

A Hardware Interfacing And Control Protocol

Using RobotBASIC
And The Propeller Chip

John Blankenship & Samuel Mishal



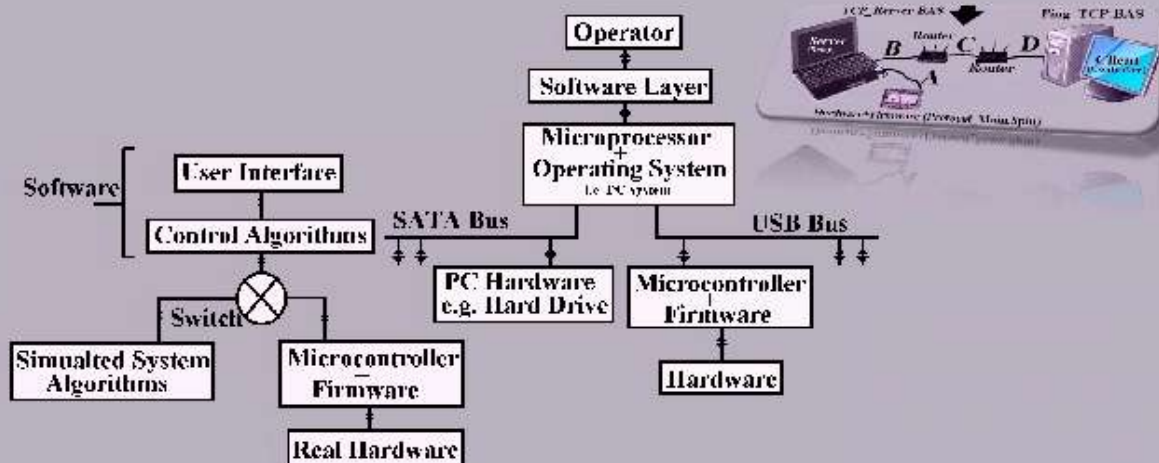
The PC used to allow control of electronic hardware by interfacing through its I/O ports and operating system that allowed deterministic real-time control. The PC grew progressively more complex and powerful at doing GUI, multitasking, 3D graphics, virtual memory management, and much more. But it also became extremely convoluted to program while also prohibiting any access to low-level I/O systems. In the meantime, microcontrollers (μ Cs) were becoming more powerful, easier to program and cheaper. Naturally, engineers and hobbyists are opting to use them for their projects instead of the PC. However, most μ Cs lack the data storage and processing power as well as the user interfacing facilities of the PC and many find themselves wishing to combine the two.

This book aims to show *techniques and strategies* that can be implemented with any μ C and any PC programming language to create a *protocol* for interfacing and combining the two where the shortcomings of each are overcome by the capabilities of the other.

We illustrate the principles with Parallax's multi-cored Propeller – a single chip with eight 32-bit processors running in parallel and sharing a common RAM. With its powerful programming language (Spin) it facilitates implementing *multitasking and parallel processing*, which are at the core of the book's outlined techniques.

RobotBASIC is used as the PC programming language (interpreter/compiler) for its powerful readily usable tools that enable a programmer of any expertise to create GUI programs and to effect hardware communications that would need a high level of programming proficiency in other languages.

To illustrate the strategies with concrete examples we create a few interesting projects using quite a variety of hardware (motors and sensors) that are typical of most devices you are likely to require in an electronics project (e.g. Robots).



A Hardware Interfacing And Control Protocol

**Using RobotBASIC
And The Propeller Chip**

John Blankenship & Samuel Mishal

Copyright © 2011 by
John Blankenship & Samuel Mishal

ISBN-13: 978-1438272849

ISBN-10: 1438272847

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system without the prior written permission of the copyright owner.

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Images of proprietary devices and sensors are reproduced with the permission of the manufacturing companies.

The information and projects in this book are provided for educational purposes without warranty. Although care has been taken in the preparation of this book, neither the authors or publishers shall have any liability to any person or entity with respect to any special, incidental or consequential loss of profit, downtime, goodwill, or damage to or replacement of equipment or property, or damage to persons, health or life, or any costs of recovering, reprogramming, or reproducing any data, caused or alleged to be caused directly or indirectly by the information contained in this book or its associated web site.

The source code for the program listings in this book (and much more) is available to the readers at
<http://www.RobotBASIC.com>

Contents At A Glance

Table Of Contents	i
Foreword	ix
Preface	xi
Acknowledgement	xiii
1- Introduction	1
2- Hardware & Software Setup	15
3- Testing the Hardware and Serial Communications	25
4- Basic Communications and I/O	37
5- Multitasking & Parallel Processing	49
6- A Communications Protocol	103
7- Adding More Hardware	123
8- More Advanced Hardware	183
9- Severing the Tether	239
10- RobotBASIC's Inbuilt Protocol	257
11- Further Improvements	289
Appendix A: Web Links	301
Appendix B: Tables & Schematics	305
Index	317

Table Of Contents

Foreword	ix
Preface	xi
Acknowledgement	xiii
1- Introduction	1
1.1 Why Do We Need the PC?	1
1.1.1 Advantages of Using a PC	4
Effective Operator Interfacing	4
Processing Power	4
Algorithmic and Data Processing Power	4
Utilizing Simulations	4
Access to the Internet or LAN	5
1.1.2 Versatility and Reusability	5
1.2 A Paradigm Shift	7
1.2.1 The RobotBASIC Advantage	8
1.2.2 Various Arrangements	9
1.3 Distributed Parallel Processing	9
1.3.1 A Remote Computational Platform (RCP)	9
1.4 What Will You Learn?	11
1.5 What Do You Need To Know?	11
1.6 An Overview of the Chapters	12
1.7 Icons Used In This Book	13
1.8 Webpage Reference Links in This Book	14
1.9 Downloading the Source Code of the Book	14
2- Hardware & Software Setup	15
2.1 Hardware Setup	15
2.2 Software Setup	20
2.2.1 Ensuring the Propeller Chip is Blank	23
2.3 Summary	23
3- Testing the Hardware and Serial Communications	25
3.1 Testing the LEDs	25
3.2 Testing the Pushbuttons	26
3.3 Asynchronous Serial Communication	27
3.4 Testing Communication with the PST	28
3.4.1 Floating Input Pins	29
3.5 Communicating RB Through the Programming Port	30
3.5.1 A Note About String and Byte Buffers	31

3.5.2 Problem with Resetting the Propeller (use F11 not F10)	33
3.6 Communicating RB Through the Propeller Plug (PP)	33
3.7 Communicating With the PST and RB Simultaneously	35
3.8 Summary	36
4- Basic Communications and I/O	37
4.1 Sending Data From RB to the Propeller	38
4.1.1 Sending Characters, Bytes, Words, Longs and Floats with RB	39
4.1.2 Extracting Numbers from a Received Buffer	40
4.2 Receiving Data with a GUI Display	41
4.2.1 Serial Streaming Speeds and Buffering	41
4.2.2 Hand Shaking	42
4.2.3 Data Remapping	43
4.2.4 An Example of Data Remapping	43
4.3 Sending and Receiving Data with a GUI Display	44
4.3.1 An Exercise	45
The Solution	46
4.3.2 An Exercise in Troubleshooting Weird Problems	46
4.4 Summary	47
5- Multitasking & Parallel Processing	49
5.1 Multitasking Using Interrupts	50
5.1.1 RobotBASIC Simulation of a Microcontroller	50
5.1.2 Using Interrupts in RobotBASIC	51
5.2 Multitasking Using Polling	52
5.2.1 Polling in RobotBASIC	53
5.2.2 Polling on the Propeller Chip	53
5.2.3 Counting Time in Spin	54
Integer Multiplication Overflow	55
Determining the Clock Frequency	55
5.3 True Multitasking with Parallel Processing	55
5.3.1 Using Helper Modules	56
5.3.2 Using Multiple Microcontrollers	56
5.4 Parallel Processing with the Propeller Chip	57
5.4.1 Modularization in Preparation	57
A Variable's Address in Memory (Pointer)	58
A Brief Note About Objects and Methods	59
5.4.2 Initial Multitasking With Polling	59
5.4.3 Achieving Initial Parallelism	61
The Relationship Between Cogs, Methods and Objects	62
Cogs and Stack Space	63
5.4.4 Systematic Debugging of Complex Programs	63
5.4.5 Sources For Obtaining Help With Difficult Problems	69
5.4.6 Parallel Processing Contention for Resources	72
5.5 Objects, Semaphores and Flags	72
5.5.1 Creating Objects	72

5.5.2 Utilizing Semaphores	77
What is a Semaphore?	78
Using a Semaphore	79
5.5.3 Tighter Control With Flags	81
5.6 Parallel-Parallel Processing	84
5.7 Stack Overflow	86
5.8 A Musical Keyboard	87
5.8.1 A Different Way of Sharing RAM	87
5.8.2 Creating Frequencies (Numerically Controlled Oscillator)	88
5.8.3 Testing the Speaker Firmware	89
5.8.4 A Piano Keyboard Player	92
An Exercise	95
Solution	96
5.8.5 Some Thoughts and Considerations	96
5.9 Parallel Programming Can Create Puzzling Errors	97
5.9.1 An Example of a Parallel Processing Trap	97
The Problem	98
The Solution	99
5.9.2 An Example of a Propeller Specific Trap	99
The Problem	100
The Solution	100
5.10 Logistical Planning for Parallelism With the Propeller	101
5.11 Summary	101
 6- A Communications Protocol	 103
6.1 A Better Protocol?	103
6.1.1 A Protocol Enables More Control	104
6.1.2 Specifying the protocol	104
6.1.3 Implementing the Protocol	106
6.2 Fault Tolerance With Recovery	115
6.3 GUI Instrumentations	116
6.4 Versatility of the Protocol	120
6.4.1 A Thought Exercise	120
6.4.2 Another Exercise	121
The solution	122
6.5 Summary	122
 7- Adding More Hardware	 123
7.1 Utilizing the PPDB	124
7.2 Controlling Servomotors	126
7.2.1 A Simplistic Method For Driving a Servomotor	128
The Limitation of This Methodology	129
Using Helper Modules	129
7.2.2 The Propeller Advantage	129
7.2.3 Control With RobotBASIC	130
7.2.4 Using Servo32V7.Spin	133

7.3 Using an Ultrasonic Ranger	135
7.3.1 Showing the Ping))) Values on the PST	136
7.3.2 Using the Ping))) With an RB Program	137
7.4 Using Two Potentiometers	140
7.4.1 Testing the Pots	142
7.4.2 Using the Pots With an RB Program	143
7.5 Putting It All Together	146
7.5.1 Modifying the Others Object	148
7.5.2 Modifying the Reader Object	150
7.5.3 Checking the New Hardware + Firmware + System So Far	150
A Simple Test RB Program	151
Using Program_10_B.Bas	152
An Exercise in Versatility	152
7.6 Adding the Motors Object	153
7.6.1 Further Modifications of the Main Object	155
7.6.2 Verifying the Motors Object	157
Simple Test For The Motors	158
Another Exercise In Versatility	158
7.7 RB Programs to Exercise the Entire System	160
7.7.1 Flexibility, Facility and Simplicity	167
7.7.2 A Really Simple Program	167
7.8 Improving the Motors Object	168
7.8.1 Allowing For Distance and Angle	169
7.8.2 Eliminating Jitter	169
Determining the Command Turnaround Frequency	170
7.8.3 Modifying the System Parameters	171
7.8.4 Timeout Range Remapping	171
7.8.5 Avoiding Serial Communications Timeout	172
7.8.6 The Modified Firmware	173
7.8.7 Testing the New Firmware	177
7.8.8 Robot Moves	178
7.9 An Exercise	180
7.9.1 The Solution	180
7.10 Summary	181
 8- More Advanced Hardware	 183
8.1 Adding a Compass	183
8.1.1 Using the Compass	190
8.1.2 Inter-Cog Communications and Complex Object Interaction	191
8.1.3 Using the Compass Calibration	192
Manual Compass Calibration	193
Automatic Compass Calibration	193
Complexity of Programming the Automatic Calibration:	193
8.1.4 A Simulated Compass Instrument	193
8.2 A Procedural Strategy for Adding Other Hardware	198
8.2.1 Commands in the Protocol So Far	199

8.2.2 A Procedural Strategy For Extending the Hardware	199
Procedure For Adding a New Hardware or Command	200
8.2.3 Controlling Motors Separately	200
Testing the New Commands	204
8.2.4 Controlling a Ping))) on a Turret	204
A Radar Application	207
8.3 Saving The System Parameters to EEPROM	209
8.3.1 EEPROM Limitations	209
8.3.2 Required Changes to The Firmware	211
8.3.3 CRC and Validity Check	211
8.3.4 The New Commands & Firmware	212
8.3.5 Testing the EEPROM Commands	217
8.4 Adding an Accelerometer	219
8.4.1 Adding the Accelerometer Commands to the Protocol	219
8.4.2 Incorporating the H48C in the Protocol	221
8.4.3 Testing the New Command	224
8.4.4 Three Dimensional Animation of Airplane Pitch, Roll & Heading	225
8.5 Using the QTI Infrared Line Sensors	230
8.5.1 The New Firmware	230
8.5.2 Testing the QTI	231
8.6 Adding Sound	232
8.6.1 The New Firmware	233
8.6.2 Testing the Speaker	235
8.6.3 An Exercise	235
8.7 The Final System Firmware	236
8.8 Summary	238
 9- Severing the Tether	 239
9.1 Wireless With RF, Bluetooth or XBee	239
9.1.1 XBee	240
The XBee Advantage	241
The XBee Disadvantage	241
9.1.2 Bluetooth	242
The Bluetooth Advantage	243
The Bluetooth Disadvantage	243
9.1.3 Pure Radio Frequency	243
The RF Advantage	244
The RF Disadvantage	244
9.1.4 Summary of the Wireless Options	245
9.2 Wi-Fi & Internet	245
9.2.1 TCP and UDP Networking Protocols	245
9.2.2 The Topology	247
The Software Side	248
The Hardware Side	248
The Client and the Server	248
The Required Modifications	249

An Example Topology_____	249
A Testing Topology_____	249
9.2.3 IP Address and Port_____	250
9.2.4 The Server Program_____	251
9.2.5 The Client Program_____	252
The Serial Link Program_____	252
The LAN Program_____	252
9.2.6 Running the LAN System_____	254
9.2.7 Converting a More Complex Program_____	254
9.3 Summary _____	256
 10- RobotBASIC's Inbuilt Protocol_____	 257
10.1 The RobotBASIC Simulator_____	259
10.2 How Does RB's Protocol Work?_____	260
10.3 The PPDB Hardware as a Robot Emulator_____	262
10.3.1 Ranger and Turret_____	264
10.3.2 Reading the Compass_____	266
10.3.3 Reading the QTI Line Sensors_____	267
10.3.4 Other Devices_____	268
10.3.5 Handling Errors With the RB Simulator Protocol_____	269
10.3.6 Your Turn to Have a Go_____	271
10.4 The RobotBASIC Simulator Protocol Advantage_____	271
10.4.1 A Case Study_____	273
The Design Advantage_____	273
The Debugging Advantage_____	274
The Exhaustive Testing Advantage_____	275
10.4.2 Implementation Onto the Real Robot_____	276
Sensory Data Mapping To RB's Requirements_____	277
10.4.3 An Exercise_____	279
A Comment About Feedback Control_____	279
10.5 A Simplistic Inertial Navigation System_____	280
10.5.1 The Experiment_____	280
10.5.2 The Results_____	281
10.5.3 The Program's Details_____	282
10.5.4 A Brief Note About Sampling Rates and the Nyquist Limit_____	286
10.6 Summary _____	287
 11- Further Improvements_____	 289
11.1 Extending Our Protocol_____	289
11.1.1 Example of an Extended Protocol_____	290
The Extended Firmware_____	291
Software for Testing the Extended Firmware_____	296
Table of Extended Protocol Commands_____	297
11.1.2 Working the Extended Protocol Over the TCP Link_____	297
11.2 Improvements For the Robotic Control Protocol_____	297
11.3 A RobotBASIC Robotic Operating System (RROS)_____	299

11.4 Summary	300
Appendix A: Web Links	301
Appendix B: Tables & Schematics	305
Final Protocol Objects Hierarchy Map	305
Extended Protocol Objects Hierarchy Map	306
Figure B.1: System's Conceptual Schematic	307
Figure B.2: Propeller Pin Utilization	308
Figure B.3: Hardware Connection Schematics	309
Figure B.4: Photograph of the Final PPDB Hardware Arrangement	310
Table B.1: Final Firmware Protocol Command Codes	311
Table B.2: System Parameters Mapping Formulas	312
Table B.3: Extended Firmware Command Codes	313
Table B.4: RobotBASIC Inbuilt Protocol Command Codes	314
Figure B.5: Protocol State Diagrams	315
Index	317

Preface

The objective of this book is to provide an attainable solution for effecting communications between a PC and electronic hardware. You might wish to have an electronic device do some tasks like switching relays or actuating some motors and reading some transducers, while the PC does the Artificial Intelligence (AI). You might have a system that carries out some complicated tasks and you wish to use the PC to display data and control instrumentation using an effective and ergonomic GUI. Perhaps you have distributed nodes of sensors collecting data over a wide area and you want to use the PC as the central controller for data collection, storage and analysis (see Chapter 10) with the nodes communicating with the PC over wireless links or even across a LAN, WAN or the Internet (see Chapter 9).

Until now, anyone creating a control application had to make a choice; use the microcontroller or the PC for their projects. As a student, hobbyist, or engineer you most likely have wished to utilize the PC's capabilities in your projects - keyboard, data storage, Internet connectivity, arrays, floating point math, graphical user interface (GUI), 2D and 3D graphics and more. While some of the more powerful microcontrollers (e.g. the Parallax multi-core Propeller Chip) can actually do many of these things, it can be complex to implement any one of them, let alone several at the same time. In contrast, these features are already available and readily usable on a PC.

In the past, using a PC in electronic control projects was common practice and quite easy to do. The PC used to have I/O ports that were easily usable to interface with electronics projects and it was easily programmable to do deterministic timing without being unpredictably preempted by a complex multitasking operating system performing a myriad of other jobs. The PC grew progressively more complex, powerful and sophisticated with GUI, multitasking, 3D graphics, virtual memory management, and much more. But it also became extremely convoluted to program while also prohibiting any access to low-level I/O systems; as a result most are nowadays only using the PC as a cross compiler to upload programs to a microcontroller.

In the meantime, microcontrollers (μ Cs) have been steadily advancing in capabilities and becoming more powerful, easier to program and much less costly. Naturally, engineers and hobbyists are opting to use them for their projects instead of the PC. However, most μ Cs lack the data storage and processing power as well as the user interfacing facilities of the PC and many engineers and hobbyists find themselves wishing to combine the two. Even though μ Cs are preferred for controlling electronics hardware and for robotics projects, it is evident that microcontroller-based applications can benefit greatly from the PC's capabilities.

This book will detail a new conceptual model as a design strategy for incorporating the PC with μ Cs in your projects so that you no longer have to choose between the two (see Chapter 1). We aim to show *techniques and strategies* that can be implemented with many μ C and most PC programming languages to create a *protocol* for interfacing and combining the PC and μ C (or multiple microcontrollers) where the shortcomings of each are overcome by the capabilities of the other. We will expound *methodologies* for implementing a *firmware layer* on top of any amalgamation of hardware to act as a conduit for a *software layer* to carry out *real time control* of the overall integrated system. The details and particulars of the hardware and software are incidental. Rather, what is of primary significance is how the combination implements a *communications protocol* (Chapter 6).

Although we utilize a specific microcontroller ([Parallax multi-core Propeller Chip](#)¹) and a specific PC programming language ([RobotBASIC](#)²), you should be able to utilize the strategy and methodology developed here as a *template* to create your *own* system of any combination of hardware controlled by a PC software of your design through a protocol of your devising implemented by a firmware layer applicable to your system.

The Propeller Chip – a multi-cored processor in a single chip with eight 32-bit processors running in parallel and sharing a common RAM – with its powerful programming language (Spin) facilitates implementing *multitasking and parallel processing*, which are the crux of the book's outlined techniques (see Chapter 5).

RobotBASIC is used as the PC programming language (interpreter/compiler) for its powerful readily usable tools that enable a programmer of any expertise to create GUI programs and to effect hardware communications that would need a high level of programming proficiency in other languages. Another advantage of RB is its suite of commands and functions that can be used with great ease to carry out communications over a LAN, WAN or the Internet. As we will see in Chapter 9, the ease with which we accomplish control over the network of the complex system developed here would have required a book on its own to explain had it been implemented in another language.

To illustrate the strategy with concrete examples we use a variety of hardware modules that are typical of most devices you are likely to require in a electronics projects (see Chapters 7 and 8). Despite the fact that the hardware is most often used for robotic projects, it is sufficiently general to be of utility in numerous systems because the devices typify most of what may be utilized with a microcontroller.



We hope this book will give you a running start on the way to achieving your own system using a PC and microcontrollers to create complex electronics control projects. Whether you want to build a Robot (Chapter 10) or an RUV (Remote Underwater Vehicle) using parallel processing (Chapter 5), with GUI Instruments such as a Compass and Accelerometer and SONAR (Chapters 7 and 8), or you wish to collect data from distributed loggers over the Internet (Chapter 9) and want to store the data on a central PC to graph and analyze the collected information (Chapter 10), we hope you will find the techniques and projects in this book helpful in accomplishing your own projects.

Introduction

A While back, the PC, with its parallel port, ISA and PCI buses, and serial port provided a viable and powerful as well as moderately easy to program controller for electronic hardware projects. The PC had easily expandable I/O buses and with the microprocessors of the time it was easy to implement Assembly or higher level programs that utilized the *interrupts* ability of the processor to achieve multitasking and *deterministic real time control*, where you could create accurate and repeatable signals. With the support of the Operating System (OS), resources such as File I/O, Graphics, Mouse, Speaker, and so forth were easily accessible and of major utility to an electronics hardware control project.

With the ever-increasing tighter control by the OS over the facilities of PC and its processor, programming hardware I/O on the PC became progressively more convoluted with each new version of the Windows OS. Furthermore, the fact that the operating systems these days are continuously performing tasks in the background makes it extremely hard to implement deterministic real time control. To aggravate the situation even further, the operating system now prohibits and denies the use of the PC's hardware through programming languages without the use of special SDKs (Software Development Kits). And to add further difficulties, PC's no longer have any readily usable serial ports or parallel ports, and the bus is almost impossible to use.

All this means that engineers desiring to use a PC to control electronics systems have to resort to using specialized hardware and software designed by companies that have the inside knowledge of how to bypass the OS obstacles. For example, LabVIEW™ provides hardware products that can be used on the PC's I/O bus along with a proprietary specialized programming interface to utilize these devices. Such systems lack the versatility and flexibility desired by many engineers, and are usually overly costly. You, of course, can still use the PC to do hardware interfacing if you have the appropriate SDKs and are versed with Visual C++ and the COM model and know how to use DLLs and .NET programming and have a degree in computer science with many years of experience and so on and so forth.

Due to the tremendous difficulty in bypassing the OS obstacles surrounding the PC's hardware, many hobbyists find it exceedingly prohibitive in time and cost to program a PC for interfacing with external electronics. With the availability of powerful and easy-to-use microcontrollers, many hobbyists find it a lot easier and cheaper to use them for their projects and nowadays are mostly using the PC only as a cross-compiler to program the μ Cs through IDEs (Integrated Development Environments) provided by μ C manufacturers. This is a very regrettable situation because the PC can be an extremely important and utilitarian component in a hardware control project for numerous compelling reasons.

1.1 Why Do We Need the PC?

Consider the following program and its resulting output shown in Figure 1.1.

```
Inlineinputmode
Input "Enter your name:",Name
Input "Enter the year you were born:",BYear
Age = round(year(now())-ToNumber(BYear,0))
Print "hello ",Name," you are ",Age," years old"
```

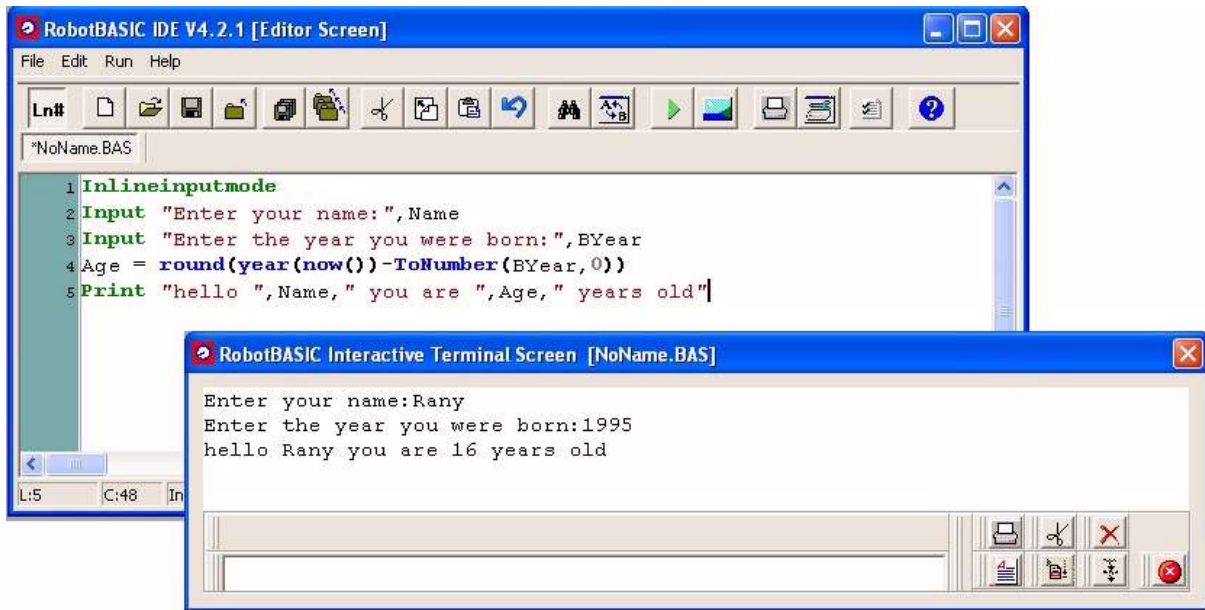


Figure 1.1: A simple user interface program

Despite the fact that this program is only 5 lines of code, and despite its simple actions, you would be extremely hard pressed to produce an equivalent process using a microcontroller alone. What most people take for granted about using a PC system with an appropriate program such as the above, are the numerous support systems that underlie the resulting overall interaction. The program may seem simple at face value but in fact it is an extremely complex one. As a user of the programming language (e.g. RobotBASIC) you did not have to concern yourself with a plethora of details. These details are neither trivial nor simple. If in reality you had to implement all the processes that enable the above program to work you might have to spend months and you would have to be a software engineer of the highest caliber.

Let's examine what the 5 lines of code accomplish. The first line is not important for now. The second and third lines each carry out two actions. They display a message on the screen and then wait for an input from the user. In order to do these two deceptively simple actions, numerous calls to underlying OS facilities have to be performed. The program has to request from the OS permission to output to the screen. It has to also tell the OS which window it is outputting to and what coordinates. It has to tell the OS what font and what color to output to the screen. All this on top of what the content of the output string is. This content itself had to be retrieved from its location in the RAM of the PC. This action of acquiring the text from some memory area in itself requires numerous calls to OS facilities. Waiting for a user input, again, necessitates countless calls to the OS to be able to interact with the keyboard and interpret its input. More memory actions have to be performed in order to store the user key presses and collate them into a string.

The code in the fourth line performs a staggering amount of work. It may not seem so when you look at it. At the logical level the code converts the user's birth year, inputted in text, to a numerical value and defaults to 0 if it is an invalid value. It then obtains the current year and subtracts the given year and stores the result in a memory variable.

On the hardware level it is prohibitive to list here what actions are needed. However, consider the facilities made available in this line. We had to determine the current date. We had to figure out the year from that date. We also had to perform the action of converting the user's input from a text representation of the year to a numerical value. This is a very involved algorithm in and of itself. You would need a program bigger than the original one just to perform this action. To be able to do the mathematical calculations, again, the program has to make many calls to the OS. This fourth line of code alone requires a program of thousands of lines if you were to implement all the necessary low-level functions it performs. The code in the fifth line is similar to the second and third in that it outputs to the screen and it also needs a lot of background processing to be able to concatenate all the required output strings and number.

The above description does not even scratch the surface of what the RobotBASIC language is in fact performing for you when you type and run the simple 5 lines of code above. This is precisely the power and utility of the PC when used with an appropriately simple to use yet powerful language. If you had to program the above on a microcontroller, you as the programmer would have to take care of *all* the necessary sub-systems to be able to accomplish the transaction with the user, the keyboard, the screen, the real time clock and the math processor.

Notwithstanding all the complexity, the above program can still be achieved on a capable microcontroller. This is because the input and output mechanisms are relatively simple. Imagine now if the program's action was similar to that shown in Figure 1.2 (also see Figures 8.11, 8.15 or 10.2). The GUI (Graphical User Interface) alone would be impossible to achieve on almost all the microcontrollers available nowadays. Even if you did achieve a modicum of what can be accomplished on the PC, the amount of work would be prohibitive and then there will be hardly any memory or I/O lines left over to do anything else. And the final outcome will not even begin to approach the quality attainable on a PC.

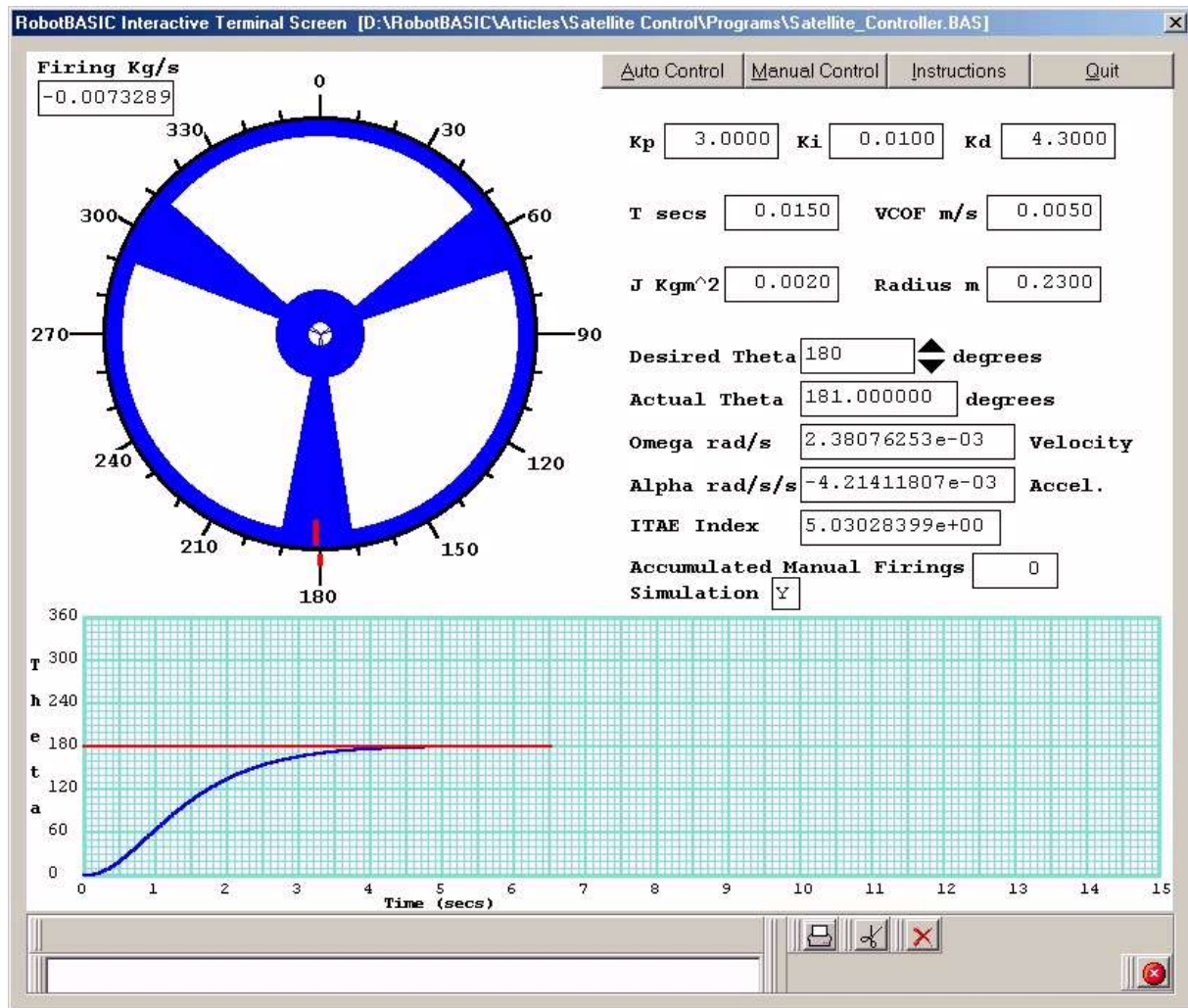


Figure 1.2: An example of a GUI program in action



Many of the limitations of microcontrollers discussed below do not apply to the Propeller multi-core Chip, as you will see throughout this book. You can do things with the Propeller never thought possible with a microcontroller. It has eight processors (cogs) in one chip that can operate simultaneously, either independently or cooperatively, sharing common resources through a central hub. In fact the PPDB (Figure 2.2) used in later chapters can be made into a PC more powerful than some of the PCs of not too long ago.



Many modern microcontrollers are also very powerful and all the work we do in this book with the Propeller is very much applicable to these microcontrollers. The techniques we will elaborate in the next ten chapters are just as achievable with these capable microcontrollers as they are with the Propeller.

1.1.1 Advantages of Using a PC

What is not widely appreciated is that even though microcontrollers (μ Cs) are easy to program and are seemingly able to do just about anything, in fact they are very limited when compared to microprocessors (μ Ps). Most hobbyists will not usually be hindered by the limitations since their projects are often not overly complicated and often a single μ C is sufficient for most projects.

Effective Operator Interfacing

A μ C is just that – a **controller**. It is designed with the express purpose of controlling hardware. A μ C is superb for controlling digital I/O and even in certain cases some analog I/O as well. If a project requires hardware control without much user interaction then a μ C is the best possible choice. However, if the project requires more extensive **operator interfacing and data processing** then you need to use a μ P, which is a lot more suited to doing just that – **processing**.

Using a PC with its graphics capabilities in a control project you can create an **ergonomic operator interface**. You can use GUI components and 2D and 3D graphics to provide the user with **intuitive** and effective **feedback** and **control** over the system (see Figures 1.2, 8.11 and 8.15).

Processing Power

A μ C is limited in the amount of RAM and ROM available to it. Unlike a μ P, which is designed to process data, a μ C does not make available its **memory buses** and has a fixed memory. This means that there is no way to expand the memory available to it except by using some of its I/O lines. Indeed, you can, with ingenuity and sufficient finagling, make a microcontroller achieve some impressive acts. Even so, that is not what a μ C was designed for. The aphorism “**Horses For Courses**” comes to mind here. Of course you can use a screwdriver as a hammer, but think how much better it would be to use an actual hammer.

Algorithmic and Data Processing Power

Most μ Cs are limited in their ability to manipulate **arrays** and perform **floating-point** as well as other high-level math operations. Even simple **multiplication** and **division** are limited or in some cases hard to implement. Simple projects may not require many mathematical calculations, but more complex projects will usually require the processing power of a PC.

To accomplish most **Artificial Intelligence (AI) algorithms**, structures such as **Multi-Dimensional Arrays, Files, Databases, Queues, Lists, Binary trees, Graphs, Stacks, Searching, Sorting, Fast Fourier Transforms** and much more are necessary. There are not many μ Cs that can be programmed to handle such constructs at the level required by even simple AI projects. Consider for instance the case of controlling a robotic arm. Most μ Cs would not even approach adequacy for some of the number crunching required to calculate the forward and reverse kinematics of a 5-degrees of freedom arm. Calculating the Jacobian alone would task the majority μ Cs to the extreme.

Utilizing Simulations

An effective and powerful design methodology in engineering is to use **simulations**. Simulations provide an extremely effective method for testing a system before spending much time and money building the real hardware. A simulated system allows engineers to try out various algorithms and ideas, to examine what-if situations and to hone the control algorithms. All this can be accomplished with safety and minimal expenditure.

Once a simulation is perfected it can be used to train operators while the physical system is being built. A simulation enables catastrophic training scenarios to be thrown at the operator with none of the obvious ramifications. Think of a flight simulator where a pilot can fail and crash and still go back home that evening to his family unscathed.

Once the hardware system is available the very same programs that controlled the simulations can be used to control the real hardware instead of the software simulation algorithms that emulated the hardware. The time spent developing the simulation would have been efficiently used and becomes an integral part of the overall design process. Operators do not need to be retrained and there is no need to translate the control algorithms to the native language of the hardware microcontroller. Moreover, the control algorithms can be as complex as needed without being hindered by limitations in the processing ability of the microcontroller. No new equipment is required to effect the user-interface since the same PC systems used for the simulation are used with the real hardware.

An example of such a system is shown in Figure 1.2 above; also see Chapter 10 for more examples. Figure 1.3 below is a schematic layout of how this can be conceptually achieved. If you look at Figure 1.2 on the middle right hand part of the image, just above the graph area, you would see a box labeled *Simulation*. If this box is set to N (no) then a user interacting with the system would be in fact interacting with the real hardware being driven by the program. If the box is set to Y (yes) then the interaction would be with the algorithms that simulate the hardware. Notice that the very same user interface is used for both the real and simulated interaction.

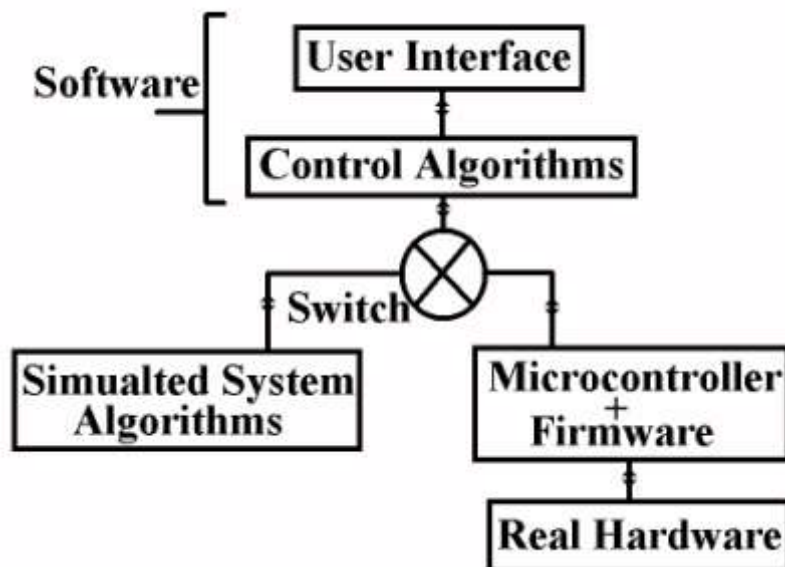


Figure 1.3: Simulation/Real Hardware Control Conceptual Schematic

Access to the Internet or LAN

Most μ Cs do not have the capacity to provide a TCP or UDP stack. To enable a microcontroller to communicate over the Internet one has to use a specialized module. This adds extra expense to the project and may not be a versatile option. If we use a PC in the project then the PC can also act as the conduit for achieving Internet communications using a wired or wireless link (Wi-Fi). Additionally if the link between the PC and the hardware is also wireless (XBee), then the hardware would be able to communicate through the Internet or LAN completely wirelessly. See Chapter 9 for how to implement such a system and see Figure 9.7 for various layouts.

1.1.2 Versatility and Reusability

What makes the PC such a versatile device? A PC is a Rolodex, a diary, a personal planner, a book, a typewriter, a CD player, a DVR – the list is endless. However, when you first start the machine it is none of that. What makes it become all these things is its ability to run programs that make it accomplish the tasks necessary for acting as the appropriate analogue.

What is a PC? It is a set of **hardware** with a capable μ P appropriately programmed with the right Operating System (**firmware**). When you want to make the PC perform a particular task you give it a series of instructions (**software**) that

it can understand. This software tells the PC what hardware to use as well as how and when in order to be able to emulate the analogous tasks. See Figure 1.4.

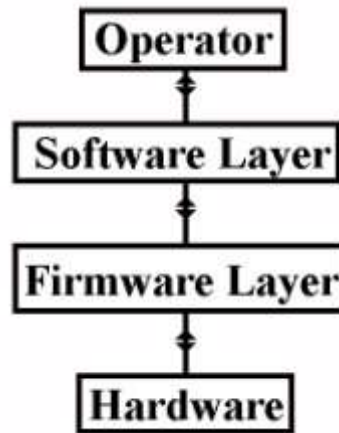


Figure 1.4: Conceptual model of a PC system

This conceptual model is the secret of the power and *versatility* of a PC system. Imagine if every time you wanted to make the PC perform a different task to what it is currently doing you had to:

- ☐ Fire up another machine.
- ☐ Load a program on the machine.
- ☐ Connect the PC to the machine.
- ☐ Write a program in the machine that has *all* the firmware as well the software required.
- ☐ Compile them.
- ☐ Upload the compiled result to the PC.
- ☐ Unplug the PC.
- ☐ Run the PC.
- ☐ Test if the new software is working.
- ☐ If it is not repeat the above steps after having first used the other machine to fix the problem.
- ☐ If you need to do a new action on the PC, repeat the above steps having first devised the necessary new software for the action.

You can imagine that not many people would be using computers. The above process would soon get to be too irksome to say the least. Yet, if you have not noticed, that is exactly what we do every time we want to run a new program on a μ C. Notice too, that it is not just the software that we upload to the μ C, rather it is the firmware as well as the software. We don't normally think of it that way. We think of the uploaded program as one program. However if you are using things like LCDs, Key Pads, Serial Ports and so forth, then every program you do has a common set of basic underlying subroutines that make these devices function. Most often you just cut and paste or `#include` these routines into your program. But these subroutines in fact constitute a firmware. Your new code would be the software.

We also tend to think of the μ C as being independent of the PC. But in reality it is not. The μ C would not be versatile if we did not have the PC. We would not be able to quickly and easily change its action (load it with new software). So in fact, ***the PC is a crucial and integral part of the life cycle of a μ C system.*** Read the previous statement again. Mull over it for a few minutes. What makes the μ C versatile and useful is the PC. Without the PC, using a μ C would be quite aggravating and perhaps impossible.

If what you need is to make a μ C based system be a versatile one, you will need the PC. However, with the traditional method of using a PC just as a cross-compiler and IDE platform to program the μ C, the PC constitutes only an *implicit function* of the final resulting system once the μ C is carrying out the designed task.



Don't think of the PC as just a μ P. The PC is a **complex combination of systems** that are the culmination of over 60 years of engineering expertise by thousands of innovators. Every time you use a PC you are "riding on the shoulders of giants". By opting to incorporate a PC in your design you are starting from an advanced position instead of from scratch.

1.2 A Paradigm Shift

What we are proposing in this book is a new **paradigm**. What we want is to **make the microcontroller an explicitly integral component of an overall PC system**.

On a PC when we want to load new software we do not need another machine to do so. We even can use a programming language on the PC itself, to write a new software if a commercial one is not available. However on a μ C system in the traditional way we use it, this would not be possible. Nevertheless, if we expand our **conceptual perception** of what a μ C system is and regard the PC as an integral component in the system then in fact we can write software and run it on the μ C without relying on an external device, since now the PC is actually part of the system.

This new paradigm is not as simple as just **thinking** of the PC as important. Rather, it is a concrete and decisive action that has to be taken to realize the benefits of such a new concept. We need to setup the μ C to be a sub-system of the PC as an overall unit. Just like the PC has a hard disk or a sound card or an LCD screen, so will the μ C be yet another hardware sub-system in the PC's repertoire of peripheral devices, much like a printer or a scanner and so forth,

What you may not have actually realized about devices that constitute a PC system is that in fact many of them have their own μ Cs onboard. Hard Disks these days are almost standalone devices. In actuality, PC peripheral devices communicate with the μ P using a μ C (or even a μ P) of their own, utilizing the **SATA bus lines**. **USB ports** are nothing more than another kind of **bus line** to the PC's μ P.

In concrete terms, the μ C + firmware become a substitution for the old parallel ports and serial ports. All you need is the right programming language and you can successfully make the PC an electronics hardware control and experimentations platform just like in the old days, but with even more power and versatility as well as functional utility. See Figure 1.5.

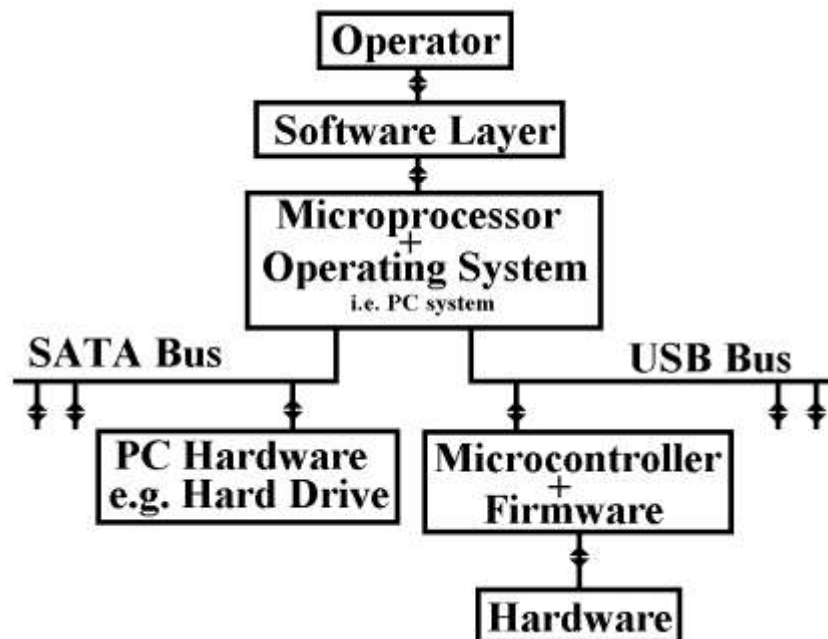


Figure 1.5: Conceptual model of the new paradigm



The PC alone is no longer a viable hardware controller. The μ C alone is a poor user interfacing and data processing platform. However, with our *new paradigm* and using RobotBASIC to be able to communicate through the USB ports with the appropriate software and firmware we can convert the PC and μ C *together* into a very powerful, efficient, and versatile hardware control and experimentation platform with ergonomic and effective user interfaces. The next ten chapters of this book will show you how to program a μ C with the right firmware to make it an integral subsystem of the PC. This way you can carry out control of electronic hardware as easily as using a PC.

1.2.1 The RobotBASIC Advantage

As you saw above, all you need to implement the new paradigm is a programming language that makes it easy to communicate with the microcontroller. There are a plethora of languages out there that can *eventually* achieve this. Many of them however are complex and have very steep learning curves. Many of them also require a lot of resources on the PC and cannot be used on the fly. They need installation and cannot be used from such devices as a flash drive or a CD. All of them are quite powerful; however, you would need lots of experience to be able to use them at a functional level.

What is desirable is a language that can be used at any level of expertise and yet produces programs at a level a professional in other languages would produce. [RobotBASIC](#)² (RB) is one such language. There are numerous advantages to using RB listed on its web site. The ones of immediate import to our paradigm shift are the ability to:

- Communicate with devices on the USB ports such as a microcontroller or XBee transceiver.
- Communicate with Bluetooth devices.
- Communicate Over the Internet or LAN using TCP or UDP.
- Fully Control the U4x1 family of devices from [USBmicro](#)³ (see later).

With RB's 2D and 3D graphics engines and its extensive GUI components, combined with its numerous commands and functions for math and matrices and File I/O (low and high level) as well as the tremendously easy syntax, RB enables even the most novice programmer to create programs for controlling hardware with ergonomic and professional looking interfaces (see Figure 1.2 or 8.15)

Another major advantage of RB is its integrated *robot simulator* as well as its associated *robotic hardware communications protocol*. See Chapter 10 for more details on both these systems. As we saw in the previous section, a simulation should be an indispensable part of the design cycle carried out by a prudent engineer.



One very effective and convenient method to implement the new paradigm explained above is through the U4x1 USB I/O family of devices from [USBmicro](#)³. RobotBASIC has an extensive set of functions that enable easy use of the U4x1's port I/O, SPI and I-Wire communications, control of two Stepper Motors and control of High-Voltage-High-Current built in Relays. This family of devices is an excellent and powerful substitute for a μ C in certain classes of projects (or to use in conjunction with a μ C) that you may want to consider. In addition to the information resources available at the USBmicro web site, we have an [in-depth tutorial](#)⁶⁶ teaching how to use these devices on our web site.



The RobotBASIC IDE and compiled RB programs can run under any Windows OS version from 95 to W7 from a CD or Flash drive with no installation required. Also RB makes it easy to interact with the parallel port and ISA/PCI buses on older PCs. This makes RB ideal for making use of old PC's and giving them a new life as electronics hardware experimentation platforms, instead of a reason for spouses to complain about them taking too much storage space.

1.2.2 Various Arrangements

There are various alternatives for how to incorporate a μC as an extension of the PC's hardware:

1. Laptop, Notebook or Desktop directly wired through a USB to an appropriate USB to TTL Serial converter which then is wired to two of the μC 's I/O lines as Rx and Tx lines.
This option is not very mobile if you use a Desktop, but even with a Notebook it may be too bulky for some situations (e.g. a small robot). The U4x1 devices would be an excellent option here as well.
2. A PC Motherboard + SD card or USB flash memory to hold the OS and software, directly wired to the μC as in option one above.
This option is mobile but not very convenient if you require user interfacing and active real-time system monitoring. However, this option is great for mobile applications that require the computational augmentation the PC motherboard provides. You can also combine it with option 3. Again, the U4x1 devices would be an excellent option here too.
3. The PC is connected to a wireless transceiver (see Chapter 9) through its USB port. The μC is also connected to a compatible transceiver through two of its I/O lines. The transceivers act as a wire replacement between the PC and the μC .
This is the most versatile alternative and it has the power of being mobile and at the same time providing user interfacing and real time systems monitoring. This would be the option of choice for a distributed or a mobile system (e.g. robots or monitoring stations).

1.3 Distributed Parallel Processing

Complex engineering systems comprise numerous subsystems that can be thought of as a collection of subtasks. You should divide a complex system into simpler subsystems (just like you do for a complex programming project). Each subtask can be controlled by a dedicated μC along with some additional circuitry. The overall project is coordinated by the PC as a master controller which communicates with the various subordinate μC s. The *distributed processing* provided with this *divide-and-conquer* strategy, allows the PC to require less I/O conduits than would have been needed if it had to control all the sub-processes directly. Also due to the *parallel processing* provided by the various μC s, *multitasking* is readily achievable (see Chapter 5).

On the PC you can have an overall controller software program or even multiple programs running in parallel, with each program controlling one USB port that carries throughput to the μC . These software programs can also communicate with each other using hard disk files or the UDP protocol (despite being on the same PC) to transfer data between each other if the need arises.

The PC provides the *AI Brain*. The microcontrollers only deal with reading *transducers* and activating *actuators* but not with why they need to do so. The PC decides *what* and *why* and *delegates the how* to the microcontrollers. The μC is programmed with the appropriate firmware to be able to communicate with the PC software and to be able to control the various hardware components it is dedicated to. The firmware is therefore quite simple with only sufficient complexity to independently control its subtask according to parameters transmitted to it by the PC software.

1.3.1 A Remote Computational Platform (RCP)

A PC used in the manner described above can be easily converted into a *Remote Computational Platform (RCP)* by using wireless or Wi-Fi connections to all the subordinate μC s. This provides levels of functionality and diversity that facilitate many interesting possibilities.

The RCP also acts as an *operator interface* node that provides operators with information about and control over the system and with the ability to *reconfigure the system dynamically* (i.e. while it is working) and/or to *override* the system's automatic actions when required. This remote control can also take place across the Internet (see Chapter 9).

There are numerous advantages in having an RCP. Think of Planetary exploration. If you have an orbiting RCP that controls multiple surface Rovers, you can simultaneously explore multiple regions, rather than being limited by one

explorer. Also each individual explorer is simple and expendable. The RCP stays “safe” up in orbit and does not incur the possibility of damage during landing. Since there are numerous explorers, there would be no problem if one or more are damaged during the landing. You will still be able to achieve the mission or reassign another rover to take over the task of its defunct “sibling”. With this option a robotic platform can be kept small. Only the sensory and actuations systems are needed onboard and perhaps some gyros and accelerometers – in the case of airborne or seagoing platforms – for doing attitude control or an INS (inertial navigation).



⚠️ Also with the RCP option, once the robot is configured and its onboard microcontrollers programmed, it never needs to be tampered with again. ***All the work can now be done through the PC to make the robot do different tasks and actions*** depending on the projects. You can even ***reconfigure*** the robot in ***real time*** while it is ***in the field*** still doing its work. You can convey to it imperatives to make it alter its previously assigned behavior remotely while it is still in the field.

Some *lateral thinking* and a *paradigm shift in conceptualization* are necessary to appreciate this kind of robot. Most people think that an autonomous robot has to be human like. We humans do not have an RCP – or do we; food for thought. An autonomous robot is still autonomous even though it is using *additional not onboard brains*.

Another advantage of this idea is that you can have multiple robots sharing the same RCP to act as a *hive* or *matrix* of robots. They can then intercommunicate and be orchestrated all at the same time through the RCP. Moreover, the RCP can provide information to the hive that is otherwise not possible to obtain by the individual robots. Imagine having a robot able to access the Internet to collect some data it requires (e.g. GPS augmentation, weather data, satellite imagery). Think of a hive that is distributed over remote places but yet can communicate and orchestrate actions by using the Internet as a communications link. Researchers call hive members “Agents”. Currently this kind of structure is under intense research. **RESISTANCE IS FUTILE.**

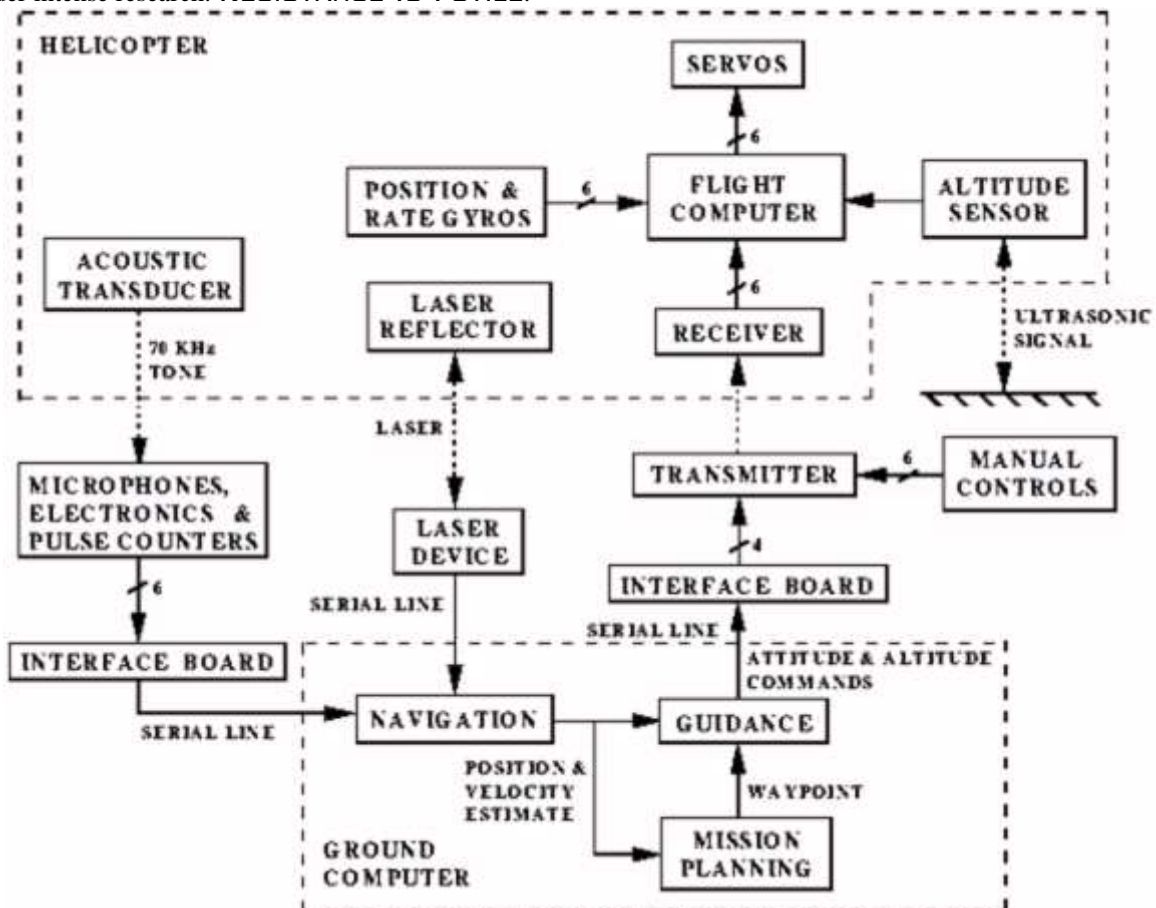


Figure 1.6: An RCP research system’s conceptual model.

See Figure 1.6 for the conceptual model of an RCP used by [a researcher at MIT](#)⁴ to implement a helicopter control system. The controller onboard the helicopter is able to autonomously maintain the vehicle's attitude. But to go places or to change altitude the decisions come from the ground controller (PC). The RCP communicates through three serial links with three μ C-based devices. One uses RF transceivers. Another uses a LASER link (similar to Infrared). The third uses Ultrasound. Notice how the RCP provides the *mission planning* aspect – in other words the AI.

1.4 What Will You Learn?

The goal of this book is to show a *strategic methodology* for implementing the new paradigm expounded in Section 1.2. We will gradually evolve a series of programs into a capable and functional *firmware layer* that can be used to carry out the *communications protocol* between the μ C and the *software system* running on the PC.

Along the way we will use an amalgamation of hardware (such as an accelerometer, ultrasound ranger, infrared line sensors, compass, servomotors, potentiometers and more) to demonstrate the utility of the *overall concept* by controlling the hardware through software programs to carry out *real time control* of the *integrated system*. Even though we are using specific hardware and a specific μ C and PC programming language, the aim is not to teach these particular systems or even the particular protocol. Rather the aim is that you would be able to *utilize the methodology as a template to achieve your own requirements*.

The *multitasking and parallel processing* concept is of paramount importance and you will need to use it regardless of the particulars of your system. This book will utilize the concept almost right from the start.

There are four levels at which a person can acquire a new skill:

- **Rote:** where a skill can be repeated only by emulation, with no understanding for why it is performed so.
- **Understanding:** where one still can only repeat the skill but now with an understanding for why it is applied within the particular application with which one is familiar.
- **Application:** where the skill can be applied in different situations and with understanding. However, there is no additional innovation of technique.
- **Correlation:** where one can adapt the skill to apply it to new applications in an innovative manner.



We hope this book will induce you all the way to the correlation level where no matter what new situation you face you would find the information acquired here an inspiration for you in creating your own unique and innovative solutions.

1.5 What Do You Need To Know?

This is not a book about learning how to program RobotBASIC or Spin. Nor is it about how to use particular devices. All these are skills best acquired from the resources mentioned in Appendix A. However, this book is about how to create a system to allow a PC to control electronics hardware in an efficient and useful way.

You are not expected to be an expert in any of the systems used in the book. In most cases we do show enough detail to be useful even to a novice. However, you are expected to be at a level of knowledge where you can read code and discern the algorithms in it. If you are not familiar with certain syntax, you are expected to read the manuals and learn about the particulars you are not sure about.

We will use RobotBASIC (www.RobotBASIC.com) and Spin (www.Parallax.Com/Propeller) at an intermediate to advanced level and there will be a few programming techniques and tricks to achieve efficient results. Some of these may be explained in some detail. However, you are expected to be sufficiently versed in both languages to be able to follow along with the explanations since not *every* detail might be expounded. You should be familiar with these

languages, at the very least, beyond the beginner level. You can find tutorials for both at their respective web sites (see items 1, 2, 62, 63, 65 in Appendix A).

In RobotBASIC we will use techniques for serial communications and for communication across the Internet. Detailed Tutorials for both these can be found at our website (see items 18-20, 57 in Appendix A). Additionally there are numerous YouTube video tutorials about the RobotBASIC language (see item 63 in Appendix A).

There are also five other books that teach RobotBASIC at an advanced as well as a beginner and intermediate levels. There are many links on the RB web site but also see item 62 in Appendix A. The book *Hardware Interfacing With RobotBASIC, the Fundamentals* is designed to be a precursor to this book for beginners(search for it on www.Amazon.com or see the link www.RobotBASIC.com).

Similarly, for the Spin language as well as most of the hardware used here, there are tutorials, specification sheets, example code and much more on the Parallax web site (see Appendix A). Also see www.parallax.com/propeller/qna and www.parallax.com/propeller.

You are expected to be able to read schematics and translate them into physical wiring arrangements. We assume that you are versed with electronics hardware and are able to determine what you need from specification sheets and other information resources that would augment whatever detail we give in this book.

1.6 An Overview of the Chapters

In **Chapter 2** we list the required hardware and software, so you can collect the necessary equipment before you start building the projects and prepare it to be ready for later chapters.

In **Chapter 3** we develop programs to test the initially simple hardware setup. This verifies the hardware and software systems and provides a working starting point. Also it provides base line programs for carrying out serial communications that can be evolved as we progress through the book. We also learn certain important facts about serial communications buffers.

In **Chapter 4** we develop further sophistication in the software establishing some GUI programming techniques. We further develop the serial communications techniques required to achieve effective interaction between the PC and the Propeller. We also learn about some pitfalls in serial communications and how to avoid them using software handshaking and how to use software to complement and enhance the hardware and to work around certain limitations that may arise.

In **Chapter 5** we delve into the all-important concepts of Multitasking and Parallel Processing. We look at the three different techniques of Polling, Interrupts and Parallel Processing. We learn about timing and timers in RB and Spin. We learn about memory sharing using pointers in Spin. We also learn about semaphores and flagging. For examples of parallel processing we utilize frequency generation and use that to create musical tones and tunes on a speaker. Additionally, we learn about avoiding some elusive traps while utilizing parallel programming in general and the Propeller Chip in particular.

In chapters prior to **Chapter 6**, we utilize *ad hoc protocols* to effect the communications as required by the systems being developed at the time; every program had a different technique and a different standard. This would be sufficient for small one-off projects but not adequate for complex more general ones. In Chapter 6 we develop a *standard protocol* to effect the communications on a more versatile and robust level. We then demonstrate how the protocol provides fault adaptability and tolerance as well as versatility while using it in complex GUI software programs that provide professional looking instrumentation applications on the PC.

Before **Chapter 7** only simple hardware was utilized to experiment with the techniques being learned. This aided in keeping the complexity at a minimum while concentrating on the algorithmic content rather than being mired in the details and intricacies of hardware. In Chapter 7, armed with the sophistication of parallel processing and a versatile communications protocol, we start imparting more complexity to the hardware. We add an ultrasound ranger, two

continuous motion servomotors and two potentiometers. Initially we develop each system on its own and develop simple test programs in firmware and software to establish a base line mechanism for using them. We then integrate them into one overall system. We gradually evolve the firmware developed in Chapter 6 to allow the software to control and interact with the hardware by means of the established protocol with everything functioning in parallel in a smooth and controlled manner. We then go on to impart more abilities to the firmware and also develop another complex GUI software program to utilize the improved firmware and hardware.



In **Chapter 8** we add further hardware and outline a general and methodical strategy for incorporating *any* hardware into the firmware and protocol. Additionally, we learn more sophisticated programming techniques in both Spin and RobotBASIC. We add a compass, an accelerometer, a standard servomotor to be a turret for the ranger, infrared line sensors, ability to save system parameters to an EEPROM, and a better way to use a speaker. We also learn about RB's 3D graphics engine and see how to develop professional looking instrumentation.

In **Chapter 9** we see how to make the hardware system remote from the controlling PC using wireless communication with systems such as the XBee and Bluetooth. Another method for achieving *remote control* is over a Local Area Network with Wi-Fi or across the globe using the Internet. We do this using RB's simple to use yet powerful suite of TCP commands and functions.

In **Chapter 10** we look at the RobotBASIC simulated robot and see how to use RB's inbuilt protocol to effect control over the hardware developed in previous chapters using it as a *robot emulator*. In fact, the protocol developed in chapter 6 and implemented in Chapters 7 and 8 is followed by RB's inbuilt protocol exactly. In this chapter we see how to use the simulator to develop a program to make the simulated robot move in the simulated environment on the screen. But then we see how the very same program with the change of a single number can be made to drive the hardware. All this is possible due to RB's intrinsic protocol that follows the same standards we develop throughout the book. We then go on to use the simulator protocol to develop a simplistic INS (inertial navigation system) to prove how versatile the protocol can be.

In **Chapter 11** we examine some of the limitations of the firmware and we discuss and suggest possible improvements. As an example for how some of these improvements can be implemented we go ahead and create an extended firmware that applies some of those suggestions. We also talk about the soon to be developed RROS (RobotBASIC Robotic Operating System) which is a more sophisticated and general version of the strategies and techniques elucidated in this book.

1.7 Icons Used In This Book

The icon  denotes a point of interest of which you should be aware. The icon  denotes a warning about something that could lead to problems if you are not fully aware of the pertinent facts. The icon ☺ is to prompt you to laugh whenever *we think* we made a joke. You might think otherwise but you *should* laugh regardless; it is good for the mind.

In code listings we will sometimes draw attention to some lines of code in particular from among the other lines in the listing. There are three levels (other than normal code):

Normal code

First level is Bolded text in the listing.
 Called Bold code or lines.

Second level is in white text on a dark gray background.
 Called Highlighted code or lines.

Third level is white text on black background.
 Called Reverse code or lines.



We often refer to RB or the Propeller or Spin by saying something to the effect: “you will send to RB...” or “RB will expect...” meaning a program created in RB running on the PC either within the RB IDE or as a compiled executable (exe) running as a standalone program in the OS. Likewise for the Propeller or Spin when we say “the Propeller will ...” or “Spin wants to...” we mean a program written in Spin (or PASM or both) then compiled and uploaded to the Propeller and is currently running on the Propeller.

1.8 Webpage Reference Links in This Book

We use many devices and refer to many items that can be viewed on the Web. In the book’s text such items are underlined and numbered with a superscripted number. You need to use the superscript number adjacent to the reference and index in the list given in appendix A to find the full URL address of the relevant link. You will also find Appendix A included in a PDF file in the downloadable Zip file that contains all the source code of the book (see Section 1.9). This will be useful since you can click on the link in the PDF file to visit the site instead of having to type the URL by hand in the browser.

1.9 Downloading the Source Code of the Book

You can download from www.RobotBASIC.com a Zip file containing all the code (Spin and RB) organized in folders for each chapter. Additionally there is a file called System_References.PDF, which has in it all the appendices at the end of this book and a selection of some of the figures but in color. There will also be an additional download file containing corrections for any critical errors in the book. There will be no need to download this file since it will remain empty, of course ☺.

Multitasking & Parallel Processing

In Chapter 4 we gained experience in communicating RB and the Propeller and we developed a system for controlling the process by having RB initiate the interaction. The Spin program repeatedly waits for RB to send information. Once the data from the PC is received, the Propeller responds by using the received information to set or interrogate certain hardware and then sends its information. The RB program uses that information and goes on to send the next information. The process repeats ad infinitum. This is an excellent procedure in that we have an orderly system with no swamping and buffer overflow due to disparate and asynchronous transfer rates and processing speeds.

However, we do have a slight glitch with this methodology. The system is fine if the Spin program does not need to do anything else other than wait for RB to send its data. Consider these lines of code from Program_03.Spin:

```
repeat
  outA[23..16] := RB.RX      ' receive the byte and set the LEDS
  RB.TX(inA[7..5])          ' read the buttons and send the states
```

RB.RX is the method used to receive a byte of data from RB. This method will continue trying to receive the byte *forever*. The Spin program will not proceed to the next statement until the byte is received. This, of course is exactly what we want since the next statement sends data and we did not want this to take place until RB is ready to receive it. But, this becomes a problem if we want the Spin program to do other things in the background while it is waiting for the byte from RB to arrive. Unfortunately, with this strategy we cannot do this.

The concept of doing things in the background while waiting for other things to happen is called **Multitasking**. Another related concept is called **Parallel Processing** which is another way to do multiple tasks at the same time or what *appears to be* at the same time. Consider if we wanted Program_03.Spin to also blink an LED at the same time it is waiting to receive the byte from RB. With the current program this is not possible, since the **RB.RX** method will wait for the byte and there is no way to go off to do something else occasionally.

In this chapter we will examine how we can achieve this multitasking action. There are three ways we can achieve multitasking in a program:

- Interrupts
- Polling
- Parallel Processing

5.1 Multitasking Using Interrupts

This option is not available for us using the Propeller and Spin. The Propeller is a parallel processing microcontroller, which, as we will see later, is a much better option than interrupts. So interrupts will not be much use in projects using the Propeller (and a good thing too). Nonetheless, this is an option that is widely used with other microcontrollers and may be something you would like to use in other projects. RobotBASIC is able to perform **Interrupt-Driven** processing, and we will use it to learn briefly about this option using RB programs. Even so, despite the fact that RB makes it easy to learn about interrupt-driven programming, it is a complex issue and is hard to achieve **real** multitasking with it. You are much better off using the Propeller which is an amazing technology that makes it painless to achieve **real** multitasking without having to acquire a PhD in computer science before you do so.

What exactly is an interrupt? Well, as the name implies, it is a signal that occurs while a program is performing a task that forces the program to branch to a particular place in code memory and execute some action, then go back to where it was when it was interrupted to proceed where it left off. The interrupt can be any one of a variety of things. It can be the press of a button, or the arrival of data on a serial port, or the tick of a clock. In microcontrollers, for example, it can be the change of state (e.g. high to low) on an I/O pin, or the overflow of a register, or a transition on an encoder, and the like.

This is actually the way all microprocessors and microcontrollers have been achieving multitasking up until the advent of multi-cored processors not too long ago. We will not delve into this now antiquated, yet ubiquitous, methodology other than to see it in action because RB makes it very simple to do so.

In fact, interrupt operation is not **really** multitasking. It just appears to be so due to the speed of processing achievable with microcontrollers and processors. In reality the processor is only doing one task at a time, since while off attending to the interruption the main task it was executing is halted. Nevertheless, if attending to the interruption takes only a few lines of code, then the main task will appear to have never been halted. But since the action carried out in response to the interruption has been accomplished along with the actions in the main process then both appear to us mere humans as if they were executed simultaneously.

Compare this to the human brain. The human brain is capable of true multitasking in that it can attend to the eyes and the muscles in your arm and hand while also still making your heart beat and receive information from your ears and nose.

5.1.1 RobotBASIC Simulation of a Microcontroller

Before going on to examine how RB interrupts work, let's have a look at an RB program that simulates something you can do with a microcontroller. Let's say we have a microcontroller that blinks an LED at a particular on/off duration:

Blinker_01.Bas

```
i=0 \ duration = 500 \ data clr;white,red
t = timer()
while true
  circlewh 10,10,30,30,red,clr[i]
  delay duration \ i = !i
  //if timer()-t > duration then i = !i \ t=timer()
wend
```

This program works as desired and blinks an LED on for 500 ms and off for 500. Try changing the duration. For now ignore the commented bold line.

In fact the program is faulty:

1. It does not account for the time it takes to execute code. So it is not really at the desired rate. To verify this run Blinker_01_B.Bas (see below). After about a minute or so the **perceived** count of seconds as counted by the number of blinks will start to lag behind the **actual** lapsed time in seconds. The reason is that the perceived

time as counted by the number of blinks does not take into account the time it took to execute the code for the loop and for the counting and so forth. This takes very little time of course and if the duration was larger you may not even see any discrepancy for a long time. The shorter the duration the quicker you will see a lag. Try changing the 200 to 100 (in Blinker_01_B.Bas) and see what happens, also change it to 700 and see what happens. In summary, this method of counting time is faulty but works for slow rates and for a low count.

2. The real problem however, is that while the program, hence the processor, is executing the delay statement it cannot do anything else. The delay duration is just wasted time.

Blinker_01_B.Bas

```
i=0 \ duration = 200 \ data clr:white,red
t = timer() \ n=0
while true
  circlewh 10,10,30,30,red,clr[i]
  delay duration \ i = !i
  n++ \ xstring 10,300,n*duration/1000;(timer()-t)/1000
wend
```

We can solve both problems in Blinker_01.Bas by commenting out the highlighted line and un-commenting the bold line. With this change we are using a timer so that the LED is blinked at the right rate which is not affected by the time it took to execute other lines of code. Also since the program does not sit in a *delay* which does nothing else other than count time we can now do other things inside the loop. For example with this method we can now blink other LEDs at different rates (see Blinker_02.Bas), while with the previous version we would not have been able to do so.

Blinker_02.Bas

```
Main:
  data clr:white,red,white,green,white,yellow
  data rates;200,500,1000
  data states;0,0,0
  data timers;timer(),timer(),timer()
  while true
    for i=0 to 2
      circlewh 10+100*i,10,30,30,clr[i*2+1],clr[states[i]+i*2]
      if timer()-timers[i] > rates[i]
        states[i] = !states[i]
        timers[i]=timer()
      endif
    next
  wend
End
```

5.1.2 Using Interrupts in RobotBASIC

Now let's see how an interrupt may be used. Say there is a pushbutton that when pushed the program Blinker_02.Bas should toggle the color of the LED between blue and red. The bold and highlighted lines in Blinker_03.Bas (see below) are the new lines added to implement the action.

Notice that the bold lines constitute what is called the *interrupt handler*; code that will be executed whenever the interrupt occurs. The handler has to do certain *initialization* tasks, then the *work* it needs to do, and before returning it must do certain *finalization* tasks. In a microcontroller the initialization tasks are to, for instance, disable further interrupts, clear certain flags, save the current program counters, and stack pointers and so forth. The finalization tasks are to update registers and re-enable interrupts reinstate the program counter and pop the stacks and such. With RB, initialization and finalization tasks (but nowhere as complicated) are also necessary as explained in the RobotBASIC help file. Also, it is necessary that an interrupt handler be brief and to only have a small amount of code to be executed. Otherwise the interruption will be too long and the multitasking *illusion* would be lost.

Blinker_03.Bas

```

Main:
  addbutton "Blue",10,60
  onButton bHandler
  data clr$;white,red,white,green,white,yellow
  data rates;200,500,1000
  data states;0,0,0
  data timers;timer(),timer(),timer()
  while true
    for i=0 to 2
      circlewh 10+100*i,10,30,30,clr$(i*2+1),clr$(states[i]+i*2)
      if timer()-timers[i] > rates[i]
        states[i] = !states[i]
        timers[i]=timer()
      endif
    next
  wend
End
bHandler:
  lb = LastButton() //initialization
  if lb == "Blue"
    renamebutton lb,"Red" \ clr$(1) = blue
  elseif lb == "Red"
    renamebutton lb,"Blue" \ clr$(1) = red
  endif
  onButton bHandler //finalization
return

```

It is important to note that the above seems all too easy. This is because RobotBASIC is an excellent language that enables doing such things easily. However, with microprocessors and microcontrollers achieving interrupt handling is not an easy or trivial task. There are numerous considerations and obstacles that can make interrupts fail if not designed and coded correctly. Additionally, in the programs above, RB did scads of housekeeping for you in the background, alleviating the need for you the programmer to have to do all those intricate and confusing details. On the other hand with a microcontroller you have to attend to all these details yourself.

In any case, we will not use this method with the Propeller chip since there is no need for interrupts due to its ability to do *real* multitasking without having to resort to the illusion of one. If you opt to use another microcontroller, then you will need to learn about its interrupt capabilities and how to program for them. If your processor does not support interrupts, then you need to consider using another one. It will not be easy to achieve viable multitasking without an effective interrupt mechanism.

5.2 Multitasking Using Polling

The second method for achieving multitasking is yet another illusion. Polling is the action of occasionally glancing over to see if something else other than the task at hand needs attending to. Think of *polling as a self-imposed interrupt*. Imagine you are working on your computer and are typing something. Your work requires that you answer emails when they arrive. If you have setup your email program to sound a bell whenever an email arrives, you have an interrupt. However if you do not have that ability then you can elect to, either regularly or whenever you feel like it, stop your typing and go over to the email program to check if there is an email.

The polling mechanism can be fine if every time you go to check for an email there happens to be one and moreover, it has not been sitting there for too long. If you frequently go to check and there is no email then you are wasting too much time. If you go there too seldom and emails pile up or you lose certain ones because you did not attend to them on time or they sit there for too long, then again you are not functioning correctly.

Interrupts are in fact the optimal method for this kind of multitasking in that you only abandon the task at hand when emails arrives and do not waste time checking when there are none. Also with interrupts you will never miss an email due to not going there in time to check if one has arrived. Polling is not an efficient mechanism for handling time-critical and frequent interruptions. However, it is an option that you can use and in many situations it is an adequate strategy and is easy to implement.

5.2.1 Polling in RobotBASIC

[Cut Out]

5.2.2 Polling on the Propeller Chip

[Cut Out]

5.2.3 Counting Time in Spin

[Cut Out]

Integer Multiplication Overflow

[Cut Out]

Determining the Clock Frequency

[Cut Out]

5.3 True Multitasking with Parallel Processing

Interrupts and Polling are functional methods and are what has been traditionally used in numerous viable systems; they work well. Polling is simple but not easy to make optimal. Interrupts is the better of the two methods but is hard and complex to program.

The third alternative, Parallel Processing, is in fact the most effective alternative. With parallel processing we can achieve *real* multitasking instead of the *illusion* of it. In the past this option has been expensive and complicated and only available to few systems. With the advent of the most innovative microcontroller, the Propeller Chip, all this has changed. It is now possible to implement parallel processing cheaply, easily and effectively. It is truly an innovation and a revolution on many levels.

What is parallel processing? There was a movie a while back called Multiplicity that starred Michael Keaton. In it Michael was overtaxed by the number of things he had to juggle in his life. As one person he could not be in two places at the same time. He could not pick up the children from school while attending a meeting at work and painting the fence. If only he could have multiple versions of himself. He could then do all those tasks *simultaneously*. You cannot really be picking up the children from school in one part of town and then occasionally jump over to the other side of town to attend to a meeting when it is time for you to speak. So the option of polling or interrupting is not possible in this situation. The only way for Michael to multitask these life obligations is to either, allocate them non-overlapping time slots and allow for travel from one to the other, or he can clone himself and assign each clone the various tasks. Being clones of course they are just as capable as Michael. Michael and his clones can all be doing disparate tasks *independently and simultaneously*.

There is one limitation however. If a task requires that two or more Michaels have to be using the car to travel in different directions then only one Michael can use the car and the other Michaels will have to wait until the car

becomes free. Also, it is not advisable that any other Michaels should have “access” to Michael’s wife other than the original Michael. But Michael’s wife and the other Michaels may have different opinions on that.

Well, enough with Michael and his clones, let’s look at the Propeller. One of the amazing things about the Propeller chip is that it is in fact 8 microcontrollers in one chip (with a surprisingly reasonable price tag). Another remarkable thing about it is the Spin language. This high-level language is easy to learn and easy to use but more importantly it has all the tools you need to create *real parallel processing* with exceptional ease and elegance.

As an example, consider what a non-trivial robot system has to accomplish:

1. Control motors with PWM which require constant updating
2. If wheel encoders are used then constant attention has to be given to the quadrature signals to calculate and keep fresh the current count.
3. Attend to various sensors like Bumpers and Infrared or maybe line sensors.
4. Other systems such as compasses or GPS etc. will also have to be interacted with.
5. If the robot is doing any communications to a central command then this too will have to be performed.

A single processor will be extremely tasked to accomplish the above and even interrupts and polling would not be adequate due to too many interruptions. For instance a wheel quadrature counter can never really be made to function in a system that has to do all the above and at the same time give proper interrupt or polling time slots to be able to not miss quadrature states.

5.3.1 Using Helper Modules

One solution is to use *helper modules*. For instance a [motor controller module](#)²¹ allows a microcontroller to employ *set-it-and-leave-it* approach to controlling a robot’s wheels. This in effect is parallel processing. Since the module allows the controller to specify the direction and speed of the motor and then go off to do whatever it needs to do without having to worry about maintaining the PWM signals required to keep the motors running.

There are numerous helper modules like these that free up the microcontroller and allow it to manage other tasks. In fact with this methodology the microcontroller is nothing more than an overall manager of various other controllers. Most of these modules are in themselves microcontrollers dedicated to doing nothing but the task they are supposed to do (e.g. pulse the motors). If there are no available or affordable modules that can do a task you require and wish to accomplish in parallel then you can easily design your own helper module utilizing a microcontroller to do the task.

Frequently the control of these modules is achieved with a communication between the main controller and the controller onboard the module. Often this control boils down to the main controller sending a byte or two of data (settings and parameters). The module’s controller then uses this data to set up its parameters then continues accordingly doing what it needs to do independently and in parallel with the other actions of the main controller.

This strategy is in reality what makes it possible today to design effective robots that can be controlled with controllers of modest capabilities. Many projects on today’s robots would be quite impossible if it were not for the employment of helper modules such as are available at www.Parallax.com and many other similar web sites.

5.3.2 Using Multiple Microcontrollers

Some disadvantages of the helper-modules strategy of achieving parallel processing and true multitasking is that the modules are not cheap and the variety of interfacing protocols required is bewildering and cumbersome.

Imagine if you had the ability to utilize many microcontrollers with minimal wiring and cheaply and where all of them can communicate with each other via a shared memory rather than through a bit-banging serial protocol (slow). This would be ideal. We won’t be limited to available modules, we won’t incur prohibitive expenses, and we would have no bottleneck in communications.

Well, that is exactly what the Propeller Chip is. It is 8 microcontrollers in one package that share 32KB of RAM. Moreover, the Propeller makes it possible to achieve parallelism with effectiveness that would be hard to achieve otherwise.

5.4 Parallel Processing with the Propeller Chip

We will now convert Program_03.Spin into a parallel processing program. In fact, you have been using the Propeller's parallel processing ability ever since Chapter 3.4. You may not have realized that the FDS and SM serial drivers each use one of the 8 sub-microcontrollers in the Propeller Chip. Whenever we used these drivers we were in effect already utilizing parallel processing. The FDS (or SM) object runs in its own COG (the sub-microcontroller is called COG in the Propeller Chip's parlance).

If you think about what the FDS and the SM modules do you will realize the power of these objects. They, independently of your program, sit in the background listening to the RX Pin (receiver pin) to see if any data is coming and then if data comes in they do the Bit-Banging required to achieve the Asynchronous Serial Communications; they then store that data in a memory area in the shared RAM (receive buffer). Your module can then call methods to extract the data. Also when you use the **Tx()** or **Dec()** or **Str()** methods in the modules you are in fact sending the data to the shared RAM (send buffer) which the object will then send out on the TX pin while also checking if it is allowed to send and so on.

All this is happening in parallel to other tasks you are doing in the main cog. Our programs so far have only utilized one cog (the start up one) and have not utilized any parallel processing save for the FDS and SM objects. So how do we do our own multiprocessing using our own parallel processes? Well, that is exactly what we are going to do from this point onwards. We will progressively build up to a complex and intricate (yet easy to understand and achieve) system that will be a major step towards creating a powerful hardware control system (e.g. a robot) using the Propeller and RobotBASIC as partners.

We will start by gradually converting Program_03.Spin to be a parallel processing system and then add to it some more functionality. All this will serve the purpose of comparing how the program can be made infinitely more versatile and capable than its *linear-flow* counterpart. Armed with the knowledge and experience that the next few sections will provide, we will have the tools required to create the complex system needed to achieve our final overall objective of interfacing and controlling a complex hardware system using the PC.

5.4.1 Modularization in Preparation

[Cut Out]

A Variable's Address in Memory (Pointer)

[Cut Out]

Brief Note About Objects and Methods

[Cut Out]

5.4.2 Initial Multitasking With Polling

[Cut Out]

5.4.3 Achieving Initial Parallelism

[Cut Out]

The Relationship Between Cogs, Methods and Objects

[Cut Out]

Cogs and Stack Space

[Cut Out]

5.4.4 Systematic Debugging of Complex Programs

[Cut Out]

5.4.5 Sources For Obtaining Help With Difficult Problems

[Cut Out]

5.4.6 Parallel Processing Contention for Resources

[Cut Out]

5.5 Objects, Semaphores and Flags

In Section 5.4 we achieved a major step forward towards our objective. We managed to create parallelism with three processes (5 really with the FDS and SM) running independently and truly simultaneously:

1. The **Main** cog doing the byte receiving and sending as well as blinking an LED on P23.
2. The **SetLEDs()** cog setting the LEDs according to the byte received by **Main**.
3. The **ReadPins()** cog reading the pushbuttons and setting the byte to be sent by **Main**.
4. The **FDS** cog doing serial data bit banging to and from RB.
5. The **SM** cog doing serial data bit banging to the PST.

In fact though, we really do not get a full appreciation for the parallelism since cog 2 and 3 are not really doing much that truly requires the power of parallelism. Of course we are still in the process of advancing towards a useful and powerful system and we have to proceed gradually. Nevertheless, we did get a feel for this parallelism when we allowed cog 2 and 3 to output to the PST, albeit in an intermingled manner.

In this section we are going to press forward, adding more complexity. We are still in the learning process, so don't worry we will proceed in small surmountable steps. We will:

- Divide the project into objects, adding some more parallel actions.
- Solve the problem of jumbled output to the PST by using Semaphores
- Manage the Parallelism further with Flags

5.5.1 Creating Objects

[Cut Out]

5.5.2 Utilizing Semaphores

Recall how in Program_05_E.Spin (Section 5.4.4) we used output to the PST but when we let both cogs stream out to the PST we had a problem with the bytes from each being intermingled with the other and the output was a useless jumble of data from each shuffled up into an unreadable mess.

In this improvement of our program we are going to use PST debugging and we will let both cogs as well as the **Main** cog output messages to the PST while also working the LEDs and pushbuttons and receiving and sending data to RB as

well as keeping the LEDs blinking. With all this action we will make decisive use of parallelism and multitasking. See Figure 5.4 for a conceptual schematic the system.

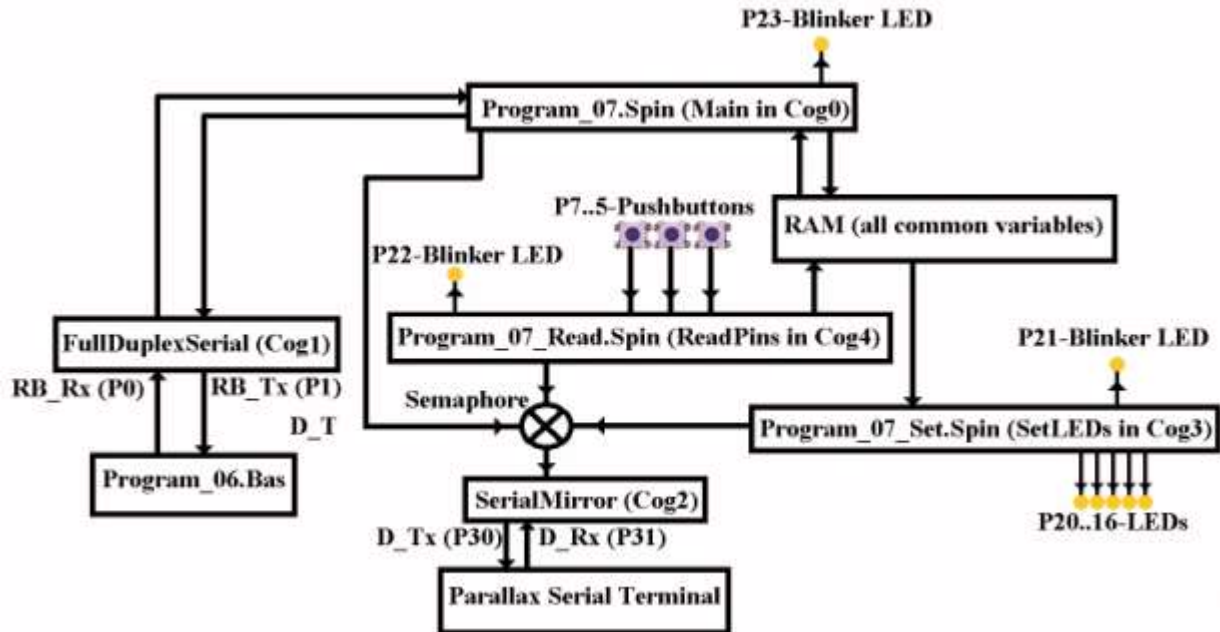


Figure 5.4: A schematic of the various objects and cogs in the new system showing how they interact. Cog numbers are just for reference, they are not necessarily the actual order.

Notice how **Main**, **ReadPins()** and **SetLEDs()** are all using the SerialMirror object. Because every object must instantiate its own version of any other objects it uses, we must instantiate the SM object in all the objects. However, only one of them must call the **Start()** method of the SM object. This should be done by the top-level-object (**Main**).

This is why we used SM for debugging with the PST instead of FDS. It is because SM can handle multiple objects using it with only *one* cog, while the FDS has to have multiple cogs. So the FDS is wasteful of cogs. However, notice that SM will only do this for the same RX/TX pins for all the objects using it. Since we want different pins for sending to RB we use the FDS as the object for that. We cannot use SM with another call to its **Start()** method with different pins.

Also notice that the FDS and SM objects do not share RAM with the other objects. They have their own RAM area that does not need to be accessed by outside objects because the objects provide getters and setters (for example the **Tx()**, **Rx** and **Dec()** method). Strictly speaking all the objects share the same RAM but in *logical* terms the RAM for the FDS and for SM are not accessible or visible to the other objects.

Study Figure 5.4 well. It is a good way of understanding what we have achieved so far and for developing a feel for the way parallel processing is going on and appreciating how the shared RAM is a great way for connecting the **Main** cog with the **SetLEDs** and **ReadPins** cogs. The next thing we need to do in Program_07.Spin is to create a mechanism where only one of the three independent and parallel cogs can send through the SM object at a time.

What is a Semaphore?

The word *Semaphore* means *an apparatus for signaling, such as the arrangement of lights, flags, and mechanical arms on railroads*.

Imagine the three cogs are like trains trying to use a *single* crossing. You obviously need a signal to tell the trains to wait before they cross while another train is using the crossing. No train can attempt to move through the crossing until it has a green light to do so. Once it has acquired the green light it should then cross and once clear of the crossing it should release the green light. Other trains will then be able to attempt to turn the green light on. Only one train can have the green light at a time.

The above mechanism of Semaphores is exactly what we will use to stop data jumbling when cogs are trying to send data simultaneously to the PST. Only the top-level object should create the semaphore (**LockNew**) and then it must pass the address of the signal to the other cogs so that they can try to lock it (**LockSet**). A cog will only send data through the PST if it has managed to acquire the lock. Once it finishes sending it should then release the lock (**LockClr**). This way, the three cogs will be guaranteed a turn to send data through this one *shared resource*.



The three Spin statements used to utilize semaphores are:

LockNew(): to create the required semaphore and store its ID in a variable. *This is only performed once for each semaphore (maximum of 8)* by the top-level-object and then it makes the *address* of the ID variable available to all other cogs that need to manipulate the semaphore.

LockSet(): to try to capture the semaphore. If the semaphore is in use by another cog then the function will return true, if it is not then it will return false, but that also means that it is now captured by the calling cog. There is no specific need to take any other action. If **LockSet()** returns false then it is now captured by the calling cog. If it returns true then it is not captured by the calling cog. *Note the logic.*

LockClear(): to release the semaphore when it is no longer needed by the cog. A cog that acquires a semaphore *must also release it*. If it does not release it then other cogs that may require it will never be able to do their action. Even the cog that has the semaphore may not be able to do any more work again if it does not release the semaphore before it tries to lock it again (e.g. in a loop).

Using a Semaphore

[Cut Out]

5.5.3 Tighter Control With Flags

In the previous section we resolved the problem of intermingling bytes sent from the cogs simultaneously by using a very clever technique that the Propeller + Spin make extremely easy to implement. Another technique related to semaphores is *flagging*. Using flagging, a process signals another to go ahead and do something that should only be performed when flagged and once it's finished doing so it should clear the flag which also serves the purpose of telling the signaling process that the task is finished.



A flag serves as a two way signal between two processes where one raises the flag and the other lowers it.

With this mechanism even though the two processes might be running in parallel and at different speeds, the controller process can signal the other to tell it that some data is ready or that it is ok to do something. The other process can check for the flag state (polling) and if it is raised the process does what it is supposed to do and then lowers the flag. This indicates to the master process that the work is done. Sometimes, depending on the task, the process may lower the flag before it finishes doing the work if the action permits that kind of synchronization. This way the flag raiser can go on doing something else while the flagged process can be working at its pace processing the flagged action.

In Program_08.Spin (and its subordinate objects) we will use flags between the **Main** cog and the two other cogs (2 flags). The flags are basically to let the other cogs know that **Main** has output data to the PST and so the other cogs can output their data too. In this manner **Main** can get in a word edge wise instead of being outspoken by the much more verbose other cogs. When you run Program_08.Spin and observe the output on the PST window you will see that now the **Main** message is visible a lot more often and that the output from all the cogs is taking place in a lot more orderly manner. Also observe that the RB interaction is quite timely too and that the three Blinker LEDs are also blinking on time.

[Cut Out]

Using Semaphores and Flags we managed to tame the chaos caused by unbridled parallel processing. We utilized the power and convenience of multiple microcontrollers doing their work in *parallel but yet in orchestrated unison*.

Semaphores are also a great mechanism for coordinating memory access. Imagine if two processes share a buffer in RAM. One writes to it and the other reads from it. Imagine if the buffer is a few bytes long. If a process reads the buffer while another is still writing to it then it is possible that the reader would be reading jumbled data of old and new bytes. Semaphores should be used to synchronize this process.

Notice in the program how we used a byte variable in the **Dat** section of the top-level-object. The individual flags are the Least Significant (first from the right) two bits of the byte variable.

Rather than passing yet one more parameter to the other cogs we made use of the fact that the **Flags** variable in the **Dat** section comes directly after the **Semaphore** variable. And since we are already passing the *address* of the **Semaphore** variable to the other cogs, we obtain the **Flags** variable by reading the byte after the **Semaphores** variable. Thus the use of **Byte[Semaphore][1]** since **Byte[Semaphore][0]** would be the byte which is the **Semaphore** variable itself then [1] is the byte right after and therefore is the **Flags** variable



When sharing RAM variables between cogs, we need to pass the addresses of these variables from the object that contains them to the cogs in the other objects. This can be achieved by passing an address for *each* variable, but this is wasteful. A better mechanism is to ensure that all the necessary variables are arranged in a *contiguous* block of RAM we will call a *buffer*. Then the address of the top of the buffer is passed to the other cogs. This buffer can then be used as an *array* of data. The cog using the buffer can *index* into the buffer as it needs to obtain the **Longs**, **Word** or **Bytes** it needs. Of course the arrangement has to be known so that the correct indexing can be used. However, not all the variables in the buffer have to be of the same type. But care has to be taken to ensure that they are *aligned properly*.

Notice the use of the **FlagMask** constants. These are used to check if the flag is set in the respective cogs by masking out the appropriate bit from the byte that contains all the flags. Also the inverted mask is used to reset the flag.

The flags are not set until **Main** has actually sent some data to the PST. When the flags are set, the other cogs contend between each other for the semaphore to write to the PST. Also the cog does not clear its flag until after it has already written out to the PST. This assures that the cog will continue to contend for the Semaphore until it has written data out to the PST. **Main** also needs to contend for the Semaphore because the two cogs might still be trying to write out and we do not want it to clash.

Semaphore ensures that no two cogs can write out at the same time. The flags are a way for the less frequent writer (**Main**) to not be swamped out and rarely be able to get hold of the Semaphore.

5.6 Parallel-Parallel Processing

The Propeller Chip has a mechanism to create even more parallelism. It is like parallel processing on top of parallel processing. This mechanism is called *counters*. Every cog has two of them. Each counter can do all sorts of actions that once configured can be left alone and they will continue to do their action while the cog is free to do other actions. So this is like two parallel processes going on within the cog and the cog is doing its work in parallel to the others; *Parallel-Parallel processing*.

There are numerous things these counters can do. As a useful example we are going to modify the top-level-object (Program_08.Spin) instead of blinking the LED on P23 to slowly vary the brightness of the LED from off to full brightness and then gradually dimmer until off again. This will continue as long as the cog is active.

We modified Program_08.Spin to make Program_09.Spin. However the sub-objects remain the same. This illustrates the use of making objects since the Program_08_Set.Spin and Program_08_Read.Spin will be used again with Program_09.Spin.

The accompanying RB program remains to be Program_06.Bas since the new system is the same as far as the RB program is concerned. The only difference is that instead of blinking the P23 LED on/off **Main** will use a counter in the *duty mode* to control the level of voltage on P23. This causes the LED to vary in brightness. We will set it so that the LED will repeatedly increase in brightness from off to fully bright in 255 steps over 1 second and then dim back to off in 255 steps over 1 second.



The principle is something similar to Pulse Width Modulation (PWM). It is similar in effect but not the same in action. In action it is more aptly called Pulse Frequency Modulation (PFM). Rather than vary the duty of a constant frequency signal we vary the frequency of a constant duty signal.

[Cut Out]

5.7 Stack Overflow

[Cut Out]

5.8 A Musical Keyboard

As you saw in Section 5.6 the counters in the Propeller Chip can be quite interesting. One of the modes for using the **ctrA** or **ctrB** counters in a cog is to generate a signal of a particular frequency. In this section we will make use of this ability to make musical sounds on a [Piezoelectric Speaker \(Part#900-00001\)](#)²⁷.

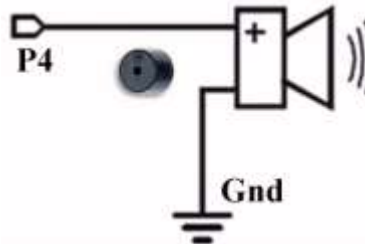


Figure 5.5: Piezoelectric Speaker Connection Schematic.

We will use RB and the Propeller Chip to allow a user to play music on a Piano Keyboard by clicking with the mouse on a graphical representation of the keyboard. Furthermore, there will be a button that when pushed will start playing a tune. The new program Piano.Spin is a modification of Program_09.Spin. We will still use Program_08_Read.Spin but we will not use Program_08_Set.Spin. Instead we will make a new object called Piano_Set.Spin.

Piano.Spin is a major modification of the original object in that we now will receive 4 bytes not one as before. These bytes will be used to create a Long integer (32 bits) using the Little-Endian arrangement since that is what the Propeller uses to store its 32-bit integers. Other actions are as before in changing the brightness of the LED on P23 and everything else as in Section 5.6.

5.8.1 A Different Way of Sharing RAM

Other changes from the old program are that we no longer need the **ReceivedByte** buffer area and also when we **Set.Start()** we no longer need to pass along the buffer address either. This is because we are using a new style of passing the value to the other object and thus to the other cog (see discussion about objects in Section 5.4.1). This is achieved with the **Public PlayNote()** method in the **Piano_Set** object. We use the method to pass the value of the frequency to the *object*. But this does not pass it to the *cog*; to make it available to the cog, the method has to store the value into a variable in the RAM to which the cog has access (**Frequency**). This variable is accessible to the cog since it is in the same object as the cog's method. But the variable is not accessible to other objects.

This new method of passing parameters to other objects and on to the cog in the object is effective because it achieves tighter encapsulation. Nevertheless, it is a bit wasteful in that there are function calls to be made and stacks to be pushed and popped and so forth. This can be wasteful in both stack size requirement and in speed. However, encapsulation and hiding of variables may be a desirable property in certain situations.

In this case we will use this method just as an illustration of this option. It would have been more efficient to have given **Main** access to the shared variable and let it set the variable and then the sub-object would see the change. Nonetheless, in certain situations using this methodology might be desirable for other than encapsulation. Sometimes calling methods provides *sequencing control* where some actions are only performed when the method is called as opposed to when the variable changes value which has to be monitored (polled) or by using flagging as we have been doing.



The Propeller is a 32-bit processor and a Long in its memory is a 32-bit number (4 bytes). You can also access the 4 bytes as individual bytes. If you have a variable **f** declared as a Long you can access its individual 4 bytes using **f.Byte[n]** where **n** ranges from 0 to 3.

The Propeller uses the Little-Endian format to store integers. So if we have an integer in memory that is \$A3_12_BC_45 then in RAM it is actually stored as 4 bytes where the first byte (byte 0) is \$45 and the next byte (byte 1) is \$BC and so forth. So when we look at **f.Byte[0]** we will see \$45 and so on.

Piano.Spin will wait for 4 bytes to arrive from the **RB** program one by one. When the first one comes in it will be set in the **f.Byte[0]**. The next received will be saved in the next byte (1) and so on. When all 4 arrive the Long value would then be fully formed as a 32-bit integer and it will be passed to the **Piano_Set** object using the **PlayNote()** method which uses it to set the **Frequency** variable in its RAM space where the cog has access to it.

We will still use the **Program_08_Read** object just as before to read the status of the pushbuttons. **Main** will send that value back to **RB** to serve as a signal to proceed with sending the next 4 bytes and also the **RB** program may use the pushbuttons' status if needed like before.

The new **Piano_Set** object will not set the LEDs on P20..P16 any longer. Instead it will use the **Frequency** value to set the **frqB** register of a counter as will be explained shortly.

5.8.2 Creating Frequencies (Numerically Controlled Oscillator)

The new object will setup the **ctrB** counter to be in the NCO (Numerically Controlled Oscillator) mode. In this mode the counter will make a pin go high as long as the 32nd bit (bit 31) on the **phsB** register is high and low when it is low. And since the counter will add the value of **frqB** to **phsB** every clock tick then we need to set the **frqB** value so that bit 31 of the **phsB** register will go high and low to generate the right frequency. The formula is:

$$\text{frqB} = \text{Required Frequency} * 2^{32} / \text{clock-frequency.}$$

Since the clock-frequency we are using is 80_000_000 (80 MHz) then $2^{32}/80_000_000 = 53.678$

To generate a frequency of say 1708 we need to set **freqB** to the value **round(1708*(2.0^32)/80e6)**. When we send this value to the Propeller, it assigns it to **freqB** and also sets P4 to be an output pin. If P4 is connected to a Piezoelectric Speaker the right tone would be generated.

The object will also blink an LED on P21 (as before). When **Frequency** is other than -1 it will be assigned to **freqB** and P4 will be set as an output pin to allow the oscillations to start. It will also set **Frequency** to -1 to prevent replaying the same note for ever. When the **PlayNote()** method is invoked it will also start a stopwatch timer. This timer is used to stop the note playing if no new note (frequency value) is received before a certain timeout (5 secs) by making P4 an input pin which will disable the oscillation, effectively stopping the signal. If a new value is received before the timeout then of course it will change the **freqB** value which starts a new frequency and reset the timer.

The new firmware is much like the old system but now RB will have to send the 32-bit (long Integer) frequency value as 4 bytes with the LSByte first and the firmware will receive those bytes and then send a byte back to RB (the pushbuttons status as before).

5.8.3 Testing the Speaker Firmware

The RB program needs to calculate the value to be sent to the Propeller using:

```
N = (2.0^32)/80e6
FreqValue = round(ActualFrequency*N)
```

FreqValue will then be sent using:

```
SerialOut BuffWrite("", 0, FreqValue)
```

The function **BuffWrite()** is used to create a byte buffer with the 4-byte (32 bits) integer in it. RB also uses the Little-Endian format and so byte 0 is also the LSByte. When **SerialOut** sends the byte buffer all 4 bytes would be sent to the send buffer and then RB would take care of sending these 4 bytes to the Propeller one at a time.

Examine the program **Speaker_Tester.Bas** to see how this is implemented in the **PlayNote()** subroutine. Also notice how the main program generates random frequencies. The program will not do much else for the sake of simplicity. Notice all the bold code lines to see how the discussion above is implemented in code.



The Propeller is a 3.3V chip, so the Speaker will not be very loud. You may have to be close to it to hear the sounds well. We will see how to increase the volume of the sound in Chapter 8.



If you are running on an Me or XP machine you will be able to hear the sounds generated on the PC speaker if you set the variable **Port** to 0 and also uncomment the highlighted line. Do not do this if you have a Vista machine because it may give you an error.



Another way we can send the 4 bytes of the Long Integer value is to use the **GetByte()** function in a loop:

```
N = (2.0^32)/80e6 \ FreqValue = Round(ActualFrequency*N)
For I=0 to 3
    SerialOut GetByte(FreqValue,I)
Next
```

Speaker_Tester.Bas

```
//Speaker_Tester.Bas
//works with Piano.Spin
Port = 8 //change this as per your system
Main:
    setcommport Port,br115200
```



```

while true
    call PlayNote(random(3000)+500,600)
wend
End
//-----
sub PlayNote(F,D,&B)
    xstring 1,1,"Note = ",F;"Duration = ",D,spaces(10) //display data
    B = 0 \ c = 1
    if Port == 0 //if not serial
        //sound F,D //play on speaker...only XP machines
    else //otherwise
        N = round(F*2.0^32/80e6) //convert to frqA values
        SerialOut BuffWrite("",0,N) //send the 4 bytes of the Long LSByte first
        /****this is another way to do the same thing but is commented out
        for i = 0 to 3 //send the 4 bytes of the Long
            serialout getbyte(N,i) //LSByte first
        next
        *****/
        delay D //delay
        serbytesin 1,m,c //get the confirmation byte (buttons state)
        if c then B = getstrbyte(m,1) //get the value
    endif
Return (c==1) //return true or false if there was a byte received
//=====

```

[Cut Out]

5.8.4 A Piano Keyboard Player

Now that we tested the new firmware we will write an interesting program utilizing the new firmware to allow a user to interact with a graphical Piano Keyboard on the PC screen. The user can click on the key and will hear the notes playing on the Piezoelectric Speaker on the Propeller. Additionally, there will be a button on the screen that will allow the user to hear a tune playing repeatedly until the button is pushed again. The tune is “Jingle Bells”. See Figure 5.6.

The program is a complex one but it basically uses the **PlayNote()** subroutine we saw in `Speaker_Tester.Bas` to play the note that the user clicks the mouse over. The program will have to determine the following:

1. Which key the user is pushing. This is determined by:
 - a. The position of the mouse when clicked
 - b. The color of the key under the mouse
2. What the frequency of that key is. This is calculated from:
 - c. The key’s scale
 - d. The Key’s position within the scale (Note)

Remember that there are seven normal notes and five sharps in each scale. Also there are 5 scales as drawn on the screen with the middle scale being the middle C-scale. Once the key’s scale and note are determined, the actual frequency value is determined from an array of frequencies.

All the code in the program is to draw the keyboard and to determine the key being pushed and its frequency. Once the frequency is determined, it is played by sending it to the Propeller.

Another action the program provides is the ability to play a tune. This is *similar* to the RTTTL tunes on cell phones. The tune is defined as a series of notes and durations with also the ability to define the scale and pauses. The tempo and the code of the duration determine the actual time in milliseconds the note will play. Again, the note’s frequency is determined from the scale and the note’s position in the scale. These two as before determine the frequency value from the array of frequencies.

Examine the listing below to see how all the above logic is implemented.

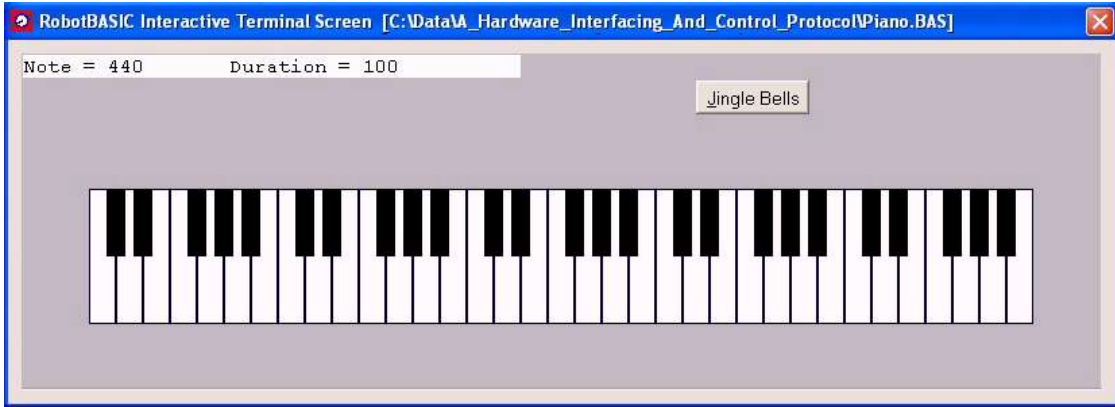


Figure 5.6: Screenshot of Piano.Bas in action



In Piano.Bas and many of the programs to come we use RB's Call/Sub subroutines with variable parameters, by reference parameters, local variable scoping and global variables with the use of the _ operator. See [RobotBASIC Subroutines.PDF](#)⁷² for a tutorial on this powerful feature of RB.

Piano.Bas

```
//Piano.Bas
//Works with Piano.Spin
Port = 0 //set this as per your system
Main:
  GoSub Initialization
  while true
    call CheckMouse()
    while PlayTune
      call Play_Tune(Tempo)
    wend
  wend
end
//=====
Initialization:
  GoSub SetUpNotes
  GoSub SetUp_Jingles
  clearscr gray
  WOffset = 50 \ WW=20
  for i=0 to 7*5-1 //draw the normal keys
    rectanglehw WOffset+WW*i,100,WW,100,rgb(0,0,50)
  next
  BW = 14 \ BOffset = 63
  for i=0 to 7*5-1 //draw the sharp keys
    rectanglehw BOffset+WW*i,100,BW,50,black,black
    if i#7 == 1 || i#7 == 5 then i++ //some sharps not allowed
  next
  data NoteMap; 0,2,4,5,7,9,11
  data SharpMap; 1,3,0,6,8,10
  setcommport Port,br115200
  clearserbuffer
  PlayTune = false
  AddButton "&Jingle Bells",500,20
```

```

    onButton bHandler
Return
//=====
sub bHandler() //button interrupt handler
    lb = LastButton()
    if left(lb,3) == "&Ji"
        RenameButton lb,"&Stop"
        _PlayTune = true
    else
        RenameButton lb,"&Jingle Bells"
        _PlayTune = false
    endif
    onbutton bHandler
return
//=====
SetUpNotes: //frequencies array
    data Notes;32.703,34.648,36.708,38.891,41.203,43.654
    data Notes;46.249,48.999,51.913,55.0,58.27,61.735
    data Notes;65.406,69.296,73.416,77.782,82.407,87.307
    data Notes;92.499,97.999,103.83,110.0,116.54,123.47
    data Notes;130.81,138.59,146.83,155.56,164.1,174.61
    data Notes;185.0,196.0,207.65,220.0,233.08,246.94
    data Notes;261.63,277.18,293.66,311.13,329.63,349.23 'middle C
    data Notes;369.99,391.99,415.31,440.0,466.16,493.88
    data Notes;523.25,554.37,587.33,622.25,659.26,698.46
    data Notes;739.99,783.99,830.61,880.0,932.33,987.77
    data Notes;1046.5,1108.7,1174.7,1244.5,1318.5,1396.9
    data Notes;1480.0,1568.0,1661.2,1760.0,1864.7,1975.5
    data Notes;2093.0,2217.5,2349.3,2489.0,2637.0,2793.8
    data Notes;2960.0,3136.0,3322.4,3520.0,3729.3,3951.1
    S=-2 \ P=-1 \ C=0 \ CS=1 \ D=2 \ DS=3 \ E=4
    F=5 \ FS=6 \ G=7 \ GS=8 \ A=9 \ AS=10 \ B=11
Return
//=====
SetUp_Jingles: //RTTTL codes for the tune
    Tempo = 1500
    data Song;S,5,E,8,E,8,P,32,E,4,P,32,E,8,E,8,P,32,E,4,P,32
    data Song;E,8,G,8,P,32,C,4,D,16,P,32,E,2,P,16
    data Song;F,8,F,8,P,32,F,8,F,16,P,32,F,8,E,8,P,32
    data Song;E,8,E,16,P,32,G,8,G,8,F,8,D,8,P,32,C,2
Return
//=====
sub Play_Tune(Tempo) //play all the notes in the tune list
    Scale = 4
    FOR i = 0 TO MaxDim(Song,1)-1 step 2
        if Song[i] = _P //if a pause
            Frequency = 0
            Duration = Tempo/Song[i+1]
        elseif Song[i] = _S //if scale change
            Scale = Song[i+1]
            continue
        else
            Frequency = Notes[Song[i]+12*Scale] //determine freq from scale & note
            Duration = Tempo/Song[i+1] //determine duration from tempo
        endif
        call PlayNote(Frequency,Duration)
    
```

```

    if !_PlayTune || !PlayNote__Result then _PlayTune = false \ break
next
Return
//=====
sub PlayNote(F,D,&B)
  xysting 1,1,"Note = ",F;"Duration = ",D,spaces(10) //display data
  B = 0 \ c = 1
  if _Port == 0 //if not serial
    //sound F,D //play on PC speaker...only XP machines
  else
    N = round(F*2.0^32/80e6) //convert to frqA values
    SerialOut BuffWrite("",0,N) //send the 4 bytes of the Long LSByte first
    delay D //delay
    serbytesin 1,m,c //get the confirmation byte (buttons state)
    if c then B = getstrbyte(m,1) //get the value of the buttons' states
  endif
Return (c==1) //return true or false if there was a byte received
//=====
sub CheckMouse() //determine which key is pushed
  readmouse x,y,b //read mouse
  if !b then call PlayNote(0,1) \ return //if no click then no sound
  c = pixelclr(x,y) //get the color under the mouse
  if c == white
    x = (x-_WOffset)/_WW //convert x to key number
    Scale = 1+x/7 \ Note = NoteMap[x#7] //convert to note number & scale
  elseif c == black
    x = (x-_BOffset)/_WW //convert to key number
    Scale = 1+x/7 \ Note = SharpMap[x#7] //convert to note number & scale
  endif
  if c == white || c == Black //if there is a note
    Frequency = Notes[Note+12*Scale] //convert to frequency
    call PlayNote(Frequency,100) //play it
  endif
Return

```

An Exercise

In Piano.Bas above, when you press the button on the screen to start playing the tune you can stop the tune by pushing the button again. Is it possible to accomplish the same action with the hardware pushbuttons? The firmware returns the status of the pushbuttons on the hardware; therefore it is possible to have the tune start/stop by pushing a button on the hardware – say the one on P5. If the tune is already playing, pushing the pushbutton on P5 should stop it, and if the tune is not already playing then it should be started; P5 will behave like the button on the RB screen. The tune can be started or stopped (toggled) by pushing either the RB screen button or the hardware P5 pushbutton.

Can you implement the required software changes in the program Piano.Bas to apply the above improvements? What is needed is to use the byte returned by the Propeller with the status of the pushbuttons to decide whether to play the tune if it is not already playing or to stop it if it is already playing. Try to do so without reading the hints. The solution is given below.



The above interaction illustrates how the software can be made to act as a surrogate for the hardware but also augment and enhance it (e.g. the piano GUI keyboard). Also imagine if you had a library of tunes and you wanted to allow the user to select one from a list of tunes. In the software it would be easy to do this (see **AddListBox** command in RB). However, in the hardware you would need additional hardware.

Hint: The buttons are active low

Hint: Remember that the RB pushbutton needs to be renamed to reflect the state. If the tune is playing it should say **Stop** and if the tune is not playing it should say **Jingle Bells**. This way the button will continue to work correctly in conjunction with the hardware button.

Hint: Look at the **Main** section in the RB program and see what determines if a tune is to be played or not.

Hint: A few lines of code are needed in the subroutine **PlayNote()** just above the **endif** statement. These lines should check to see if the hardware button is pushed. But also to check what is the current condition of play. If playing then stop, if not then start.

Solution

In the **PlayNote()** subroutine just after the line:

```
if c then B = getstrbyte(m,1) //get the value of the buttons' states
```

Add the following lines:

```
if (~B) & 0%001 //the buttons are active low and P5 is the LSBit
  if _PlayTune
    RenameButton "&Stop","&Jingle Bells"
  else
    RenameButton "&Jingle Bells","&Stop"
  endif
  _PlayTune = ! _PlayTune
  delay 200 //delay to eliminate button bounce
endif
```

5.8.5 Some Thoughts and Considerations

Sections 5.8.3 and 5.8.4 highlight something very interesting. Consider what we did. In Section 5.8.3 we used a **hardware** setup with a **firmware** and a **protocol** to play some random notes by using simple **software**. The protocol allowed the software to send values which the firmware knew what to do with and that caused the hardware to generate an audible frequency on the speaker.

In Section 5.8.4 we used the very same hardware, firmware and protocol; **nothing changed**. The hardware does not know anything about how to play a tune. It does not have any user interface (only the pushbuttons). It has no means of organizing tempo or determining if the user wanted to play Jingle Bells or not. There was nothing in the hardware that even told it what to do if the user did push the pushbuttons. **The hardware and firmware knew nothing except how to play a note of a particular specified frequency and read a pushbutton status**. Yet, when we changed the software we had a sophisticated overall system.

The PC did not have the means of generating the sound. It had no speaker and no frequency generator. However, when combined with the hardware it had the means to do so. The hardware had no means of effecting a user interface, yet when combined with the PC it had the means to do so.

Later in the exercise we saw how powerful the cooperation between the software and firmware can be. The firmware had absolutely nothing to relate the press of a pushbutton to any action other than to send it on the serial port. It had no logic to make it into a toggle switch for playing a musical tune. As far as the hardware is concerned there is absolutely no relationship between the speaker and the pushbuttons. But, with the addition of a few lines of software we made it possible for the hardware to become an intelligent device.

Think about this for a moment. If you are standing away from the PC and do not see the screen while the RB program is running, you can push the button on P5 and hear the song play. You push it again and it stops. The hardware is now doing something intelligent. It knows when you push the button whether to play a song or not. It knows whether a song is already playing to stop it and vice versa. How can it even know that? There is nothing in the firmware that tells it that. **It has decision abilities that were not even programmed at all in the firmware**. The firmware is doing something

it was never specifically programmed to do. This is a powerful concept. Three different version of software, using the very same hardware, accomplished different actions. The behavior of the very same hardware changed drastically *only by changing the software*.

This is what we are trying to accomplish here. We want to be able to make hardware do different things but without having to reprogram the firmware on it. Without having to keep changing the low-level Operating System of the hardware we can add new software to make it do new actions.



The hardware and firmware know *how* to do a low-level action. The software knows the *when* and *why* the hardware needs to do something without having to be hindered with the details of how. It is like a company. The boss knows why she needs a certain product and when it has to be created. The boss has the bigger picture in view; she knows what she wants done to be able to make the company successful. However, she does not know how to create the product. She has no idea how to use a lathe or how to weld. So she employs people who do. She knows the why and when. The employees know the how. Alone the boss could not accomplish her vision. Alone the employees do not have the drive. Together they form a successful endeavor beneficial to all (at least most of the time).

5.9 Parallel Programming Can Create Puzzling Errors

Programming for parallel processing has many pitfalls that can cause quite a lot of puzzling bugs. Often the cause is a lack of appreciation for what can occur when parallel processes are interacting. Other times the cause can be misunderstanding what the mechanisms provided by a system such as the Propeller Chip can do.

Coming from the traditional linear flow programs it is often hard to switch over to a mode of thinking that allows for the nuances of parallel processing. The Propeller Chip enables the creation of parallelism with ease, nonetheless, there are things the Propeller cannot do for you. You still have to consider carefully all the intricacies of interaction that are required to assure proper sequencing and orchestration of the various independent processes.

You have to remember that despite the programs in each cog being linear programs, the overall system is not. Each cog can be in a totally undeterminable state in as far as another cog is concerned. We have already seen one type of this problem where parallelism can be puzzling. When we finally had the program working in Section 5.4 we had the problem of the PST output being a jumble of letters where the messages from all the cogs were *shuffled* together. The fact that the cogs were sending their messages simultaneously through the one serial port was the problem and we devised a mechanism for orchestrating them using semaphores in Section 5.5. We also had to use flags to further control the output of the cogs to stop one swamping and obscuring the output of the others.

5.9.1 An Example of a Parallel Processing Trap

[Cut Out]

5.9.2 An Example of a Propeller Specific Trap

[Cut Out]

5.10 Logistical Planning for Parallelism With the Propeller

[Cut Out]

5.11 Summary

In this chapter we:

- ❑ Studied Multitasking using interrupts in RobotBASIC.
- ❑ Created Parallelism using Polling in RobotBASIC and Spin.
- ❑ Learned about timing in Spin.
- ❑ Learned about variable addresses (pointers) in Spin.
- ❑ Learned about some objet-based programming in Spin.
- ❑ Learned how to start cogs working in Parallel.
- ❑ Examined the relationship between cogs, methods and objects.
- ❑ Learned how to debug puzzling problems.
- ❑ Learned how to use Semaphores and Flags to avoid contentions.
- ❑ Learned about Stack Space.
- ❑ Learned about using counters in the Duty and NCO modes.
- ❑ Utilized a counter to generate sound.
- ❑ Seen how the PC and Propeller can create synergetic relationship through the protocol, firmware and software.
- ❑ Learned about some possible traps in using the Propeller and parallelism.
- ❑ Considered some aspects of the logistics of planning for parallel programs.

More Advanced Hardware

In Chapter 7 we added some interesting hardware. In this chapter we will add more hardware that despite being slightly more complicated than what we had so far, is nonetheless easy to integrate into our system due to the versatility and robustness of the protocol. The actual hardware used is immaterial and your requirements may dictate a different set of devices. What is important, are the *principles* involved in incorporating the hardware within the system. Section 8.2 will expound a *procedural strategy* that makes adding any hardware to the *protocol* a simple endeavor. The details will differ from one device to another, but the overarching principle for how the *firmware* makes the devices available to *software* by means of a protocol is what interests us here. We will add:

- A compass
- Ability to control the motors individually
- A turret for the ultrasound ranger
- A mechanism to save the system's parameters to the EEPROM and to reset them to factory settings
- An accelerometer
- Three Infrared line sensors
- A speaker



To keep track of all the modifications and hardware, see Appendix B for the complete details of the final setup of the system as it will be once we complete adding all the hardware in this chapter. See Figures B.1, B.2 and Table B.1. Also see Figure 8.15.



See Section 8.2 for a *procedural strategy* for adding hardware to a system that implements our protocol.

8.1 Adding a Compass

[Cut Out]

8.1.1 Using the Compass

[Cut Out]

8.1.2 Inter-Cog Communications and Complex Object Interaction

[Cut Out]

8.1.3 Using the Compass Calibration

The HMC6352 compass module has a very good resolution. It is accurate to 1°, which is as accurate as any robot may need. However, the readings can be affected by surrounding magnetic fields. One way to minimize this error is to calibrate the module in the environment it is to be used in.

The compass has a very easy and effective inbuilt calibration. All you have to do is invoke the calibration process, which lasts 20 seconds. It is important to keep the module level and to turn it slowly through two complete turns. We have provided a method in our protocol to perform this calibration. There are two ways you can perform this; a *manual calibration* and an *automatic calibration*.

Manual Compass Calibration

In this method sending a command of 24 with a parameter of 2 causes the firmware to start the calibration process but *in the background*. It will also immediately return the 5 bytes without delay, *allowing the RB program to continue processing and to issue other commands as needed*.

This mode does *not* cause a timeout since it returns immediately. Also remember that the compass needs to be turned around slowly preferably twice and on a level surface. This can be accomplished manually by hand, or the RB program can issue the turn command (12 or 13) in a loop for 20 seconds. Just do not try to issue further compass commands before 20 seconds are out; you will get 0 if you do.

Automatic Compass Calibration

In this mode you will issue command 24 with a parameter other than 2. The firmware will then start the calibration process but it will automatically cause the motors to keep turning for a period of 20 seconds.

This mode *will* cause a timeout, unless you have modified the timeout parameters in both the firmware and RB using the commands to do so before performing the calibration. However, you do not really need to do so. Just take appropriate measures in your RB code to handle the time out. The best way to do that is to use a delay of 20 seconds in the code right after issuing the command.

Do note that even though the firmware will time out and so will RB you still won't be able to do any further actions for 20 seconds. During those 20 seconds the motors will be turning (i.e. rotating the robot).

The manual method is better because you have more control and it does not cause timeouts. However, the automatic option is useful in that turning the motors is performed automatically. Also since you cannot issue any further commands until the 20 seconds are over it means you are not likely to try to use the compass before the calibration is completed.

Complexity of Programming the Automatic Calibration:

How does the automatic calibration achieve turning the motors? The way described in Section 8.1.2 (option4). The **Others** cog flags the **Motors** cog after specifying the command 12 in the command buffer with a parameter of 1. It then waits for 20 seconds repeating the flagging and commanding to keep turning the motors. When the 20 seconds are over it returns to **Main**. This is why the time out occurs. Because both **Motors** and **Others** are busy for the duration, you must not issue any more commands that require either of these two cogs.



This procedure is a very good illustration of how the *inter-cog interactions* can be achieved. If you require this kind of control you now have an effective *template* to follow.

8.1.4 A Simulated Compass Instrument

We will now develop a program that displays the compass heading in a more interesting manner than just numbers on the screen. Compass_Animation.Bas is similar to Compass_Tester.Bas above but instead of printing out the heading as a text number it calls to subroutine called **DisplayCompass()**. However, you will notice that the subroutine is not

listed in the program. It is part of a library of subroutines called Instruments.Bas. The Compass_Animation.Bas program knows how to use the subroutine because the Instruments.Bas library has been *included* in the program. This is the purpose of the line:

```
#include "..\Utilities&IncludeFiles\Instruments.Bas"
```

This line tells the program where to find the file that has the subroutine. When Compass_Animation.Bas runs it will look for the file Instruments.Bas in the directory called Utilities&IncludeFiles that is in the parent directory of the one in which Compass_Animation.Bas resides. That is the reason we had the “..\Utilities&IncludeFiles\” before the name of the file. When Compass_Animation.Bas finds the library file, it incorporates it as if it were part of the program and when a call to **DisplayCompass()** is made it works.



The advantage of placing subroutines in a library is that many programs can use the subroutines. We will do precisely this with many programs to come. You will notice that Instruments.Bas has another subroutine that we will use later, so ignore it for now.

The **DisplayCompass()** subroutine implements an authentic looking Compass Instrument like ones found on boats or airplanes and it will behave very much like a real instrument. The subroutine is designed to be versatile and generic. You can pass optional parameters to it to configure where to place the instrument on the screen and how many gradations it will display. Additionally it will display the numeric value of the heading (not available on a real device). All the parameters are optional and if you do not specify any they will have default values. In the main program the subroutine is used in its default mode. We will use the same subroutine in Section 8.4.3 (see Figure 8.11), but by passing it different parameters, the instrument will be different in size and position.

If you are not a pilot or navigator, the heading markings might look to you as if they are the wrong way around. We will not go into the details of this here since this is not a book on navigation – but this is in fact how it is on a real compass instrument in real life.



The subroutine allows for a way to make the instrument display the markings in a more intuitive manner. The main program will provide a checkbox that you can uncheck to make the instrument have the gradations increase to the right. This illustrates how using a programming language like RobotBASIC can be a major advantage when creating GUI instrumentation. You can simulate authentic looking and behaving instruments or you can improve on the old mechanisms and increase the *ergonomic* effectiveness and create a more amenable *human interface*.

The HMC6352 is in fact just like a real compass. It has to be level to read accurate headings. If you pitch and roll the heading will change even if you did not turn. Again, we will not discuss the reasons for this, but notice how the compass heading changes when you do any roll or pitch. Pickup the PPDB and keep the reference axis pointing in the same direction and keep it straight and level. Note the heading. Now tilt the PPDB to the right or left or downward and upward. Notice how the heading changes. The change is in fact a predictable value depending on the bank angle and direction as well as what latitude you are at and what heading you are facing. A [gyroscopic device](#)²⁹ does not give different heading readings when you pitch and roll.

In order to make the display flicker free, the subroutine uses RobotBASIC's back-buffered screen graphics (**Flip on**). Comment out the line in the main program that says **Flip On** and observe what happens.



The HMC6352 compass module is just like a real compass and is subject to all compass errors: Variation, Deviation, Dip, Acceleration/Deceleration and Pitch and Roll.

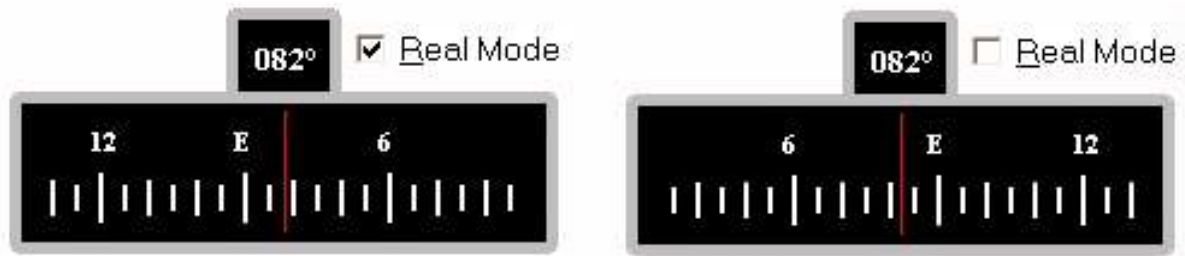


Figure 8.4: Screenshot of Compass_Animation.BAS simulating an authentic looking GUI Compass Instrument. Notice the difference between the real mode (left) and enhanced mode (right).

Compass_Animation.Bas

```
//Compass_Animation.Bas
//works with Firmware_Main.Spin
Port = 8 //change this as per your system
#include "..\Utilities\IncludeFiles\Instruments.Bas"
Main:
  setcommport Port,br115200
  flip on
  call SendCommand(24,1,s) //see if there is a compass
  if SendCommand_Result
    if !getstrbyte(s,5) then print "no compass available" \Terminate
  endif
  //uncomment the following three lines to invoke a manual calibration
  //print "Calibration in progress...rotate two turns while level"
  //call SendCommand(24,2)
  //delay 20000
  AddCheckBox "Mode",430,230,"&Real Mode",1,0
  while true
    call SendCommand(24,0,s) //read the compass
    if !SendCommand_Result then continue
    x = (getstrbyte(s,4)<<8)+getstrbyte(s,5)
    call DisplayCompass(!GetCheckBox("Mode"),x)
    Flip
  wend
end
//-----
sub SendCommand(C,P,&s)
  m = ""
  serialout C,P
  serbytesin 5,s,x
  if x < 5 then m= "Comms Error"
  xystring 500,20,m,spaces(30)
  return (x == 5)
```

Instruments.Bas (a library of reusable subroutines)

```
//Instruments.Bas
//to be used as an #include file in all programs
//that need to display the instruments
//-----
sub DisplayCompass(Mode,H,x,y,f,t)
  fnt = "Times New Roman"
  if !vType(Mode) then Mode = 0
  Mode = Limit(Mode,0,1)*2-1
```

```

if !vType(H) then H = 0
if !vType(x) then x = 400
if !vType(y) then y = 300
if !vType(f) then f = 2
if !vType(t) then t = 10
dim xx[2,t*2]
erectanglewh x-20,y-75,40,35,9,gray
rectanglewh x-20,y-75,40,35,gray,black
xyText x-12,y-65,Format(H,"000°"),fnt,10,fs_Bold,white,black
erectanglewh x-t*f*5-t,y-40,2*(t*f*5+t),60,9,gray
rectanglewh x-t*f*5-t,y-40,2*(t*f*5+t),60,gray,black
n= H # 5
for i=0 to t-1
    xx[0,i] = H-n-5*i \ if xx[0,i] < 0 then xx[0,i] += 360
    if xx[0,i] == 360 then xx[0,i] = 0
    xx[1,i] = Mode*f*(n+i*5)
    xx[0,i+t] = H+5-n+5*i \ if xx[0,i+t] == 360 then xx[0,i+t] = 0
    xx[1,i+t] = -Mode*f*(5-n+i*5)
next
for i=0 to t*2-1
    l = 5 \ hh = xx[0,i]
    if !(hh#10) then l = 7
    if !(hh #30)
        hh /= 10
        if hh == 0
            hh = "N"
        elseif hh == 9
            hh = "E"
        elseif hh == 27
            hh = "W"
        elseif hh == 18
            hh = "S"
        endif
        xytext x-xx[1,i]-5,y-30,hh,fnt,8,fs_Bold,white,black
        l = 10
    endif
    line x-xx[1,i],y-l,x-xx[1,i],y+l,2,white
next
line x,y-36,x,y+15,1,red
return
//-----
sub DisplayAttitude(Pitch,Roll,Cx,Cy,r,LW,CW)
    if !vType(Pitch) then Pitch = 0
    if !vType(Roll) then Roll = 0
    if !vType(r) then r = 100
    if !vType(Cx) then Cx = 400
    if !vType(Cy) then Cy = 300
    if !vType(LW) then LW = 2
    if !vType(CW) then CW = 10
//horizon
T = -Roll-Pitch \ TT = -Roll+Pitch+pi()
x1 = cartx(r,T) \ y1 = carty(r,T)
x2 = cartx(r,TT) \ y2 = carty(r,TT)
x3 = (x2+x1)/2 \ y3 = (y2+y1)/2
Circle Cx-r,Cy-r,Cx+r, Cy+r
line x1+Cx,y1+Cy,x2+Cx,y2+Cy,LW,red

```

```

//ground and sky
for i=-3 to 3 step 6
    T1 = -Roll-Pitch+dtor(i) \ TT1 = -Roll+Pitch+pi()-dtor(i)
    x1 = cartx(r,T1) \ y1 = carty(r,T1)
    x2 = cartx(r,TT1) \ y2 = carty(r,TT1)
    x4 = (x2+x1)/2 \ y4 = (y2+y1)/2
    j = brown
    if i < 0 then j= lightcyan
    floodfill Cx+x4,Cy+y4,j
next
//ground texture arrays
if !vType(_DAI_Flag)
    dim DAI_b[0]
    data DAI_b;5,10,20,40,60
    dim DAI_a[0]
    data DAI_a;0,dtor(30),-dtor(180),-dtor(40),dtor(10),-dtor(140)
    _DAI_Flag = true
endif
//horizontal ground texture
for i=0 to 4
    T1 = -Roll-Pitch+dtor(DAI_b[i]) \ TT1 = -Roll+Pitch+pi()-dtor(DAI_b[i])
    x1 = cartx(r,T1) \ y1 = carty(r,T1)
    x2 = cartx(r,TT1) \ y2 = carty(r,TT1)
    line x1+Cx,y1+Cy,x2+Cx,y2+Cy
next
//diagonal ground texture
j=dtor(20) \ i=T+j
repeat
    x1 = cartx(r,i) \ y1 = carty(r,i)
    line Cx+x3,Cy+y3,Cx+x1,Cy+y1
    i += j
until abs(i) > abs(TT-j+.2)
Arc Cx-r,Cy-r,Cx+r, Cy+r,,,CW,gray //instrument rim
//roll gradations
for k=0 to maxdim(DAI_a)-1 step 3
    i = -Roll+DAI_a[k] \ j=DAI_a[k+1]
    TW = CW/2
    if k >=3 then TW = 2
    rr1 = r+TW \ rr2 = r-TW
    repeat
        x1 = cartx(rr1,i) \ y1 = carty(rr1,i)
        x2 = cartx(rr2,i) \ y2 = carty(rr2,i)
        line Cx+x1,Cy+y1,Cx+x2,Cy+y2,2,white
        i -= j
    until i < -Roll+DAI_a[k+2]-.2
next
//roll or bank indicator
for j=-2 to 2 step 4
    i = -dtor(90-j)
    x1 = cartx(r+CW/2,i) \ y1 = carty(r+CW/2,i)
    x2 = cartx(r-CW/2,i) \ y2 = carty(r-CW/2,i)
    line Cx+x1,Cy+y1,Cx+x2,Cy+y2,3,red
next
//small airplane
rr = r/10
circlewh Cx-2,Cy-2,4,4,white

```

```

line Cx,Cy,Cx,Cy+rr-1,2,white
Arc Cx-rr,Cy-rr,Cx+rr,Cy+rr,pi(),pi(),2,white
Line Cx-rr,Cy,Cx-4*rr,Cy,2,white
Line Cx+rr,Cy,Cx+4*rr,Cy,2,white
Return

```

8.2 A Procedural Strategy for Adding Other Hardware

As you have seen so far, because of the way the system is designed, adding hardware is extremely simple and routine. As a matter of fact, the hardware we added covers almost every category of hardware that you are likely to want to incorporate into your system.

List 1: Categories of Hardware

- a) Digital hardware with On/Off type I/O (Pushbuttons, LEDs)
- b) Digital to Analog output (Dimmer LED)
- c) Pulsating Frequency output (Blinking LEDs and Speaker)
- d) Analog To Digital input with RC-Time (Pots)
- e) Controlling Servomotors (Servomotors)
- f) Counting Time Intervals (Ping and RC-Time)
- g) I²C I/O (Compass)
- h) RS232 I/O (FDS, SM)
- i) Using Counters (in Duty, NCO, and Edge Detector modes)

List 2: Programming Techniques Required to Develop the Firmware

- j) Using Semaphores and Flags
- k) Using Parallelism
- l) Using Polling
- m) Sharing RAM
- n) Inter-Cog communications and control
- o) Creating objects and methods

Just about any hardware that you are likely to want to add as well as the programming techniques required to add them to the firmware are most likely to belong to one of the above categories. Let's have a look at some hardware that we may wish to add to a project:

Table 8.1: Possible Hardware and its Category

Hardware	Category
Bumper Switch ³⁰	a
Infrared Proximity Sensors ³¹	a or c
QTI Line Sensors ¹⁴	a or d
PIR Movement Sensor ³²	a
Turret ³³	e
Accelerometer ³⁴	g
GPS ³⁵	h or g
DC motors ³⁶	e
Thermometer ³⁷	G
2-Axis Joystick ³⁸	D
Sound Impact Sensor ³⁹	A
5-Way button ⁴⁰	A
Piezoelectric Speaker ²⁷	I
Quadrature System ⁴¹	H

8.2.1 Commands in the Protocol So Far

In our protocol so far we have allowed for many possible commands and Table 8.2 below is a good overview.

Table 8.2: List of protocol command codes at this stage.

Command	Code	Parameter	Updates Critical Sensors	Data Returned
Stop Motors	0	0	Yes	None
Forward	6	Amount	Yes	None
Backwards	7	Amount	Yes	None
Turn right	12	Amount	Yes	None
Turn Left	13	Amount	Yes	None
Read the Compass	24	0	Yes	Last two bytes
Check if the compass is available	24	1	Yes	4 th byte 0, 5 th byte is 1 for yes or 0 for no
Calibrate the Compass	24	2=Manual 3=Automatic	Yes No	None
Read the Pots	66	0	No	First 4 bytes
Read the Ping)))	192	0	Yes	Last two bytes
Set P20..P18 LEDs	1	LED States	Yes	None
Set P21 Frequency	2	Hz Value	Yes	None
Reset the Propeller	255	0	No	None
Set P23 LED brightness	200	Level	Yes	None
P22 LED Blink duration	201	Level	Yes	None
Set 2 nd byte receive Timeout1	202	N x 10ms	Yes	None
Set operations Timeout2	203	N x 10ms	Yes	None
Set L_Speed	240	Speed	Yes	None
Set T_Speed	244	Speed	Yes	None
Set L_Timeout	241	N X 10 ms	Yes	None
Set T_Timeout	245	N X 10 ms	Yes	None
Set StepTime	242	N X 10 ms	Yes	None
Set TurnTime	243	N X 10 ms	Yes	None

8.2.2 A Procedural Strategy For Extending the Hardware

Much of the hardware you may wish to add is likely to be just a matter of deciding what category it is under (List 1 and List 2 and Table 8.1) and then looking at the commands in Table 8.2 to decide which command resembles it best. Once you have decided on this, use the command from Table 8.2 as a template for adding the new hardware. It is not just hardware that we might want to add. We may also want to add more housekeeping commands.

List 3: There are three types of commands:

1. Ones that set/change system parameters (**Main** object but can be any of the objects)
2. Ones that do something in the background and do not need to be commanded (**Reader** object)
3. Ones that carry out a task and then
 - a. Do not return data (**Motors** object but can be **Others** too)
 - b. Return data in the last two bytes of the primary send buffer (**Others** object)
 - c. Return data in all or some bytes of the secondary send buffer (**Others** object)

Procedure For Adding a New Hardware or Command

To add a new command you need to

- i. Decide which category of hardware it is from Lists 1 and 2 and Table 8.1.
- ii. Decide what type of command it will be from List 3.
- iii. Select a template command from Table 8.2.
- iv. Modify the appropriate object to incorporate the methods needed to interact with the hardware and fill the send buffer if required. If you decide that you need a new object then use one of the existing objects as a template and modify it as needed.
- v. Add any constants in the **CON** section.
- vi. Add any variables in the **Var** or **Dat** section.
- vii. Instantiate any required supporting objects in the **Obj** section and invoke their **Start()** methods in the **Initialization** method.
- viii. Decide on a code for the command (make sure there is no clashing) and what parameters it has to be passed.
- ix. Add the Case statement in the Case block to call the method. This should follow the template command.
- x. Add the Case Statement in the **Main** object to allow for the new case statement in the subordinate object. This should follow the template command.

To illustrate the process we will now add new commands to:

- ☐ Allow for actuating the motors separately in any direction for a certain number of steps or to keep them on (Section 8.2.3)
- ☐ Allow the Ping))) mounted on a turret to be turned by 90 degrees right and left before measuring the distance (Section 8.2.4).
- ☐ Save all the system parameters to the EEPROM. We will also extend the system to read them from the EEPROM upon boot up if there are any valid saved ones (Section 8.3) and also allow for resetting them to factory settings.
- ☐ Add an Accelerometer (Section 8.4).
- ☐ Add three QTI infrared line sensors (Section 8.5).
- ☐ Add a speaker similar to Chapter 5.8 (Section 8.6).

8.2.3 Controlling Motors Separately

If you have noticed with our commands for controlling the motors they can only be rotated together. This is what we need on a robot for example. However, it may be desirable to control the motors separately in certain occasions, such as if you wish to effect curved turns where the center of turning is not the center of the robot's wheel axis. Also, if the motors are used in a process other than robotics, we may want to be able to control the motors as separate entities.

In the process of implementing independent control, we want to illustrate how following the procedure outlined in Section 8.2.2 makes the process simple and quick.

- | | |
|------------|--|
| Step i: | It is still a servomotor. |
| Step ii: | It is like commands 6,7 in Table 8.2. |
| Step iii: | Commands 6 and 7, but allowance has to be made for different processing. |
| Step iv: | We will do them in the Motors object (see bold lines in the listing). |
| Step v: | Not required. |
| Step vi: | See Listings (added variables in Motors object) |
| Step vii: | Not required; but for Main we changed the name of the Motors object to allow for the new version. |
| Step viii: | Codes 8/9 move the right motor forward/backward and Codes 10/11 for the left motor. We will have it so that parameter 0 means stop, parameter 255 means stay on. Any other number is for n-steps. But the command will not wait for the steps to be completed, it will always return immediately.
Therefore, We will need to allow for timing and switching the motors off in the Process0 method. |
| Step ix,x: | See bold lines in the listings. |

Here are the listings of the new Firmware_Motors_B.Spin and Firmware_Main_B.Spin. Notice the bold code performs the steps above. We will only list areas where there are changes. The rest of the code is as before. The **Others** and **Reader** objects are not changed and we will use the same ones as before.

[Cut Out]

Testing the New Commands

[Cut Out]

8.2.4 Controlling a Ping))) on a Turret

If we mount the Ping))) on a [Servomotor turret](#)³³ like the one sold by Parallax, we can extend the utility of the ranger because we can then turn it left and right. The way we have previously implemented the ranger command (#192) the parameter is ignored. If we allow the parameter to specify the angle right of the straight ahead and if we add another command (#193) and allow the parameter to define left of the straight ahead then we would have full control over the turret and once the turret finishes the turn, a reading of the Ping can be taken and returned as before.



Since now the Ping))) would be mounted on a turret the connection to it should now be made using a 3-wire cable (just like the one for the servomotors) to one of the 8 servomotor headers on the PPDB (see Figure B.3 in Appendix B). The - pin (black) on the header goes to the Ping's Ground pin. The + (red line) goes to the 5V pin on the Ping))) and the S pin (white) goes to the Sig pin on the Ping. There is no longer a need for the 1K Ω since the servos header has a 150 Ω resistor already and that should suffice. See Figure B.3 in Appendix B.

As discussed in the strategy outlined in Section 8.1.2, we will have the **Main** cog control the *division of labor*. **Main** will receive the command 192 or 193. It will then flag the **Motors** cog to turn the Servomotor of the turret (we will use a [standard servomotor](#)⁴² on P15) to move to the correct position. When the **Motors** flag is lowered **Main** will then flag the **Others** cog as before to read the Ping))).

To implement the required changes we will follow the steps in Section 8.2.2. The changes are only to **Main** and **Motors**. Another thing to note here is that due to the division of labor aspect the **Others** object needed no changes at all despite the new commands being all to do with the Ping))) which is read by the **Others** cog. We also added a constant in the **Motors** object so that we can data-map the value 0-90 into in a number that causes the turret to be truly turned 0 to 90 degrees, where 0 is straight ahead. Remember that servomotors have the 1500 microseconds pulse as the center. We want to limit the Max and Min values so that the motor will turn 90 degrees either way. Thus we can use the number 0 to 90 as

ServoSignal := 1500+n*Max/90 'for the left turns

ServoSignal := 1500-n*Min/90 'for the right turns

We need to experiment to determine what Max and Min have to be. We can always of course add two more commands to set these values at run-time from RB. But we will leave this up to you. Use Servo_01.Spin to experiment with the servomotor to see what values set it to about 90 degrees either way and use these values. Remember we are using P15 as the motor's signal pin so change the Pin number to 15. We did this and for our motor the Max and Min are both 800 (2300-1500 = 800 and 1500-700 = 800)

[Cut Out]

A Radar Application

To test the new firmware we made the interesting program Turret_Radar.Bas. What you should pay most attention to is the **Ranger()** subroutine. Notice in this subroutine how we check if the angle is negative or positive and send the appropriate command accordingly (192 or 193) with the angle made positive. The other two subroutines are what implements the RADAR simulation. Notice the **SaveScr** and **RestoreScr** commands and also the usage of **cartx()** and **carty()**. In Chapter 10 we will see a slightly different version of this program.

Turret_Radar.Bas

```
//Turret_Radar.Bas
//works with Protocol_Main_C.Bas
Port = 8 //set this as per your system
Main:
  setcommport Port,br115200
  call RadarScreen()
  call Radar()
End
//-----
sub Ranger(Angle,&Value)
  C = 192 \ Value=-1 \ m= "Comms Error"
  if Angle < 0 then C = 193
  Angle = Limit(Abs(Angle),0,90)
  serialout C,Angle
  serbytesin 5,s,x
  if x == 5
    Value = (getstrbyte(s,4)<<8)+getstrbyte(s,5)
    m = spaces(40)
  endif
  xyText 600,10,m,,10,,red
return (x==5)
//-----
sub RadarScreen()
  for i=1 to 400 step 50
    arc i,i,800-i,800-i,0,pi(),2,gray
  next
  for i=0 to 180 step 20
    th = dtor(i) \ r = 400
    line r,r,r+cartx(r,th),r-carty(r,th),1,gray
  next
  savescr
return
//-----
sub Radar()
  j=-90 \ i=1
  while true
    call Ranger(j,V)
    if V < 0 then continue
    V *= 400/23000. \ th = dtor(j-90)
    x = cartx(V,th) \ y = carty(V,th)
    circlewh 400+x-5,400+y-5,10,10,red,red
    j += i \ if abs(j)==90 then i= -i \ restorescr
  wend
return
```

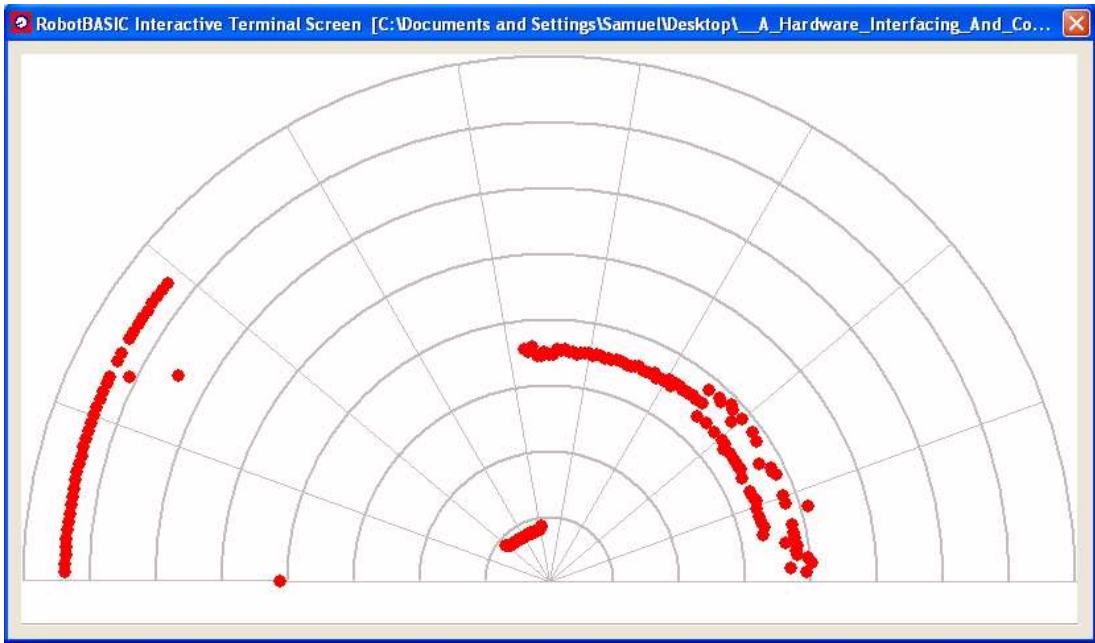


Figure 8.5: Screenshot of Turret_Radar.Bas in action.

8.3 Saving The System Parameters to EEPROM

As you have seen so far and as detailed in Table 8.3 below, we are able to change the values of the parameters shown in the table. As the system stands you can set the values but whenever the Propeller is rebooted the values will always revert to the ones assigned to them in the programs as shown in the listings. Some are operational, but most would be nice to retain so that whatever value you assigned to them last would be the value upon reboot.

Table 8.3: List of changeable system parameters

Parameter	Purpose	Cog
L_Speed	The speed for the motors in forward/backward travel	Motors
T_Speed	The speed of the motors while turning	Motors
StepTime	The time needed to accomplish a step of forward/backward travel	Motors
TurnTime	The time needed to accomplish a degree of turning	Motors
L_TimeOut	The time to leave motors on until a new command arrives in linear tavel	Motors
T_TimeOut	The time to leave motors on until a new command arrives in turning	Motors
P22 Duration	The on/off duration for the P22 LED in the Reader cog	Main
TimeOut1	The timeout period to wait for the parameter to arrive	Main
TimeOut2	The timeout period to wait for a command to finish	Main
P21 frequency	The blinking frequency for the P21 pin in the Others cog	
P23 Level	The voltage level for the dimmer LED in the Main cog.	

8.3.1 EEPROM Limitations

[Cut Out]

8.3.2 Required Changes to The Firmware

[Cut Out]

8.3.3 CRC and Validity Check

[Cut Out]

8.3.4 The New Commands & Firmware

In addition to rearranging things so that the two parameters that are not already in the contiguous buffer are moved over to the buffer in **Main**, we will also use the [Basic I2C Driver.Spin](#)⁶⁸ as the object that has all the necessary I²C protocols to communicate with the 24LC256 EEPROM and store/read data from it (included in the downloadable zip file too).

We will provide two new commands

Command code 5:

If the parameter is 0 it will store the current system parameters as they are in RAM to the EEPROM. It will return in the 4th and 5th bytes of the primary send buffer a \$01 if the operation succeeds or a \$00 if not.

If the parameter is 1 the system parameters will be restored *in RAM only* to the factory settings. This does *not* affect the EEPROM. If you want the factory settings to be in effect on the next boot up you must also issue another command 5 with parameter 0 to save the RAM parameters to the EEPROM.

If the parameter is 2 the system parameters as they are in the EEPROM will be sent out to the PST as text numbers. The PST screen will then display the values.

If the parameter is greater than 2 the system parameters that are in the RAM will be sent to the PST as text numbers.

Command code 4:

The system parameter (from RAM) is sent to RB using the secondary send buffer with the first 4 bytes being the system parameter in Little-Endian format. That is the 1st byte (byte 0) is the LSByte and the 2nd byte is the next byte and so on. The fifth byte is set to 0 to indicate that the requested parameter is a valid one.

Which system parameter is sent depends on the parameter of the command. If the requested parameter number is too large then all the returned 5 bytes will be 0xFF to indicate a wrong requested parameter number (i.e. -1). The order is from 0 to N (N=10 for now). See Table B.2 for the order and description of the parameters (or Table 8.4).

Modifications are mostly to the **Main** object, with two new methods in **Motors** to return a pointer to the buffer and to restore the factory settings. Rearrangement of **Others** implements the new setup for the parameters that are now stored as part of the buffer in **Main** instead of as variables in **Others**.

The new firmware suite is called Firmware_XXXX_D.Spin where XXXX is Main, Others, and Motors. **Reader** is not changed and is not renamed. All the changes are bold lines in the listings. Only changed areas are listed.



Remember that command 5 with parameter 1 will restore the factory settings but *only in RAM*. If you wish to also reset the EEPROM so that the settings will be factory settings on the next reboot, you must also save the restored factory settings to the EEPROM (command 5 parameter 0).

[Cut Out]

8.3.5 Testing the EEPROM Commands

EEPROM_Tester.Bas exercises all the new commands. Compile the new Firmware_Main_D.Spin and save to EEPROM (F11) then run the RB program. You may want to also run the PST and have it so that it does not disable when it loses focus because we want to go to the RB program and interact with it

The program will

- ☐ Print all the EEPROM parameters (none to start with and they all should be 0) on the PST screen.
- ☐ Print all the RAM parameters (should be as the constants in the program code) on the PST screen.
- ☐ Print on the RB screen all the RAM parameters (same as 2).
- ☐ Modify some of the parameters.
- ☐ Save the parameters to the EEPROM and check if successful.
- ☐ Reset the Propeller and wait for it to reboot.
- ☐ Print all the EEPROM parameters (now they should be the same as set in 4) on the PST screen.
- ☐ Print all the RAM parameters (should be the same as 7) on the PST screen.
- ☐ Print all the RAM parameters (same as 8) on the RB screen.
- ☐ Restore Factory Settings.
- ☐ Save the parameters to the EEPROM.
- ☐ Print all the RAM parameters (should be as the constants in the program code) on the PST screen.
- ☐ Print all the EEPROM parameters (should be as in 12) on the PST screen.



In the program when printing the parameters to the RB screen we will use an extra count (12 instead of just 11) to see how reading an invalid parameter returns -1 (0xFFFFFFFF).

EEPROM_Tester.Bas

```
//EEPROM_Tester.Bas
//works with Firmware_Main_D.Spin
Port = 8 //change this is as per your system
Main:
  setcommport Port,br115200
  call SendCommand(5,2) 'print EEPROM params to PST
  call SendCommand(5,3) 'print out RAM params to PST
  for i=0 to 11 //using an extra to demo how it returns -1
    call SendCommand(4,i,s)
    print BuffreadI(s,0)," "
  next
  call SendCommand(200,250) 'set the P23 brightness
  call SendCommand(201,0) 'no blinking on P22
  call SendCommand(2,0) 'no blinking on P21
  call SendCommand(5,0,s) 'save to EEPROM
  m = "Saving to the EEPROM failed"
  if SendCommand_Result
    if getstrbyte(s,5) then m = "Saving to the EEPROM succeeded"
  endif
  print m
  print "resetting the propeller . . . wait 3 secs"
  call SendCommand(255,0) 'reset the Propeller
  delay 3000 'wait for Prop to finish reboot
  print
  call SendCommand(5,2) 'print EEPROM params to PST
  call SendCommand(5,3) 'print out RAM params to PST
  for i=0 to 11
    call SendCommand(4,i,s)
    print BuffreadI(s,0)," "
```

```

next
print "Resetting factory settings and saving to the EEPROM"
call SendCommand(5,1) 'restore factory settings
call SendCommand(5,0) 'save to EEPROM
call SendCommand(5,1) 'print RAM params to PST
call SendCommand(5,2) 'print EEPROM params to PST
print "all done"
end
//-----
sub SendCommand(C,P,&s)
  serialout C,P
  serbytesin 5,s,x
  if x != 5 then return false
return (x==5)

```

8.4 Adding an Accelerometer

An accelerometer module is a very useful device in many robotic projects. To that end we will incorporate the [H48C Tri-Axis Accelerometer module](#)³⁴ (see Figure 2.10). The connection schematic is shown in Figure 8.7 below.

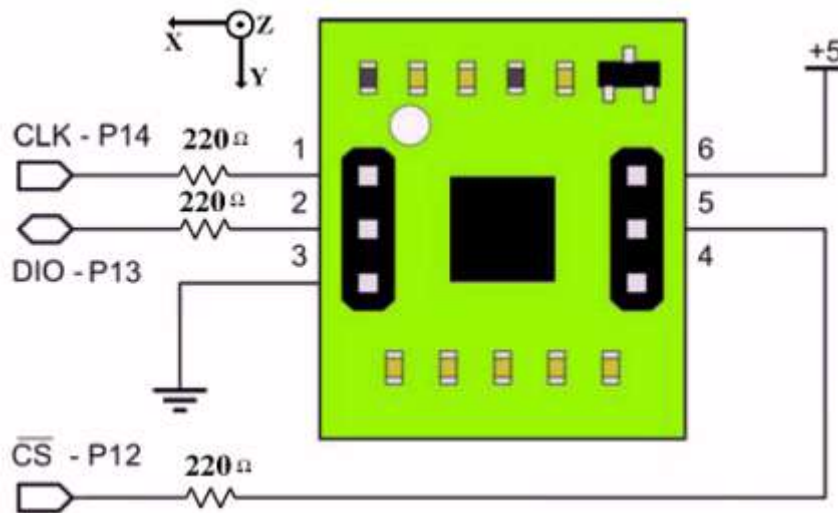


Figure 8.7: H48C Connection Schematic

8.4.1 Adding the Accelerometer Commands to the Protocol

The accelerometer provides acceleration values for the three axes (x,y,z). From these values we can also *calculate* the tilts of these axes using math functions in RB. We need to obtain 3 parameters from the device; all will be 16 bit numbers (actually 12 see later). We can make our protocol return these values in the 4th and 5th bytes of the primary send buffer. This would then require RB to send a command to request each axis one at a time. This might not be quite good enough especially if you consider that the acceleration values are usually needed to control a robot in a very dynamic situation where we would need these values as quickly as possible. Algorithms that need acceleration values are for controlling a walking robot or a balancing robot for instance. In such situations any delay in obtaining the readings may cause algorithms to be sluggish or even fail altogether.

A better alternative is to use the secondary buffer. However, there is a snag. Since the data is 16 bits that means we need 2 bytes for each value and since there are 3 we would need 6 bytes. However, our protocol only allows 5 bytes

and it is not possible to return all three values in one go. We would require two commands to get all three values. This may be acceptable if we are working in two dimensions and we mount the device to give us the most advantageous orientation of the x-y-plane. After all many robots have been designed with only 2-axis accelerometers and the algorithms worked quite well. If we want to use all three axes then the application should not be a very dynamic where acceleration values change too rapidly for the software's sampling rate since reading the z-axis as a separate command requires another command cycle.

There is a way to return all three readings in 5 bytes; all three values can be obtained with one command and one communications cycle and therefore as fast as our protocol allows. The data from the H48C does not quite need 16 bits since the maximum value can only be 4095 (0xFFF). So the maximum value is 3 nibbles. To transmit all three values we need 9 nibbles, which fit quite easily within our 5 bytes (10 nibbles) with 1 nibble to spare.

The 48HC values for the axes are actually voltage DAC values in reference to a reference voltage value. We will need to read this value for maximum accuracy. However this needs to be performed once upon startup since it is not going to vary during the operations of the device. It should almost always be $(4095/2) \pm 2$, which is between 2045 and 2049 with it almost always being 2047. Of course you can always read the reference voltage every time before reading the axes' values but this will slow the whole operation a little and for a very dynamic system you can dispense with reading the reference voltage except for the first time. The gained accuracy is inconsequential as compared to the loss in speed.

The command to interrogate the H48C will have two modes depending on the parameter passed to it. If the parameter is other than 1 then the values of the axes' acceleration are returned in the secondary send buffer as 5 bytes. If the parameter is 1 the command will return the value of the reference voltage in the 4th and 5th bytes of the primary send buffer.

The procedure is as follows:

- ❑ Before using the H48C commands for the first time issue command 70 with parameter 1 and reconstitute the reference voltage value from the 4th and 5th bytes (MSByte first) and store the value (vRef).
- ❑ Whenever you need the acceleration value issue command 70 with a parameter of 0 (or any number other than 1) and reconstitute the values for the x-axis from the first three nibbles (xRef), the y-axis from the next three nibbles (yRef) and the z-axis from the last three nibbles (zRef).
- ❑ Once you have the raw values for the axis readings you can calculate the actual acceleration in reference to 1g using the formula (replace x with y and z for the other axis):
$$xG = (xRef - vRef) * 0.0022$$
- ❑ To obtain tilt angles you can either calculate them from the raw data after subtracting vRef or from the calculated g values (xG above) using the **aTan2()** function in RB. So for example to get the tilt of the x-Axis you would do
$$\mathbf{aTan2(xRef - vRef, zRef - vRef)}$$

to get the angle in radians or
$$\mathbf{rTod(aTan(xRef - vRef, zRef - vRef))}$$

to get the angle in degrees.

To reconstitute the axes' values from the 5 bytes (see Figure 8.8):

- ❑ First byte and the MS-Nibble of the second byte constitute the xRef value.
- ❑ LS-Nibble of the second byte and the third byte constitute the yRef value.
- ❑ The fourth byte and the MS-Nibble of the fifth byte constitute the zRef value.

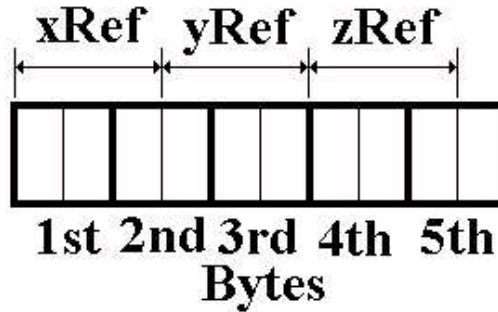


Figure 8.8: Order of the Acceleration Reference Values In the buffer.

You will see all this in the code listings in the Spin program and in the RB program.



The orientation of the device on the PPDB itself has to be taken in consideration if you need the readings to be in reference to the PPDB axes. See Figure 8.7 for how the positive x-axis and y-axis are oriented and also the z-axis is pointing upwards from the plane of the figure. Also see Figure 8.10.

8.4.2 Incorporating the H48C in the Protocol

[Cut Out]

8.4.3 Testing the New Command

In the previous section you saw how the values of the three axes were placed in the 5-byte buffer. On the RB side the 5 bytes have to be broken up and the nibbles extracted to reconstitute the numbers. You can see all this in the listing of H48C_Tester.Bas in the **Read_H48c()** user defined subroutine. The subroutine will read the **vRef** and the axes values if the passed parameter **vRef** is 0 and if it is not 0 then only the axes' values will be read. The subroutine is a useful one you can use in other programs to read the H48C values.

Also notice in the main program how the tilt angles are calculated using the **aTan2()** and **rToD()** functions to get the angle in degrees.



For later programs that require the use of the H48C we will put the subroutine **Read_H48C()** in the Instruments.Bas include file we used previously in Section 8.1.4.

H48C_Tester.Bas

```
//H48C_Tester.Bas
//Works with Firmware_E.Spin
Port =8 //change as per your system
Main:
  fmta = "#0.0000 " \ fmbt = " #00 "
  setcommport Port,br115200
  v=0
  while true
    call Read_H48C(v,x,y,z,gX,gY,gZ)
    xstring 10,10,v;x-v;y-v;z-v;spaces(20)
    xstring 50,30,Format(gX,fmta),Format(gY,fmta),Format(gZ,fmta)
    xstring 50,50,Format(rtod(atan2(gX,gZ))-90,fmbt)
    xstring -1,-1,Format(rtod(atan2(gY,gZ))-90,fmbt)
    xstring -1,-1,Format(rtod(atan2(gZ,gY)),fmbt)
```

```

    //v = 0    //uncomment this to refresh the vRef all the time
wend
End
//=====
sub SendCommand(C,P,&s)
    m = ""
    serialout C,P
    serbytesin 5,s,x
    if x < 5 then m= "Comms Error"
    xystring 500,20,m,spaces(30)
return (x == 5)
//=====
sub Read_H48C(&vRef,&xRef,&yRef,&zRef,&xG,&yG,&zG)
    xRef = 0 \ yRef = 0 \ zRef = 0
    xG = 0 \ yG = 0 \ zG = 0
    if vRef == 0
        call SendCommand(70,1,s) //read vRef
        if !SendCommand__Result then return false
        vRef = (getstrbyte(s,4)<<8)+getstrbyte(s,5)
    endif
    call SendCommand(70,0,s) //read the axes
    if !SendCommand__Result then return false
    xRef = (getstrbyte(s,1) << 4) + (getstrbyte(s,2) >>4)
    yRef = ((getstrbyte(s,2)&0x0F) << 8) + getstrbyte(s,3)
    zRef = (getstrbyte(s,4) << 4) + (getstrbyte(s,5) >> 4)
    xG = (xRef-vRef)*.0022 //convert to g-forces
    yG = (yRef-vRef)*.0022
    zG = (zRef-vRef)*.0022
return true

```

8.4.4 Three Dimensional Animation of Airplane Pitch, Roll & Heading

Using acceleration data has numerous uses in the field of engineering. Combined with heading data from a compass you can control cars, airplanes, ships, submarines and robots. You can create balancing and walking robots. You can control a robot arm with accurate positioning.

Using acceleration data (with a gyroscopic unit) you can create a very viable Inertial Navigation System. An INS uses acceleration data to calculate speeds and translations (distances) from a start point. With an INS you do not even need a compass to know where you are. Using the translations in the three axes you can calculate how high and where you are quite accurately. The math is quite complex; it is not just straightforward integration. There are nuances to things like using filtering to filter out noisy data and combining data from other sources like gyroscopes and compasses. See Chapter 10 for an implementation of a *very simplistic* but entertaining INS of sorts.

Another use for acceleration data is calculation of Pitch and Roll. Pitch is the angle between the longitudinal axis (body) of an airplane (for instance) and the horizontal. Roll is the angle between the lateral axis (wings) and the horizontal. For full control of an airplane one also needs the Heading, which is the angle between the x-axis and the magnetic north pole. This is obtained from a compass.

If our PPDB were to be placed on an airplane with the HMC6352 and H48C we would be able to acquire information on the attitude of the airplane in 3D-Space. To calculate the pitch and roll we assign the longitudinal axis as the x-axis and the lateral axis as the y-axis. Heading will be a rotation around the z-axis.

To calculate pitch we will need to find the tilt angle the longitudinal (x-axis) is making with the vertical (z-axis). This is $\text{aTan}(gX, gZ)$. Likewise the roll is the angle of the y-axis with the vertical. This is $\text{aTan}(gY, gZ)$. For the heading we will use the compass reading.

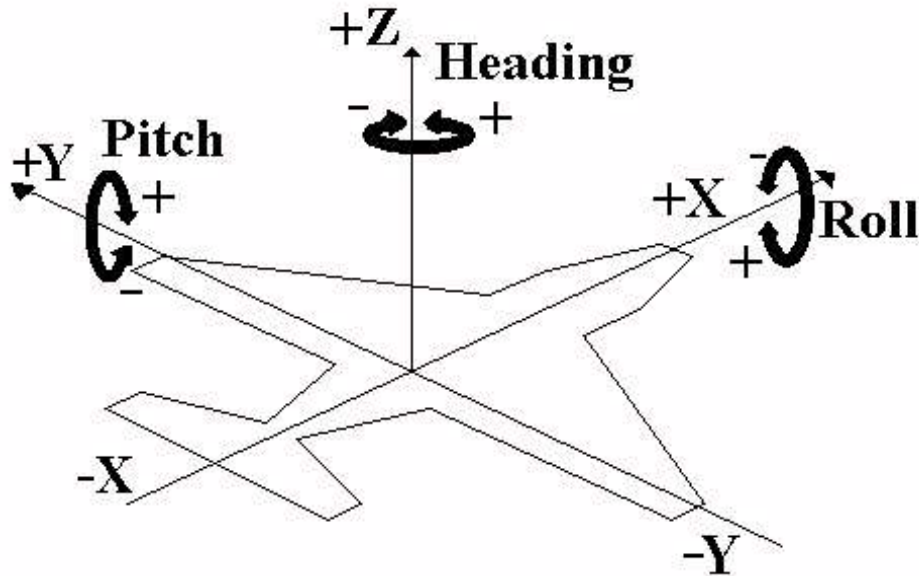


Figure 8.9: Pitch, Roll and Heading



When we calculate the tilt of the x-axis or y-axis we are actually calculating the angle it makes with the vertical using **aTan2(gX,gZ)**. If the plane is level this angle will be 90° ($\pi/2$). We need to subtract $\pi/2$ from the calculated tilts to have the correct pitch and roll values.



In RobotBASIC the **aTan(x,y)** function requires the first parameter to be the x-value ($\cos\theta$) and the second the y-value ($\sin\theta$). In other languages (e.g. C++) it might be the other way round.



The tilt calculations require that the device **not** be experiencing any forces other than gravity. Any additional forces would introduce additional acceleration, which would render the tilt calculations incorrect due to the additional component accelerations. A better way to obtain pitch and roll is to use a gyroscopic device.

The program H48C_Plane.Bas will use the power of RB and the power of our protocol and firmware to display a 3D-Animation of the orientation in 3D space of a simple airplane representation. We will keep the program as simple as possible so not to cloud the issues with too much detail. Nevertheless, you will be quite impressed. You will be able to pitch and roll and turn the PPDB and you will see the airplane figure respond according to your movements. Moreover, there will also be a Compass instrument and Attitude Indicator (AI) instrument. The response is instantaneous. When you consider what is going on you will be quite surprised at how responsive and dynamic the display is. The system will:

- ☐ Send a command over the serial link to ask for the H48C data
- ☐ Receive the 5 bytes
- ☐ Reconstitute the 3-axis raw data
- ☐ Calculate the g-forces
- ☐ Calculate the tilt angles of the x and y axes
- ☐ Send a command to get the HMC6352 heading
- ☐ Reconstitute the heading
- ☐ Use these angles to calculate the transformation of the airplane's body coordinates
- ☐ Use RB's graphics engine to transform the body coordinates using matrix transformations

- ❑ Use RB's graphics engine to transform the 3D body coordinates into 2D coordinates
- ❑ Draw the airplane representation on the 2D screen
- ❑ Draw the Compass instrument including its required calculations
- ❑ Draw the Attitude Indicator (Artificial Horizon Indicator) and its required calculations

All the above has to be performed continuously and rapidly enough to be able to display a faithful representation of the plane and the two instruments in response to moving the PPDB in 3D-space in a convincing animation.

H48C_Plane.Bas uses RB's 3D graphics engine to transform the airplane's body coordinates (3D) and to calculate the screen coordinates (2D) of these transformed points so as to draw the plane on the screen. The program also uses some of the math functions in RB to calculate geometric properties of the AI instrument and to plot it and the compass instrument in 2D.

The transformations are rotations around the x,y and z axes. Heading is a rotation around the z-axis. Pitch is a rotation around the y-axis. Notice, it is the y-axis since pitch is a tilt of the x-axis as if the plane is hinged by its lateral (y) axis. Likewise Roll is a rotation around the x-axis. See Figure 8.9. The origin of the axes is at the center of gravity of the plane.

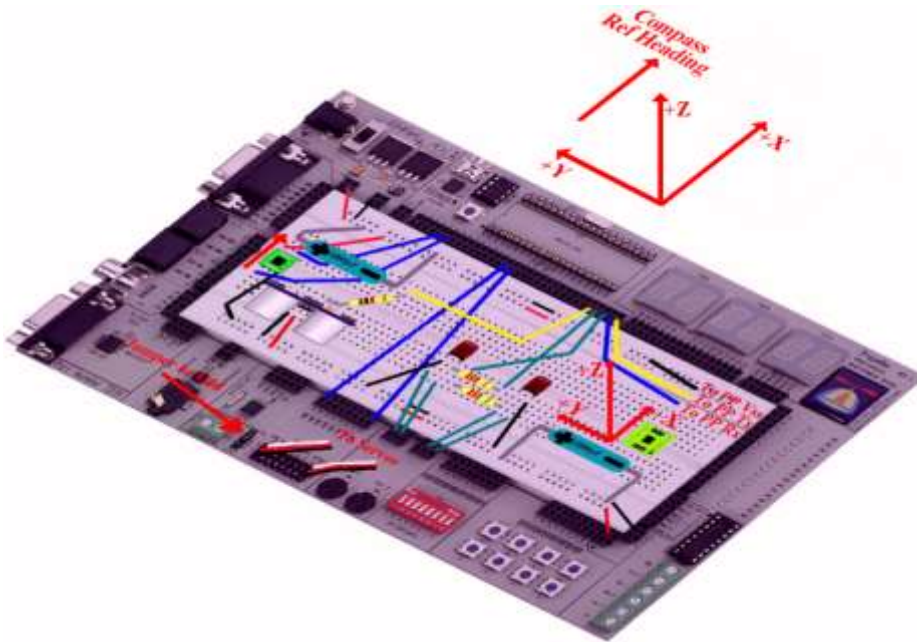


Figure 8.10: Compass and Accelerometer orientation and axes setup.

In the program we use the **Read_48HC()** subroutine that we used in the H48C_Tester.Bas program. Also notice the **PlotPlane()** subroutine. You are of course already familiar with the **SendCommand()** routine. Additionally, notice how the **Initialization** routine creates the body coordinates of the airplane. The airplane is not an elaborate image, all we want is to see the principle in action and complicating the program would not serve that purpose.

The H48C was placed on the PPDB so that the positive x-axis is pointing in the same direction as the heading reference on the HMC6352 compass (Figure 8.10).

The RobotBASIC 3D-graphics engine follows the right-handed coordinate system standard. The H48C Axis system obeys the right-handed standard too (see Figure 8.10). However, when tilt angles are calculated the positive Y-axis tilting down to the left (Figure 8.10) is a positive angle. This is opposite to the right-handed standard where a rotation around the x-axis as shown in Figure 8.10 is to the right. Thus we will need to make the tilt angle negative before we use it to calculate the rotation transformation around the x-axis (see bold lines in the listing below). The same for the Heading; the right-handed standard dictates that a turn to the left (Figure 8.10) is positive while compass turns are

positive to the right. Therefore we also need to negate the compass heading before using it in the rotation transformation around the Z-axis (see highlighted lines in the listing below).

In the listing you will notice that the line of code to transform the body coordinate points for the heading is commented out. This is because to control the plane you want the picture to be oriented as if you are looking at the plane from behind. If the line is uncommented the plane will be rotated in 3D-space and you will not be able to orientate yourself for the picture correctly. Try to uncomment the line and see how this affects your *perspective*.

Notice the use of the *include* file `Instruments.Bas` we used in Section 8.1.4. You already saw how we used the `DisplayCompass()` subroutine. We will use the routine in this program too, with some parameters to force the instrument to be of a certain size and position on the screen. The subroutine `DisplayAttitude()` is a similarly versatile and generic subroutine to draw an Attitude Indicator (AI) instrument for pitch and roll. The AI is a simple one but quite functional and gives an excellent feedback in addition to the 3D airplane representation of the roll (bank) and pitch. The subroutine `PlotPlane()` takes care of all the calculations and plotting of the 3D plane on the 2D computer screen. Also now the subroutine `Read_H48C()` from the program `H48C_Tester.Bas` in Section 8.4.3 has been moved to the include file, so there is no listing of it in the program below.

The 3D airplane and both the Compass and AI instruments are animations using RB's easy and intuitive graphics commands with a few mathematical functions. We will not explain the details. You should find it easy to figure out the program by just reading the code (also read the code of `Instruments.Bas` in Section 8.1.4). If there are commands and functions in the program with which you are not familiar, look them up in the RobotBASIC Help file. All the commands that start with **ge** are the graphics engine commands. RB's matrix manipulation commands are another powerful feature that enables the program to be so small yet so powerful.

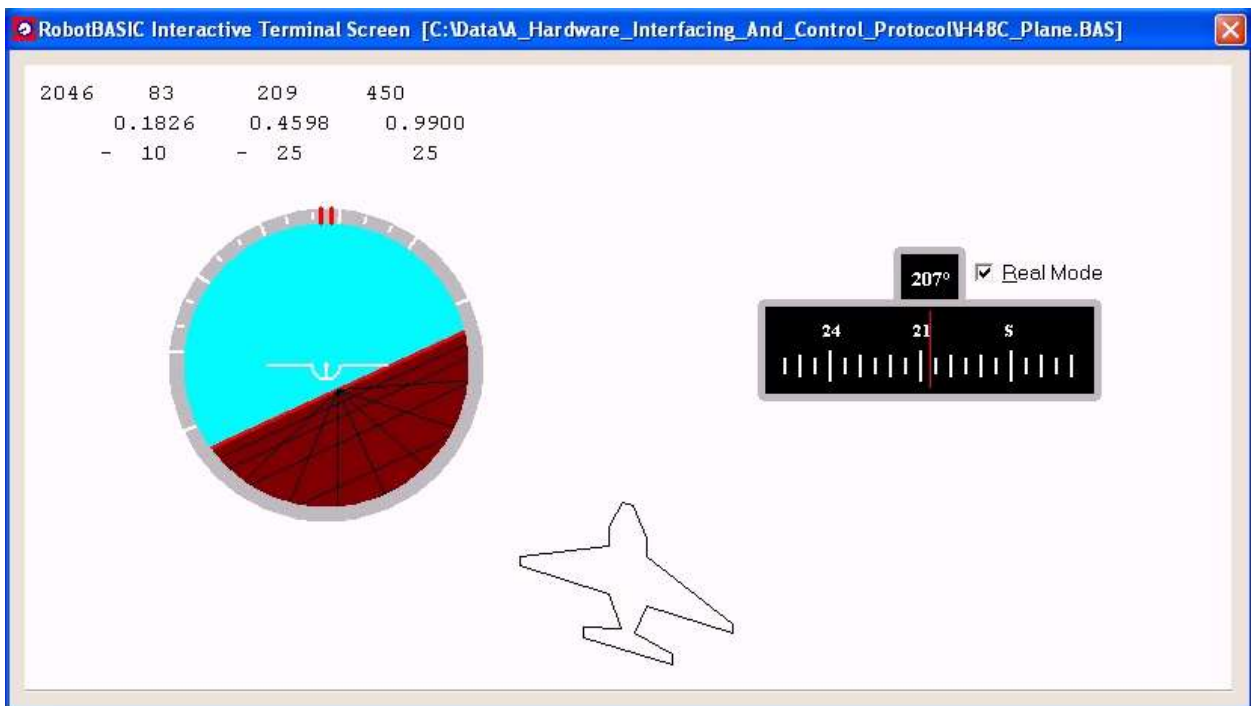


Figure 8.11: Screenshot of H48C_Plane.Bas

H48C_Plane.Bas

```
//H48C_Plane.Bas
//works with Firmware_Main_E.Spin
Port = 8 //change this as per your system
#include "..\Utilities\IncludeFiles\Instruments.Bas"
Main:
```

```

gosub Initialization
while true
    call Read_H48C(v,x,y,z,gX,gY,gZ)
    xystring 10,10,v;x-v;y-v;z-v;spaces(20)
    xystring 50,30,Format(gX,fmta),Format(gY,fmta),Format(gZ,fmta)
    xystring 50,50,Format(rtod(atan2(gX,gZ))-90,fmtb)
    xystring -1,-1,Format(rtod(atan2(gY,gZ))-90,fmtb)
    xystring -1,-1,Format(rtod(atan2(gZ,gY)),fmb)
    //v = 0    //uncomment this to refresh the vRef all the time
    call SendCommand(24,0,s)
    H = 0
    if SendCommand__Result then H = (getstrbyte(s,4)<<8)+getstrbyte(s,5)
    call PlotPlane(gX,gY,gZ,H)
wend
End
//=====
Initialization:
    fmta = "#0.0000    " \ fmb = "    #00    "
    setcommport Port,br115200
    v = 0
    data plane;9,.5,6,1.5,4,1.5
    data plane;0,8.5,-1,8.5,-1,1.5,-4,.5,-5,3.5,-6,3.5
    data plane;-6,-3.5,-5,-3.5,-4,-0.5,-1,-1.5,-1,-8.5,0,-8.5
    data plane;4,-1.5,6,-1.5,9,-0.5,9,.5

    data Eye;170,pi(),pi(.30),1550,400,350 //rho,theta,phi,d,Cx,Cy
    dim Plane[maxdim(plane)/2,5]
    mconstant Plane,0
    for i=0 to maxdim(plane)/2 -1
        Plane[i,0] = plane[i*2]
        Plane[i,1] = plane[i*2+1]
    next
    flip on
    AddCheckBox "Mode",630,130,"&Real Mode",1,0
return
//=====
sub SendCommand(C,P,&s)
    m = ""
    serialout C,P
    serbytesin 5,s,x
    if x < 5 then m= "Comms Error"
    xystring 500,20,m,spaces(30)
return (x == 5)
//=====
sub PlotPlane(gX,gY,gZ,H)
    thX = atan2(gX,gZ)-pi(.5) //x-tilt
    thY = atan2(gY,gZ)-Pi(.5) //y-tilt
    mcopy Plane,K //refresh body coordinates array
geRotateA K,-thY,1 //rotate around x-axis i.e. Roll
                //negative to orientate for Right-Handed Standard
    geRotateA K,thX,2 //rotate around y-axis i.e. Pitch
//geRotateA K,dctor(-H),3 //rotate around z-axis i.e. Heading
                //negative to orientate for Right-Handed Standard
    ge3dto2da K,Eye //calculated screen coordinates
    for i=1 to maxdim(K)-1 //plot the plane
        line K[i-1,3],K[i-1,4],K[i,3],K[i,4]

```

```
next
call DisplayCompass(!GetCheckBox("Mode"),H,600,200)
call DisplayAttitude(thX,-thY,200,200)
flip
clearscr
return
```

8.5 Using the QTI Infrared Line Sensors

[Cut Out]

8.5.1 The New Firmware

[Cut Out]

8.5.2 Testing the QTI

[Cut Out]

8.6 Adding Sound

[Cut Out]

8.6.1 The New Firmware

[Cut Out]

8.6.2 Testing the Speaker

[Cut Out]

8.6.3 An Exercise

[Cut Out]

8.7 The Final System Firmware

Our firmware is now complete. We will change the hardware to be wireless in the next chapter, but that requires no firmware changes and just a minor hardware change (also see Chapter 11). In the zip file with all the source code there is a folder called `Final_Protocol`. In it you will find all the files needed to compile and upload the firmware to the Propeller's EEPROM (F11). Now that the firmware is complete you should write it to the EEPROM so you can use the PPDB with any programs you develop. You would not have to change the firmware again – unless you want to add new hardware. We have renamed all the four objects to `Protocol_XXXX.Spin` where the `XXXX` is `Main`, `Reader`, `Others`, or `Motors`. The *top-level* object is `Protocol_Main.Spin`.

In the subfolder are also copies of all the third-party objects that we used. You will also find all the RB programs that work with the new firmware. They are mostly copies of the programs we have seen so far placed there for your

convenience. There is ***one new program*** called `Complete_System_Tester.Bas` (See Figure 8.15 below). This program incorporates all the generic and versatile subroutines from all the other programs that we developed during the progression through Chapters 7 and 8. The program performs a dazzling number of tasks all going on simultaneously and yet in real time. It is a testament to the power of our protocol, the Propeller Chip and RobotBASIC.

See Appendix B for the following:

- A tree of objects hierarchy for the Final Protocol
- A tree of objects hierarchy for the Extended Protocol (from Chapter 11)
- Figure B.1: The System's Conceptual Schematic
- Figure B.2: Propeller Pin Utilization
- Figure B.3: Hardware Connections Schematics.
- Figure B.4: Picture of the final PPDB setup
- Table B.1: List of Protocol Command Codes for the Final Protocol
- Table B.2: System Parameters Value Mapping when using the Final Protocol
- Table B.3: List of Extended Protocol Command Codes (as in Chapter 11), in addition to Table B.1
- Figure B.5: Protocol State Diagrams.

Here is a list of all programs that work with the Final or Extended Protocols

- `Complete_System_Tester.Bas` (a new program for the final firmware)
- `Compass_Tester.Bas`
- `Compass_Animation.Bas`
- `EEPROM_Tester.Bas`
- `H48C_Plane.Bas`
- `H48C_Tester.Bas`
- `Individual_Motors.Bas`
- `Piano_2.Bas`
- `Ping_03.Bas`
- `Pots_03.Bas`
- `Program_12_ReallySimple.Bas`
- `QTI_Tester.Bas`
- `RobotMoves_12.Bas`
- `Servomotor_12.Bas`
- `Speaker_Tester_2.Bas`
- `Turret_Tester.Bas`



`Complete_System_Tester.Bas` (Figure 8.15) is a variation on `Program_12_Advanced.Bas` to incorporate all the new hardware from Chapter 8. It is an interesting and comprehensive GUI system for testing the firmware and hardware. Also, it has numerous reusable and versatile subroutines that you may want to use or emulate in your software. Notice how the Compass and AI instruments are now smaller and repositioned. The same include file was used to draw them as before. You may want to migrate some of the subroutines in the program to another include file so that you can use them in your own programs. **Read_48HC()** for example and **mmDistance()** have already been moved to the `Instruments.Bas` include file.

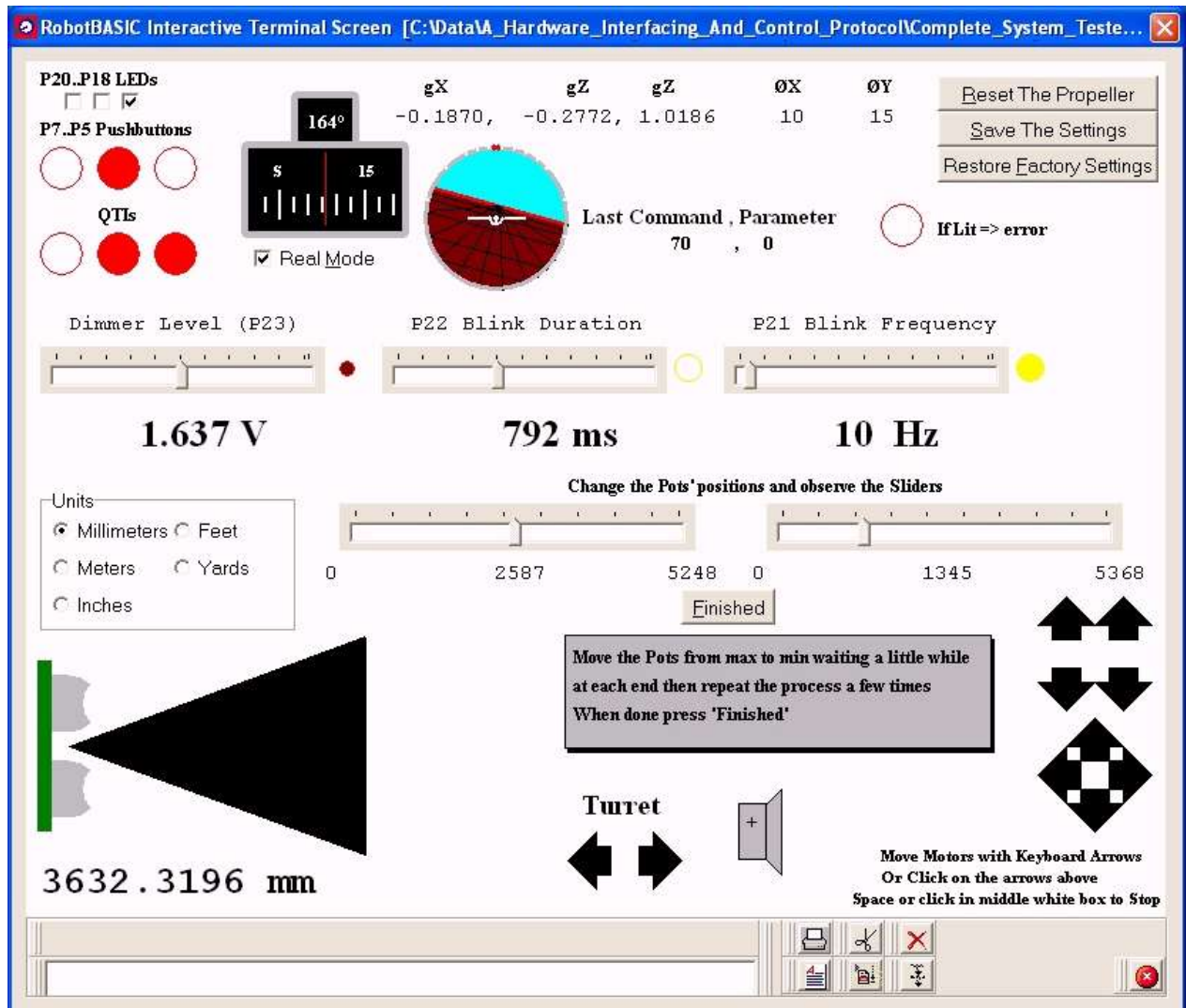


Figure 8.15: Complete_System_Tester.Bas Screenshot. Pots calibration is in progress.

8.8 Summary

In this chapter we:

- ❑ Added an HMC6352 compass unit.
- ❑ Learnt about inter-cog interactions.
- ❑ Learnt about a procedural strategy for adding other hardware in a systematic manner.
- ❑ Added a turret for the Ping))).
- ❑ Added the ability to control the motors individually.
- ❑ Added the mechanisms to save/retrieve the system parameters to/from EEPROM.
- ❑ Added an H48C accelerometer unit.
- ❑ Developed authentic looking simulated Compass and Attitude Indicator instruments.
- ❑ Utilized the **#Include** command in RB to include a library of routines into our program so that we won't have to rewrite these useful subroutines.
- ❑ Utilized RB's 3D graphics engine.
- ❑ Added 3 QTIs.
- ❑ Added a Piezoelectric Speaker with the ability to play RTTTL like musical tunes.

Severing the Tether

Our system so far has a direct connection with a serial wire between the hardware and the PC. This is an acceptable option if the hardware and environment allow for it. For instance you can have a Notebook on top of a robot. However, more often, we would prefer the hardware to be remote from the control system.

Imagine you have multiple hardware systems distributed throughout an area. You can have one PC with multiple control programs each controlling one of the distributed systems. The user can then interact with each system and have an overall picture of the entire area and can interact with the various remote hardware all being controlled simultaneously and in parallel. Alternatively you can have one program controlling all the remote systems where each is a sub-process of a multifaceted and distributed overall process.

The strategies and methodologies we developed into our system facilitate the implementation of a remote control system effortlessly. There are two ways to implement remote control:

- Using a Radio Frequency transceiver with or without an inbuilt advanced protocol such as Bluetooth or XBee.
- Using the Internet or Local Area Network with a TCP or UDP protocol using a wired Ethernet connection or wireless Wi-Fi.

9.1 Wireless With RF, Bluetooth or XBee

The system can be made into a wireless one with surprising ease. In fact, as far as programming the firmware and software is concerned we would need almost no changes. Even the hardware will hardly need any change. If we ignore the debugging connection to the PST (only needed during the development stage) then the system we have now is as depicted in Figure 9.1(A). Of course, you can also hook up the PST along with the wireless for further debugging when required.

All we have to do to achieve a remote system as depicted in Figure 9.1(B) is:

- ❑ On the hardware side replace the Propeller Plug with a suitable wireless transceiver.
- ❑ On the PC add a compatible wireless transceiver.

The best system for our purposes is the XBee transceiver. Another very good alternative is to use the Bluetooth system. A third choice is to use normal Radio Frequency digital transceivers, but it is not simple to make this option work well.

Both the Bluetooth and XBee transceivers can be configured to act in a mode called “transparent” mode. This mode is basically a wireless replacement for the serial cable, and in both systems we can use them in exactly the same way we have been using the Propeller Plug with a USB cable; except there is no cable. Just as the PP was plugged to our hardware using the TX and RX pins so would be the Bluetooth or XBee device.

The only difference is that on the PC end we need a BT or XBee device too. However, in both cases the device plugs in the USB port giving us practically the same logical setup as we had with the PP. The protocol used by these systems provides us with a seamless and error-resilient link to make it appear as if they were connected with a wire.

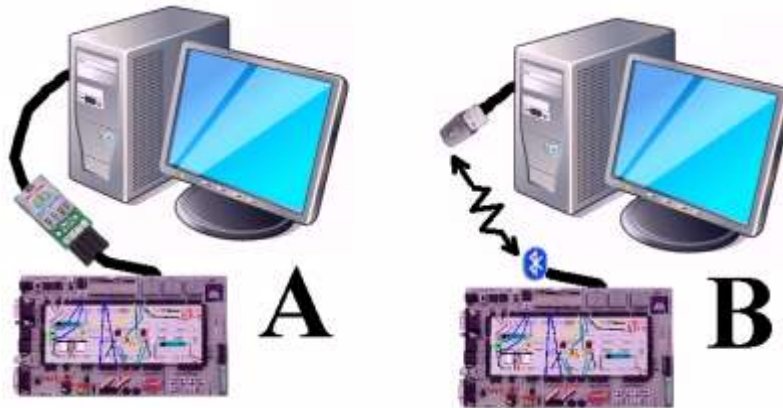


Figure 9.1: Wired and Wireless Systems

9.1.1 XBee

[Cut Out]

The XBee Advantage

[Cut Out]

The XBee Disadvantage

[Cut Out]

9.1.2 Bluetooth

[Cut Out]

The Bluetooth Advantage

[Cut Out]

The Bluetooth Disadvantage

[Cut Out]

9.1.3 Pure Radio Frequency

[Cut Out]

The RF Advantage

[Cut Out]

The RF Disadvantage

[Cut Out]

9.1.4 Summary of the Wireless Options

Regardless of which system you use, the end result is that:

- ❑ The hardware will now use the RX and TX pins on the transceiver hardware in place of the PropPlug.
- ❑ The baud rate may have to be changed on the firmware and software.
- ❑ If the transceiver requires a little more software control than was used with the PropPlug, like for instance an initialization sequence or some handshaking then the firmware will have to achieve that.
- ❑ The PC side will have to undergo some onetime setup procedure to install the dongle or other hardware required to actuate the PC-side transceiver and make it available as a com port.
- ❑ The software will have to use the new com port in place of the one that was used for the PP. Also the baud rate may have to be changed according to the new hardware.

All of the above is not complicated at all. More importantly, the protocol, firmware, software and system we have developed would not have to undergo any major alteration.

Converting our system to a wireless one imparts it with versatility and utility that makes it a powerful setup for many projects. On a robot it becomes a vastly viable robot that can be controlled with an easy to program and yet AI capable language (e.g. RB) enabling the robot to perform more interesting projects. But what is even more advantageous is that when we need to reprogram the robot to do different things we just change the software, not the robot's firmware. This makes it possible to use the robot in a less burdensome manner. Also (as you will see in Chapter 10) with RB's inbuilt simulator and simulator protocol you can try out the algorithms safely and effectively on the simulator, then with the addition of one line of code start controlling the real robot.

9.2 Wi-Fi & Internet

Another way we can provide remote control to our system is by using the TCP or UDP networking protocols. Moreover, this option provides limitless range. With the wireless system we are limited to line of sight and range of the device. With the TCP/UDP option the controller PC can be halfway around the world away from the hardware.

This is not an idea to take lightly. As you will see shortly, with *surprisingly few lines of code* we can give our system the ability to run software on one PC controlling hardware connected to another PC either directly or wirelessly (as in Section 9.1). The hardware-side PC can be within the same LAN or even in a LAN across the Internet. Moreover, either side may be connected directly to the LAN router or wirelessly using Wi-Fi.

Figure 9.7 illustrates the various possible combinations for effecting this amazingly versatile methodology. Our system as we have designed it so far will not require any change whatsoever in the firmware side. All the changes are to be implemented in the RobotBASIC software, and besides, these changes are completely simple thanks to the power RB provides.

9.2.1 TCP and UDP Networking Protocols

TCP is a networking protocol using the Client/Server model. UDP on the other hand utilizes the peer-to-peer model. UDP is an adequate protocol but is not as robust as TCP. It is sufficient if you are going to limit your connectivity to within a LAN but not across the Internet. Using a UDP connection over the Internet requires an additional protocol layers to assure communications reliability. A TCP connection already has all the reliability assurance needed.

Our system functions just as well with UDP or TCP due to our protocol. It provides an additional layer for data packet synchronization, which is one of the shortcomings of UDP. A UDP packet sent earlier may arrive at the destination later than one sent later. Our protocol provides a solution to the problem by sending out a few bytes and then waiting for a reply before sending the next packet. The outcome is that our protocol works equally with UDP or TCP.

We will not use UDP in this book. If you wish to do so, you can find out all about UDP and TCP in a document from our web site called [RobotBASIC_Networking.Pdf](#)⁵⁷. The document has many projects and intricate details explaining how to implement control across the Internet and LAN using TCP and UDP. You should read it along with this chapter to assure maximum proficiency in all aspects of RB's networking facilities.

We will not go into the details of how to setup **Port-Forwarding** to enable across the Internet communication through firewalls and routers. This is all explained in the aforementioned PDF. Here we will assume that you have read it and are able to use RB to communicate between two computers either on the same LAN or across the Internet. We will only deal with how to use RB's Internet commands and functions to add to our system the ability to control hardware from a PC other than the one directly or wirelessly connected to the hardware.

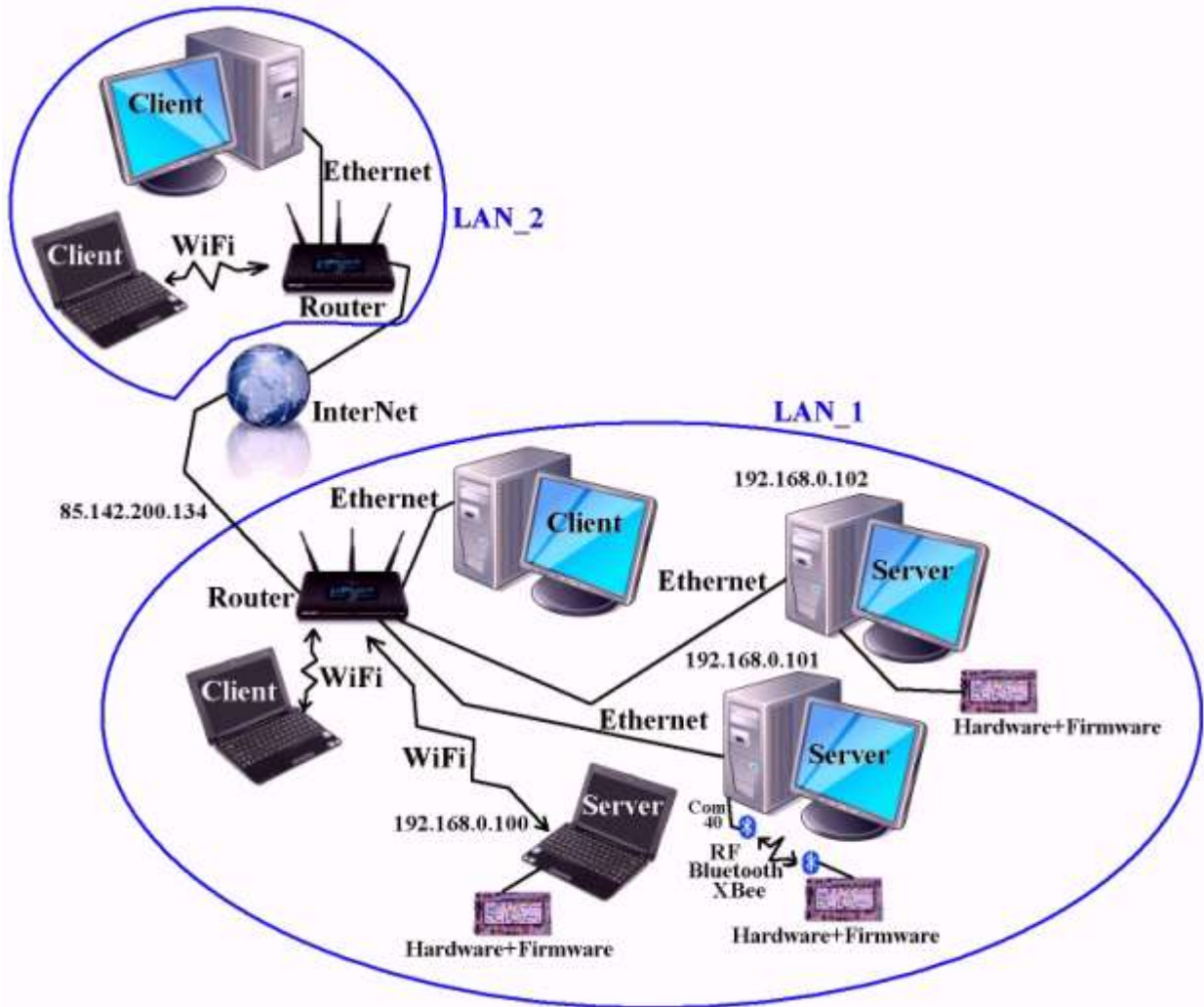


Figure 9.7: Various configurations for LAN or Internet Remote Control. Shown is the TCP model but the same topology would also apply to UDP.

9.2.2 The Topology

At the top of Figure 9.8 is a depiction of the system we have been using thus far in the book. The laptop controls the hardware + firmware through Link-A using an RB program (software). In Section 9.1 we showed how Link-A can be made wireless. In this section we want to transform our setup to be as depicted in the lower section of Figure 9.8. We will use TCP, so one computer has to be a *client* and the other has to be a *server*. The software application will be migrated over to a remote PC which is the *client*. Some modification of the software will be required to make it run over the TCP link in place of the serial link as before.

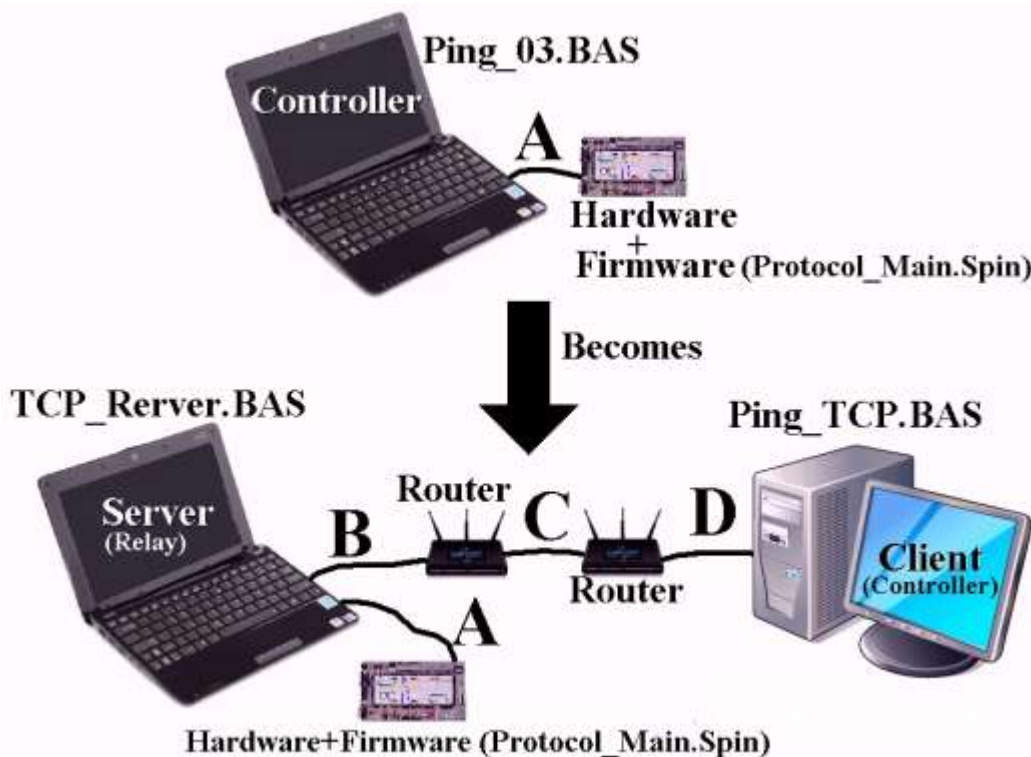


Figure 9.8: Converting our setup to work over a LAN or the Internet.

The PC controlling the hardware is now the *server*. It no longer runs the hardware control software. Instead it runs specialized software acting as a *relay* between the *client* PC and the firmware on the hardware. This specialized software is called TCP_Server.Bas and is the same regardless of the controller software running on the client and regardless of the hardware being controlled locally by the server. The server receives the command and parameter bytes from the *modified control software* running on the client PC and relays them to the *unmodified firmware* on the hardware. The server also receives the 5 bytes from the firmware and relays them to the client PC.

Notice that in the setups of Figure 9.8 there are the links:

- **A:** can be either a direct wired link (e.g. PP) or a wireless link (e.g. XBee)
- **B:** can be a directly wired Ethernet link to the router or a Wi-Fi wireless link.
- **C:** can be a non-link in that the two routers are actually the same router, or it can be a link across the Internet between two separate routers.
- **D:** can be a directly wired Ethernet link to the router or a Wi-Fi wireless link.

Each link can be one of two possibilities. There are 4 links. So there are 16 possible combinations of the setup.

As you can see we need to modify the controller software application for each application. However, the TCP_Server.BAS software is the same for all. We will develop this new software and you can use it to act as the relay software regardless of the firmware or the controller application as long as they both follow our protocol.

The Software Side

Remember that in our protocol the RB software sends two bytes to the firmware and then expects 5 bytes back. In the setup at the top of Figure 9.8 the RB software sends its two bytes on the USB port that is either directly connected to the hardware using the PropPlug or indirectly using the XBee. When it looks for the bytes to be received it also looks on the same com port.

When we separate the control program to a PC across the LAN or Internet there will be no com port to use for sending/receiving. Instead there will be an Ethernet link from the PC to a router. The link may be a direct Ethernet wired link or it can be a wireless Wi-Fi link. The final outcome is that the PC is linked to a router and is able to send TCP packets.

When the program wants to send its two bytes it will have to send them over the Ethernet link; likewise for receiving the 5 bytes. So in our RB control program such as Ping_03.Bas we will have to change the sending/receiving commands from serial commands to networking (TCP) commands.

The Hardware Side

As far as the hardware + firmware we used a connection to the PP or XBee to effect the communication with the PC running the RB control program. But now we cannot use that since the control program is now across the LAN/Internet.

What we need is a method for our firmware to receive/send across the LAN/Internet. There are many devices that provide an easy way to implement a TCP stack on a microcontroller such as the Propeller. One such device is the [Spinneret Web Server](#)⁵⁸. However, this device does not provide Wi-Fi and the hardware would have to be wired directly to a router.

There is a very effective and amazingly simple to implement method for providing our hardware with a link to the LAN/Internet without any change to the hardware or firmware. *No change whatsoever*. Neither the hardware nor the firmware would have to be changed.

The method is to use a *relay* PC. The hardware communicates with the relay PC as if it were the controller PC. As far as the firmware is concerned it does not care at all what the controller PC does. As long as it receives the two bytes and is able to send the 5 bytes back, the firmware does not care what the controller PC does with the 5 bytes and neither does it care how the controller PC gets the two bytes the firmware receives. As far as the firmware is concerned the PC sending to it and receiving from it is the controller PC and it does not care what it does to effect the communications protocol. No changes at all are needed to the firmware and hardware to communicate with the *relay* PC whether it is directly or wirelessly linked.

This relay PC receives from the control program over the TCP link the two bytes and sends them to the firmware across the serial link. It then receives from the firmware the 5 bytes and sends them across the LAN/Internet to the RB program. This relay PC can be directly wired to the router or using a Wi-Fi link.

The relay PC runs a special RB program (TCP_Server.Bas) that effects all the serial communications as well as the TCP communications.

The Client and the Server

In our arrangement using the TCP protocol (bottom of Figure 9.8), the PC running the control software (e.g. Complete_System_Tester_TCP.Bas) will be the *Client* side. The PC that acts as the relay PC and connects to the hardware is the *Server* side.

The server PC will have to have a specialized program that sends/receives bytes to and from the client PC on the TCP link and then sends/receives those very same bytes to and from the hardware across the serial link (TCP_Server.BAS).

The control software will have to be changed so that instead of sending/receiving on a serial port it will do so on a TCP port.

The Required Modifications

The required changes to the Control Application Software (e.g. Ping_03.Bas) are:

- ☐ Change the send/receive commands to do so using TCP commands instead of serial commands.
- ☐ Add the relay program to run on the relay (server) PC

The required changes to the firmware are:

- ☐ No changes are needed

The required changes to the Hardware:

- ☐ No changes are needed

The required changes to the link between the hardware and the PC:

- ☐ Since the relay PC is acting as a surrogate controller, no changes are required to the connection hardware.

Requirements to add the Networking ability:

- ☐ The client PC has to be linked to a **router** with a Wi-Fi or direct Ethernet link.
- ☐ The server PC has to be connected to a router with a Wi-Fi or direct Ethernet link.
- ☐ Two routers connected to the Internet if you want to effect across the Internet control. If you only need LAN control then only one router is needed and it does not need to be connected to the Internet.

An Example Topology

To illustrate the power of this option imagine you have a robot with the hardware and firmware similar to that developed in Chapters 7 and 8. You have done all the testing you need. You then made it wireless as we did in Section 9.1 then you tested again and all is working.

Now you move the control program to another PC and do the required changes to it as we will describe later. This PC is now called the Client PC. You also install on the old PC the new server software that will be described later. This PC is now called the Server PC.

After accomplishing the above you run the server program which will start listening on the TCP for incoming bytes. Then you run the control program (on the Client PC) which will start sending and receiving to and from the relay program (Server). The server will also send to and receive from the robot over the wireless serial link.

The outcome is that we have a robot that can be controlled from a PC either across the Internet half way around the world or in a different building. If the robot is big enough you can dispense with the RF link and have the Server PC onboard the robot with a Wi-Fi link to the router.

A Testing Topology

For the sake of testing and experimenting it would be inconvenient to keep moving between two PCs, also you may not have two PCs available. Can you still do all the required testing? Indeed you can. This is yet another testament to RB's versatility and power. Just as you can run two programs on the same PC simultaneously, you can run two versions of RB on the same PC, simultaneously. You can run two instances (or as many as you want) of RB's IDE on the same PC. You will have to do some rearrangement and organization of the windows to see all the output simultaneously but that is what you do anyway with other software.



Now you can dispense with the second PC. You can run the RB client software as well as the RB server software on the same PC. This way the Server and the Client PCs are the same PC running two instances of RB and each instance has the required RB program.

When the client software runs it will still be sending the two bytes through the router, but now the router will route the bytes back to the same PC and the server software will receive them just as if the two programs were running on different PCs. The server will then send the two bytes to the firmware on the serial com port.

When the hardware sends the 5 bytes they will be received by the server software on the serial com port. The server will then send the 5 bytes to the router. But now the router will send them back to the same PC and the client software will receive them, completing the protocol just as we need.



The fact that the Client and Server software are running on the same PC makes no difference as far as the routing of the data through the router. The outcome is now we can view the actions of the Server and Client on the same screen on the same PC and we do not have to keep moving from one PC to another.



Once development and testing are over, you ought to procure another PC and run the system on two PCs across the LAN. After that you ought to arrange with a colleague to run the Client software on his PC on his LAN across the Internet to a LAN with your PC acting as the Server and linked to the firmware + hardware.



To accomplish across the Internet control you will need to know how to setup your router to carry out **port forwarding** to allow for Firewalls and IP address masking and so forth. All this is explained in detail in the document [RobotBASIC_Networking.Pdf](#)⁵⁷ which is requisite reading to appreciate the programming in the next section.

9.2.3 IP Address and Port

You are expected to have read [RobotBASIC_Networking.Pdf](#)⁵⁷ and therefore very little detail will be given about networking with RobotBASIC. We urge you to read this document since the information in it is vital for the proper understanding of what is to come.

An important thing to realize with networking is that for a client to be able to link to a server it has to know two things:

1. The Server PC's IP address.
2. The Port the Server Program is using.

The above data are straight forward if the client PC and the server PC are both on the same LAN (i.e. sharing the same router). However, if the two PCs are on separate LANs (i.e. different routers) then the above two pieces of information are much more complicated to obtain and arrange. How to do that is explained in the reference PDF document above.



For the purpose of our programs it is immaterial how you obtain the IP/Port combination as long as you have the correct values. As you will see in the server program below, when it starts it will display the Local IP address and port number. But keep in mind that the Local IP address is the one to be used in the client program **only** when the client and server are local to each other (using the same router).



Remember that you have to do more work to obtain the proper IP Address if the PCs are on different LANs and also that you have to set the router on the server side to perform **port forwarding** to forward the communication to the server.

We will not concern ourselves with the details of across the Internet communications; we will just use LAN addresses. The programs are not affected at all. All you have to do to make the same programs work with PCs on different LANS is to replace the IP address on the client software with the correct one instead of the one reported by the server program. All this is expounded upon in fine detail in the above PDF document.

9.2.4 The Server Program

The server program we develop here implements the strategies discussed in Section 9.2.2. The program is not GUI nor does it need to be so. In fact making it GUI would be counterproductive. The program is supposed to be a **relay** between the client PC and the firmware. Therefore we want it to be as fast as possible. This means that the program should in fact do the minimal possible processing to minimize the time between receiving a byte from the client and sending it out to the firmware and vice versa. Nevertheless, all you have to do to make it GUI is add the GUI programming elements; the functionality requirements for it to act as a relay between the client and the firmware remain the same regardless of the user interface.

The program establishes a serial link to the firmware then starts listening over the TCP port. It also displays the Port number and IP address it is using so that you can set them on the client side. Also it clears the serial as well as the TCP buffers just in case there were leftover data from previous operations.

The program then repeats the following tasks forever:

- ❑ Display the status of the TCP link for diagnosis purposes.
- ❑ Checks if there are any bytes in the TCP receive buffer. If there are any it reads them and sends them out through the serial port.
- ❑ Checks the serial receive buffer and if there are any bytes it reads them and sends them out through the TCP port.



This simple short program running on the server is all that is needed to act as a relay between the Propeller hardware + firmware and the control application program running on the client (remote) PC. The program is the same regardless of the software application on the client and regardless of the hardware + firmware as long as they obey the protocol.



Due to the design of TCP_Server.Bas it can act as a server (relay) for any protocol that you are likely to devise. See Chapter 11 for how the server still works with the quite different sending and receiving that occurs on some of the extended commands in the extended protocol of Chapter 11.

TCP_Server.Bas

```
//TCP_Server.Bas
SerialPort = 16           //change this per your system
SerialBaud = br115200    //and this
TCPS_Port = 50000        //change this if you need to do so
xystring 10,10,"IP:",TCP_LocalIP();"Port:",TCPS_Port //show Local IP address
xystring 10,50, "Status      :"
Main:
  setcommport SerialPort,SerialBaud //connect to the serial port
  tcps_serve(TCPS_Port)             //start the Server
  TCPS_Read() \ clearserbuffer      //clear both buffers
  while true
    xystring 100,50,TCPS_Status(),spaces(30) //display status
    if TCPS_BuffCount() then serout TCPS_Read() //if TCP bytes send to serial
    CheckSerBuffer n
    if n then Serin s \ TCPS_Send(s) //if serial bytes send them to TCP
  wend
end
```

9.2.5 The Client Program

To easily understand the changes required in a program that will run through the LAN/Internet and communicate with the firmware using our protocol, we will develop a simple program. Thus we can see the changes without the burden of complexity. We will develop a program that will turn the motors forward for one step in a continuous loop. We will first see what it looks like in the normal format using the serial port to perform normally as we have been doing all along. The program is later converted to function over the TCP. This way you can see what is required. Later we will modify a program we developed in a previous chapter to show that changing a complicated program is not much more work. The same steps taken to convert the simple program apply regardless of the complexity of the software application.

The Serial Link Program

TCP_Tester_Normal.Bas will cause the wheels to turn repeatedly for one step. Notice the **SendCommand()** subroutine. This subroutine performs the tasks of sending out the two bytes and then receiving the 5 bytes. It does so using the serial port as normal. Run the program as we have been doing all along using Protocol_Main.Spin for the firmware.

TCP_Tester_SerialLink.Bas

```
//TCP_Tester_SerialLink.Bas
//works with Protocol_Main.Spin
Port = 16 //change this as per your system
Main:
  setcommport Port,br115200
  while true
    call SendCommand(6,1) //move the motors forward one step
  wend
End
//=====
sub SendCommand(C,P,&s,&n)
  Serialout C,P //send the two bytes
  serbytesin 5,s,n //receive the 5 bytes with timeout
return (n == 5) //return true or false depending on number of arrived bytes
```

The LAN Program

Before we run TCP_Tester_LAN.Bas let's discuss the code. Study the listing below and try to figure out what it accomplishes. It ought to be self-documenting since RB's syntax is very intuitive.

Compare it with TCP_Tester_Normal.Bas. The LAN program has a few more constants defined at the top. The **Main** section is almost the same except that it initializes a connection to the Server instead of the com port. To be more thorough here we ought to test for success and issue a message if the connection is not successful. We left this out for simplicity.

The primary difference between the two programs is the **TCP_SendCommand()** subroutine. The new subroutine will do the communications over the TCP. The first thing is to clear the TCP receive buffer to discard of any bytes from a previous communication. Then the command code and parameter bytes are sent over the TCP link to the server.

The variable **s** is cleared and the receive count is zeroed, then the current serial communications timeout period is obtained and assigned to the variable **T** with an additional 150 ms to allow for the time to transfer the bytes from the server to the client. This value in **T** will be used to monitor a timeout in case the 5 bytes never arrive. Remember with the serial link we checked for the 5 bytes with a timeout. This mechanism will do the same for the TCP link and will keep to the same timeout duration as the serial link but with an additional 150 ms to allow for the extra relaying.

We then start a timer to act as a stopwatch and enter the *polling* loop where the buffer is checked for the correct number of bytes (5) to arrive. When they arrive they are extracted and put in the by-reference variable **s** to pass them

back to the caller code. Also the number of bytes is stored into the other by-reference parameter **n** again to pass it back to the caller code. The subroutine returns true or false depending on if 5 bytes were received or not.

This subroutine is really all we need for other programs to convert them for LAN use. We need to change all calls to the **SendCommand()** subroutine to the **TCP_SendCommand()** subroutine. Of course we also need to connect to the correct server and remove any serial port related code. In Section 9.2.7 we will see how to do all this for a more complex program we developed in a previous chapter.



We will add the subroutine **TCP_SendCommand()** to the Instruments.Bas include file from Chapter 8.1.4 so that many other programs can use it.

TCP_Tester_LAN.Bas

```
//TCP_Tester_LAN.Bas
//works through the LAN with the Protocol_Main.Spin
//Over the LAN or Internet
//Port = 16 //change this as per your system
TCPS_IP = "192.168.0.100" //change this to the correct server's IP address
TCPS_Port = 50000 //change this if you change the one in the server
Main:
    //setcommpport Port,br115200
    TCPC_Connect(TCPS_IP,TCPS_Port) //connect to the server
    while true
        //call SendCommand(6,1) //move the motors forward one step
        call TCP_SendCommand(6,1) //move the motors forward one step
    wend
End
//=====
sub SendCommand(C,P,&s,&n)
    Serialout C,P //send the two bytes
    serbytesin 5,s,n
return (n == 5) //return true or false depending on number of arrived bytes
//=====
sub TCP_SendCommand(C,P,&s,&n)
    TCPC_Read() //read the buffer to clear it
    TCPC_Send(char(C)+char(P)) //send the two bytes
    s = "" \ n=0
    GetTimeOut T \ T += 150 //get the actual timeout limit and add
                            //150 ms to allow for processing
    t=timer() //start a timer
    repeat //wait for the 5 bytes to arrive with timeout
        if TCPC_BuffCount() != 5 then continue //if not arrive keep waiting
        s = TCPC_Read() \ n=5 //if arrived get them from the buffer
        break //since bytes have arrived then end the loop
    until timer()-t > T //repeat until timeout
return (n == 5) //return true or false depending on number of arrived bytes
```

9.2.6 Running the LAN System

To run TCP_Tester_LAN.BAS do the following:

- ❑ Make sure Protocol_Main.Spin is compiled and uploaded to the hardware using F11. You can use F10 but we will be resetting the Propeller for a robustness test later. Besides, the firmware is now stable and you will not need to do any more changes, so it should be loaded in the EEPROM for all future work.
- ❑ Make sure that the hardware is connected to the Server PC either using the wired (PP) method or the wireless method.
- ❑ Make sure TCP_Server.Bas is running on the server. When it runs note its IP address and Port number as displayed. You need these to set them in TCP_Tester_LAN.Bas before running it.
- ❑ If you are going to use another PC then move to that one. If you are going to use the same PC as the client and server then start another instance of RB's IDE.
- ❑ Load TCP_Tester_LAN.Bas and edit the **TCPS_IP** and **TCPS_Port** variables to be the correct values as displayed by the Server program.
- ❑ Run the program.
- ❑ You should now see the motors on the hardware moving.

You may wish to swap over to the server's window and watch how the displayed messages reflect the status of the link. Also do one more thing that will impress you (we hope). Switch the PPDB off then wait a few seconds then switch it back on and note how control is resumed without having to reset either the server or the client programs.



That is all there is to it. You may not realize it but you have just accomplished an amazing feat. With just a few lines of code you managed to control hardware across a LAN (and wirelessly, if you did the wireless connection). This is not a small accomplishment, and it was all made possible because of a well-designed firmware and software system with the help of the great Propeller Chip and RobotBASIC's power and ease.

9.2.7 Converting a More Complex Program

To illustrate converting a more complex program we will do so for Ping_03.Bas. The new version is listed below and is called Ping_TCP.Bas. All the new and modified lines are in highlighted code. We have also converted the following files; they are not listed but you will find them in the downloadable Zip file. Study them to see how they are converted in a very similar manner to Ping_TCP.Bas. The subroutine **TCP_SendCommand()** is what makes it all very simple (it is now in the Instruments.Bas include file).

Command_Turnaround_TCP.Bas
 Compass_Animation_TCP.Bas
 Complete_System_Tester_TCP.Bas.
 H48C_Plane_TCP.Bas
 Piano_TCP.Bas
 Ping_TCP.Bas
 Pots_TCP.Bas
 Servomotrs_TCP.Bas
 Turret_Radar_TCP.Bas



You may also be especially interested in Complete_System_Tester_TCP.Bas. When this program runs over the LAN/Internet you will be able to appreciate the power of combining RobotBASIC with the Propeller Chip, but above all the utility of a good well designed protocol.



The program below uses the subroutine **TCP_SendCommand()** we saw in Section 9.2.5 which is now residing in the include file Instruments.Bas.

Ping_TCP.BAS

```

//Ping_TCP.Bas
//works with Protocol_Main.Spin
//but over the LAN or internet
Port = 0 //change this as per your system
#include "..\Utilities\IncludeFiles\Instruments.Bas"
TCPS_IP = "192.168.0.100" \ TCPS_Port = 50000
tcpc_Connect(TCPS_IP,TCPS_Port)
Main:
  gosub Initialization
  while true
    call ReadPing(t)           //get the time value from the propeller
    if ! ReadPing_Result then continue //if failed loop back
    call mmDistance(,t,D)      //convert to mm
    n=getrbgroup("Units")-1    //get desired units
    x=Format(D*factors[n],"0.0####") //convert to units and format
    call DrawSignal(Px,Py,t,x+unitn[n]) //draw beam and display value in
units
  wend
end
//-----
Initialization:
  setcommport Port,br115200
  data units;"Millimeters","Meters","Inches","Feet","Yards"
  data unitn; " mm"," m"," in"," ft"," Yrds"
  data factors;1.,.001,Convert(.001,cc_MTOIN)
  data factors;Convert(.001,cc_MTOIN)/12
  data factors;Convert(Convert(.001,cc_MTOIN)/12,cc_FTOYRD)
  AddRBGroup "Units",10,10,200,100,2,mToString(units)
  SetRBGroup "Units",1
  Px = 10 \ Py = 300
  call DrawPing(Px,Py)
  savescr
  Flip on
Return
//-----
sub ReadPing(&P)
  P = -1
  //serialout 192,0           //signal the Propeller
  //serbytesin 5,s,n          //receive the value bytes
  call TCP SendCommand(192,0,s,n) //this routine is in the include file
  if n < 5 then return false //if error just return false
  P = (getstrbyte(s,4)<<8)+getstrbyte(s,5) //reconstitute the value
Return true
//-----
sub DrawPing(x,y) //draw picture of Ping device
  s = 30
  for i=-1 to 1 step 2
    line x,y+s*i,x+20,y+s*i,40,gray
    circlewh x+30,y-20+s*i,40,40,white,white
  next
  rectanglewh x-20,y-2*s,25,s*4,white,white
  rectanglewh x-2,y-s*2,10,s*4,green,green
Return
//-----

```

```
sub DrawSignal(x,y,d,str)
  d *= 500.0/22000      //scale value to screen coordinates
  restorescr
  xx1 = x+20+CartX(d,dtor(20))
  yy1 = y+CartY(d,dtor(20))
  Line x+20,y,xx1,yy1
  xx2 = x+20+CartX(d,dtor(-20))
  yy2 = y+CartY(d,dtor(-20))
  Line x+20,y,xx2,yy2
  Line xx1,yy1,xx2,yy2
  if d > 3 then floodfill x+20+d/2,y,black
  xytext x,y+80,str,,20,fs_bold
  flip
return
//-----
```

9.3 Summary

In this chapter we:

- ❑ Examined various ways to make the hardware remote from the controller PC.
- ❑ Used XBee transceivers to make the link wireless.
- ❑ Considered the Easy Bluetooth and D-Link dongle as an alternative method for wireless links.
- ❑ Considered pure RF modules but saw that they are not as convenient as the above two alternatives.
- ❑ Examined various configurations to implement LAN or Internet connection between the hardware and the controller PC.
- ❑ Developed a Server program to act as the server/relay between the hardware and controller PC.
- ❑ Converted previously developed control programs to work over the LAN/WAN.

RobotBASIC's Inbuilt Protocol

The firmware as implemented by Protocol_Main.Spin represents a complex and quite interesting system, with hardware that typifies devices likely to be encountered in applications such as a robot. In fact the PPDB as it is now (Figure B.4 in Appendix B) can be mounted on the chassis of a robot (e.g. [Boe-Bot](#)⁵⁹) and we would be able to control it quite adequately. Let's have a look at what we have:

- Ultrasound Ranger on a Turret.
- Compass.
- 3xQTI line or drop-off sensors.
- Accelerometer.
- Pushbutton that can be replaced by a [Bumper Switch](#)³⁰ (See Figure 10.1).
- 2xPushbuttons that can be replaced with [Infrared Proximity Sensors](#)³¹ (see Figure 10.2).
- 2x Continuous Rotation Servomotors.
- Piezoelectric Speaker.
- 2xPotentiometers that can be replaced with [Photo Resistors](#)⁶⁰ with an appropriate value capacitor, or even better with [Photo Diodes](#)⁶¹. Although this requires a slight change in the circuit, it is still an RC circuit just with a slightly different setup. Another possible device is a Thermistor for heat source detection instead of light.

This collection of hardware enables us to drive the robot and determine its heading while avoiding objects and even following a line. One bumper would not be sufficient but you can very easily add more. We can even control an airplane or a walking robot. You may wonder how can this be accomplished when there is no specific code in the *firmware* that implements object avoidance or heading tracking or line following. There is no code that is in anyway specific to a robot. How can we make the system achieve robot specific tasks when there is nothing in the firmware that would carry out any such tasks? The answer, as you have seen throughout the book, is the *software*. What gives the system versatility and ability is the partnership between:

- Hardware
- Firmware
- Software

Our system is very much like a PC, which is a collection of hardware with firmware (OS and BIOS) and software applications. All we need to make our system *multifaceted* is to add different software, as you have already seen in previous chapters. In this chapter, we will use the same system in an interesting and novel manner.



Figure 10.1: Snap-Action Bumper Switch



Figure 10.2: A very handy Infrared Proximity Sensor.

Normally, while programming new features on a robot, we

- ❑ Place the robot on a pedestal to prevent it from moving when the wheels turn.
- ❑ Connect it to the programming IDE on the PC (usually).
- ❑ Download a program we think is a working one.
- ❑ Unplug it.
- ❑ Take it off the pedestal and place it in the environment in which it is supposed to function (assuming we have one).
- ❑ Normally things will probably not function quite as expected.
- ❑ We need to make sure the robot does not fall off edges or cause damage or be damaged.
- ❑ Stop the robot, pick it up.
- ❑ Repeat the whole process with what we think is a fix for the latest bug.
- ❑ Often it is not easy to determine why the robot is not acting as expected and may have to iterate the process numerous tedious times, usually guessing at what fixes might work since we did not have an adequate means of diagnosing the problem.

Imagine that you have a system that allows you to experiment with robotic algorithms using a simulator. Moreover, imagine that the very same program you developed and verified using the simulator can now be run on a real robot painlessly and without having to go through the above quite cumbersome and tiresome process. Furthermore, while the program is running and the real robot is carrying out its tasks, you can collect telemetry from it and even command it to change its actions in real time while it is in the field and may even be inaccessible (e.g. submarine rover). Telemetry serves two purposes:

- Data about the environment and the sensory status of the robot. This data is an integral part of the robot's mission.
- Diagnostics data for the internal integrity of the robot's mechanisms.

These two sources of information make it extremely easy to diagnose any shortcomings or bugs in the robot or the algorithms it is executing.

Well, stop imagining, because that is exactly what we can now do. By following the methodology we have outlined in Chapters 5 and 6 and demonstrated in Chapters 7, 8 and 9 we can build a robot with any combination of actuators and transducers and then fully control it from a PC. The PC can run any software you wish to make the hardware combination you created do anything within its capabilities.

Despite the limitations of the hardware we utilized in Chapters 7 and 8, a robot fitted with it and our firmware can be an adequate robot. In this chapter we will use our hardware as a robot *emulator* to illustrate the efficacy of our protocol. However, the book [*Enhancing The Pololu 3pi With RobotBASIC*](#)⁶⁹ decisively proves the power of the *principles* outlined here while creating a very real, amusing and capable robot. The book improves upon the stock Pololu 3pi robot by implementing our protocol on the 3pi's ATmega328 microcontroller programmed in C. This further verifies the versatility and adaptability of our protocol proving that the details of the microcontroller, hardware or programming languages are incidental.

10.1 The RobotBASIC Simulator

One of the distinguishing features of RB is its integrated robot simulator. There are [four books that deal with the RB simulator](#)⁶² and teach how to use it. There are also [many YouTube video tutorials](#)⁶³ showing how to use the simulator and RB's other features. So we won't go into too much detail here; we will just deal with aspects of RB's simulator that pertain to our strategy throughout this book. We will show how programs developed in previous chapters can be made to work through the simulator's protocol and how simulation programs can work with our hardware by changing the value of a single variable and using some data mapping techniques as we have seen throughout.

Let's start with a very simple example. The following is *pseudo code* for an *algorithm* that ought to make a robot – if we had one aptly equipped and easily programmable – move around forever in a square pattern:

```
Repeat the following forever
  Repeat the following 4 times
    Move forward 70 steps
    Turn about 90 degree to the right
```

It looks simple enough! Go put on the kettle and while your cup of tea is brewing mull over this *thought experiment*. You have a robot with a nice microcontroller and two wheels with two motors. What would it take to make it perform the above algorithm? How much programming would you have to do? How soon do you reckon it will be from the moment you thought about making a robot do the above action before you actually see one doing it?

Type the following distinctly simple program in RobotBASIC and run it:

```
rLocate 400,300
repeat
  for i=0 to 3
    rForward 70
    rTurn 90
  next
until false
```

First, notice the amazing similarity between the pseudo code above and the actual implementation in RB's language. Second, notice how *quick* and *easy* it was to see a robot actually performing the devised algorithm. But this is not a real robot, you say. It is never going to be as satisfying as a real robot, you say. Fine, let's make it a real robot. We have all that great hardware we developed in Chapters 7. What would it take to make the exact same program above drive the real servomotors so that if the PPDB were mounted on wheels driven by the servomotors it would have moved as the simulated robot did? Initially, you might think that you need to add code to take care of sending the command code 6 over the serial port and then receive the 5 bytes and so forth; also the same for turning.

Use the PPDB system as we have done in Chapter 8 or 9 (wireless). Now, add the following two lines of code *to the top* of the above program. Make sure the value of **Port** is set to what you have been using to communicate with the Propeller and then run the program,

```
Port = 16 //change this as per you system
rCommport Port,br115200
```

What just happened? The simulated robot did not move as before. In its place the servomotors on the PPDB started moving. Had the servos been connected to the chassis of a robot it would have actually moved in a similar manner to the simulated robot. It may not in reality circumscribe a perfect square, but if the system were calibrated correctly it could have actually made a pretty good square.

How was this achieved? There were no **SerialOut** commands as we had so far in all the previous chapters. Try again to run the program above with the additional lines but set **Port** to 0. What happens now? The simulated robot is now moving. Change the number back to what it was and run the program again. Again the PPDB is the one doing the actions. This is how simple it is to make the hardware and firmware we have developed behave as if it were a robot.

Consider the versatility and potential power of the actions above:

1. You experiment with the simulator to test an algorithm.
2. Change **Port** from 0 to a valid port where there is a robot programmed with a firmware that obeys our protocol.
3. Run the program and the real robot starts moving.
4. To stop the robot all you have to do is stop the program; you do not have to go to it and switch it off.
5. You want to change the behavior.
6. Switch **Port** to 0.
7. You implement any program changes to do whatever new thing you wanted.
8. Go back to step 1.

Change the statement **rTurn 90** to **rTurn -90** and make sure **Port** is 0. Run the program and watch the simulated robot. It is now going round in squares with left hand turns instead of right hand turns. That is how simple it is to change the behavior of the simulated robot. Just as easily, you can also change the behavior of the *emulated robot* (the PPDB). Change **Port** back to the port number for your system; now the hardware without ever touching it or reprogramming it or uploading to it anything is behaving in a different manner.

Compare the above to the development cycle on a real robot using the traditional methods. There are more advantages that we will discuss in Section 10.4. But first let's see how the RB simulator protocol functions. Also let's see how convenient it is to use this built in protocol as compared to using the raw serial commands we were using in the previous chapters.



This, purely and simply, is the power of what we have achieved. You can make your robot do different things by just changing a few lines of code on a PC screen and immediately have your robot change its behavior without ever connecting it to anything or even touching it (if using a wireless link).

10.2 How Does RB's Protocol Work?

[Cut Out]

10.3 The PPDB Hardware as a Robot Emulator

[Cut Out]

10.3.1 Ranger and Turret

Remember the program Turret_Radar.Bas back in Chapter 8.2.4? The program Turret_Radar_RB.Bas listed below shows how simple it is to do the same program using the simulator. Keep **Port** set to 0 for now. The only changes are the highlighted lines.

Turret_Radar_RB.Bas

```
//Turret_Radar_RB.Bas
//works with Protocol_Main.Bas
Port = 0 //set this as per your system
Main:
  rcommport Port,br115200
  rLocate 400,400
  rinvisible gray,red
  call RadarScreen()
  call Radar()
End
```

```
//-----
sub Ranger(Angle,&Value)
  m= "Comms Error"
  Value = rRange(Angle)
  if Value >= 0 then m = spaces(40)
  xyText 600,10,m,,10,,red
return (Value >= 0)
//-----
sub RadarScreen()
  for i=1 to 400 step 50
    arc i,i,800-i,800-i,0,pi(),2,gray
  next
  for i=0 to 180 step 20
    th = dtor(i) \ r = 400
    line r,r,r+cartx(r,th),r-carty(r,th),1,gray
  next
  savescr
return
//-----
sub Radar()
  j=-90 \ i=1
  while true
    call Ranger(j,V)
    if V < 0 then continue
    f = 1/sqrt(2) \ if _Port then f = 400/23000.0
    V *= f \th = dtor(j-90)
    x = cartx(V,th) \ y = carty(V,th)
    circlewh 400+x-5,400+y-5,10,10,red,red
    j += i \ if abs(j)==90 then i= -i \ restorescr
  wend
return
```

Notice the **rInvisible** command. This command tells the simulator to ignore the colors gray and red. This is because we are drawing the radar screen markings and return spots using these colors. Without the command the simulator's ranger would see them as objects. To avoid this, we use the command to tell the simulator that these colors are not to be considered as objects.

The main change is in the **Ranger** subroutine. It now uses the **rRange()** function to read the distance instead of sending an explicit command over the serial port. If the simulator is active (**Pot = 0**) then the function will read the simulated distances. If the variable **Port** is set to other than 0 then **rRange()** would use the real Ping))) and the real turret and return real distances (actually the raw data is time in microseconds).

Notice in the **Radar()** subroutine how we now multiply the value returned from the ranger by a different factor when the simulator is active than when the real robot is active. To decide whether it is the real robot or the simulator the variable **_Port** is checked. Since we are inside a **call/sub** subroutine we need to specify that we need the global variable. So we use **_Port** instead of **Port** to indicate that the global variable is the desired one.



Call/Sub subroutines use *local-scoping* of variables. To access a *global variable* you have to prefix the variable's name with the operator **_** to indicate a global variable not a local variable with the same name. See [RobotBASIC_Subroutines.PDF](#)⁷² for a tutorial on the powerful features of Call/Sub subroutines in RB.

In order to draw the sonar data on the screen we need to scale the measured distance so as to make it fit within the allotted screen scale. Since the simulated ranger returns the range in pixels while the real ranger returns the reading in microseconds, we will need to scale these readings using different scale factors. The real ranger will return a value with a maximum of 2300; we know this from experience we gained using it in Chapter 7. What is the maximum return on

the simulated ranger? In the Radar program we placed the robot at coordinates 400,400. So the maximum reading the simulated sonar can return is the distance to the corner of the screen. This would of course be **Sqrt(2)*400** (Pythagorean Theorem).

To scale a reading X ranging from 0 to N to a reading Y ranging from 0 to 400 we use the formula
$$Y = X * 400/N$$

So we need to multiply the ranger value by a factor F where F is $400/(\text{sqrt}(2)*400)$ in the simulated case (i.e. $F = 1/\text{sqrt}(2)$). In the real ranger case you might be tempted to use **mmDistance()** to convert it first to millimeters or inches before scaling. But that is entirely unnecessary. The scaling factor can be calculated for the pure raw return value without having to convert it to distance units. Since we know the maximum is 2300 then $F = 400/2300.0$. Notice that we use the floating point 2300.0 not just the integer 2300. This is to force a floating point division rather than integer division which would not be correct. Look at the bolded lines in the **Radar()** subroutine

Run the program with **Port** set to 0 and see how the simulated return is rectangular as expected since the room is rectangular. Now set **Port** to the value corresponding to the PPDB communications port you have been using and see that the same program is working with the real turret and Ping))) and that the plotted sonar returns are real objects.



This mechanism of using a different factor for the simulator than for the real robot is to do with the *data remapping* concept. It is necessary since the simulator's world coordinates and measurements are not the same as ones of the real world. To make the real robot's values correspond to the simulated values we need to data remap (*transform*) the values to keep them both to a *related reference scale*. We will see more of this later.



The program Turret_Radar_RB.BAS is not something to be taken lightly. You need to read this program carefully and make sure you understand its action well.

- It demonstrates almost all the strategies, methodologies and procedures we have outlined in this book.
- It decidedly demonstrates how you can relate a simulated environment to a real one and how *the same program* works with either environment by flipping a switch (**Port** value in this case).
- It demonstrates how the PC can be a crucial and integral component in a sophisticated engineering project.
- In short, *this program is the embodiment of what we are trying to achieve*.

10.3.2 Reading the Compass

[Cut Out]

10.3.3 Reading the QTI Line Sensors

[Cut Out]

10.3.4 Other Devices

[Cut Out]

10.3.5 Handling Errors With the RB Simulator Protocol

[Cut Out]

10.3.6 Your Turn to Have a Go

[Cut Out]

10.4 The RobotBASIC Simulator Protocol Advantage

Three engineering disciplines are required in the design of a robot:

- Mechanical
- Electronic
- Software

The first two are of vital importance but limited duration. Once the robot's mechanical and electronic hardware is designed and manufactured the involvement of these two engineering disciplines comes to an end.

The software aspect is at two levels

- Firmware
- Artificial Intelligence

The firmware aspect of the software is also of limited duration. Once the firmware is designed and is working well with the electronics and the mechanics then there is nothing more to be done.

This leaves the AI level of software. This aspect of the software engineering of a robot should in reality never be over, as with a human being. Once a human is grown up there is nothing more in the biomechanical and bioelectrical aspects of the human body that change (except for the insidious deterioration due to aging and diseases – c'est la vie).

However, mental development never stops (with some people at least).

A closer analogy is to the PC. When you buy a PC the hardware is pretty much fixed and the OS is also mostly fixed (?). However, you are constantly installing new software to make it do new things. Using your computer is fun because you can change software. No one really enjoys changing the hardware or the OS. It is most probable that you don't much enjoy the interminable upgrading of the OS and you do not really relish forking out dollars for hardware. We bet that what you enjoy while using your computer is using new software that makes it do new things – a Flight simulator, a photo editor and of course, RobotBASIC.

So the same hardware + firmware become a new device by changing the software. It is not the aim to keep changing the hardware and the firmware. The aim is to change the software.

The same thing applies to robots. The fun is not to keep tinkering with the hardware and programming of the microcontroller on board (well, some may enjoy that as a goal in itself). Rather the aim is to make the robot do useful and interesting things. The enjoyment would be enhanced and the efficiency would be increased if we could achieve this goal with minimal tinkering with the hardware and minimal plugging and unplugging of the robot to keep changing its firmware.

That is exactly where RB comes into the picture. If we have a robot that has the right hardware and the right OS (firmware) we can then interact with it through RB programs to make it do new and interesting things. RB software makes the robot behave in different manners to do different and interesting things and all without having to modify the robot.

But the RobotBASIC advantage goes even further. The RB simulator provides another layer of enjoyment and effectiveness. With the simulated robot you can develop the software and try out its algorithms and nuances without the associated dangers to a physical robot that might be damaged when it does not behave correctly.

Furthermore, robots are expensive and you are more likely to be sharing the robot with others (perhaps in an educational environment). If you had to wait for your turn before you could develop your ideas you would be wasting much valuable time. With the simulator you can try out ideas, strategies and algorithms and when you do get a turn on the robot you can *immediately* try them out without having to translate to another language or reprogram the hardware.

As you have seen in Section 10.1, the ability of the RB simulator to transfer control to a real robot with the exact same software that was developed with the simulator is a major advantage.

As a teacher or club organizer you can develop a robot with all the hardware you need and then program the microcontroller with the firmware required to implement the protocol outlined in the previous chapters. Once this is achieved, you won't need to reprogram or reengineer the hardware on the robot again. You now have a platform that can be used to do interesting projects and to teach students all about robotics. This is easily accomplished with a PC running RB (or any other language you want).

You can use the simulator to *introduce* beginners to the whole concept of sensors and feedback control. Use the simulator to introduce *algorithmic development* principles. Then when the students/members are at a more advanced level you can *port* them over ☺ to the real robot. They can now use the real robot to do things without having to spend an inordinate time trying to build a robot. They do not have to be electronics engineers or mechanical engineers or experts in C and Assembly and microcontrollers. They can just concentrate on the software aspects in the convenient environment of the PC with all its advantages (keyboard, mouse, GUI, etc.).

Imagine if every time you wanted to teach a computer programming class, the participants had to build a computer from scratch first. How much time would be left over to teaching programming?

The day that *programming* robots becomes the focus, rather than *building* them, is the day more people will be interested in robotics. Once you reach the critical mass of innovators working on the Artificial intelligence aspects of the robot project and not only its low level engineering, we think an explosion in the robotics field will occur that will bring about future applications to rival the software engineering field.



There are programmers today creating applications of paramount importance and utility who cannot distinguish a transistor from a resistor or a servomotor from a stepper motor. They are not electronic or mechanical engineers; yet, they make computers do amazing feats. We need the same level of expertise to be applied to robots. Give people a robot platform equivalent to a PC and let's see what innovations will result. With our proposed strategy and the methodology outlined in this book you can do precisely that. You can give robots that can be programmed just like a PC. The more we do this the closer we will be to that Robot-Lotus 123, or Robot-dBase, or Robot-Linux. And, once that occurs, robotic applications will boom onto the new frontier.

10.4.1 A Case Study

Let's illustrate the advantages of our strategy with a small project. Let's assume that we have a robot that has a sonar sensor and a bumper sensor both in the front of the robot. We want the robot to move around a room avoiding obstacles. We do not want it to get stuck and it would be nice if it managed to cover the whole room eventually.

The Design Advantage

We think about it and we come up with the following algorithm:

```
Repeat the following forever
  While there is nothing in front of the robot
    Move forward 1 step
  Decide to turn left or right randomly
  While there is something in front of the robot
    turn 1 degree in the decided direction
  Turn an additional random value between 20° and 50° in the same direction
```

To test the validity of the above algorithm, we will implement it using the RobotBASIC simulator with a simulated environment of a few obstacles. The program below is the first attempt:

Simualtor_1.Bas

```
//Simulator_1.Bas
Main:
  GoSub Initialization
```



```

repeat
  while rRange() > RangeLimit //while path is not blocked
    rForward 1
  wend
  T = (random(1000)>500)*2 -1 //determine a random turn direction
  while rRange() <= RangeLimit //keep turning until path is clear
    rTurn T
  wend
  rTurn T*(20+random(30)) //turn a little more in the same direction
until false
end
//-----
Initialization:
  RangeLimit = 30
  //draw some obstacle for the simulation
  erectanglewh 295,90,310,110,10,blue
  erectanglewh 700,400,100,140,10,blue
  erectanglewh 200,300,100,140,10,blue
  erectanglewh 600,230,100,40,10,blue
  rlocate 200,200
return

```

The **Initialization** subroutine creates some obstacles in the environment and defines a value for the ultrasound range to be considered as minimal distance to an obstacle.

We decide if there is something in front of the robot by using the sonar ranger (highlighted line). The reverse line is the way we decide to turn left or right. If the random number is bigger than 500 then the expression **(random(1000)>500)** results in 1 otherwise in 0. If we multiply that by 2 and subtract 1 the result is either -1 or 1, which determines if we turn left or right. Also notice that we turn until there is no obstacle as determined by the ranger and then also turn an additional random amount between 20 and 50.

With this simple and easily developed program we can try out our ideas on the RB simulator safely and conveniently and *on our own time*; we do not have to wait until there is a robot available to us. Notice the close analogy between the syntax of the program (**Main**) and the pseudo code of the algorithm. This alone is a major advantage.



Notice how the simulator enabled us to create an environment that may task the efficacy of the algorithm. With a real robot it may be inconvenient and expensive or untenable to create a good testing environment, let alone a *variety* of environments.

The Debugging Advantage

Upon running the program, straight away the robot crashes. It seems the Ranger is not working. So we add the following line *just after the repeat statement* to examine what the ranger is reading:

```
xystring 1,1, rRange(); spaces(20)
```

From this simple debugging ability we determined that the ranger is actually working but it may be that our limit for it is too small. So we increase the limit to 200, but that seems to not work either and besides the robot is keeping too far away from obstacles.

All this illustrates why the simulator is an indispensable tool. Imagine doing what we just did with a real robot and having to change its firmware. How could we even be able to read what its ranger is reporting? Even if we had an LCD on board what are we to do, follow it around trying to read the display? Maybe we can have it store its reading and then we can download and read them. Maybe we can have it send its readings to a PC wirelessly. If we are going to do this

why even bother with programming the robot. Why not have the firmware as we have done in Chapter 8 and then we always know everything we want to know on the convenient PC screen and when we want to change programs we just change the RB program.

Can you now see how the system we have developed is a major advantage? Furthermore, combined with RB's simulator it is a "mega-major" advantage ☺.

Let's go back to the simulation. It seems we need to use the bumper sensor in addition to the sonar. Let's use the bumper to catch any sonar misses. So we change the line:

```
while rRange() > RangeLimit
to
while !(rBumper() & 0%0100) && rRange() > RangeLimit
```

How we decide if there is something in front of the robot is now a combination of two things:

1. Bumpers.
2. Ultrasound ranger.

The reason we use a bumper is to make sure there is nothing that may have been missed by the ultrasound. Also notice how the bumper value is bitwise-ANDed (&) with 0%0100. This is because the simulator has four bumper sensors all reported in one byte where the third bit (bit 2) is the front bumper, so we mask out all the other bits from the value returned by the **rBumper()** function. When turning to move away from the obstacle we used just the ultrasound **rRange()** value.



You now can see the advantages a simulator provides due to superior debugging tools during the algorithmic development stage.



Remember on our hardware we only have one bumper sensor (simulated by the pushbutton on P5), and we are going to assume it represents the front bumper. So some *data mapping* is needed when we eventually move over to the real hardware (see later).

The Exhaustive Testing Advantage

Now run the new version. It seems to work, but will it eventually fail? Again, we have another illustration of the RB advantage. Even though the algorithm seems to be working now, we have only tried it for a short time and with only one environment. Can we try it out for a long time and with *various* random environments?

Imagine doing that with a real robot. How would you obtain various random environments for a real robot? How long do you have to stick around just watching it?

With the simulator we can do it so simply. Below is a new program that creates a random environment and lets the robot roam for a minute then tries out a new environment. We can run this program and leave it running for hours and that way we can try out many environments over many hours. If the robot never collides then we can be assured that the algorithm works. Change the value of **TimeLimit** to say 300 or 600 to let the robot try out each new environment for longer than just a minute (60).



Note that the simulator provides options during the development stages that *cannot* be realized with a real robot.

Study how we create the random environment and note the use of the functions **vType()**, **Limit()** and **Random()**. Also the **KeyDown()** function is used to enable the detection of a spacebar press which a user can press to go to the next

random environment instead of having to wait for the timeout. So if an environment is not to your liking then just press the space bar to go on to the next one.



You do not need to stick around for the duration of the testing; you can go out to lunch. If the robot crashes it will halt the program and display a message. It will stay in that state until you come back. If when you come back from lunch the robot is still scurrying about then you know there was no failure. Your algorithm was tested with numerous environments and for a sufficiently long time on each. If no crash occurs then you can be reasonable assured the algorithm is a **robust** one.

Simulator_2.Bas

```
//Simulator_2.Bas
Main:
  GoSub Initialization
  while true
    if timer()-t > TimeLimit
      GoSub Initialization
      t = timer()
      waitnokey
    endif
    GoSub RoamAround
  wend
end
//-----
RoamAround:
  while !(rBumper()&0%0100) && rRange() > RangeLimit //while path is not
blocked
    if keydown(kc_Space) then t = 0 \ return
    rForward 1
  wend
  T = (random(1000)>500)*2 -1 //determine a random turn direction
  while rRange() <= RangeLimit //keep turning until path is clear
    if keydown(kc_Space) then t = 0 \ return
    rTurn T
  wend
  rTurn T*(20+random(30)) //turn a little more in the same direction
Return
//-----
Initialization:
  if !vType(FirstTime)
    t = 0 \ TimeLimit = 60000 //60 secs
    RangeLimit = 30
    LineWidth 10
    FirstTime = false
  endif
  clearscr
  //draw some obstacle for the simulation
  for i=0 to 3
    x = 50+ Random(700) \ y = 50+random(500)
    W = limit(random(400)+x,80,790)-x
    H = limit(random(300)+y,80,590)-y
    rectanglehw 50+random(750),50+random(550),W,H,blue
```

```
next
rlocate 30,30,135-random(45)
return
```

10.4.2 Implementation Onto the Real Robot

In addition to all the above advantages during the design stage, debugging stage, and the testing stage, we have one more advantage during the implementation stage. Implementation on a real robot is absolutely painless. There really is no actual implementation since there is no programming to be done on the robot. All the programming was already done when we created the firmware that implements the protocol.

To implement the algorithm on the real robot we just enable the communications link and start running the very same code on the real robot. There is no need to translate the code to another language. There is no need to plug the robot into a PC to program it; assuming, of course, that we have already accounted for *data remapping* in our program.

Sensory Data Mapping To RB's Requirements

[Cut Out]

10.4.3 An Exercise

[Cut Out]

A Comment About Feedback Control

[Cut Out]

10.5 A Simplistic Inertial Navigation System

An INS is used on airplanes, submarines, deep sea rovers, satellites, rockets, and a plethora of sophisticated vehicles that require accurate self-contained (i.e. not dependent on external devices) navigation. If an INS is small enough and accurate enough you can place one on the end-effector of a robotic arm and you would be able to drive the manipulator to any position in the workspace of the arm and orient the end effector to any required attitude in 3D space.

To implement a proper INS you need a tri-axis accelerometer unit as well as a tri-axis gyroscopic unit. Also you will need a computer capable of doing floating point math and some complex algorithms to perform filtering (e.g. Digital filters).

The principle is quite simple (unfortunately, the implementation is not). Using the gyroscopic unit you can obtain pitch, roll and yaw of the vehicle. Using the accelerometer unit you obtain acceleration readings in all three axes. Knowing the tilt angles from the gyroscopic unit you can calculate the amount of gravity acceleration in each of the axis. Then you can subtract the gravity components of the acceleration from the system's acceleration components along the axes, which result in the dynamic acceleration on each axis due to forces in three dimensions on the vehicle other than gravity.

Once you know the acceleration on each axis, you use numerical integration (e.g. trapezoidal) to calculate the velocity vectors along each axis and integrate that to obtain displacement vectors for each axis. With this data, and using vector algebra, you can calculate the vehicle's position in 3D. When combined with some great-circle navigation formulas (or rhumb-line – see RB's suite of navigation commands and function), you can calculate the latitude and longitude of the vehicle. The data can also give altitude (or depth).

Using an INS you can continuously track the vehicle's position on earth as well as its velocity and heading without using any external devices (e.g. GPS) and you do not even need a compass. You do however need a known starting

reference point and heading. Also due to precession errors (if using mechanical gyroscopes) the system needs to be occasionally updated with a reference correction.

In our hardware we have half of the required hardware to achieve a proper INS. With the H48C you can measure accelerations on the three axes. You have seen how we used that to calculate roll and pitch. Of course, when we did, we assumed that there were no other forces on the board other than gravity. Therefore any acceleration in the axis direction would have been due to tilting of the axis, which results in a vector component of the vertical g-force along the axis. With this component vector we can calculate the tilt of the axis using trigonometry (e.g. $a \tan 2()$ or $a \cos()$).

Obviously while doing tilt calculation, accelerations due to forces other than gravity cause the calculated tilts to be erroneous due to the additional components of the other accelerations.

Conversely, the H48C can be used to calculate accelerations due to forces other than gravity on the device. The problem is that if the device is not perfectly level (i.e. there are tilts) then the component of gravity resolved along the axis would offset the total acceleration on the axis. However, if we know this component then we can easily calculate the acceleration due to dynamic forces. That is why a gyroscopic unit is needed.

10.5.1 The Experiment

On the Parallax web site there is a most excellent manuscript called [Smart Sensors And Applications](#)⁶⁴ and is part of the [Stamps In Class](#)⁶⁵ educational series. In Chapter 6 of the document there is an interesting experiment to calculate and graph the displacement of an RC car by using an accelerometer and the BS2 microcontroller.

The write-up of the activity has great details of the required processes and the principles behind them. The task of calculating all the integrations and the graphing was delegated to an Excel worksheet loaded with the data after having collected it during the experiment by logging it in the BS2's EEPROM.

This meant that the experiment had to be run and the data logged in the BS2's memory. Then when done you had to plug the BS2 to a computer and run another program on it to download the logged data. Then you had to copy the data and start the Excel program. Then you had to paste the data into the spreadsheet and write the appropriate formulas. And then you had to use the facilities of the spreadsheet to graph the data. To do another experiment you had to repeat this whole process.

One of the advantages of our protocol and the overall strategy of using a PC to interface to the hardware is that graphical displays and data collection as well as the floating-point math are easily achieved, and are an integral part of the system. So you won't have to resort to third-party software or go through the inconvenience of data formatting and so forth. Since RobotBASIC is a programming language you can use it to do all the data logging, calculations and graphing. In addition, all that can be repeated as often as you need without any plugging and unplugging, changing programs or any of the associated hassles.

To illustrate the power and efficacy of what we have achieved throughout this book we will use the simulator protocol to do a similar experiment as in the above educational document. We will move the PPDB by hand back and forth and then calculate its acceleration, velocity and displacement and log all that and graph it.

Rather than assume that the hardware is level and because we do not have a gyroscopic unit we will have to do an initial calibration before any dynamic forces are applied. We also have to assume that any initial tilts will not change. Using this setup we can then assume that the changes of accelerations along the axes from the original static reading are due to the dynamic forces. Knowing these accelerations, we can use digital integration to calculate velocities and displacements. The values will then be graphed.

Place the PPDB on a *steady and flat* surface that does not rock or shake and has no bumps. Then run the `Simplistic_INS.Bas` program. The program will run through the calibration then will sound a tone on the speaker. Once the tone stops start moving the PPDB on the surface.

A good experiment is to move the PPDB forward some distance then backwards past the starting point then back to the starting point. When you are done press the *Finish* button on the screen. The RB program will do all the calculations then display two graphs showing the acceleration, velocity and displacement in the x-axis against time and also in the y-axis. See Figure 10.4.

As you can see, the graph of the displacement shows that the PPDB was moved about 300 millimeters forward (positive) then backwards (negative) which was in fact the case.

10.5.2 The Results

There are shortcomings in the system. Despite the initial calibration being an attempt to eliminate any residual tilt, you can see that even when the system is static and there is no movement, the acceleration can be quite noisy. This noise can cause errors when integrating and even worse when integrating twice. Integration is a summing process and any systematic errors tend to accumulate. This is especially true if the sampling rate is not adequate for the dynamic frequencies in the system. See the next section for a brief discussion of this.

If you look at Figure 10.4 on the left side (x-Axis plot) and if you observe the right end of the graph you will notice that the velocity did not go back to 0 as it should. This of course is also causing the displacement to continue to increase. Notice the problem is even worse on the y-Axis. See Section 10.5.4 for a possible reason for this.

The system is simplistic and does not take into account any calibration errors in the H48C, temperature effects, tilt or noise. To improve the system we need a filtering algorithm to filter the noise out of the acceleration data. We also need a gyroscopic unit to eliminate any g-forces due to axial tilting. Another, shortcoming of the system is the sampling rate. Too slow a sampling rate may cause problems for the integration process due to the aliasing aspect (Nyquist Criterion see Section 10.5.4).

Despite all the shortcomings, the system performed surprisingly well and the displacement was not far off the mark. Our aim was to demonstrate how our methodology enables:

- An easy and effective user interfacing.
- Floating point and other complex mathematical calculation to be an integral part of the system.
- Data logging and storage to be an integral part of the system.
- Data graphing and presentation to be an integral part of the system.

The experiment achieved the aim adequately.

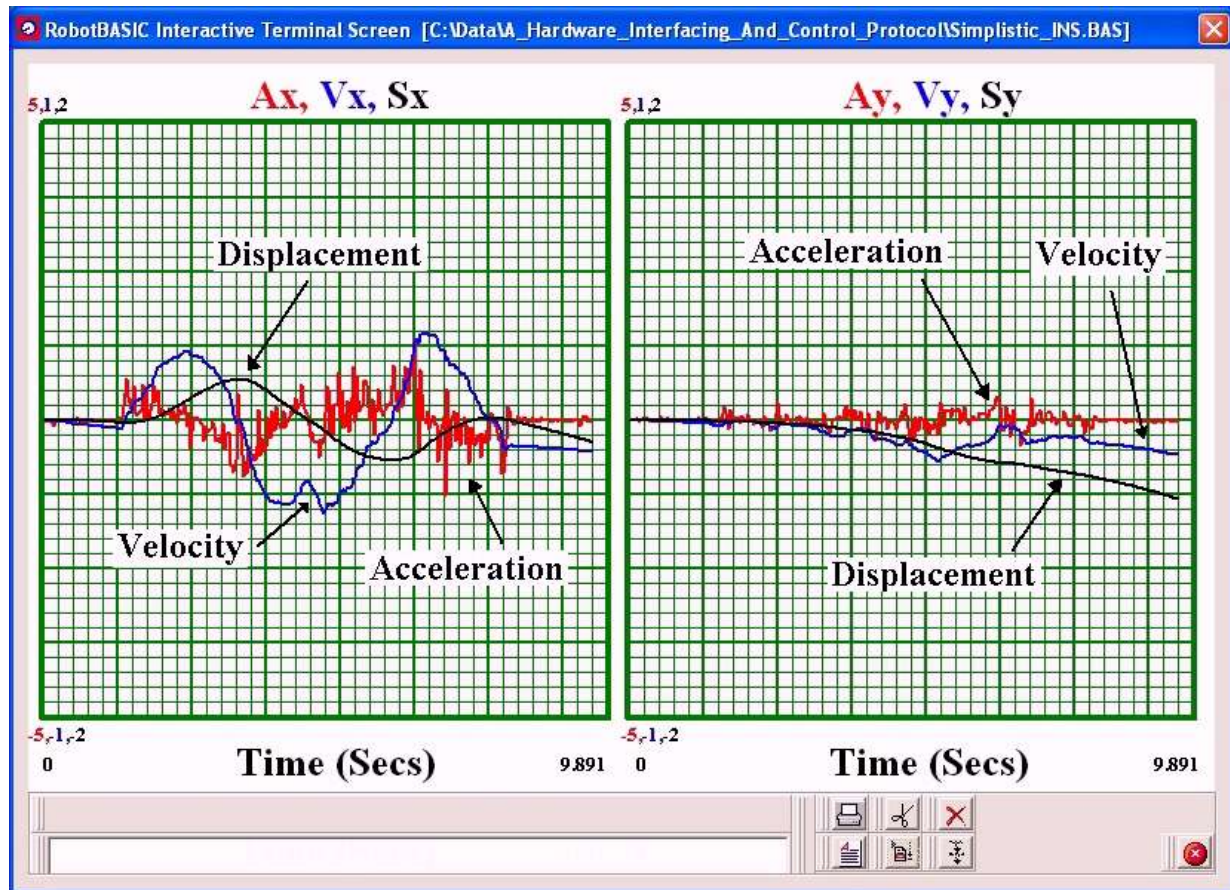


Figure 10.4: Results of running Simplistic_INS.Bas

10.5.3 The Program's Details

The program uses `rCommand()` of the simulator to send the H48C commands and uses the same calculations we did before to reconstitute the axes' accelerations. The calibration process averages 100 readings and that value is subtracted from all subsequent readings to eliminate any inherent tilt.

The formula used to calculate the velocity is

$$V_i = V_{i-1} + A_i * \Delta t_i$$

The formula used to calculate the displacement is

$$S_i = S_{i-1} + V_{i-1} * \Delta t_i + (A_i / 2) * \Delta t_i^2$$

Rather than use a constant interval for the value of Δt_i we calculate the actual value. We use a timer to note the time each sample was taken. Thus we can obtain an accurate time interval between the samples.

To plot the graphs we use RB's graphics commands, which are quite versatile and convenient. Arrays are used extensively in the program to log the data and to make it easy to plot them.

The program will also save the results to a file called `Simplistic_INS.CSV`. This is a text file that can be viewed in the RB editor. However, it is also a file that follows a standard that can be viewed in Excel to display the data as a spreadsheet. So you can view the data if you need to in Excel. Study how simple it was to accomplish this in the `SaveToFile()` subroutine.

Notice the subroutine **Calculate2()** (with a 2). This is an alternative to the **Calculate()** subroutine. To use it change the highlighted line to say **Call Calculate2()**. This routine will perform some filtering on the acceleration data before it uses it in the integration to calculate the velocity and displacement. The filtering is accomplished using a *polynomial-curve-fitting* algorithm using the command **mPolyFit** in RB.



The program did not need to use the simulator protocol. It could have been written using normal serial communications commands as in previous chapters. We used the simulator protocol to illustrate the versatility of our whole system.

Simplistic_INS.Bas

```
//Simplistic_INS.Bas
//works with Protocol_Main.Spin
Port = 16 //change this as per your system
#include "..\Utilities\IncludeFiles\Instruments.Bas"
//-----
Main:
  rcommport Port,br115200
  rLocate 600,300
  gosub Calibrate           //calibrate
  Call CollectTheData()     //Collect the data
  Call Calculate()          //do calculations
  Call SaveToFile()         //save all the results to a disk file
  Call Plot()               //graph the data
End
//=====
sub SendCommand(C,P,&s)
  m = ""
  s = rCommand(C,P) \ x = Length(s)
  if x < 5 then m= "Comms Error"
  xystring 500,20,spaces(30)
return (x == 5)
//=====
Calibrate:
  print "Calibrating....please wait"
  Sgx = 0 \ Sgy=0 \ N = 100
  for i=0 to N-1
    call Read_H48C(0,,,gX,gY) //read N readings
    Sgx += gX \ Sgy += gY
  next
  data values;Sgx/N,Sgy/N //initialize array with average
  clearscr
Return
//=====
sub CollectTheData()
  clearscr
  print "When the tone stops start moving the Board"
  print ".....Press Finish when done....."
  rCommand(73,70) \ delay 3000 //sound tone and wait 4 secs
  rCommand(73,0) //stop the tone
  addbutton "&Finish",300,300 \ FocusButton "&Finish"
  data values;timer() //starting time
  for i=0 to 5000 //limit to 5000 readings (4 minutes)
    call Read_H48C(0,,,gX,gY) //read the accelerations
    data values;gX,gY,timer() //insert in an array
```



```

    if lastbutton() != "" then break //exit if button pushed
next
RemoveButton "&Finish"
Return
//=====
sub Calculate()
ClearScr \ Print "Calculating.... Please Wait"
data T;0 \ data Ax;0 \ data Ay;0 //init arrays
data Vx;0 \ data Vy;0 \ data Sx;0 \ data Sy;0
for i=3 to maxdim(values)-1 step 3
    I = i/3 \ J=I-1
    data T; (values[i+2]-values[2])/1000 //timing array
    dt = (values[i+2]-values[i-1])/1000 \ dt2 = dt*dt/2
    data Ax; (values[i]-values[0])*9.81 //Acceleration
    data Ay; (values[i+1]-values[1])*9.81
    data Vx; Ax[I]*dt+Vx[J] //V[i]=A*dt+V[i-1]
    data Vy; Ay[I]*dt+Vy[J]
    data Sx; Ax[I]*dt2+Vx[J]*dt+Sx[J] //S[i]=(A*dt^2)/2+V[i-1]*dt+S[i-1]
    data Sy; Ay[I]*dt2+Vy[J]*dt+Sy[J]
next
return
//=====
sub Calculate2()
ClearScr \ Print "Calculating.... Please Wait"
data T;0 \ data Ax;0 \ data Ay;0 //init arrays
data Vx;0 \ data Vy;0 \ data Sx;0 \ data Sy;0
n = Maxdim(values)/3
dim AccX[2,n], AccY[2,n]
mconstant AccX,0 \ mconstant AccY,0
for i=3 to maxdim(values)-1 step 3
    I = i/3 \ J=I-1 \ t = (values[i+2]-values[2])/1000
    AccX[0,I] = t \ AccY[0,I] = t \ data T;t
    AccX[1,I] = (values[i]-values[0])*9.81
    AccY[1,I] = (values[i+1]-values[1])*9.81
next
m = 5 \ data CoefficientsX;m //change m to 3 or 4 to see the effects
data CoefficientsY;m
mPolyFit AccX,CoefficientsX
mPolyFit AccY,CoefficientsY
for I=1 to n-1
    ax = 0 \ ay = 0 \ J = I-1
    for j=0 to m
        ax = ax+CoefficientsX[j]*T[I]^j
        ay = ay+CoefficientsY[j]*T[I]^j
    next
    dt = T[I]-T[J] \ dt2 = dt*dt/2
    data Ax;ax \ data Ay;ay
    data Vx; Ax[I]*dt+Vx[J] //V[i]=A*dt+V[i-1]
    data Vy; Ay[I]*dt+Vy[J]
    data Sx; Ax[I]*dt2+Vx[J]*dt+Sx[J] //S[i]=(A*dt^2)/2+V[i-1]*dt+S[i-1]
    data Sy; Ay[I]*dt2+Vy[J]*dt+Sy[J]
next
Return
//=====
sub SaveToFile()
s = "Time,Ax,Vx,Sx,Ay,Vy,Sy"+crlf()

```

```

for i=0 to maxdim(T)-1
    s += ""+T[i]+", "+Ax[i]+", "+Vx[i]+", "+Sx[i]
    s += ", "+Ay[i]+", "+Vy[i]+", "+Sy[i]+crlf()
next
mFromString tA,s
mTextFW tA,"Simplistic_INS.CSV"
return
//=====
sub Plot()
    //plot the graphs
    Clearscr \fnt = "Times New Roman"
    data GPx_Specs;10,100,380,400
    mGraphPaper GPx_Specs //graph paper
    data GPy_Specs;405,100,380,400
    mGraphPaper GPy_Specs
    //label the graphs
    for i=0 to 1
        xyText 150+400*i,65,"A"+char(ascii("x")+i)+", ",fnt,20,fs_Bold,red
        xyText ,, "V"+char(ascii("x")+i)+", ",fnt,20,fs_Bold,blue
        xyText ,, "S"+char(ascii("x")+i),fnt,20,fs_Bold,black
        xyText 400*i,80,"5",fnt,10,fs_Bold,red
        xyText ,, "1",fnt,10,fs_Bold,blue
        xyText ,, "2",fnt,10,fs_Bold,black
        xyText 400*i,505,"-5",fnt,10,fs_Bold,red
        xyText ,, "-1",fnt,10,fs_Bold,blue
        xyText ,, "-2",fnt,10,fs_Bold,black
        xyText 140+i*400,515,"Time (Secs)",fnt,20,fs_Bold
    next
    mT = max(T)
    for i=0 to 1
        xyText 10+i*400,525,"0",fnt,10,fs_Bold
        xyText 350+i*400,525,Format(mT,"###0.0##"),fnt,10,fs_Bold
    next
    //plot Ax,Vx,Sx
    m=maxv(maxv(abs(Max(Ax)),abs(Min(Ax))),5)
    Data GxA_Specs;10,100,370,400,red,,, -m,m //graph specs..force limits
    mPlotXY GxA_Specs,T,Ax //plot acceleration against time
    m=maxv(maxv(abs(Max(Vx)),abs(Min(Vx))),1)
    Data GxV_Specs;10,100,370,400,,,, -m,m //graph specs..force limits
    mPlotXY GxV_Specs,T,Vx //plot the velocity against time
    m=maxv(maxv(abs(Max(Sx)),abs(Min(Sx))),2)
    Data GxS_Specs;10,100,370,400,Black,,, -m,m //graph specs..force limits
    mPlotXY GxS_Specs,T,Sx //plot displacement against time
    //plot Ay,Vy,Sy
    m=maxv(maxv(abs(Max(Ay)),abs(Min(Ay))),5)
    Data GyA_Specs;405,100,370,400,red,,, -m,m //graph specs..force limits
    mPlotXY GyA_Specs,T,Ay //plot acceleration against time
    m=maxv(maxv(abs(Max(Vy)),abs(Min(Vy))),1)
    Data GyV_Specs;405,100,370,400,,,, -m,m //graph specs..force limits
    mPlotXY GyV_Specs,T,Vy //plot the velocity against time
    m=maxv(maxv(abs(Max(Sy)),abs(Min(Sy))),2)
    Data GyS_Specs;405,100,370,400,Black,,, -m,m //graph specs..force limits
    mPlotXY GyS_Specs,T,Sy //plot displacement against time
Return

```

10.5.4 A Brief Note About Sampling Rates and the Nyquist Limit

[Cut Out]

10.6 Summary

In this chapter we:

- ❑ Looked at the RobotBASIC robot simulator and examined many of the advantages of using a robot simulator.
- ❑ Saw how the simulator's inbuilt protocol resembles the protocol we developed in the previous chapters.
- ❑ Saw how the hardware we developed could be used as a robot emulator and that if it is actually installed on the chassis of a robot it would in fact be functional as a robot controller.
- ❑ Saw that the RB simulator protocol can be used to control the hardware instead of using normal serial commands.
- ❑ Saw that with the change of one number (and some data remapping) we can have a simulated robot program start controlling a real robot.
- ❑ Converted many of the programs used to control the hardware in previous chapters to use with the simulator commands.
- ❑ Used the simulator protocol to develop a simplistic INS program to calculate displacements from acceleration data.
- ❑ Saw another application of the power, versatility and simplicity of using RobotBASIC as an integral part of a system to facilitate data presentation, data logging and data storage.

Appendices

Appendix B Tables & Schematics

Final Protocol Objects Hierarchy Map

[Cut Out]

Extended Protocol Objects Hierarchy Map

[Cut Out]

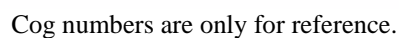


Figure B.2: Propeller Pin Utilization

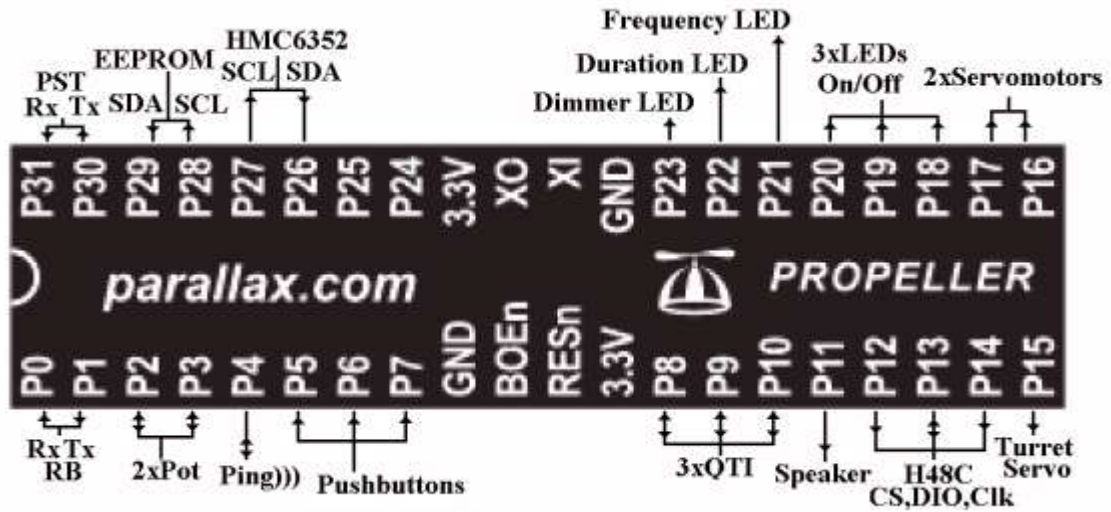


Figure B.3: Hardware Connection Schematics

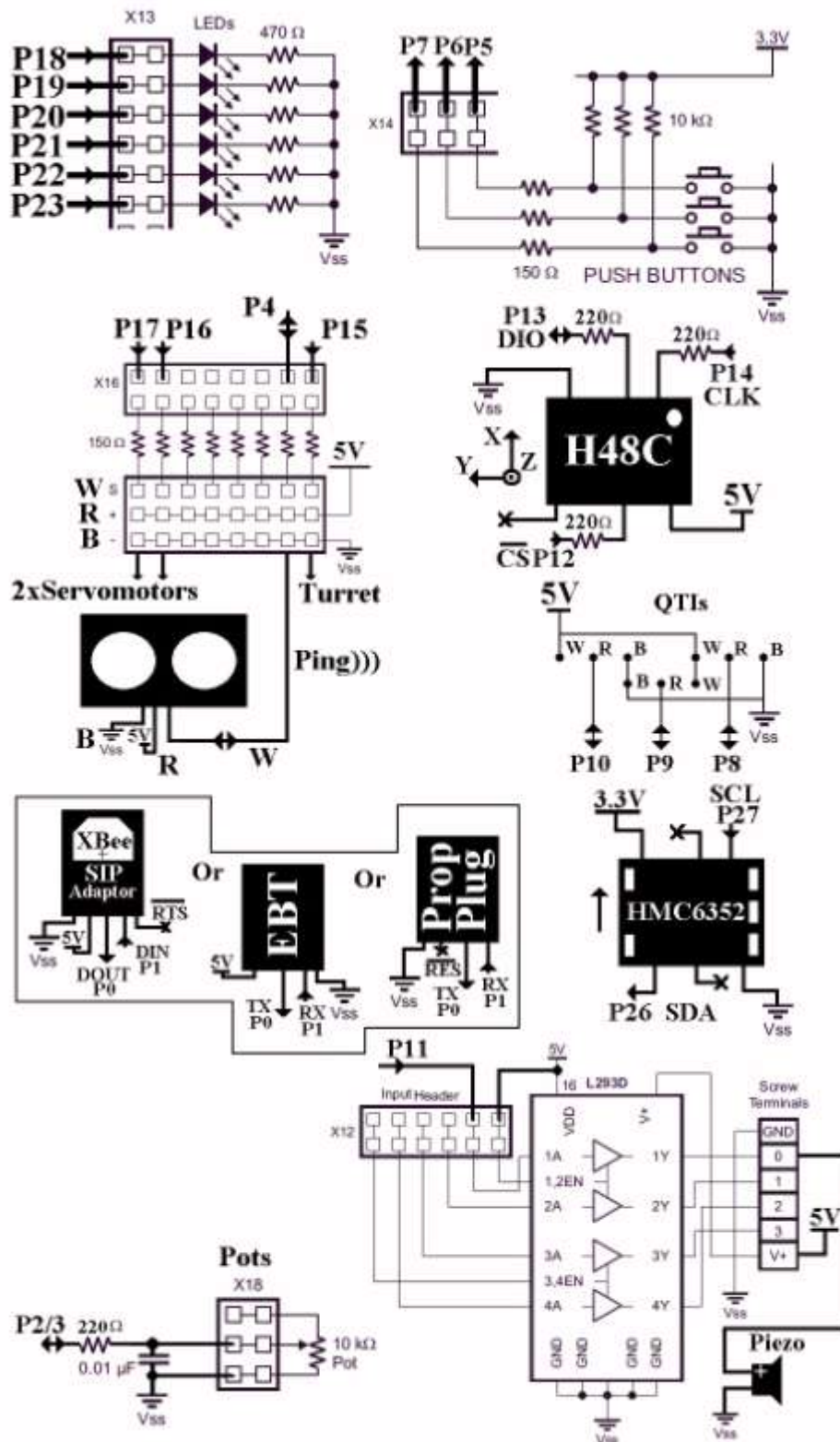


Figure B.4: Photograph of the Final PPDB Hardware Arrangement

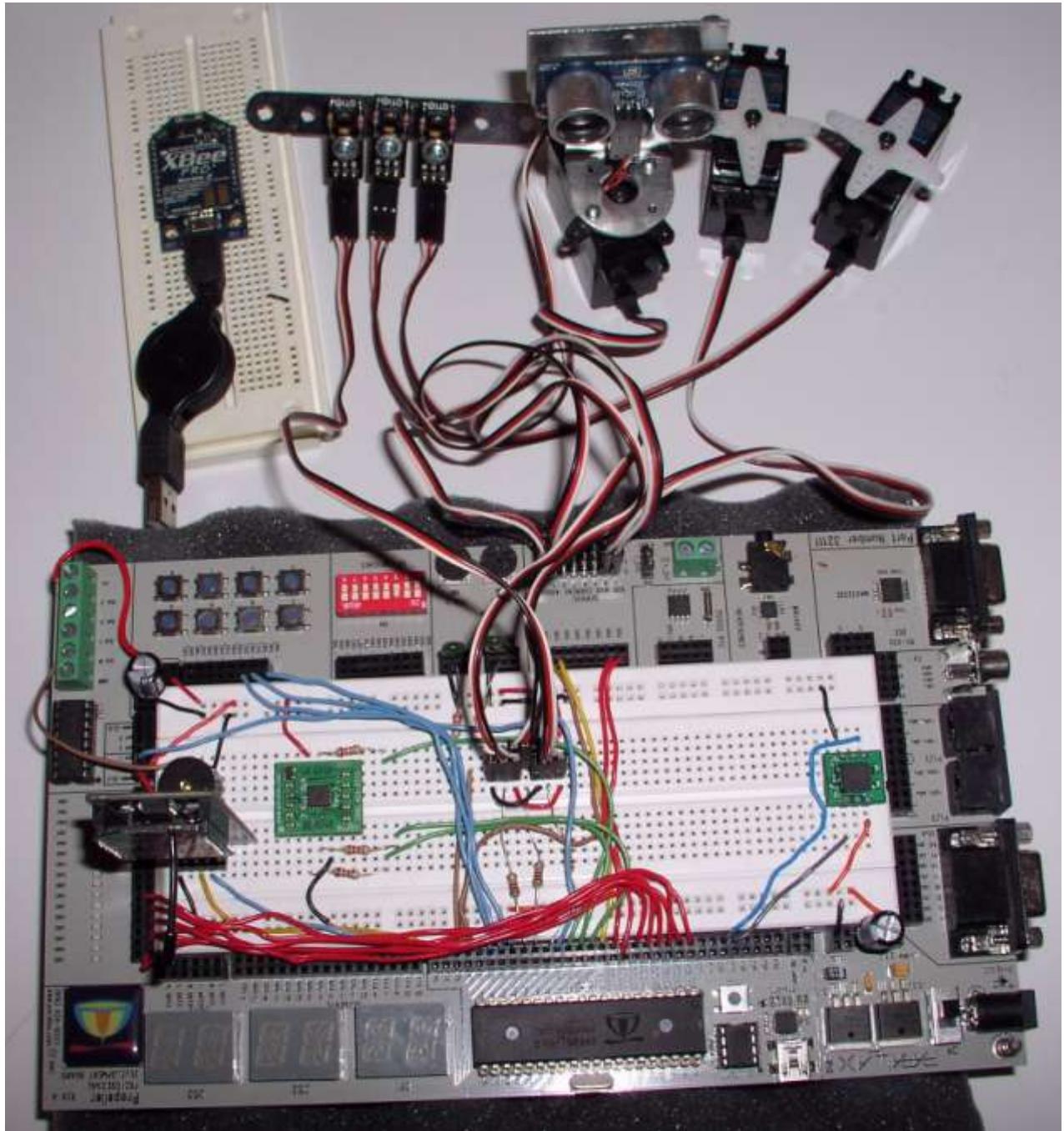


Table B.1: Final Firmware Protocol Command Codes

Command	Code	Parameter	Updates Three Critical Sensors	Data Returned
Stop Motors	0	0	Yes	None
Get System Parameter's value from RAM	4	Parameter# (See Table 8.6)	No	Value of Parameter Little-Endian in the first 4 bytes If success 5 th bytes is 0 if failed then \$FF
Save or Display System Parameters from RAM or EEPROM	5	0=Save 1=Restore Factory Settings 2=List EEPROM 3=List RAM	Yes	Only when saving. 5 th byte is 1 if success or 0 if not
Motors On P17..P16 Forward Backward	6 7	Amount	Yes	None
Turn right motor on P16 forward Backward	8 9	Number Of Steps 0=off 255= all the time	Yes	None
Turn left motor on P17 forward Backward	10 11	Number Of Steps 0=off 255= all the time	Yes	None
Turn Motors on P17..P16 Right Left	12 13	Amount	Yes	None
Read the Compass On P27..P26	24	0	Yes	Last two bytes
Check if a compass is available	24	1	Yes	4 th byte = 0, 5 th byte =1 if available or 0 if not
Calibrate the Compass	24	2=Manual 3=Automatic	Yes No	None
Read the Pots On P3..P2	66	0	No	First 4 bytes
Read the H48C on P14..P12	70	0=Get Axis 1=Get vRef	Yes/No	vRef in 4 th and 5 th bytes Axis in 5 bytes
Play a note on the Speaker on P11	73	Note# (0-84)	Yes	None
Read the Ping))) on P4 and Move Turret on P15 Right and Left	192 193	Angle (0-90)	Yes	Last two bytes
Set P20..P18 LEDs	1	LED States	Yes	None
Set P21 Blink Frequency	2	Hz Value	Yes	None
Reset the Propeller	255	0	No	None
Set P23 LED brightness	200	Level	Yes	None
P22 LED Blink duration	201	Level	Yes	None
Set 2 nd byte receive Timeout1	202	N x 10ms	Yes	None
Set operations Timeout2	203	N x 10ms	Yes	None
Set L_Speed	240	Speed	Yes	None
Set T_Speed	244	Speed	Yes	None
Set L_Timeout	241	N X 10 ms	Yes	None
Set T_Timeout	245	N X 10 ms	Yes	None
Set StepTime	242	N X 10 ms	Yes	None
Set TurnTime	243	N X 10 ms	Yes	None

Table B.2: System Parameters Mapping Formulas

When using the command to read back a system parameter the RB program will receive it from the Propeller as it is stored in the RAM, which is a 32-bit Integer. To convert the number into a value from 0 to 255 which is the value sent from an RB program to the Propeller, we will use a factor with which to multiply the value received from the Propeller. The table below shows these factors; F is the clock frequency of the Propeller, which should be 80 MHz for our purposes. The top part are the values in the Main object and the lower part are in the Motors object.

Parameter Description	Command Number	Parameter Order	Multiply Factor
P22 blinker LED Duration	201	0	$255/2/F$
P21 blinker LED frequency	2	1	1
P23 dimmer LED brightness level	200	2	1
TimeOut1 for receiving second byte	202	3	$2/F$
TimeOut2 for a command to finish its working	203	4	$2/F$
L_Speed for speed of motors in linear movement	240	5	1
T_Speed for speed of motors in turning	244	6	1
Step_Time the time for keeping motors on to go one step linearly	242	7	0.1
TurnTime is the time to keep the motors to turn 1 degrees	243	8	0.1
L_TimeOut is the time to keep the motors on after finishing a step in linear movement	241	9	0.1
T_TimeOut is the time to keep the motors on after finishing a turn in turning movement.	245	10	0.1

Table B.3: Extended Firmware Command Codes (In addition to Table B.1)

Command	Code	Parameter	Updates Three Critical Sensors	Data Returned
Read the H48C 3-Axis values adjusted for the reference voltage.	71	0 (1-byte)	No	3xLongs (12 bytes). Each is a 4-byte Little-Endian Long.
Get All System Parameter's value from EEPROM or RAM	254	0 = EEPROM 1 = RAM (1-byte)	No	12xLongs, 1 st long is count (11). The other 11 longs are the values of the 11 system parameters. Little-Endian 4-byte longs.
P22 LED Blink duration as a duration in Propeller ticks where 80_000_000 is 1 second.	211	Time duration 4-bytes Little-Endian Long	Yes	None
Set 2 nd byte receive Timeout1 as a duration in Propeller ticks where 80_000_000 is 1 second	202	Time duration 4-bytes Little-Endian Long	Yes	None
Set operations Timeout2 as a duration in Propeller ticks where 80_000_000 is 1 second	203	Time duration 4-bytes Little-Endian Long	Yes	None
Set L_Timeout in milliseconds	250	Milliseconds 4-bytes Little-Endian Long	Yes	None
Set T_Timeout in milliseconds	253	Milliseconds 4-bytes Little-Endian Long	Yes	None
Set StepTime in milliseconds	251	Milliseconds 4-bytes Little-Endian Long	Yes	None
Set TurnTime in milliseconds	252	Milliseconds 4-bytes Little-Endian Long	Yes	None

Table B.4: RobotBASIC Inbuilt Protocol Command Codes

Command/Function	Code	Parameter	Updates Three Critical Sensors	Data Returned	Error
rLocate ne_X,ne_Y	3	ne_X	Yes	None	None
rForward +ne_Amount	6	ne_Amount	Yes	None	Halts program
-ne_Amount	7	ne_Amount	Yes	None	Halts program
rTurn +ne_Amount	12	ne_Amount	Yes	None	Halts program
-ne_Amount	13	ne_Amount	Yes	None	Halts program
rCompass()	24	0	Yes	Last two bytes	-1
rSpeed ne_Speed	36	ne_Speed	Yes	None	None
rLook({+ne_Angle})	48	ne_Angle	Yes	Last two bytes	-1
({-ne_Angle})	49	ne_Angle	Yes	Last two bytes	-1
rGPS vn_X,vn_Y	66	0	No	First 4 bytes	-1,-1
rBeacon(ne_Color)	96	ne_Color	Yes	Last Two bytes	-1
rChargeLevel()	108	0	Yes	Last two bytes	-1
rPen ne_State	129	ne_State	Yes	None	None
rRange({+ne_Angle})	192	ne_Angle	Yes	Last two bytes	-1
(-ne_Angle)	193	ne_Angle	Yes	Last two bytes	-1
rCommand(ne_Command,ne_Data)	ne_Command	ne_Data	No	String with 5 bytes	Empty buffer

Figure B.5: Protocol State Diagrams

