

An Introduction to C Programming for FIRST Robotics Applications*

Eugene Brooks and David Brooks
FIRST Team 1280

December 28, 2005

Abstract

The goal of this manual is to introduce the C programming concepts that are useful in developing a program for the robot controller in a FIRST competition robot. It is not intended to cover C programming in general as a lot of the C programming language is not used in robot programming. This manual contains both basic material, material that is specific to the PIC micro-controller, and a detailed discussion of a number of advanced concepts that are difficult to find elsewhere. Although the intent is to be reasonably self contained, the pace is rapid and a programming novice will likely find it useful to augment the instruction contained herein with material from the references.

*Copyright © 2004, 2005 by Eugene D. Brooks III. Permission is granted to acquire a personal copy, electronic or printed, by any means, and use this document for the purpose of learning C programming for FIRST robotics applications. This copyright places no restrictions on any third party who may be helping you use this document for its intended purpose, or may be helping you to acquire a copy of the document for its intended purpose. The author encourages such assistance. All other rights are reserved.

Contents

1	Introduction	5
2	Extensible Interactive C (EiC)	5
3	Hello World	7
4	The C Programming Language	9
4.1	The Composition of a C Program	9
4.2	Integer Types and their Properties	10
4.3	Variable Definitions and Declarations	12
4.4	Functions	13
4.5	Data Hiding	15
4.6	Expressions and Operators	16
4.6.1	Arithmetic Operators	16
4.6.2	Conditional Operators	16
4.6.3	Logical Operators	17
4.6.4	Assignment Operators, and Expressions	17
4.6.5	Bitwise Operators	18
4.6.6	Cast Expressions	19
4.6.7	Conditional Expressions	19
4.6.8	Compound Expressions	20
4.7	Conditional Code	20
4.8	The Switch Statement	22
4.9	Looping	23
4.10	Pointers	25
4.11	Arrays	25
4.12	Character Constants	26
4.13	Type Qualifiers	27
5	Storage Class, Scope and Linkage	28
5.1	Storage Class Modifiers	29
6	Binary	31
6.1	Positive Integers	31
6.2	Twos-Compliment Signed Arithmetic	33
6.3	Fixed-Point Arithmetic	37

7	Special Issues/Limits on the Robot Controller	39
8	Formatted Output	40
8.1	The printf() library function	40
8.2	Printf() in the Robot Controller	41
8.3	Extending Formatted Output	43
9	The C Pre-Processor	48
9.1	File Inclusion	48
9.2	Macro Expansion	48
9.3	Conditional Compilation	50
10	Classic C Programming Errors	51
10.1	if(expr) without braces	51
10.2	The = and == operators are distinct	52
10.3	The bitwise and logical operators are distinct	52
10.4	Seeing the ++ trees in the forest	53
11	The Robot Packet Loop	53
12	Checking Your Robot Controller	55
13	Polled Wheel Counters	58
14	Measuring time	59
15	Controlling the Air Compressor	61
16	Analog Feedback	61
17	Time Integration	64
18	Integrating a Rotational Rate Gyro	66
19	Interrupt Programming	69
19.1	Saving and Restoring Program Context	71
19.2	Race Conditions	72
20	Interrupt Based Wheel Counter and Gyro	77

21 State Machine Programming	83
Glossary	85
References	94
Index	95

List of Tables

1	Integer types provided by EiC	10
2	Integer Types Provided by the C18 C Compiler	11
3	Arithmetic Operators in C	16
4	Conditional Operators in C	17
5	Logical Operators in C	17
6	Assignment Operators in C	18
7	Bitwise operators in C	19

1 Introduction

The intended audience includes high school school students with a good command of algebra and some basic programming experience, high school teachers who are leading FIRST competition teams, and engineers who are mentoring teams. The latter are expected to be competent with regard to programming, but may benefit from detailed discussion of some of the more advanced techniques involved in programming the control system of a FIRST competition robot.

This manual describes numerous software mechanisms that are used on the Team 1280 competition robot. Code that implements many of the programming methods discussed here can be found on the Team 1280 web site: <http://srvhsrobotics/eugenebrooks/>. Other FIRST teams are encouraged to make use of these routines and code snippets when integrating the code for their robot control program, to the degree that what we have written is found to be useful.

Although the text is written in a useful order, the novice may benefit from access to C programming texts of a more tutorial nature, while the expert will want to get right to the last few sections that address advanced techniques. An extended table of contents, and an index, are provided to make it easier to find useful discussions in the document. An extended glossary is provided at the end of the manual to provide the novice with a suitable reference for some of the terms used in the manual. Most of the terms are relatively standard in the computer science field, those that are not were made up by the authors.

This document is a work in progress, getting roughly a yearly update or edition that is posted prior to the kickoff for each FIRST robotics competition season. The most recent version may be obtained from <http://srvhsrobotics.org/eugenebrooks/IntroCProg.pdf>, and will be posted to the Chief Delphi web site, <http://www.chiefdelphi.com>, as a white paper if it continues to stay within the size limits of such posts.

2 Extensible Interactive C (EiC)

We will use the Extensible Interactive C (EiC) interpreter developed by Edmond J. Breen as our vehicle for learning. This interpreter is a relatively

self-contained implementation of the C programming language. It has been extended to support executable statements outside of functions. The EiC interpreter is freely available and runs on a variety of platforms including Windows, which we use to develop programs for the robot controllers. The fact that EiC is self-contained allows us to learn C without using an extraneous build environment that will not be used for robot programming. The fact that EiC is an interpreter also provides beginners with immediate feedback by typing declarations and executable statements directly into a terminal window.

The documentation, source code, and binary distributions of EiC are available at: <http://eic.sourceforge.net>. A binary distribution of EiC for Windows, including the functions developed for this manual, is available at <http://srvhsrobotics.org/eugenebrooks/EiC.zip>. We recommend that you use this latter source to avoid having to type in the examples described in this document, though it may provide useful debugging practice.

When you unpack the zip file downloaded from either of the sources listed above you will obtain a folder named “EiC”. Move this folder to the “C” drive, as “C:EiC”. You will want to put your practice code in files so that they may be easily read into the EiC interpreter again and again as you try various examples. These files must be put in the folder “C:EiC” for the interpreter to be able to find them.

Open the “C:EiC” folder to browse its contents. You will find an application named “EiC”, among other files and folders. Double clicking on this application will start it in a terminal window. Use the special EiC command

```
:exit [enter]
```

in order to exit EiC, where [enter] means hit the “enter” or “return” key on your keyboard. If you obtained the EiC distribution from the SRVHS robotics site there will also be a number of “.c” files in this folder. You will add to this pool of examples when you write your own routines as you go through this manual.

The EiC interpreter has decent error handling abilities. If you type input that results in an error, the interpreter will clean up to the last valid statement after spitting out a verbose error message after which you can try a corrected version of the input. The only real rub is the continuation character, “\”, that is required for multi-line statements. The only reasonable way around this problem is to put more complex code examples in files, and read them into the interpreter at the command line interface using the “#include” statement.

We are presenting elements of the C programming language quite formally, showing examples of how C programming constructs might be used to program portions of the robot control system. Some of the presentation is tutorial, while some is not. One normally learns a programming language over a period of a year in high school, or during a one-semester course in college. Our pace is much faster than that and we cover only those topics that are needed to program the robot. Some of the material is basic, but we also cover advanced material that you would never see in an introductory course in C programming.

Once you get comfortable with EiC, it will be time for you to try programming either the EDU or the competition Robot Controller (RC). This is accomplished using the Microchip C-BOT¹ compiler environment provided in the FIRST kit of parts, or available from Innovation First, using one of the many available code examples as a starting point. See the Innovation First web site, <http://www.innovationfirst.com>, for documentation and example programs in order to get started in this endeavor.

3 Hello World

The ubiquitous introduction to C programming in almost every text on the subject is the one that prints “Hello World!” on the terminal console. We make no exception here because going through this exercise offers quite a bit of value in terms of learning the mechanics of the EiC environment. To this end, use notepad to create a file named “hello.c” in the C:EiC folder. The easiest way to do this is to copy one of the files that are already there, making sure that you do not damage the original, and then modify it. In notepad, your file, “hello.c”, should appear as follows:

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

Once you have done this, save the file and exit notepad, noting that the “.c” in the file name is “hidden” in the parlance of the Windows interface. If the

¹This is the Integrated Development Environment, including the MPLAB C18 C compiler.

“.c” is not hidden, you likely have a file named “hello.c.c” and this will cause confusion.

Double click on the EiC application to start it. The command line window for EiC will appear with copyright information, followed by a prompt “EiC1>”, shown in green in this manual. At this point, you can read in the file and execute the `main()` function using the interaction with the command line interface shown below. In this manual, prompts from the EiC interpreter are shown in green, your input is shown in black, values of expressions reported by EiC are shown in blue, and program output produced by `printf()`, or other output calls, is shown in red. Additionally, places where you need to hit the “enter”, or “return”, key on your keyboard are shown with `[enter]`. We will follow these conventions in all of the example interactions with the EiC interpreter shown in this manual.

```
EiC1> #include "hello.c" [enter]
      (void)
EiC2> main(); [enter]
Hello World!
      0
EiC3> :exit [enter]
```

Lets review what we have learned in this exercise. We can use notepad to edit a C program, saving it in a file located in the C:EiC folder, with the “.c” in the file name hidden. These files are loaded into the EiC interpreter using the `#include "FILE.c"` statement, where FILE is the visible portion of the name of the file. The C program itself might also use an include statement. The one in this example uses “<” and “>” instead of quotes to delimit the name of the file to include. The difference here is that the file is looked for in the “include” sub-folder of the EiC folder, this being a standard location for header, “.h”, files that provide declarations for functions available in the system library. Once we loaded the program with the include statement, the function that was defined by loading the program was called with the

```
main();
```

statement that we typed. At this point, the `main()` routine was called, printing the greeting and returning the numerical value, 0.

The curious might explore further. You might consider changing the “0” appearing to the right of the “return” keyword to some other number, deleting the “\n”, or making some other change to the text within the quotes

in order to see what happens to the behavior of the program. What is going on will become clear as you make your way through this manual. You will have to redo the include statement and the call to main in the EiC terminal window after editing “main.c” in order for you to see the effect of making changes to the file.

4 The C Programming Language

We describe a subset of the C programming language that is found to be useful in writing a program to control your robot. The discussion is intended to be formally correct, but is often simplified in order to avoid going into a lot of detail that isn't useful in robot programming. One strategy for learning this material is to skim it, and then refer back to it for a more detailed understanding as you go through the robot programming examples that follow in later sections.

4.1 The Composition of a C Program

A C program consists of a number of modules produced when the C compiler processes the source files. In EiC there is only one module, the “file” which is being input in the terminal window. A module of a C program consists of:

- type definitions,
- data declarations,
- data definitions,
- function declarations, and
- function definitions.

What these things are will become clear as you work your way through this manual. The components of a C/EiC program may be mixed in any order, given the following rules:

- types must be defined before they are used,
- variables must be declared/defined before they are used,

- functions must be declared before they are used in a function definition, and
- functions must be defined before they are called.

4.2 Integer Types and their Properties

There are a number of pre-defined types in C. We will make use of the types that correspond to integer values. In Table 1 we list the standard integer types available in the C programming language and their properties in the implementation provided by EiC.

Type Name	Bits	Bytes	Minimum Value	Maximum Value
char	8	1	-128	127
signed char	8	1	-128	127
unsigned char	8	1	0	255
short	16	2	-32768	32767
unsigned short	16	2	0	65535
int	32	4	-2147483648	2147483647
unsigned int	32	4	0	4294967295
long	32	4	-2147483648	2147483647
unsigned long	32	4	0	4294967295

Table 1: Integer types provided by EiC

The C programming language does not completely pin down the characteristics of these types and this immediately leads to portability issues between different implementations. In Table 2 we list the integer types provided by the C18 compiler that we use to prepare programs for the robot. There are significant differences, vis-a-vis EiC.

Whether the `char` type is signed or not is not pre-ordained in the C programming language. This decision is left up to the implementation. The `char` type is signed in both of the implementations of C we will deal with and this has become somewhat standard. Note that the `int` type does not have the same width, or range of values, in the EiC and C18 compilers, but the `short` type does and offers a 16-bit wide integer. The `long` type also has the same characteristics in the two implementations, offering a 32-bit integer.

Type Name	Bits	Bytes	Minimum Value	Maximum Value
char	8	1	-128	127
signed char	8	1	-128	127
unsigned char	8	1	0	255
short	16	2	-32768	32767
unsigned short	16	2	0	65535
int	16	2	-32768	32767
unsigned int	16	2	0	65535
short long	24	3	-8388608	8388607
unsigned short long	24	3	0	16777215
long	32	4	-2147483648	2147483647
unsigned long	32	4	0	4294967295

Table 2: Integer Types Provided by the C18 C Compiler

By the time you go through the exposition of binary and twos-compliment signed arithmetic in this manual, you will understand how the range of values is a consequence of the number of bits available in each type, and whether or not it is signed.

To add to the vagaries of integer types, the C implementation for the robot controller supports the `short long` type, in both signed and unsigned varieties, this being a 24-bit wide integer. The amount of memory available for data in the robot controller is quite restricted and you might find this type useful in squeezing memory a bit, if you need to.

There is no doubt that your head is spinning at this point, and we have only covered the integer types of the C programming language, along with some of their vagaries. This information is quite important because your program will produce un-predictable results if you exceed the limits of the types of the variables you declare or define. Knowing the possible range of values the variables in your program might need to deal with and arranging that the integer types chosen accommodate these needs is an important part of your programming task.

Let us turn now to using EiC to learn how to declare/define variables and try some simple executable statements that explore some of these issues. Start EiC by double clicking on its icon located at `C:EiC/EiC`, or on a desktop shortcut if you created one. In the displays that follow, prompts

and output produced by EiC are green, your code input is in black, and output produced by print statements is in red. The token [enter] refers to your hitting the return, or enter, key on your keyboard. As before, the special “:exit[enter]” input causes EiC to exit and the window to close. Lets start with some variable declaration/definitions in different widths, and some executable statements that explore simple arithmetic. The strings delimited by /* and */ are C style comments. Don’t type these into a terminal window to EiC as it will happily ignore them, but it is very important for you to comment your C programs so that they can be understood by others, and even yourself later on.

```
EiC1> char a = 127; [enter] /* Define a, with the max value for a char. */
      (void)                /* The “definition” had no value. */
EiC2> a = a - 1; [enter]  /* Subtract 1 from a. */
      126                  /* The value of the expression a-1. */
EiC3> a; [enter]
      126                  /* The value was held properly by a. */
EiC4> a = a + 2; [enter]
      128                  /* Note that a+2, an expression, exceeds a’s range! */
EiC5> a; [enter]
      -128                 /* Exceeding a’s range has nasty consequences! */
EiC6> :exit [enter]      /* Type this when you want the window to close. */
```

In C, the default type for an expression is an `int` and this is the reason the value 128 was output for the value of the expression “a+2” for input line EiC4. The EiC interpreter then truncated the value to 8 bits when storing it in “a”. When “a” was reloaded from memory and interpreted as a `signed char`, the value -128 was the result.

Exactly what happened will become clear when we explore binary arithmetic later on, but it is quite clear that one must be careful about the range of values that various types support when programming a robot. The power applied to a motor is controlled with an `unsigned char` value. If you assign it with a value outside the range of an `unsigned char` you will see un-predictable results. You will have created a hazard for the robot and the people around it.

4.3 Variable Definitions and Declarations

Let us examine a few variable definitions in order to ascertain how they work:

```
unsigned char i = 7;
int j;
unsigned int k = 346;
int l = -457;
```

Most of these definitions contain an initial value for the variable, except for the one defining “j”. When a variable is defined outside of a function and is not explicitly initialized, its initial value is zero. When an expression is used to initialize such a variable, it just be a “constant expression” that can be evaluated with the code is compiled.

The reader may have noted, at this point, that the ‘;’ is ubiquitous in the C programming language. The C programming language is not sensitive to white space, that is, the compiler does not care if it sees a space, a tab, newline or carriage return. Given this situation, the ‘;’ character is used to tell the compiler that the end of the statement has been reached.

The reader may notice that we have not shown any data declarations. A data definition indicates the type, name, and possibly the initial value of a variable. It causes the storage associated with the variable to be allocated and initialized. If no initial value is provided, and the data definition occurs outside of a function, the initial value is zero. If no initial value is provided, and the data definition occurs inside a function,² the initial value is undefined. A data declaration only tells the compiler the type and name of a variable, but does not allocate storage or initialize its value. It is expected that the associated data definition will be found elsewhere in the file, or in another file. If there is no data definition for a given variable, an error will result when the loader attempts to link the modules together to build the program.

An example of a data declaration is

```
extern int i;
```

where the compiler is being told that “i” is an integer, but its definition is to be found somewhere else. A second example is the declaration of arguments in a function definition that we cover in the next section.

4.4 Functions

We need to learn how to declare and define functions, also known as sub-routines, and we need to develop an understanding of how the position of a

²We ignore the `static` storage class modifier for now.

data definition affects its properties. Let us consider, first, a declaration for a function:

```
int func(char c, int i);
```

This is a function declaration, in function prototype form, and this is the only form you should be using.³ It indicates that the function `func()` returns an `int`, takes a `char` as its first argument and an `int` as its second argument. As it is only a function declaration, it does not provide the actual code associated with the function. Once the EiC interpreter, or a C compiler, sees the declaration for a function, it can construct the interface that calls it. The function must be defined using a function definition, however, for the function to actually be called in EiC. Analogously, if the function is not defined somewhere in a C program, a load time error will result when the modules composing the program are linked together.

It is through function definitions that programs are constructed in C. As an example, consider a possible definition for the function, `func()`, declared above.

```
int func(char c, int i) {
    int j = 7;
    j = j + i + c;
    return j;
}
```

First, the declaration of arguments in the function definition, inside the `()`, must be consistent with the declaration of the function that might have occurred earlier. Second, it is important to understand that the declared arguments of the function are just that, declarations in the sense described above. The code that calls the function arranges that the data passed to it is allocated and initialized, via the expressions placed in the argument list when the function is called. The code for the function starts with the opening brace, “{”, and ends with the closing brace, “}”.

As with any set of statements within a pair of braces, the first of these may provide definitions/declarations for variables, followed by executable statements. Any local variables⁴ defined here exist only while the function is active, and thus are re-allocated and re-initialized every time the function

³The use of variable names in the function prototype is superfluous, but can be of mnemonic assistance to the programmer.

⁴Ignoring the `static` keyword for now.

is called. The `return` causes the value of the expression to its right to be returned to the caller.

4.5 Data Hiding

As an example of calling a function from within a function, and to illustrate the associated data hiding issues, consider the following:

```
int l = 7;
int foo(int r, int s) {
    int l = 9;
    s = s + 7;
    return l + r + s;
}
void bar(void) {
    int i = 7;
    int j = 8;
    l = i + j;
    l = foo(l, i+j);
}
```

First, you will notice that there is no separate function declaration for `foo()` in this case. The compiler has seen the function definition for `foo()` before it is used in `bar()`, so it knows how to process the arguments and handle the returned value. The function declaration is required only if the compiler has not seen the definition prior to seeing the call. You will also notice that the argument list and returned value for `bar()` is `void`, indicating that it takes no arguments and returns no value.

You will note that there is a global variable, “`l`”, and a local variable defined in `foo()`, also named “`l`”. The local one hides the global one as far as the code in `foo()` is concerned, in this instance. In this case, `foo()` adds 7 to “`s`”, then adds the value of the local “`l`” to the values of “`r`” and “`s`”, its arguments, and then returns the result. In the call to `foo()`, within `bar()`, the value (a copy) of the global “`l`” is provided as the first argument, this becomes “`r`” within `foo()`. The value of `i+j` is provided as the second argument, and this becomes “`s`”. Blocks of code may be nested arbitrarily and new data definitions/declarations within a nested block will always override the data definitions/declarations from outside.

4.6 Expressions and Operators

The expression is ubiquitous in the C programming language. Expressions can be used to initialize variables, although a constant expression (one involving only constants that can be calculated at compile time) must be used to initialize a variable that is defined outside of a routine. An expression may be enclosed in parentheses, (), and there are precedence rules for operators that enforce order of evaluation if you do not use them. The set of operators available for the construction of expressions in C is rich, and highly compact (cryptic at first sight).

4.6.1 Arithmetic Operators

Let us consider some examples of the relatively standard operators for arithmetic. In Table 4.6.1, we show the simple arithmetic expression operators available in C. We show these operators with variables, but any expression can be used for both the right and left hand side.

Expression	Value
k	value of k
-k	minus k
k + l	sum of k and l
k - l	k minus l
k * l	k times l
k / l	k divided by l
k % l	remainder of k divided by l

Table 3: Arithmetic Operators in C

4.6.2 Conditional Operators

Conditional code is supported with a set of relational operators, shown in Table 4, that can be used anywhere an expression can be used in the language. As was the case for arithmetic operators, any expression can be substituted for the variables used.

At this point, one can demonstrate the power available in C to express complicated expressions, that perhaps seems cryptic at first sight.

Expression	Value
<code>k < l</code>	one if k is less than l, zero otherwise
<code>k <= l</code>	one if k is less than or equal to l, zero otherwise
<code>k > l</code>	one if k is greater than l, zero otherwise
<code>k >= l</code>	one if k is greater than or equal to l, zero otherwise
<code>k == l</code>	one if k is equal to l, zero otherwise
<code>k != l</code>	one if k is not equal to l, zero otherwise

Table 4: Conditional Operators in C

```
((i + (k * c)) >= (q + s)) * (q - c)
```

You should be able to work out what this expression means at this point: if $(i + (k * c))$ is greater than or equal to $(q + s)$, the result of the expression is $(q - c)$, otherwise it is zero.

4.6.3 Logical Operators

A set of logical operators, also clothed in the expression paradigm, is shown in Table 5.

Expression	Value
<code>k l</code>	one if k is non-zero, OR l is non-zero, zero otherwise
<code>k && l</code>	one if k is non-zero, AND l is non-zero, zero otherwise
<code>!k</code>	one if k is zero, zero otherwise

Table 5: Logical Operators in C

4.6.4 Assignment Operators, and Expressions

In C, the assignment operator is also an expression and has shortcuts for common idioms. Some of these shortcuts are shown in Table 6. As an assignment is an expression, its value can be used in larger expressions, producing expressions with side effects. Assignment expressions can have unusual problems that arise from the width of the type that is involved in the assignment. Consider the following code that you can easily try with EiC:

Expression	The value is the value assigned
<code>k = 1</code>	assign to k the value of 1
<code>k += 1</code>	equivalent to <code>k = k + 1</code>
<code>k -= 1</code>	equivalent to <code>k = k - 1</code>
<code>k *= 1</code>	equivalent to <code>k = k * 1</code>
<code>k /= 1</code>	equivalent to <code>k = k / 1</code>

Table 6: Assignment Operators in C

```
int i;
int j = 255;
int k = 1;
char c;
c = i = j + k;
```

After execution of this code, “c” has the value 0, while “i” has the value 256. If you study and understand Sections 6 and 6.2 of this manual, you will develop an understanding of why this happens. If the last statement is changed to

```
i = c = j + k;
```

both “c” and “i” will have the value zero. One must be careful to choose types that can hold the values of expressions that might occur in your program.

4.6.5 Bitwise Operators

Finally, operators are provided to manipulate the bits that compose an integer value. These operators are shown in Table 7.

Most of the bit oriented operators have “assignment” versions, but we won’t enumerate them. It is recommended that you try out these operators with simple variables using EiC. It will help you develop a good understanding of what they do. The shift operators are quite useful as more efficient substitutes for multiply and divide by powers of 2, but one must be careful. See Section 6 for the details.

Expression	Value
<code>k & l</code>	bitwise AND
<code>k l</code>	bitwise OR
<code>k ^ l</code>	bitwise exclusive OR (XOR)
<code>~k</code>	bitwise NOT
<code>k << l</code>	k shifted to the left by l bits
<code>k >> l</code>	k shifted to the right by l bits

Table 7: Bitwise operators in C

4.6.6 Cast Expressions

Sometimes it is required to force the compiler to handle a value as a different type in an expression in order to obtain the desired arithmetic properties. To do this, we use a *cast expression*. This takes the form $(type)$, where *type* is the desired type. An example is the need to cast a `char` argument to `printf()` in order to get the desired promotion of the value to an `(int)` when programming the robot controller.

```
char a;
printf("a = %d\n", (int)a);
```

Casts can also be used to truncate a wider type in an expression.

4.6.7 Conditional Expressions

The *conditional expression* is rarely used, but of immense value when it is required. This expression takes the following form:

```
(expr1 ? expr2 : expr3)
```

First, `expr1` is evaluated. If it found to be non-zero, `expr2` is evaluated and provides the value of the conditional expression. If `expr1` was zero, `expr3` is evaluated, instead, to provide the value of the conditional expression. A favorite use of the conditional expression is for the absolute value:

```
#define ABS(x) ((x) < 0 ? -(x) : (x))
```

4.6.8 Compound Expressions

This expression resembles an argument list for a function call, without the function name.

```
(expr1, expr2, ... , exprN)
```

The component expressions are evaluated in sequence, from left to right, and the value of the resulting expression is the last one. The compound expression can be useful if you want to produce side effects before returning the final value of the expression. Note: Function arguments do not have a defined evaluation order in the C programming language, in spite of what might be implied by the compound expression.

4.7 Conditional Code

Up to this point, we have considered only linear code where one statement is executed after another, with no changes in the flow of execution that is driven by data values. With basic expressions: arithmetic, conditional, logical and bitwise under our belts we can now consider programming constructs that affects the flow of code execution. A construct that is heavily used in robot controller code is the if-then-else statement. At its most complex, it takes the following form:

```
if(EXP1){
    BLOCK1
}
else if(EXP2){
    BLOCK2
}
else if(EXP3){
    BLOCK3
}
...
else {
    BLOCKELSE
}
CODEREST
```

First, EXP1 is computed and if it is non-zero, the declarations and code represented by BLOCK1 is executed. If this happens, the computer is done with the if-then-else construct and execution is picked up with the code

represented by `CODEREST`. Failing this, the value of `EXP2` is computed and if it is non-zero the code represented by `BLOCK2` is executed. Failing the test of every test expression, which requires a lot of computing and testing for a large number of “else if” cases, the code represented by `BLOCKELSE` is executed. Only one of the code blocks is executed. The “else if” cases may come in any number, or may not exist at all. The last “else” case is optional as well.

An instructive if-then-else case is contained in the file “tryif.c”:

```
#include <stdio.h>
void tryif(int i) {
    if(i < 0) {
        printf("foo\n");
    }
    else if(i < 2) {
        printf("bar\n");
    }
    else if(i == 1) {
        printf("spam\n");
    }
    else {
        printf("foobarspam\n");
    }
}
```

You can load this routine using the include statement listed below. Try it out in EiC, remembering that we are showing the output of `printf()` in red:

```
EiC1> #include "tryif.c" [enter]
      (void)
EiC2> tryif(-1); [enter]
foo
      (void)
EiC3> tryif(0); [enter]
bar
      (void)
EiC4> tryif(1); [enter]
bar
      (void)
EiC5> tryif(5); [enter]
foobarspam
      (void)
```

```
EiC6> :exit [enter]
```

Why does it never print spam?

4.8 The Switch Statement

Consider a situation wherein you want to build an if-then-else construct that has a large number of “else if” cases where the test is a simple comparison of equality to an integer value. For cases near the bottom, a very large number of comparisons would have to be executed, and fail, to get there. This is very inefficient. A better approach would be to put the code addresses to jump to in a table, and index it with the integer value so that you jump directly to the correct block of code to execute. In the C programming language the switch statement provides this functionality. An example switch statement can be found in the file “sw.c”, displayed below:

```
#include <stdio.h>
void sw(int i)
    switch(i) { /* Select code to execute according to the value of i. */
    case 0: /* The case labels can be regarded as "indexes" into the code. */
        printf("executing case 0\n");
        break; /* Break out of the switch. */
    case 1:
        printf("executing case 1\n");
        break;
    case 2:
        printf("executing case 2\n");
        /* No break so the next case is executed. */
    case 3:
        printf("executing case 3\n");
        break;
    default:
        printf("executing default\n");
        break; /* Optional, but good practice. */
    }
    /* A break out of the switch lands you here. */
}
```

This file defines a function, `sw()`, that takes an integer argument and executes a switch statement using the value. The switch statement is opened with “`switch(i){`”, switching on the value of “`i`”, so to speak. Each occurrence

of “case %d:” delimits a block of code for which execution is indexed using the expression in the parentheses. The “break” statement causes execution to break out of the switch statement, otherwise execution falls through the next “case”.

Let us try sw() using EiC:

```
EiC1> #include "sw.c" [enter]
      (void)
EiC2> sw(0); [enter]
executing case 0
      (void)
EiC3> sw(1); [enter]
executing case 1
      (void)
EiC4> sw(2); [enter]
executing case 2
executing case 3 /* Remember that missing break statement? */
      (void)
EiC5> sw(3); [enter]
executing case 3
      (void)
EiC6> sw(10); [enter]
executing default
      (void)
```

The switch statement is very useful when coding a state machine to control a robot during autonomous operation. We will return to this topic in Section 21.

4.9 Looping

The construct “while (*expression*) { *block of code* }” keeps executing *block of code* while *expression* continues to be non-zero. As an example, consider the code in the file “trywhile.c”.

```
#include <stdio.h>
int a = 0;
void trywhile(void){
    while(a < 4) {
        printf("a = %d\n", a);
```

```

        a += 1;
    }
}

```

Try it out using EiC:

```

EiC1> #include "trywhile.c" [enter]
      (void)
EiC2> trywhile(); [enter]
a = 0
a = 1
a = 2
a = 3
      (void)
EiC3> a; [enter]
4

```

It would be instructive for you to try changing the initial value for “a”, the value used in the comparison, and the value used in the increment, and test the behavior of the output.

A particular “while” loop has a prominent role in the robot controller, this being a loop of the form “while(1) { *a really big block of code* }”. What does it do?

Another looping construct that might be useful is the “for” loop. This looping construct takes the form “for(*expr1*; *expr2*; *expr3*) { *block of code* }”. The loop starts off by evaluating *expr1*. The *block of code* is executed repeatedly, as long as *expr2* is non-zero. At the end of each execution of the block of code, *expr3* is evaluated and it is useful if this expression has a side effect on the loop index. As a simple example, consider “tryfor.c”.

```

#include <stdio.h>
int a;
void tryfor(){
    for(a = 0; a < 4; a += 1) {
        printf("a = %d\n, a);
    }
}

```

This loop prints exactly the same thing as “trywhile.c”, above. The advantage of the for loop is that it places the loop control statements in one place where they can easily be found and understood. Each style of loop has its useful role in a C program.

4.10 Pointers

The example in Section 11 uses a special operator, “&”, to pass the address of a variable to a routine. The C programming language supports the concept of pointers, the simplest use of them being to pass the address of a variable to a function so that the function can modify it. The address operator is “&”, and the corresponding indirection operator is “*”. The “*” operator is also used for multiplication. The C compiler sorts out the correct interpretation from the context.

As an example of passing an argument by address, consider the code example below:

```
void bar(int *iptr) {
    *iptr = 7;
}
void foo(void) {
    int i = 6;
    bar(&i);
    /* "i" now has the value 7 */
}
```

The argument to the routine `bar()` is declared as “`int *iptr`” which means that “`int *iptr`” is an `int`, or that “`iptr`” points to an `int`. When “`int *iptr`” is assigned, the `int` data location that “`iptr`” points to is being set. In the call to `bar` the argument “`&i`” is passed. The value of the expression, “`&i`”, is the address of “`i`”.

This is the limit for pointers in this manual, but pointers have a very significant role in the C programming language and allow you to create and manipulate very complex data structures. You are encouraged to learn more about them from one of the books listed in the references.

4.11 Arrays

Although rarely used to program a robot, the C programming language provides the concept of an array of data that can be indexed by an integer value. The character strings passed as the first argument for `printf()` are a special form of an array of `char` values, known as a null terminated string, with the “expression value” of such a “string constant” being a pointer to the first `char` in the string. An array can be explicitly defined as follows:

```
int iarray[10];
```

In this case an array of 10 `int` values has been defined.

Individual elements of the array are accessed with the expression “`iarray[i]`”, where “`i`” is an `int` value used to index the array. An index value of 0 provides the first element of the array. In this case, an index value of 9 provides the last element of the array. Incorrect index values are a runtime error in a C program. The name of the array, itself, provides a pointer to the first element of the array.

Arrays are rarely used in code for the robot controller, but one place we will use them is in developing a function to format `long` types into decimal format for printing. The default `printf()` routine for the robot controller does not support output of the `long` type in decimal format, so this is something that we must write code for. You also might need to provide a special output function to print integer values that represent “fixed point” values. We will cover this in Section 8.3.

4.12 Character Constants

Printable characters, and some non-printable ones, are represented as character constants. These are simply special representations for integer values, so that you can easily see what will appear on the console terminal, or in a file, when the corresponding integer value is printed as a single character. The integer value corresponds to the binary code that is signaled on the cable connecting the RC to the console computer, for instance. A character constant is usually a single character, enclosed in single quotes. Examples are ‘`a`’, ‘`b`’, and ‘`c`’. There are a few character constants that start with the ‘`\`’ character that are useful, these being the tab, ‘`\t`’, the newline, ‘`\n`’, and the very special character, ‘`\0`’, the numeric value of which is zero. To develop a clear understanding that these are just integer values, try the following in EiC:

```
EiC1> 'a';[enter]
      97
EiC1> 'b';[enter]
      98
EiC1> 'c';[enter]
      99
EiC1> '0';[enter]
      48
EiC1> '1';[enter]
```

```
49
EiC1> '2';[enter]
50
EiC1> '3';[enter]
51
EiC1> '\t';[enter]
9
EiC1> '\n';[enter]
10
```

The ordering of the integer values corresponding to '0' through '9' and 'a' through 'f' should give you a hint as to how the `printf()` routine produces formatted output.

4.13 Type Qualifiers

The C programming language offers two ways to tell the compiler that there is something special about a given variable, offering the compiler some information that is relevant for optimization purposes. The first of these is the `const` type qualifier. The keyword is placed just before the type in a declaration or definition of the variable. This keyword tells the compiler that the value of the variable will not be changing, so that suitable optimizations can be employed. The compiler also warns if it encounters code that attempts to modify the variable.

The second, and most important type qualifier for our purposes, is the `volatile` keyword. This keyword, also placed ahead of the type in a declaration or definition, informs the compiler that the contents of the location may change spontaneously, so that it does not optimize away references to the variable, although there is little risk of this in our robot controller for anything wider than a `char`. The `volatile` type qualifier is recommended for global variables that are updated in a routine that is used to service an interrupt, and is good practice, regardless of any specific knowledge you might have with regard to the behavior of the C18 C compiler and the target computer architecture it is compiling code for.

5 Storage Class, Scope and Linkage

Until now, we have kept our discussion with regard to how the properties of variables are derived from the locations of their declarations and definitions highly simplified. It is useful at this point to discuss more formally the concepts of *storage class*, *scope* and *linkage* for variables and how declarations and definitions can be modified in order to influence these properties. The requirement for understanding these issues cannot be avoided because programs for the robot controller are implemented using more than one source file.

A variable is a location in memory, the interpretation of which depends upon its two main attributes: its *type* and its *storage class*. The type infers the number of bytes of memory that are associated with the variable and how the compiler interprets these bytes as a value during program execution. The storage class determines the lifetime of the memory allocation associated with the variable.

There are two storage classes: *automatic* and *static*.⁵ A variable with the automatic storage class exists only during the time period that the block of code containing the definition for the variable is active. The compiler allocates the memory associated with an automatic variable when the block of code containing the definition is entered, and is then free to recycle this memory as soon as the block of code containing the definition is exited. A variable in the automatic storage class is initialized each time the block of code containing the definition for the variable is entered. Because of this, a variable with the automatic storage class can be initialized with any expression, and lacking an initialization the variable will contain whatever garbage recycled memory might contain.

A variable with the static storage class is allocated in a persistent memory location that exists outside of any function call and is initialized only once at the start of program execution. It is never recycled by the compiler. Because of this, a variable with the static storage class can be initialized only with a constant expression that can be evaluated at compile time. Lacking an initialization, a variable with the static storage class will contain zero at the start of program execution.

So far, in Section 4.3, we have indicated that a variable defined outside

⁵Not to be confused with the `static` keyword, one way to allocate data with the *static* storage class.

the definition of a function is initialized once, with a constant expression at compile time, and that the variable is persistent throughout the execution of the program. This corresponds to the static storage class. We have also indicated that variables defined inside a block of code are not persistent between function calls and are initialized with any provided expression each time the block of code is entered. This corresponds to the automatic storage class.

A variable name also has a *scope*, the region of the program in which it is known, and a *linkage* that controls whether or not a variable by the same name in another scope refers to the same storage location. The scope of a definition or declaration starts at the line below where it occurs, and continues until the closing brace of the block the variable is declared or defined in, or the end of the file for variables defined outside of a function. The scope of the argument declarations for a function is from the opening brace of the function definition, to the closing brace. As noted earlier, new declarations or definitions of variables having the same name, within a contained block of code, override declarations and definitions from outside the block, until the closing brace of the block they occur in.

5.1 Storage Class Modifiers

There is an assortment of storage class modifiers available in the C programming language, not to be confused with the storage classes themselves, that influence these properties:

- **auto** indicates the automatic storage class. This is the default for variables defined inside a function. Since this is the only place that this keyword can be used the *auto* keyword is completely redundant, and should not be used.
- **register** is a programmer hint that the variable will be heavily used and that the compiler should make an attempt to put it in a machine register. It also indicates the automatic storage class, and the programmer is making a promise not to take the address of the variable. The register declaration is of little use on an eight bit micro-controller that has just a few registers. The register allocation algorithms of optimizing compilers are much better than user hints, these days, and have removed the need to use this storage class modifier for the purpose of optimization. The promise that the address is not going to be taken,

checked by the compiler, may have value when you are debugging your program.

- `static` defines a variable in the static storage class. When used outside of a function, the variable can not be linked to from another file. Inside a block of code, the variable it defines is still in the static storage class but has a scope local to the block of code it is defined in. A function may also be declared with a `static` storage class modifier, denying linkage to the function from outside the file that it is defined in.
- `extern` is used to declare a variable that has linkage to a definition for a variable with the same name, occurring outside of any function either in the same compilation unit (file), or in another. Such a variable has the static storage class.

As an example of the use of storage class modifiers, consider the following:

```
extern int h;
static int i;
void foo(void){
    static int j = 7;
    int k = 5;
    extern int l;
}
```

The definition for “h” is to be found elsewhere and linked to, and the linker will report an error if it is not found. Because of the `static` storage class modifier, “i” can not be linked to outside of the file containing its definition. The variable, “j”, can only be seen within the function `foo` and starts the program execution with the value 7. Because of the `static` storage class modifier, if the function `foo()` were to modify the value of “j”, it would see the modified value, and not 7, the next time `foo()` was called. The variable, “k”, gets freshly allocated and initialized each time `foo()` is called, and would contain garbage upon entry to `foo()` if it were not explicitly initialized. Finally, because “`extern int l;`” occurs inside a block of code, the scope of the name “l” is only within this block of code.

Historically, multiple definitions (outside of functions) of the form “`int i;`” (without the initialization) were allowed in the C programming language, and still are today. This wart in the language is patched up using the notion of the *tentative definition*, discussed in the **Glossary**. It is also the case that “`extern int i = 5;`” was accepted outside of a function as if it was “`int`

`i = 5;`”, historically, and still is today. This usage draws a warning with a modern C compiler.

Additionally, there is the quite serious issue of consistency of declarations of variables and functions involved in linkages between modules that can be the source of hard to find program bugs. It is recommended that all of these problems be sidestepped by rigorously employing the following conventions:

- If a variable, or function, is not intended to be linked to from outside the file in which it is defined, use the `static` keyword in order to prevent such linkage. Linkage without properly setting up consistent declarations in a header file is a frequent source of errors.
- If linkage from outside the module is intended, make sure that the variable is explicitly initialized, for example, by using `int foo = 5;` to force a definition. Put a declaration, `extern int foo;` for the variable in a header, “.h”, file that is included in the file containing the definition, and include it in any file requiring linkage to the variable. The purpose of including the header file in the file containing the definition is to have the compiler check the consistency of the declaration with the definition, preventing problems arising from any inconsistencies.
- For a function, always use a function prototype in the definition, and make sure that a function prototype is used in a header file that is intended for use by any file requiring linkage to the function definition. Include this header file in the file containing the function definition so that the compiler can check consistency of the declaration with the definition.

By following these guidelines, the compiler will let you know about errors that you would otherwise spend many hours debugging. You can make much better use of this valuable time by adding new features to your robot program!

6 Binary

6.1 Positive Integers

Integer values are represented by the computer in binary. By understanding binary we can understand how computers do arithmetic, what the shift operators do in C, and how shifts relate to division and multiplication by

powers of two. Our more practical goal is to understand how to exploit the binary implementation of integers in order to perform fixed-point arithmetic. This form of arithmetic is useful when we want finer precision than 1, but do not want to pay the high software overhead associated with floating point arithmetic on an 8 bit micro controller.

We use “base 10” for our arithmetic because we have 10 fingers, and the fact that the tables involved for multiplication and division in this base are easily memorized by the average person. A large number, such as 3579, is interpreted with the place value number format as follows:

$$3579 = 3 \times 10^3 + 5 \times 10^2 + 7 \times 10^1 + 9 \times 10^0$$

The place value number format, the existence of a symbol for zero, and long division are the corner stones of decimal arithmetic as we know it. Early mathematics used base 60 arithmetic because this base is evenly divided by 2, 3, 4, 5, 6, 10, 12, 15, 20 and 30. This is the reason why we measure time and angles in the way we do today. Base 12 was also quite popular because this base is evenly divided by 2, 3, 4 and 6. This is the reason why the number twelve is popular in the English system of measurement. This is the legacy of a period when long division had not yet been invented, sophisticated mathematics was the domain of elite scholars, and serfs only needed to divide up what was for dinner.

Computer logic is either on or off. This can be used to provide a representation of numbers in binary. Computers perform their arithmetic in the binary, or base 2, number system. A large number, such as 101101, is interpreted with the place value number format in an analogous way as it is done for decimal arithmetic.

$$101101 = 1 \times 10^5 + 0 \times 10^4 + 1 \times 10^3 + 1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$$

No doubt you are screaming at the techie humor now, as there are indeed 10 kinds of people in the world, those who understand binary, and those who do not!

$$101101 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

At this point, the reader should be able to calculate the range limits for N-bit unsigned binary numbers, and we suggest that you go through this exercise. The answer is that an N-bit unsigned number has the range 0 through $2^N - 1$.

We can easily multiply/divide a decimal number by 10 by shifting it to the left or right by one digit. A quick inspection of the binary number system demonstrates that we can multiply/divide a positive binary number by 2 by shifting it to the left/right by one bit. This technique is often used in order to obtain more efficient substitutes for multiply and divide by powers of two, especially on platforms such as our PIC micro-controller where multiply, and particularly divide, are expensive. We must remember, however, to be careful with negative numbers.

6.2 Twos-Compliment Signed Arithmetic

A universal goal in the development of computer hardware, from the vacuum tube days to the highly integrated circuits of today, has always been to get the most functionality possible out of the logic hardware that you put into the computer. The binary description of numbers above is directly implemented with logic gates. A computer would be useless, however, if it did not deal with negative numbers.

One approach to negative numbers is to dedicate a bit in the variable to indicate whether the number is negative or not; as much is done for the IEEE floating point data representation in computers today. There is however, a much more elegant and efficient solution available for integers. It has become the standard for modern digital computers.

Twos-complement signed arithmetic is founded in a very formal understanding of what a negative number is. Like the case for the place value representation of numbers, you may take this for granted! What is a negative number? It is the number that you add to a given positive number that provides zero as the result. Additionally, the negative of zero must be zero. To this end, if we were to find a transformation for a “positive” binary number, the result of which produced zero when it was added to the original; we would have a negative number, wouldn’t we? We would, provided that this same transformation performed on zero results in zero.

This is possible within the context of an unsigned binary adder, and a bitwise not operator, but we must learn to accept overflow as our friend. It is likely that this is the one and only case where overflow is our friend on a computer!

This magic operation is called the twos-compliment operation. It goes as follows:

1. invert the bits of the binary number: mapping 1 to 0, and 0 to 1
2. add 1 to this intermediate result, letting any overflow drop on the floor

```

(original number)  00101101
(invert the bits)  11010010
(add one)         11010011

```

If you are having trouble with this, 1+1 is 10, that is “0” and then you carry the “1”; 1+1+1 is 11, that is “1” and you carry the “1”. Now, observe what happens if we add the twos-compliment transformed number to the original, with our unsigned binary adder, again embracing overflow as our friend:

```

00101101
+11010011
100000000

```

Remembering that we have an eight bit binary adder in this case, and that last carry to bit nine gets dropped on the floor, we have a number that when added to the original gives zero. If the original number is 45, and it is, the twos-compliment transformed number must be -45, if it is interpreted in eight-bit twos-complement signed arithmetic. Applying the twos-compliment operation to 0, you find that it produces 0, giving us a viable representation of signed arithmetic in an eight-bit format.

There are 2^N possible values for a binary number with N bits, and 0 is mapped to itself by the twos-compliment operation. This leaves an odd number of values remaining, so there must be something special about at least one other value. For the eight-bit case, this value is:

```

10000000
(invert the bits)  01111111
(add one)         10000000

```

We see that this value is transformed to itself by the twos-compliment operation, so it can’t have a corresponding value of opposite sign expressed in 8 bits. What value is it? Let us add one

```
10000001
```

then apply the twos-compliment operator to change the sign of this, as yet, unknown value:

```

1000001
(invert the bits)  0111110
(add one)         0111111

```

This number is 127 as far as our unsigned binary adder is concerned. Given this fact, the unknown value that has no opposite-signed companion expressible in eight bits must be -128. We now have the result that a `signed char`, an eight-bit twos-compliment signed integer, can handle values in the range from -128 to 127. If we add one to 127, we get an abrupt jump to -128.

If 01111111 is the largest positive integer value for an eight-bit twos-compliment signed integer, then all of the positive values have a 0 in the high order bit, and all of the values with a 1 in the high order bit are negative. This is a general feature of twos-compliment signed integers, making it easy to test for a negative value.

Let us return to shifts and the equivalence between left/right shifts and multiply/divide for unsigned values. Does this apply to twos-compliment signed integers? The answer seems clear for positive values, as long as a left shift does not move a 1 into the high order bit, or past it; and for right shifts in general.

What about the case for negative numbers? The situation for left shifts can be studied by applying the twos-compliment operation on the successive values: 1, 2, 3, 4, 5, 6; shifting the result to the left one bit; and then applying the twos-compliment operator again to see what negative value was produced by the shift. The bit pattern characteristic of the higher order bits of a negative number becomes clear, indicating the condition where things would break down. The result is that a left shift applied to a negative number does multiply by two, as long as no 0 bits are shifted into, or through, the high order bit. By employing a suitably wide integer format, this is not a problem. The same risk would exist for any integer, it just occurs a little earlier for a signed one.

What about right shifts applied to negative twos-compliment numbers? First, we need to understand just what the C compiler does with such an operation, and then we need to understand what the consequences are. For this, we look ahead to Section 8.3, and employ the `pbchar()` routine located in the file “pb.c”. Try the following using EiC:

```
EiC1> #include "pb.c" [enter] /* Load formatting functions for binary. */
      (void)
EiC2> char a = 1; [enter] /* What does 1 look like in binary? */
      (void)
EiC3> pbchar(a); [enter]
00000001
      (void)
```

```

EiC4> a = ~a; a += 1; [enter] /* Apply twos-compliment by hand. */
      -1                      /* We got the expected result. */
EiC5> pbchar(a); [enter]      /* Display -1 in binary. */
11111111
      (void)
EiC6> a <<= 1; [enter]
      -2                      /* Left shift is multiply by 2. */
EiC7> pbchar(a); [enter]
11111110
      (void)
EiC8> a <<= 2; [enter]
      -8                      /* Left shift by two bits is multiply by 4. */
EiC9> pbchar(a); [enter]
11111000
      (void)
EiC10> a >>= 1; [enter]
      -4                      /* The result of right shift by one bit is attractive. */
EiC11> pbchar(a); [enter]
11111100
      (void)
EiC12> a = -7; [enter]        /* Check right shifts for a negative number. */
      -7
EiC13> pbchar(a); [enter]
11111001
      (void)
EiC14> a >>= 1; [enter]
      -4                      /* Ouch! 7/2 is 3 . */
EiC15> a = -3; [enter]
      -3
EiC16> pbchar(a); [enter] /* It is as if the "1" bit should not have shifted off. */
11111101
      (void)

```

The incorrect results for the right shift of negative numbers stems from the “1” bits that are being shifted off to the right. The general solution for employing shifts to divide a negative number by two is to first change its sign, shift it the required number of bits to the right, and then change the sign back so that it is again negative.

If you understand the last two sections, you have begun to understand binary. There are further tricks employed to enable the use of an unsigned multiplier to deal with signed integers, and the C implementation for the robot controller does this in order to use the 8 bit unsigned multiplier to handle multiplies for all of the C data types, but we will leave this, and more, to your college course in computer architecture.

6.3 Fixed-Point Arithmetic

In this section, we will describe an approach to handling binary fixed-point arithmetic within the confines of integer arithmetic using the C compiler. The industrial strength version can be found in reference [5].

It is advantageous to employ fixed-point arithmetic for those situations where you need finer precision than 1, but do not want to pay the costs associated with the software floating point libraries offered by the C18 C compiler. Software floating point runs slowly and the extensive library support required can take up a lot of precious code space. In addition to this, support for formatted output of floating point types to the console is missing and there is little sense in using a data type that you can't adequately monitor on the console.

To understand how we can implement binary fixed-point arithmetic, let us return to the binary representation of integers, 101101 being our universal example.

$$101101 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

The computer just sees this as an integer. Because of this, it provides add, subtract, multiply, and divide operations.

Your code, however, is free to interpret the integer any way that you like as long as you take care of any consequences with regard to any imagined factor. Suppose, then, that we imagine a red “binary point” placed among the bits as follows: 101.101. What does it mean to interpret the binary representation of the integer in this manner?

$$101.101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

The red ‘+’ is being used to indicate where the binary point is placed. The portion of the number to the right of the binary point is now understood to be 5/8. By interpreting the integer as “a number of eighths” we can keep

track of values with a precision of one eighth. One might think that this is not much of an improvement, but by using the `long` type we have enough extra bits to use in this manner that we can perform tasks such as integrating a rotational rate gyro with significantly improved accuracy.

Our integer type is interpreted in eighths, or any precision you choose within the available number of bits. That the “add” and “subtract” operations provided by the computer have no consequences at all can be confirmed with simple algebra.

$$\begin{aligned} i/8 + j/8 &= (i + j)/8 \\ i/8 - j/8 &= (i - j)/8 \end{aligned}$$

The computer happily adds or subtracts the integers, and the result is still in eighths.

The case for multiply is not as trivial.

$$i/8 \times j/8 = (i \times j)/64$$

The computer happily computes $(i \times j)$, but we must keep track of the fact that the result is now in sixty-fourths, and not eighths. One must divide the result by eight in order to obtain a correctly normalized fixed-point number. What we have to be careful about is using an integer type that is wide enough in order to not lose any bits “off the top” during the multiply, corrupting the final result. The expression that produces the correctly normalized result, then, is:

$$(i*j)/8$$

One can take this even further by examining the remainder and producing a correctly rounded result, but this would only be icing on the cake.

The divide has a similar issue:

$$i/8 \div j/8 = (i/j) \times (1/8 \div 1/8) = i/j$$

The computer happily computes (i / j) , but we need the correctly normalized result including any fractional part. We can normalize the result by multiplying by 8, but when we do this the resulting fractional part would be zero. Consider, as an example, $3/2 = 1.5 = \text{“001.100”}$ (in our binary fixed-point interpretation). If we used (i/j) multiplied by 8 we would get “001.000”, which is in error. What can we do about this? If we multiply `i` by 8 before we divide, our divide operation will not truncate the desired fractional part and will leave us with the correct normalized result in eighth’s:

$$(i*8)/j$$

The only issue is whether or not the multiply will cause an over-run of the intermediate value, corrupting the result. Just as was the case for the multiply of binary-fixed point, we must make sure to use an integer wide enough so that this will not happen.

This fixed point scheme, using a 32 bit wide integer type, is capable of supporting any requirements you might have for finer precision in your robot program. Depending on just how large the numbers you need to work with are, you might be able to squeeze things into 16 bits. The additional multiplies and divides by powers of two can be handled with shifts if one is careful about negative numbers.

7 Special Issues/Limits on the Robot Controller

There are a number of special issues/limits that affect programs running on the Robot Controller.

- **Stack Frame Size:** The limit for the total number of bytes allowed in local (automatic) variables is 120 bytes. This corresponds to non-extended mode, as referred to in the C18 users guide.
- **Parameter Frame Size:** The limit for the total number of bytes that may be passed as arguments to a function is 120 bytes. The stack and parameter frames are distinct.
- **Section Size Limit:** The size limit for a “section” as defined in the C18 user manual is 256 bytes. This limit applies, for instance, to the total amount of memory of the static storage class that may be defined in a single file. *Dirty tricks* can be used to build larger data structures if they are required.
- **Access to Program Memory:** Variables can be placed in program memory using the special `rom` storage class modifier. Arrays placed here can exceed the section limit of 256 bytes, but are restricted to read only use.⁶ This data does not live in the same address space as normal data, so you must proceed carefully when using addresses of `rom` data.

⁶The C18 compiler quietly accepts assignments to `rom` data, but the assignments are toothless when executed on the EDU robot controller. Additionally, the C18 compiler manual mentions functions to copy strings to/from `rom` data, but it looks like it is really

- **Little Endian Byte Order:** The bytes that represent integer values in the robot controller occur least significant byte first, and then bytes of increasing significance as the memory address is increased. This is known as “little endian” byte ordering, and is useful knowledge if you want to employ *dirty tricks*.
- **promotion in expressions:** The C programming language specifies that all narrower types in expressions are promoted to `int`, the `int` type normally being the size of a machine register. For efficiency purposes in the robot controller, expressions are handled with the widest type involved. If the correctness of your code depends upon handling of `char` values as `int` in expressions, you must explicitly cast to `int`.
- **promotion of char arguments:** The C programming language specifies that all function arguments not explicitly declared in function prototypes are to be promoted to `int` if they are of a narrower type. In our robot controller, again for efficiency reasons, arguments of the `char` type passed to functions without prototypes are passed as `char`, and must be explicitly cast to `int` if the function expects an `int`.

8 Formatted Output

The robot controller has the capability of producing formatted output that is seen on the terminal console that is open after downloading a new program. This capability is very useful for debugging your program, sorting out just which variable a given switch is wired to, and so forth. Sending data back to your computer takes time, especially when transmitted over the the radio. You should use conditional compilation to put the formatted output in place only when you need to debug your program.

8.1 The `printf()` library function

The function `printf()` is used to provide terminal output in most implementations of C, and provides console output for a program running in the robot controller while the computer is connected to the programming port. The proper declaration for `printf()`, and its associated functions, is included in

read only memory.

your program by using the `#include <stdio.h>` directive. The `printf()` function takes a string as its first argument, with occurrences of the `%` character in the string indicating the presence and nature of additional arguments. In this case `%d` indicates that an integer argument is to be found as an additional argument and formatted in decimal for output. As an example, consider the following code that you can try in EiC:

```
int i = 7;
int j = 45;
int k = -34;
printf("k = %d j = %d, and (i+1) = %d\n", k, j, i+1);
```

will cause

```
k = -34, j = 45, and (i+1) = 8
```

to be printed.

Each occurrence of `%d` causes the next argument to be interpreted as an integer and printed in decimal notation. The association of the `%d` in the formatting string, the additional argument, and the values printed in the output is shown in color. There are several useful formatting instructions:

- `%d` format an `int` in signed decimal,
- `%c` format an `int` as a single printable character,
- `%u` format an `unsigned int` in decimal
- `%x` format an `unsigned int` in hex,
- `%l` format a `long` in decimal,
- `%ul` format an `unsigned long` in decimal,
- `%lx` format an `unsigned long` in hex,
- `%s` for a null terminated string,

along with many others.

8.2 `Printf()` in the Robot Controller

The runtime environment of the robot controller provides a restricted implementation of `printf()`, the restriction being that some format specifications

are not supported. In addition to the missing format specifications, `char` arguments passed to `printf()` in the robot controller have a special issue that arises because the robot controller is an eight bit microprocessor. The proper declaration for the `printf()` is obtained by including a header file with:

```
#include <printf_lib.h>
```

The `printf()` function provided in the robot controller takes a *string constant* as its first argument. Attempts to use a string constructed in variable memory will produce (`null`) output, or undefined behavior... This is due to the fact that program memory and data memory are distinct on the robot controller, and string constants are placed in program memory. The string constant may have format specifications that cause `printf()` to locate and format additional arguments. The supported format specifications are:

- `%s`: format a *string constant*,
- `%lx`: format an `unsigned long` in hex format, using lower case,
- `%d`: format an `int` argument in signed decimal format,
- `%x`: format an `unsigned int` in hex format, using lower case,
- `%u`: format an `unsigned int` in decimal format,
- `%X`: format an `unsigned int` in hex format, using upper case,
- `%b`: format an `unsigned int` in binary format.

A notable exception for the `printf()` provided in the robot controller is the lack of support for printing a `long` in decimal format. We will address this deficiency, and also consider support for fixed point decimal formats, in Section 8.3.

Some degree of control of leading zeros and the minimum field width for the formatted output can be obtained with further modifications to the format specifications. A number interposed between the `%` and the format indicator, for instance `%4d`, indicated the minimum field width to use in formatting the number. If there is a leading zero in field width specification, for instance `%04d`, leading zeros are used to pad the minimum field width.

There is an important issue that you must pay careful attention to when using `printf()` in your program for the robot controller. Because of the

non-standard handling of `char` arguments in the robot controller, you must explicitly cast `char` and pre-defined “bit” variables to `int` when passing them as arguments to `printf()`. The required casts are demonstrated in the code snippet below:

```
char c;
int i;
long l;
printf("c = %d, p2_sw_aux2 = %d, i = %d, l = %lx\n",
      (int)c, (int)p2_sw_aux2, i, l);
```

8.3 Extending Formatted Output

The `printf()` library call provided in the default code for the robot controller does not support decimal output of 32 bit values in a format other than hex, and the EiC `printf()` library call does not support formatted output in a binary format. Additionally, if one uses fixed point arithmetic, one might want the ability to have the robot output these values on the console in a decimal format in order to support debugging. To this end, and also for the purposes of getting more practice in understanding how to write C programs that do heavier lifting, we examine functions to serve these needs.

First, lets look at the problem of formatting output in binary. Routines are provided that print integer types in binary, located in the file “`pb.c`”. For our purposes, we will examine the function `pbint()`.

```
#include <stdio.h>
void pbint(int i) { /* Print "i" in binary format. */
    int bits;
    bits = sizeof(i) * 8 - 1; /* The number of bits in "i", minus 1. */
    while(bits >= 0) { /* While there are more bits left. */
        if((1 << bits) & i) { /* Check the corresponding bit if "i". */
            printf("1"); /* Print "1" if it was a 1. */
        }
        else {
            printf("0"); /* Print "0" if it was a 0. */
        }
        bits --; /* One bit to the right for the next trip through. */
    }
    printf("\n"); /* The ubiquitous optional newline. */
}
```

This function computes the number of bits in the type, based on the number of bytes returned by `sizeof()` and the fact that each byte has 8 bits. It then picks off the value of the bits, starting with the most significant, and prints '1' if the bit is a one, and 0 if the bit is a zero. It adds a newline before returning, and this might be optional in your code that could mix binary formatted output with output generated with `printf()`. You are encouraged to explore the effects of shift operations on signed and unsigned integer types, printing the result using `pbint()`, in order to develop an intuitive understanding of what is going on.

Let us now turn to the question of decimal output for an integer. We need to be able to do this for `unsigned long` types in the robot, if we are to debug/monitor portions of the program that use this type. A strategy for formatting `long` types in decimal is to break the long up into digits, or numbers, that the provided `printf()` can handle, and this is what we will do. While we are at it, we design the interface to be useful for the task of printing fixed point binary numbers, as well as handling `signed long` types with a wrapper.

```
#include <stdio.h>

#define RLEN 10 /* Maximum decimal digits for an unsigned long. */

/* The function puln(unsigned long i, unsigned char n)
prints "i" in decimal format, using leading zeros to fill
out a minimum field width of "n" characters.
*/
void puln(unsigned long i, unsigned char n) {
    unsigned char result[RLEN]; /* Stores the result digits, as numbers. */
    unsigned char reversed[RLEN]; /* Storage for the digits as they come off. */
    char rpos; /* Index for reversed array. */
    char rindex; /* Index for result array. */
    int j;

    /* The minimum field width is one.
    */
    if(n == 0) {
        n = 1;
    }
    /* Use remainder and divide in a loop to peel
off the digits, least significant first.
    */
    rpos = 0;
    while(i > 0) {
```

```

        reversed[rpos] = i % 10;
        rpos += 1;
        i /= 10;
    }
    /* At this point the digits are in the reversed
    array and rpos records the count.  If rpos is zero,
    the argument was zero.  We must first place the
    required number of leading zeros in the result.
    */
    rindex = 0;
    while((rindex + rpos) < n) {
        result[rindex] = 0;
        rindex += 1;
    }
    /* Fill in the digits, reversing the order.
    */
    while(rpos > 0) {
        result[rindex] = reversed[rpos-1];
        rindex += 1;
        rpos -= 1;
    }
    /* This is revolting, but it beats the problem of not
    being able to pass a character string in data memory
    to printf().  */
    for(j = 0; j < rindex; j += 1) {
        printf("%d", (int)result[j]);
    }
}

```

Wrapper functions, using the function defined above to handle the heavy lifting, providing a simple interface for signed and unsigned long types are easy.

```

/* Print a signed long.
*/
void pl(long i) {
    if(i < 0) {
        printf("-");
        i = -i;
    }
    puln(i, 1);
    printf("\n");
}

/* Print an unsigned long.

```

```

        */
void pul(unsigned long i) {
    puln(i, 1);
    printf("\n");
}

```

At this point, the only useful output routine that we have not provided an exposition of is one to print a fixed binary point integer in decimal format. All of the ideas and software components required to accomplish this goal can be found in the preceding pages. The strategy is to:

1. If the number is negative, print '-' and then change its sign.
2. Shift the number to the right by the number of bits to remove the fractional part and then call `puln()` to print this result.
3. Print the decimal point.
4. Construct a mask to isolate the fractional part.
5. At this point, we have the fractional part that tells us the number of halves, quarters, eighths, or smaller as the case may be. We need the number of tenths, hundredths, thousandths; whatever resolution is consistent with the precision we started with. Supposing we want hundredths, and we have sixty-fourths, we multiply by 100 and divide by 64. The expensive operation, the divide, is handled with a shift because it is a power of 2. Using a shift to divide by a power of two only applies to a positive number, but we have already taken care of that.
6. To properly print the fractional part, you have to print it with appropriate leading zeros, but we had enough foresight to build this capability into our primary formatting function for the `unsigned long` type.

```

/* The function void pfp(long i, unsigned char bits, unsigned char digits)
prints a binary fixed point value in decimal format. The arguments are:
i, the integer holding the fixed point value; bits, the number of bits to
place after the binary point; and digits, the number of digits after the
decimal point to print.
*/
void pfp(long i, unsigned char bits, unsigned char digits) {
    long intpart;

```

```

long mask;
long powten;
unsigned char j;

/* If the argument is negative, print the sign and work
with the absolute value so that shifts give us a divide.
*/
if(i < 0) {
    printf("-");
    i = -i;
}

/* Shift the argument down to obtain the integer
part, and turn the result over to puln() to handle.
*/
intpart = i >> bits;
puln(intpart,1);

printf(".");

/* Construct the mask for the fractional part.
*/
mask = 0;          /* All zeros. */
mask = ~mask;     /* All ones. */
mask <<= bits;    /* Shift in bits zeros. */
mask = ~mask;     /* Flip to get bits ones. */

/* Use the mask to obtain the fractional part.
*/
i &= mask;

/* Convert the fraction to a decimal fraction.
*/
powten = 1;
for(j = 1; j <= digits; j += 1) {
    i *= 10;
    powten *= 10;
}
i >>= bits;

/* At this point, we print the decimal
fraction with the correct number of digits
using leading zeros.
*/
puln(i, digits);

```

```
    /* And finally, the optional newline.
       */
    printf("\n");
}
```

9 The C Pre-Processor

Some of the features of the C programming language are implemented in the context of a pre-processor. The pre-processor handles file inclusion, macro definition/expansion and conditional compilation.

9.1 File Inclusion

In the C programming language, a *file* is compiled in order to produce a module that is linked with others in order to produce the executable program. This *file*, however, rarely comes from a single file. It is produced by including additional files through the action of include statements.

The include statement has the form:

```
#include <FILENAME>
```

or

```
#include "FILENAME"
```

The difference between the two versions is that the form that delimits the file name with < > looks in system provided directories for the file, while the one that delimits the file name with " " looks in the directory where the source program was found, first, before looking in the system provided directories.

Conventionally, these files are called *header files* and they have names that end in “.h”, but this is only a convention as has been demonstrated earlier in this manual. Header files are usually used to hold macro definitions and external declarations.

9.2 Macro Expansion

You will find that the proliferation of constants in a C program can be quite a bother, especially if you need to change their values. The same constant can

be used for one purpose in a number of places in the program, while being used for an entirely different purpose in other places. Suppose you need to hunt down all of the occurrences of a given constant value, used for a specific purpose, and change them to a different value. This can be an onerous, error prone, task.

You will also find that code patterns tend to be repeated in your program. Code is much easier to write and maintain if these patterns can be coded in one place, and then used a number of times. One must be careful to use macros, however, in a way that increases program clarity, as opposed to decreasing it.

The C programming language has a built in macro pre-processor that allows you to associate names with a given constant, or code expression, capitalized names being used by convention. Once a name is defined for the purposes of the macro pre-processor, occurrences of the name later in the code are expanded to their corresponding values (via simple text substitution) before the resulting “C code” is turned over to the compiler. As an example useful to your RC code, consider the value used to stop a motor.

```
#define STOP 127
pwm01 = STOP;
pwm02 = STOP;
```

The “`#define`” statement takes a name that has not already been defined and a string to substitute for later occurrences of the name that appear in the file. Macros may be defined to provide any special values that you might find useful in multiple places in your code. This means that you can affect all such cases by changing just one line.

The macro expansion capability provided with the C programming language supports the concept of arguments. Macros may be defined with any number of arguments. As an example, consider the following macro definition:

```
#define MULT2(a,b) ((a)*(b))
```

This macro could be used in a program as:

```
a = b * MULT2(c+d,e+f);
```

and gets expanded to:

```
a = b * ((c+d)*(e+f));
```

There are several important details to note here. The macro argument, `a`, is blindly substituted with the string of characters, `c+d`, without any knowledge of the C programming language and what the programmer might be

intending to do. The situation is the same for the second argument, `b`. Understanding the precedence of multiplication over addition, you can see the trouble that you will get into if you don't use the parentheses. Use the parentheses in macro definitions for arithmetic expressions in order to avoid this hazard.

The substitution string for a macro extends until the end of the line. In the case of complicated macros, it can be useful to break this up into a number of lines, using the line continuation, “`\`”, character for macro definitions. Here is an example of this construct.

```
#define ADDTOTHREE(a,b,c,d) a += (d);\
b += (d);\
c += (d)
```

The statement:

```
ADDTOTHREE(i,j,k,l*m+q);
```

gets expanded to:

```
i += (l*m+q);j += (l*m+q);k += (l*m+q);
```

Note that the macro gets expanded as if the multiple lines in its definition were joined without spaces. The algorithm is to join the lines into one long one, and then perform substitutions. If you need the spaces, you must put them in. For even further convenience, previously defined macro definitions may be used in macro definitions.

```
#define STOP 127
#define ALLSTOP pwm01 = STOP; pwm02 = STOP
```

9.3 Conditional Compilation

The macro pre-processor also supports conditional compilation and this can be very useful in your RC program. Consider, for instance, the possibility that you might want to use lots of `printf()` calls to debug your code, but want to easily take them in and out as they will slow the execution of your RC program when they are not needed. You can cause code to be eliminated from the program based on whether or not a macro is defined. Consider the following example:

```
#define DEBUG
#ifdef DEBUG
    printf("The value of a is %d\n", a);
#endif
```

The “`#endif`” statement marks the end of the code that is controlled by the “`#ifdef`”. You can have numerous `printf()` calls sprinkled through your code for debugging and monitoring purposes, and you can cause them all to go away by changing the “`#define DEBUG`” pre-processor directive to “`#undef DEBUG`” in one place in your program.

A favorite trick of the trade is to use an “`#ifdef NOTDEF`” to comment out code that you might want to put back in later on. The controlled code can be put back in again by making the easy change to “`#ifndef NOTDEF`”. This is very useful because C comments do not nest. There is quite a bit more meat to the macro pre-processor. You are referred to one of the books in the references for the details.

10 Classic C Programming Errors

The C programming language offers a very compact and powerful way to express operations in a program. This expressive power can be quite frustrating to inexperienced programmers because of subtle differences in the meaning of statements that “look the same” to the inexperienced eye. Generally, new programmers experience the errors listed in this section and develop a programming style that avoids them by trial and error over a long period of time.

10.1 `if(expr)` without braces

An extremely difficult to find error arises from the use of an `if()`, or its accompanying `else` statement, without curly braces. The problem arises when a single statement is listed below the `if()` with an indent, and this indent tempts you to put in an additional statement thinking that it will also be controlled by the `if()`.

```
if(expr)
    a = b;
    c = d;
```

The statement shown in red is *not* controlled by the `if()`. It is executed

unconditionally. The indent is guiding you to the wrong conclusion about what the code does, and the compiler happily ignores the indent.

The rule of thumb to avoid this problem is to always keep the controlled statement on the same line as the `if()`, or use braces if the statement controlled by the `if()` is to be placed on a new line. By doing this, the correct behavior is obtained when a new statement is added. The “look” of the code matches the “meaning” of the code, and you do not spend hours correcting for this bug.

```
if(expr) {
    a = b;
    c = d;
}
```

10.2 The = and == operators are distinct

A frequent bug for novice programmers is the use of the assignment operator, “=”, when the equality operator, “==”. should have been used. Consider the following `if()` statement:

```
if(a = 0)
```

Code controlled by this `if()` statement will never be executed. It is almost always the case that the programmer intended the equality operator, “==”, even when the expression on the right hand side is not a constant, but most C compilers do not warn about this usage. You will simply have to learn to be sensitive to the error.

10.3 The bitwise and logical operators are distinct

The *bitwise and*, “&”, operator is not the same as the *logical and*, “&&”, operator. The results of

```
if(1 & 2) statement;
```

and

```
if(1 && 2) statement;
```

are very different, the first example never resulting in the execution of `statement;` while the second one always results in the execution of `statement;`. It is wise to use the logical operators as appropriate. The use of bitwise operators should always put you in the mode of thinking about what is going on with the bits.

10.4 Seeing the ++ trees in the forest

The postfix operators, ++ and --, are often used in a manner that substitutes `a++` for `a+=1`, and `a--` for `a-=1`. This can lead to errors involving the true meaning of a postfix operator, hidden much as trees are hidden in a forest. The assignment expression `b=a++` produces a different result than the assignment expression `b=a+=1`. In the first case, the value of `a` is propagated to `b`, and then the value of `a` is incremented by one. In the second case, the value of `a` is incremented by one, and then this incremented value is propagated to `b`. Failing to keep track of this difference in semantics can cause errors that are hard to find if the ++ operator is used where its postfix meaning is not important. You should restrict your use of postfix ++ and -- operators to those times when the postfix meaning is important so that their occurrence warns you to think carefully about the code. The prefix form of these operators, ++`a` and --`a`, have no utility worth their addition to the forest.

11 The Robot Packet Loop

At this point, the reader probably wonders when something akin to what goes on in the robot control program will be covered, or at least when we will tell you that we have done so. When the robot is operated by human operators, the robot controller (RC) is linked to the operator interface (OI) via a packet protocol that is transmitted by the radio modems, or through the tether cable. The RC and OI trade packets every 26.2 milliseconds, and in doing so the RC learns the values of the switches and potentiometers on the OI. It also has the opportunity to change lights and other indicators on the OI.

If the handshake provided by the packet protocol fails for some reason, (by radio interference, or by the program returning packets late), the motor outputs of the robot are stopped for the duration of the failure.

The `Process_Data_From_Master_uP()` function, or subroutine, is called in order to take care of this task. It is called by code in the RC every time a packet is available from the OI during the operator control portion of the match. The `Getdata()` and `Putdata()` calls in this function are essential. Without them the packet protocol fails and your robot stops.

```
void Process_Data_From_Master_uP(void) {
    /* Get the packet from the OI, storing the data in rxdata.
    This causes variables such as p1_y to reflect the setting of a joystick. */
    Getdata(&rxdata);
    pwm01 = p1_y; /* Set the left motor power to the value of the left joystick. */
    pwm02 = p3_y; /* Set the right motor power to the value of the right joystick. */
    if(p2_sw_aux1){
        pwm05 = 1; /* If the "winch out" switch is thrown, winch motor to full reverse. */
    }
    else if(p2_sw_aux2){
        pwm05 = 254; /* If the "winch in" switch is thrown, winch motor to full forward. */
    }
    else {
        pwm05 = 127; /* Otherwise, set winch motor to stop. */
    }
    /* Send the return packet, constructed from data stored in txdata, back to the OI. */
    Putdata(&txdata);
}
```

In our example above, there is unconditional code that simply sets motor outputs from joystick values obtained from the OI, and conditional code that sets the winch motor values using digital inputs from switches on the custom control box. This is an example of simple programming for human control of the robot. Once you have learned all of the “pre-defined” variable names that correspond to switches and potentiometers on the OI, and those for the motor and relay control outputs on the robot, this type of code is easy to write and debug.

The RC controller has its own analog and digital inputs that can be used to sense conditions on the robot and these can be used to implement limit switches for additional control of motors. A robot may have a large number of motors and relays, controlled by settings on the OI combined with a number of limit switches on the robot. The human control section of the RC program can end up with hundreds of lines of code devoted to this task. Below, we

demonstrate how you might employ limit switches in order to prevent a motor on your robot from going too far.

```
void Process_Data_From_Master_uP(void) {
    Getdata(&rxdata);
    if(p2_sw_aux1 && rc_dig_in01){
        pwm05 = 1; /* Only if the limit switch on RC digital input 01 is open. */
    }
    else if(p2_sw_aux2 && rc_dig_in02){
        pwm05 = 254; /* Only if the limit switch on RC digital input 02 is open. */
    }
    else {
        pwm05 = 127; /* Otherwise, set motor to stop. */
    }
    Putdata(&txdata);
}
```

Note that the variables associated with switch inputs on the OI are 0 when the switch is open, while the variables associated with digital inputs on the RC are 0 when the switch is closed. See Table 5 for the action of the `&&` operator. Refer to the documentation provided by Innovation First [4] for the hardware details.

12 Checking Your Robot Controller

It is very important that you check out the inputs and outputs of your robot controller and operator interface during the first few days of the build period. Although it is very unlikely that you will find that the controller is defective, it can happen and it is important to learn of any defects early enough during the build process that you will have time to obtain a replacement part from Innovation First. In addition to being of use in checking out the robot controller and operator interface, the same program can be used to wring out the mappings of custom switches you have wired in a control box. It is also useful for testing whether limit switches work before you actually put motors under power.

You can arrange that a variable with the static storage class (see Section 5) remembers individual input settings. When a change is detected, your code can print out a message identifying the variable that has changed and what the new value is. The required code is easily replicated and edited

using cut and paste. This is done with a section of code along the lines of the snippet displayed below. For an understanding of ABS(x), refer to Sections 9.2 and 4.6.7.

```

#define ABS(x) ((x) < 0 ? -(x) : (x))    /* absolute value of x */
#define HYSTERISIS 4
void Process_Data_From_Master_uP(void) {
    int temp;
    Getdata(&rxdata);

    /* Digital inputs on the robot controller. Repeat this pattern
    for the desired number of digital inputs, 01, 02, 03, ...
    These inputs are tested by putting a switch on the digital input
    to test, and then toggling it on and off. Each time the input
    changes a message identifying the input and the new value is
    printed.
    */
    {static int lastrc_dig_in01;
      if((temp = (int)rc_dig_in01) != lastrc_dig_in01) {
          lastrc_dig_in01 = temp;
          printf("rc_dig_in01 = %d\n", lastrc_dig_in01);
      }
    }

    /* Analog inputs on the robot controller. Repeat this pattern
    for the desired number of analog inputs, 01, 02, 03, ..., on the
    robot controller. An input is tested with a potentiometer, and
    each time the potentiometer is adjusted a message will be printed.
    The HYSTERISIS check is taking care of changes caused by noise
    on the input line.
    */
    {static int lastanalogin01;
      if(ABS((temp = Get_Analog_Value(rc_ana_in01))
              - lastanalogin01) > HYSTERISIS) {
          lastanalogin01 = temp;
          printf("RC_Analog_In01 = %d\n", lastanalogin01);
      }
    }

    /* Repeat this pattern for each analog input, and each switch
    input, on each of the four operator interface ports. These are
    tested by attaching a joystick and adjusting its settings.
    */
    {static int lastp1_y;
      if(ABS((temp = (int)p1_y) - lastp1_y) > HYSTERISIS) {
          lastp1_y = temp;
          printf("p1_y = %d\n", lastp1_y);
      }
    }
}

```

```

    }}
    {static int lastp1_sw_trig;
    if(ABS((temp = (int)p1_sw_trig) != lastp1_sw_trig) {
        lastp1_sw_trig = temp;
        printf("p1_sw_trig = %d\n", lastp1_sw_trig);
    }}

    /* A test that the OI is transmitting the autonomous mode
    bit correctly.  You will have to make your own "competition
    port" switch box to test this.
        */
    {static int lastautonomous_mode;
    if(ABS((temp = (int)autonomous_mode) != lastautonomous_mode) {
        lastautonomous_mode = temp;
        printf("autonomous_mode = %d\n", lastautonomous_mode);
    }}

    Putdata(&txdata);
}

```

The idea here is for you to write one program that is useful to test all of the available inputs on the robot controller and the operator interface, and use this to make sure that your equipment is fully functional. This type of code is something that is quite suitable for the application of the macro pre-processor, adding to clarity.

```

#define CHECK_RC_DIG_IN(var, lastvar, pvar) {\
    static int lastvar;\
    if((temp = (int)var) != lastvar) {\
        lastvar = temp;\
        printf(pvar, lastvar)\
    }\
}

```

The code that checks the digital inputs for changes, printing them as they occur, is now:

```

CHECK_RC_DIG_IN(rc_dig_in01, lastrc_dig_in01, "rc_dig_in01 = %d\n")
CHECK_RC_DIG_IN(rc_dig_in02, lastrc_dig_in02, "rc_dig_in02 = %d\n")
CHECK_RC_DIG_IN(rc_dig_in03, lastrc_dig_in03, "rc_dig_in03 = %d\n")
...

```

The code now resembles a table, and as a result its correctness is easily confirmed. Following this lead, you can construct suitable macros for all of the digital and analog inputs that you want to check.

You should take the next step by coding connections from suitable inputs to the relay and PWM outputs, so that you can test *all* of the available outputs on the robot controller. You should also test the indicator lights on the OI if you plan to depend on them.

13 Polled Wheel Counters

It can be quite profitable for you to endow your robot with the ability to measure distance on the floor as it travels. This can be very useful for the autonomous mode when the robot needs to make a sequence of measured runs and turns in order to achieve some goal. One can use a variety of sensors to implement wheel counters. We use the banner optical sensors looking at reflective tape placed on the wheel rims of the robot, or at tape placed on a drive shaft. The goal of the wheel counter code is to keep track of light/dark transitions, counting them and thereby measuring distance traveled on the floor. If you are patient enough to stick on all the reflective tape involved, you can measure distance on the floor very accurately.

One way to accomplish this is to have the RC poll the sensors at a very high speed and keep track of light/dark and dark/light transitions. In the example below, we demonstrate how to poll the sensors in the `Process_Data_From_Local_IO()` call, in order to count the transitions. A key issue is that you do not want to miss a light/dark transition, perhaps while processing the Packet Cycle with the OI, so there is some extra code devoted to providing a diagnostic for this.

```
int left, oldleft, leftcounter;
long leftstringlength, minleftstringlength;

void Process_Data_From_Local_IO(void) {
    if((left = ((int)rc_dig_in10)) != oldleft) {
        oldleft = left;
        leftcounter += 1;
        if(leftstringlength < minleftstringlength) {
            minleftstringlength = leftstringlength;
        }
        leftstringlength = 1;
    }
    else{
        leftstringlength += 1;
    }
}
```

```
}
```

Noting the occurrence of “left” in the variable names, it might be obvious that the banner sensor for the left wheel counter is tied to “rc_dig_in10” and that one can also put a wheel sensor on the other side of the robot, using “right” in a second set of variable names. This code keeps track of the past sensor output using the variable “oldleft” to save the prior value of the sensor, and increments “leftcounter” when it changes.

The code involving “leftstringlength” is keeping track of the number consecutive times that the sensor is read to have the same value. The corresponding variable “minleftstringlength” keeps track of the minimum number of times the same value occurs. Remembering that the RC might “get busy” while the reflective strip is coming by at high speed, it is important that we check this diagnostic using `printf()` while the robot is being operated and make sure that the RC is not missing any light/dark transitions at the speeds that one plans to use the counters.

Code that controls the actions of the robot during the autonomous mode can zero the wheel counters when required, and can take suitable actions when they reach values indicating that a suitable distance has been traveled on the floor.

It is also possible to maintain a wheel counter using the interrupt capability of the PIC micro-controller. This can be dangerous ground if you do not pay attention to the hazards involved. See section 19.

14 Measuring time

It is very useful for your robot to be able to measure time. Elapsed time might be used in place of distance measurements to control stages of an autonomous mode, or might be used by the robot to determine that a goal in the autonomous mode will not be reached (due to the fact that the robot has encountered an obstruction) and the autonomous code can be programmed to give up instead of risking damage.

Remembering that the packet cycle is a fixed interval of 26.2 milliseconds, time can be measured in “ticks” of 26.2 milliseconds, roughly 38 ticks per second, by counting the number of trips through the packet cycle. There is a packet cycle in both the operator control and the autonomous sections of the code, the latter being handled in the `User_Autonomous_Code()` routine. You need to make sure that the type you use for the counter will not over-run for

the time interval you want to measure. You reset the counter to zero every time you want to start the timer, so to speak. It is perfectly acceptable for the timer to over-run when you are not using it.

As an example of the use of a timer, consider the code below:

```
int timecounter = 0;
int tripped = 0;
void Process_Data_From_Master_uP(void) {
    /* Get the packet from the OI, storing the data in
    rxdata. This causes variables such as p1_y to reflect
    the setting of a joystick. */
    Getdata(&rxdata);
    /* If the top button on the left joystick is
    pressed and the timer is not already tripped. */
    if(p1_top && !tripped) {
        tripped = 1; /* Remember that it was pressed. */
        timecounter = 0;
    }
    if(tripped && (timecounter < 72)) {
        pwm01 = 254; /* forward */
        pwm02 = 254;
    }
    else {
        pwm01 = 127; /* stop */
        pwm02 = 127;
        tripped = 0;
    }
    timecounter += 1; /* Increment the time counter. */
    /* Returns a packet to the OI, containing
    values that have been set. */
    Putdata(&txdata);
}
```

The `int` used for the timer, and remembering that the timer was tripped, must be defined outside of the function because this memory must persist outside of the function call. Each time the function is called, the counter is incremented. Given the fact that the counter is an `int`, providing a maximum value of 32767, the maximum time that this timer can measure is a little more than 14 minutes. This is much longer than a competition match.

In this example, the motors are stopped unless the top button on the joystick is pressed. When the button is pressed, the RC remembers that this has happened and runs both drive motors forward for about 2 seconds. Once the time period has expired, the memory that the timer had been tripped is

reset, and the coded mechanism will once again be ready to be tripped. The equivalent loop on the EDU controller runs at a slightly faster rate.

15 Controlling the Air Compressor

The kit of parts contains a pressure actuated switch that opens at 115 psi and closes at 95 psi. It is intended (required) that you control the compressor by reading this switch with the robot controller, and then control the compressor with a spike relay.

```
void Process_Data_From_Master_uP(void) {
    Getdata(&rxdata);

    relay1_fwd = !rc_dig_in01;

    Putdata(&txdata);
}
```

This assumes that you are controlling the air compressor with a spike relay wired to the relay 1 output, and the pressure switch is wired to digital input 1. The switch closes, producing 0, when the pressure is below 95 psi. The switch opens, producing a 1, when the pressure goes above 115 psi. Refer to Table 5 for the action of the ! operator.

Although your program should never command the spike relay to run the motor in reverse, it is a good idea to wire the motor to the spike in a manner that precludes the possibility. An error in the program might reverse the motor and this will blow the fuse if it is already running in the forward direction.⁷ This is done by wiring the motor negative lead to the negative input to the spike, instead of the negative output. When wired in this manner, the spike can only turn the motor on and off in the forward direction.

16 Analog Feedback

A common control problem with a FIRST robot is one of controlling the position of something driven by a motor, perhaps an arm, but one could also control a linear position with a rack and pinion setup. If students attempt

⁷The author learned this the hard way in the good old days of PBASIC controllers.

to control the arm *open loop*, one tends to have a situation where the arm is flailing back and forth. This creates a safety hazard in addition to control difficulties.

This problem is solved with analog feedback. You set up a sensor, usually a potentiometer, that allows the robot controller to track the position of the arm. The computer compares the indicated position to the desired position, and applies a force through the motor to cause the arm to move to the desired position. The robot controller can consistently respond to the position of the arm much faster and more accurately than a person can.

If the force/torque is linear in error correction, the system behaves as if a spring is attempting to return the arm to the desired position. The arm will persistently oscillate about the desired point. To damp the oscillations, one must apply a damping force that is proportional to the rotational speed of the arm, but opposite to the motion. The speed is measured by remembering the position of the arm on the last packet cycle. The idea is the same as what happens when you put a shock absorber on a car, but you do it through a force applied electrically with the motor. In essence, you are producing electric molasses. When this is done properly, and the settings are carefully tuned, the arm will quickly go to a new position commanded by the computer with minimum, if any, overshoot. In the parlance of a physicist, what you are creating is a critically damped harmonic oscillator.

The arm control code from our 2004 robot, Seabiscuit, is displayed below. There are a lot of tuning details, but it provides for very effective analog feedback control of the arm we used on this robot. This code is called every time through the operator control loop, and through the autonomous control loop, in order to cause the arm to head to the desired position.

```
/* The routine set_arm_position(position) sets the motor drive
power for the arm. The input position is meant to be the value
returned from the arm control joystick, multiplied by four in
order to take care of the 8/10 bit inconsistency between analog
inputs on the RC and those from the OI. The arm is also
controlled during the autonomous mode, using programmed
values in the state machine. The value read from the arm
position potentiometer is low when the arm is stowed. The
value read from the operator control is low when angled
towards the operator, and high when pushed away from the
operator. The difference of these two signals is processed in
order to produce the signal, relative to STOP, to drive the arm
motor with. It is important to get the signs right, or the arm
will head away from the desired position, or experience
```

growing oscillations. It is best to test with the chain drive for the arm disconnected, observing the behavior of the motor. Once you have it right, hook the chain up.

```

*/
void set_arm_position(int position) {
    int armreading;
    int armdrive;
#define STOWEDLIMIT 40 /* Limit in the stowed direction. */
#define BACKLIMIT 800 /* Travel limit over the back. */
#define DRIVELIMIT 115 /* Drive motor no harder than this. */
    armreading = Get_Analog_Value(rc_ana_in01);
    armdrive = position - armreading;
    armdrive /= 2;
#define DAMPING
#ifndef DAMPING
    /* The old arm reading must be a variable declared
    outside of a routine so that it remembers the value
    between successive calls.
    */
    armdrive -= (armreading - oldarmreading) * 2;
    oldarmreading = armreading;
#endif
    if(armdrive < -DRIVELIMIT) {
        armdrive = -DRIVELIMIT;
    }
    else if(armdrive > DRIVELIMIT) {
        armdrive = DRIVELIMIT;
    }
    if((armdrive < 0) && (armreading < STOWEDLIMIT)) {
        armdrive = 0;
    }
    if((armdrive > 0) && (armreading > BACKLIMIT)) {
        armdrive = 0;
    }
    if(armdrive > WINDOW) {
        armdrive += 137; /* Biggest no power value, for this
        victor. One must make sure that adding
        this to DRIVELIMIT will not exceed the
        maximum value for a PWM output. */
    }
    else if(armdrive < -WINDOW) {
        armdrive += 125; /* Smallest no power value, for this
        victor. One must make sure that this is
        greater than -DRIVELIMIT. */
    }
}

```

```

else {
    armdrive += STOP;
}
pwm03 = armdrive;

/* Inhibit arm swing if arm off switch is up.
*/
if(((int)rc_dig_in01) == 1) {
    pwm03 = STOP;
}
}

```

It is important to note the action controlled by `rc_dig_in01`. There are usually plenty of spare digital inputs on the robot controller, and you should use them to inhibit features of your robot that might pose a danger in the pits, enabling them only when you want to use them. Switches wired in this manner can also control variations of your autonomous program, and are used to disable features of the robot that have become broken without changing wiring or the program.

17 Time Integration

Before we discuss using a rational rate gyro to determine the angular position of a robot, we need to discuss the concept of integrating a function over a time interval. This is a basic concept from calculus, although we will avoid the mathematical machinery of calculus in this discussion. We will be using a method known as numerical integration in our program for the robot controller, but as we are attempting to address a target audience that understands only algebra, we will avoid a detailed numerical analysis. This topic can be understood by anyone who has a good understanding of high school algebra.

Suppose our robot starts out at a particular position, x_0 , at time t_0 . Suppose, also, that the velocity of the robot as a function of time is given by the function $v(t)$. Given t_0 , x_0 and $v(t)$, we would like to know the position of the robot as a function of time, $x(t)$.

If the velocity of the robot is a constant, v_0 , the solution to this problem is easy. The position of the robot is given by the equation

$$x(t) = x_0 + v_0(t - t_0) \quad . \quad (1)$$

The solution for the more general case where the velocity is given by $v(t)$ can be constructed by considering a sequence of N time steps of size $\Delta t = (t - t_0)/N$, during which the velocity is well approximated by a constant, v_i , with i denoting the time step and having values ranging from 1 to N . In this case, the formula for a constant velocity can be used to calculate the change in the position for each time step, and the position at time t is given by the sum

$$x(t) = x_0 + v_1\Delta t + v_2\Delta t + \dots + v_{N-1}\Delta t + v_N\Delta t = x_0 + \Delta t \sum_{i=1}^N v_i \quad (2)$$

The v_i can be the $v(t)$ evaluated at the beginning of each time step, at the end, or somewhere in between, it does not matter if the time steps are small enough. This summation process delivers an accurate position as a function of time, $x(t)$, if N is large enough. Drawing the velocity, $v(t)$, as a graph, the change in the position is the area under the curve between t_0 and t .

Our robot is capable of sampling an analog signal at a periodic interval determined by the packet exchange rate with the operator interface, $\Delta t = 0.0262$ seconds, and somewhat more quickly if we use interrupts to sample the analog signal. We could use the formula given by equation (2) directly, using the packet loop interval to define the size of the time step, but our accuracy would suffer because of the relatively large size for the time step. To improve accuracy, we use what is known as the trapezoidal rule to evaluate the time integral. There are more sophisticated schemes for numerical integration, but the trapezoidal rule will be sufficient for our purposes.

In the trapezoidal rule, the function $v(t)$ is taken to have the form of a straight line connecting the two measured values at each end of the time step, and the area contributed by the time step is calculated using the appropriate rectangle and triangle. If v_0 is the velocity at the beginning of the time step, and v_1 is the velocity at the end, the position at the end of the time step is calculated as follows.

$$x(t_0 + \Delta t) = x_0 + \Delta t v_0 + \Delta t \frac{v_1 - v_0}{2} = \frac{\Delta t}{2}(v_0 + v_1) \quad (3)$$

To integrate the input of a sensor, in time, all we need to do is read successive values, one for each packet cycle with the operator interface, remembering the value from the prior packet cycle, and perform simple addition. Referring to our earlier discussion above, we see that we are using the average of the

velocity at the beginning and the end of the time step in order to evaluate the change in the position during the time step. The accuracy of our numerical integration improves if we increase the rate at which we sample the signal and are very careful about how we implement the arithmetic.

18 Integrating a Rotational Rate Gyro

A rotational rate gyro is a sensor that produces an analog signal that corresponds to the rate of rotation of the device about some axis. In prior years, a gyro was provided in the kit of parts that hooked directly to the analog input on the robot controller, taking its power from the robot controller itself. If no gyro is provided in the kit, you can put together your own gyro as additional electronics using the AMD ADXRS75EB, ADXRS150EB or ADXRS300EB available from Digikey. They correspond to maximum rotational rate measurement capabilities of 75, 150 and 300 degrees per second. These parts are “evaluation boards” that resemble computer chips that can be soldered in a vector board, or your own printed circuit board, that mounts the voltage regulator and/or power line filters that are required, depending upon the rules.

The gyro produces an analog voltage signal that ranges from zero to five volts, depending upon the rate at which the gyro is being rotated around its measurement axis. A voltage close to 2.5 volts represents zero rotation. This voltage is measured using one of the analog inputs on the robot controller using the `Get_Analog_Value()` function. For our purposes, we assume that we have wired the gyro to analog input 10, so the correct call would be `Get_Analog_Value(rc_ana_in10)`. This function returns the voltage read as an `int`, the value of which ranges from 0 to 1023.

The gyro signal can be used directly to cause the robot to turn at a fixed rate. This capability might be used to cause the robot to go straight during user mode (if desired), despite the directional bias that is always present in the drive train. We have found that human operators become accustomed to this bias and correct for it without computer assistance. One might also use the gyro to maintain the angle of an appendage.

In this section, we will discuss integrating the gyro in order to measure turns. It is often the case that a robot does not turn well, or the robot might encounter some obstruction when turning, and using feedback to measure the turn may make executing an accurate turn quite reliable during autonomous

mode.

To measure angular position with the rotational rate gyro, we need to integrate its signal as a function of time, according to calculus. The gyro outputs a signal corresponding to the rate of rotation, not the actual angle traversed. If the gyro says it is turning at 5 degrees per second for instance, and this goes on for 4 seconds, the total distance turned is 20 degrees. If one imagines the gyro signal as being displayed on an oscilloscope, the total distance turned is the area under the curve drawn, as a function of time, on the screen. This area might increase or decrease as a function of time due to the gyro signal crossing zero. “Zero” is whatever voltage is read when the gyro is not turning.

The basic version of the code to integrate the gyro is shown below. In this case, 511 is the average signal from the stationary gyro turned into an integer value. The variables involved must be defined outside of a function so that their values persist between calls to the routine that handles the packet cycle, and calls that execute the code snippet shown below.

```
#define GYROAVE 511
long newgyrovalue = 0;
long oldgyrovalue = 0;
long gyrointegral = 0;
    newgyrovalue = Get_Analog_Value(rc_ana_in10);
    gyrointegral += ((oldgyrovalue + newgyrovalue) / 2) - GYROAVE;
    oldgyrovalue = newgyrovalue;
```

This code snippet runs each time the robot controller processes the packet cycle, every 26.2 milliseconds. Each time through, the gyro is read and the area under the line connecting the old and the new value is added to the gyro integral. The new value is saved so that it is available as the old value on the next loop.⁸

Code of this sort will work just fine for measuring robot turns executed within a few seconds during autonomous mode. To execute a measured turn, the robot controller zeros the gyro integral and then turns until the integral reaches the value corresponding to the angle desired. The robot then goes on with any further business it has. There may be instances, however, where you need to deal better with the errors involved. The first error arises from the quantization that occurs because the gyro value is stored as an integer.

⁸You need to initialize `oldgyrovalue` once in the startup code that initializes the RC program at power up, or the value of the integral will be corrupted. Alternatively, the integral can be zeroed before a measured turn is made.

The second source of error is that one bit of precision is lost when the sum of the new gyro value and the old gyro value is divided by two. If the sum is odd, a bit of precision is lost when the value is shifted one bit to the right in order to divide by two.

To understand the effects of quantization error, consider that the values returned by `Get_Analog_Value()` are integers corresponding to the voltage returned by the gyro. Given that the signal from the gyro ranges from 0 to 5 volts as the rotational rate ranges from 150 to +150 degrees per second (for the ADXRS150EB), and that the corresponding range of integer values is 0 through 1023, the quantization error for the measurement is 0.29 degrees per second. The gyro may be stationary, but the value read by the robot controller might indicate a motion of up to 0.29 degrees per second in one direction or the other. For a 15 second autonomous period this could lead to a total error of 4.3 degrees. The potential error for a one second turn is only 0.3 degrees, quite tolerable, but we can do better.

If you make repeated measurements of the gyro with `Get_Analog_Value()` you will find that it does not return the same value even though the gyro is perfectly still. Expected values would be close to 511, plus or minus a few. By making repeated measurements we can obtain the average value and use this average as the definition of zero for the purposes of integrating the gyro signal. We can do this because there is some noise present in the signal from the gyro and the robot controller is sensitive enough to measure it. The noise, as described in reference [8], is our friend. At this point it should be clear why we spent time on binary, and binary fixed point arithmetic, earlier in this manual. The average value is interpreted as a binary fixed point number.

The code that integrates the gyro signal using fixed point arithmetic is deceptively simple. It is run each cycle through the packet loop:

```
#define GYROAVE 8183      /* sixteenths */
long newgyrovalue = 0;
long oldgyrovalue = 0;
long gyrointegral = 0;
    newgyrovalue = Get_Analog_Value(rc_ana_in10) << 4;
    gyrointegral += ((oldgyrovalue + newgyrovalue) >> 1) - GYROAVE;
    oldgyrovalue = newgyrovalue;
```

In this case we are using the average value returned by the gyro in sixteenths in the sense of our earlier binary fixed-point discussion. We are using an average value of 8183 as an example. You will have to make your own measurement to determine the correct value for your robot and gyro, and

you might choose to work in higher precision. Remembering our discussion with regard to shifts, the left shift by 4 bits is multiplying by 16. The right shift by one bit is a safe divide by 2 because the value returned by the gyro is never negative, and the fact that the values were already left shifted provide that we never lose precision by shifting off a low order 1 bit.

The code to compute and print the average of the stationary gyro is straight forward.

```
long gyroave = 0;
long count = 0;
while(count < (16 * 1000)) {
    gyroave += Get_Analog_Value(rc_ana_in10);
    count += 1;
}
gyroave /= 1000;
printf("GYROAVE = ");
pl(gyroave); /* pl() is as defined earlier in this document. */
```

This code is placed so that it is executed every packet cycle in a stand-alone program used for the purpose of measuring and printing the average. In this case we are computing and printing the gyro average in sixteenths. Additionally, we are measuring the average over an additional factor of 1000 iterations to make sure that we measure the average over a suitable length of time. Given 26.2 milliseconds for each packet cycle, you can calculate how long this code will take to run. One may also arrange to delay the beginning of the count or take the count in response to a button being pressed so that the average may be measured a few times in order to check for consistency.

As a final note, it is important that you sample the gyro at a rate that is at least twice the bandwidth of the gyro. The gyro, as shipped, has a bandwidth of 40 hertz. If you use the packet cycle to integrate the gyro, this operates at 38 hertz and you should drop the bandwidth of the gyro by at least a factor of 2 by installing a suitable capacitor. Alternatively, you can increase the sampling rate by using timer interrupts, see Section 19. Sampling at rates higher than twice the bandwidth of the gyro will provide more accurate time integration of high frequency signals the gyro might deliver.

19 Interrupt Programming

There are many situations where the polled approach to handling sensor input can lead to performance limitations. An example of polling can be

found in Section 13 where the RC program keeps checking the wheel sensors while waiting for a packet to arrive from the OI. Once a packet arrives, `Process_Data_From_Master_uP()` is called to read the packet into memory, computes the desired outputs for motors and solenoids, and then return a packet to the OI. The work performed in the `Process_Data_From_Master_uP()` call can take quite a bit of time. During this time some counts from wheel sensors might go by without being noticed. The result is effectively a speed limit for the robot while the wheel counters are being used.

It would be much better if a change in the state of the digital input associated with the wheel sensors caused the robot controller to interrupt and automatically increment the variable that is keeping track of the wheel count. The interrupt response can occur during the `Process_Data_From_Master_uP()` call, assuring that a wheel count is never missed. We can make this happen by suitably configuring and programming low priority interrupts in the RC.

An interrupt is an interruption of the normal path of program execution that would otherwise occur in the RC. The RC can be executing any line of code when an interrupt occurs. When the interrupt arrives the processor is forced to jump to a subroutine, called the Interrupt Service Routine (ISR), or interrupt handler. The job of the ISR is to figure out what caused the interrupt, take the appropriate action, and then return to normal program execution as quickly as possible. It must do this without corrupting any of the data structures in use before the interrupt occurred.

It is important to keep the code in the ISR as lightweight as possible. Calling `printf()` to format some output from within an ISR is very dangerous, although it is possible to do such things using custom code that buffers formatted output using interrupts. One needs to understand the potential impact of every routine that is called from an ISR. Blindly calling complex routines can cause unpredictable results.

There are at least two serious problems associated with using interrupts that one must be careful about. Failing to pay close attention to the details involved will lead to a robot program that appears to work correctly most of the time, but has very strange gremlins pop up every few minutes, or as rarely as every few hours. These problems happen when interrupt service routines interfere with normal program execution. These gremlins can be very difficult to track down after they creep into the code.

19.1 Saving and Restoring Program Context

The ISR uses the same processor registers and static memory space as the code that is running when the interrupt occurs. This data will be corrupted in the user program if the data is not correctly saved and restored by the ISR. If one runs the timer based gyro integration code, that we will discuss later, without properly saving and restoring the `.tmpdata` section, the `printf()` calls that print the output value of the gyro signal will be corrupted every 10 seconds or so. It is instructive to demonstrate this malady by changing the `save` statement.

To address this problem, the one must arrange that the ISR saves and restores any memory resources that are used during the servicing of the interrupt. Documentation on this issue is found in section 2.9.2.4 `ISR CONTEXT SAVING` of the *MPLAB C18 C Compiler User's Guide*. The compiler managed resources of concern are, at least:

- `PROD`: If servicing the interrupt results in calling a function that returns 16 bit wide data the `PROD` file register must be saved and restored.
- `.tmpdata`: If an interrupt service routine calls another function, the normal function's temporary data section must be saved and restored.
- `MATH_DATA`: If the interrupt service routine calls a library function that returns 24 bit or 32 bit data, the math data section must be saved and restored.

Additionally, your interrupt code may make use of your own statically allocated variables and you may need to save and restore these variables if they are used in a manner that could interfere with normal program execution.

Once the data that needs to be saved and restored by the ISR is identified, arranging to save and restore it is a simple matter. A `pragma` statement is used to identify the ISR and any memory locations and need to be saved and restored by the ISR. An example of this `pragma` statement can be found in `user_routines_fast.c` in the default code.

```
#pragma interruptlow InterruptHandlerLow save=PROD
```

It is likely that saving only the `PROD` register will not be sufficient. The examples we discuss in this manual call further routines, requiring saving and restoring `.tmpdata`, and it pays to be conservative when it comes to taking a risk on the gremlins. It is a good idea to save all of the compiler managed resources unless there is good reason do do otherwise.

```
...   save=PROD,section("MATH_DATA"),section(".tmpdata")
```

In the event that you have a global variable named “myvariable” that needs to be saved and restored by the ISR, you can add it to the pragma statement as follows:

```
...   save=PROD,section("MATH_DATA"),section(".tmpdata"),myvariable
```

The task of saving and restoring this data in the ISR does take time and as a result increases the amount of time required to service an interrupt. If the interrupt handler code is very small, and fast, the overhead of saving and restoring data that did not need protection can become significant. Data that is not accessed in the call chain of the interrupt handler can be removed from the list to be saved and restored, but one must be very certain about this.

19.2 Race Conditions

The processor in the RC is an eight-bit microprocessor that handles wider types such as 16 and 32 bit integers one byte at a time. If the user application is reading a variable that the interrupt routine is changing the interrupt could occur between the independent reads of the bytes that compose the variable. An inconsistent and completely bogus value can result. This hazard is often missed by programmers implementing code that uses interrupts. The result of ignoring this hazard can be weird behavior that can occur very rarely and quite unpredictably.

To understand the hazard, imagine our wheel counter example where we are using a `volatile unsigned int`⁹, “i”, to record the counts. The computer handles the `int` as two distinct bytes, “ihigh” and “ilow”, so to speak, that each take on the values 0 to 255. When 1 is added to “ilow” when it is equal to 255, it overflows to zero and a carry is generated. The carry is then added to the higher order byte, “ihigh”.

Each time the interrupt occurs, the ISR adds one to the wheel counter. From the perspective of the ISR, there is no difficulty as long as it is the only routine that is modifying the counter. Each time the ISR is called in response to an interrupt, it dutifully adds one to “i”. It first reads “ilow”,

⁹The C18 compiler manual recommends using the `volatile` type qualifier for all global variables that are updated in interrupt handlers. This is to prevent the compiler from “optimizing away” multiple memory accesses to them.

adds one to the value and then stores “ilow”. If the increment of “ilow” generates a carry, however, it then reads “ihigh”, adds one to the value and then stores “ihigh”.

The problem occurs if the interrupt happens while the normal program is in the middle of reading “i”, this being done one byte at a time using “ihigh” and “ilow”. Suppose that the program code is looking to change its behavior in some manner when the counter is greater than or equal to (ihigh=3,ilow=5), the counter value is in fact (3,255), and the ISR is going to change it to (4,0). Suppose, as well, that the program code reads “ihigh” before the interrupt and reads “ilow” after the interrupt. In this case, the value of “ihigh” that will be read is 3, while the value of “ilow” that will be read is 0, and the program will think that the value of the counter is (3,0), when it was “half way between (3,255) and (4,0)”, so to speak. The program should take some action for any counter value above (3,5), and doesn’t because of the hazard posed by the fact that the integer is being read one byte at a time. Perhaps the processor will have better luck during the next packet cycle, and it usually does, but you will think that your robot has a good case of the gremlins when strange things happen, and they can be very nasty gremlins indeed!

You might think that this hazard is an inherent one associated with the eight-bit micro-controller that you can’t do much about. The solution to this problem is actually quite simple. You simply read the variable “i” twice, and if you get the same value both times you know that interrupt routine did not change the variable during the read and you can proceed, safely trusting the value of “i”. If you don’t get the same value twice, try again.

The relatively safe situation from the perspective of the ISR changes as soon as the normal program is allowed to change the value of “i”, perhaps to reset it in order to start another measured distance. In this situation the tables are turned, and the number that the interrupt service routine sees can be inconsistent, with corrupted results. Consider the situation where the normal program writes ihigh=0, and then ilow=0, in order to set “i” to zero, where the initial value was (0,255). If the ISR interrupts in order to add one in between the two writes, the result will be (1,0) when it should have been either 0, or 1. Remembering that (1,0) has the value 256, your counter is a long ways away from where it should be and you might discover the next leg of the autonomous trip running short now and then due to those pesky gremlins!

We use a sentinel flag to prevent this problem, although there are many

strategies that one can use. Prior to zeroing the counter, a single byte flag is turned on in order to inform the interrupt handler that the user code is in the process of zeroing the counter. If the interrupt handler sees this sentinel, it does not take its usual action of incrementing the counter and instead drops the increment on the floor. This is a reasonable outcome for the rare event that the interrupt occurs while the user code is zeroing the counter.

At this point, you can appreciate the difficulties associated with using interrupts, and now understand why FIRST does not offer support for their use. The value of using interrupts may be worth the extra care involved. If you are going to use them, you will need to carefully analyze the hazards involved, and you will need to have a relatively good understanding of what is going on at the machine level to fully appreciate all of the hazards you will encounter.

As a practical example, and proof that we are not just speculating about these hazards when using interrupts, we troll the implementation of clock interrupts found in `edu_clock.zip` found at <http://www.kevin.org/frc> for this class of problem.¹⁰ You will need the EDU robot controller in order to run this example.

Download and unpack `edu_clock.zip`. You will need to unpack this directly on the C drive, `C:`, or in a directory located there with a relatively short path name in order to avoid a name length limit in the C-BOT development environment. Build the EDU clock demo and upload it into the EDU robot controller for execution. It will print messages in the `COM1` terminal window of the form:

```
EDU Clock Demo Initialized...
Elapsed time = 000:00:00:01
Elapsed time = 000:00:00:02
Elapsed time = 000:00:00:03
Elapsed time = 000:00:00:04
...
```

Now that you have a working EDU clock demo program, modify the bottom of `user_routines_fast.c` as follows:

```
#include "printf_lib.h"
void Process_Data_From_Local_IO(void) {
    /* oldmsClock must be saved between calls.
```

¹⁰We note that there is no defect in the code as provided, we are simply using it as a vehicle to modify in order to demonstrate the potential hazard.

```

        */
static unsigned long oldmsClock = 0;
unsigned long newmsClock;
newmsClock = msClock;
/* If we have a little byte trip from
the devil that we weren't expecting...
*/
if(newmsClock < oldmsClock) {
    printf("Hit: old = %lx, new = %lx\n",
           oldmsClock, newmsClock);
}
oldmsClock = newmsClock;
}

```

When you modify, build, and load this program into the EDU robot controller it will print the usual clock notices, interspersed with the output of print statements (in hex) that indicate a hit on the interrupt hazard.

```

Hit: old = 0dff new = 0d00
Hit: old = 020ff new = 02000
Hit: old = 02eff new = 02e00
Hit: old = 040ff new = 04000

```

You can expect a hand-full of hits in a time period as short as 20 seconds, but the places and actual values printed will vary from run to run. There is a pattern, however. The old value always has a ff in the low order byte, while the new value has a 00. This is because the read of `msClock` was split by the clock interrupt, when the low order byte of `msClock` was ff in hex, or 255 in decimal. The high order bytes of `msClock` appear to have been read first, followed by the ISR adding one to `msClock` which added 1 to 255 to generate 0 and a carry, then add the carry to the high order byte that had already been read by the normal program, and then the low order byte, now zero, is read by the program. The result of this is the corrupted value that is tested for, and reported, in our modification of Kevin's example.

We can clean this problem up by reading `msClock` twice, *making sure that msClock was declared with the volatile type modifier*, and be persistent until the same value is read both times.

```

#include "printf_lib.h"
void Process_Data_From_Local_IO(void) {
    static unsigned long oldmsClock = 0;
    unsigned long newmsClock;
    unsigned long newmsClockcopy;

```

```

    /* Assuming that the C18 compiler will not
    optimize away the extra reference to msClock.
       */
    newmsClock = msClock;
    newmsClockcopy = msClock;
    while(newmsClock != newmsClockcopy) {
        newmsClock = msClock;
        newmsClockcopy = msClock;
    }
    /* If we have a little byte trip from
    the devil that we weren't expecting...
       */
    if(newmsClock < oldmsClock) {
        printf("Hit: old = %lx, new = %lx\n",
              oldmsClock, newmsClock);
    }
    oldmsClock = newmsClock;
}

```

This modification removes the hazard, and we never get a hit. Now, you might ask why Kevin's `Display_Time()` routine never printed a bogus time value. Examining this routine, we find

```

    if(Clock > Old_Clock)

```

on line 169 of `clock.c`, and with your understanding of this section you realize now that there is much more to this statement than meets the eye! In addition to printing the new time when the one second clock ticks forward, it ignores the ticks backward that occur now and then.¹¹ You must be very careful about changing this statement to something else that you might think would be equivalent for your purposes.

It is worthwhile to note that the code to be used in order to assure the safe reading of a variable updated in an interrupt handler is a good candidate for a macro definition. It can also be encapsulated in a function.

```

    /* Copy "b" to "a", safely, where "b" may be getting
    updated in an interrupt handler, and "type" is the type
    of both "b" and "a".
       */
    #define SAFEREAD(a,b,type) {type copy;\
    a = b; copy = b;\
    while(a != copy) {a = b; copy = b}}

```

¹¹Clock is updated 1000 times less frequently than `msClock`, but the same hazard exists nevertheless. Refer to Murphy's law...

The code to safely obtain the value for `newmsClock` is now:

```
SAFEREAD(newmsClock, msClock, unsigned long)
```

You can employ interrupts to good effect, but you must take the lessons of this section to heart and proceed very carefully. When obscure statements are providing protection from a hazard, make sure that you document it with a comment so that future changes to the code do not inadvertently release those gremlins. Murphy's law applies in this situation. If you are going to attempt the use of interrupts, read the material covering them in the C18 users manual and the data sheet for the PIC micro-controller itself very carefully.

20 Interrupt Based Wheel Counter and Gyro

It is instructive to bring the concepts discussed in the earlier section home by providing a full exposition of interrupt based wheel counters and time integration of a rotational rate gyro. In this case, we are assuming that the hall effect gear tooth sensor available in the kit of parts has been employed to provide a low-to-high transition to digital input 01 for the left wheel, and digital input 02 for the right wheel, each time a gear tooth passes by the sensitive region of the hall effect sensor. A timer is used to sample the rotational rate gyro at a uniform rate higher than twice the bandwidth of the gyro. An example RC program that implements two interrupt based wheel counters and the time integration of a rotational rate gyro can be found at <http://srvhsrobotics.org/eugenebrooks/IntWheelGyro.zip>. An exposition of this code is provided below.

Digital inputs 01 and 02 can be configured to generate an interrupt to the micro-controller for each low-to-high transition. The code to do this, called from within `User_Initialization()` in `user_routines.c`, is:

```
void Initialize_Interrupts(void) {
    /* Initialize external interrupt 1 (INT2 on the 18F8520).
    This is wired to digital input 1 in the IFI Robot Controller.
    */
    TRISBbits.TRISB2 = 1;      /* Configure it as an input. */
    INTCON3bits.INT2IP = 0;    /* Low priority. */
    INTCON2bits.INTEDG2 = 1;   /* Trigger on rising-edge. */
    INTCON3bits.INT2IF = 0;    /* Clear it before enabling interrupt. */
    INTCON3bits.INT2IE = 1;    /* Enable interrupt. */
}
```

```
/* Initialize external interrupt 2 (INT3 on the 18F8520).
This is wired to digital input 2 on the IFI Robot Controller.
The settings are the same as for interrupt 1, above.
*/
TRISBbits.TRISB3 = 1;
INTCON2bits.INT3IP = 0;
INTCON2bits.INTEDG3 = 1;
INTCON3bits.INT3IF = 0;
INTCON3bits.INT3IE = 1;
}
```

The timer used to control the integration of the rotation rate gyro is configured using the following routine. As was the case for the wheel counters, this is called from within the `User_Initialization()` routine.

```

void Initialize_Timer_1(void) {
    /* Initialize the 16 bit timer register.
    */
    TMR1L = 0x00;
    TMR1H = 0x00;

    /* Set the prescaler for a 10 MHz rate. With the interrupt happening on the
    overflow of a 16 bit counter, the interrupt rate is 10MHz / 216, 152.59 Hz.
    The timer interrupt rate can be reduced by a factor of 2, 4, or 8 by picking
    one of the other prescaler values.
    */
    T1CONbits.T1CKPS0 = 0; // T1CSP1 T1CSP0
    T1CONbits.T1CKPS1 = 0; // 0 0 1:1 prescaler (10MHz)
                        // 0 1 1:2 prescaler (5MHz)
                        // 1 0 1:4 prescaler (2.5MHz)
                        // 1 1 1:8 prescaler (1.25MHz)
                        //
    T1CONbits.T10SCEN = 0; /* Timer 1 internal oscillator disabled. */
    T1CONbits.TMR1CS = 0; /* Use the internal clock. */
    T1CONbits.RD16 = 1; /* 16 bit access for the timer register */
    IPR1bits.TMR1IP = 0; /* Set interrupt priority to low. */
    PIR1bits.TMR1IF = 0; /* Clear any existing interrupt. */
    PIE1bits.TMR1IE = 1; /* Interrupt on overflow, from FFFF -> 0. */
    T1CONbits.TMR1ON = 1; /* Enable the timer. */
}

```

It takes a lot of reading of the documentation for the PIC micro-controller to learn what the above statements do. You are referred to Kevin Watson's `frc_interrupts` template code and the rather lengthy documents for the PIC micro-controller for further details and examples.

The low priority interrupt handler is configured in the routine `InterruptHandlerLow()` that can be found in `user_routines_fast.c`. This routine is called for any low priority interrupt. The tests in the if-then-else determine which interrupt actually occurred, controlling execution of code that clears the interrupt and then calls the handler for the interrupt.

```

#pragma interruptlow InterruptHandlerLow save=PROD,section("MATH_DATA"),section(".tmpdata")
void InterruptHandlerLow () {
    if (PIR1bits.TMR1IF && PIE1bits.TMR1IE) { /* Timer 1 *
        PIR1bits.TMR1IF = 0; /* Clear the interrupt. */
        Timer1_Int_GyroIntegralHandler(); /* Call the handler. */
    }
    else if (INTCON3bits.INT2IF && INTCON3bits.INT2IE) { /* The INT2 pin is RB2/DIG I/O 1. */
        INTCON3bits.INT2IF = 0; /* Clear the interrupt. */
    }
}

```

```

    Int_1_Handler();          /* Call the handler. */
}
else if (INTCON3bits.INT3IF && INTCON3bits.INT3IE) { /* The INT3 pin is RB3/DIG I/O 2. */
    INTCON3bits.INT3IF = 0;      /* Clear the interrupt. */
    Int_2_Handler();          /* Call the handler. */
}
else {
    CheckUartInts();          /* For Dynamic Debug Tool or buffered printf features. */
}
}
}

```

At this point, we can examine the handlers for the interrupts as well as the routines that allow the main loop to access the counter variables. First, we examine the handler for the left wheel counter. This handler requires a sentinel variable, `Int1HandlerSentinel`, and the counter itself, `LeftWheelCount`. If the sentinel is zero, the handler increments the wheel counter.

```

static volatile char Int1HandlerSentinel = 0;
static volatile int LeftWheelCount = 0;
void Int_1_Handler(void) {
    if(Int1HandlerSentinel == 0) {
        LeftWheelCount += 1;
    }
}

```

The user code needs to be able to reliably read the value of the wheel counter variable. The multiple reads required to do this are encapsulated in the access routine. The value of the wheel counter is read until the same value is obtained twice, and this value is returned to the user code.

```

int GetLeftWheelCount(void) {
    int Old, New;
    Old = LeftWheelCount;
    while(Old != (New = LeftWheelCount)) {
        Old = New;
    }
    return Old;
}

```

The user code needs to be able to reliably zero the counter variable for the wheel. The problem here is that the interrupt handler might increment the counter variable while the user code is in the process of zeroing it. This problem is solved by using the sentinel variable to indicate that the counter is being zeroed, and having the interrupt handler check the sentinel before incrementing the counter.

```

void ZeroLeftWheelCount(void) {
    Int1HandlerSentinel = 1;
    LeftWheelCount = 0;
    Int1HandlerSentinel = 0;
}

```

The routines that perform the running time integration of the rotational rate gyro hooked to analog input 01 use the same strategies to avoid the hazards of interrupts. The integral of the gyro is maintained in 256ths. Refer to the fixed binary point arithmetic discussion of Section 6.3. On each timer interrupt, the area of the gyro signal since the last sample is added to the running sum and then the value sampled is saved for the next time step. The average signal of a stationary gyro is subtracted to produce a time integral with a reasonably low drift while the gyro is stationary. The average signal of a stationary gyro is measured separately, with 256 consecutive interrupts, taking the average of many trials. A sentinel is used to prevent corruption of the gyro when the integral is being zeroed by the user code.

```

static volatile signed long GyroIntegral = 0;           // in 256ths
static volatile unsigned char GyroIntegralSentinel = 0;
static volatile unsigned long OldGyroValue = 0;
static volatile unsigned long NewGyroValue = 0;
static volatile unsigned long GyroAverage = 132610;    // in 256ths
void Timer1_Int_GyroIntegralHandler(void) {
    if(GyroIntegralSentinel == 0) {
        NewGyroValue = Get_Analog_Value(rc_ana_in01);
        GyroIntegral += ((OldGyroValue + NewGyroValue) << 7) - GyroAverage;
        OldGyroValue = NewGyroValue;
    }
    else {
        OldGyroValue = Get_Analog_Value(rc_ana_in01);
    }
}

```

The access routine for the time integral of the gyro uses the same strategy as the wheel counter code to obtain a reliable value. One can add an additional divide in this routine to obtain any desired calibration for the angular change, such as degree or radian measure.

```

signed long GetGyroIntegral(void) {
    signed long Old, New;
    Old = GyroIntegral;
    while(Old != (New = GyroIntegral)) {
        Old = New;
    }
    return Old;
}

```

Finally, the routine to zero the time integral of the gyro uses the same sentinel variable strategy as was used for the wheel counter.

```

void ZeroGyroIntegral(void) {
    GyroIntegralSentinel = 1;
    GyroIntegral = 0;
    GyroIntegralSentinel = 0;
}

```

To obtain the average of the gyro signal in 256ths, we use a different handler for the timer interrupt. The handler sums the gyro for 256 timer interrupts, and routines are provided to read and zero the sum.

```

static volatile signed long GyroPolledAverage = 0;
static volatile unsigned char GyroPolledAverageSentinel = 0;
static volatile unsigned int GyroPolledAverageCounter = 0;
void Timer1_Int_GyroAverageHandler(void) {
    if((GyroPolledAverageSentinel == 0) && (GyroPolledAverageCounter < 256)) {
        GyroPolledAverage += Get_Analog_Value(rc_ana_in01);
        GyroPolledAverageCounter += 1;
    }
}
signed long GetGyroAverage(void) {
    signed long Old, New;
    Old = GyroPolledAverage;
    while(Old != (New = GyroPolledAverage)) {
        Old = New;
    }
    return Old;
}
void ZeroGyroAverage(void) {
    GyroPolledAverageSentinel = 1;
    GyroPolledAverage = 0;
    GyroPolledAverageCounter = 0;
    GyroPolledAverageSentinel = 0;
}

```

The packet loop is then modified to periodically zero the gyro average, and print the value of the gyro average after enough time has elapsed to collect the sample of 256 values. The running average of a number of samples taken in this manner is computed and inserted into the code for the initial value of `GyroAverage`. The full code for two wheel counters and a timer based gyro integrator can be found at <http://srvhsrobotics/eugenebrooks/interrupt.zip>.

21 State Machine Programming

A state machine is a very useful way to organize code for an autonomous mode. Relatively complex autonomous modes supporting a wide variety of sensor inputs can be organized using this programming concept.

A state machine has a finite set of states. They are usually stored in an integer variable referred to as the *state variable*. The state machine starts off in a given state, referred to as the *start state*. Transitions to new states are governed by the actions of the robot as goals are achieved. Once the last goal is achieved by the robot, the state machine enters a state where the robot takes no further action other than to wait for the operator controlled portion of the match.

In order to illustrate a basic state machine, we will use a very simple one that sequences through a set of states, spending a prescribed amount of time in each one. We can use Kevin's clock example for this purpose. Starting with the unmolested `edu_clock.zip` file, modify the `Process_Data_From_Master_uP()` routine as follows:

```
unsigned long oldclock = 0;
int state = 0;
void Process_Data_From_Master_uP(void) {
    unsigned long newclock;
    unsigned long newclockcopy;

    Getdata(&rxdata);

    /* Get a reliable clock value from the msClock
    variable maintained by the interrupt timer.
    */
    newclock = msClock;
    newclockcopy = msClock;
    while(newclock != newclockcopy) {
        newclock = msClock;
    }
}
```

```

        newclockcopy = msClock;
    }

    switch(state) {
    case 0:
        if((newclock - oldclock) > 2000) {
            oldclock = newclock;
            printf("state was %d, switching to 1\n", state);
            state = 1;
        }
        break;
    case 1:
        if((newclock - oldclock) > 5000) {
            oldclock = newclock;
            printf("state was %d, switching to 2\n", state);
            state = 2;
        }
        break;
    case 2:
        if((newclock - oldclock) > 3000) {
            oldclock = newclock;
            printf("state was %d, switching to 3\n", state);
            state = 3;
        }
        break;
    }
    Putdata(&txdata);
}

```

We are using the millisecond clock value, `msClock`, that is maintained by the timer interrupt in order to tell time. The code involving `newclockcopy` is ensuring that we do not get a corrupted value for the millisecond clock when reading, `msClock`. Refer to Section 19 to understand the hazard that is being dealt with here. The code is using a past clock value, `oldclock`, to measure time relative to the last time the clock was saved.

This state machine spends 2 seconds in state 0, 5 seconds in state 1, and three seconds in state 2, finally ending up in state 3. In the state machine for your robot, you might use measured time, measured distance traveled, and/or measured turns in order to cause the robot to execute a complex set of motions intended to achieve the desired goal for the autonomous mode of a given game. The state machine can control motor speeds and the positions of any appendages in order to achieve its autonomous goal.

It is advised that you use macros to give names to the state values, see

Section 9.3, making it easy to track things mnemonically, adding and deleting states at will while you refine your autonomous mode.

Glossary

analog input: A signal line on the robot controller or the operator interface that the computer is capable of reading the voltage of. For direct analog inputs on the robot controller a voltage of zero volts causes 0 to be read in the program; a voltage of five volts causes 1023 to be read. For an indirect analog input from the operator interface an input voltage of five volts provides a value of 255. In addition to this difference in scale, an input voltage that drops close to zero on the operator interface causes the value read by the program to suddenly jump to 127. This is done for safety reasons. Leave the normally grounded end of the 100K potentiometer open on the OI in order to avoid this problem.

atomic operation: A computer operation is said to be atomic if its constituent parts are not further dividable. No partially complete stages of an atomic operation are visible in a program. The PIC micro-controller is an eight bit processor, providing atomic loads and stores to memory locations that are declared with the single byte `char` type. Loads and stores of multiple byte `int` and `long` types are not atomic, with the processor loading and storing the multiple bytes composing these memory locations independently. The fact that loads and stores of memory locations wider than a `char` are not atomic becomes significant when writing code that accesses variables maintained in interrupt handlers.

binary: A representation of a number where the “digits”, also known as bits, are only 1, or 0. Computers represent numbers in this manner.

binary distribution: A compiled software package usually maintained by another programmer. A binary distribution contains executable files that are ready to run on a computer.

bit: The smallest instance of data that computer deals with, directly expressed in the voltage value found in a digital circuit. The voltage is “high” or “low,” indicating that the value of the bit is “1” or “0.” When we wire signals to the digital inputs of the robot controller, or operator interface, we are arranging to input bits.

byte: The smallest addressable memory data item. In all modern computers this is composed of 8 bits. In the C programming language, this corresponds to the `char` type.

call chain: Suppose `foo()` calls `bar()`, and `bar()` calls `spam()`. This sequence of calls is referred to as a call chain. The consequence of calling `foo()` is that the entire call chain is executed, that is the consequences of calling `foo()` include the consequences of calling all of the routines in the call chain.

character constant: The integer value associated with a printable character, represented for instance as `'a'` in a C program. The same integer value is used for the same character when it appears in a string constant, and this is the actual binary code that is signaled on the cable connected to a terminal or printer in order to make the character appear.

compiler: A computer program that converts a text based programming language that humans write into a machine language based module that the target computer can execute. The module is combined with other modules, if required, in order to resolve undefined variables and functions.

compilation unit: C programs are built by linking together a number of compilation units, or modules. When a file is compiled by the compiler a compilation unit is constructed, from the file being compiled, and all of the other files that are included by virtue of `#include` statements in the file being compiled.

data declaration: A statement that declares a named variable and associates with it a type, but does not allocate the actual memory to be used or initialize it.

data definition: A statement that declares a named variable, associates with a type, allocates storage and possibly provides an initial value for the variable. A data definition may stand alone, in that it provides the declaration.

data hiding: The scheme by which a local variable in a function, or an enclosed block of code, over-rides a more global variable that would otherwise be used when the given variable name is encountered by the compiler. Without data hiding, it would be impossible to use local variables for temporary values without running the risk of a name collision with a global variable, corrupting it.

data memory: The memory that stores the data that the central processing unit operates on. This memory is organized as an array of bytes, and the computer reads/writes sequences of bytes in memory in order to implement wider integer values or strings.

digital input: A voltage input line on the operator interface, or the robot controller, that can be read by the computer to produce a digital, single bit, value. The digital inputs on the RC and the OI have opposite semantics. On the OI you connect a switch between the input and ground, only. When the switch is closed, the corresponding digital input reads 1. The digital inputs on the RC are TTL compatible. You may connect a switch, or a TTL logic signal, to a digital input on the RC. When the switch is closed, or the TTL signal voltage is low, the corresponding digital input is read as 0. Otherwise 1 is read.

dirty tricks: A "trick of the trade" that is outside the documented specifications of the C programming language. A dirty trick is likely to run into problems with porting such code to other machines.

fast loop: The portion of the RC program that is repeatedly executed at high speed, outside of the packet cycle code where a packet from the OI is read, outputs adjusted, and a packet is sent back to the OI. The fast loop allows the RC to poll sensors on the robot at a lower latency than the 26.2 millisecond delay associated with the packet cycle.

function: A modular section of code that may be called by other functions. It allows the programmer to define a common operation that may be used from many places in the program. Typical functions may be handed arguments by the calling function, and may provide a returned value; but are not required to do so. This programming construct is also referred to as a subroutine, mostly when it does not return a value.

function declaration: A statement that provides a name for a function, and specifies arguments and returned value, if any. It allows the compiler to construct the calling interface for code that calls the function.

function definition: The extended section of program code that the compiler processes in order to produce the machine code associated with the function. Just as is the case for a data definition, if the compiler sees a function definition a separate function declaration is not required.

function prototype: A more detailed form of function declaration that

includes information about the number and types of function arguments. As far as the author is concerned, this is the only kind of function declaration that you should be using!

gremlin: *An imaginary mischievous sprite.* Not just any sprite, but one that you feel must be responsible for that unexplained fault you have just experienced with some device, especially a mechanical or electronic one. Pilfered from: <http://www.worldwidewords.org/weirdwords/ww-gre1.htm>, without permission. This site is well worth a visit.

hex: A base 16 format for unsigned integers, where the digits are 0-9,a-f. The digit 'a' corresponds to the value 10, the digit 'f' corresponds to the value 15. This format compresses the size of a formatted integer, and since the base is a power of two the formatting code does not involve any arithmetic. A number in hex is conventionally started with "0x", in order to remove any ambiguity should all the digits be less than 'a'.

high order bit: In the position value representation of a N-bit unsigned binary number, the high order bit is the one on the left that represents the largest power of two. In a N-bit twos-compliment signed integer, the high order bit also the one on the left, but this bit only indicates that the number is negative.

indirection: The act of getting at the actual data item that a pointer refers to.

in-line code: Code that accomplishes a task without resorting to a function call and its attendant overheads. If one finds that a given code pattern is being used repeatedly, one may be able to express this chunk of code with a macro that has the visible appearance of a function, but without the overhead.

interpreter: A computer program that interprets and executes a programming language. This kind of programming language is called an "interpreted language."

interrupt service routine (ISR): A subroutine in the robot controller program that is designated to service an interrupt. When the interrupt occurs, the processor jumps to the ISR that takes the action required in response to the interrupt.

keyword: A reserved name in a computer language that is indelibly tied to

a specific meaning in the language. Some examples in C are: `char`, `int`, `long`, `return`, `while`, `for`, `switch` and `break`. You are not allowed to use keywords of a language in variable names, etc...

library: A collection of modules that may be searched for definitions that are needed and linked into the program if a given module provides a needed definition.

linking/loading: The process by which modules are put together in order to construct a program with all data and functions being defined. In EiC, there is only one module. In C programs for the robot controller, the code we will write/modify will exist in several modules, and will use a number of modules from a library.

machine code: The instructions that the computer's central processing unit (CPU) reads in order to be told what data to operate on, and what to do with the results. In our robot controller, the code for the program is loaded into non-volatile memory that does not go away when the robot controller is powered down.

module: The program element produced when the C compiler processes a file. A module may have undefined functions or data that are to be found in other modules.

Murphy's Law: That which can go wrong, will.

newline: The character that causes output on a terminal or printer to trip to the start of the next line. It is turned into a carriage return and line feed by a printer driver. It is represented by the character sequence `"\n"` in a string, and `'\n'` as a single char.

normalize: Shifting the bits in a fixed, or floating, point result so that the value of the number is correctly represented in the chosen format.

operator: A mathematical operation that maps a number, or several numbers, to a new number. The C programming language has a rich set of operators available and there is very little functionality in a computer that can't be connected to with them, easily.

operator interface (OI): The operator interface provided by Innovation First that is used to send operator input to the robot controller using a radio modem or tether cable. See: <http://www.innovationfirst.com/FIRSTRobotics/>.

open loop: A method of control that simply applies the value returned from the joystick directly to the PWM output controlling the motor. This is how the drive is usually handled. If something like an arm is controlled this way, the relatively slow eye-hand coordination of humans tends to cause the arm to move much like a hammer. By using analog feedback control, the joystick can control the position of the arm, or the actual measured speed of the robot.

over-run: An integer type in C provides support for a range of values. If arithmetic causes an intermediate result, or stored result, to exceed the range of values supported by the type, we have an over-run, or range error, with corrupted results. This can be a severe disaster for motor control on the robot, where the motor will suddenly switch from forward to reverse at high speed, and possibly cause physical damage to the robot, or risk to students nearby. The term over-run also applies to scribbling out-of-bounds in an array reference, using an array index that is out of range.

packet cycle: The portion of the RC code that reads a packet from the OI, adjusts motors in response to operator controls, and then sends a packet back to the OI. This is handled by a routine that gets called every time the “fast loop” notices that a packet is available from the OI.

place value number format: A number system where the position of a digit in a string indicates the value that it represents. We take this for granted now, but it was not always so. The reader is encouraged to find and read books documenting the mathematics of early human history, and learn just how much got done without such a number system.

poll: The act of checking some input, repetitively, in order to spot the times when it changes value.

precedence rules: When a number of different operators are used in a complex expression, without parentheses to control the order of evaluation, a set of rules, called precedence rules, control the order of evaluation of the operators. For instance, the expression $a + b * c + d$ is evaluated by performing the multiply first, and then the two additions from the left to the right. The author uses parentheses, making it a point to not depend upon detailed knowledge of the precedence rules.

robot controller (RC): The robot controller provided by Innovation First that is used to control the robot.

See: <http://www.innovationfirst.com/FIRSTRobotics/>.

Pulse Width Modulation (PWM): A method of controlling motors where the voltage is either on, or off, and not in-between. The amount of power, or torque, is proportional to the percentage of time that the drive voltage is on. This style of control minimizes the power dissipated by the circuit controlling the power.

quantization error: When measuring a voltage with an analog to digital converter, the voltage is an analog value while the value produced by the analog to digital converter is the “closest value” that does not exceed the analog value. The error made in doing this is referred to as the quantization error.

rom: The C-18 specific keyword to place data in the same memory as is used for the code, this memory being read only memory (rom) in our micro-controller. String constants are automatically placed in code memory.

shift: An operation on an integer wherein the result is produced by shifting the bits in the binary representation of the number to the left, or right. It is equivalent to multiplying, or dividing, by powers of two, and is often the fastest way of multiplying/dividing by powers of two on a computer.

side effect: An expression, or function, might return a value and in addition might have, or not have, an additional side effect on other data in memory.

sizeof: In C, a special operator, `sizeof()` is provided to return the size, in bytes, of an object.

state machine: A style of program coding that is used to track progress to a goal, making it easy to organize what is to be done in order to reach the goal. A state machine is very useful in programming a robot for autonomous operation, but is can also used for other purposes. An example is network programming, where state machines are used to keep track of the stages of execution of a network protocol. State machines are also used to implement compilers.

storage class: A variable in a C program has a storage class associated with it, the default being controlled by just where in the program code the variable is defined. The storage class controls whether the variable can be seen in other modules, and whether the variable is persists between calls to a function where it was defined.

string: A sequence of bytes used to represent a string of, usually, printable characters. This sequence of bytes is terminated with a byte having the value zero, by convention, so the program knows where the end of the string is. This type of string is called a “null terminated” string in C. A string may be a constant, or may be a variable that is an array of char values in memory.

string constant: A constant null terminated string. The value of the expression associated with it is a pointer to the first character. This appears in the form “abcd”, in a C program where there are four characters and the terminating null in the string for this example.

subroutine: In C, these are formally called functions, but we tend to call functions that do not return a value a subroutine. This was the convention in Fortran.

tentative definition: A declaration of the form, `int foo = 5;`, outside of a function, is a definition. It allocates the storage for `foo` and provides its initial value. Two such definitions, either in the same file, or in different files, are not allowed. The possibly different initial values would be in conflict. A declaration of the form, `int foo;`, without the initialization, is a *tentative definition*. Most linkers will happily link all of these instances of `foo` to one memory location, with a zero initial value if no initialization is to be found.

troll: One method of fishing is to slowly drag your bait on a line behind the boat, waiting for a fish to strike. There are other interpretations of the word, but the authors are a fishermen.

type: The type of a variable controls how the computer will interpret the bits in the associated memory location.

type definition: A statement defines a new type to the compiler. A name is associated with the type, and the compiler is told what the type consists of. After the type is defined, it can be used just like a pre-defined type. We have avoided this topic, but very complex types can be defined in C and these types can be used to implement complex data structures.

upload: The act of pushing a file “up” into a server on a network, or pushing a program into the RC on the robot.

undefined behavior: This is computer speak for a malfunction of the computer program such that results will be unpredictable.

variable: A named memory location that stores data manipulated by the

computer program. A variable has a type, and an associated range of values.

wrapper: A function that is used to call a lower level utility function useful in more than one circumstance. It handles the simpler details before calling the lower level function.

References

- [1] Brian W. Kernighan, Dennis Ritchie and Dennis M. Ritchie, *C Programming Language (2nd Edition)*, Prentis Hall, 1988.
- [2] Bradley L Jones and Peter Aitken, *Sams' Teach Yourself C in 21 Days (Sixth Edition)*, Sams Publishing, 2002.
- [3] Tony Zhang and John Southmayd, *Sams' Teach Yourself C in 24 hours (Second Edition)*, Sams Publishing, 2000.
- [4] The documentation provided by Innovation First:
<http://www.innovationfirst.com/FIRSTRobotics/documentation.htm>
- [5] Frank Testa, *Fixed Point Routines*, Microchip Application Note AN617, www.microchip.com.
- [6] Kevin Watson, *RC and EDU example code*, <http://kevin.org/frc/>
- [7] Eugene Brooks, *Making your own PC boards*,
<http://srvhsrobotics.org/eugenebrooks/MakingPCBs.pdf> .
- [8] Chris Hibner, *Increasing A/D Resolution Using Noise*, white paper available at www.chiefdelphi.com, 11-07-2003.

Index

- absolute value, 19
- analog feedback control, 62
- binary, 32
 - fixed point arithmetic, 37
 - right shifts fail if negative, 35
 - shifts and multiply/divide, 33
 - twos-compliment, 33
- compound expression, 20
- conditional compilation, 50
- conditional expression, 19
- data
 - declaration consistency, 31
- dirty tricks, 39, 40, 87
- formatted output
 - for long integers, 45
 - in binary, 43
 - in decimal fixed point, 46
- function
 - caller, 15
 - declaration consistency, 31
 - declaration of arguments, 14
 - prototype, 14
- integrating a gyro, 66
 - using fixed point, 68
- interrupt
 - the gremlin, 72
 - the gremlin cure, 75
 - the gremlin demo, 74
- linkage, 29
 - deny by default, 31
 - denying for a function, 30
 - denying for a variable, 30
- macros
 - as mnemonics for constants, 49
 - macros in definitions of, 50
 - multiple line definitions for, 50
 - with arguments, 49
- printf(), 7, 40
 - cast char to int, 43
 - format specifications, 41
 - in the robot controller, 42
- quantization error, 68
- scope
 - of a variable, 29
- side effects, 20
- state machine, 83
- storage class, 28
 - automatic, 28
 - modifiers, 29
 - static, 28
- tentative definition, 30, 92
 - avoid by convention, 31
- volatile, 27, 72, 75