3

PROGRAMMING THE PROPELLER WITH SPIN

The Propeller can be programmed in several languages, including the Propeller's native languages, **Spin** and **Propeller Assembly (PASM),** as well and as C/C++. The Propeller Experiment Controller (PEC) software is currently available in Spin, so we will only discuss use of this language. Spin is a **high-level language** created specifically for the Propeller. High-level languages are designed to be easy to use, but do not directly give instructions to a microcontroller or computer. Instead, high-level languages must be transformed into a **low-level language** that is specific to the hardware being used. For the Propeller, the low-level language is PASM. The transformation from high-to low-level languages can be done by **interpreting** during operation, or by **compiling** before a program begins. Generally, interpreted code is slower than compiled code. Spin presents an interesting intermediate between an interpreted and a compiled language. Spin is initially compiled into bytecode, but the bytecode itself is interpreted as a program runs. PASM does not require interpretation and is thus much faster than Spin. However, as a low-level language, PASM is also generally harder to use. Fortunately, Spin programs can launch PASM programs in a new cog, allowing an easy-to-use Spin program to also implement fast, powerful PASM functions. The PEC uses this technique in several cases. The code for communication with an SD card, for example, is written in PASM and is very fast. However, it can be controlled by the user with simple and efficient Spin statements.

In the following sections, we will describe the basic concepts of Spin programming. Although Spin is generally not case-sensitive, we will use capitalization for emphasis and to help distinguish code from other text. As with the overview of

hardware, this will not be a comprehensive guide to Spin. The Propeller Manual (Martin, 2011) should remain the primary resource. This chapter can serve as a tutorial to Spin and programming in general. However, it is difficult to provide a simple "Hello World" tutorial on a microcontroller without first introducing other concepts. In order to demonstrate the principles described here, an understanding of connecting devices, such as buttons and LEDs, and displaying information to the Parallax Serial Terminal or an LCD screen will be required. Much of this is explained in later sections; the Parallax Serial Terminal is explained at the end of this chapter, while connecting buttons, LEDS, and LCD screens are described in Chapter 8. Therefore, to get the most out of this chapter, an initial reading of the chapter should be followed by attempts to demonstrate the programs described here. Additionally, returning to this chapter after becoming more familiar with the concepts described in later chapters will also be useful.

Program Blocks

Spin programs are built from six programing blocks, each with a unique function: CON, VAR, OBJ, PUB, PRI, and DAT. Blocks are specified simply by typing their name. Code within each block is typically indented to enhance readability. A program can have multiple types of each block, in any order, but generally follow the above ordering convention. Each program starts running with the first PUB block, and therefore each program must have one PUB block. All other blocks are optional. Within each block, a variety of instructions can be used, many specific to that block. For all blocks, no end-of-line characters are needed and instructions are generally not case-sensitive.

The **CON** block is also known as the constant block. In this section, the program defines **constants** that will never change during the program operation. The CON block can be used to define global configuration values and assign useful names to numerical values to make a program easy to read and modify. For example, a constant NumberOfTrials can define the number of trials (a single occurrence of the experimental protocol) that will occur in an experiment. In this example, NumberOfTrials can be defined as the number 10. Any portion of the program that refers to the constant NumberOfTrials actually uses the numeric value of the constant, in this case 10. If the number of trials in the experiment ever needs to be modified, changing the value of the constant also updates every reference to the constant in the program. Constants are also often similarly used to define I/O pin connections. The system clock speed is also set in the CON block using the _clkmode and _xinfreq instructions. Note that these clock speed instructions are case-sensitive.

The **VAR** or variable block is used to reserve space for variables that will change during the program. **Variables** are different from constants as variables can change, while constants cannot. For example, a variable ResponseLatency can

be created to store the time until the first response during each trial. ResponseLatency for each trial can then be saved in a data spreadsheet or used by other parts of the program. The size of a variable is also assigned in the VAR block. In Spin, variables can be of several sizes. **Byte** variables range from 0 to 255 and are best for values that will remain very small. **Word** variables are larger, ranging from 0 to 65,535. **Long** variables are the largest and range from –2,147,483,648 to +2,147,483,647. Most variables in experiments will be longs. As there is limited memory on the Propeller chip, it is beneficial to use smaller sized variables when possible, however it is unlikely that an experiment will use the Propeller's entire memory. It is important to remember that, in the VAR block, variables are only assigned a name; values will be assigned at a later time.

The **OBJ** or object block is used to declare **objects** that will be used by the program. Each object is another program that can be used by the current program. Use of objects keeps programs organized and makes it easier to share common code between multiple programs. Each instance of an object is given a short reference name, then these abbreviations can be used to refer to code inside the objects. Using objects written by others can also save time and allow access to complicated functions that may be difficult to create. Many objects specialized for certain tasks, such as communication protocols, mathematical functions, or speech emulation can be obtained, for free, on Parallax's **Propeller Object Exchange**. These objects are created by Parallax staff and other Propeller users, both professionals and hobbyists.

The **PUB** or public block is used to define code that can be used in a program or by other programs. Code in PUB blocks are known as **methods**. Each method in a program has a unique name listed after the PUB statement. Each program always begins with the first public method, commonly named "Main".[1] A program will only execute code in the first method unless it is explicitly told to start another method. Although programs only require one public method, dividing a program into multiple methods makes it easier to read and allows a program to efficiently reuse a section of code. If one program imports another program as an object, the first program can use any of the object program's public methods. These methods are public, and thus free to use. Any variables that were declared in a VAR block can be assigned values in a public method.

The **PRI** or private block is identical in function to a PUB block with two exceptions. First, a program must contain one public method to run. A program with only a private method will not run. Second, a program cannot use its objects' private methods. These methods are private, and thus can only be used by the program containing them. Generally, private methods are used to

[1] Note that while it is grammatically correct to include punctuation within quotations, in this document we give priority to preserving clarity of code. In cases where quotation marks are used to refer to a specific aspect of code, such as a method name or a string, punctuation is placed outside the quotation marks.

ensure that another program does not accidentally affect a background process in an object. For example, the Propeller Experiment Controller software uses an object with a private experimental clock method. The clock method is private so that the user never accidentally interferes with the experimental clock. Instead, public methods are available to safely interact with the clock.

The **DAT** or data block is used to store data in a program. Similar to a VAR block, all data will be byte-, word-, or long-sized. However, DAT blocks differ from VAR blocks in a few important ways. First, data in DAT blocks can be both named and assigned a value. Second, DAT blocks are optimized to store large chunks of data, and therefore the syntax for declaring data in a DAT block is different than that of VAR blocks. Third, for programs used as objects, each instance of an object will have unique variables but identical data. This is an advanced but powerful distinction that can be used for many effects. For example, an object may be created to communicate with a heart rate monitor. An experiment program may use several heart rate monitor objects to interface with multiple monitors. Each heart rate monitor object can record data from its monitor into individual VAR blocks, while all the objects can share important information about interfacing with the monitors in the DAT block. Finally, the DAT block is also used to store PASM code.

In any block, **code comments** can be used to clarify the purpose of the block, or of individual lines of code. In Spin, in-line comments are provided after a single quotation mark (e.g., 'in-line comment), while multiline comments are provided between curly brackets (e.g., {multiline comment}). Informative code comments are also provided within the code of many programs.

Data Types

Spin uses several data types to represent data in CON, VAR, and DAT blocks. Table 3.1 provides a quick overview of the data types that can be used in each block. The data types will be briefly defined here with the following sections elaborating on implementation as more programing concepts are introduced. **Integers,** or non-fractional numbers, can be declared as byte-, word-, or long-sized in VAR or DAT blocks. In CON blocks, no size-declaration is required. Fixed-length integer **arrays** may also be used by declaring the size of the array in a VAR or DAT block. Each integer in the array will be byte-, word-, or long-sized and can be changed as needed. However, no new items may be added to the array. Floating point numbers, or **floats**, are numbers with a fractional component. These are more difficult for computers and are not supported by many microcontrollers. The Propeller, however, does support float constants in the CON block, and float objects are available to allow the Propeller to implement float variables and mathematics when required. The simplest solution when working with fractional numbers in many microcontrollers, is simply to scale

the numbers up until fractions are not a concern. For example, when recording temperature with one decimal place, such as 98.6°F, all values could be multiplied by 10 to remove the need for float mathematics. If data are later transferred to a computer, the integer numbers can be easily transformed back into floats.

Table 3.1
Data Types

Data Type	CON	VAR	DAT
Integer	Yes	Yes	Yes
Integer Array		Yes	Yes
Float	Yes		
String		Yes	Yes

Spin also supports integer numbers in several formats. What we think of as traditional numbers are known as decimal (base 10) numbers in mathematics and computer science. Unless otherwise noted, all numbers in Spin are decimals. Binary numbers can be used by placing a percent sign before the number; for example, %101 is a binary number equivalent to the decimal number 3. Hexadecimal numbers can be similarly noted with the $ symbol, such as $4E. For all numeric data types, an underscore can be used to break up a number to increase readability. For example, the number 123456 can be written as 123_456 or 12_3456 without changing the way the number is interpreted.

Byte-sized numbers can also be interpreted as **characters**. Characters are any symbol that can be displayed on a screen, including numeral characters, the alphabet, and other symbols. For example, the decimal number 65 can be interpreted as the character "A". In many programing languages, including Spin, quotation marks are used to differentiate characters from variables and numbers. For example, A + 1 may refer to the variable or constant A, plus the number 1. However, "A" + "1" refers to the character "A" plus the character "1". Although byte-sized numbers can be interpreted as characters, they are still treated as numbers in mathematical operations such as addition and subtraction. For example, the instruction "A" + "1" is really just another way of stating 65 + 49. Some characters (decimals 0 to 31, and 127) also have special purposes that vary between applications. The decimals 11 and 13, for example, often indicate the start of a new line in text documents. Appendix C provides ASCII character charts that show the decimal and hexadecimal representations for each character.

Spin also supports **string** data. Strings are a data-type used to represent text and can be considered a string of characters. In Spin, strings are denoted with plain, double quotation marks, such as in the string: "string data". It is important to note that some programs automatically change quotation mark characters. For example, Microsoft Word (Microsoft Corporation; Redmond, Washington)

may automatically change "string data" into “string data”. The second quotation mark format will not work in most programming languages. Be aware of this when attempting to use code originally written in a word processor. Each character in a string is an individual byte. For strings, a special character is needed to indicate end of the string. Spin uses null-terminating strings that always end with the null character, 0. To put this into perspective, consider the string "Hello!" Note that the string is contained within quotation marks, a common approach for programing languages. This string can be represented in an array of bytes as 72, 101, 108, 108, 111, 33, 0. Each byte in the preceding list represents one of the characters in "Hello!" with the final byte, 0, indicating the end of the string. In Spin, string variables must be word-sized. The variable will not contain the string itself but will instead contain the location of the string in memory using a word-sized address.

In Spin, a program and all related variables are stored in the central hub’s 32 kB of RAM. Each byte in RAM, from byte 0 to byte 32,767, can be addressed individually using a variety of instructions. The range of 0 to 32,767 is best represented by a word-sized variable, therefore when the location of a byte is needed, a word-sized address is usually used. For most purposes, the address of variables in memory is not a major concern. However, it does become important with certain types of data and operations, particularly when using strings.

Assigning and Manipulating Data

Table 3.2
Common Operators

Operator	Use	Description
=	X = Y	Constant assignment
:=	X := Y	Variable assignment
+	X + Y	Add
-	X – Y	Subtract
*	X * Y	Multiply
/	X / Y	Divide
++	X ++	Increment
--	X --	Decrement
//	X // Y	Modulus
#>	X #> Y	Limit minimum
<#	X <# Y	Limit maximum
^^	X = ^^Y	Square root
||	X = ||Y	Absolute value

Spin uses a variety of **operators** to assign values to data types and perform mathematical operations. Table 3.2 provides descriptions of some common assignment and mathematical operations. One important note is that the assignment operator for constants differs from the variable assignment operator. Values for constants are assigned in the CON block using the = constant assignment operator (see Figure 3.1). The constants may also be assigned in list form (see Figure 3.2). To assign values to variables in PUB or PRI blocks, use the := variable assignment operator. Other operators, like those used for mathematics, are relatively straightforward and resemble those used by many other languages.

```
CON
    X = 1
    Y = 2
VAR
    BYTE Z
    BYTE DataArray[3]
 PUB Main
    Z := X + Y
    DataArray[0] := X
    DataArray[1] := Y
    DataArray[2] := Z
```

Figure 3.1: Assignment example program 1.

Figure 3.1 shows a simple example of assignment and mathematical operations. In this example, the constants X and Y are declared and assigned values in the CON block. The variables Z and DataArray are declared in the VAR block. Note that both are byte-sized variables, thus Z and each of the 3 bytes in DataArray can range from 0 to 255. In the public method, Main, Z is assigned to be the sum of X and Y and is therefore 3. X and Y are constants and cannot be assigned new values. Each byte in the array, DataArray, is also assigned a value by referring to the index of that byte. Note that the index starts at 0 instead of 1, this is a common trend to programming. The values of DataArray become 1, 2, and 3.

VAR and DAT blocks use slightly different syntax for integers and arrays. This is because DAT blocks are designed to contain large amounts of similarly sized data. In many cases, the use of VAR or DAT block is a matter of preference. Figure 3.2 is an alternative to the code seen in Figure 3.1. The VAR block has been replaced with a DAT block, and the PUB block is now empty. All data are declared and assigned in the new DAT block. The constants in this example are also assigned in list form. Although assigning multiple constants in list form reduces the number of lines in a program, it may make those lines more difficult to read. This technique is best used for a group of related constants.

```
CON
    #1, X,          #2, Y
PUB Main

DAT
    Z               BYTE            X+Y
    DataArray       BYTE            X, Y, Z
```

Figure 3.2: Assignment example program 2.

String assignment and manipulation is a little more complex. If using a combination of VAR and PUB blocks, a word-sized variable must first be declared in the VAR block. The variable will contain the location of the string in memory. In the PUB block, the STRING instruction can then be used to create a string and assign the string's address to a variable. Individual characters in the string can then be manipulated using the BYTE instruction. Figure 3.3 shows an example of these string techniques. In the VAR block, A is declared as a word-sized variable and B is declared as an array of 4 bytes. In the public method, Main, A is set to the string "Hello!", then the bytes in B are assigned values. At the end of the program, the B can be interpreted as the string "leo", or as a byte array containing the bytes 108, 101, 111, 0.

```
VAR
    WORD A
    BYTE B[4]
PUB Main
    A := STRING("Hello!")
    B[0] := BYTE[A][2]
    B[1] := BYTE[A][1]
    B[2] := BYTE[A][4]
    B[3] := 0
```

Figure 3.3: Assignment example program 3.

Figure 3.4 shows the same result using a DAT block. In this case, the DAT block is much more concise than the VAR block. Also, notice that A is defined as a byte array instead of a word-sized variable. This is because in Figure 3.3, A is a word-sized variable that can be used to represent *any* address in memory. For the string variable A, the address of the first byte of the string is saved using the STRING instruction. In Figure 3.4, A is a byte array, and A already refers to the first byte of the string.

```
PUB Main

DAT
    A   BYTE    "Hello!", 0
    B   BYTE    A[2], A[1], A[4], 0
```

Figure 3.4: Assignment example program 4.

I/O Pins

One of the major benefits to using microcontrollers is the way they interface with external devices. In Spin, several registers, DIRA, INA, and OUTA, are used to control the I/O pins. **DIRA** is the direction register and is used to set the I/O pins to input or output mode. Although it can be modified in several ways, the simplest manner is to use the pin number as an index and set it to 0 for an input, or 1 for an output. For example, the instruction DIRA[5] := 0 sets I/O pin 5 to an input. The input state of pins can be read in a similar manner with the **INA** instruction. For example, the instruction x := INA[5] assigns variable x to be the input state of pin 5, 0 for off and 1 for on. For pins in output mode, the **OUTA** instruction can be used to turn the output of a pin off or on, similar to how the DIRA instruction can change a pin's mode. In order to use a pin as an output, it must first be set to output mode using the DIRA instruction. Each cog has independent I/O registers. A cog can only control an I/O pin if its DIRA register has been set accordingly. To prevent interference between cogs, all pins are inputs by default. A pin will remain in input mode only if no cog sets it to an output. However, a pin enters output mode if any cog sets it to an output.

Figure 3.5 shows an example of using the I/O instructions. In the CON block, the pin numbers for three LEDs are saved. Recording pin numbers in the CON block is optional, but greatly increases program readability and makes the program easier to alter. In the PUB block, the direction for LED pins 1 and 2 are set to output mode, and all LED pins are turned on. LED3, however, will not turn on as it was never set to output mode using the DIRA instruction.

```
CON
    LED1 = 5
    LED2 = 4
    LED3 = 3
PUB
    DIRA[LED1] := 1
    DIRA[LED2] := 1
    OUTA[LED1] := 1
    OUTA[LED2] := 1
    OUTA[LED3] := 1
```

Figure 3.5: I/O control example program.

Time Control

Spin uses several instructions to record time and to execute code at certain times. Most important are the _clkmode and _xinfreq instructions. In the CON block, these instructions are used to set the system clock mode. Most Propeller development boards have a 5 MHz crystal oscillator that helps to maintain very precise system time. The _clkmode and _xinfreq instructions tell the Propeller how to use the crystal oscillator. Although these techniques can provide advanced control, it is best to use lines of code seen in Figure 3.6 in any program where precise control of time is required (provided the Propeller is connected to a 5 MHz crystal). Unlike most instructions in Spin, these two instructions are case-sensitive.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
```

Figure 3.6: Time control example program.

During operation, the **CNT** instruction can be used to read the value of the system counter. However, CNT can be a little arbitrary and unintuitive as the system counter is not a standard unit of time. The CLKFREQ instruction provides a good solution. It refers to the number of CNTs in 1 second. These two instructions are most commonly used with the WAITCNT instruction. **WAITCNT** is used to pause a cog's activities until CNT reaches a certain value. The paused cog will not proceed past that line of code; however other cogs will function normally. For example, WAITCNT(CLKFREQ * 2 + CNT) tells the Propeller to pause until CNT reaches twice the number of counts in

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    LED = 10
PUB Main
    DIRA[LED] := 1
    OUTA[LED] := 1
    WAITCNT(CLKFREQ * 2 + CNT)
    OUTA[LED] := 0
    WAITCNT(CLKFREQ * 2 + CNT)
    OUTA[LED] := 1
    WAITCNT(CLKFREQ * 2 + CNT)
    OUTA[LED] := 0
```

Figure 3.7: I/O and time control example program.

a second, plus the current value of CNT. In other words, pause for 2 seconds. As this method can be a little cumbersome for beginners, the PEC offers easier alternatives.

Figure 3.7 shows an example of I/O and time control. The clock mode and frequency are set in the CON block, and the pin number for an LED is saved. In the Main method, the LED is set to an output. The LED turns on, the cog pauses for 2 seconds, turns the LED off and then pauses for another 2 seconds. This process repeats a second time.

Program Flow Control

Program flow in Spin can be controlled by conditionals and iteration (see Table 3.3). **Conditionals** execute a section of code if a statement is evaluated to be true. Spin, like many languages uses **IF** conditionals. IF statements use **Boolean operators** (see Table 3.4) to evaluate a statement. If the statement is true, an indented section of code is executed. If the statement is not true, an ELSEIF statement may evaluate additional statements. If none of the statements are true, an ELSE statement may be used to execute code. Consider the example in Figure 3.8. The CON block is used to set the clock mode, record pin numbers, and set a value for X. In the Main method, all pins are set to outputs, then an IF conditional is used to control the program flow. If X is 1, LED1 is turned on. Otherwise, if X is 2, LED2 is turned on. If neither of these is true, then LED3 is turned on. In this manner, different values of X can change the way the program runs.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    LED1 = 1
    LED2 = 2
    LED3 = 3
    X = 1
PUB Main
    DIRA[LED1] := 1
    DIRA[LED2] := 1
    DIRA[LED3] := 1
    IF X == 1
        OUTA[LED1] := 1
    ELSEIF X == 2
        OUTA[LED2] := 1
    ELSE
        OUTA[LED3] := 1
```

Figure 3.8: IF conditional example program.

Spin also uses **CASE** conditionals that compare the value of a statement to several potential values and execute code that matches the current value. If a matching value is not found, code in an OTHER section is executed. The end result is similar to that of IF conditionals. Figure 3.9 shows a CASE implementation of the program in 3.8. For most applications, the choice of IF or CASE conditionals is a matter of preference.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    LED1 = 1
    LED2 = 2
    LED3 = 3
    X = 1
PUB Main
    DIRA[LED1] := 1
    DIRA[LED2] := 1
    DIRA[LED3] := 1
    CASE X
        1: OUTA[LED1] := 1
        2: OUTA[LED2] := 1
        OTHER: OUTA[LED3] := 1
```

Figure 3.9: CASE conditional example program.

Iteration, or repetition of a section of code, is implemented through **REPEAT** statements (see Table 3.3). Unlike many languages that use different instructions for each type of iteration, Spin uses variants of REPEAT loops for all forms of iteration. To repeat a section of indented code forever, simply use REPEAT with no qualifiers. To repeat a section of code a specific number of times, use the REPEAT X form, where X is the desired number of iterations. The repeat loop may also iterate a variable, X, through a specified ranged, Y to Z, by using the form REPEAT X FROM Y TO Z. Repeat loops may also execute code UNTIL an evaluation is true or WHILE an evaluation is true. For any form of repeat loop, the NEXT statement can be used to skip to the next iteration, while the QUIT statement can be used to exit the loop entirely.

Figure 3.10 shows an example of an infinite repeat loop. The CON block and first lines of the Main method are used to setup the system clock and I/O pins. Then, the repeat loop begins and an IF conditional inside the loop evaluates the state of the button. If the button is off, the LED is turn off. Otherwise the LED is turned on. This repeat loop will continue until the Propeller is turned off, resulting in the LED being on only when the button is held.

Figure 3.11 shows a more complex example of a repeat loop. In this example, constants are not used to name the I/O pins. Instead the DIRA and OUTA instructions refer to the pin numbers directly. The first lines of the Main method set pins 1 to 3 as outputs. The outer repeat loop will be repeated ten times, but it also contains another repeat loop. The inner repeat loop increments the variable X from 1 to 3. The inner loop also turns the output pin represented by X on, then

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    LED = 1
    Button = 2
PUB Main
    DIRA[LED] := 1
    DIRA[Button] := 0
    REPEAT
        IF INA[Button] == 0
            OUTA[LED] := 0
        ELSE
            OUTA[LED] := 1
```

Figure 3.10: REPEAT loop example program 1.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
VAR
    BYTE X
PUB Main
    DIRA[1] := 1
    DIRA[2] := 1
    DIRA[3] := 1
    REPEAT 10
        REPEAT X FROM 1 TO 3
            OUTA[X] := 1
            WAITCNT(CLKFREQ/4 + CNT)
        OUTA[1] := 0
        OUTA[2] := 0
        OUTA[3] := 0
```

Figure 3.11: REPEAT loop example program 2.

waits a quarter second before repeating again. This loop will repeat three times, then all output pins are turned off. The program flow then returns back the outer repeat loop to repeat the code nine more times. As a result of this program, the Propeller will turn on pins 1 to 3, a quarter-second apart, then turn all of them off. This process repeats ten times.

Table 3.3
Conditional and Iteration Instructions

Instruction	Description
IF	If conditional
ELSEIF	Else if — use with if conditional
ELSE	Else — use with if conditional
CASE	Case conditional
OTHER	Other — use with case conditional
REPEAT	Repeat
REPEAT X	Repeat X times
REPEAT X FROM Y TO Z	Repeat and increment X from Y to Z
REPEAT UNTIL X == Y	Repeat until a condition is true
REPEAT WHILE X == Y	Repeat while a condition is true
NEXT	Go to the next iteration of a repeat loop
QUIT	Quit a repeat loop

Table 3.4
Boolean Operators

Operator	Use	Description
==	X == Y	Is equal
<>	X <> Y	Is not equal
<	X < Y	Is less than
>	X > Y	Is greater than
=<	X =< Y	Is equal or less
=>	X => Y	Is equal or more
AND	X == Y AND X == Z	And
OR	X == Y OR X == Z	Or
NOT	X == Y NOT X == Z	Not

Methods

Many programs contain repetitive sections of code that can be reused if moved to modular methods. A method is simply a section of code that executes a task. So far, we have only described single-method programs, but most programs will use several methods. Adding a new method is as simple as adding a new PUB block. Consider the example in Figure 3.12. This program uses the Pythagorean theorem to calculate the hypotenuse, C, from the sides A and B. The program does this three times using similar instructions. It is clearly repetitive. Note that, to simplify this example, numbers have been chosen that result in an integer hypotenuse so that no float mathematics are required.

```
CON
    _clkmode = xtal1 + pll16x
    xinfreq = 5_000_000
VAR
    BYTE A
    BYTE B
    BYTE C
PUB Main
    A = 5
    B = 12
    C = ^^(A*A + B*B)
    A = 12
    B = 9
    C = ^^(A*A + B*B)
    A = 6
    B = 8
    C = ^^(A*A + B*B)
```

Figure 3.12: Pythagorean theorem program example 1.

To remove repetition from a program, a second method may be implemented. Secondary methods allow a program to efficiently reuse a section of code. They will not run unless called by the primary method. The primary method may also pass **parameters**, or information, to the secondary methods, and in turn, the secondary method may optionally use the **RETURN** instruction to return a result to the primary method on completion. Figure 3.13 shows a much more concise variation of the program in Figure 3.12 using a second method. As with Figure 3.12, the program starts with the first listed method, in this case Main. The Main method then calls the FindHypotenuse method and passes the integers 5 and 12 as the parameters A and B. The FindHypotenuse method calculates the hypotenuse and returns the result to the Main method. The Main method then calls the FindHypotenuse method two more times with different parameters.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
PUB Main
    FindHypotenuse(5, 12)
    FindHypotenuse(12, 9)
    FindHypotenuse(6, 8)
PUB FindHypotenuse(A, B)
    RETURN ^^(A*A + B*B)
```

Figure 3.13: Pythagorean theorem program example 2.

Multitasking

Distributing a program across multiple methods is a common programing technique. Where the Propeller excels, however, is in executing these methods across multiple cogs. The **COGNEW** statement can be used to launch a method in a new cog, allowing the first cog to continue executing its method. As cogs will need some space to run a method, the user must supply a temporary workspace,

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    LED = 1
    Button = 2
VAR
    LONG MS
    LONG CogStack[50]
PUB Main
    COGNEW(Blink, @CogStack)
    REPEAT
        IF INA[ButtonPin] == 0
            MS := 2000
        ELSE
            MS := 500
PUB Blink
    DIRA[LED] := 1
    REPEAT
        OUTA[LED] := 0
        WAITCNT(CLKFREQ/1000 * MS + CNT)
        OUTA[LED] := 1
        WAITCNT(CLKFREQ/1000 * MS + CNT)
```

Figure 3.14: Multi-cog example program 1.

called a stack space, as an array of longs. The required stack space is difficult to determine. Typically, a large space is initially provided and decreased later if the program needs more memory. Parallax's Stack Length object (Martin, 2010) can also be used to determine optimal stack size.

Figure 3.14 shows an example implementation of a multi-cog approach. The CON block sets up the system clock and saves pin numbers for an LED and button. The VAR block declares a long-sized integer called MS; this variable will be used to represent a millisecond value. The VAR block also creates a long array called CogStack for the second cog to use as a stack space. All cogs will have access to the constants and variables. The program starts with the first method, Main. The Main method launches a new cog to run the Blink method and provides CogStack as its stack space. Note the **@ operator**. This lets the new cog know to look at the address of CogStack in memory, instead of the value of the first long in the array. The Main method then repeatedly checks the input state of the button. If it is off, MS is set to 2000, if it is on, MS is set to 500. Meanwhile, the Blink method in the second cog sets LED to an output. Then, it repeatedly turns the pin off, pauses, turns the pin on and pauses again. The WAITCNT statement pauses for CLKFREQ/1000 * MS + CNT. The use of CLKFREQ/1000 transforms the system count into milliseconds, and when multiplied by MS, will cause the cog to pause for MS milliseconds. Taking the methods of both cogs into consideration, the result of this program is an LED that blinks once every 2 seconds if a button is not pressed, or once every half second if a button is pressed.

The example in Figure 3.14 is a relatively simple example of multitasking. A clever programmer could easily find a way to implement the same function within a single cog. However, with eight cogs, there is no harm in multitasking for the sake of convenience, and some complex tasks are simply not possible to implement simultaneously in a single cog. Other instructions, such as COGID, COGINIT, and COGSTOP can be used to more precisely start and stop the activity of individual cogs. For example, the program in Figure 3.15 is very similar to that of Figure 3.14, except that the blinking cog will turn off after 100 blinks.

Objects

In addition to using multiple methods within a program, Spin also enables the use of methods from other programs. These secondary programs are called objects. Often, an object contains a library of methods dedicated to a specific task, such as float mathematics, audio generation, or I^2C communication protocols. A program may use several objects, or multiple instances of the same object. Object programs may also call on other objects themselves. The result is a very modular approach to programing, and the ability to implement advanced techniques through objects written by Parallax staff and other experienced programmers. Many objects are available on Parallax's Propeller Object Exchange.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    LED = 1
    Button = 2
VAR
    LONG MS
    LONG CogStack[50]
PUB Main
    COGNEW(Blink, @CogStack)
    REPEAT
        IF INA[ButtonPin] == 0
            MS := 2000
        ELSE
            MS := 500
PUB Blink
    DIRA[LED] := 1
    REPEAT 100
        OUTA[LED] := 0
        WAITCNT(CLKFREQ/1000 * MS + CNT)
        OUTA[LED] := 1
        WAITCNT(CLKFREQ/1000 * MS + CNT)
     COGSTOP
```

Figure 3.15: Multi-cog example program 2.

In addition to reading the descriptions online, open the objects and read the code comments. Often the code comments contain detailed information about how to use the objects.

Objects are imported into a program using the OBJ block. Each object is given its own name. Public methods within that object can then be used by the instruction Object.Method, where Object is the name given in the OBJ block, and Method is the name of a public method within the object. A program cannot, however, use an object's private methods. Figure 3.16 shows an object-based example of the Pythagorean theorem program discussed earlier. In this example, the program MainProgram.spin imports the separate file PythagoreanTheorem.spin as an object. Then, the main program uses the methods of the PythagoreanTheorem.spin file.

Parallax Serial Terminal

One especially useful object is the **Parallax Serial Terminal** (Martin, Lindsay, and Gracey, 2009). This object allows a Propeller to communicate with the

```
MainProgram.spin
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
OBJ
    Pyth: "PythagoreanTheorem"
PUB Main
    Pyth.FindHypotenuse(5, 12)
    Pyth.FindHypotenuse(12, 9)
    Pyth.FindHypotenuse(6, 8)

PythagoreanTheorem.spin
PUB FindHypotenuse(A, B)
    RETURN ^^(A*A + B*B)
```

Figure 3.16: Pythagorean theorem program example 3.

Parallax Serial Terminal application, or other serial terminal application, on a personal computer, typically by using the same cable used to program the Propeller. The combination Parallax Serial Terminal object and application can be used to display information from the Propeller on a computer screen for many purposes, including transmitting data from the Propeller, testing programing concepts, or locating and correcting errors in a program. The Parallax Serial Terminal object may already be available in the object library folder (see the Software section of Chapter 4). If not, it can be downloaded from the Propeller Object Exchange.

To use the Parallax Serial Terminal as an object, first import the object. Then, use the Parallax Serial Terminal's Start method. The Start method requires a baud rate, or the rate that data are passed between the Propeller and the computer, as a parameter. The default baud rate is 115,200 bits per second. Make sure the serial terminal application also is set to the same baud rate. To use a fast communication rate like this, the clock mode needs to be set in the constant section. Note that the Parallax Serial Terminal does use one cog. Figure 3.17 shows the basic requirements of using the serial terminal.

Now that the Parallax Serial Terminal is running, a variety of methods can be used to send information from the Propeller to the computer. Each method should be preceded by the object reference. In this case, we will use the convention PST.Method. The PST.Char method prints a single byte character on the serial terminal. Remember that certain characters have special properties. For example, the instruction PST.Char(13) will cause the serial terminal to start a new line. The PST.CharIn method will allow the Propeller to receive a character typed on the serial terminal. Similarly, the PST.Dec and PST.DecIn methods

allow the serial terminal to print a decimal number, or the Propeller to receive a decimal number from the serial terminal.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
OBJ
    PST: "Parallax Serial Terminal"
PUB Main
    PST.Start(115_200)
```

Figure 3.17: Parallax Serial Terminal example program 1.

The Parallax Serial Terminal also allows the Propeller to display entire strings on the computer using the PST.Str method. Recall that strings are actually arrays of byte characters that always end with the null character, 0. The PST.Str method requires the address in memory of the first character of the string. It will display all the characters in the string, starting with the first, and stop displaying characters when it reaches the null character. For a one-time use string, the instruction PST.Str(STRING("Your string here")) can be used. The STRING instruction inside PST.Str provides the appropriate address in memory required by the Parallax Serial Terminal. For string variables, in the VAR or DAT section, the @ operator can be used to tell the Parallax Serial Terminal to look for the address of the variable in memory, instead of the data value at that address. For example, the instruction PST.Str(@StringVariable) can be used to print string variables. Several other methods are available to control other aspects of the Parallax Serial Terminal, such as starting new lines, changing the cursor positions, clearing the screen, etc. Open the Parallax Serial Terminal object and read the method descriptions for more information.

The Parallax Serial Terminal is an excellent tool for demonstrating and debugging programs. It can also be used to test many of the previous examples from this chapter. Consider the Pythagorean theorem example from Figure 3.16. The Parallax Serial Terminal could be included in the program to display the results of the program on the computer. Figure 3.18 shows just one potential way to integrate the Parallax Serial Terminal. In this example, the Parallax Serial Terminal is imported and started. A 5-second delay is then included to ensure that the user has the Parallax Serial Terminal application on the computer ready to display information. The serial terminal then prints information about the hypotenuse of each of the three triangles from the example in Figure 3.16. After completion, the Parallax Serial Terminal stops, freeing that cog again. Figure 3.19 shows the results of the program in Figure 3.18.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
OBJ
    Pyth: "Pythagorean Theorem"
    PST: "Parallax Serial Terminal"
PUB Main
    PST.Start (115_200)
    WAITCNT(CLKFREQ*5+CNT
    PST.Str((STRING("Starting program."))
    PST.NewLine
    PST.Str((STRING("Hypotenuse of 5 and 12 = "))
    PST.Dec(Pyth.FindHypotenuse(5, 12))
    PST. NewLine
    PST.Str((STRING("Hypotenuse of 12 and 9 = "))
    PST.Dec(Pyth.FindHypotenuse(12, 9))
    PST. NewLine
    PST.Str((STRING("Hypotenuse of 6 and 8 = "))
    PST.DEC(Pyth.FindHypotenuse(6, 8))
    PST.Stop
```

Figure 3.18: Parallax Serial Terminal example program 2.

```
Starting program.
Hypotenuse of 5 and 12 = 13
Hypotenuse of 12 and 9 = 15
Hypotenuse of 6 and 8 = 10
```

Figure 3.19: Parallax Serial Terminal output from Figure 3.18.