically used for asynchronous communication. That is, there is not a common clock between the transmitter and receiver to keep them synchronized with one another. To maintain synchronization between the transmitter and receiver, framing start and stop bits are used at the beginning and end of each data byte in a transmission sequence as shown in Figure 6.21b. A parity bit may also be included.

BeagleBone UART Subsystem Description

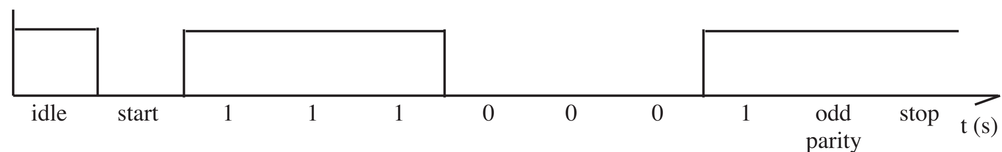
BeagleBone is equipped with six UART channels designated UART 0 through UART 5. UART 0 is dedicated to the USB port while UART 1–5 are available to the user via the expansion ports as summarized in Figure 6.22.

To properly configure the UART system the following parameters must be set:

- baud rate,
- character size,
- parity type, and
- flow control type.

Least significant digit	Most significant digit 0x0_	0x1_	0x2_	0x3_	0x4_	0x5_	0x6_	0x7_
0x_0	NUL	DLE	SP	0	@	P	`	p
0x_1	SOH	DC1	!	1	A	Q	a	q
0x_2	STX	DC2	“	2	B	R	b	r
0x_3	ETX	DC3	#	3	C	S	c	s
0x_4	EOT	DC4	$	4	D	T	d	t
0x_5	ENQ	NAK	%	5	E	U	e	u
0x_6	ACK	SYN	&	6	F	V	f	v
0x_7	BEL	ETB	‘	7	G	W	g	w
0x_8	BS	CAN	(	8	H	X	h	x
0x_9	HT	EM	)	9	I	Y	i	y
0x_A	LF	SUB	*	:	J	Z	j	z
0x_B	VT	ESC	+	;	K	[	k	{
0x_C	FF	FS	‘	<	L	\	l	|
0x_D	CR	GS	-	=	M	]	m	}
0x_E	SO	RS	.	>	N	^	n	~
0x_F	SI	US	/	?	O	_	o	DEL

a) ASCII Table



b) UART waveform for “G” with odd parity

Figure 6.21: ASCII Code. The ASCII code is used to encode alphanumeric characters. The “0x” indicates hexadecimal notation in the C programming language [Barrett and Pack, 2006].

UART Channel 0 Serial Connection

A connection may be established between the host computer and BeagleBone Black via UART channel 0. UART channel 0 is accessible via the 6-pin header connector located next to header pin P9, as shown in Figure 6.23. A USB-to-TTL serial cable (USB console cable, Adafruit #954) connects the host to BeagleBone Black. Care must be taken to insure connections are made as shown in Figure 6.23. The red cable (5 VDC) should **not** be connected to the UART channel 0 header connector.

Embedded within the USB-to-TTL serial cable is a Prolific (Technology for Tomorrow) PL-2303HX Rev A USB-to-Serial Controller chip. Drivers and installation instructions for the cable are provided at `www.prolific.com.tw`. With drivers installed you can access BeagleBone Black using the PuTTy terminal emulator software.

UART channel	serial port	RX	TX	CTS	RTS
UART1	/dev/tty01	P9, pin 26 Mux mode: 0	P9, pin 24 Mux mode: 0	P9, pin 20 Mux mode: 0	P9, pin 19 Mux mode: 0
UART1	/dev/tty02	P9, pin 22 Mux mode: 1	P9, pin 21 Mux mode: 1	P9, pin 37 Mux mode: 1	P9, pin 38 Mux mode: 1
UART1	/dev/tty03	not avail on header	not avail on header	not avail on header	not avail on header
UART1	/dev/tty04	P9, pin 11 Mux mode: 6	P9, pin 13 Mux mode: 6	P8, pin 35 Mux mode: 6	P8, pin 33 Mux mode: 6
UART1	/dev/tty05	P8, pin 38 Mux mode: 4	P8, pin 37 Mux mode: 4		

Note: UART0 is accessible via the Serial Debug Port, 6 pin – header.

Figure 6.22: BeagleBone UART summary.

UART Support in Bonescript

Earlier in the book we mentioned Bonescript is a rapidly evolving, open-source development platform. Bonescript consists of a JavaScript library of functions to rapidly develop a variety of physical computing applications. It will grow in capability as other users develop additional features to enhance Bonescript. In that light, Bonescript functions to support UART serial communications are available at: `https://github.com/voodootikigod/node-serialport`. The two primary functions are:

- serialOpen(port, options, [callback]) and
- serialWrite(port, data, [callback]).

UART Support Via Termios API and dts Files in C

The UART features are usually not accessed directly but instead through a series of support functions via an Applications Programming Interface (API). We use the termios API in the following example. The API provides helpful functions for asynchronous communications. To access termios features, the following files should be included within a program: stdio.h, termios.h, fcntl.h, and unistd.h.

The BeagleBone provides UART support via device tree overlays. A listing of available overlays is displayed using [Molloy, 2015].

```
# cd /lib/firmware
/lib/firmware# ls *UART*
```

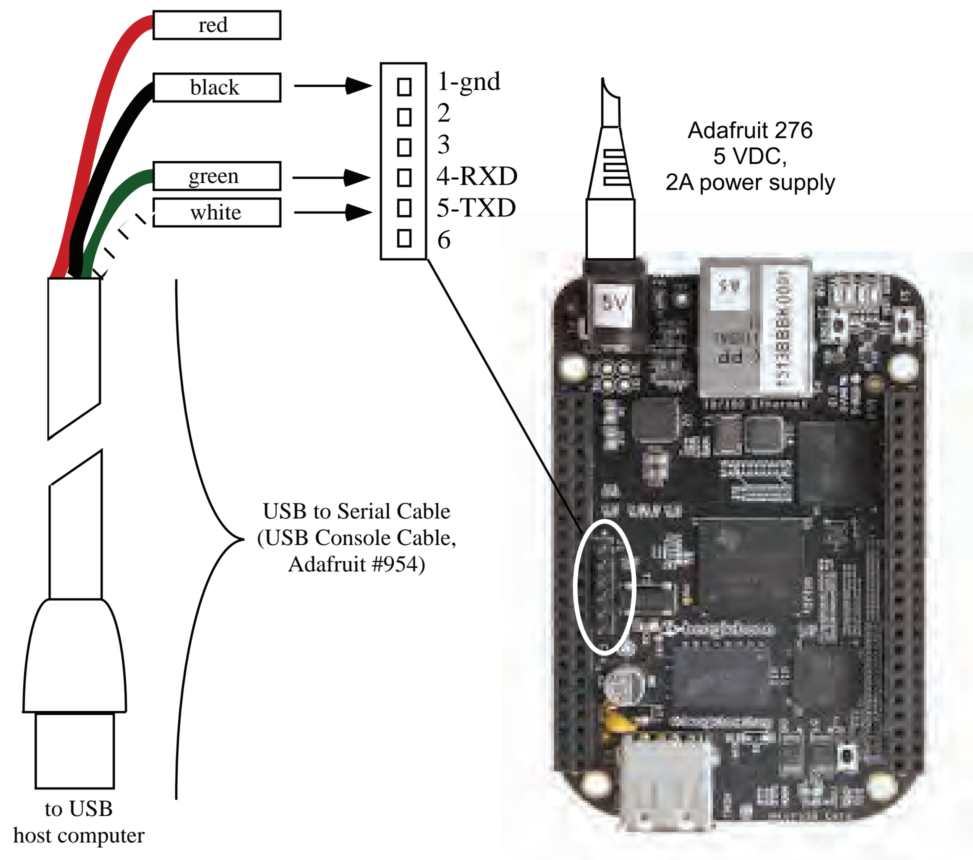


Figure 6.23: BeagleBone access via UART channel 0. (Illustrations used with permission of Texas Instruments (www.TI.com).)

It must be emphasized that "slots" and "pins" must be loaded upon startup to interact with the BeagleBone Black device tree.

In the following example we use UART1. To load and verify the device tree overlay for UART1 use:

```
/lib/firmware# sudo su -c ``echo BB-UART1 > $SLOTS''
/lib/firmware# cat $SLOTS
```

The device directory (/dev) should now have an entry for ttyO1.

Example 1: In this example UART1 is configured for transmission at 9600 BAUD and continuously communicates the ASCII character "G."

As a friendly reminder, before executing the sample code insure the SLOTS, PINS, and the appropriate device tree overlays have been loaded. The SLOTS and PINS are loaded using:

```
# export SLOTS=/sys/devices/bone_capemgr.9/slots
# export PINS=/sys/kernel/debug/pinctrl/44e10800.pinmux/pins
```

To load the device tree overlay for UART channel 1, use the following commands [Molloy, 2015]:

```
# cd /lib/firmware
/lib/firmware# sudo su -c ``echo BB-UART1 > $SLOTS''
```

Example 2: As an alternative, the UART1 device tree overlay may be loaded using [Steinkuehler]:

```
# config-pin overlay cape-universal
# config-pin overlay BB-UART1
```

In this example BeagleBone UART1 is configured for tranmission and 9600 Baud and repeatedly sends the character G via UART1 TX pin (P9, 24).

```
1 //*********************************************************************
2 //uart1.c - configures BeagleBone uart1 for tranmission and 9600 Baud
3 //and repeatedly sends the character G via uart1 tx pin (P9, 24)
4 //
5 //Note: Before executing the sample code insure the SLOTS, PINS, and
6 //the appropriate device tree overlays have been loaded.
7 //*********************************************************************
8
9 #include <stdio.h>
10 #include <stddef.h>
11 #include <time.h>
12 #include <termios.h>
13 #include <fcntl.h>
14 #include <unistd.h>
15 #include <sys/types.h>
16 #include <string.h>
17
18 int main(void)
19 {
20 //define file handle for uart1
21 FILE *ofp_uart1_tx, *ofp_uart1_rx;
22
23 //uart1 configuration using termios
24 struct termios uart1;
25 int fd;
```

```
26
27 //open uart1 for tx/rx, not controlling device
28 if((fd = open("/dev/ttyO1", O_RDWR | O_NOCTTY)) < 0)
29   printf("Unable to open uart1 access.\n");
30
31 //get attributes of uart1
32 if(tcgetattr(fd, &uart1) < 0)
33   printf("Could not get attributes of UART1 at ttyO1\n");
34
35 //set Baud rate
36 if(cfsetospeed(&uart1, B9600) < 0)
37   printf("Could not set baud rate\n");
38 else
39   printf("Baud rate: 9600\n");
40
41 //set attributes of uart1
42 uart1.c_iflag = 0;
43 uart1.c_oflag = 0;
44 uart1.c_lflag = 0;
45 tcsetattr(fd, TCSANOW, &uart1);
46
47 char byte_out[] = {0x47};
48
49 //set ASCII character G repeatedly
50 while(1)
51   {
52   write(fd, byte_out, strlen(byte_out)+1);
53   }
54
55 close(fd);
56 }
57
58 //************************************************************
```

BeagleBone may be equipped with RS-232 compatible features using the BeagleBone RS232 Cape. Full documentation and software support is available for the Cape [CircuitCo, 2015].

6.10.3 SERIAL PERIPHERAL INTERFACE (SPI)

The Serial Peripheral Interface or SPI also provides for two-way serial communication between a transmitter and a receiver. In the SPI system, the transmitter and receiver share a common clock source. This requires an additional clock line between the transmitter and receiver but allows for higher data transmission rates as compared to the USART. The SPI system allows for fast and efficient data exchange between microcontrollers or peripheral devices. There are many SPI compatible external systems available to extend the features of the microcontroller. For ex-

ample, a liquid crystal display or a multi-channel digital-to-analog converter could be added to the processor using the SPI system.

SPI Operation

The SPI may be viewed as a synchronous 16-bit shift register with an 8-bit half residing in the transmitter and the other 8-bit half residing in the receiver, as shown in Figure 6.24. The transmitter is designated the master since it is providing the synchronizing clock source between the transmitter and the receiver. The receiver is designated as the slave. A slave is chosen for reception by taking its Slave Select line low. When the line is taken low, the slave's shifting capability is enabled.

SPI transmission is initiated by loading a data byte into the master configured SPI Data Register. At that time, the SPI clock generator provides clock pulses to the master and also to the slave via the serial clock pin. A single bit is shifted out of the master designated shift register on the Master Out Slave In (MOSI) processor pin on every serial clock pulse. The data is received at the MOSI pin of the slave designated device. At the same time, a single bit is shifted out of the Master In Slave Out (MISO) pin of the slave device and into the MISO pin of the master device. After eight master serial clock pulses, a byte of data has been exchanged between the master and slave designated SPI devices. The serial transmission does not have to be bi-directional. In these applications the return line from the slave to the master device is not connected.

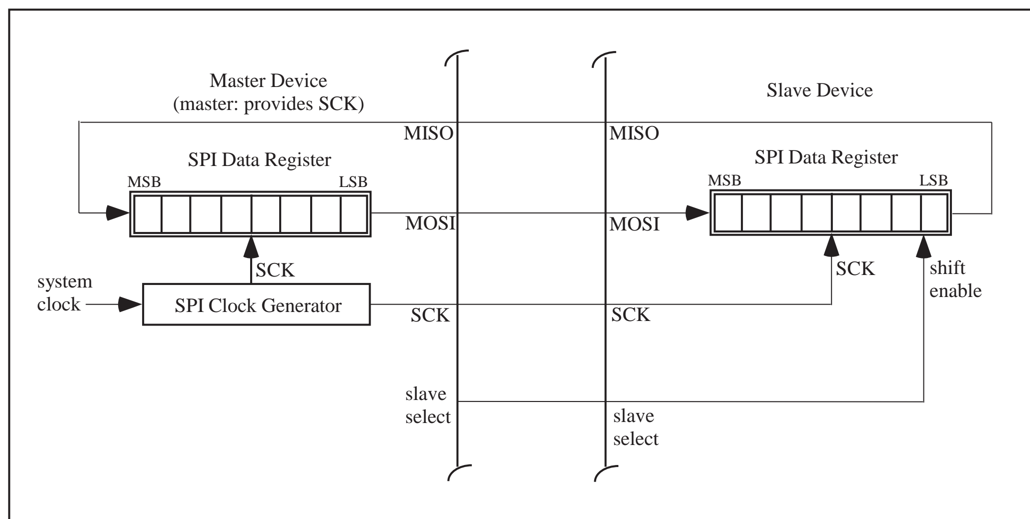


Figure 6.24: SPI overview.

Bonescript SPI Support

An SPI signal may be generated on many different BeagleBone pins using the Bonescript function shiftOut. The format of the function is:

```
shiftOut(dataPin, clockPin, bitOrder, value);
```

The bitOrder may be set for MSBFIRST (most significant bit first) or LSBFIRST (least significant bit first). In the example below the value $0 \times AC$ is continuously sent out via BeagleBone P8, pin 3. The SPI clock signal is sent via P8, pin 11. The bit rate in this example is approximately 1,500 bits per second. The SPI data and clock signals may be examined using an oscilloscope or logic analyzer.

```
1 //***********************************************************
2
3 var b = require('bonescript');
4
5 setup = function() {};
6
7 var spi_data_pin = ``P8_13'';
8 var spi_clk_pin = ``P8_11'';
9
10 b.pinMode(spi_data_pin, b.OUTPUT);
11 b.pinMode(spi_clk_pin, b.OUTPUT);
12 b.shiftOut(spi_data_pin, spi_clk_pin, b.LSBFIRST, 0xAC);
13
14 //***********************************************************
```

BeagleBone SPI Features in C

For additional flexibility and increased data rate, BeagleBone dedicated SPI features may be used. BeagleBone is equipped with two SPI channels designated SPIO0 and SPI1. The SPI features may be accessed via the header pins, as shown in Figure 6.25. The HDMI interface must be disabled to use SPI channel 1 on BeagleBone Black.

As with the other BeagleBone systems, there is considerable support for the SPI system via the device tree overlays. To enable the SPI system via overlays, the following commands may be used [Steinkuehler]:

```
# config-pin overlay cape-universal
# config-pin P9.17 spi
# config-pin P9.18 spi
# config-pin P9.21 spi
# config-pin P9.22 spi
```

The SPI related files are then loaded to the "/dev" directory. You can examine the directory contents using:

BeagleBone Serial Peripheral Interface (SPI) features		
Header pin	**Signal name**	**Mode**
SPIO0		
P9.17	spi0_cs0	0
P9.18	spi0_d1	0
P9.21	spi0_d0	0
P9.22	spi0_sclk	0
SPI1		
P9.28	spi1_cs0	3
P9.29	spi1_d0	3
P9.30	spi1_d1	3
P9.31	spi1_sclk	3

Figure 6.25: BeagleBone SPI features.

```
# cd /dev
/dev#  ls sp*
```

You should see entries for spidev 1.0, 1.1, 2.0, and 2.1.

To test the SPI system, locate a BeagleBone Black compatible copy of "spidev_test.c" on the Internet. Compile and execute the code using techniques discussed earlier in the chapter. When executed the program reports spi mode, bits per word, maximum speed, and the data transferred. When executed the data transmitted will be reported as all "FF." If SPI pins P9.18 (spi0_d1) is connected to P9.21 (spio_d0) and the program executed, the data reported will be the same as provided in the array defined within "spidev_test.c." To further experiment with the SPI features, modify the "spidev_test.c" program [Molloy].

There is considerable information on this topic and C support functions available in the following examples.

Example 1: The article "BeagleBone Black Enable SPIDEV" posted on `www.elinux.org/BeagleBone_Black_Enable_SPIDEV` provides step-by-step instructions to access SPI features on BeagleBone Black. It also provides sample programs for testing.

Example 2: Adafruit provides an excellent example employing a device tree overlay for SPI channel 0 (SPI0). The reader is highly encouraged to work this example. It is entitled "Introduction to the BeagleBone Black Device Tree" and is available for download from `www.adafruit.com`.

Example 3: LED strip. LED strips may be used for motivational (fun) optical displays, games, or for instrumentation-based applications. In this example we control an LPD8806-based LED

strip using Energia. We use a one meter, 32 RGB LED strip available from Adafruit (#306) for approximately $30 USD [www.adafruit.com].

The red, blue, and green component of each RGB LED is independently set using an 8-bit code. The most significant bit (MSB) is logic one followed by seven bits to set the LED intensity (0–127). The component values are sequentially shifted out of BeagleBone Black using the Serial Peripheral Interface (SPI) features. The first component value shifted out corresponds to the LED nearest the microcontroller. Each shifted component value is latched to the corresponding R, G, and B component of the LED. As a new component value is received, the previous value is latched and held constant. An extra byte is required to latch the final parameter value. A zero byte $(00)_{16}$ is used to complete the data sequence and reset back to the first LED [www.adafruit.com].

Only four connections are required between the BeagleBone Black and the LED strip, as shown in Figure 6.26. The connections are color coded: red-power, black-ground, yellow-data, and green-clock. It is important to note the LED strip requires a supply of 5 VDC and a current rating of 2 amps per meter of LED strip. In this example we use the Adafruit #276 5V 2A (2000 mA) switching power supply [www.adafruit.com].

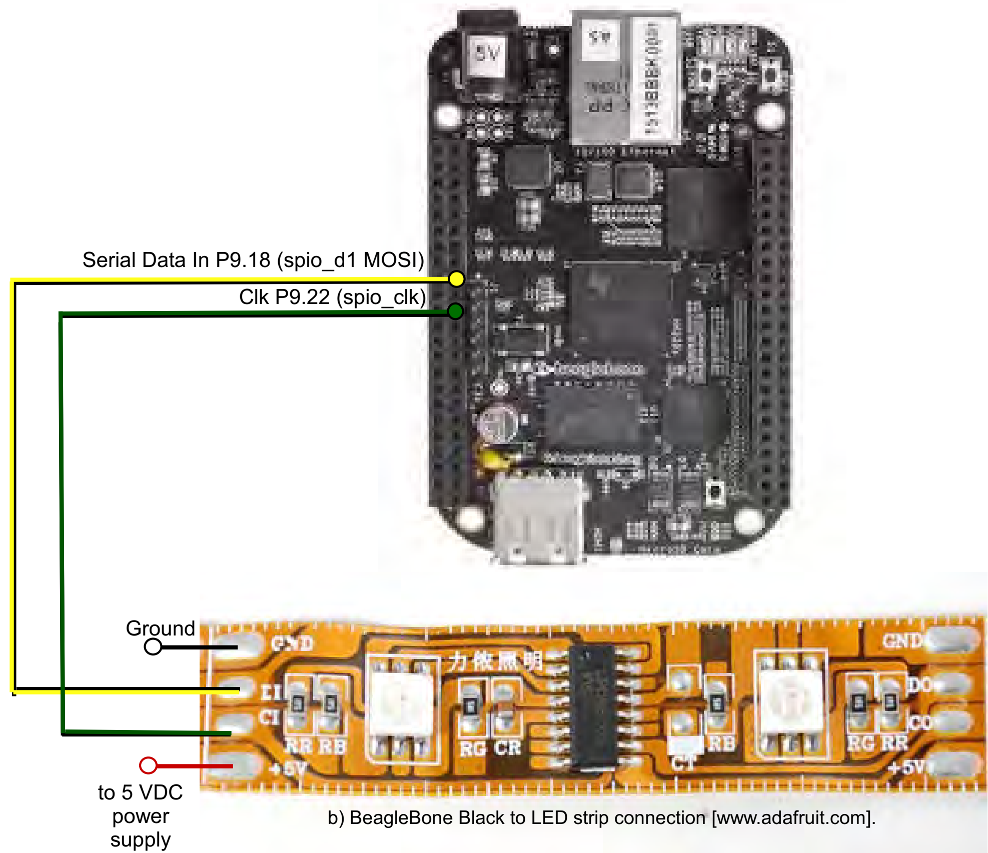
In this example each RGB component is sent separately to the strip. The example illustrates how each variable in the program controls a specific aspect of the LED strip. Here are some important implementation notes.

- SPI must be configured for most significant bit (MSB) first.
- LED brightness is seven bits. Most significant bit (MSB) must be set to logic one.
- Each LED requires a separate R-G-B intensity component. The order of data is G-R-B.
- After sending data for all LEDs. A byte of (0×00) must be sent to return strip to first LED.
- Data stream for each LED is: 1-G6-G5-G4-G3-G2-G1-G0-1-R6-R5-R4-R3-R2-R1-R0-1-B6-B5-B4-B3-B2-B1-B0.

The SPI communication code was adapted from an example contained within "Exploring BeagleBone Tools and Techniques for Building with Embedded Linux," a great book by Derek Molloy. The code example was used with permission of Derek Molloy.

```
1  //*********************************************************************
2  //spi_led_strip.c
3  //-- Demonstrates BeagleBone Black SPI configuration
4  //-- Demonstrates LED strip interface and operation
5  //
6  //Adapted from:
7  //-- spidev_test.c  [www.kernel.org]
8  //-- Original spidev_test.c  program developed by Anton Vorontsov
9  //-- spidev_test.c :
10 //   This program is free software; you can redistribute it and/or modify
11 //   it under the terms of the GNU General Public License as published by
12 //   the Free Software Foundation; either version 2 of the License.
13 //-- The SPI transmission portion of the code was written by Derek Molloy.
14 //-- The LED strip portion of the code was written by Steve Barrett.
15 //
16 //Insure universal and SPI capes are loaded and SPI0 pins are configured
17 //for SPI operation using config-pin utility
```

a) LED strip by the meter [www.adafruit.com].



b) BeagleBone Black to LED strip connection [www.adafruit.com].

Figure 6.26: BeagleBone Black controlling LED strip [www.adafruit.com].

```
18  //   P9.17    spi0_cs0
19  //   P9.18   spi0_d1
20  //   P9.21   spi0_d0
21  //   P9. 22   spio_sclk
22  //
23  //LED strip LDP8806 - available from www.adafruit.com (#306)
24  //
25  //Connections:
26  // - External 5 VDC supply - Adafruit 5 VDC, 2A (#276) - red
27  // - Ground - black
28  // - Serial Data In  -  BeagleBone pin P9.18 (spio_d1 MOSI) - yellow
29  // - CLK - BeagleBone pin P9.22 (spio_clk) - green
30  //
31  //Variables:
32  // - LED_brightness - set intensity from 0 to 127
33  // - segment_delay  - delay between LED RGB segments
34  // - strip_delay    - delay between LED strip update
35  //
36  //Notes:
37  // - SPI must be configured for Most significant bit (MSB) first
38  // - LED brightness is seven bits.  Most significant bit (MSB)
39  //    must be set to logic one
40  // - Each LED requires a separate R-G-B intensity component.  The order
41  //    of data is G-R-B.
42  // - After sending data for all strip LEDs.  A byte of (0x00) must
43  //    be sent to return strip to first LED.
44  // - Data stream for each LED is:
45  //1-G6-G5-G4-G3-G2-G1-G0-1-R6-R5-R4-R3-R2-R1-R0-1-B6-B5-B4-B3-B2-B1-B0
46  //
47  //Note:  Before executing the sample code insure the appropriate
48  //       device tree overlays have been loaded.
49  //
50  //*********************************************************************
51
52  //include files
53  #include <stdio.h>            //input and output operations
54  #include <fcntl.h>            //function operations
55  #include <unistd.h>           //symbolic constants and types
56  #include <sys/ioctl.h>        //input and output control
57  #include <stdint.h>           //defines integral type aliases
58  #include <linux/spi/spidev.h>//SPI support
59
60  #define SPI_PATH    "/dev/spidev1.0" //define path to SPI files
61  #define LED_strip_latch   0x00
62
63  //function prototypes
64  void clear_strip(void);
65  int   SPI_transfer(unsigned char);
66
67  //SPI file variables
68  unsigned int fd;
69  unsigned char null = 0x00;
70
71  //LED strip variables
72  unsigned char strip_length = 32;    //number of RGB LEDs in strip
73  unsigned char segment_delay = 1;    //delay in seconds
74  unsigned char strip_delay =    1;   //delay in seconds
75  unsigned char    LED_brightness;    //0 to 127
76  unsigned char    position;          //LED position in strip
77
78                                      //SPI configuration
79                                      //define transfer structure
80  int transfer(int fd, unsigned char send[], unsigned char receive[], int length)
81    {
82    struct spi_ioc_transfer transfer;          //transfer structure
83    transfer.tx_buf = (unsigned long) send;    //sending data buffer
84    transfer.rx_buf = (unsigned long) receive;//receiving data buffer
85    transfer.len = length;                     //buffer length
86    transfer.speed_hz = 125000;                //speed [Hz]
87    transfer.bits_per_word = 8;                //bits per word
88    transfer.delay_usecs = 0;                  //delay [us]
89
90  //send SPI message (fields, buffers)
91    int status = ioctl(fd, SPI_IOC_MESSAGE(1), &transfer);
92    if (status < 0)
93      {
94      perror("SPI: SPI_IOC_MESSAGE Failed");
95      return -1;
96      }
97    return status;
98    }
99
100 //main function
101 int main()
102   {
103   unsigned int i=0;                   //file handle, loop counter
104   unsigned char value;                //sending only a single char
105   uint8_t bits = 8, mode = 3;         //8-bits per word, SPI mode 3
```

```
106     uint32_t speed = 125000;              //Speed is 1 MHz
107
108                                           // Set up SPI properties
109     if ((fd = open(SPI_PATH, O_RDWR))<0)
110        {
111        perror("SPI Error: Can't open device.");
112        return -1;
113        }
114
115   if (ioctl(fd, SPI_IOC_WR_MODE, &mode)==-1)
116      {
117      perror("SPI: Can't set SPI mode.");
118      return -1;
119      }
120
121   if (ioctl(fd, SPI_IOC_RD_MODE, &mode)==-1)
122      {
123      perror("SPI: Can't get SPI mode.");
124      return -1;
125      }
126
127   if (ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits)==-1)
128     {
129      perror("SPI: Can't set bits per word.");
130      return -1;
131     }
132
133    if (ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits)==-1)
134      {
135      perror("SPI: Can't get bits per word.");
136      return -1;
137      }
138
139   if (ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed)==-1)
140     {
141     perror("SPI: Can't set max speed HZ");
142     return -1;
143     }
144
145     if (ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed)==-1)
146        {
147        perror("SPI: Can't get max speed HZ.");
148        return -1;
149       }
150
151                                        //Verify SPI properties
152     printf("SPI Mode is: %d\n", mode);
153     printf("SPI Bits is: %d\n", bits);
154     printf("SPI Speed is: %d\n", speed);
155     printf("Counting through all of the LEDs:\n");
156
157
158    SPI_transfer(LED_strip_latch);     //reset LED strip to first segment
159    clear_strip();                     //all strip LEDs to black
160    sleep(1);
161
162                                       //increment green intensity of strip LEDs
163    for(LED_brightness = 0; LED_brightness <= 60; LED_brightness = LED_brightness + 10)
164      {
165      for(position = 0; position<strip_length; position = position+1)
166        {
167        SPI_transfer(0x80 | LED_brightness);  //Green - MSB 1
168        SPI_transfer(0x80 | 0x00);            //Red   - none
169        SPI_transfer(0x80 | 0x00);            //Blue  - none
170        sleep(segment_delay);
171        }
172      SPI_transfer(LED_strip_latch);          //reset to first segment
173      sleep(strip_delay);
174      }
175     clear_strip();                           //all strip LEDs to black
176     sleep(1);
177
178                                              //increment red intensity of strip LEDs
179    for(LED_brightness = 0; LED_brightness <= 60; LED_brightness = LED_brightness + 10)
180      {
181      for(position = 0; position<strip_length; position = position+1)
182        {
183        SPI_transfer(0x80 | 0x00);            //Green - none
184        SPI_transfer(0x80 | LED_brightness);  //Red   - MSB1
185        SPI_transfer(0x80 | 0x00);            //Blue  - none
186        sleep(segment_delay);
187        }
188      SPI_transfer(LED_strip_latch);          //reset to first segment
189      sleep(strip_delay);
190      }
191    clear_strip();                           //all strip LEDs to black
192    sleep(1);
193
```

```
194                                          //increment blue intensity of strip LEDs
195   for(LED_brightness = 0; LED_brightness <= 60; LED_brightness = LED_brightness + 10)
196     {
197     for(position = 0; position<strip_length; position = position+1)
198       {
199       SPI_transfer(0x80 | 0x00);             //Green - none
200       SPI_transfer(0x80 | 0x00);             //Red   - none
201       SPI_transfer(0x80 | LED_brightness);   //Blue  - MSB1
202       sleep(segment_delay);
203       }
204     SPI_transfer(LED_strip_latch);      //reset to first segment
205     sleep(strip_delay);
206    }
207   clear_strip();                      //all strip LEDs to black
208   sleep(1);
209  }
210
211 //*************************************************************
212
213 void clear_strip(void)
214 {
215   //clear strip
216   for(position = 0; position<strip_length; position = position+1)
217       {
218       SPI_transfer(0x80 | 0x00);             //Green - none
219       SPI_transfer(0x80 | 0x00);             //Red - none
220       SPI_transfer(0x80 | 0x00);             //Blue  - none
221        }
222   SPI_transfer(LED_strip_latch);             //Latch with zero
223   sleep(1);                                  //clear delay
224 }
225
226 //*************************************************************
227 //SPI_transfer:  transmits a single byte of data
228 //*************************************************************
229
230 int SPI_transfer(unsigned char  SPI_data)
231   {
232   if (transfer(fd, (unsigned char*) &SPI_data, &null, 1)==-1)
233      {
234      perror("Failed to update the display");
235      return -1;
236       }
237   }
238 //*************************************************************
```

6.11 PRECISION TIMING

Processors may be used to accomplish time related tasks including generating precision digital signals, generating pulse widths of a specific duration or measuring the parameters of an incoming signal. Also, pulse width modulation (PWM) techniques may be used to vary the speed of a motor or control specialized motors such as a servo motor. In this section we describe some of the timing features available on BeagleBone. We begin with a review of timing related terminology followed by a review of some of the BeagleBone timing features. We conclude with several timing related examples. In the next section we discuss BeagleBone's PWM features.

6.11.1 TIMING-RELATED TERMINOLOGY

In this section, we review timing related terminology including frequency, period, and duty cycle.

Frequency: Consider a signal $x(t)$ that repeats itself. We call this signal periodic with period T, if it satisfies

$$x(t) = x(t + T).$$

To measure the frequency of a periodic signal, we count the number of times a particular event repeats within a one second period. The unit of frequency is Hertz or cycles per second. For

example, a sinusoidal signal with a 60 Hz frequency means that a full cycle of a sinusoid signal repeats itself 60 times each second or every 16.67 ms. The period of the signal is 16.67 ms.

Period: The reciprocal of frequency is the period of a waveform. If an event occurs with a rate of 1 Hz, the period of that event is 1 s. To find a period, given the frequency of a signal, or vice versa, we simply need to remember their inverse relationship $f = \frac{1}{T}$ where f and T represent a frequency and the corresponding period, respectively.

Duty Cycle: In many applications, periodic pulses are used as control signals. A good example is the use of a periodic pulse to control a servo motor. To control the direction and sometimes the speed of a motor, a periodic pulse signal with a changing duty cycle over time is used. The periodic pulse signal shown in Figure 6.27a is on for 50% of the signal period and off for the rest of the period. The pulse shown in Figure 6.27 b is on for only 25% of the same period as the signal in Figure 6.27a and off for 75% of the period. The duty cycle is defined as the percentage of the period a signal is on or logic high. Therefore, we call the signal in Figure 6.27a as a periodic pulse signal with a 50% duty cycle and the corresponding signal in Figure 6.27b, a periodic pulse signal with a 25% duty cycle. These features are discussed in more detail in the PWM section.

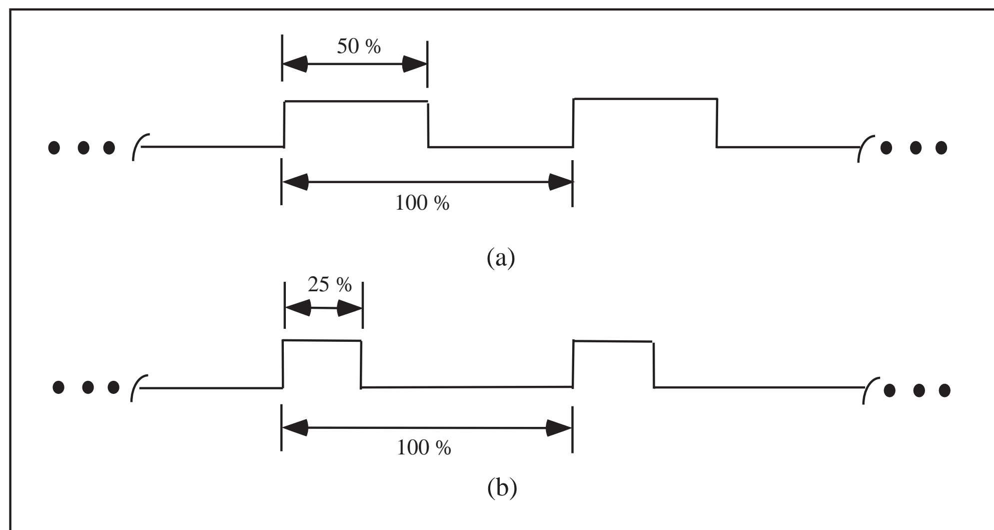


Figure 6.27: Two signals with the same period but different duty cycles. The top figure (a) shows a periodic signal with a 50% duty cycle (the signal is logic high for 50% of the total period) and the lower figure (b) displays a periodic signal with a 25% duty cycle [Barrett and Pack, 2006].

6.11.2 BEAGLEBONE TIMING CAPABILITY

Depending on version and power supply source, BeagleBone is clocked from 500 to 1 GHz. This allows BeagleBone to measure the characteristics of high-frequency input signals or generate high-frequency digital signals. We limit our discussion to the timing functions available in C and also the Linux system.

Earlier in the chapter we provided an example to blink an LED at 5 s intervals (led3.c). In this example we used the timing features available within the C programming language. The “time” function in ANSI C returns the current calendar time in seconds that have elapsed since January 1, 1970. Time hacks may be taken at different times for use in delay functions or to measure intervals with the resolution of seconds. The ANSI C library also provides a “difftime” function that provides the difference in time between two time hacks. The difference is in seconds as a double type variable [Kelley and Pohl, 1998]. The <time.h> header file must be included to use these features.

To achieve better time resolution the Linux “gettimeofday” function may be used. The function returns the current time in seconds and microseconds in a timeval structure since January 1, 1970.

```
1 struct timeval{
2   time_t          tv_sec;      //seconds
3   susecondS_t     tv_usec;     //microseconds
4 };
```

To use these features the <sys/time.h> header file must be included.

Example: In the following example, the “gettimeofday” function is used to generate a 100 Hz, 50% duty cycle signal on header P8, pin 12 (GPIO1_12 designated as gpio44). The signal may be easily changed to 100 kHz signal simply by changing the argument of the “delay_us” from 5000 to 5.

```
1 //*****************************************************************
2 //sq_wave.c:  generates a 100 Hz, 50% duty cycle signal on header
3 //P8, pin 12 (GPIO1_12 designated as gpio44).
4 //*****************************************************************
5
6 #include <stdio.h>
7 #include <stddef.h>
8 #include <time.h>
9 #include <sys/time.h>
10
11 #define  OUTPUT  "out"
12 #define  INPUT   "in"
13
14 void delay_us(int);
15
```

```
16 int  main (void)
17 {
18 //define file handles
19 FILE *ofp_export, *ofp_gpio44_value, *ofp_gpio44_direction;
20
21 //define pin variables
22 int pin_number = 44, logic_status = 1;
23 char* pin_direction = OUTPUT;
24
25 ofp_export = fopen("/sys/class/gpio/export", "w");
26 if(ofp_export == NULL) {printf("Unable to open export.\n");}
27 fseek(ofp_export, 0, SEEK_SET);
28 fprintf(ofp_export, "%d", pin_number);
29 fflush(ofp_export);
30
31 ofp_gpio44_direction = fopen("/sys/class/gpio/gpio44/direction", "w");
32 if(ofp_gpio44_direction == NULL) {printf("Unable to open
      gpio44_direction.\n");}
33 fseek(ofp_gpio44_direction, 0, SEEK_SET);
34 fprintf(ofp_gpio44_direction, "%s",  pin_direction);
35 fflush(ofp_gpio44_direction);
36
37 ofp_gpio44_value = fopen("/sys/class/gpio/gpio44/value", "w");
38 if(ofp_gpio44_value == NULL) {printf("Unable to open gpio44_value.\n")
      ;}
39 fseek(ofp_gpio44_value, 0, SEEK_SET);
40 logic_status = 1;
41 fprintf(ofp_gpio44_value, "%d", logic_status);
42 fflush(ofp_gpio44_value);
43
44
45 while(1)
46   {
47   delay_us(5000);
48   if(logic_status == 1) logic_status = 0;
49     else logic_status = 1;
50   //write to gpio44
51   fprintf(ofp_gpio44_value, "%d", logic_status);
52   fflush(ofp_gpio44_value);
53   }
54 fclose(ofp_export);
55 fclose(ofp_gpio44_direction);
56 fclose(ofp_gpio44_value);
57 return 1;
58 }
59
60 //*****************************************************************
61
```

```
62 void delay_us(int desired_delay_us)
63 {
64 struct timeval  tv_start;   //start time hack
65 struct timeval  tv_now;     //current time hack
66 int elapsed_time_us;
67
68 gettimeofday(&tv_start, NULL);
69 elapsed_time_us = 0;
70
71 while(elapsed_time_us <  desired_delay_us)
72   {
73   gettimeofday(&tv_now, NULL);
74   if(tv_now.tv_usec >= tv_start.tv_usec)
75     elapsed_time_us = tv_now.tv_usec - tv_start.tv_usec;
76   else
77     elapsed_time_us = (1000000 - tv_start.tv_usec) + tv_now.tv_usec;
78   //printf("start: %ld \n", tv_start.tv_usec);
79   //printf("now: %ld \n", tv_now.tv_usec);
80   //printf("desired: %d \n", desired_delay_ms);
81   //printf("elapsed: %d \n\n", elapsed_time_ms);
82   }
83 }
84
85 //*****************************************************************
```

6.12 PULSE WIDTH MODULATION (PWM)

In this section, we discuss a method to control the speed of a DC motor using a pulse width modulated (PWM) signal. If we turn on a DC motor and provide the required voltage, the motor will run at its maximum speed. Suppose we turn the motor on and off rapidly, by applying a periodic signal. The motor at some point can not react fast enough to the changes of the voltage values and will run at the speed proportional to the average time the motor was turned on. Similarly, by changing the duty cycle of the periodic signal, we can control the speed of a DC motor. Suppose again we want to generate a speed profile shown in Figure 6.28. As shown in the figure, we want to accelerate the speed, maintain the speed, and decelerate the speed for a fixed amount of time.

As an example, for passenger comfort, an elevator control system does not immediately operate the elevator motor at full speed. The elevator motor speed will ramp up gradually from stop to desired speed. As the elevator approaches the desired floor, it will gradually ramp back down to stop.

Earlier in this chapter we discussed the signal parameters of frequency and duty cycle. A PWM signal maintains a constant baseline frequency. The duty cycle of the signal is varied as required by the specific application. Also, the polarity of the signal may be active high or active low.

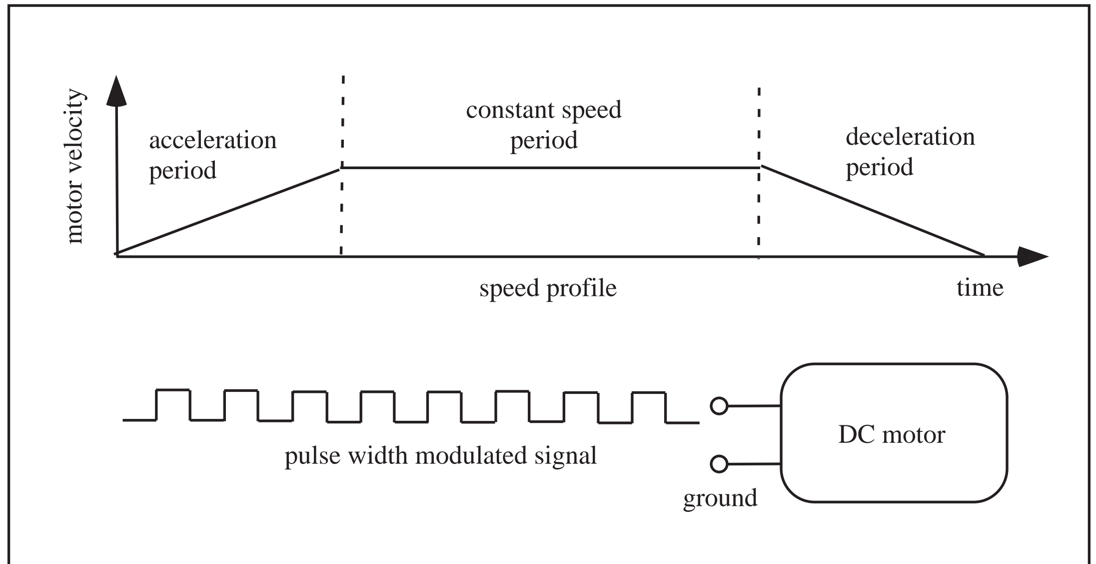


Figure 6.28: Speed profile of a DC motor over time with a PWM signal applied [Barrett and Pack, 2006].

Pin Number	Pin Name	Export Number
P9.22, P9.31	EHRPWM0A	0
P9.21, P9.29	EHRPWM0B	1
P9.42	ECAPPWM0	2
P9.14, P8.36	EHRPWM1A	3
P9.16, P8.34	EHRPWM1B	4
P8.19, P8.45	EHRPWM2A	5
P8.13, P8.46	EHRPWM2B	6
P9.28	ECAPPWM2	7

Figure 6.29: BeagleBone Black PWM features.

6.12.1 BEAGLEBONE PWM SUBSYSTEM (PWMSS) DESCRIPTION

The description provided here was adapted from the *AM335X PWMSS Driver's Guide*. BeagleBone is equipped the PWMSS system. It is subdivided into the:

- enhanced high resolution PWM (eHRPWM) system,
- enhanced Captured (eCAP) system, and

- enhanced Quadrature Encoded Pulse (eQEP) system.

Due to space limitations we only discuss the eHRPWM system in detail. However, a summary of BeagelBone Black features is provided in Figure 6.29. The eHRPWM system is supported by 16-bit timers for period and frequency. The eHRPWM system consists of two instances of two channels each. The instances and channels are designated as:

```
ehrpwm.i:j
```

where i is the instance and j (0 or 1) is the channel.

For example, in BeagleBone, EHRPWM1A (P9, pin 14) is designated ehrpwm.1:0 and EHRPWM1B (P9, pin 16) is designated ehrpwm.1:1.

To configure and use the PWM system a four-step process is followed.

1. Configure the PWM pin for output.
2. Request the PWM device.
3. Configure the PWM device.
4. Start (and Stop) the PWM device.

The steps are accomplished using the BeagleBone Linux file system. We provide the Linux commands to accomplish each step in an upcoming section.

6.12.2 BONESCRIPT PWM SUPPORT

A PWM signal may be generated via Bonescript using the analogWrite function. In an earlier example we used the analogWrite function to modulate the intensity of an LED. The "analogWrite" function generates a 1 kHz pulse width modulated signal at the specified pin. The duty cycle (the percentage of time the 1 kHz signal is a logic high within a period) is set using a value from 0–1. In the next chapter we provide a laser light show example using the PWM features of Bonescript.

6.12.3 PWM DEVICE TREE OVERLAY AND C SUPPORT FUNCTIONS

This section is based on information provided in "Working with PWM on a BeagleBone Black [http://briancode.wordpress.com]." As in previous examples, the appropriate device tree overlay must first be loaded and then related files configured. We follow the same approach here.

Begin by loading the universal cape using the command [Steinkuehler]:

```
#echo cape-universaln > /sys/devices/bone_capemgr.*/slots
```

Once the overlay is loaded, accompanying support files related to the overlay may be examined at: "/sys/devices/ocp.*" The following command sequence may be used [Molloy, 2015]:

```
#cd /sys/devices/ocp.*
/sys/devices/ocp.3# ls
```

The "config-pin" utility allows the user to determine the state of a specific BeagleBone Black pin or change its multiplexer configuration. To obtain a list of pin settings for the loaded universal cape, the following command may be used [Steinkuehler]:

```
#config-pin -l
```

The configuration options for a specific pin may be examined using [Steinkuehler]:

```
#config-pin -l <pin>
```

For this PWM example, we examine P9.14, EHRPWM1A, export number 3 (from Figure 6.29).

```
#config-pin -l P9.14
default  gpio  gpio_pu  gpio_pd  pwm
```

The pin is configured for PWM operation using:

```
#config-pin P9.14 pwm
```

By exporting "3" to the pwm export file, the support files for the PWM channel are established in directory $/sys/class/pwm/pwm3$. This is accomplished using:

```
# echo 3 > /sys/class/pwm/export
# cd /sys/class/pwm/pwm3
device  duty_ns  period_ns  polarity  power  run  subsystem  uevent
```

PWM parameters (duty cycle and period) may then be loaded to appropriate files in units of nanoseconds.

Example: To set a 10 kHz frequency and a 20% duty cycle, load the following constants:

```
#cd /sys/class/pwm/pwm3/
/sys/class/pwm/pwm3/# echo 20000 > duty_ns
/sys/class/pwm/pwm3/# echo 100000 > period_ns
/sys/class/pwm/pwm3/# echo 0 > polarity
/sys/class/pwm/pwm3/# echo 1 > run
```

The PWM signal may be stopped using:

```
/sys/class/pwm/pwm3/# echo 0 > run
```

Example: The following example uses BeagleBone Black PWM features to provide a "fade" effect on PWM pin P9, pin 14.

As a friendly reminder, before executing the sample code insure the appropriate device tree overlays have been loaded.

```
1 //*********************************************************************
2 //pwm.c
3 //
4 //This uses PWM to output a "fade" effect to P9_14
5 //
6 //Note: Before executing the sample code insure the appropriate
7 //device tree overlays have been loaded.
8 //*********************************************************************
9
10 #include <stdio.h>
11 #include <stddef.h>
12 #include <time.h>
13
14 #define OUTPUT "out"
15 #define INPUT  "in"
16
17 int main (void)
18 {
19 //define file handles
20 FILE *pwm_period, *pwm_duty, *pwm_polarity, *pwm_run;
21
22 //define pin variables
23 int period = 500000, duty = 250000, polarity = 1, run = 1;
24 int increment = 1;
25
26 pwm_period = fopen("/sys/class/pwm/pwm3/period_ns", "w");
27 if(pwm_period == NULL) {printf("Unable to open pwm period.\n");}
28 fseek(pwm_period, 0, SEEK_SET);
29 fprintf(pwm_period, "%d", period);
30 fflush(pwm_period);
31
32 pwm_duty = fopen("/sys/class/pwm/pwm3/duty_ns", "w");
33 if(pwm_duty == NULL) {printf("Unable to open pwm duty cycle.\n");}
34 fseek(pwm_duty, 0, SEEK_SET);
35 fprintf(pwm_duty, "%d", duty);
36 fflush(pwm_duty);
37
38 pwm_polarity = fopen("/sys/class/pwm/pwm3/polarity", "w");
39 if(pwm_polarity == NULL) {printf("Unable to open pwm polarity.\n");}
40 fseek(pwm_polarity, 0, SEEK_SET);
41 fprintf(pwm_polarity, "%d", polarity);
42 fflush(pwm_polarity);
43
44 pwm_run = fopen("/sys/class/pwm/pwm3/run", "w");
45 if(pwm_run == NULL) {printf("Unable to open pwm run.\n");}
46 fseek(pwm_run, 0, SEEK_SET);
47 fprintf(pwm_run, "%d", run);
48 fflush(pwm_run);
```

```
49
50 while(1)
51   {
52 if(duty >= period)
53    {
54   increment = -1;          //set to decrement when duty reaches period
55    }
56 else if(duty <= 0)
57    {
58    increment = 1;          //set to increment when duty reaches 0
59    }
60 duty += increment;         //update duty cycle
61 fseek(pwm_duty, 0, SEEK_SET);
62 fprintf(pwm_duty, "%d", duty);
63 fflush(pwm_duty);
64   }
65
66 fclose(pwm_period);
67 fclose(pwm_duty);
68 fclose(pwm_polarity);
69 fclose(pwm_run);
70
71 return 1;
72 }
73 //
     ****************************************************************************
```

6.13 INTERNET OF THINGS—NETWORKING

A hot new concept in the microcontroller world is the Internet of Things (IoT). IoT describes a world in which embedded computing devices are part of the Internet structure. This allows for embedded computer controlled systems to be monitored and controlled via the existing Internet structure. This section describes the multiple capabilities of BeagleBone to be connected within small (I2C and CAN), medium (LAN), and large (Internet) capabilities. Specifically, BeagleBone is equipped with several networking systems the I2C for small area (circuit board) networking, the Controller Area Network (CAN) for system level networking the Ethernet 10/100 PHY for local area networks (LAN), and Internet capability. We discuss each system in turn.

6.13.1 INTER-INTEGRATED CIRCUIT (I2C) BUS

The Inter-IC bus (I2C) provides a method to interconnect multiple system components together residing in a small, circuit board size area. The I2C system is also referred to as the IIC bus or the two-wire interface (TWI). The I2C system consists of a two wire 100 k bps (bit per second) bus. The 100 k bps bus speed is termed the standard mode but the bus may also operate at higher data

rates. There are multiple I2C compatible peripheral components (e.g., LCD displays, sensors, etc.) [I2C, 2000].

A large number of devices (termed nodes) may be connected to the I2C bus. The I2C system uses a standard protocol to allow the nodes to send and receive data from the other devices. All nodes on the bus are assigned a unique 7-bit address. The eighth bit of the address register is used to specify the operation to be performed (read or write). Additional devices may be added to the I2C based system as it evolves [I2C, 2000].

The basic I2C bus architecture is shown in Figure 6.30. The two wire bus consists of the serial clock line (SCL) and the serial data line (SDA). These lines are pulled up to logic high by the SCL and the SDA pull up resistors. Nodes connected to the bus can drive either of the bus lines to ground (logic 0). Devices within an I2C bus configuration must share a common ground [I2C, 2000].

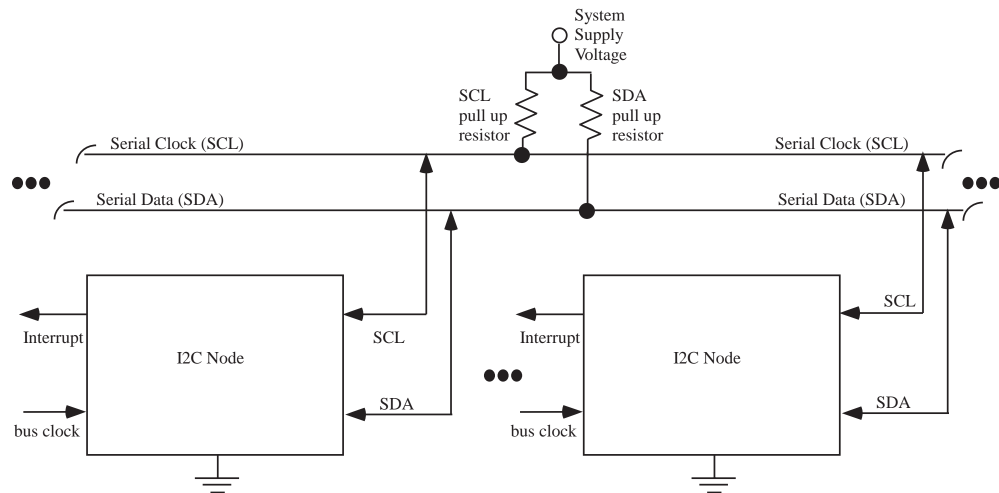


Figure 6.30: I2C configuration.

BeagleBone I2C Subsystem Description

BeagleBone is equipped with two I2C channels designated I2C1 and I2C2. The I2C2 channel is dedicated for EEPROM use and should not be tampered with. The I2C1 channel is available at P9 pin 17 for I2C1_SCL (Mode 2) and P9 pin 18 for I2C1_SDA (Mode 2). There is considerable additional information on this topic and C support functions available through the Linux Documentation Project [Coley, 2014].

I2C Bonescript Support

Earlier in the book we mentioned Bonescript is a rapidly evolving, open source development platform. Bonescript consists of a JavaScript library of functions to rapidly develop a variety of physical computing applications. It will grow in capability as other users develop additional features to enhance Bonescript. In that light, Bonescript functions to support I2C communications are available at: `https://github.com/korevec/node-i2c`. Functions supporting I2C communications include:

- i2cOpen(port, address, options, [callback]),
- i2cScan(port, [callback]),
- i2cWriteByte(port, byte, [callback]),
- i2cWriteBytes(port, command, bytes, [callback]),
- i2cReadByte(port, [callback]),
- i2cReadBytes(port, command, length, [callback]), and
- i2cStream(port, command, length, [callback]).

6.13.2 CONTROLLER AREA NETWORK (CAN) BUS

The Controller Area Network or CAN bus was originally developed for the automotive industry in the 1980's. There are two different CAN protocols: A the basic or standard version and B the extended or full version. The CAN protocol allows a number of processors or nodes to be linked via a twisted pair cable. The nodes may exchange data serially at up to 1 Mbit/s data rates. Each node on the CAN bus can serve as the master and can send or receive data messages over the bus [COMSOL, 2015].

The CAN message format consists of four different types of frames: data, remote, error and overload. The data frame is shown in Figure 6.31. Each frame consists of a series of fields. Embedded in the message is an identifier. For an incoming frame, each node will examine the identifier to determine if the frame is intended for the node. This allows a message to be sent to a specific node or group of nodes [COMSOL, 2015].

BeagleBone CAN Subsystem Description

BeagleBone may be equipped with CAN features using either the TT3201 CAN Cape or the BeagleBone CAN Bus Cape. Each Cape is supported with full documentation and software support [Coley, 2014; CircuitCo, 2015].

6.13.3 ETHERNET

Earlier in this chapter we discussed methods of connecting BeagleBone Black to a host computer (desktop or laptop) via a mini-USB cable. This is a good method of using the host's keyboard

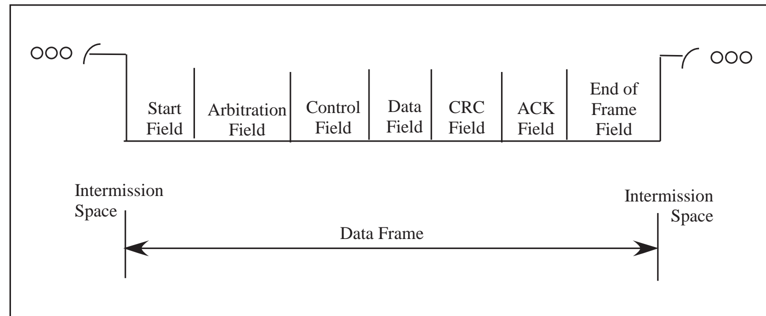


Figure 6.31: CAN data frame [COMSOL, 2015].

and monitor. The connection length between the host and the BeagleBone Black is limited to 5 m (16.4 ft). If the BeagleBone Black is going to operate in a remote application (e.g., security system, greenhouse, weather station, security, and environment of a remote out building, etc.), the Ethernet may be used. The maximum length of an Ethernet cable is 100 m (328 ft).

BeagleBone is equipped with 10/100 Ethernet capability via the SMSC LAN8710A integrated circuit to provide local area network (LAN) capability. This is the common LAN protocol used in many commercial and home networks. The 10/100 refers to 10 Mbps and 100 Mbps data rates.

The LAN8710A chip implements the Media Independent Interface (MII) physical layer (PHY) of the Open Systems Interconnection (OSI) model. The PHY implements the hardware send and receive protocol by breaking a serial data stream into frames. The PHY is configured for auto negotiation which allows two connected devices to choose common transmission parameters [Coley, 2014; SMSC, 2012].

To establish an Ethernet connection between a host Windows-based computer and BeagleBone the following steps may be used [Richardson, 2014].

- Connect your Internet router to BeagleBone Black via an Ethernet cable as shown in Figure 6.32.
- Insure BeagleBone Black is being supplied by a 5 VDC, 2 A power supply (Adafruit #276).
- Download PuTTY. It is a free and open-source terminal emulator available for download at `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`.
- Once PuTTY is downloaded and launched, enter “beaglebone.local” for the host address and choose an “SSH” type connection.
- The SSH application establishes a connection between the host computer and BeagleBone Black.

- When prompted, respond with “root” and hit [Enter].
- When prompted for password, hit [Enter].
- The BeagleBone may now be controlled remotely from the host computer.

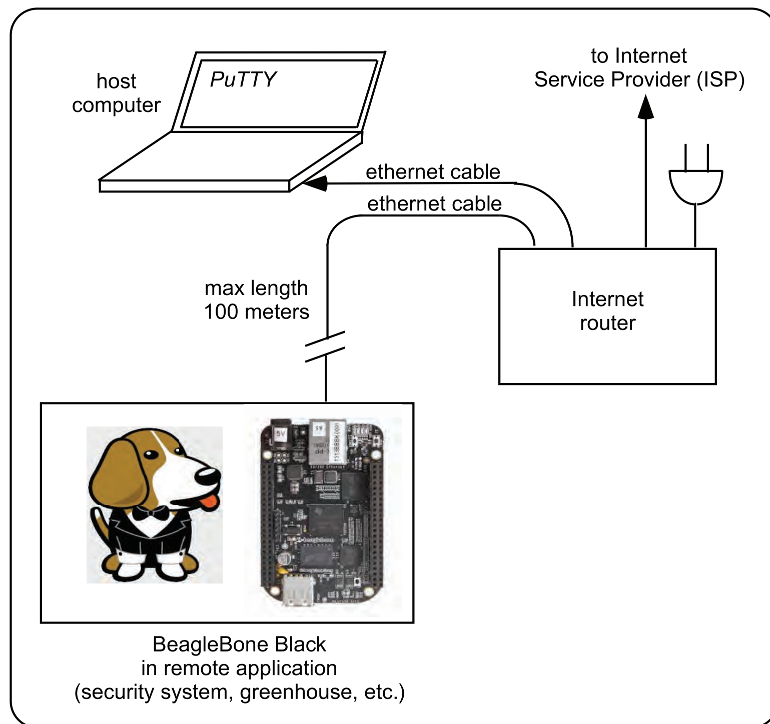


Figure 6.32: BeagleBone ethernet connection. (Illustrations used with permission of Texas Instruments (www.TI.com).)

A Bonescript application can be launched remotely from the Linux command line. Insure you are in the demo directory (where the program resides) using the following command:

```
# /var/lib/cloud9/demo/
```

Once in the demo directory, the program is launched using:

```
# node <filename>.js
```

6.13.4 INTERNET

In Chapter 7 we provide several methods of assembling a stand-alone BeagleBone Black computer using a keyboard, mouse, HDMI compatible display, and a USB hub. Once the computer is

assembled, connect your Internet router to BeagleBone Black via an Ethernet cable. The Internet may now be accessed from the Linux desktop.

6.14 LIQUID CRYSTAL DISPLAY (LCD) INTERFACE

BeagleBone provides full support for both 24- and 16-bit LCD interfaces. A 7-in LCD Cape is available from Circuitco. The TFT LCD has 800 by 480 pixel resolution and also supports touch screen features. The Cape provides an easy to use 39-pin interface for the large LCD display. Also, Linux Ångström distribution images after 6.18.12 directly support the Cape. Full support data for the LCD is provided in *BeagleBone LCD7 Cape Rev A3 System Reference Manual*.

If an application can not support use of the 39-pin Cape interface and other desired subsystems, a smaller footprint LCD may be employed. Serial LCDs employing a UART or SPI interface are readily available. In the following example, we equip BeagleBone with a two line, 16-character display employing a parallel interface.

Example: In this example we equip BeagleBone with the Sparkfun LCD-09052 16 by 2-character LCD. This is a 3.3 VDC LCD with White on black characters. The interface between BeagleBone and the LCD is provided in Figure 6.33.

6.14.1 C SUPPORT FUNCTIONS

The C code for the LCD interface and support functions are provided in Appendix B.

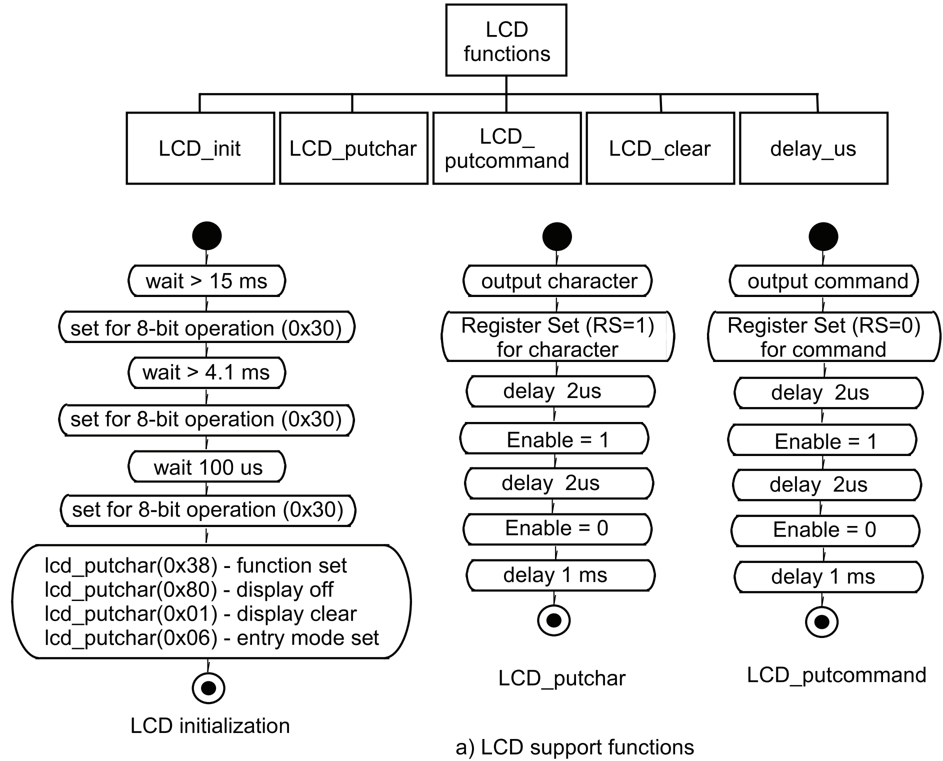
6.15 INTERRUPTS

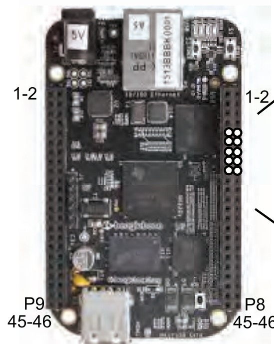
A processor normally executes instructions in an orderly fetch-decode-execute sequence as dictated by a user-written program as shown in Figure 6.34. However, the processor must be equipped to handle unscheduled (although planned), higher priority events that might occur inside or outside the processor. To process such events, a processor requires an interrupt system.

The interrupt system onboard a processor allows it to respond to higher priority events. Appropriate responses to these events may be planned, but we do not know when these events will occur. When an interrupt event occurs, the processor will normally complete the instruction it is currently executing and then transition program control to interrupt event specific tasks. These tasks, which resolve the interrupt event, are organized into a function called an interrupt service routine (ISR). Each interrupt will normally have its own interrupt specific ISR. Once the ISR is complete, the processor will resume processing where it left off before the interrupt event occurred.

6.15.1 BONESCRIPT INTERRUPT SUPPORT

Bonescript provides an “attachInterrupt” function to support interrupt processing on Beaglebone. The function associates an interrupt event with the desired actions to be accomplished when the event occurs. The function has three arguments: the pin to monitor for the interrupt event, the





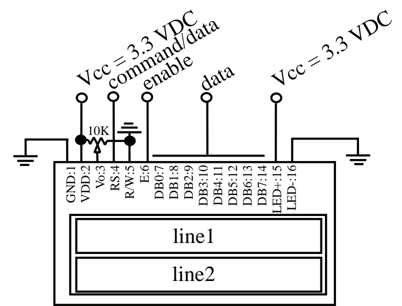


Figure 6.33: BeagleBone LCD interface. (Illustrations used with permission of Texas Instruments (www.TI.com).)

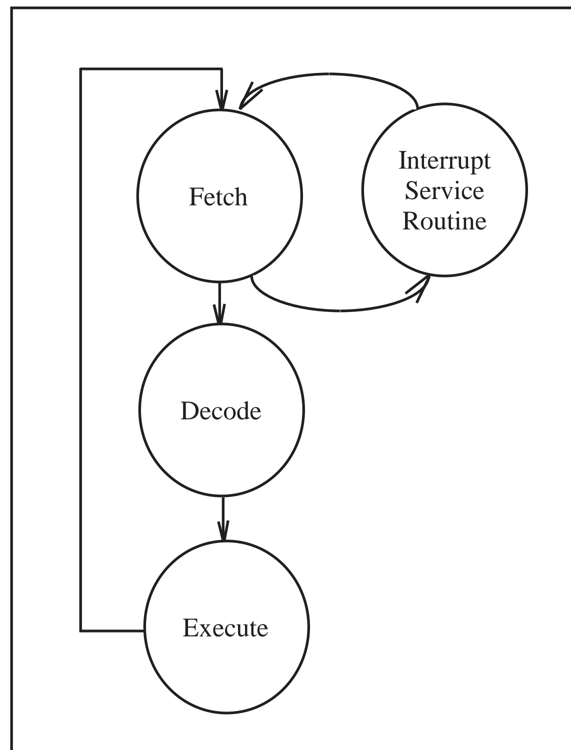


Figure 6.34: Processor interrupt response.

type of pin activity to initiate the interrupt, and the name of the interrupt service routine (ISR) to be executed when the interrupt event occurs.

Example: In this example a tactile switch is connected to P8, pin 15. When P8, pin 15 experiences a rising edge the ISR is executed. In the main loop of the program a green LED is blinking at 100 ms intervals. When the switch is depressed and released, creating a rising edge on P8, pin 15 the ISR is executed which blinks a red LED at 50 ms intervals. A circuit diagram for this example is provided in Figure 6.35.

```
1 //*************************************************************
2 var b = require('bonescript');
3
4 setup = function() {};
5
6 var greenLED = ''P8_13'';
7 var redLED = ''P9_14'';
8 var inputSW = ''P8_15'';
9 var main_delay = 100;
```

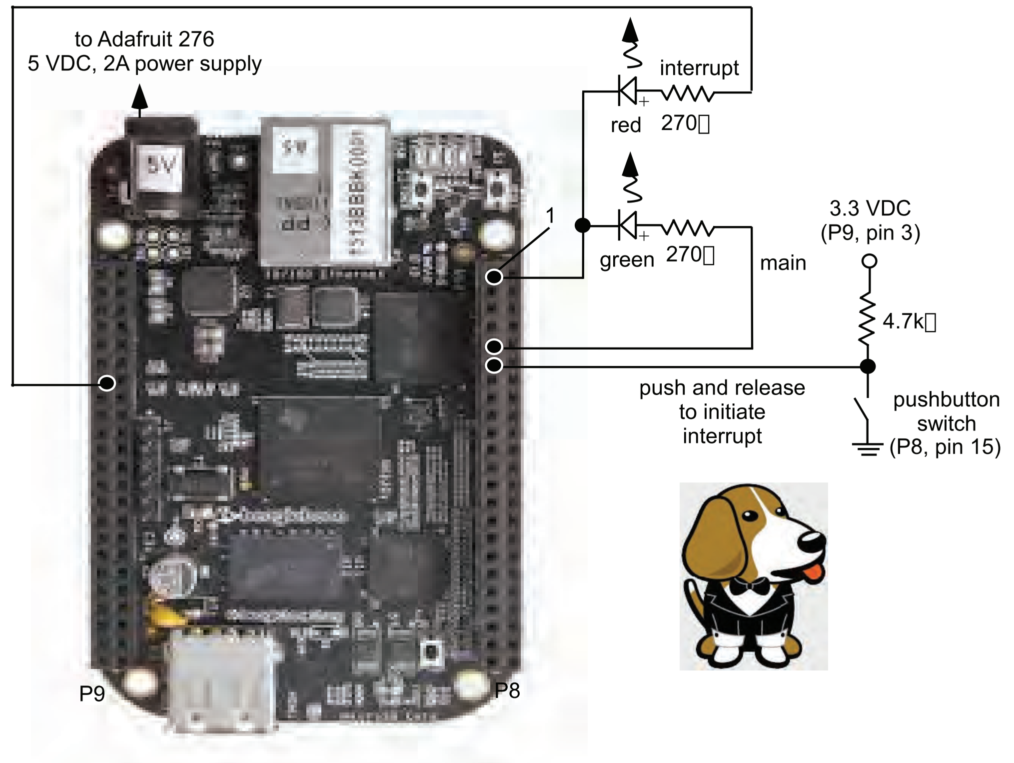


Figure 6.35: Bonescript interrupt example.In the main program loop a green LED is blinking at 100 ms intervals. When the switch is depressed and released, creating a rising edge on P8, pin 15 the ISR is executed which blinks a red LED at 50 ms intervals. (Illustrations used with permission of Texas Instruments (www.TI.com).)

```
10 var inter_delay = 50;
11 var state = b.LOW;
12
13 b.pinMode(greenLED, b.OUTPUT);
14 b.pinMode(redLED, b.OUTPUT);
15 b.pinMode(inputSW, b.INPUT);
16 b.attachInterrupt(inputSW, RISING, blink_red);
17
18 function loop()
19     {
20     state = (state == b.HIGH) ? b.LOW : b.HIGH;
21     b.digitalWrite(greenLED, state);
22     setTimeout(loop, main_delay);
23     }
```

```
24
25 function blink_red()
26 {
27 b.digitalWrite(redLED, b.HIGH);
28 setTimeout(blink_red2, inter_delay);
29 }
30
31 function blink_red2()
32 {
33 b.digitalWrite(redLED, b.LOW);
34 setTimeout(blink_red3, inter_delay);
35 }
36
37 function blink_red3()
38 {
39 b.digitalWrite(redLED, b.HIGH);
40 setTimeout(blink_red4, inter_delay);
41 }
42
43 function blink_red4()
44 {
45 b.digitalWrite(redLED, b.LOW);
46 }
47 //***********************************************************
```

6.16 PROGRAMMABLE REAL-TIME UNITS

Imagine purchasing a high-performance truck with an attached trailer. Much to your surprise, inside the trailer are two high-performance sports cars included with the truck. Sounds exciting, perhaps far-fetched! We have a similar situation with BeagleBone Black.

BeagleBone Black is equipped with two Programmable Real-Time Units (PRUs). Formally, they are within the Programmable Real-Time Unit and Industrial Communication Subsystem the PRU-ICSS. The PRU-ICSS system is illustrated in Figure 6.36. The PRU-ICSS consists of two 32-bit Reduced Instruction Set Computing (RISC) PRUs. A RISC processor uses a collection of simple instructions to accomplish tasks. The PRUs operate at 200 MHz and are equipped with 8 Kbytes of Random Access Memory (RAM), a shared 12 Kbyte RAM, an enhanced general purpose input output (GPIO) unit, and an interrupt controller (INTC). The PRUs can access subsystems within the PRU-ICSS but may also interact with the host BeagleBone Black processor. The PRUs are not under the direct control of the Linux operating system and are therefore useful for real time tasks. Operating at 200 MHz, the PRUs complete a RISC instruction every 5 ns. This allows for fast, predictable, precise real-time control. These features are important to develop custom peripherals, interfaces, and signals [Texas Instruments, AM335X PRU–ICSS Reference Guide].

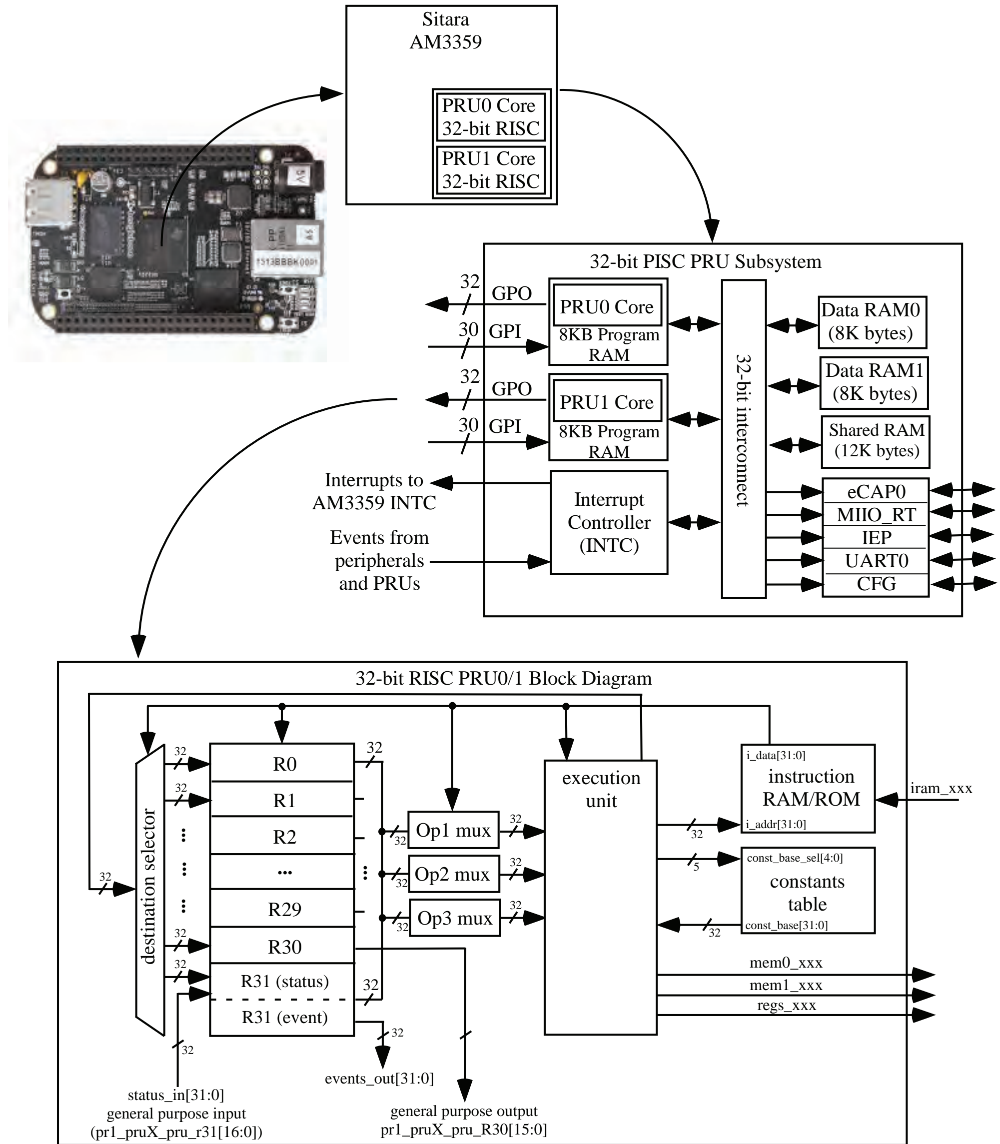


Figure 6.36: Programmable real-time unit and industrial communication subsystem the PRU-ICSS [PRU-ICSS Reference Guide, 2013].

In this section we provide the bare essential information needed to get started with the PRUs. We begin with an architecture overview, a PRU assembly overview, discuss PRU development tools, and conclude with an end-to-end programming example. Much of the information here is condensed from the following Texas Instruments resources. They are readily available online at [www.ti.com].

- *AM335X PRU-ICSS Reference Guide*
- *AM335X ARM-AX Microprocessors (MPUs) Technical Reference Manual*
- *PRU Assembly Reference Guide*
- *Programmable Real-time Unit Subsystem Training Material*
- *PRU Assembly Language Tools User's Guide*
- *AM335X Linux Application Loader User Guide*

6.16.1 ARCHITECTURE OVERVIEW

Taking a closer view of Figure 6.36, the BeagleBone Black hosts the Sitara AM3359 processor which hosts the two 32-bit RISC PRU cores designated PRU0 and PRU1. The PRUs are each equipped with 8 Kbytes of program RAM. Through the 32-bit interconnect, the PRUs have access to data RAMs designated Data RAM0 and Data RAM1, 12 Kbytes of shared RAM, an Interrupt Controller, general purpose input and output, and a number of special purpose input/output systems (eCAP0, MIIO_RT, IEP, UART0, and CFG).

The RISC architecture for each PRU is provided in the lower portion of Figure 6.36. Each PRU is equipped with a complement of 32, 32-bit registers designated R0 to R31. Register R30 provides for general purpose outputs (pr1_pruX_pru_R30[15:0]) and register R31 provides for status input and events output. Register R31 also provides for general purpose input (pr1_pruX_pru_R31[16:0]). The placeholder "X" is "0" for pru 0 and "1" for pru 1.

Register contents serve as input to the execution unit. Three separate operands may be fed to the execution unit simultaneously. The execution unit may also use inputs from the constants table. The execution unit performs operations on the operands as specified by the assembly language program. The operations may be arithmetic, logical, flow control, register operations, or input/output. The results of the operations may be fed back to a specific register or sent outside the PRU.

6.16.2 PRU MEMORY MAP

Different components within a specific PRU can be accessed by the PRU, the other PRU, or the host processor. Figure 6.37 provides the address (provided in hexadecimal (0x)) to these accessible PRU components. For each component the address is provided when accessed from PRU0, PRU1, and the BeagleBone Black processor, respectively.

PRU-ICSS Subsystem	PRU0 point of view	PRU1 point of view	BBB point of view
Data 8KB RAM0/1	RAM0:0x0000_0000	RAM1:0x0000_0000	RAM0:0x0000_0000
Data 8KB RAM1/0	RAM1:0x0000_2000	RAM0:0x0000_2000	RAM1:0x0000_2000
Data 12 KB RAM2 (shared)	0x0001_0000	0x0001_0000	0x0001_0000
INTC	0x0002_0000	0x0002_0000	0x0002_0000
PRU0 Control Registers	0x0002_2000	0x0002_2000	0x0002_2000
PRU0 Debug	---	---	0x0002_2400
PRU1 Control Registers	0x0002_4000	0x0002_4000	0x0002_4000
PRU1 Debug	---	---	0x0002_4400
CFG	0x0002_6000	0x0002_6000	0x0002_6000
UART0	0x0002_8000	0x0002_8000	0x0002_8000
IEP	0x0002_E000	0x0002_E000	0x0002_E000
eCAP 0	0x0003_0000	0x0003_0000	0x0003_0000
MII_RT_CFG	0x0003_2000	0x0003_2000	0x0003_2000
MII_MDIO	0x0003_2400	0x0003_2400	0x0003_2400
PRU0 8KB IRAM	---	---	0x0003_4000
PRU1 8KB IRAM	---	---	0x0003_8000
System OCP_HPx	HP0:0x0000_0000	HP1:0x0000_0000	HP1:0x0000_0000

Figure 6.37: PRU memory map [PRU-ICSS Reference Guide].

6.16.3 PRU INTERRUPT SYSTEM

The PRUs are equipped with a flexible Interrupt Controller (INTC) illustrated in Figure 6.38. Interrupts mapped to PRU0/1 may be generated from system peripheral components or from events within the PRU. These interrupts are collectively referred to as system events. The system events are mapped to channels 0–9 and then to host 0–9. Interrupt channel 0 has the highest priority. Host interrupts are then mapped to PRU register R31 bits 30 and 31.

6.16.4 PRU PIN MAPPING TO BEAGLEBONE BLACK

The PRU features may be accessed through the BeagleBone Black external pins. As described earlier in the book, each BeagleBone Black pin has multiple functions mapped to them. Internal multiplexers are used to determine which feature will be mapped to the pin in a specific application. Provided in Figure 6.39 is an abbreviated table of PRU general purpose input and output pins accessible from the P8 and P9 headers. Other PRU pins, not contained in the table, are also mapped to BeagleBone Black P8 and P9 headers. As described earlier in this chapter, default multiplexer settings may be changed by modifying the BeagleBone Black device tree.

6.16.5 PRU ASSEMBLY PROGRAM (PASM)

The PRUs are programmed in assembly language using the PRU Assembly Program (PASM). PASM is available for download from the Texas Instruments website. Assembly language provides a tight interface between software and the PRU hardware. Only the bare essentials of assembly

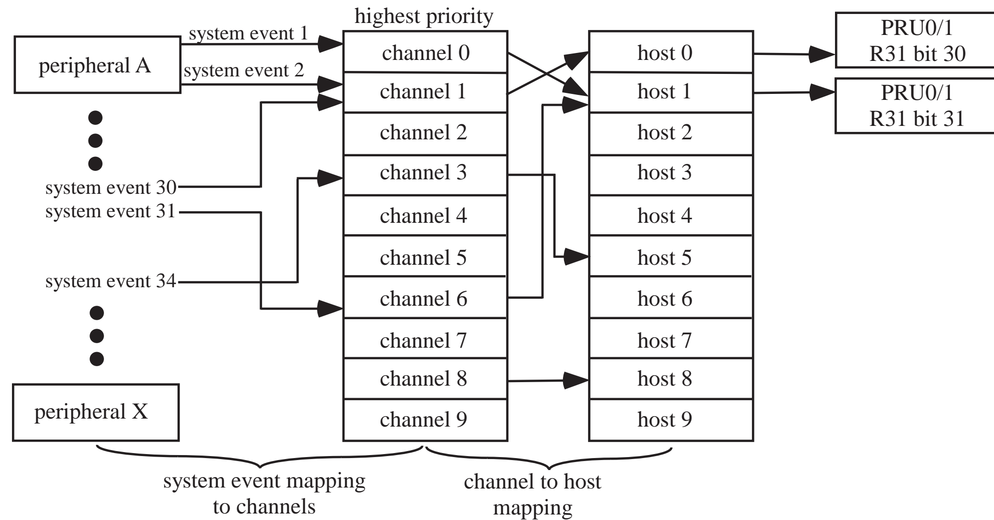


Figure 6.38: PRU interrupt controller [PRU-ICSS Reference Guide, 2013].

language programming are provided here. Excellent tutorial information is available from the Texas Instruments website including:

- the *PRU Assembly Reference Guide*, and the
- *Programmable Real-time Unit Subsystem Training Material.*

An overview of the PASM assembler is provided in Figure 6.40. A PASM program consists of directives, preprocessor commands, labels, and instructions. The PASM program file should end with the "p" suffix.

A directive is also known as a dot "." command because the commands are preceded by a dot. They are used to control the assembler and define user-defined data types. Details on using these commands are provided in the *PRU Assembly Reference Guide.*

The preprocessor commands are also known as hash (#) commands because the commands are preceded by a hash symbol. These command types are used to control the assembler preprocessor. The commands are used to include additional files (#include) or for text substitution using the define (#define) command. Additional hash commands are defined in the *PRU Assembly Reference Guide.*

Labels are used to give memory addresses a name. This allows for assembly code to be more readable and also eases the writing of assembly language programs.

The final category of commands is instructions. As can be seen in Figure 6.40, instructions fall into five different categories:

P8 PRU General Purpose input/output

Pin	Name	DT Offset	Mode 5	Mode 6
P8, pin11	GPIO1_13	0x034		pr1_pru0_pru_r30_15 (output)
P8, pin 12	GPIO1_12	0x030		pr1_pru0_pru_r30_14 (output)
P8, pin 15	GPIO1_15	0x03C		pr1_pru0_pru_r31_15 (input)
P8, pin 16	GPIO1_14	0x038		pr1_pru0_pru_r31_14 (input)
P8, pin 20	GPIO1_31	0x084	pr1_pru1_pru_r30_13 (output)	pr1_pru1_pru_r31_13 (input)
P8, pin 21	GPIO1_30	0x080	pr1_pru1_pru_r30_12 (output)	pr1_pru1_pru_r31_12 (input)
P8, pin 27	GPIO2_22	0x0E0	pr1_pru1_pru_r30_8 (output)	pr1_pru1_pru_r31_8 (input)
P8, pin 28	GPIO2_24	0x0E8	pr1_pru1_pru_r30_10 (output)	pr1_pru1_pru_r31_10 (input)
P8, pin 29	GPIO2_23	0x0E4	pr1_pru1_pru_r30_9 (output)	pr1_pru1_pru_r31_9 (input)
P8, pin 30	GPIO2_25	0x0EC	pr1_pru1_pru_r30_11 (output)	pr1_pru1_pru_r31_11 (input)
P8, pin 39	GPIO2_12	0x0B8	pr1_pru1_pru_r30_6 (output)	pr1_pru1_pru_r31_6 (input)
P8, pin 40	GPIO2_13	0x0BC	pr1_pru1_pru_r30_7 (output)	pr1_pru1_pru_r31_7 (input)
P8, pin 41	GPIO2_10	0x0B0	pr1_pru1_pru_r30_4 (output)	pr1_pru1_pru_r31_4 (input)
P8, pin 42	GPIO2_11	0x0B4	pr1_pru1_pru_r30_5 (output)	pr1_pru1_pru_r31_5 (input)
P8, pin 43	GPIO2_8	0x0A8	pr1_pru1_pru_r30_2 (output)	pr1_pru1_pru_r31_2 (input)
P8, pin 44	GPIO2_9	0x0AC	pr1_pru1_pru_r30_3 (output)	pr1_pru1_pru_r31_3 (input)
P8, pin 45	GPIO2_6	0x0A0	pr1_pru1_pru_r30_0 (output)	pr1_pru1_pru_r31_0 (input)
P8, pin 46	GPIO2_7	0x0A4	pr1_pru1_pru_r30_1 (output)	pr1_pru1_pru_r31_1 (input)

P9 PRU General Purpose input/output

Pin	Name	DT Offset	Mode 5	Mode 6
P9, pin 24	UART1_TXD	0x184	---	pr1_pru0_pru_r31_16 (input)
P9, pin 25	GPIO3_21	0x1AC	pr1_pru0_pru_r30_7 (output)	pr1_pru0_pru_r31_7 (input)
P9, pin 26	UART1_RXD	0x180	---	pr1_pru1_pru_r31_16 (input)
P9, pin 27	GPIO3_19	0x1A4	pr1_pru0_pru_r30_5 (output)	pr1_pru0_pru_r31_5 (input)
P9, pin 28	SPI1_CS0	0x19C	pr1_pru0_pru_r30_3 (output)	pr1_pru0_pru_r31_3 (input)
P9, pin 29	SPI1_D0	0x194	pr1_pru0_pru_r30_1(output)	pr1_pru0_pru_r31_1 (input)
P9, pin 30	SPI1_D1	0x198	pr1_pru0_pru_r31_2 (output)	pr1_pru0_pru_r31_2 (input)
P9, pin 31	SPI1_SCLK	0x190	pr1_pru0_pru_r30_0 (output)	pr1_pru0_pru_r31_0(input)
P9, pin 41	CLKOUT2	0x1B4	pr1_pru0_pru_r30_16 (input)	---

Figure 6.39: PRU general purpose input and output pins [PRU-ICSS Reference Guide, 2013].

- arithmetic,
- logical,
- flow control,
- register load/store, and
- input/output operations.

The general format of the assembly language command is the assembler instruction mnemonic followed by a comma separated list of parameters. The parameters used in the instruction may be registers, labels, numbers (immediate values), or entries from the constant table, as shown in Figure 6.41.

The next section provides details on obtaining, downloading, and configuring PRU software development tools.

PRU Assembly (PASM) Programming

Directives (".")
.origin
.entrypoint
.setcallreg
.macro, .mparam, .endm
.struct, .ends, .u32, .u16, .u8
.assign
.enter
.leave
.using

Preprocessor Commands (#)
include
#define

Instructions

Labels - denote program address

Arithmetic
ADD
ADC
SUB
SUC
RSB
RSC

Logical
LSL
LSR
AND
OR
XOR
NOT
MIN
MAX
CLR
SET
SCAN
LMBD
MOV

Flow Control
JAL
JMP
QBGT
QBGE
QBLT
QBLE
QBEQ
QBNE
QBA
QBBS
QBBC
HALT
SLP

Register Load/Store
LDI
LBBO
SBBO
LBCO
SBCO

Input/Output Operations

PSEUDO OPS
ZERO
MVIB
MVIW
MVIO
WBS
WBC
CALL
RET

PSEUDO OPS

Figure 6.40: PRU assembly language [PRU-ICSS Reference Guide, 2013].

Register Parameter Types

Parameter Name	Meaning	Examples
REG, REG1, REG2...	Register field from 8 to 32 bits	r0, r1.w0, r3.b2
Rn, Rn1, Rn2...	32-bit register field (ro through r31)	r0, r1
Rn.tx	Any 1 bit register field	r0.t23, r1.b2.t5
Cn, Cn1, Cn2,...	32-bit constant table entry (co through c31)	c0, c1
bn	specifies field that must be b0, b1, b2 or b3... denoting ro.bo, ro.b1, r0.b2, and r0.b3	b0, b1
LABEL	Any valid label denoting an instruction address	loop1
IM(n)	any immediate value from 0 to n	#23, 0b01101, 2+2, &r3w2
OP(n)	Union of of REG and IM(n)	r0, r1.w0

PRU Register Accesses

Suffix	Range of n	Meaning
.wn	0 to 2	16 bit field with a byte offset of n with the parent field
.bn	0 to 3	8 bit field with a byte offset of n with the parent field
.tn	0 to 31	1 bit field with a byte offset of n with the parent field

Figure 6.41: PRU register description [PRU-ICSS Reference Guide, 2013].

6.16.6 DEVELOPMENT PROCESS

The Debian release of the BeagleBone Black operating system contains support for both PRUs including the PRU Assembler (PASM) and the PRU Linux Application loader. A flow chart describing the process to program the PRUs is provided in Figure 6.42. Each step is described briefly followed by an illustrative example. This information was compiled, from a number of excellent, available BeagleBone Black PRU examples [www.element14.com; Coley, 2014].

It must be emphasized there are two different programs that must be written, assembled/-compiled, and executed for each PRU-based application: (1) the assembly language program for execution on the PRU ($< filename.bin >$), and (2) the compiled C program for execution on BeagleBone ($< filename.o >$) with the purpose of uploading the "bin" file to the PRU and interacting with it. Here is a brief summary of steps.

1. **Download and install PRU software support.** As previously mentioned, the Debian release has PRU software support included (PRU Assembler and PRU Linux Application Loader).
2. **Enable the PRU system onboard BeagleBone Black.** The universal cape enables the PRU. The cape is installed using:

   ```
   #echo cape-universaln > /sys/devices/bone_capemgr.*/slots
   ```

   Alternatively the cape may be loaded using [Steinkuehler]:

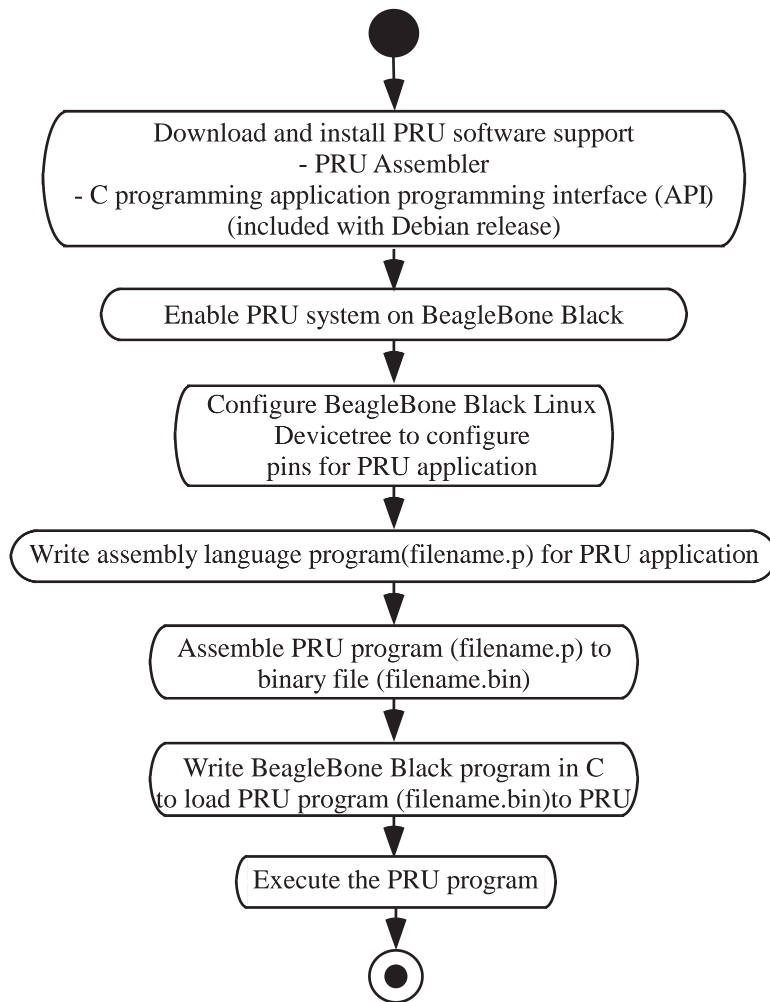


Figure 6.42: PRU software development process [PRU-ICSS Reference Guide, 2013].

```
# config-pin overlay cape-universal
```

3. **Configure specific pins for PRU operation.** Figure 6.39 provides a summary of PRU general purpose input and output pins. It may be used to select a specific pin (or pins) for a PRU application. Selected pins may be configured for PRU operation using the "config--pin" utility. For example, pin P8.11 (GPIO1_13) may be configured for PRU operation using [Steinkuehler]:

```
# config-pin -l P8.11
default  gpio  gpio_pu  gpio_pd  pruout  qep
# config-pin P8.11 pruout
```

4. **Write assembly language program (filename.p) for PRU application.** To develop your own PRU application, it is helpful to find a similar example program and modify it to meet your needs. The program that follows was provided by www.element14.com. It was originally modified from PRU_memAccess_DDR_PRUsharedRAM available on the Texas Instruments website [TI, www.TI.com].

   Once written, the assembly language program is assembled using [Molloy, 2015]:

```
# pasm -b  <filename.p>
```

   This will result in a file named filename.bin.

5. **Write BeagleBone Black program in C to load PRU program to PRU.** As previously mentioned it is helpful to modify an existing example. Samples are available from the Texas Instruments website.

   As a friendly reminder, the program is executed on BeagleBone Black and controls interaction with the onboard PRUs. It uses pre-existing functions to initialize the PRU, open and initialize the PRU interrupt, initialize the PRU example (written is assembly language, execute the example on the PRU, wait for the program to finish, and then provide status on the program.

   Once written, the program is compiled and linked with related libraries using [Molloy, 2015];

```
#gcc filename.c -o filename -lpthread -libraryname
```

6. **Execute the PRU program.** This is accomplished using:

```
#./filename
```

Example: In this example an LED connected to BeagleBone Black pin P8.11 (GPIO1_13, pr1_pr0_pru_r30_15) is placed under PRU control and blinked ten times. The program that follows was adapted from an example provided by www.element14.com. It was originally modified from PRU_memAccess_DDR_PRUsharedRAM available on the Texas Instruments website [TI, www.TI.com]. The website contains the *.p, *.c, and the *.hp files. The files have been renamed pru_ex.c, pru_ex.p, and pru_ex.hp for convenience. We take a step-by-step approach to develop, assemble/compile, and execute the program.

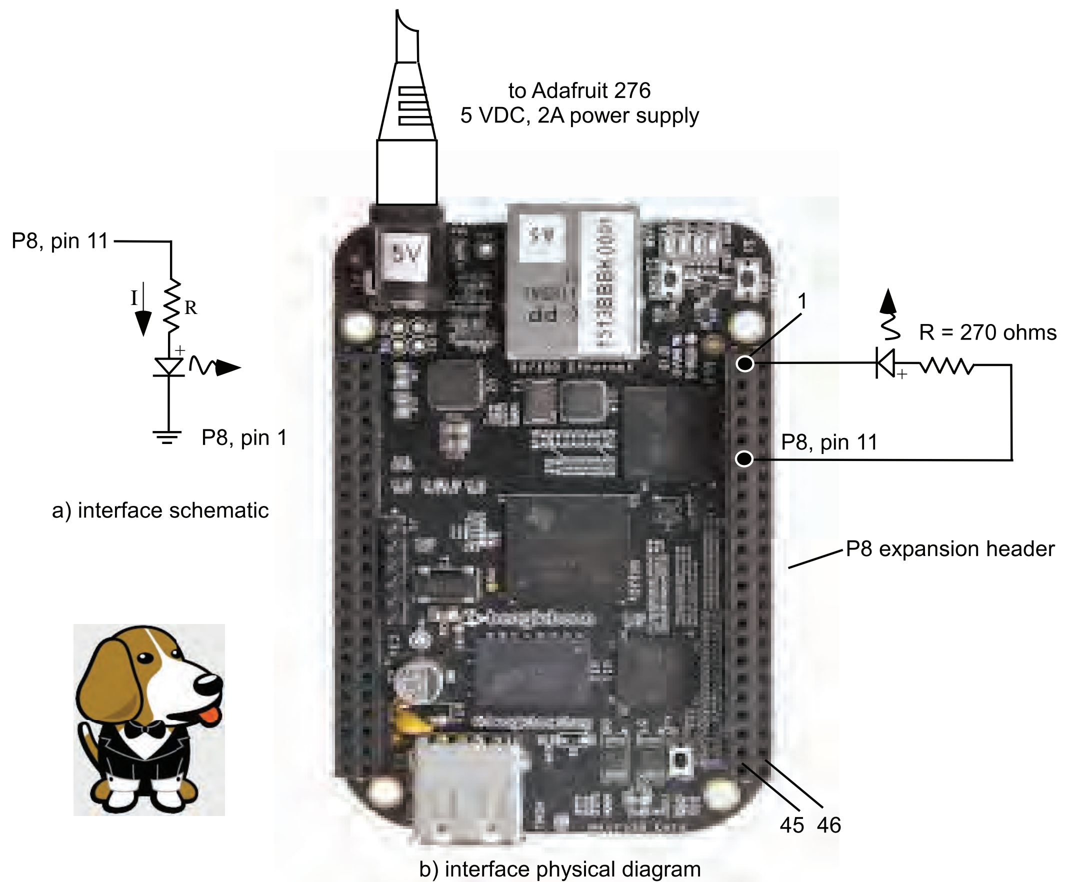


Figure 6.43: LED under PRU control. (Illustrations used with permission of Texas Instruments (www.TI.com).)

1. **Enable the PRU system onboard BeagleBone Black.** The universal cape enables the PRU. It is installed using [Steinkuehler]:

```
#echo cape-universaln > /sys/devices/bone_capemgr.*/slots
```

Alternatively the cape may be loaded using [Steinkuehler]:

```
# config-pin overlay cape-universal
```

2. **Configure specific pins for PRU operation.** In this example an LED connected to BeagleBone Black pin P8.11 (GPIO1_13, pr1_pr0_pru_r30_15) is placed under PRU control and blinked ten times. The circuit diagram is provided in Figure 6.43. Pin P8.11 (GPIO1_13, pr1_pr0_pru_r30_15) may be configured for PRU operation using [Steinkuehler]:

```
# config-pin -l P8.11
default  gpio  gpio_pu  gpio_pd  pruout  qep
# config-pin P8.11 pruout
```

3. **Write assembly language program for PRU.** In this example an LED connected to BeagleBone Black pin P8.11 (GPIO1_13, pr1_pr0_pru_r30_15) is placed under PRU control and blinked ten times. The program uses the following registers:

   - **Register R0:** is used to store the delay count
   - **Register R1:** contains the loop counter. For this specific example, a value of 10 is used.
   - **Register R30:** is used to turn the LED connected to pin P8.11 (GPIO1_13, pr1_pr0_pru_r30_15) on and off. Recall, PRU register R30 provides for general purpose outputs. Figure 6.39 provides information of available PRU pins. The Mode 6 column indicates the register and bit to control a specific pin. For example, P8.11 is controlled by register R30, bit 15. In the example, we set and clear this bit to set and clear BeagleBone Black pin P8.11.

   The program uses the following assembly language instructions. Details on each instruction are provided in the *PRU Assembly Reference Guide*.

   - **ST32:** is a macro defined in the header file that stores 32 bits from source to destination.
   - **LBCO:** Load Byte Burst with Constant Table Offset command reads a block of data from memory to a specified register file. It takes the form:

     ```
     LCBO  Destination,   Source, (+ optional offset),  Number of bytes
     LCBO  REG1,          Cn2, OP(255,                  IM(124)
     LCBO  r0, CONST_PRUCFG, 4, 4
     ```

   - **CLR:** Clears specified bit in the source and copies result to destination.

     ```
     CLR r0, r0, 4
     ```

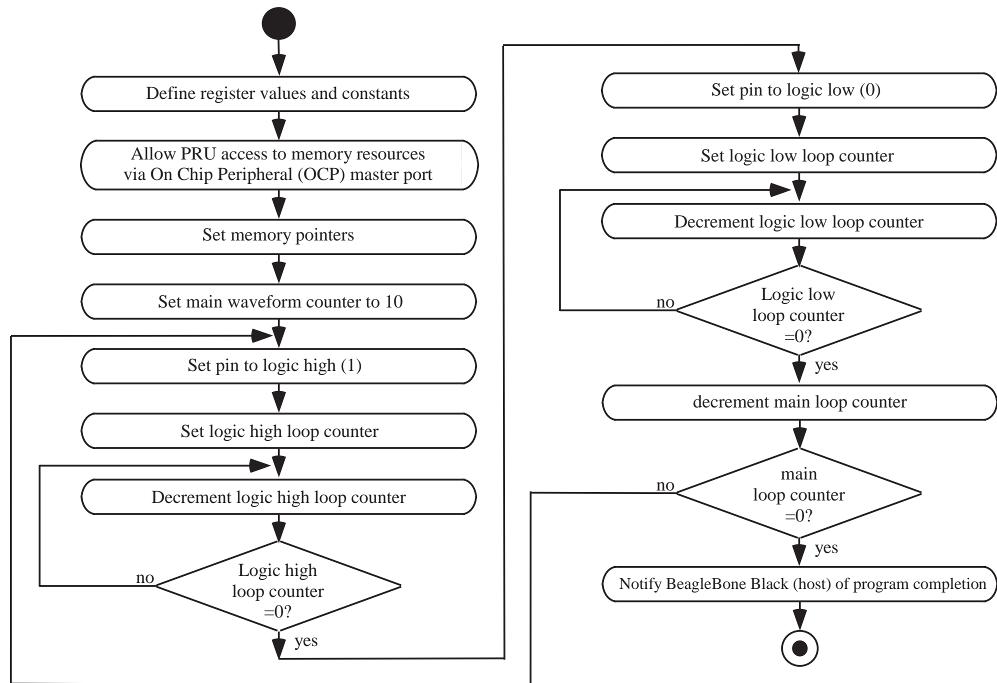


Figure 6.44: PRU example flow. LED USR0 will flash ten times.

- **SBCO:** Store Byte Burst with Constant Table Offset writes a block of data from the register file to memory.

```
SBCO  Source, Destination, (+ optional offset),  Number of bytes
SBCO  r0, CONST_PRUCFG, 4, 4
```

- **MOV:** Moves from source to destination

```
MOV   DEST, SOURCE
```

- **SBBO:** Store Byte Burst writes a block of data from the register file to memory.

```
SBBO  REG1,  Rn2, OP (255),  IM (124)
SBBO  r2, r1, 5, 8  //copy 8 bytes from r2/r3 to memory address r1+5
```

- **SUB:** Unsigned integer subtract

```
SUB  REG1, REG2, OP(255)      //REG1 = REG2 minus OP(255)
```

- **QBNE:** Quick Branch if Not Equal instruction jumps if the value of OP(255) is not equal to REG1.

```
QBNE  LABEL, REG1, OP(255)
```

```
1  //******************************************************************
2  // pru_ex.p
3  //
4  //Copyright (C) 2012 Texas Instruments Incorporated - http://www.ti.com/
5  //
6  //Redistribution and use in source and binary forms, with or without
7  //modification, are permitted provided that the following conditions
8  //are met:
9  //
10 //Redistributions of source code must retain the above copyright
11 //notice, this list of conditions and the following disclaimer.
12 //
13 //Redistributions in binary form must reproduce the above copyright
14 //notice, this list of conditions and the following disclaimer in the
15 //documentation and/or other materials provided with the
16 //distribution.
17 //
18 //Neither the name of Texas Instruments Incorporated nor the names of
19 //its contributors may be used to endorse or promote products derived
20 //from this software without specific prior written permission.
21 //
22 //THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
23 //"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
24 //LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
25 //A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
26 //OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
27 //SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
28 //LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
29 //DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
30 //THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
31 //(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
32 //OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
33 //=====================================================================
34 //Copyright (c) Texas Instruments Inc 2010-12
35 //
36 //Use of this software is controlled by the terms and conditions found in
37 //the license agreement under which this software has been supplied or
38 //provided.
39 //*********************************************************************
40
41 .origin 0                    //start next instruction at code offset 0
42 .entrypoint START            //specify code entry point to debugger
43
44                              //header file contains definitions of
45 #include "pru_ex.hp"         //constants and macros used within program
46
47 START:
48    //Allow PRU to access shared memory assets within host processor
49    //Enable On Chip Peripheral (OCP) master port
50    // - allows communication between the PRUs and host processor
51    // - provides for shared memory access
52    //
53    //LCBO: Load Byte Burst with Constant Table command - used to read
54    //a block of data from memory data into register r0 from the memory
55    //address CONST_PRUCFG + 4. This refers to the PRU-ICSS CFG
56    //configuration registers for the PRU system SYSCFG register
57    //defined in the header file and the SPRUH73K reference.
58
59    LBCO    r0, CONST_PRUCFG, 4, 4
60
61    //Clears STANDBY_INIT bit in SYSCFG register to enable OCP master
62    //port. Allows PRU to write to memory locations outside the PRU
63    //memory space - BeagleBone pins.
64    //SBCO: Store Byte Burst with Constant Table Offset
65
66    CLR     r0, r0, 4
67    SBCO    r0, CONST_PRUCFG, 4, 4
68
69    //Configure the programmable pointer register for PRU0 by setting
70    //c28_pointer[15:0] field to 0x0120.
71    //This will make C28 point to 0x00012000 (PRU shared RAM).
72    //The CTPPR0 register: Constant Table Programmable Pointer Register0.
73    //Allows PRU to set up 256-byte page index for entries 28 and 29 in
74    //the PRU constant table.
75
76    MOV     r0, 0x00000120          //MOV: copy value
77    MOV     r1, CTPPR_0
78    ST32    r0, r1
79
80    //Configure the programmable pointer register for PRU0 by setting
81    //c31_pointer[15:0] field to 0x0010.
82    //This will make C31 point to 0x80001000 (DDR memory).
83
84    MOV     r0, 0x00100000
85    MOV     r1, CTPPR_1
86    ST32    r0, r1
```

```
87
88      //Load values from external DDR Memory into Registers R0/R1/R2
89
90      LBCO      r0, CONST_DDR, 0, 12
91
92      //Store values from read from the DDR memory into PRU shared RAM
93
94      SBCO      r0, CONST_PRUSHAREDRAM, 0, 12
95
96      //test GP output
97
98      MOV r1, 10
99
100                                      //Execute main loop (LOOP) 10 times
101 LOOP1:
102     SET r30.t15                      //set GPIO1_13 to logic one
103
104                                      //Recall operating freq: 200 MHz
105     MOV r0, 0x00f00000               //load high time counter
106                                      //loop counter 15,728,640
107                                      //approximate 15.7 ms high time
108
109 DEL1:
110     SUB r0, r0, 1                    //SUB: unsigned integer subtract
111                                      //  decrement r0
112     QBNE DEL1, r0, 0                 //QBNE: quick branch if not equal
113                                      //counter at 0?
114                                      //If not, loop back to DEL1
115
116     CLR r30.t15                      //set GPIO1_13 to logic zero
117
118                                      //Recall operating freq: 200 MHz
119     MOV r0, 0x00f00000               //load low time counter
120                                      //loop counter 15,728,640
121                                      //approximate 15.7 ms high time
122 DEL2:
123     SUB r0, r0, 1                    //SUB: unsigned integer subtract
124                                      //  decrement r0
125     QBNE DEL2, r0, 0                 //QBNE: quick branch if not equal
126                                      //  counter at 0?
127                                      //If not, loop back to DEL2
128
129     SUB r1, r1, 1                    //SUB: unsigned integer subtract
130     QBNE LOOP1, r1, 0                //QBNE: quick branch if not equal
131                                      //  counter at 0?
132                                      //If not, loop back to LOOP
133
134                                      //Send notification to Host for
135                                      // program completion
136     MOV       r31.b0, PRU0_ARM_INTERRUPT+16
137
138     HALT                             // Halt the processor
139 //************************************************************
```

The program is then assembled with the corresponding header file (.hp). The (.hp) file contains definitions and macros useful for the program.

```
1  //************************************************************
2  //pru_ex.hp
3  //
4  //Copyright (C) 2012 Texas Instruments Incorporated - http://www.ti.com/
5  //
6  //Redistribution and use in source and binary forms, with or without
7  //modification, are permitted provided that the following conditions
8  //are met:
9  //
10 //Redistributions of source code must retain the above copyright
11 //notice, this list of conditions and the following disclaimer.
12 //
13 //Redistributions in binary form must reproduce the above copyright
14 //notice, this list of conditions and the following disclaimer in the
15 //documentation and/or other materials provided with the
16 //distribution.
17 //
18 //Neither the name of Texas Instruments Incorporated nor the names of
19 //its contributors may be used to endorse or promote products derived
20 //from this software without specific prior written permission.
21 //
22 //THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
23 //"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
24 //LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
25 //A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
26 //OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
27 //SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
28 //LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
```

```
29  //DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
30  //THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
31  //(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
32  //OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
33  //
34  //=====================================================================
35  //Copyright (c) Texas Instruments Inc 2010-12
36  //
37  //Use of this software is controlled by the terms and conditions found in the
38  //license agreement under which this software has been supplied or provided.
39  //*********************************************************************
40  //file:    pru_ex.hp
41  //brief:   PRU_memAccess_DDR_PRUsharedRAM assembly constants.
42  //(C) Copyright 2012, Texas Instruments, Inc
43  //author      M. Watkins
44  //*********************************************************************
45
46  #ifndef _PRU_memAccess_DDR_PRUsharedRAM_HP_
47  #define _PRU_memAccess_DDR_PRUsharedRAM_HP_
48
49  //*********************************************************************
50  //Global Macro definitions
51  //*********************************************************************
52
53  //Refer to mapping in the file -\prussdrv\include\pruss_intc_mapping.h
54  #define PRU0_PRU1_INTERRUPT      17
55  #define PRU1_PRU0_INTERRUPT      18
56  #define PRU0_ARM_INTERRUPT       19
57  #define PRU1_ARM_INTERRUPT       20
58  #define ARM_PRU0_INTERRUPT       21
59  #define ARM_PRU1_INTERRUPT       22
60
61  #define CONST_PRUCFG             C4
62  #define CONST_PRUDRAM            C24
63  #define CONST_PRUSHAREDRAM       C28
64  #define CONST_DDR                C31
65
66  //Address for the Constant table Block Index Register (CTBIR)
67  #define CTBIR           0x22020
68
69  //Address for the Constant table Programmable Pointer Register 0(CTPPR_0)
70  #define CTPPR_0         0x22028
71
72  //Address for the Constant table Programmable Pointer Register 1(CTPPR_1)
73  #define CTPPR_1         0x2202C
74
75
76  .macro  LD32
77  .mparam dst,src
78      LBBO    dst,src,#0x00,4
79  .endm
80
81  .macro  LD16
82  .mparam dst,src
83      LBBO    dst,src,#0x00,2
84  .endm
85
86  .macro  LD8
87  .mparam dst,src
88      LBBO    dst,src,#0x00,1
89  .endm
90
91  .macro ST32
92  .mparam src,dst
93      SBBO    src,dst,#0x00,4
94  .endm
95
96  .macro ST16
97  .mparam src,dst
98      SBBO    src,dst,#0x00,2
99  .endm
100
101 .macro ST8
102 .mparam src,dst
103     SBBO    src,dst,#0x00,1
104 .endm
105
106 //*********************************************************************
107 //Global Structure Definitions
108 //*********************************************************************
109
110 .struct Global
111     .u32 regPointer
112     .u32 regVal
113 .ends
114
115 //*********************************************************************
116 //Global Register Assignments
```

```
117 //***********************************************************************
118
119 .assign Global, r2, *, global
120
121 #endif //_PRU_memAccess_DDR_PRUsharedRAM_
122
123 //***********************************************************************
```

4. **Write BeagleBone Black C program to load assembly language program to PRU.**

```
1  //***********************************************************************
2  //pru_ex.c
3  //
4  //Copyright (C) 2012 Texas Instruments Incorporated - http://www.ti.com/
5  //
6  //Redistribution and use in source and binary forms, with or without
7  //modification, are permitted provided that the following conditions
8  //are met:
9  //
10 //Redistributions of source code must retain the above copyright
11 //notice, this list of conditions and the following disclaimer.
12 //
13 //Redistributions in binary form must reproduce the above copyright
14 //notice, this list of conditions and the following disclaimer in the
15 //documentation and/or other materials provided with the
16 //distribution.
17 //
18 //Neither the name of Texas Instruments Incorporated nor the names of
19 //its contributors may be used to endorse or promote products derived
20 //from this software without specific prior written permission.
21 //
22 //THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
23 //"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
24 //LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
25 //A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
26 //OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
27 //SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
28 //LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
29 //DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
30 //THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
31 //(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
32 //OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
33 //=======================================================================
34 //Copyright (c) Texas Instruments Inc 2010-12
35 //
36 //Use of this software is controlled by the terms and conditions found in
37 //the license agreement under which this software has been supplied or
38 //provided.
39 //***********************************************************************
40 //pru_ex.c
41 //
42 //The PRU reads three values from external DDR memory and stores these
43 //values in shared PRU RAM using the programmable constant table
44 //entries. The example initially loads 3 values into the external DDR
45 //RAM. The PRU configures its Constant Table Programmable Pointer
46 //Register 0 and 1 (CTPPR_0, 1) to point to appropriate locations in the
47 //DDR memory and the PRU shared RAM. The values are then read from the
48 //DDR memory and stored into the PRU shared RAM using the values in the
49 //28th and 31st entries of the constant table.
50 //***********************************************************************
51 //Include Files
52 //***********************************************************************
53
54 // Standard header files
55 #include <stdio.h>
56 #include <sys/mman.h>
57 #include <fcntl.h>
58 #include <errno.h>
59 #include <unistd.h>
60 #include <string.h>
61
62 // Driver header file
63 #include "prussdrv.h"
64 #include <pruss_intc_mapping.h>
65
66 //***********************************************************************
67 //Explicit External Declarations
68 //***********************************************************************
69
70 //***********************************************************************
71 //Local Macro Declarations
72 //***********************************************************************
73
74 #define PRU_NUM      0
75 #define ADDEND1      0x98765400u
76 #define ADDEND2      0x12345678u
```

```
77  #define ADDEND3          0x10210210u
78
79  #define DDR_BASEADDR        0x80000000
80  #define OFFSET_DDR      0x00001000
81  #define OFFSET_SHAREDRAM 2048        //equivalent with 0x00002000
82
83  #define PRUSS0_SHARED_DATARAM     4
84
85  //*********************************************************************
86  //Local Typedef Declarations
87  //*********************************************************************
88
89
90  //*********************************************************************
91  //Local Function Declarations
92  //*********************************************************************
93
94  static int LOCAL_exampleInit ( );
95  static unsigned short LOCAL_examplePassed ( unsigned short pruNum );
96
97  //*********************************************************************
98  //Local Variable Definitions
99  //*********************************************************************
100
101 //*********************************************************************
102 //Interrupt Service Routines
103 //*********************************************************************
104
105
106 //*********************************************************************
107 //Global Variable Definitions
108 //*********************************************************************
109
110 static int mem_fd;
111 static void *ddrMem, *sharedMem;
112 static unsigned int *sharedMem_int;
113
114 //*********************************************************************
115 //Global Function Definitions
116 //*********************************************************************
117
118 int main (void)
119 {
120 unsigned int ret;
121 tpruss_intc_initdata pruss_intc_initdata = PRUSS_INTC_INITDATA;
122
123 printf("\nINFO: Starting %s example.\r\n", "PRU_example");
124
125 /* Initialize the PRU */
126 prussdrv_init ();
127
128 /* Open PRU Interrupt */
129 ret = prussdrv_open(PRU_EVTOUT_0);
130 if(ret)
131   {
132   printf("prussdrv_open open failed\n");
133   return (ret);
134   }
135
136 /* Get the interrupt initialized */
137 prussdrv_pruintc_init(&pruss_intc_initdata);
138
139 /* Initialize example */
140 printf("\tINFO: Initializing example.\r\n");
141 LOCAL_exampleInit(PRU_NUM);
142
143 /* Execute example on PRU */
144 printf("\tINFO: Executing example.\r\n");
145 prussdrv_exec_program (PRU_NUM, "./pru_ex.bin");
146
147 /* Wait until PRU0 has finished execution */
148 printf("\tINFO: Waiting for HALT command.\r\n");
149 prussdrv_pru_wait_event (PRU_EVTOUT_0);
150 printf("\tINFO: PRU completed transfer.\r\n");
151 prussdrv_pru_clear_event (PRU_EVTOUT_0, PRU0_ARM_INTERRUPT);
152
153 /* Check if example passed */
154 if ( LOCAL_examplePassed(PRU_NUM) )
155   {
156   printf("Example executed succesfully.\r\n");
157   }
158 else
159   {
160   printf("Example failed.\r\n");
161   }
162
163 /* Disable PRU and close memory mapping*/
164 prussdrv_pru_disable(PRU_NUM);
```

```
165 prussdrv_exit ();
166 munmap(ddrMem, 0x0FFFFFFF);
167 close(mem_fd);
168
169 return(0);
170 }
171
172 //**********************************************************************
173 //Local Function Definitions
174 //**********************************************************************
175
176 static int LOCAL_exampleInit ( )
177 {
178     void *DDR_regaddr1, *DDR_regaddr2, *DDR_regaddr3;
179
180     /* open the device */
181     mem_fd = open("/dev/mem", O_RDWR);
182     if (mem_fd < 0) {
183         printf("Failed to open /dev/mem (%s)\n", strerror(errno));
184         return -1;
185     }
186
187     /* map the DDR memory */
188     ddrMem = mmap(0, 0x0FFFFFFF, PROT_WRITE | PROT_READ, MAP_SHARED, mem_fd, DDR_BASEADDR);
189     if (ddrMem == NULL) {
190         printf("Failed to map the device (%s)\n", strerror(errno));
191         close(mem_fd);
192         return -1;
193     }
194
195     /* Store Addends in DDR memory location */
196     DDR_regaddr1 = ddrMem + OFFSET_DDR;
197     DDR_regaddr2 = ddrMem + OFFSET_DDR + 0x00000004;
198     DDR_regaddr3 = ddrMem + OFFSET_DDR + 0x00000008;
199
200     *(unsigned long*) DDR_regaddr1 = ADDEND1;
201     *(unsigned long*) DDR_regaddr2 = ADDEND2;
202     *(unsigned long*) DDR_regaddr3 = ADDEND3;
203
204     return(0);
205 }
206
207 static unsigned short LOCAL_examplePassed ( unsigned short pruNum )
208 {
209     unsigned int result_0, result_1, result_2;
210
211      /* Allocate Shared PRU memory. */
212     prussdrv_map_prumem(PRUSS0_SHARED_DATARAM, &sharedMem);
213     sharedMem_int = (unsigned int*) sharedMem;
214
215     result_0 = sharedMem_int[OFFSET_SHAREDRAM];
216     result_1 = sharedMem_int[OFFSET_SHAREDRAM + 1];
217     result_2 = sharedMem_int[OFFSET_SHAREDRAM + 2];
218
219     return ((result_0 == ADDEND1) & (result_1 == ADDEND2) & (result_2 == ADDEND3)) ;
220
221 }
222
223
224
225 //**********************************************************************
```

Once written, the program is compiled using [Molloy, 2015]:

```
#gcc pru_ex.c -o pru_ex -lpthread -lprussdrv
```

5. **Execute the PRU program.** This is accomplished using:

```
#./pru_ex
```

For additional, advanced examples using the PRUs, please see [Molloy, 2015].

6.17 SUMMARY

In Chapters 1–5 of this book we employed BeagleBone as a user-friendly processor. We accessed its features and subsystems via the Bonescript programming environment. In this chapter we began to shift focus and "unleash" the power of BeagleBone as a Linux-based, 32-bit, super-scalar ARM Cortex A8 processor. We began with a brief review of the C and C++ tool programming chain followed by examples on how to interact with the digital and analog pins aboard the processor. We then took a closer look at the features and subsystems available aboard BeagleBone. We spent a good part of the chapter describing the exposed functions of BeagleBone. These are the functions accessible to the user via the P8 and P9 extension headers. We concluded the chapter with an introduction to the onboard PRUs. Throughout this chapter we provided sample programs on how to interact with and program the exposed functions.

6.18 REFERENCES

- *AM335X ARM—AX Microprocessors (MPUs) Technical Reference Manual.* Texas Instruments, SPRUH73H, 2013.
- *AM335X PRU—ICSS Reference Guide.* Texas Instruments, SPRUHF8A, 2013.
- *AM335X PWMSS Driver's Guide.* 2015. Texas Instruments; `http://processors.wiki.ti.com/index.php/AM335x_PWM_Driver's_Guide`.
- *AM335x Sitara Processors Technical Reference Manual.* 2014. Texas Instruments, SPRUH73K.
- *AM335X Linux Application Loader User Guide,* Texas Instruments. PRU Linux Application Loader, `http://processors.wiki.ti.com/index.php/PRU_Linux_Application_Loader`, Texas Instruments, 2015.
- Barrett, S. and Pack, D. 2005. *Embedded Systems Design and Applications with the 68HC12 and HCS12.* Upper Saddle River, NJ: Pearson Prentice Hall.
- Barrett, S. and Pack, D. 2006. *Processors Fundamentals for Engineers and Scientists.* Morgan & Claypool Publishers; `www.morganclaypool.com`
- Barrett, S. and Pack, D. 2008 *Atmel AVR Processor Primer Programming and Interfacing.* Morgan & Claypool Publishers; `www.morganclaypool.com`
- *BBB—Working with the PRU-ICSS/PRUSSv2.* 2013; `www.element14.com`.
- CAN Tutorial, `http://www.computer-solutions.co.uk/`, Computer Solutions Ltd, 2015.

- CircuitCo:BeagleBone LCD7, http://elinux.org/CircuitCo:BeagleBone_LCD7, 2015.
- CircuitCo–Printed Circuit Board Solutions, www.circuitco.com, 2015.
- Coley, G. 2014. *BeagleBone Black Rev C.1 Systems Reference Manual.* Richardson, TX: BeagleBoard.org Foundation; www.beagleboard.org.
- *Device Tree Usage*. 2005. http://devicetree.org/Device_Tree_Usage.
- Dulaney, E. 2010. *Linux All-In-One for Dummies.* Hoboken, NJ: Wiley Publishing, Inc.
- *Enable PWM on BeagleBone with Device Tree Overlays.* 2003. www.hipstercircuits.com.
- Horowitz, P. and Hill, W. 1990. *The Art of Electronics.* 2nd ed. New York: Cambridge University Press.
- Hughes-Croucher, T. and Wilson, M. 2012. *Node Up and Running.* Sebastopol, CA: O'Reilly Media, Inc.
- Kelley, A. and Pohl, I. 1998. *A Book on C—Programming in C.* 4th ed. Boston, MA: Addison Wesley.
- Kiessling, M. 2012. *The Node Beginner Guide: A Comprehensive Node.js Tutorial.*
- Korsch, J. and Garrett, L. 1988. *Data Structures, Algorithms, and Program Style Using C.* Boston, MA: PWS-Kent Publishing Company.
- Likely, G. 2015. *Linux and the Device Tree*; https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt.
- Molloy, D. 2015. *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux.* Indianapolis, IN: John Wiley & Sons.
- Pantelis Antoniou. 2015. "BeagleBone and the 3.8 Kernel;" www.elinux.org/BeagleBone_and_the_3.8_Kernel.
- Pollock, J. 2010. *JavaScript.* 3rd ed. New York: McGraw Hill.
- *PRU Linux Application Loader.* 2015. Texas Instruments; http://processors.wiki.ti.com/index.php/PRU_Linux_Application_Loader.
- *PRU Assembly Language Tools User's Guide.* 2014. Texas Instruments, SPRUHV6A.
- *PRU Assembly Reference Guide.* 2015. Texas Instruments. http://processors.wiki.ti.com/index.php/PRU_Assembly_Reference_Guide.

- Richardson, M. 2014. *Getting Started With BeagleBone.* Sebasatopol, CA: Maker Media.
- Steinkuehler, C. 2015. *BeagleBone Universal IO.* `https://github.com/cdsteinkuehler/beaglebone-universal-io`.
- SMSC. *LAN8710A/LAN8710Ai Small Footprint MII/RMII 10/100 Etthernet Transceiver with HP Auto—MDIX and flexPWR Technology.* 2012.
- *The I2C—Bus Specification.* 2000. Version 2.1, Philips Semiconductor.
- Traynor, B. `www.elinux.org/Beagleboard:TerminalShells`, 2015.
- Vander Veer, E. 2005. *JavaScript for Dummies.* 4th ed. Hoboken, NJ: Wiley Publishing, Inc.
- von Hagen, W. 2007. *Ubuntu Linux Bible.* Indianapolis, IN: Wiley Publishing, Inc.

6.19 CHAPTER EXERCISES

1. How does BeagleBone interact with a host computer?
2. Describe how to properly interface an LED to a processor.
3. Develop a glossary of Linux commands introduced in this chapter.
4. Given a sinusoid with 500 Hz frequency, what should be the minimum sampling frequency for an analog-to-digital converter, if we want to faithfully reconstruct the analog signal after the conversion?
5. If 12 bits are used to quantize a sampled signal, what is the number of available quantized levels? What will be the resolution of such a system if the input range of the analog-to-digital converter is 1.8 VDC?
6. A flex sensor provides 10 K ohm of resistance for 0° flexure and 40 K ohm of resistance for 90° of flexure. Design a circuit to convert the resistance change to a voltage change. (Hint: consider a voltage divider). Then design a transducer interface circuit to convert the output from the flex sensor circuit to voltages suitable for the BeagleBone ADC system.
7. Does the time to convert an analog input signal to a digital representation vary in a successive-approximation converter relative to the magnitude of the input signal? Explain.
8. Summarize the differences between the USART, SPI, and I2C methods of serial communication.
9. What is the primary difference between the UART and SPI serial communication systems?

10. What is the ASCII encoded value for "BeagleBone?"
11. What is the purpose of an interrupt?
12. Describe the flow of events when an interrupt occurs.
13. What is the primary advantage of the PRU system?
14. Modify the PRU example to generate a continuous square wave. What is the maximum frequency possible from the PRU?

CHAPTER 7

BeagleBone "Off the Leash"

Objectives: After reading this chapter, the reader should be able to do the following:

- Enjoy the full power and rapid prototyping features of the Bonescript environment.
- Develop the hardware and Bonescript based control algorithm for an inexpensive laser light show.
- Develop the hardware and Bonescript based control algorithm for an arbitrary waveform generator.
- Develop the hardware and Bonescript based control algorithm for a robot arm.
- Develop the hardware and Bonescript based control algorithm for a weather station.
- Develop the hardware and Bonescript based control algorithm for a Speak & Spell like device.
- Develop the hardware and a C based control algorithm for the Dagu Rover 5 treaded robot.
- Describe multiple Linux compatible open source libraries.
- Explore OpenCV computer vision features as a case study of available open source libraries.
- Program BeagleBone using OpenCV functions to capture and display facial images and place a moustache on the face.
- Construct Boneyard II—a portable BeagleBone platform.

7.1 OVERVIEW

In the early chapters of this book, we examined the Bonescript environment as a user-friendly tool to rapidly employ BeagleBone features right out of the box. In this chapter, we revisit Bonescript and demonstrate its power as a rapid prototyping tool to develop complex, processor-based, expandable systems employing multiple BeagleBone subsystems. Specifically, we develop a Bonescript based weather station and Speak & Spell like device. We then illustrate C-based system development. We construct a control system for a Dagu Rover 5 treaded robot. We use code developed in the previous chapters as building blocks for this system. We conclude the chapter by taking a brief look at the rich variety of open source libraries available to the Linux developer.

As a case study, we review some of the fundamental features of the OpenCV computer vision library and employ them in a fun image processing task to plant moustaches on face images. As a Linux-based processor with a clock speed as high as 1 GHz, BeagleBone is equipped to handle complex image processing tasks not possible with a microcontroller-based board.

We have carefully chosen each of these projects to illustrate how BeagleBone may be used in a variety of project areas including instrumentation intensive applications (weather station), in assistive and educational technology applications (Speak & Spell), in motor and industrial control applications (Dagu robot), and calculation intensive image processing applications (moustache cam).

7.2 BONEYARD II: A PORTABLE LINUX PLATFORM-BEAGLEBONE UNLEASHED

In the first five chapters of the book BeagleBone was "leashed" to a host computer. This is a good way to efficiently use the features of the host computer in application development. However, BeagleBone can be quickly unleashed and converted into a standalone Linux computer, as shown in Figures 7.1, 7.2, and 7.3. We dubbed this project Boneyard II. This capability would be especially useful for developing applications that will be used in a remote, autonomous application.

The Boneyard II is quickly assembled using off-the-shelf components including:

- an original BeagleBone (700 MHz) or black (1 GHz) processor board;
- the Circuitco LCD7 display;
- a Adafruit mini-keyboard;
- a USB hub;
- a mini USB mouse; and
- and a Pelican 1200 case.

All of the components may be purchased for under US $300. A configuration diagram is provided in Figure 7.4. This is truly a plug-and-play system. Simply by connecting the components as shown and powering up the system, a standalone, 1 GHz computer may be assembled for under $300.

7.3 BONEYARD III: A LOW-COST DESKTOP LINUX PLATFORM

BeagleBone Black may be used to assemble a low-cost desktop Linux platform. As mentioned earlier in the book, BeagleBone Black supports the High-Definition Multimedia Interface (HDMI).

Figure 7.1: Boneyard II–a standalone Linux computer. (Photo courtesy of Barrett, 2013.)

Figure 7.2: Boneyard II–a standalone Linux computer. (Photo courtesy of Barrett, 2013.)

Figure 7.3: Boneyard II—a standalone Linux computer. (Photo courtesy of Barrett, 2013.)

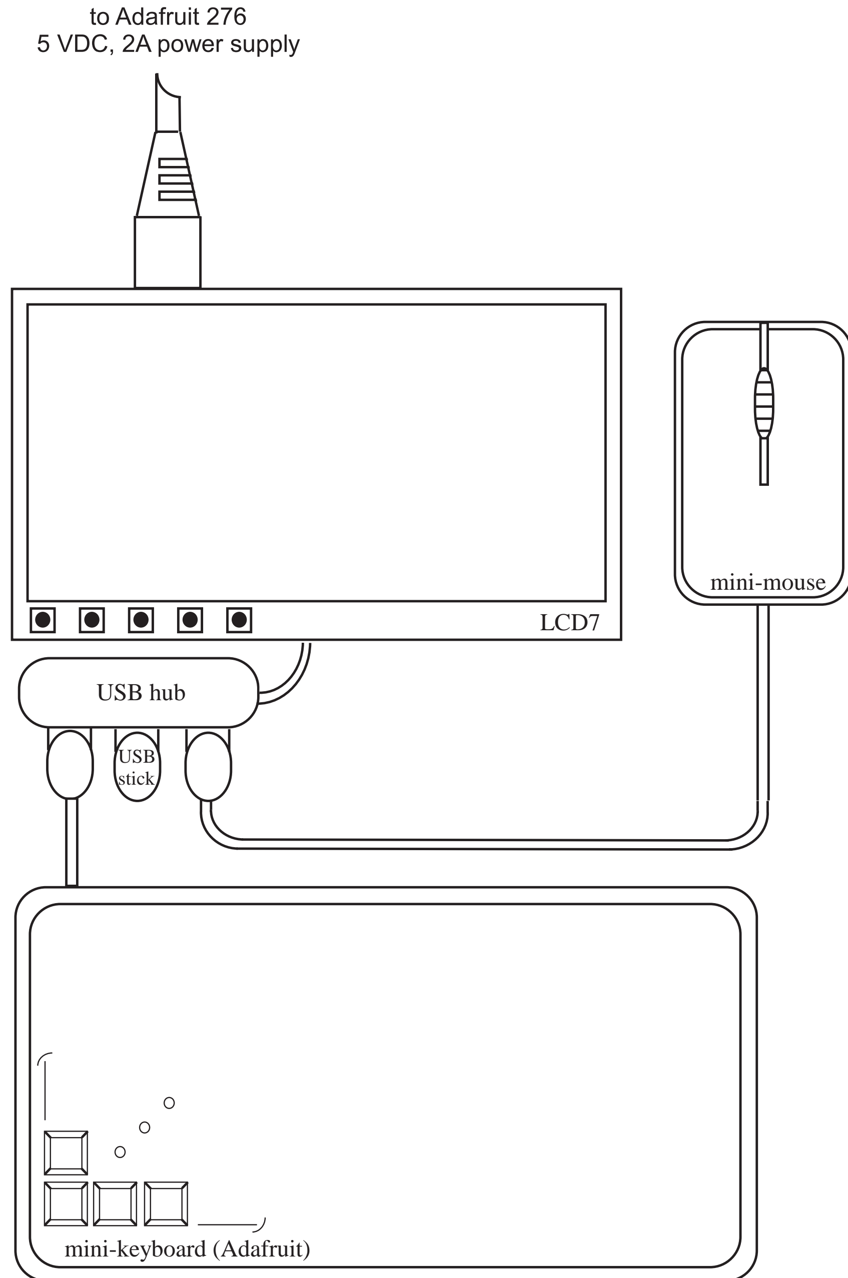


Figure 7.4: Boneyard II connection diagram.

The HDMI features are accessible via a microHDMI connector. BeagleBone Black can be equipped with an HDMI display via an HDMI to microHDMI cable as shown in Figure 7.5.

The Boneyard III is quickly assembled using off-the-shelf components including:

- BeagleBone Black processor board;
- HDMI display;
- HDMI to microHDMI cable;
- a Adafruit mini-keyboard;
- a USB hub; and
- a mini USB mouse.

All of the components may be purchased for under US $300. A configuration diagram is provided in Figure 7.5. This is truly a plug-and-play system. Simply by connecting the components as shown and powering up the system, a standalone, 1 GHz desktop computer may be assembled for under $300.

7.3.1 ACCESSING BONESCRIPT

Once configured you can access Bonescript using Boneyard III by starting the Firefox browser resident within the Debian Linux release and navigating to the Cloud 9 IDE (`http://192.168.7.2:3000`).

7.3.2 ACCESSING THE INTERNET

Once configured, Boneyard III may be connected to the Internet via the Ethernet features aboard BeagleBone Black. Use a standard Ethernet cable to connect from your Ethernet router to the Ethernet connection on BeagleBone Black. Once connected, the internet may be accessed by starting the Firefox browser resident within the Debian Linux release.

Once on the Internet, you can upgrade to the latest version of Bonescript by accessing the BeagleBone terminal prompt (Applications – > System Tools – > Terminal) and entering the following commands:

```
>cd /var/lib/cloud9
>git stash
>git pull
```

7.4 APPLICATION 1: INEXPENSIVE LASER LIGHT SHOW

An inexpensive laser light show may be constructed from two servos. In this example we use two Futaba 180° range servos (Parallax 900–00005, available from Jameco #283021) mounted, as

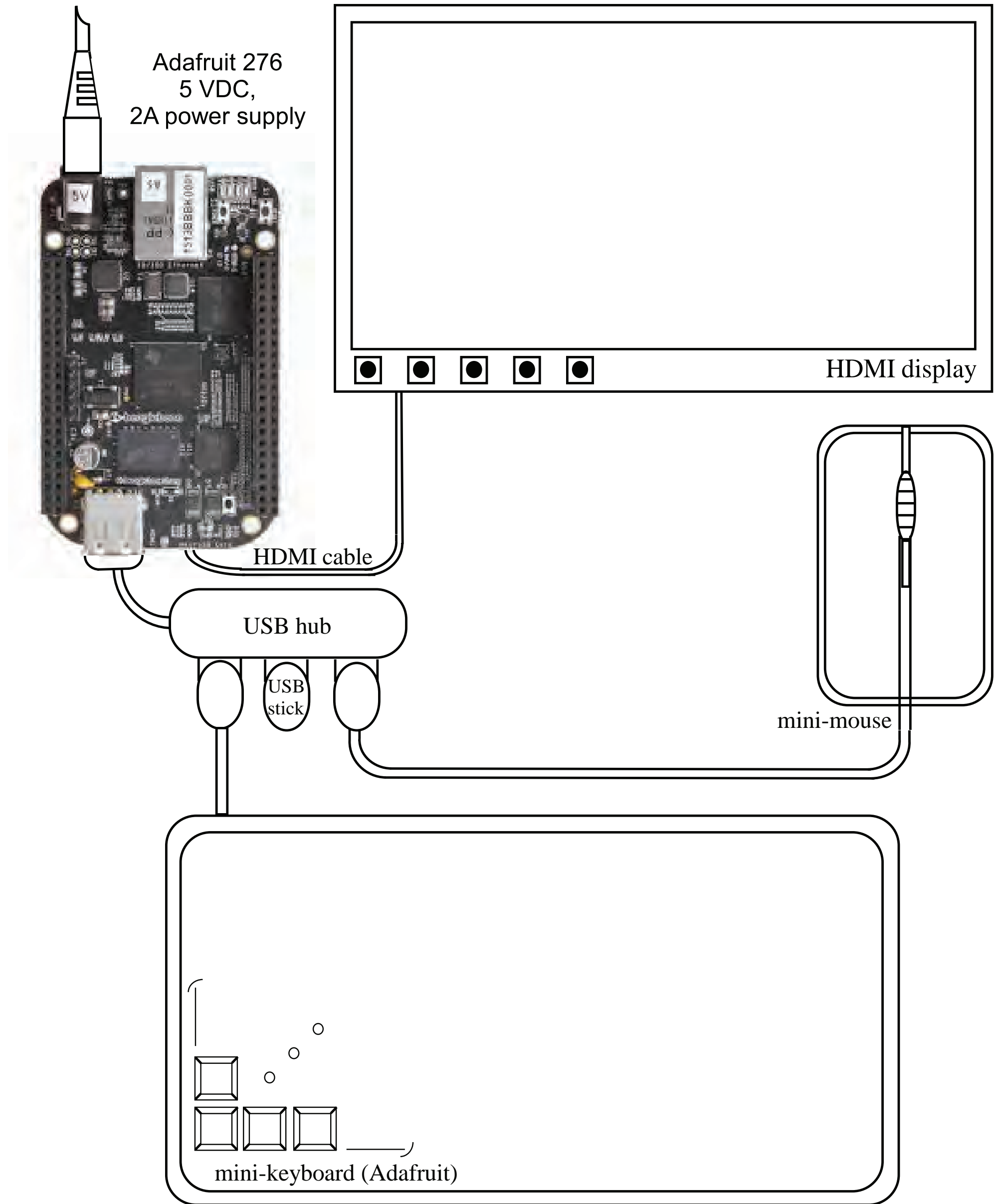


Figure 7.5: Boneyard III connection diagram—a low-cost desktop Linux platform. (Illustrations used with permission of Texas Instruments (`www.TI.com`).)

shown in Figure 7.6. The servos expect a pulse every 20 ms (50 Hz). The pulse length determines the degree of rotation from 1000 μs (5% duty cycle, $-90°$ rotation) to 2000 μs (10% duty cycle, $+90°$ rotation). The X and Y control signals are provided by BeagleBone Black. The X and Y control signals are interfaced to the servos via LM324 operational amplifiers. The laser source is provided by an inexpensive laser pointer.

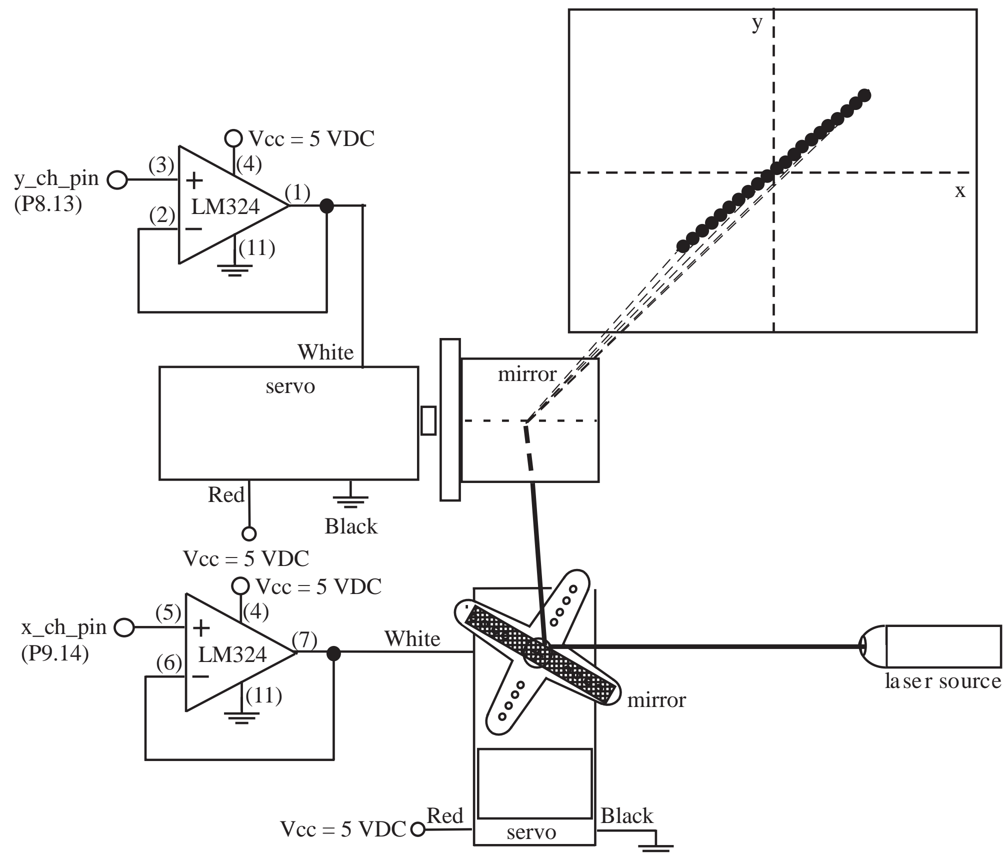


Figure 7.6: Inexpensive laser light show.

The laser_light_show.js program sends the same signal to both channel outputs (x_ch_pin, y_ch_pin) and traces a line with the laser. The analogWrite functions are configured with a 50 Hz baseline frequency. The duty cycle for each servo is varied from 5% duty cycle ($-90°$ rotation) to 10% duty cycle ($+90°$ rotation). The update_position function is called every 40 ms by the setInterval function, as shown in Figure 7.7.

```
1  //*******************************************************
2
3  var b = require('bonescript');
4
5  //initalize variables
6  var x_value = 0.05;          //5% duty cycle
7  var y_value = 0.05;          //5% duty cycle
8  var x_direction = 1;
9  var y_direction = 1;
10 var x_ch_pin = "P9_14";      //PWM equipped channel
11 var y_ch_pin = "P8_13";      //PWM equipped channel
12
13 // configure pin
14 b.pinMode(x_ch_pin, b.OUTPUT);
15 b.pinMode(y_ch_pin, b.OUTPUT);
16
17 // call function to update servos position every 40 ms
18 setInterval(update_position, 40);
19
20 // function to update servo position
21 //baseline servo frequency: 50 Hz
22 //servo duty cycle:
23 //  - x_value, y_value
24 //  - varies from 5% to 10%
25 //  - 5% corresponds to -90 degree servo rotation
26 //  - 10% corresponds to +90 degree servo rotation
27 //  - increment at 1% intervals
28
29 function update_position() {
30  b.analogWrite(x_ch_pin, x_value, 50);    //50 Hz baseline frequency
31  b.analogWrite(y_ch_pin, y_value, 50);    //50 Hz baseline frequency
32
33  //update x values
34  x_value = x_value + (x_direction * 0.01);  //increment/decrement by 1%
35  if(x_value > 0.10) {x_value = 0.10; x_direction = -1; }
36  else if(x_value <= 0.05) {x_value = 0.05; x_direction = 1; }
37
38 //update y values
39 y_value = y_value + (y_direction * 0.01);   //increment/decrement by 1%
40 if(y_value > 0.10) {y_value = 0.10; y_direction = -1;}
41 else if(y_value <= 0.05) {y_value = 0.05; y_direction = 1;}
42 }
43 //*******************************************************
```

The setInterval update rate and the increment/decrement percentage may be adjusted to vary the rate of high quickly the line is traced. Any arbitrary shape may be traced by the laser using this technique.

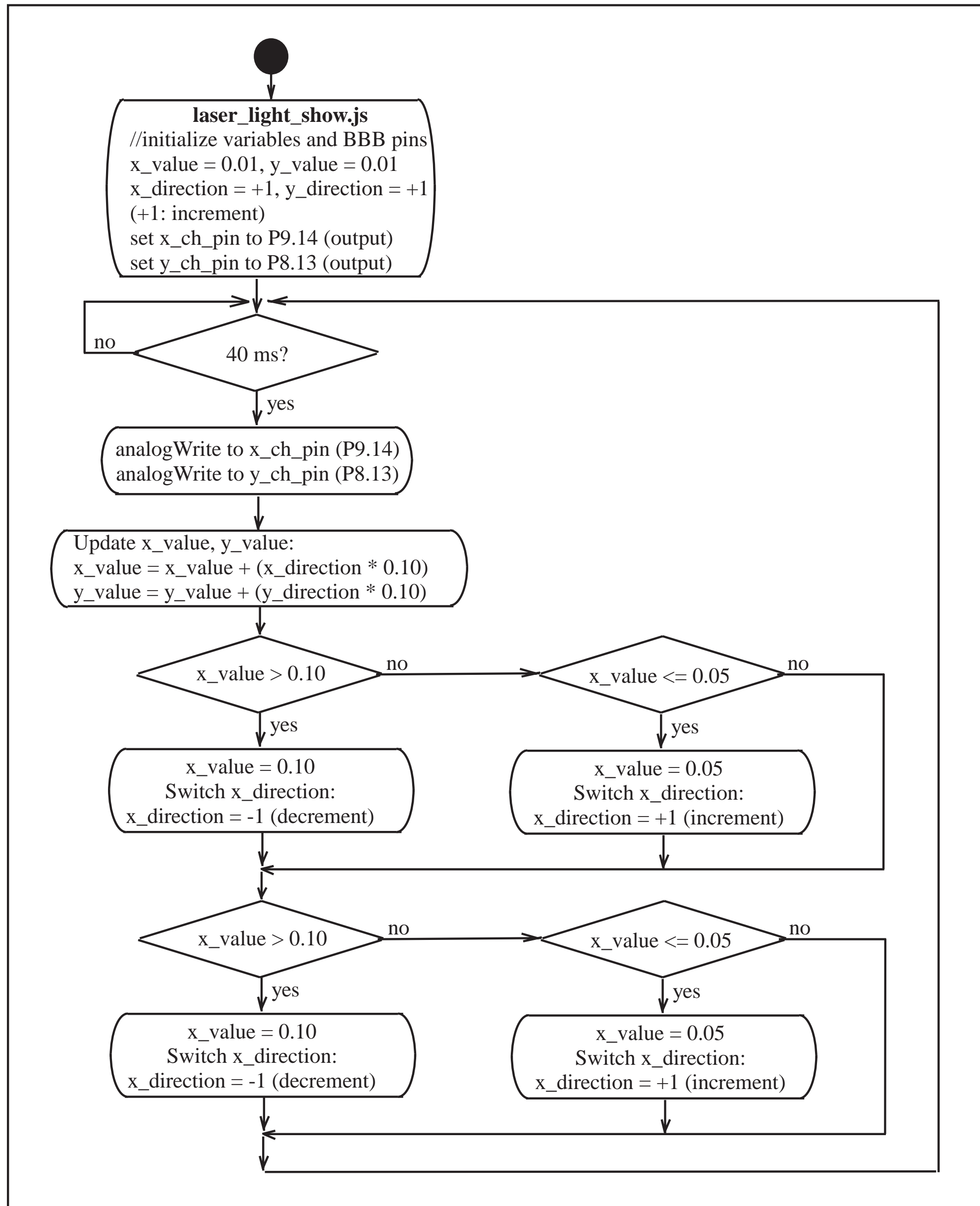


Figure 7.7: Laser light show UML.

7.5 APPLICATION 2: ARBITRARY WAVEFORM GENERATOR

The BeagleBone may be used as an arbitrary waveform generator using the built-in math functions of Node.js. Many common math functions are available in Node.js. In this example, we use the sine function to generate a rectified signal. Note the argument to the analogWrite function must be limited to values between 0 and 1.

```
1 //*****************************************************
2 var b = require('bonescript');
3
4 // setup starting conditions
5 var signal_value = 0.01;
6 var sine_value = 0;
7 var signal_pin = "P9_14";
8
9 // configure pin
10 b.pinMode(signal_pin, b.OUTPUT);
11
12 // call function to update signal every 1 ms
13 setInterval(signal, 1);
14
15 // function to update signal
16 function signal() {
17  b.analogWrite(signal_pin, sine_value);
18  signal_value = signal_value + 0.01;
19  if(signal_value >= 1.0) {signal_value = 0; }
20  sine_value = Math.abs(Math.sin(signal_value * 2 * 3.14));
21 }
22
23 //*****************************************************
```

7.6 APPLICATION 3: ROBOT ARM

Robotics are used in a wide variety of educational and industrial applications. In this section, we adapt an inexpensive (US $50) off-the-shelf robotic arm for control by BeagleBone Black. The Robotic Arm-Edge is a kit manufactured by OWI Inc. It is readily available online from a number of distributors. The arm has five degrees of freedom and may be assembled in several hours. The robot arm is controlled using five switches housed within the Wired Control Box, as shown in Figure 7.8.

To adapt the arm for control by BeagleBone Black an interface circuit is required, as shown in Figure 7.9. A total of ten digital input/output control lines from BeagleBone Black control the five robot arm motors (M1: gripper, M2: wrist, M3: elbow, M4: base, and M5: base rotation). Each motor requires a forward and reverse control signal. A logic one to either of the forward

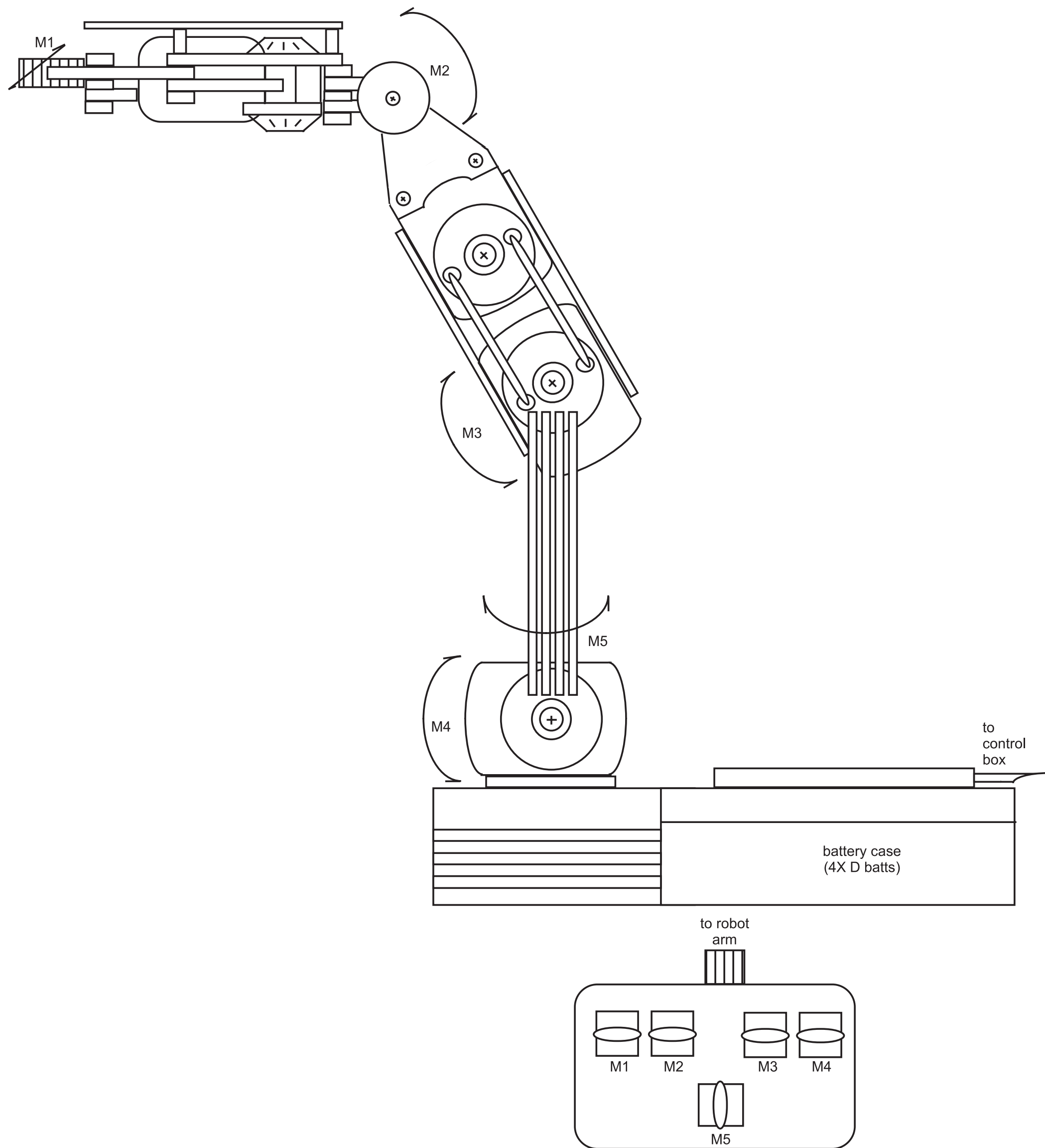


Figure 7.8: OWI Inc. Robotic Arm-Edge kit [http://www.owirobot.com].

or reverse inputs will cause the motor to move in the corresponding direction while the signal is asserted. The motor does not move when both signals are at logic zero.

Each digital signal from BeagleBone Black (at 3.3 VDC) is fed to an LM324 operational amplifier configured as a threshold detector. When a logic one (3.3 VDC) is provided to the threshold detector from BeagleBone Black, the LM324 saturates at the supply voltage less one volt (8 VDC). The output voltage from the LM324 energizes the coil of the SPST-NO (single pole, single throw, normally open) reed relay (Radio Shack #275-232). The energized coil closes the switch contacts which allow the motor to rotate. The motor supply (±) 3 VDC is provided by three D batteries which reside in the base of the robot arm. An interface circuit, as shown in Figure 7.9, is required for each of the five motors (M1-M5) of the robot arm.

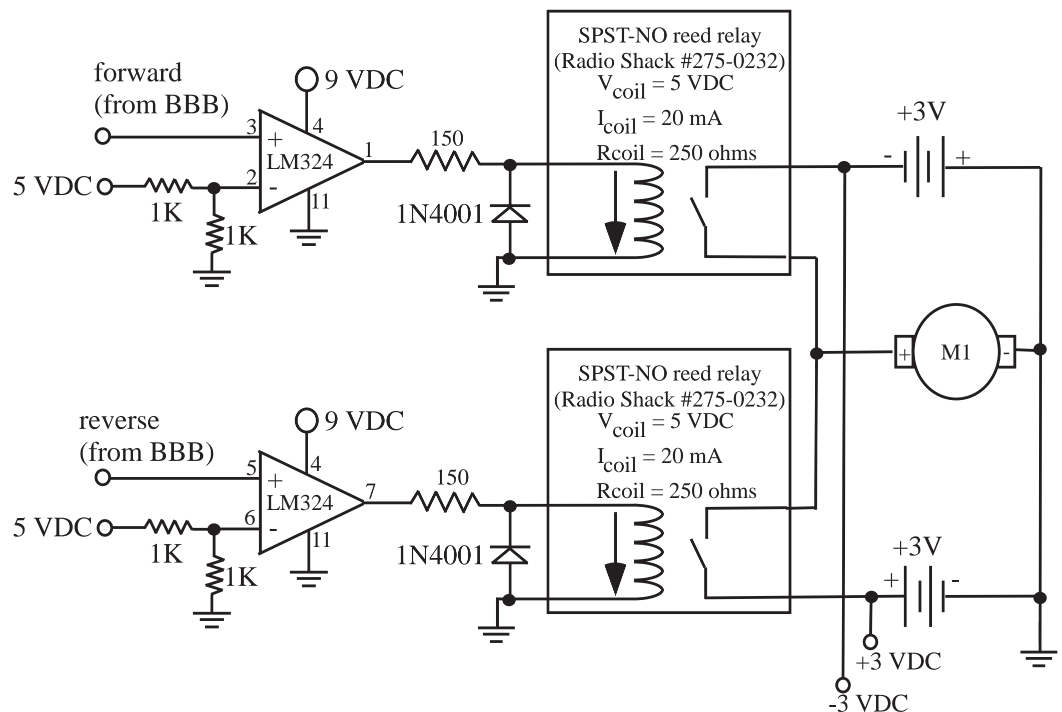


Figure 7.9: Robot arm interface circuit.

The robot arm may be programmed to complete a variety of tasks. A basic Bonescript program is provided below to sequentially move each of the motors forward and then reverse for 200 ms. The cycle is repeated every 2.1 s. The variable "steps" consists of an array of functions. The functions are sequentially called within the function "mysteps."

```
1 //****************************************************************
2 //robot_arm.js
```

```
3 //***********************************************************************
4
5 var b = require('bonescript');
6
7
8 var M1_forward = "P9_11";    //M1: gripper
9 var M1_reverse = "P9_12";
10 var M2_forward = "P9_13";   //M2: wrist motion
11 var M2_reverse = "P9_14";
12 var M3_forward = "P9_15";   //M3: elbow motion
13 var M3_reverse = "P9_16";
14 var M4_forward = "P9_17";   //M4: base motion
15 var M4_reverse = "P9_18";
16 var M5_forward = "P9_21";   //M5: base rotation
17 var M5_reverse = "P9_22";
18
19 b.pinMode(M1_forward, b.OUTPUT);
20 b.pinMode(M1_reverse, b.OUTPUT);
21 b.pinMode(M2_forward, b.OUTPUT);
22 b.pinMode(M2_reverse, b.OUTPUT);
23 b.pinMode(M3_forward, b.OUTPUT);
24 b.pinMode(M3_reverse, b.OUTPUT);
25 b.pinMode(M4_forward, b.OUTPUT);
26 b.pinMode(M4_reverse, b.OUTPUT);
27 b.pinMode(M5_forward, b.OUTPUT);
28 b.pinMode(M5_reverse, b.OUTPUT);
29
30 setInterval(loop, 2100);
31
32 function loop()
33 {
34 mysteps(done);
35 }
36
37 function done()
38 {
39 console.log("done");
40 }
41
42 function mysteps(callback)
43 {
44 //Provide a list of functions to call
45
46 var steps =
47 [
48                                   //M1: gripper motor forward
49 function(){console.log("i = " +i); b.digitalWrite(M5_reverse, b.LOW);
50           b.digitalWrite(M1_forward, b.HIGH); setTimeout(next, 200);};
```

```
51
52                                      //M1: gripper motor reverse
53 function(){console.log("i = " +i); b.digitalWrite(M1_forward, b.LOW);
54            b.digitalWrite(M1_reverse, b.HIGH); setTimeout(next, 200);};
55
56
57                                      //M2: wrist motor forward
58 function(){console.log("i = " +i); b.digitalWrite(M1_reverse, b.LOW);
59            b.digitalWrite(M2_forward, b.HIGH); setTimeout(next, 200);};
60
61                                      //M2: wrist motor reverse
62 function(){console.log("i = " +i); b.digitalWrite(M2_forward, b.LOW);
63            b.digitalWrite(M2_reverse, b.HIGH); setTimeout(next, 200);};
64
65
66                                      //M3: elbow motor forward
67 function(){console.log("i = " +i); b.digitalWrite(M2_reverse, b.LOW);
68            b.digitalWrite(M3_forward, b.HIGH); setTimeout(next, 200);};
69
70                                      //M3: elbow motor reverse
71 function(){console.log("i = " +i); b.digitalWrite(M3_forward, b.LOW);
72            b.digitalWrite(M3_reverse, b.HIGH); setTimeout(next, 200);};
73
74
75                                      //M4: base motor forward
76 function(){console.log("i = " +i); b.digitalWrite(M3_reverse, b.LOW);
77            b.digitalWrite(M4_forward, b.HIGH); setTimeout(next, 200);};
78
79                                      //M4: base motor reverse
80 function(){console.log("i = " +i); b.digitalWrite(M4_forward, b.LOW);
81            b.digitalWrite(M4_reverse, b.HIGH); setTimeout(next, 200);};
82
83
84                                      //M5: rotation motor forward
85 function(){console.log("i = " +i); b.digitalWrite(M4_reverse, b.LOW);
86            b.digitalWrite(M5_forward, b.HIGH); setTimeout(next, 200);};
87
88                                      //M5: rotation motor reverse
89 function(){console.log("i = " +i); b.digitalWrite(M5_forward, b.LOW);
90            b.digitalWrite(M5_reverse, b.HIGH); setTimeout(next, 200);};
91
92 function(){callback();}
93 ];
94
95 //Start at 0
96 var i = 0;
97
98 console.log("i = " +i);
```

```
 99 next(); //Call the first function
100
101 //Nested helper function to call the next function in 'steps'
102 function next()
103 {
104 i++
105 steps[i-1]();
106 }
107
108 //******************************************************************
```

7.7 APPLICATION 4: WEATHER STATION IN BONESCRIPT

In the early chapters of this book, we examined the Bonescript environment as a user-friendly tool to rapidly employ BeagleBone features right out of the box. In this section, we demonstrate how Bonescript may be used as a rapid prototyping tool to develop complex, processor-based systems employing multiple BeagleBone subsystems. In this first example we develop a basic weather station to sense wind direction and ambient temperature. The weather station may be easily expanded with other sensors (wind speed, humidity, etc.) The sensed values will be displayed on an LCD in Fahrenheit. The wind direction will also be displayed on a multi-LED array.

7.7.1 REQUIREMENTS

The requirements for this system include the following.

- Design a weather station to sense wind direction and ambient temperature.
- Sensed wind direction and temperature will be displayed on an LCD.
- Sensed temperature will be displayed in the Fahrenheit temperature scale.
- Wind direction will be displayed on a multi-LED array.

7.7.2 STRUCTURE CHART

To begin the design process a structure chart is used to partition the system into definable pieces. We employ a top-down design/bottom-up implementation approach. The structure chart for the weather station is provided in Figure 7.10. The system is partitioned until the lowest level of the structure chart contains "doable" pieces of hardware components or software functions. Data flow is shown on the structure chart as directed arrows. For example, raw data on wind direction sensed by the weather vane is processed as an input to the analog-to-digital converter (ADC), whereas a logic signal to active an LED to indicate wind direction is an output.

The main BeagleBone subsystem needed for this project is the ADC system to convert the analog voltage from the LM34 temperature sensor and weather vane into digital signals. Also,

a number of general purpose input/output pins will interface to the wind direction display. The wind direction display consists of a multi-LED array. Each LED has a 2.1 VDC voltage drop and a current of 10 mA.

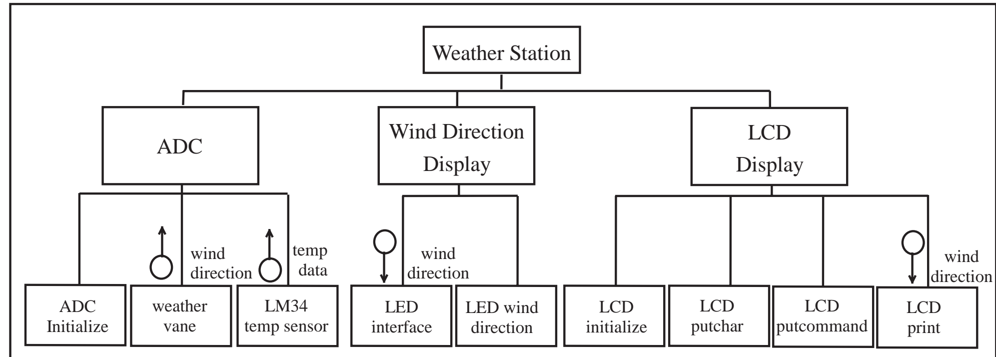


Figure 7.10: Weather station structure chart.

7.7.3 CIRCUIT DIAGRAM

The circuit diagram for the weather station is provided in Figure 7.11. The weather station is equipped with two input sensors: the LM34 to measure temperature and the weather vane to measure wind direction. Both of the sensors provide an analog output that is fed to BeagleBone's ADC system. The LM34 provides 10 mV output per degree Fahrenheit. The weather vane provides 0–1.8 VDC for 360° of vane rotation. The weather vane must be oriented to a known direction with the output voltage at this direction noted. We assume that 0 VDC corresponds to North and the voltage increases as the vane rotates clockwise to the East. The vane output voltage continues to increase until North is again reached at 1.8 VDC and then rolls over back to 0 volts. All other directions are derived from this reference point, as shown in Figure 7.12. An LCD is connected to BeagleBone, as shown in Figure 7.11. This is the same LCD interface provided earlier in this book.

7.7.4 UML ACTIVITY DIAGRAMS

The UML activity diagram for the program is provided in Figure 7.13. After initializing the subsystems, the program enters a continuous loop where temperature and wind direction are sensed and displayed on the LCD and the LED display. The system then enters a delay. The delay value is set to determine how often the temperature and wind direction parameters are updated.

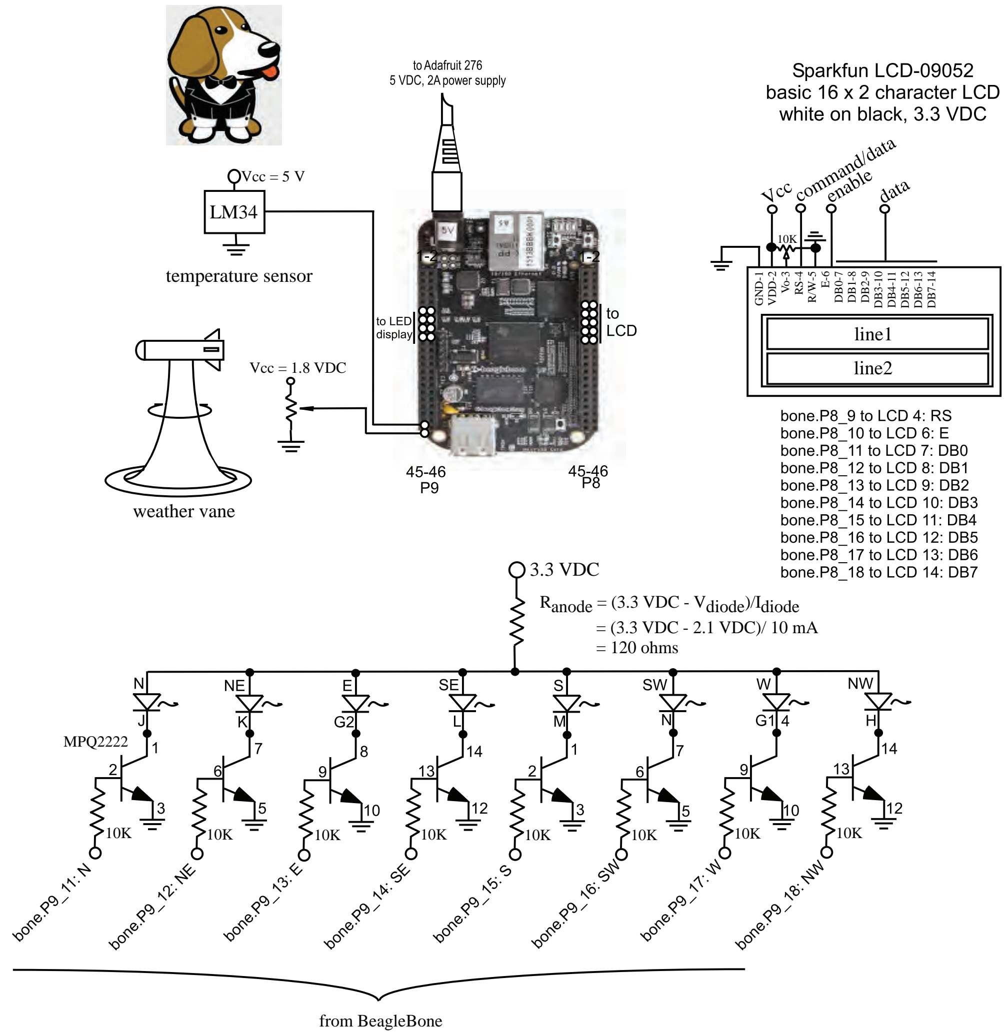


Figure 7.11: Circuit diagram for weather station. (Illustrations used with permission of Texas Instruments (www.TI.com).)

7.7.5 BONESCRIPT CODE

In this example we use the Bonescript user environment to rapidly code the control algorithm for the weather station. We use examples provided earlier in the book as building blocks to rapidly construct the code. We provide the majority of the code. The code to convert the reading from the

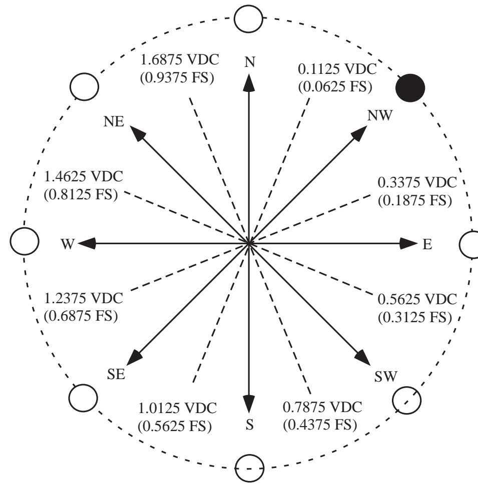


Figure 7.12: Weather vane output voltage as shown as the actual value and the normalized full-scale (FS) value.

LM34 temperature sensor and display its value and wind direction are left as an end-of-chapter exercise.

```
1 //
      ***********************************************************************
2 var b = require ('bonescript');
3
4 //sensor pin configuration
5 var wx_vane = ``P9_39'';                //weather vane
6 var temp_sen= ``P9_37'';                //temperature sensor
7 var wx_vane_value;
8 var temp_sen_value;
9
10 //wind direction LED
11 var LED_N    = ``P9_11'';               //N:  LED segment J
12 var LED_NE   = ``P9_12'';               //NE: LED segment K
13 var LED_E    = ``P9_13'';               //E:  LED segment G2
14 var LED_SE   = ``P9_14'';               //SE: LED segment L
15 var LED_S    = ``P9_15'';               //S:  LED segment M
16 var LED_SW   = ``P9_16'';               //SW: LED segment N
17 var LED_W    = ``P9_17'';               //W:  LED segment G1
18 var LED_NW   = ``P9_18'';               //NW: LED segment H
19
20 //LCD pin configuration
21 var LCD_RS   = ``P8_9'';                //LCD Register Set (RS) control
```

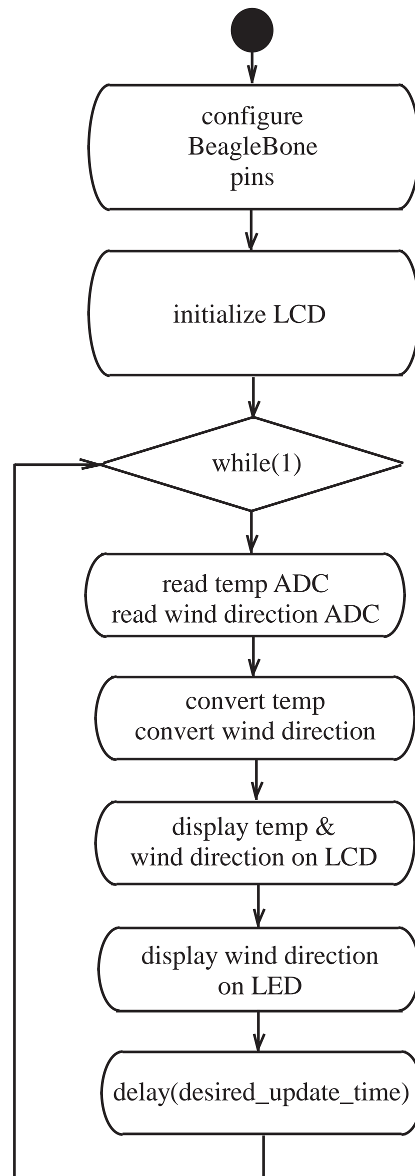


Figure 7.13: Weather station UML activity diagram.

```
22 var LCD_E   = ''P8_10'';          //LCD Enable (E) control
23 var LCD_DB0 = ''P8_11'';          //LCD Data line DB0
24 var LCD_DB1 = ''P8_12'';          //LCD Data line DB1
25 var LCD_DB2 = ''P8_13'';          //LCD Data line DB2
26 var LCD_DB3 = ''P8_14'';          //LCD Data line DB3
27 var LCD_DB4 = ''P8_15'';          //LCD Data line DB4
28 var LCD_DB5 = ''P8_16'';          //LCD Data line DB5
29 var LCD_DB6 = ''P8_17'';          //LCD Data line DB6
30 var LCD_DB7 = ''P8_18'';          //LCD Data line DB7
31
32 //wind direction pins
33 b.pinMode(LED_N,  b.OUTPUT);
34 b.pinMode(LED_NE, b.OUTPUT);
35 b.pinMode(LED_E,  b.OUTPUT);
36 b.pinMode(LED_SE, b.OUTPUT);
37 b.pinMode(LED_S,  b.OUTPUT);
38 b.pinMode(LED_SW, b.OUTPUT);
39 b.pinMode(LED_W,  b.OUTPUT);
40 b.pinMode(LED_NW, b.OUTPUT);
41
42 //LCD direction pins
43 b.pinMode(LCD_RS,  b.OUTPUT);          //set pin to digital output
44 b.pinMode(LCD_E,   b.OUTPUT);          //set pin to digital output
45 b.pinMode(LCD_DB0, b.OUTPUT);          //set pin to digital output
46 b.pinMode(LCD_DB1, b.OUTPUT);          //set pin to digital output
47 b.pinMode(LCD_DB2, b.OUTPUT);          //set pin to digital output
48 b.pinMode(LCD_DB3, b.OUTPUT);          //set pin to digital output
49 b.pinMode(LCD_DB4, b.OUTPUT);          //set pin to digital output
50 b.pinMode(LCD_DB5, b.OUTPUT);          //set pin to digital output
51 b.pinMode(LCD_DB6, b.OUTPUT);          //set pin to digital output
52 b.pinMode(LCD_DB7, b.OUTPUT);          //set pin to digital output
53 LCD_init();                            //call LCD initialize
54
55 setInterval(updateWeather, 100);
56
57 function updateWeather()
58   {
59   clear_LEDs();
60
61   //Read weather vane and temperature sensors
62   wx_vane_value  = b.analogRead(wx_vane);
63   temp_sen_value = b.analogRead(temp_sen);
64
65   //Calculate temperature
66
67   //North
68   if((wx_vane_value > 0.9375)||(wx_vane_value <= 0.0625))
69     {
```

```
70      //illuminate N LED
71      b.digitalWrite(LED_N, b.HIGH);
72      }
73
74   //Northeast
75   else if((wx_vane_value > 0.0625)&&(wx_vane_value <= 0.1875))
76     {
77     //illuminate NE LED
78     b.digitalWrite(LED_NE, b.HIGH);
79     }
80
81   //East
82   else if((wx_vane_value > 0.1875)&&(wx_vane_value <= 0.3125))
83     {
84     //illuminate E LED
85     b.digitalWrite(LED_E, b.HIGH);
86     }
87
88   //Southeast
89   else if((wx_vane_value > 0.3125)&&(wx_vane_value <= 0.4375))
90     {
91     //illuminate SE LED
92     b.digitalWrite(LED_SE, b.HIGH);
93     }
94
95   //South
96   else if((wx_vane_value > 0.4375)&&(wx_vane_value <= 0.5625))
97     {
98     //illuminate S LED
99     b.digitalWrite(LED_S, b.HIGH);
100    }
101
102  //Southwest
103  else if((wx_vane_value > 0.5625)&&(wx_vane_value <= 0.6875))
104    {
105    //illuminate SW LED
106    b.digitalWrite(LED_SW, b.HIGH);
107    }
108
109  //West
110  else if((wx_vane_value > 0.6875)&&(wx_vane_value <= 0.8125))
111    {
112    //illuminate W LED
113    b.digitalWrite(LED_W, b.HIGH)
114    }
115
116  //NE
117  else
```

```
118        {
119        //illuminate NE LED
120        b.digitalWrite(LED_NE, b.HIGH)
121        }
122      }
123
124  //***********************************************************************
125  //clear_LEDs
126  //***********************************************************************
127
128  function clear_LEDs() {
129    //reset LEDs
130    b.digitalWrite(LED_N,  b.LOW);
131    b.digitalWrite(LED_NE, b.LOW);
132    b.digitalWrite(LED_E,  b.LOW);
133    b.digitalWrite(LED_SE, b.LOW);
134    b.digitalWrite(LED_S,  b.LOW);
135    b.digitalWrite(LED_SW, b.LOW);
136    b.digitalWrite(LED_W,  b.LOW);
137    b.digitalWrite(LED_NW, b.LOW);
138    }
139
140  //***********************************************************************
141  //LCD_print
142  //***********************************************************************
143
144  function LCD_print(line, message, callback)
145  {
146  var i = 0;
147
148  if(line == 1)
149    {
150    LCD_putcommand(0x80, writeNextCharacter);//print to LCD line 1
151    }
152  else
153    {
154    LCD_putcommand(0xc0, writeNextCharacter);//print to LCD line 2
155    }
156
157  function writeNextCharacter()
158    {
159    //if we already printed the last character, stop and callback
160    if(i == message.length)
161      {
162      if(callback) callback();
163      return;
164      }
165
```

```
166   //get the next character to print
167   var chr = message.substring(i, i+1);
168   i++;
169
170   //print it using LCD_putchar and come back again when done
171   LCD_putchar(chr, writeNextCharacter);
172   }
173 }
174
175 //*****************************************************************
176 //LCD_init
177 //*****************************************************************
178
179 function LCD_init(callback)
180 {
181 //LCD Enable (E) pin low
182 b.digitalWrite(LCD_E, b.LOW);
183
184 //Start at the beginning of the list of steps to perform
185 var i = 0;
186
187 //List of steps to perform
188 var steps =
189   [
190   function(){ setTimeout(next, 15); },           //delay 15ms
191   function(){ LCD_putcommand(0x38, next); },     //set for 8-bit
        operation
192   function(){ setTimeout(next, 5); },            //delay 5ms
193   function(){ LCD_putcommand(0x38, next); },     //set for 8-bit
        operation
194   function(){ LCD_putcommand(0x38, next); },     //set for 5 x 7
        character
195   function(){ LCD_putcommand(0x0E, next); },     //display on
196   function(){ LCD_putcommand(0x01, next); },     //display clear
197   function(){ LCD_putcommand(0x06, next); },     //entry mode set
198   function(){ LCD_putcommand(0x00, next); },     //clear display, cursor
         home
199   function(){ LCD_putcommand(0x00, callback); } //clear display, cursor
         home
200   ];
201
202 next(); //Execute the first step
203
204 //Function for executing the next step
205 function next()
206   {
207   i++;
208   steps[i-1]();
```

```
209     }
210 }
211
212 //***************************************************************
213 //LCD_putcommand
214 //***************************************************************
215
216 function LCD_putcommand(cmd, callback)
217 {
218 //parse command variable into individual bits for output
219 //to LCD
220 if((cmd & 0x0080)== 0x0080) b.digitalWrite(LCD_DB7, b.HIGH);
221     else b.digitalWrite(LCD_DB7, b.LOW);
222 if((cmd & 0x0040)== 0x0040) b.digitalWrite(LCD_DB6, b.HIGH);
223     else b.digitalWrite(LCD_DB6, b.LOW);
224 if((cmd & 0x0020)== 0x0020) b.digitalWrite(LCD_DB5, b.HIGH);
225     else b.digitalWrite(LCD_DB5, b.LOW);
226 if((cmd & 0x0010)== 0x0010) b.digitalWrite(LCD_DB4, b.HIGH);
227     else b.digitalWrite(LCD_DB4, b.LOW);
228 if((cmd & 0x0008)== 0x0008) b.digitalWrite(LCD_DB3, b.HIGH);
229     else b.digitalWrite(LCD_DB3, b.LOW);
230 if((cmd & 0x0004)== 0x0004) b.digitalWrite(LCD_DB2, b.HIGH);
231     else b.digitalWrite(LCD_DB2, b.LOW);
232 if((cmd & 0x0002)== 0x0002) b.digitalWrite(LCD_DB1, b.HIGH);
233     else b.digitalWrite(LCD_DB1, b.LOW);
234 if((cmd & 0x0001)== 0x0001) b.digitalWrite(LCD_DB0, b.HIGH);
235     else b.digitalWrite(LCD_DB0, b.LOW);
236
237 //LCD Register Set (RS) to logic zero for command input
238 b.digitalWrite(LCD_RS, b.LOW);
239 //LCD Enable (E) pin high
240 b.digitalWrite(LCD_E, b.HIGH);
241
242 //End the write after 1ms
243 setTimeout(endWrite, 1);
244
245 function endWrite()
246     {
247     //LCD Enable (E) pin low
248     b.digitalWrite(LCD_E, b.LOW);
249     //delay 1ms before calling 'callback'
250     setTimeout(callback, 1);
251     }
252 }
253
254 //***************************************************************
255 //LCD_putchar
256 //***************************************************************
```

```
257
258 function LCD_putchar(chr1, callback)
259 {
260 //Convert chr1 variable to UNICODE (ASCII)
261 var chr = chr1.toString().charCodeAt(0);
262
263 //parse character variable into individual bits for output
264 //to LCD
265 if((chr & 0x0080)== 0x0080) b.digitalWrite(LCD_DB7, b.HIGH);
266   else b.digitalWrite(LCD_DB7, b.LOW);
267 if((chr & 0x0040)== 0x0040) b.digitalWrite(LCD_DB6, b.HIGH);
268   else b.digitalWrite(LCD_DB6, b.LOW);
269 if((chr & 0x0020)== 0x0020) b.digitalWrite(LCD_DB5, b.HIGH);
270   else b.digitalWrite(LCD_DB5, b.LOW);
271 if((chr & 0x0010)== 0x0010) b.digitalWrite(LCD_DB4, b.HIGH);
272   else b.digitalWrite(LCD_DB4, b.LOW);
273 if((chr & 0x0008)== 0x0008) b.digitalWrite(LCD_DB3, b.HIGH);
274   else b.digitalWrite(LCD_DB3, b.LOW);
275 if((chr & 0x0004)== 0x0004) b.digitalWrite(LCD_DB2, b.HIGH);
276   else b.digitalWrite(LCD_DB2, b.LOW);
277 if((chr & 0x0002)== 0x0002) b.digitalWrite(LCD_DB1, b.HIGH);
278   else b.digitalWrite(LCD_DB1, b.LOW);
279 if((chr & 0x0001)== 0x0001) b.digitalWrite(LCD_DB0, b.HIGH);
280   else b.digitalWrite(LCD_DB0, b.LOW);
281
282 //LCD Register Set (RS) to logic one for character input
283 b.digitalWrite(LCD_RS, b.HIGH);
284 //LCD Enable (E) pin high
285 b.digitalWrite(LCD_E, b.HIGH);
286
287 //End the write after 1ms
288 setTimeout(endWrite, 1);
289
290 function endWrite()
291   {
292   //LCD Enable (E) pin low and call scheduleCallback when done
293   b.digitalWrite(LCD_E, b.LOW);
294   //delay 1ms before calling 'callback'
295   setTimeout(callback, 1);
296   }
297 }
298
299 //*********************************************************************
```

7.8 APPLICATION 5: SPEAK & SPELL IN C

Speak & Spell is an educational toy developed by Texas Instruments in the mid-1970's. It was developed by the engineering team of Gene Franz, Richard Wiggins, Paul Breedlove, and Larry Branntingham, pictured in Figure 7.14. The Speak & Spell consists of a keyboard, display, speech synthesizer, and a slot to insert game modules. A series of educational games teach spelling skills, letter recognition skills, and memory aids are available as plug in cartridges [www.TI.com].

Figure 7.14: Speak & Spell design team from left to right: Gene Franz, Richard Wiggins, Paul Breedlove, and Larry Branntingham [www.TI.com].

In this project we design a BeagleBone based Speak & Spell. We use a small keyboard (www.adafruit.com) connected to BeagleBone via the USB port. We also use the Circuitco 7 inch LCD display (BeagleBone LCD7). For speech synthesis we use the SP0-512 text to speech chip (www.speechchips.com). The SP0-512 accepts UART compatible serial text stream. The text stream is converted to phoneme codes used to generate an audio output. The chip requires a 9600 Baud bit stream with no parity, 8 data bits, and a stop bit. Additional information on the chip and its features are available at www.speechchips.com. The BeagleBone version of Speak &

Spell is shown in Figure 7.15 and the support circuit for the SP0-512 is provided in Figure 7.16. As an alternative, one could employ speech synthesis software coupled to a sound card.

Figure 7.15: BeagleBone-based Speak & Spell. (Photo courtesy of Barrett, 2013.)

7.8.1 BEAGLEBONE C CODE

The structure chart and UML activity diagram for Speak & Spell is provided in Figure 7.17. A basic algorithm is provided below to accept input from the keyboard, output it on the LCD7 display and pass it to the SP0-512 speech synthesizer chip. This algorithm may form the basis for a number of Speak & Spell educational games.

As a friendly reminder, before executing the sample code insure the SLOTS, PINS, and the appropriate device tree overlays have been loaded. The SLOTS and PINS are loaded using:

```
# export SLOTS=/sys/devices/bone_capemgr.9/slots
# export PINS=/sys/kernel/debug/pinctrl/44e10800.pinmux/pins
```

To load the device tree overlay for UART channel 1, use the following commands:

Figure 7.16: Speech synthesis support circuit [www.speechchips.com]. (Illustrations used with permission of Texas Instruments (www.TI.com).)

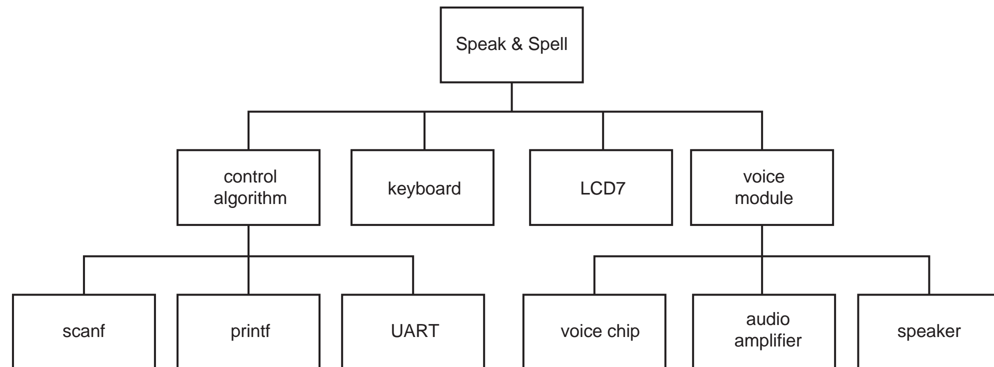


Figure 7.17: Speak & Spell structure chart.

```
# cd /lib/firmware
/lib/firmware# sudo su -c ``echo BB-UART1 > $SLOTS''
```

```
1 //*****************************************************************
2 //uart1.c - configures BeagleBone uart1 for tranmission and 9600 Baud
3 //and repeatedly sends the character G via uart1 tx pin (P9, 24)
4 //
5 //Note: Before executing the sample code insure the SLOTS, PINS, and
6 //the appropriate device tree overlays have been loaded.
7 //*****************************************************************
8
9
10 \begin{lstlisting}
11 //*****************************************************************
12 //sns.c - Speak and Spell
13 //  - prompts user for input
14 //  - prints input to screen
15 //  - provides spoken input via speech synthesis chip connected
16 //    to uart1
17 //  - configures BeagleBone uart1 for transmission and 9600 Baud
18 //
19 //Note: Before executing the sample code insure the SLOTS, PINS, and
20 //the appropriate device tree overlays have been loaded.
21 //*****************************************************************
22
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <stddef.h>
26 #include <time.h>
27 #include <termios.h>
28 #include <fcntl.h>
```

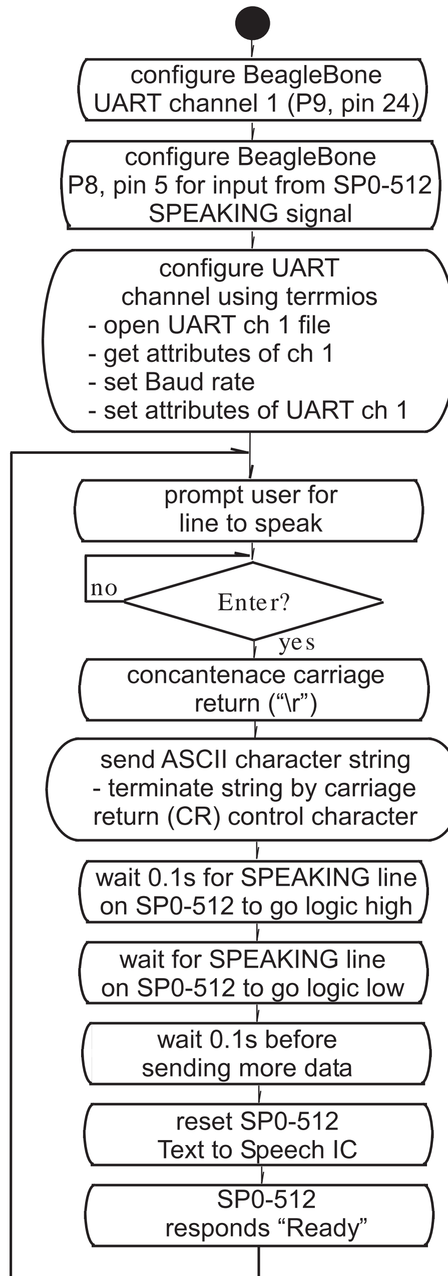


Figure 7.18: Speak & Spell UML activity diagram.

```
29 #include <unistd.h>
30 #include <sys/types.h>
31 #include <string.h>
32
33 int main(void)
34 {
35 //define file handle for uart1
36 FILE *ofp_uart1_tx, *ofp_uart1_rx;
37
38 //uart1 configuration using termios
39 struct termios uart1;
40 int fd;
41
42 //open uart1 for tx/rx, not controlling device
43 if((fd = open("/dev/ttyO1", O_RDWR | O_NOCTTY)) < 0)
44   printf("Unable to open uart1 access.\n");
45
46 //get attributes of uart1
47 if(tcgetattr(fd, &uart1) < 0)
48   printf("Could not get attributes of UART1 at ttyO1\n");
49
50 //set Baud rate
51 if(cfsetospeed(&uart1, B9600) < 0)
52   printf("Could not set baud rate\n");
53 else
54   printf("Baud rate: 9600\n");
55
56 //set attributes of uart1
57 uart1.c_iflag = 0;
58 uart1.c_oflag = 0;
59 uart1.c_lflag = 0;
60 tcsetattr(fd, TCSANOW, &uart1);
61
62 char byte_out[20];
63
64 //set ASCII character G repeatedly
65 while(1)
66   {
67   printf("Enter letter, word, statement.
68 Press [Enter].\n\n");
69   scanf("%s", byte_out);
70   printf("%s\n\n\n", byte_out);
71   write(fd, byte_out, strlen(byte_out)+1);
72   }
73
74 close(fd);
75 }
76 //*******************************************************************
```

7.9 APPLICATION 6: DAGU ROVER 5 TREADED ROBOT

In this example we control a Dagu Rover ROV5-1 robot with a C-based control system hosted on BeagleBone. The goal of the robot system is to navigate through the three-dimensional mountain maze described earlier in the book.

7.9.1 DESCRIPTION

Dagu manufactures a number of low cost educational robots and robotic arms. In this example we use the Dagu Rover ROV5-1 robot chassis. This robot is equipped with two motor driven treads. Dagu offers other robot configurations with additional motors and wheel encoders. We begin by equipping the ROV 5-1 with a plexi-glass platform, three IR sensors and a motor control interface. The robot platform is illustrated in Figures 7.19 and 7.20.

7.9.2 REQUIREMENTS

The requirements for this project are simple, the robot must autonomously navigate through the three-dimensional mountain maze without touching maze walls as quickly as possible. Furthermore, it must be capable of not getting "stuck" on the rugged terrain features.

7.9.3 CIRCUIT DIAGRAM

The circuit diagram for the robot is provided in Figure 3.14. The three IR sensors (left, middle, and right) are mounted on the leading edge of the robot to detect maze walls. The interface for the IR sensors was used earlier in the book on other projects. The sensor outputs are fed to three ADC channels (AIN0, AIN1, and AIN2). The robot motors are driven by PWM channels EHRPWM1A (P9, pin 14) and B (P9, pin 16). BeagleBone is interfaced to the motors via a Darlington transistor (TIP 120) with enough drive capability to handle the maximum current requirements of the motor. A 330 ohm resistor is used to limit base current to 5.5 mA. The resulting collector current and hence motor drive current is approximately 300 mA. The robot is powered via an external 7.2 VDC power supply umbilical to conserve battery use. The 7.2 VDC supply is routed through the 5 VDC and 3.3 VDC regulator matrix, as shown in Figure 7.21.

7.9.4 STRUCTURE CHART

The structure chart for the robot project is provided in Figure 7.22. The two main systems used in this project is the PWM system to drive the motorized treads and the ADC system to read the IR sensors.

7.9.5 UML ACTIVITY DIAGRAMS

The UML activity diagram for the robot is provided in Figure 7.23. The basic algorithm is quite straight forward. The sensor values are read and PWM command signals are issued to navigate about the maze.

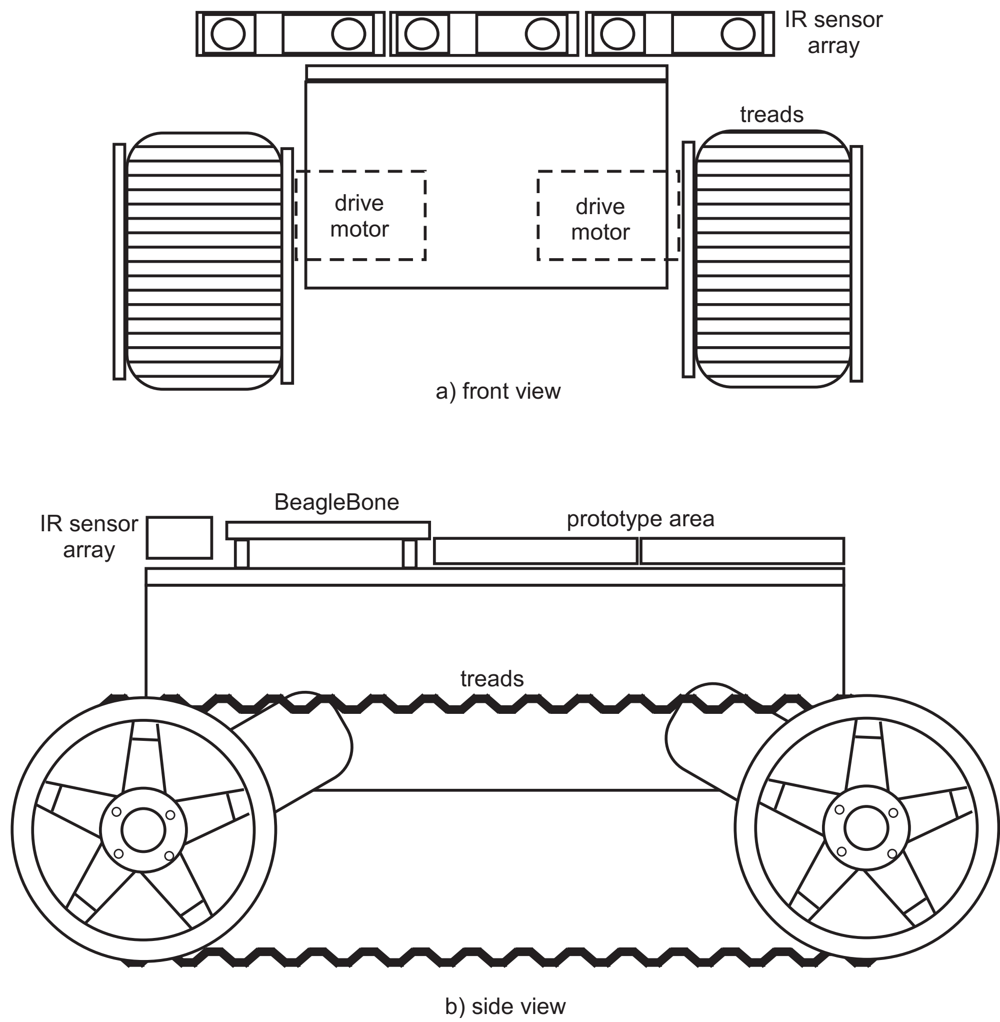
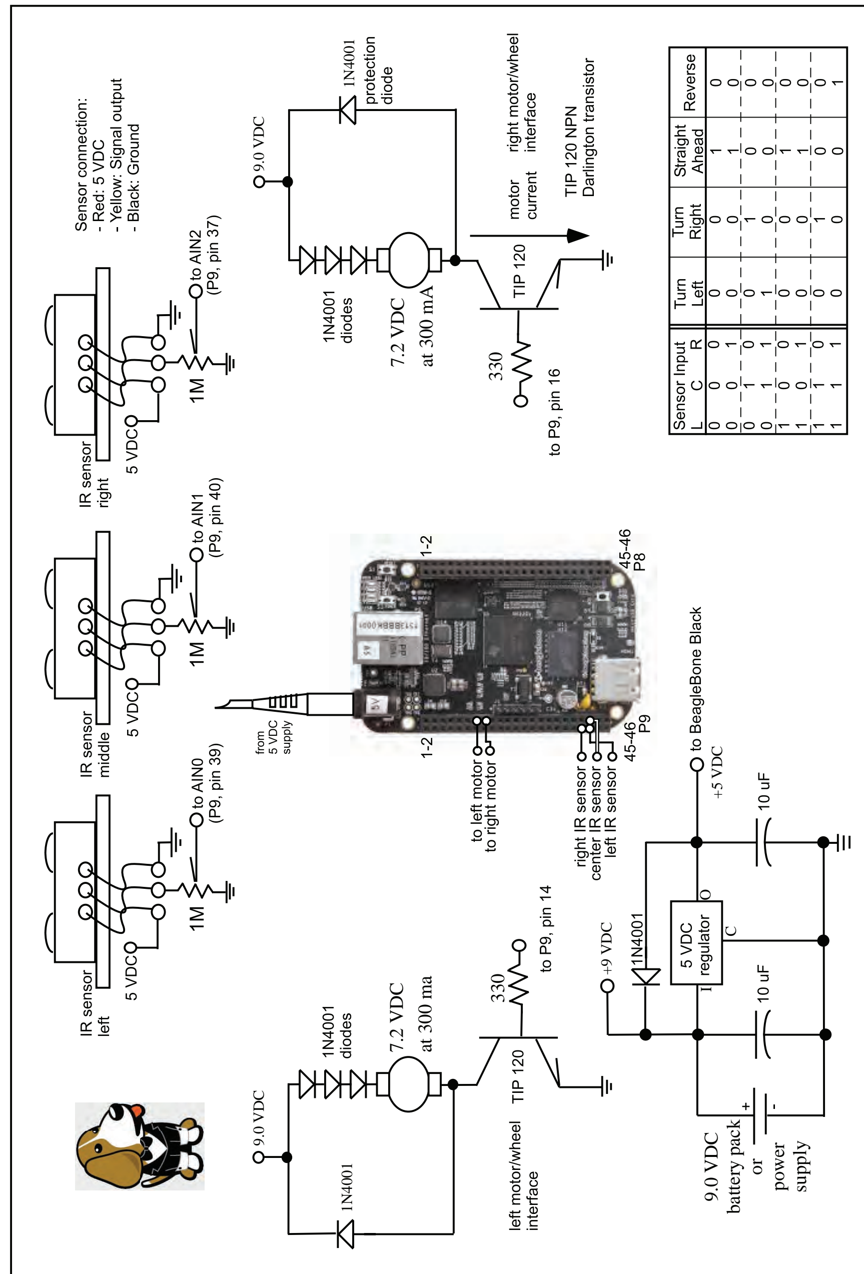


Figure 7.19: Dagu ROV5-1 robot.

Figure 7.20: Dagu ROV5-1 robot in 3D maze. (Photo courtesy of Barrett, 2013.)



Sensor Input L	C	R	Turn Left	Turn Right	Straight Ahead	Reverse
0	0	0	0	0	1	0
0	0	1	0	0	1	0
0	1	0	0	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	1	0
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	0	0	0	1

Figure 7.21: Dagu robot circuit diagram. (Illustrations used with permission of Texas Instruments. (www.TI.com).)

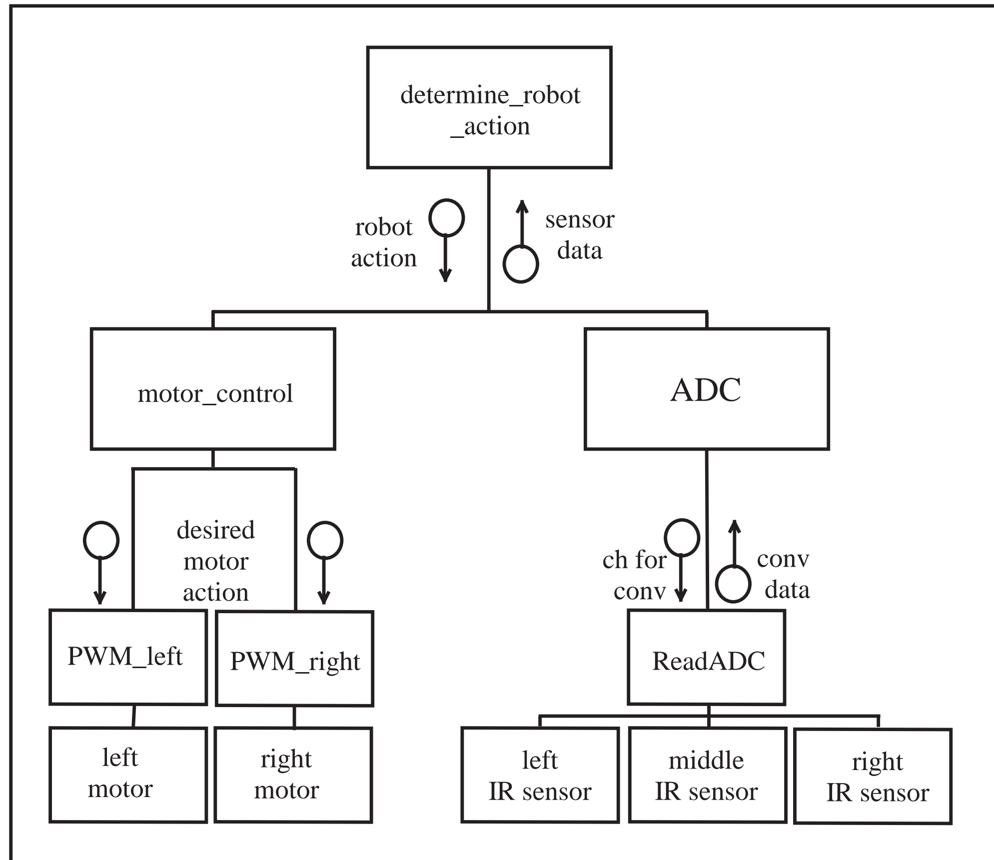


Figure 7.22: Dagu robot structure diagram.

7.9.6 BEAGLEBONE C CODE

As before we use code developed in the previous chapter as building blocks to rapidly develop the control algorithm for the Dagu robot. The printf statements are useful for algorithm development and troubleshooting. They should be commented out before testing the robot in the maze.

```
1 //*******************************************************************
2 //dagu.c
3 //
4 //
5 // Note: before executing the sample code insure the SLOTS, PINS,
6 //and the appropriate device tree overlays have been loaded.
```

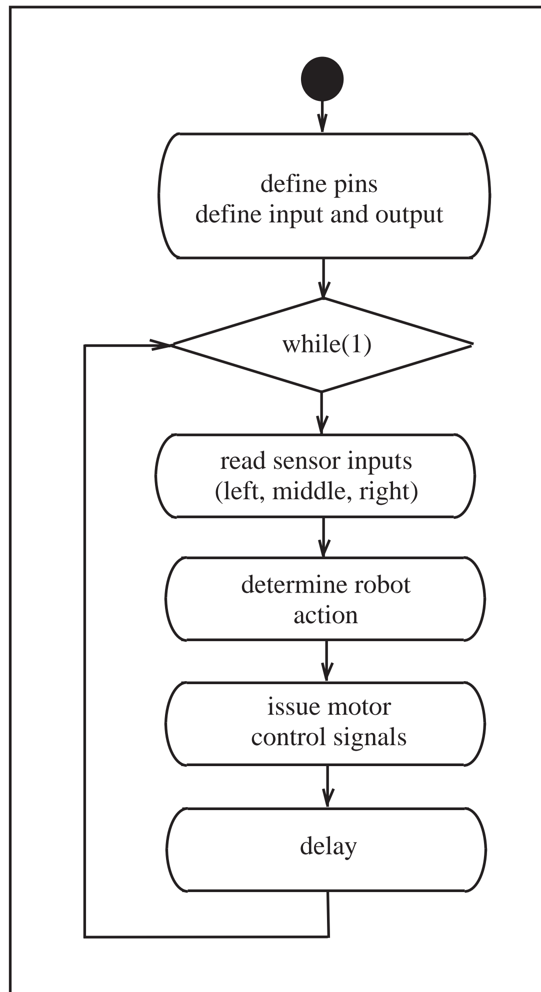


Figure 7.23: Dagu robot UML activity diagram.

```
 7 //*********************************************************************
 8
 9 #include <stdio.h>
10 #include <stddef.h>
11 #include <time.h>
12 #include <math.h>
13
14 #define   output  "out"
15 #define   input   "in"
```

```
16
17 int  main (void)
18 {
19 //wall detection threshold
20 double threshold = 1100.0;             //experimentally determined
21
22 //configure adc channels
23 //define file handles for adc related files
24 FILE *ifp_ain0, *ifp_ain1, *ifp_ain2;
25 float ain0_value, ain1_value, ain2_value;
26
27 //open adc related files for access to ain0, 1 and 2
28 ifp_ain0 = fopen("/sys/bus/iio/devices/iio:device0/in_voltage0_raw", "r
      ");
29 if (ifp_ain0 == NULL) {printf("Unable to ain0.\n");}
30
31 ifp_ain1 = fopen("/sys/bus/iio/devices/iio:device0/in_voltage1_raw", "r
      ");
32 if (ifp_ain1 == NULL) {printf("Unable to ain1.\n");}
33
34 ifp_ain2 = fopen("/sys/bus/iio/devices/iio:device0/in_voltage2_raw", "r
      ");
35 if (ifp_ain2 == NULL) {printf("Unable to ain2.\n");}
36
37 //configure pwm channels 0 and 1
38 //define file handles for channel 0 - EHRPWM1A (P9, pin 14)
39 //designated as ehrpwm.1:0
40 FILE *pwm_period0, *pwm_duty0;
41 FILE *pwm_polarity0, *pwm_run0;
42
43 //define pin variables for channel 0
44 int period0 = 500000, duty0 = 250000;
45 int polarity0 = 1, run0 = 1;
46
47 pwm_period0 = fopen("/sys/devices/ocp.3/pwm_test_P9_14.15/period", "w")
      ;
48 if(pwm_period0 == NULL) {printf("Unable to open pwm 0 period.\n");}
49 fseek(pwm_period0, 0, SEEK_SET);
50 fprintf(pwm_period0, "%d", period0);
51 fflush(pwm_period0);
52
53 pwm_duty0 = fopen("/sys/devices/ocp.3/pwm_test_P9_14.15/duty", "w");
54 if(pwm_duty0 == NULL) {printf("Unable to open pwm 0 duty cycle.\n");}
55 fseek(pwm_duty0, 0, SEEK_SET);
56 fprintf(pwm_duty0, "%d", duty0);
57 fflush(pwm_duty0);
58
```

```
59 pwm_polarity0 = fopen("/sys/devices/ocp.3/pwm_test_P9_14.15/polarity",
       "w");
60 if(pwm_polarity0 == NULL) {printf("Unable to open pwm 0 polarity.\n");}
61 fseek(pwm_polarity0, 0, SEEK_SET);
62 fprintf(pwm_polarity0, "%d", polarity0);
63 fflush(pwm_polarity0);
64
65 pwm_run0 = fopen("/sys/devices/ocp.3/pwm_test_P9_14.15/run", "w");
66 if(pwm_run0 == NULL) {printf("Unable to open pwm 0 run.\n");}
67
68 //define file handles for channel 1 - EHRPWM1B (P9, pin 16)
69 //designated as ehrpwm.1:1
70 FILE *pwm_period1, *pwm_duty1;
71 FILE *pwm_polarity1, *pwm_run1;
72
73 //define pin variables for channel 1
74 int period1 = 500000, duty1 = 250000;
75 int polarity1 = 1, run1 = 1;
76
77 pwm_period1 = fopen("/sys/devices/ocp.3/pwm_test_P9_16.15/period", "w")
       ;
78 if(pwm_period1 == NULL) {printf("Unable to open pwm 1 period.\n");}
79 fseek(pwm_period1, 0, SEEK_SET);
80 fprintf(pwm_period1, "%d", period1);
81 fflush(pwm_period1);
82
83 pwm_duty1 = fopen("/sys/devices/ocp.3/pwm_test_P9_16.15/duty", "w");
84 if(pwm_duty1 == NULL) {printf("Unable to open pwm 1 duty cycle.\n");}
85 fseek(pwm_duty1, 0, SEEK_SET);
86 fprintf(pwm_duty1, "%d", duty1);
87 fflush(pwm_duty1);
88
89 pwm_polarity1 = fopen("/sys/devices/ocp.3/pwm_test_P9_16.15/polarity",
       "w");
90 if(pwm_polarity1 == NULL) {printf("Unable to open pwm 1 polarity.\n");}
91 fseek(pwm_polarity1, 0, SEEK_SET);
92 fprintf(pwm_polarity1, "%d", polarity1);
93 fflush(pwm_polarity1);
94
95 pwm_run1 = fopen("/sys/devices/ocp.3/pwm_test_P9_16.15/run", "w");
96 if(pwm_run1 == NULL) {printf("Unable to open pwm 1 run.\n");}
97
98
99 while(1)
100   {
101   //read analog sensors
102   fseek(ifp_ain0, 0, SEEK_SET);
103   fscanf(ifp_ain0, "%f", &ain0_value);
```

```
104    printf("%f\n", ain0_value);
105
106    fseek(ifp_ain1, 0, SEEK_SET);
107    fscanf(ifp_ain1, "%f", &ain1_value);
108    printf("%f\n", ain1_value);
109
110    fseek(ifp_ain2, 0, SEEK_SET);
111    fscanf(ifp_ain2, "%f", &ain2_value);
112    printf("%f\n", ain2_value);
113
114    //implement truth to determine robot turns
115
116    //no walls present - continue straight ahead
117    if((ain0_value < threshold)&&(ain1_value < threshold)&&
118       (ain2_value < threshold))
119      {
120      run0 = 1;  run1 = 1;          //both motors on
121      fseek(pwm_run0, 0, SEEK_SET);
122      fprintf(pwm_run0, "%d", run0);
123      fflush(pwm_run0);
124
125      fseek(pwm_run1, 0, SEEK_SET);
126      fprintf(pwm_run1, "%d", run1);
127      fflush(pwm_run1);
128      }
129 /*
130    else if(...)
131      {
132
133      :
134      insert other cases
135      :
136      }
137 */
138   }
139
140 fclose(ifp_ain0);
141 fclose(ifp_ain1);
142 fclose(ifp_ain2);
143
144 fclose(pwm_period0);
145 fclose(pwm_duty0);
146 fclose(pwm_polarity0);
147 fclose(pwm_run0);
148
149 fclose(pwm_period1);
150 fclose(pwm_duty1);
151 fclose(pwm_polarity1);
```

```
152 fclose(pwm_run1);
153
154 return 1;
155 }
156 //***********************************************************************
```

7.10 APPLICATION 7: PORTABLE IMAGE PROCESSING ENGINE

Image processing is a fascinating field. It is the process of extracting useful information from an image. Image processing operations typically requires the application of a series of simple operations to an image. Each operator is sequentially swept over an image to accomplish a specific task. BeagleBone, with its 1 GHz clock speed, is ideally suited for image processing applications. In fact, BeagleBone equipped with a small keyboard and the Circuitco LCD7 liquid crystal or HDMI compatible display may be viewed as a portable image processing engine. In this section, we provide a brief introduction to image processing, the layout for a BeagleBone portable image processing engine, a brief introduction to the OpenCV image processing library and conclude with an example. The "stache cam" example uses a variety of OpenCV features to perform face recognition and place a moustache on the face. This appears to be a fun application (it is); however, it also forms the basis for a variety of assistive technology applications such as pupil tracking.

7.10.1 BRIEF INTRODUCTION TO IMAGE PROCESSING

A basic image processing system is shown in Figure 7.24. It consists of a camera equipped with a lens system, a frame grabber and a host processor to perform the image processing task. The camera captures images of objects onto a two-dimensional light sensitive array. Each element in the array is termed a picture element or pixel, as shown in Figure 7.25. The camera "snaps" an image of a scene as determined by the frame rate of the camera. Typical frames rates include 30 and 60 frames per second (fps). Other high-speed frame rates are available. Also, slower frame rates are available in high resolution cameras. Spatial resolution refers to the number of pixels within the imaging array. Although higher resolution will resolve finer object detail, it comes at the expense of increased computational cost.

Cameras are available in black and white or color. Usually, a black and white camera will register the shade or gray scale of each pixel as a single byte. A gray scale value of 0 is assigned to black while 255 is used to designate white. Shades between black and white are assigned gray scale values between 0 and 255.

Various color schemes may be used to represent an image including red-green-blue (R-G-B), hue-saturation-intensity (H-S-I), etc. They require three bytes to represent a pixel in a specific color scheme.

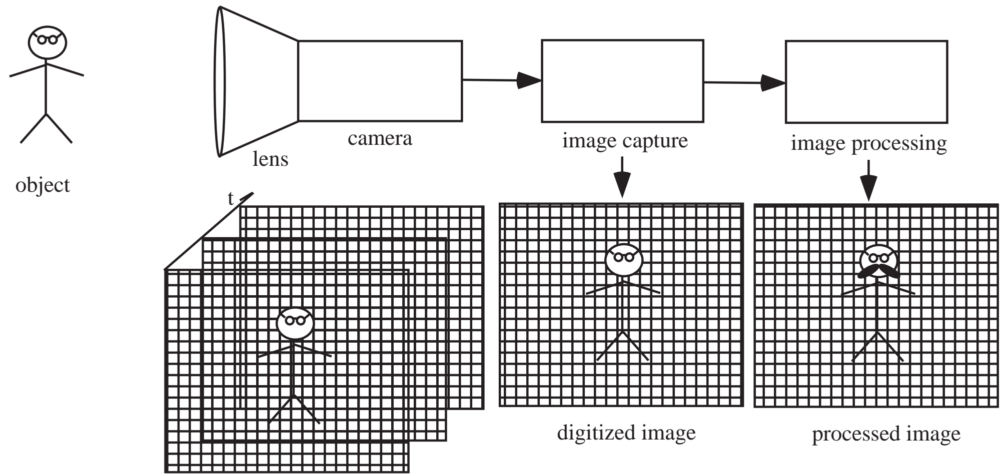


Figure 7.24: Image processing system.

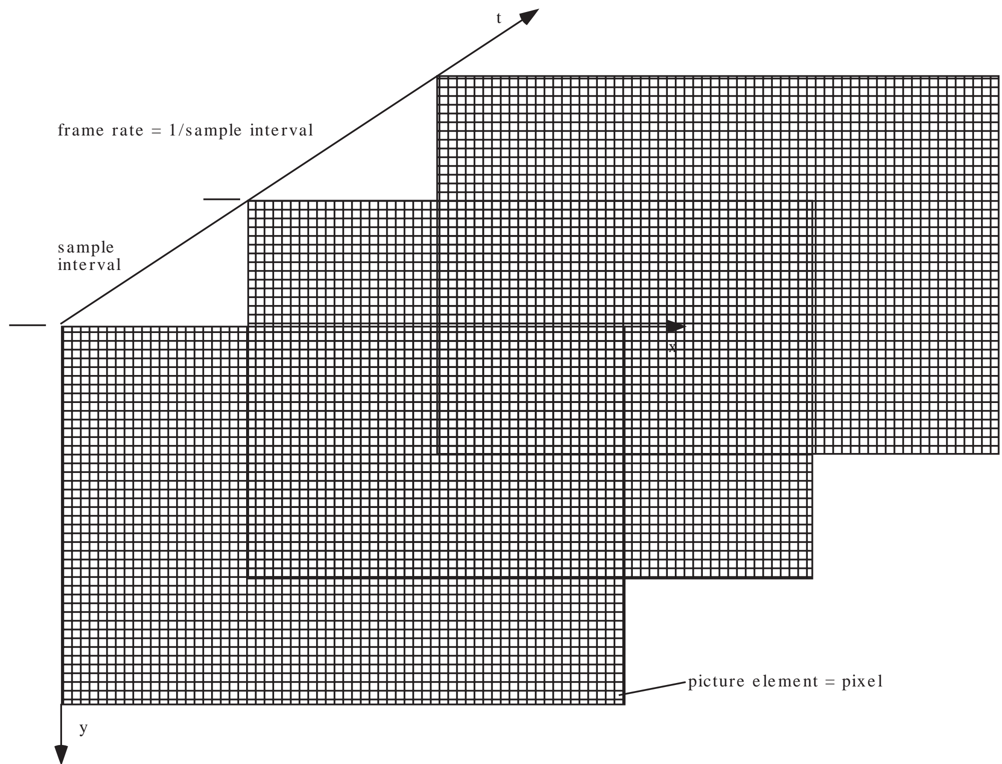


Figure 7.25: Image processing terminology.

7.10.2 IMAGE PROCESSING TASKS

There is an amazing array of image processing tasks available to extract information from an image. Figure 7.26 provides a partial overview of some of the tasks and related operations. The lower order tasks are associated with the formation and acquisition of an image. Once acquired, an image may be enhanced by a variety of techniques including thresholding, filtering, and edge enhancement and detection techniques. Higher order processing techniques are usually accomplished by applying a series of fundamental image processing operations. There are a number of techniques to efficiently store an image.

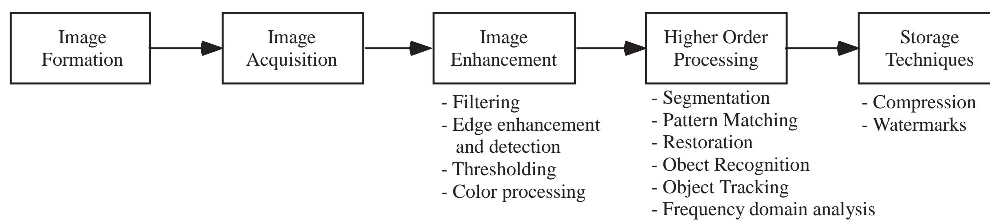
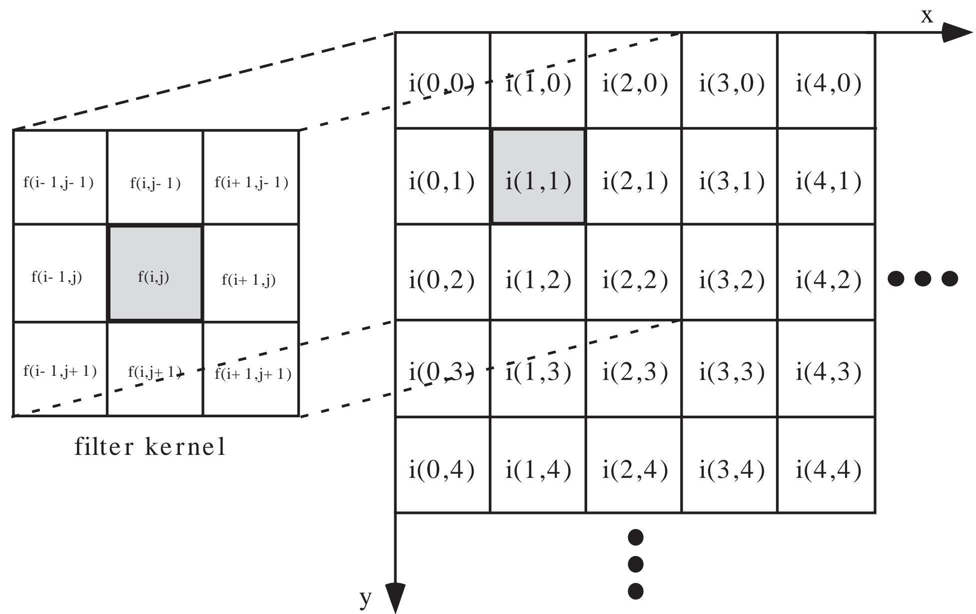


Figure 7.26: Image processing tasks.

Image processing operations typically requires the application of a series of simple operations to an image. Each operator is sequentially swept over an image to accomplish a specific task a shown in Figure 7.27. For example, the low pass filter kernel may be exhaustively applied to each pixel in an image to smooth image features. To apply a filter operator, the filter coefficients are multiplied by the image coefficients and then summed. The resulting value becomes the new pixel value at that image pixel location. The filter operator is exhaustively applied to every pixel in the image. Although the operations are quite straightforward, they require a substantial amount of processor power due to the sheer number of calculations that must be accomplished in a timely manner [Galbiati, 1990; Gonzalez and Woods, 2008]. As previously mentioned, BeagleBone is well suited for a portable image processing engine.

7.10.3 OPENCV COMPUTER VISION LIBRARY

The OpenCV Library is an open source computer vision library. The library is written in C and C++ and runs on a variety of operating systems including Linux, Windows, and Mac OS X. The library allows a designer to rapidly prototype an image processing application. Bradski and Kaehler provides an excellent tutorial on this library [Bradski and Kaehler, 2008]. With its Linux--based operating system, BeagleBone is ideally suited to host the OpenCV library. We illustrate the power and application of the OpenCV library with the “Stache Cam” application.



a) filter operation

f(i- 1,j- 1) 1/9	f(i,j- 1) 1/9	f(i+ 1,j- 1) 1/9
f(i- 1,j) 1/9	f(i,j) 1/9	f(i+ 1,j) 1/9
f(i- 1,j+ 1) 1/9	f(i,j+ 1) 1/9	f(i+ 1,j+ 1) 1/9

low pass filter kernel

f(i- 1,j- 1) - 1	f(i,j- 1) - 1	f(i+ 1,j- 1) - 1
f(i- 1,j) - 1	f(i,j) + 8	f(i+ 1,j) - 1
f(i- 1,j+ 1) - 1	f(i,j+ 1) - 1	f(i+ 1,j+ 1) - 1

high pass filter kernel

f(i- 1,j- 1) 0	f(i,j- 1) 1	f(i+ 1,j- 1) 0
f(i- 1,j) 1	f(i,j) - 4	f(i+ 1,j) 1
f(i- 1,j+ 1) 0	f(i,j+ 1) 1	f(i+ 1,j+ 1) 0

Laplacian filter kernel

b) filter operators

Figure 7.27: Image processing filters. To apply a filter operator, the filter coefficients are multiplied by the image coefficients and then summed. The resulting value becomes the new pixel value at that image pixel location. The filter operator is exhaustively applied to every pixel in the image.

7.10.4 STACHE CAM

In this section we illustrate the power and application of the OpenCV library with the "Stache Cam" application. The requirements for this system are straightforward. A video camera is used to capture images of various faces. Functions within the Open CV library are used to capture the image and automatically place a moustache on the face, as shown in Figure 7.28.

The system hardware consists of a camera (Playstation PS3 Eye), BeagleBone, the LCD3 BeagleBone Cape, and the BeagleBone Battery cape for portability. Linux provides a USB driver for the PS3 Eye camera. Different system hardware components are shown in Figure 7.29.

Stache Cam UML Activity Diagram

The UML activity diagram is provided in Figure 7.30.

C Code

```
1  //*****************************************************************
2  //Based on:
3  //https://code.ros.org/trac/opencv/browser/trunk/opencv
4  //         /samples/cpp/tutorial_code/objectDetection/
5  //         objectDetection2.cpp?rev=6553
6  //*****************************************************************
7
8  #include "opencv2/objdetect/objdetect.hpp"
9  #include "opencv2/highgui/highgui.hpp"
10 #include "opencv2/imgproc/imgproc.hpp"
11 #include "opencv2/imgproc/imgproc_c.h"
12
13 #include <iostream>
14 #include <stdio.h>
15
16 using namespace std;
17 using namespace cv;
18
19 string copyright = "\
20
21 IMPORTANT:READ BEFORE DOWNLOADING, COPYING, INSTALLING
22 OR USING.\n\\n\
23 By downloading, copying, installing or using the software
24 you agree to this license.\n\
25 If you do not agree to this license, do not download,
26 install, copy or use the software.\n\
27 \n\
28 \n\
29                          License Agreement\n\
30                For Open Source Computer Vision Library\n\
31 \n\
32 Copyright(C)2000-2008, Intel Corporation, all rights reserved.\n\
33 Copyright(C)2008-2011, Willow Garage Inc., all rights reserved.\n\
34 Copyright(C)2012, Texas Instruments, all rights reserved.\n\
35 Third party copyrights are property of their respective owners.\n\\n\
36 Redistribution and use in source and binary forms, with or without\n\
37 modification, are permitted provided that the following conditions\n\
38 are met:\n\
39 \n\
40    *Redistributions of source code must retain the above \n\
41     copyright notice, this list of conditions and the following\n\
42     disclaimer.\n\
43 \n\
44    *Redistributions in binary form must reproduce the above \n\
45     copyright notice, this list of conditions and the following \n\
46     disclaimer in the documentation and/or other materials provided \n\
47     with the distribution.\n\
48 \n\
49    *The name of the copyright holders may not be used to endorse or \n\
50     promote products derived from this software without specific \n\
51     prior written permission.\n\
52 \n\
53 This software is provided by the copyright holders and contributors\n\
54 \"as is\" and any express or implied warranties, including, but not\n\
55 limited to, the implied warranties of merchantability and fitness\n\
56 for a particular purpose are disclaimed.\n\
57 In no event shall the Intel Corporation or contributors be liable\n\
58 for any direct, indirect, incidental, special, exemplary, or\n\
59 consequential damages (including, but not limited to, procurement\n\
```

Figure 7.28: BeagleBone “stache” cam.

Figure 7.29: BeagleBone "stache" cam hardware.

```
60 of substitute goods or services; loss of use, data, or profits; or\n\
61 business interruption) however caused and on any theory of liability,\n\
62 whether in contract, strict liability, or tort (including negligence\n\
63 or otherwise) arising in any way out of the use of this software, even\n\
64 if advised of the possibility of such damage.\n\
65 \n";
66
67 /** Function Headers */
68 void detectAndDisplay(Mat frame);
69
70 /** Global variables */
71 String face_cascade_name = "lbpcascade_frontalface.xml";
72 CascadeClassifier face_cascade;
73 string window_name = "stache - BeagleBone OpenCV demo";
74 IplImage* mask = 0;
75
76 /** Command-line arguments */
77 int numCamera = -1;
78 const char* stacheMaskFile = "stache-mask.png";
79 int scaleHeight = 6;
80 int offsetHeight = 4;
81 int camWidth = 0;
82 int camHeight = 0;
83 int camFPS = 0;
84
85 /*** @function main */
86 int main(int argc, const char** argv)
```

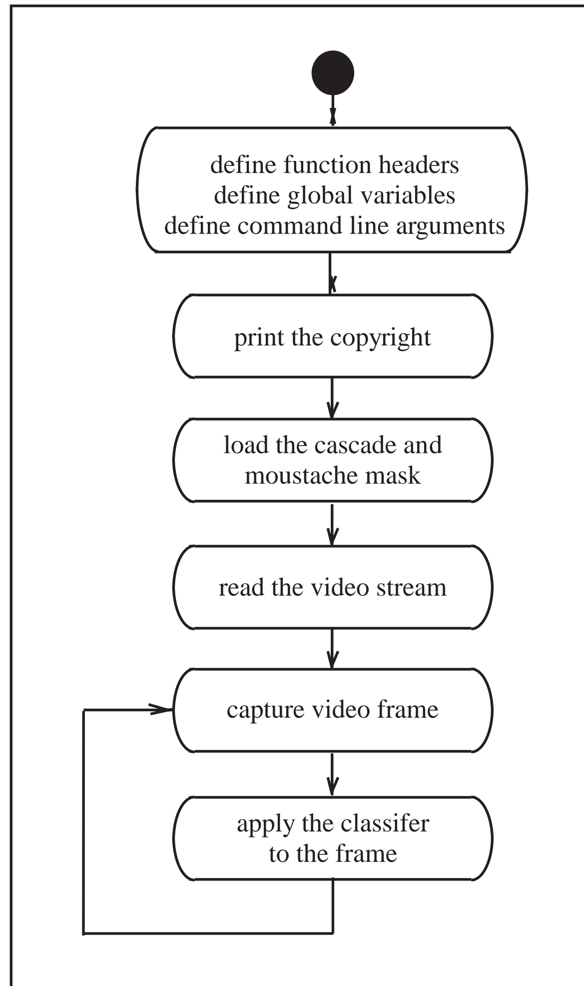


Figure 7.30: BeagleBone “stache” cam UML activity diagram.

```
87   {
88   CvCapture* capture;
89   Mat frame;
90
91   if(argc > 1) numCamera = atoi(argv[1]);
92   if(argc > 2) stacheMaskFile = argv[2];
93   if(argc > 3) scaleHeight = atoi(argv[3]);
94   if(argc > 4) offsetHeight = atoi(argv[4]);
95   if(argc > 5) camWidth = atoi(argv[5]);
96   if(argc > 6) camHeight = atoi(argv[6]);
97   if(argc > 7) camFPS = atoi(argv[7]);
98
99   //-- 0. Print the copyright
100  cout << copyright;
101
102  //-- 1. Load the cascade
103  if( !face_cascade.load(face_cascade_name) ){ printf("--(!)Error
```

```
104         loading\n"); return -1; };
105
106   //-- 1a.
107 Load the mustache mask
108   mask = cvLoadImage(stacheMaskFile);
109   if(!mask) { printf("Could not load %s\n", stacheMaskFile); exit(-1); }
110
111   //-- 2. Read the video stream
112   capture = cvCaptureFromCAM(numCamera);
113   if(camWidth) cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH,
114   camWidth);
115   if(camHeight) cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT,
116   camHeight);
117   if(camFPS) cvSetCaptureProperty(capture, CV_CAP_PROP_FPS, camFPS);
118   if(capture)
119     {
120     while(true)
121       {
122       frame = cvQueryFrame(capture);
123
124       //-- 3. Apply the classifier to the frame
125       try
126         {
127         if(!frame.empty())
128           {
129           detectAndDisplay( frame );
130           }
131         else
132           {
133           printf(" --(!) No captured frame--Break!\n"); break;
134           }
135         int c = waitKey(10);
136         if( (char)c == 'c' )
137           {
138           break;
139           }
140         }
141       catch(cv::Exception e)
142       {
143       }
144      }
145     }
146   return 0;
147 }
148
149 //************************************************
150 // @function detectAndDisplay - function detects
151 //face and places mustache in an appropriate location
152 //************************************************
153
154 void detectAndDisplay(Mat frame)
155   {
156   std::vector<Rect> faces;
157   Mat frame_gray;
158
159   cvtColor(frame, frame_gray, CV_BGR2GRAY);
160   equalizeHist(frame_gray, frame_gray);
161
162   //-- Detect faces
163   face_cascade.detectMultiScale(frame_gray,
164   faces, 1.1, 2, 0, Size(80, 80));
165
166   for(int i = 0; i < faces.size(); i++)
167     {
168     //-- Scale and apply mustache mask for each face
169     Mat faceROI = frame_gray(faces[i]);
170     IplImage iplFrame = frame;
171     IplImage *iplMask = cvCreateImage(cvSize(faces[i].width,
172                               faces[i].height/scaleHeight),
173                               mask->depth, mask->nChannels);
174     cvSetImageROI(&iplFrame, cvRect(faces[i].x,
175       faces[i].y + (faces[i].height/scaleHeight)*offsetHeight,
176       faces[i].width, faces[i].height/scaleHeight));
177
178
179     cvResize(mask, iplMask, CV_INTER_LINEAR);
180     cvSub(&iplFrame, iplMask, &iplFrame);
181     cvResetImageROI(&iplFrame);
182     }
183
184 //-- Show what you got
185 flip(frame, frame, 1);
186 imshow(window_name, frame);
187 }
188 //************************************************
```

7.11 SUMMARY

In the early chapters of this book, we examined the Bonescript environment as a user-friendly tool to rapidly employ BeagleBone features right out of the box. In this chapter we revisited Bonescript and demonstrated its power as a rapid prototyping tool to develop complex, processor-based systems employing multiple BeagleBone subsystems. We carefully chose each of the projects to illustrate how BeagleBone may be used in a variety of project areas including instrumentation intensive applications (weather station), in assistive and educational technology applications (Speak & Spell), in motor and industrial control applications (Dagu robot), and calculation intensive image processing applications (moustache cam).

7.12 REFERENCES

- Barret, J. "Closer to the Sun International." www.closertothesungallery.com
- Barrett, S. and Pack, D. 2008. *Atmel AVR Processor Primer Programming and Interfacing.* San Rafael, CA: Morgan & Claypool Publishers.
- Barrett, S. and Pack, D. 2005. *Embedded Systems Design and Applications with the 68HC12 and HCS12.* Upper Saddle River, NJ: Pearson Prentice Hall.
- Barrett, S. and Pack, D. 2006. *Processors Fundamentals for Engineers and Scientists.* San Rafael, CA: Morgan and Claypool Publishers.
- Bradski, D. and Kaehler, A. 2008. *Learning OpenCV: Computer Vision with the OpenCV Library.* Sebastopol, CA: O'Reilly.
- Coley, G. *BeagleBone Rev A6 Systems Reference Manual.* Revision 0.0, May 9, 2012, beagleboard.org; www.beaglebord.org.
- Galbiati, L. 1990. *Machine Vision and Digital Image Processing Fundamentals.* Englewood Cliffs, NJ: Prentice Hall.
- Gonzalez, R.C. and Woods, R.E. 2008. *Digital Image Processing.* 3rd ed. Upper Saddle River, NJ: Prentice Hall.
- Hughes-Croucher, T. and Wilson, M. 2012. *Node Up and Running.* Sebastopol, CA: O'Reilly Media, Inc.
- Kelley, A. and Pohl, I. 1998. *A Book on C—Programming in C.* 4th ed., Boston, MA: Addison Wesley.
- Kiessling, M. 2012. *The Node Beginner Guide: A Comprehensive Node.js Tutorial.*
- Pollock, J. 2010. *JavaScript.* 3rd ed., New York, NY: McGraw Hill.

- Vander Veer, E. 2005. *JavaScript for Dummies.* 4th ed. Hoboken, NJ: Wiley Publishing, Inc.
- von Hagen, W. 2007. *Ubuntu Linux Bible.* Indianapolis, IN: Wiley Publishing, Inc.

7.13 CHAPTER EXERCISES

1. Construct the UML activity diagrams for all functions related to the weather station.
2. Add one of the following sensors to the weather station:
    - anemometer
    - barometer
    - hygrometer
    - rain gauge
    - thermocouple

   You will need to investigate background information on the selected sensor, develop an interface circuit for the sensor, and modify the weather station code.
3. Complete the control algorithm for the weather station to convert the reading from the LM34 temperature sensor and display its value and wind direction.
4. The Dagu Magician robot under microcontroller control abruptly starts and stops when PWM is applied. Modify the algorithm to provide the capability to gradually ramp up (and down) the motor speed.
5. Modify the Dagu Magician circuit and microcontroller code such that the maximum speed of the robot is set with an external potentiometer.
6. Modify the Dagu Magician circuit and microcontroller code such that the IR sensors are only asserted just before a range reading is taken.
7. Add the following features to the Dagu Magician platform:
    - line following capability (Hint: Adapt the line following circuitry onboard the Dagu Magician to operate with the BeagleBone.);
    - two-way robot communications (use the IR sensors already aboard); and
    - voice output (Hint: Use the SP0-512 speech synthesis chip.).

CHAPTER 8

Where to from Here?

Objectives: After reading this chapter, the reader should be able to do the following.

- View this book as simply the beginning of the journey in using BeagleBone in a wide variety of applications.
- Describe the wide variety of software libraries and other resources available to the BeagleBone user.
- Appreciate the advantages in becoming an active member of the BeagleBoard.org community.

8.1 OVERVIEW

Reaching the last chapter of the book, you might be breathing a sigh of relief thinking I've completed the book. I'm at the end. Quite the contrary, this book is merely the beginning of your journey of using BeagleBone in a wide variety of applications.

In this chapter we provide a brief review of a number of software libraries and other resources available to a BeagleBone user. These resources allow access to a wide array of features to extend the capabilities of BeagleBone. We conclude with an invitation to become an active member of the BeagleBoard.org community.

8.2 SOFTWARE LIBRARIES

8.2.1 OPENCV

The OpenCV Library is an open source computer vision library. The library is written in C and C++ and runs on a variety of operating systems including Linux, Windows and Mac OS X. The library allows a designer to rapidly prototype an image processing application. Bradski and Kaehler provides an excellent tutorial on this library [Bradski and Kaehler, 2008]. With a Linux-based operating system, BeagleBone is ideally suited to host the OpenCV library.

8.2.2 QT

Qt is a C++ library that provides for the rapid development of user-friendly graphical user interfaces of GUIs. Qt readily executes on Windows, Unix, MacOS X and Linux-based embedded systems. The Qt library allows GUIs that employ buttons, scroll bars, etc. The library employs

the concept of signals and slots to link an event to a desired response [Dalheimer, 2002]. Several excellent sources on the Qt library are listed at the end of the chapter.

8.2.3 KINECT

Kinect is the motion sensing input device developed for the Xbox video game console. The Kinect allows natural user movements such as gestures or spoken commands to interact with a game. A Kinect library is available for BeagleBone operating in Linux. A good introduction to Kinect is provided by the references listed at the end of the chapter.

8.3 ADDITIONAL RESOURCES

In this section we provide pointers to a series of user-groups and other resources available to a BeagleBone user.

8.3.1 OPENROV

Earlier in the book, we introduced an underwater remote operated vehicle (ROV) project. There are a number of groups dedicated to ROV development as a way of reaching the next generation of engineers and scientists. A brief description of the groups is provided below.

- OpenROV is a do-it-yourself (DIY) group dedicated to underwater robots for exploration and adventure. The group includes amateur and professional ROV builders and operators from over 50 countries who have a passion for exploring the deep [www.openrov.com]. The OpenROV community has a BeagleBone Cape available for controlling an ROV, as shown in Figure 8.1.

- SeaPerch is an underwater robotics program to equip educators and students with resources to build an underwater Remotely Operated Vehicle (ROV). There are a number of excellent texts on underwater ROV development listed at the end of the chapter [www.seaperch.org].

8.3.2 NINJA BLOCKS

Ninja Blocks is an innovative method to sense the environment and control hardware within the home. Ninja Blocks is based on an open hardware concept where hardware, software and application information is openly shared among the Ninja Blocks community. The basic Ninja Block unit is illustrated in Figure 8.2. The basic unit includes the following components [www.ninjablocks.com]:

- wireless motion sensor,

- wireless door/window contact sensor,

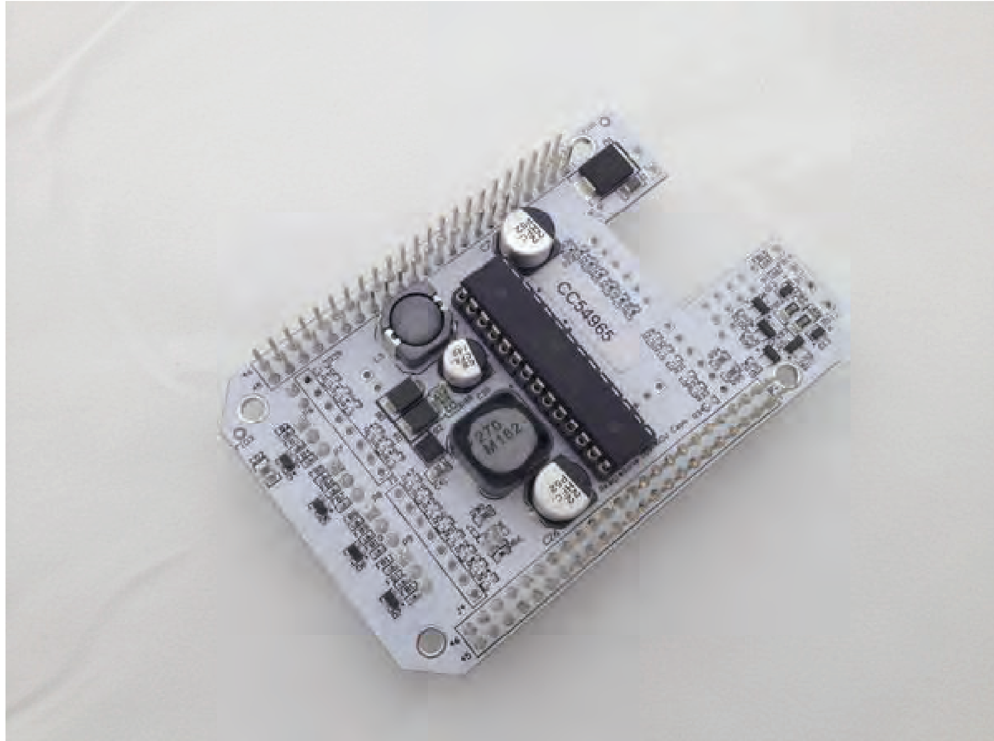


Figure 8.1: The OpenROV BeagleBone Cape [www.openrov.com].

- wireless button,
- wireless temperature and humidity sensor,
- Ninja Block equipped with a BeagleBone and an Arduino processor,
- USB Wi-Fi module,
- Ethernet Cable, and
- 5 VDC, 3 Amp power supply with connectors.

The basic unit allows ease of interface to a wide variety of devices.

8.3.3 RELATED BOOKS

BeagleBone has become quite popular. There are a number of books available to enhance the BeagleBone experience; here is a partial listing.

Figure 8.2: Ninja Blocks [www.ninjablocks.com].

- *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux* by Derek Molloy
- *BeagleBone Black Interfacing: hardware and software* by Yury Magda
- *30 BeagleBone Black Projects for the Evil Genius* by Christopher Rush
- *BeagleBone Black Programming by Example* by Agus Kurniawan
- *BeagleBone Cookbook: Software and Hardware Problems and Solutions* by Mark Yoder and Jason Kridner
- *Getting Started with BeagleBone:Linux-Powered Electronic Projects with Python and JavaScript* by Matt Richardson
- *Programming the BeagleBone Black: Getting Started with JavaScript and Bonescript* by Simon Monk
- *BeagleBone for Secret Agents* by Josh Datko
- *BeagleBone Robotic Projects* and *Mastering BeagleBone Robotics* by Richard Grimmett

- *The BeagleBone Black Primer* by Brian McLaughlin
- *BeagleBone Home Automation* by Juha Lumme
- *Building a BeagleBone Black Super Cluster* by Andreas Josef Reichel
- *Learning BeagleBone* with Hunyue Yau
- *Building a Home Security System with BeagleBone* by Bill Pretty
- *BeagleBone for Dummies* by Rui Santos and Luis Perestrelo

8.3.4 BEAGLEBOARD.ORG RESOURCES

The BeagleBoard.org community has many members. What we all have in common is the desire to put processing power in the hands of the next generation of users. BeagleBoard.org, with Texas Instruments' support, embraced the open source concept with the development and release of BeagleBone in late 2011. Their support will insure the BeagleBone project will be sustainable. BeagleBoard.org partnered with Circuitco (`www.Circuitco.com`) to produce BeagleBone and its associated Capes. The majority of the Capes have been designed and fabricated by Circuitco. Clint Cooley, President of Circuitco, is most interested in helping users develop and produce their own ideas for BeagleBone Capes. Texas Instruments has also supported the BeagleBoard.org community by giving Jason Kridner the latitude to serve as the open platform technologist and evangelist for the BeagleBoard.org community. The most important members of the community are the BeagleBoard and Bone users. Our ultimate goal is for the entire community to openly share their successes and to encourage the next generation of STEM practitioners.

8.3.5 CONTRIBUTING TO BONESCRIPT

It is important to emphasize that Bonescript is an open source programming environment. We are counting on the user community to expand the features of Bonescript. If there is a feature you need, please develop it and share it with the BeagleBoard.org community. This is easily done by submitting a pull request to `www.github.com/jadonk/bonescript`.

8.4 SUMMARY

In this chapter we provided a brief review of a number of software libraries and other resources available to the BeagleBone user. These resources allow access to a wide array of features to extend the capabilities of BeagleBone. We concluded with an invitation to become an active member of the BeagleBoard.org community.

8.5 REFERENCES

- Bohm, H. and Jensen, V. 2012. *Build Your Own Underwater Robot and Other Wet Projects.* Monterey, CA: Marine Advanced Technology Center.

- Bradski, D. and Kaehler, A. 2008. *Learning OpenCV: Computer Vision with the OpenCV Library.* Sebastopol, CA: O'Reilly, 2008.
- Dalheimer, M. 2002. *Programming with Qt.* Sebastopol, CA: O'Reilly.
- Kean, S., Hall, J., and Perry, P. 2012. *Meet the Kinect: An Introduction to Programming Natural User Interfaces.* Berkeley, CA: Apress.
- Miles, R. 2012. *Start Here! Learn the Kinect API.* Sebastopol, CA: O'Reilly.
- Moore, S., Bohm, H., and Jensen, V. 2010. *Underwater Robotics: Science, Design and Fabrication.* Marine Advanced Technology Education Center, Monterrey, CA.
- *OpenROV: Open-source Underwater Robots for Exploration and Education.* 2015; www.openrov.com
- *Seaperch,* 2015; www.seaperch.com
- Thelin, J. 2007. *Foundations of Qt Development.* Berkerly, CA: Apress.

8.6 CHAPTER EXERCISES

1. Construct a personal plan on how you will improve your BeagleBone and Linux operating system skills.
2. Develop three new features for the Bonescript environment and submit them to the BeagleBoard.org community.

APPENDIX A

Bonescript functions

Bonescript Environment

Analog input/output

function name	**Description**
var1 = analogRead(pin_name);	**Description:** Performs analog-to-digital conversion on voltage at specified pin. Analog voltage may range from 0 to 1.8 VDC. **arguments:** pin_name **returns:** Normalized value from 0 .. 1 corresponding to 0 .. 1.8 VDC.
analogWrite(pin_name, analog_value);	**Description:** Delivers analog level to specified pin via 1 kHz pulse width modulated signal. Analog level specified as normalized value from 0..1 corresponding to 0 to 1.8 VDC. **arguments:** pin_name, logic_level (0 .. 1). **returns:** None.

Figure A.1: Bonescript analog input and output functions.

Bonescript Environment

Digital input/output

function name	**Description**
pinMode(pin_name, direction);	**Description:** Sets digital pin direction (INPUT or OUTPUT). **arguments:** pin_name, direction (INPUT or OUTPUT) **returns:** None.
getPinMode(pin_name);	**Description:** Reports status, parameters of selected pin **arguments:** pin_name **returns:** parameters of selected pin
digitalWrite(pin_name, logic_level);	**Description:** Sets digital output pin logic level (HIGH or LOW). **arguments:** pin_name, logic_level (HIGH or LOW) **returns:** None.
var1 = digitalRead(pin_name);	**Description:** Reads digital input pin logic level (HIGH or LOW). **arguments:** pin_name **returns:** Logic level of specified pin (HIGH or LOW)
shiftOut(spi_dataPin, spi_clockPin, bitOrder, value);	**Description:** Provides Serial Peripheral Interface (SPI) data transmission **arguments:** pin for SPI data, pin for SPI clock, bit order (LSBFIRST, MSBFIRST), data for transmission **returns:** None.

Figure A.2: Bonescript digital input and output functions.

Bit and Byte Operators

function name	Description
return_byte = lowByte(value);	**Description:** Returns the lower (least significant) byte of the value. The value may be of any type. **arguments:** The value may be of any type. **returns:** The low order, least significant byte.
bit_value = bitRead(number, bit_position);	**Description:** Returns the logic value (1 or 0) of the specified bit position in the number. **arguments:** The number to be evaluated and the desired bit position. **returns:** The logic value (1 or 0).
bitWrite(number, bit_position, bit_value);	**Description:** Writes the specified bit position with a bit value (0 or 1) for the specified number. **arguments:** The number to be written to, the desired bit position and the desired value (0 or 1). **returns:** None.
bitSet(number, bit_position);	**Description:** Sets the specified bit position to logic high (1) for the specified number. **arguments:** The number to be written to and the desired bit position. **returns:** None.
bitClear(number, bit_position);	**Description:** Clears the specified bit position to logic low (0) for the specified number. **arguments:** The number to be written to and the desired bit position. **returns:** None.
bit_value = bit(n);	**Description:** Returns 2^n as the bit value. **arguments:** The bit position (n). **returns:** The bit value.

Figure A.3: Bonescript bit and byte operators.

Interrupts

function name	**Description**
attachInterrupt(input pin, ISR_name, trigger);	**Description:** Specifies input pin, interrupt trigger source, and interrupt service routine (ISR) name **arguments:** interrupt pin, trigger source RISING, FALLING, CHANGE), and interrupt service routine name **returns:** None.
detachInterrupt(pin);	**Description:** Detaches interrupt service from specified pin. **arguments:** Interrupt pin **returns:** None.

Constants

OUTPUT INPUT LOW HIGH RISING FALLING CHANGE

Figure A.4: Bonescript interrupt operators and constants.

APPENDIX B

LCD interface for BeagleBone in C

B.1 BEAGLEBONE LCD INTERFACE

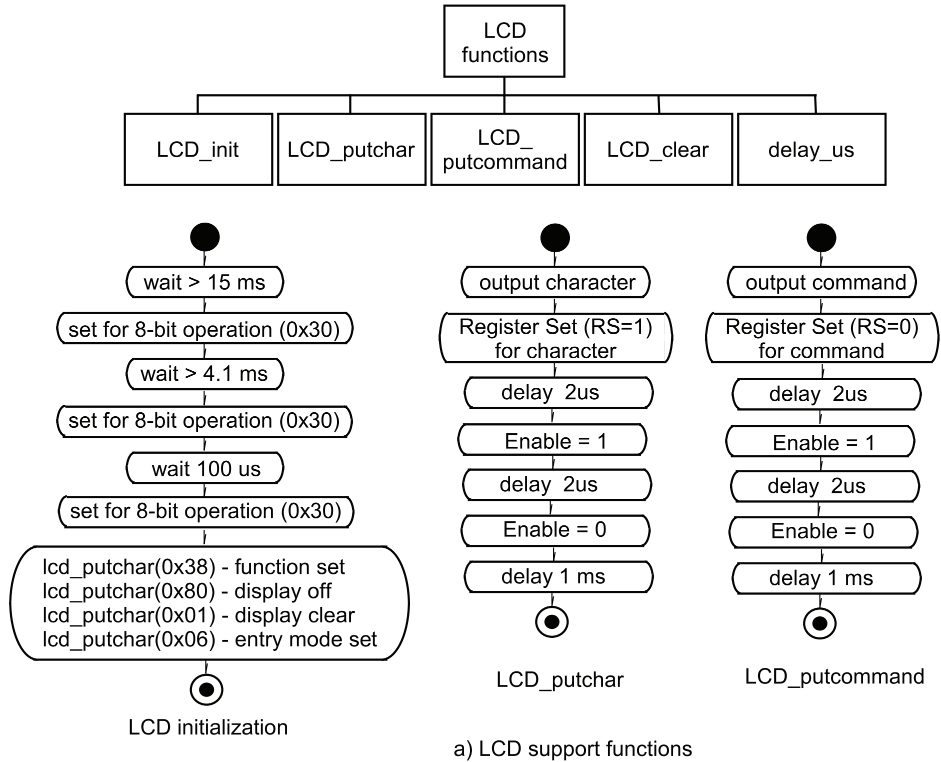
Provided in Figure B.1 are the structure chart, UML activity diagrams and connection diagram for a Sparkfun LCD–09052 basic 16 x 2 character liquid crystal display. Note the LCD operates at 3.3 VDC which is compatible with BeagleBone.

B.2 BEAGLEBONE BLACK LCD C CODE

Provided below is the LCD C code for the BeagleBone Black operating under Linux.

Please note the following pins were used in the interface:

- P8.9–RS
- P8.10–E
- P8.11–DB0
- P8.12–DB1
- P8.13–DB2
- P8.14–DB3
- P8.15–DB4
- P8.16–DB5
- P8.17–DB6
- P8.18–DB7



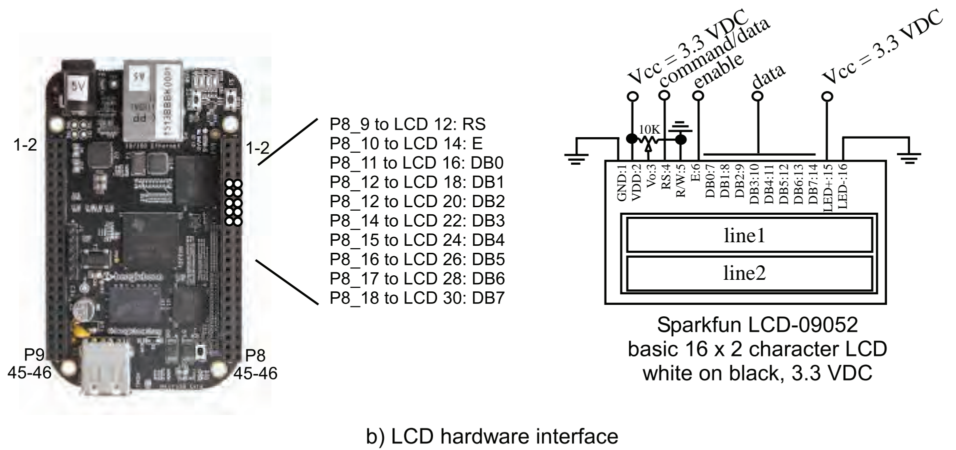


Figure B.1: LCD support data.

```
1 //***********************************************************************
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stddef.h>
5 #include <time.h>
6 #include <string.h>
7 #include <sys/time.h>
8
9 #define output "out"
10 #define input "in"
11
12 //function prototypes
13 void configure_LCD_interface(void);
14 void LCD_putcommand(unsigned char);
15 void LCD_putchar(unsigned char);
16 void LCD_print(unsigned int, char *string);
17 void delay_us(int);
18 void LCD_init(void);
19 FILE *setup_gpio(unsigned int pin_number, char* direction);
20 void set_gpio_value(FILE* value_file, unsigned int logic_status);
21
22 //LCD_RS: bone.P8_9 LCD Register Set (RS) control: GPIO2_5: pin-69
23 FILE* gpio_rs_value = NULL;
24 int pin_number_rs = 69, logic_status_rs = 1;
25 char* pin_direction_rs = output;
26
27 //LCD_E:   bone.P8_10: LCD Enable En (E): GPIO2_4: pin-68
28 FILE* gpio_e_value = NULL;
29 int pin_number_e = 68, logic_status_e = 1;
30 char* pin_direction_e = output;
31
32 //LCD_DB0: bone.P8_11: LCD Data line DB0: GPIO1_13: pin-45
33 FILE *gpio_db0_value = NULL;
34 int pin_number_db0 = 45, logic_status_db0 = 1;
35 char* pin_direction_db0 = output;
36
37 //LCD_DB1: bone.P8_12: LCD Data line DB1: GPIO1_12: pin-44
38 FILE *gpio_db1_value = NULL;
39 int pin_number_db1 = 44, logic_status_db1 = 1;
40 char* pin_direction_db1 = output;
41
42 //LCD_DB2: bone.P8_13: LCD Data line DB2: GPIO0_23: pin-23
43 FILE *gpio_db2_value = NULL;
44 int pin_number_db2 = 23, logic_status_db2 = 1;
45 char* pin_direction_db2 = output;
46
47 //LCD_DB3: bone.P8_14: LCD Data line DB3: GPIO0_26: pin-26
48 FILE *gpio_db3_value = NULL;
```

```
49 int pin_number_db3 = 26, logic_status_db3 = 1;
50 char* pin_direction_db3 = output;
51
52 //LCD_DB4: bone.P8_15: LCD Data line DB4: GPIO1_15: pin-47
53 FILE *gpio_db4_value = NULL;
54 int pin_number_db4 = 47, logic_status_db4 = 1;
55 char* pin_direction_db4 = output;
56
57 //LCD_DB5: bone.P8_16: LCD Data line DB5: GPIO1_14: pin-46
58 FILE *gpio_db5_value = NULL;
59 int pin_number_db5 = 46, logic_status_db5 = 1;
60 char* pin_direction_db5 = output;
61
62 //LCD_DB6: bone.P8_17: LCD Data line DB6: GPIO0_27: pin-27
63 FILE *gpio_db6_value = NULL;
64 int pin_number_db6 = 27, logic_status_db6 = 1;
65 char* pin_direction_db6 = output;
66
67 //LCD_DB7: bone.P8_18: LCD Data line DB7: GPIO2_1: pin-65
68 FILE *gpio_db7_value = NULL;
69 int pin_number_db7 = 65, logic_status_db7 = 1;
70 char* pin_direction_db7 = output;
71
72
73 int main(void)
74 {
75 configure_LCD_interface();
76 printf("Configure LCD \n");
77 LCD_init();  //call LCD initialize
78 printf("LCD initialize \n");
79
80 while(1)
81  {
82   LCD_print(1, "Bad to the");
83   printf("Bad to the\n");
84   delay_us(100);
85   LCD_print(2, "   Bone");
86   printf("   Bone\n");
87   delay_us(100);
88   }
89 return 1;
90 }
91
92 //*****************************************************************
93 //FILE *setup_gpio(unsigned int pin_number, char* pin_direction)
94 //*****************************************************************
95
96 FILE *setup_gpio(unsigned int pin_number, char* pin_direction)
```

```
97 {
98 FILE *exp, *gpio_value, *gpio_direction;
99 char gpio_direction_filename[40];
100 char gpio_value_filename[40];
101
102 //create direction and value file for pin
103 exp = fopen("/sys/class/gpio/export", "w");
104 if(exp == NULL) {printf("Unable to open export.\n");}
105 fseek(exp, 0, SEEK_SET);
106 fprintf(exp, "%d", pin_number);
107 fflush(exp);
108 fclose(exp);
109
110 //configure pin direction
111 sprintf(gpio_direction_filename, "/sys/class/gpio/gpio%d/direction",
112         pin_number);
113 gpio_direction = fopen(gpio_direction_filename, "w");
114 if(gpio_direction == NULL) {printf("Unable to open %s.\n",
115                                     gpio_direction_filename);}
116 fseek(gpio_direction, 0, SEEK_SET);
117 fprintf(gpio_direction, "%s", pin_direction);
118 fflush(gpio_direction);
119 fclose(gpio_direction);
120
121 //open pin value file
122 sprintf(gpio_value_filename, "/sys/class/gpio/gpio%d/value", pin_number
    );
123 gpio_value = fopen(gpio_value_filename, "w");
124 if(gpio_value==NULL){printf("Unable to open %s.\n", gpio_value_filename
    );}
125
126 printf("Opening %d (%s) returned 0x%08x\n", pin_number,
    gpio_value_filename,
127         (int)gpio_value);
128 return(gpio_value);
129 }
130
131 //***************************************************************
132 //void set_gpio_value(FILE* value_file, unsigned int logic_status)
133 //***************************************************************
134 void set_gpio_value(FILE* value_file, unsigned int logic_status)
135 {
136
137 printf(" %d -> 0x%08x\n", logic_status, (int)value_file);
138 fseek(value_file, 0, SEEK_SET);
139 fprintf(value_file, "%d", logic_status);
140 fflush(value_file);
141
```

```
142 }
143
144 //*****************************************************************
145 //void configure_LCD_interface(void)
146 //*****************************************************************
147
148 void configure_LCD_interface(void)
149 {
150
151 //Setup LCD GPIO pins
152 gpio_rs_value = setup_gpio(pin_number_rs, pin_direction_rs);
153 gpio_e_value = setup_gpio(pin_number_e, pin_direction_e);
154 gpio_db0_value = setup_gpio(pin_number_db0, pin_direction_db0);
155 gpio_db1_value = setup_gpio(pin_number_db1, pin_direction_db1);
156 gpio_db2_value = setup_gpio(pin_number_db2, pin_direction_db2);
157 gpio_db3_value = setup_gpio(pin_number_db3, pin_direction_db3);
158 gpio_db4_value = setup_gpio(pin_number_db4, pin_direction_db4);
159 gpio_db5_value = setup_gpio(pin_number_db5, pin_direction_db5);
160 gpio_db6_value = setup_gpio(pin_number_db6, pin_direction_db6);
161 gpio_db7_value = setup_gpio(pin_number_db7, pin_direction_db7);
162
163 }
164
165 //*****************************************************************
166 //LCD_init
167 //*****************************************************************
168
169 void LCD_init(void)
170 {
171 delay_us(15000);                    //wait 15 ms
172 LCD_putcommand(0x30);               //set for 8-bit operation
173 delay_us(5000);                     //delay 5 ms
174 LCD_putcommand(0x30);               //set for 8-bit operation
175 delay_us(100);                      //delay 100 us
176 LCD_putcommand(0x30);               //set for 8-bit operation
177 LCD_putcommand(0x38);               //function set
178 LCD_putcommand(0x80);               //display off
179 LCD_putcommand(0x01);               //display clear
180 LCD_putcommand(0x06);               //entry mode set
181 }
182
183 //*****************************************************************
184 //LCD_putcommand
185 //*****************************************************************
186
187 void LCD_putcommand(unsigned char cmd)
188 {
189 //parse command variable into individual bits for output
```

```
190 //to LCD
191
192 //configure DB7 value
193 if((cmd & 0x0080)== 0x0080)
194   {
195   printf("CmDB7:1");
196   logic_status_db7 = 1;
197   }
198 else
199   {
200   printf(" CmDB7:0");
201   logic_status_db7 = 0;
202   }
203 set_gpio_value(gpio_db7_value, logic_status_db7);
204
205
206 //configure DB6 value
207 if((cmd & 0x0040)== 0x0040)
208   {
209   printf(" CmDB6:1");
210   logic_status_db6 = 1;
211   }
212 else
213   {
214   printf(" CmDB6:0");
215   logic_status_db6 = 0;
216   }
217 set_gpio_value(gpio_db6_value, logic_status_db6);
218
219
220 //configure DB5 value
221 if((cmd & 0x0020)== 0x0020)
222   {
223   printf(" CmDB5:1");
224   logic_status_db5 = 1;
225   }
226 else
227   {
228   printf(" CmDB5:0");
229   logic_status_db5 = 0;
230   }
231 set_gpio_value(gpio_db5_value, logic_status_db5);
232
233
234 //configure DB4 value
235 if((cmd & 0x0010)== 0x0010)
236   {
237   printf(" CmDB4:1");
```

```
238   logic_status_db4 = 1;
239   }
240 else
241   {
242   printf(" CmDB4:0");
243   logic_status_db4 = 0;
244   }
245 set_gpio_value(gpio_db4_value, logic_status_db4);
246
247
248 //configure DB3 value
249 if((cmd & 0x0008)== 0x0008)
250   {
251   printf(" CmDB3:1");
252   logic_status_db3 = 1;
253   }
254 else
255   {
256   printf(" CmDB3:0");
257   logic_status_db3 = 0;
258   }
259 set_gpio_value(gpio_db3_value, logic_status_db3);
260
261
262 //configure DB2 value
263 if((cmd & 0x0004)== 0x0004)
264   {
265   printf(" CmDB2:1");
266   logic_status_db2 = 1;
267   }
268 else
269   {
270   printf(" CmDB2:0");
271   logic_status_db2 = 0;
272   }
273 set_gpio_value(gpio_db2_value, logic_status_db2);
274
275
276 //configure DB1 value
277 if((cmd & 0x0002)== 0x0002)
278   {
279   printf(" CmDB1:1");
280   logic_status_db1 = 1;
281   }
282 else
283   {
284   printf(" CmDB1:0");
285   logic_status_db1 = 0;
```

```
286   }
287 set_gpio_value(gpio_db1_value, logic_status_db1);
288
289
290 //configure DB0 value
291 if((cmd & 0x0001)== 0x0001)
292   {
293   printf(" CmDB0:1");
294   logic_status_db0 = 1;
295   }
296 else
297   {
298   printf(" CmDB0:0");
299   logic_status_db0 = 0;
300   }
301 set_gpio_value(gpio_db0_value, logic_status_db0);
302
303 printf("\n");
304
305 //LCD Register Set (RS) to logic zero for command input
306 logic_status_rs = 0;
307 set_gpio_value(gpio_rs_value, logic_status_rs);
308
309 //LCD Enable (E) pin high
310 logic_status_e = 1;
311 set_gpio_value(gpio_e_value, logic_status_e);
312
313 //delay
314 delay_us(2);
315
316 //LCD Enable (E) pin low
317 logic_status_e = 0;
318 set_gpio_value(gpio_e_value, logic_status_e);
319
320 //delay
321 delay_us(100);
322 }
323
324 //******************************************************************
325 //LCD_putchar
326 //******************************************************************
327
328 void LCD_putchar(unsigned char chr)
329 {
330 //parse character variable into individual bits for output
331 //to LCD
332
333 printf("Data: %c:%d", chr, chr);
```

```
334 chr = (int)(chr);
335
336 //configure DB7 value
337 if((chr & 0x0080)== 0x0080)
338   {
339   printf(" DB7:1");
340   logic_status_db7 = 1;
341   }
342 else
343   {
344   printf(" DB7:0");
345   logic_status_db7 = 0;
346   }
347 set_gpio_value(gpio_db7_value, logic_status_db7);
348
349
350 //configure DB6 value
351 if((chr & 0x0040)== 0x0040)
352   {
353   printf(" DB6:1");
354   logic_status_db6 = 1;
355   }
356 else
357   {
358   printf(" DB6:0");
359   logic_status_db6 = 0;
360   }
361 set_gpio_value(gpio_db6_value, logic_status_db6);
362
363
364 //configure DB5 value
365 if((chr & 0x0020)== 0x0020)
366   {
367   printf(" DB5:1");
368   logic_status_db5 = 1;
369   }
370 else
371   {
372   printf(" DB5:0");
373   logic_status_db5 = 0;
374   }
375 set_gpio_value(gpio_db5_value, logic_status_db5);
376
377
378 //configure DB4 value
379 if((chr & 0x0010)== 0x0010)
380   {
381   printf(" DB4:1");
```

```
382   logic_status_db4 = 1;
383   }
384 else
385   {
386   printf(" DB4:0");
387   logic_status_db4 = 0;
388   }
389 set_gpio_value(gpio_db4_value, logic_status_db4);
390
391
392 //configure DB3 value
393 if((chr & 0x0008)== 0x0008)
394   {
395   printf(" DB3:1");
396   logic_status_db3 = 1;
397   }
398 else
399   {
400   printf(" DB3:0");
401   logic_status_db3 = 0;
402   }
403 set_gpio_value(gpio_db3_value, logic_status_db3);
404
405
406 //configure DB2 value
407 if((chr & 0x0004)== 0x0004)
408   {
409   printf(" DB2:1");
410   logic_status_db2 = 1;
411   }
412 else
413   {
414   printf(" DB2:0");
415   logic_status_db2 = 0;
416   }
417 set_gpio_value(gpio_db2_value, logic_status_db2);
418
419
420 //configure DB1 value
421 if((chr & 0x0002)== 0x0002)
422   {
423   printf(" DB1:1");
424   logic_status_db1 = 1;
425   }
426 else
427   {
428   printf(" DB1:0");
429   logic_status_db1 = 0;
```

```
430   }
431 set_gpio_value(gpio_db1_value, logic_status_db1);
432
433 //configure DB0 value
434 if((chr & 0x0001)== 0x0001)
435   {
436   printf(" DB0:1\n");
437   logic_status_db0 = 1;
438   }
439 else
440   {
441   printf(" DB0:0\n");
442   logic_status_db0 = 0;
443   }
444 set_gpio_value(gpio_db0_value, logic_status_db0);
445
446 //LCD Register Set (RS) to logic one for character input
447 logic_status_rs = 1;
448 set_gpio_value(gpio_rs_value, logic_status_rs);
449
450 //LCD Enable (E) pin high
451 logic_status_e = 1;
452 set_gpio_value(gpio_e_value, logic_status_e);
453
454 //delay
455 delay_us(2);
456
457 //LCD Enable (E) pin low
458 logic_status_e = 0;
459 set_gpio_value(gpio_e_value, logic_status_e);
460
461 //delay
462 delay_us(2);
463
464 }
465
466 //*****************************************************************
467 //*****************************************************************
468 void LCD_print(unsigned int line, char *msg)
469 {
470 int i = 0;
471
472 if(line == 1)
473   {
474   LCD_putcommand(0x80);              //print to LCD line 1
475   }
476 else
477   {
```

```
478   LCD_putcommand(0xc0);              //print to LCD line 2
479   }
480
481 while(*(msg) != '\0')
482   {
483   LCD_putchar(*msg);
484   //printf("Data: %c\n\n", *msg);
485   msg++;
486   }
487 }
488
489 //***********************************************************************
490
491 void delay_us(int desired_delay_us)
492 {
493 struct timeval  tv_start;  //start time hack
494 struct timeval  tv_now;    //current time hack
495 int elapsed_time_us;
496
497 gettimeofday(&tv_start, NULL);
498 elapsed_time_us = 0;
499
500 while(elapsed_time_us <  desired_delay_us)
501   {
502   gettimeofday(&tv_now, NULL);
503   if(tv_now.tv_usec >= tv_start.tv_usec)
504     elapsed_time_us = tv_now.tv_usec - tv_start.tv_usec;
505   else
506     elapsed_time_us = (1000000 - tv_start.tv_usec) + tv_now.tv_usec;
507   //printf("start: %ld \n", tv_start.tv_usec);
508   //printf("now: %ld \n", tv_now.tv_usec);
509   //printf("desired: %d \n", desired_delay_ms);
510   //printf("elapsed: %d \n\n", elapsed_time_ms);
511   }
512 }
513
514 //***********************************************************************
```

APPENDIX C

Parts List for Projects

Chapter 1			
Description	**Qty**	**Source**	**Part number**
5 VDC, 2A power supply	1	Adafruit (www.adafruit.com)	276
LED, red	1	Jameco (www.jameco.com)	333973
270 ohm resistor, 1/4W	1	Jameco (www.jameco.com)	690726
Boneyard I			
black pelican micro case	1	Pelican Cases (www.pelican.com)	1040
3.3 x 2.1 solderless breadboad	1	Jameco (www.jameco.com)	20601
circuit board hardware mounting hardware	1	Jameco (www.jameco.com)	106551

Chapter 2			
Description	**Qty**	**Source**	**Part number**
100K potentiometer	1	Jameco (www.jameco.com)	94731
270 ohm resistor, 1/4W	1	Jameco (www.jameco.com)	690726
Sharp IR sensor GP2Y0A21YK0F with cable	1	Jameco (www.jameco.com)	164
3.3 x 2.1 solderless breadboad	1	Jameco (www.jameco.com)	20601
1M ohm resistor, 1/4W	1	Jameco (www.jameco.com)	241219
10K ohm resistor, 1/4W	1	Jameco (www.jameco.com)	691104
2N2222 transistor	1	Jameco (www.jameco.com)	38236
white LED	1	Jameco (www.jameco.com)	2160067
62 ohm resistor, 1/4W	1	Jameco (www.jameco.com)	690574

Chapter 3			
Description	**Qty**	**Source**	**Part number**
Dagu Magician robot	1	Sparkfun (www.sparkfun.com)	ROB-10825
aluminum bracket	1	local manufacture	n/a
bracket HW			
- screws	9	Jameco (www.jameco.com)	40970
- nuts	9	Jameco (www.jameco.com)	40943
- washer	9	Jameco (www.jameco.com)	106850
Sharp IR sensor GP2Y0A21YK0F with cable	3	Adafruit (www.adafruit.com)	164
3.3 x 2.1 solderless breadboad	1	Jameco (www.jameco.com)	20601
1M ohm trim potentiometer	3	Jameco (www.jameco.com)	241219
TIP 120 Darlington NPN transistor	2	Jameco (www.jameco.com)	803671
1N4001 diode	10	Jameco (www.jameco.com)	35975
330 ohm resistor, 1/4W	2	Jameco (www.jameco.com)	690742
7805 5 VDC, 1A voltage regulator	3	Jameco (www.jameco.com)	51262
9 VDC, 2A power supply	1	Jameco (www.jameco.com)	1952847

Chapter 4			
Description	**Qty**	**Source**	**Part number**
16 x 2 character LCD, 3.3 VDC	1	Sparkfun (www.sparkfun.com)	LCD-09052
10K potentiometer	1	Jameco (www.jameco.com)	241146
red LEDs	64	Jameco (www.jameco.com)	333973
220 ohm resistor, 1/4W	4	Jameco (www.jameco.com)	690700
74HC244 tristate octal buffer line driver	1	Jameco (www.jameco.com)	251424
74HC154 4-to-16 line decoder	1	Jameco (www.jameco.com)	45401
TXB0108 level shifter	1	Adafruit (www.adafruit.com)	395

Figure C.1: Parts list.

Chapter 5			
Description	**Qty**	**Source**	**Part number**
LM117T voltage regulator	1	Jameco (www.jameco.com)	23579
0.1 uF, 25 VDC capacitor	1	Jameco (www.jameco.com)	151116
5K ohm trim potentiometer	1	Jameco (www.jameco.com)	254669
240 ohm resistor, 1/4W	1	Jameco (www.jameco.com)	690718
1 uf, 25 VDC capacitor	1	Jameco (www.jameco.com)	330431
Prototype Cape kit for BeagleBone	1	Adafruit (www.adafruit.com)	572
thumb joystick	1	Sparkfun (www.sparkfun.com)	COM-09032
Breakout board for thumb joystick	1	Sparkfun (www.sparkfun.com)	BOB-09110
10 ohm resistor, 1/4W	1	Jameco (www.jameco.com)	690380
100 VDC, 5A Schottky diode (IR 50SQ100)	1	Digikey (www.digikey.com)	50SQ100CT-ND
thrusters, Shoreline Bilge Pump	3	Walmart (www.walmart.com)	user choice
Sharp IR sensor GP2Y0A21YK0F with cable	3	Adafruit (www.adafruit.com)	164
1M ohm trim pots	3	Jameco (www.jameco.com)	241219
LED, red	1	Jameco (www.jameco.com)	333973
TXB0108 3.3 VDC to 5 VDC level shifter	1	Adafruit (www.adafruit.com)	395
7404 hex inverter	1	Jameco (www.jameco.com)	49040
7408 quad AND gate	1	Jameco (www.jameco.com)	49146
200 ohm resistor, 1/4W	4	Jameco (www.jameco.com)	690697
TIP 31 NPN transistor	8	Jameco (www.jameco.com)	179354
TIP 32 PNP transistor	4	Jameco (www.jameco.com)	181841
100 VDC, 5A Schottky diode (IR 50SQ100)	8	Digikey (www.digikey.com)	50SQ100CT-ND
470 ohm resistor, 1/4W	4	Jameco (www.jameco.com)	690785
1000 uF, 25 VDC capacitor	2	Jameco (www.jameco.com)	158298
4WD robot platform (DF ROBOT, ROB0003)	1	Jameco (www.jameco.com)	2124285

Chapter 6			
Description	**Qty**	**Source**	**Part number**
LED, red	1	Jameco (www.jameco.com)	333973
LED, green	1	Jameco (www.jameco.com)	34761
220 ohm resistor, 1/4W	1	Jameco (www.jameco.com)	690700
4.7K ohm resistor, 1/4W	1	Jameco (www.jameco.com)	691024
tact pushbutton switch	1	Jameco (www.jameco.com)	199726

Figure C.2: Parts list (continued).

Chapter 7			
Description	**Qty**	**Source**	**Part number**
Boneyard II and III			
LCD7	1	circuitco (www.circuitco.com)	LCD7
BeagleBone	1	multiple sources	---
Mini USB mouse	1	local purchase	---
USB hub	1	local purchase	---
Miniature keyboard	1	Adafruit (www.adafruit.com)	857
Pelican case	1	Pelican cases (www.pelican.com)	1200
HDMI display	1	local retail	---
HDMI to microHDMI cable	1	local retail	---
Weather Station	1	Jameco (www.jameco.com)	
LM34DZ precision Fahrenheit temp sensor	1	multiple sources	155192
weather vane	1	Sparkfun (www.sparkfun.com)	---
LCD, 3.3 VDC, 16 x 2 char	1	Jameco (www.jameco.com)	LCD-09052
120 ohm resistor, 1/4W	8	Jameco (www.jameco.com)	690646
LED, red	8	Jameco (www.jameco.com)	333973
10K ohm resistor, 1/4W	8	Jameco (www.jameco.com)	691104
MPQ2222, general purpose NPN			26446
Speak and Spell			
text to speech chip	1	www.speechchips.com	SPO-512
mini-speaker, 8 ohm	1	Radioshack (www.radioshack.com)	273-0092
10K ohn resistor, 1/4W	2	Jameco (www.jameco.com)	691104
10 uF, 25 VDC	2	Jameco (www.jameco.com)	94212
4.7 uF, 25 VDC	1	Jameco (www.jameco.com)	2143460
330 ohm resistor, 1/4W	2	Jameco (www.jameco.com)	690742
LED, red	1	Jameco (www.jameco.com)	333973
LED, green	1	Jameco (www.jameco.com)	34761
10K ohm trim pot	1	Jameco (www.jameco.com)	254677
LM380N-3 2.5W audio amplifier	1	Jameco (www.jameco.com)	24037
0.1 uF, 25 VDC	1	Jameco (www.jameco.com)	151116
100 uF, 25 VDC	1	Jameco (www.jameco.com)	93761
Dagu 5 ROV			
Dagu Rover 5 tracked chassis	1	Jameco (www.jameco.com)	2143865
9.0 VDC, 2.0A power supply	1	Jameco (www.jameco.com)	1952847
1N4001 silicon diode	2	Jameco (www.jameco.com)	35975
TIP 120, NPN Darlington	2	Jameco (www.jameco.com)	32993
330 ohm resistor	3	Jameco (www.jameco.com)	690742
Sharp IR sensor GP2Y0A21YK0F with cable	3	Adafruit (www.adafruit.com)	164
1M ohm trim pot	1	Jameco (www.jameco.com)	241219
7805 5 VDC, 1A voltage regulator	1	Jameco (www.jameco.com)	51262
LM1084-3.3, 3.3 VDC voltage regulator	3	Jameco (www.jameco.com)	299735
100 uF, 25 VDC	1	Jameco (www.jameco.com)	93761
Stache Cam			
camera, PS3 Eye	1	multiple sources	PS3 Eye
LCD3	1	circuitco (www.circuitco.com)	LCD3
BeagleBone	1	multiple sources	---
Battery cape	1	circuitco (www.circuitco.com)	---
Robot Arm			
Robot Arm-Edge Kit (OWI-535)	1	Jameco (www.jameco.com)	2095023
LM324 operational amplifier	10	Jameco (www.jameco.com)	23683
1N4001 diode	10	Jameco (www.jameco.com)	35975
150 ohm resistor, 1/4W	10	Jameco (www.jameco.com)	690662
1K ohm resistor, 1/4W	20	Jameco (www.jameco.com)	690865
SPST-NO reed relay, 5 VDC, 20 mA, 250 ohm	10	RadioShack(www.radioshack.com)	275-0232

Figure C.3: Parts list (continued).

Authors' Biographies

STEVE BARRETT

Steve is a life-long teacher. He has taught at a variety of age levels from middle school science enhancement programs through graduate-level coursework. He served in the United States Air Force for 20 years and spent approximately half of that time as a faculty member at the United States Air Force Academy. Following military "retirement," he began a second academic career at the University of Wyoming as an assistant professor. He now serves as a Professor of Electrical and Computer Engineering and the Associate Dean for Academic Programs. He is planning on teaching into his 80s and considers himself a student-first teacher. Most importantly, he has two "grand beagles," Rory and Romper, fondly referred to as the "girls."

JASON KRIDNER

Jason got an early start with computing at age 9 programming his mom's Tandy Radio Shack TRS-80. He was also a big fan of Forrest Mim's *Getting Started in Electronics.* Much of his allowance was spent developing projects. He really enjoyed the adventure of trying new hardware and software projects. His goal is to bring back this spirit of adventure and discovery to the BeagleBoard.org community. While still in high school, he worked extensively with AutoCAD as a leak and flow testing company. He joined Texas Instruments in 1992 after a co-op with them while a student at Texas A&M University. He started using Linux at about the same time. Since joining T.I. he has worked on a wide variety of projects including audio digital signal processing, modems, home theater sound, multi–dimensional audio, and MP3 player development.