Chapter 3: Propeller Programming Tutorial

This chapter assumes you are familiar with the general programming concepts of other programming languages, including object-oriented languages. Discussion of some basic concepts will be presented, but some prior knowledge and programming experience is recommended.

In addition to the above, this material should be read only after reading Chapters 1 and 2. If you have not read through all of Chapter 1 and at least most of Chapter 2, please do so before continuing with this chapter. Many items presented in Chapters 1 and 2 will be referred to here, but will not be described in detail.

The following Propeller Programming Tutorial describes Propeller chip programming concepts in a step-by-step fashion with quick review notes along the way. It is best to read this chapter from its start to its finish, without skipping around, while working with your computer and the Propeller chip and trying each example as it is taught. The earlier exercises are basic in nature and each later exercise covers more advanced material.

Concept

The Propeller product (hardware, firmware and software) was designed with many well-known and also many brand-new concepts in mind. To this end, we designed the hardware, firmware, software and the two programming languages that go with it (Spin and Propeller Assembly) completely from scratch to give users the most direct and efficient control over the Propeller device.

To fully understand and utilize these tools and languages, it is best that you approach it all with an open mind. In other words, please be careful not to let legacy programming concepts and methods prevent you from experiencing the advantages made available by the Propeller chip and its programming languages. We believe that some legacy concepts do not belong in true real-time processing environments since they tend to bring turmoil to those who rely on them.

Propeller Languages (Spin and Propeller Assembly)

The Propeller chip is programmed using two languages designed specifically for it: 1) Spin, a high-level object-based language, and 2) Propeller Assembly, a low-level, highly-optimized assembly language. There are many hardware-based commands in Propeller Assembly that have direct equivalents in the Spin language. This makes learning both languages, and the use of the Propeller chip overall, much easier to handle.

The Spin language is compiled by the Propeller Tool software into tokens that are interpreted at run time by the Propeller chip's built-in Spin Interpreter. Those familiar with other programming languages usually find that Spin is easy to learn and is well-suited for many applications. With Spin you can easily perform high-level/low-bandwidth tasks and can even create code to handle some typically higher-bandwidth features like asynchronous serial communication at 19200 baud.

The Propeller Assembly language is assembled into pure machine code by the Propeller Tool and is executed in its pure form at run time. Assembly language programmers enjoy Propeller Assembly's nature and its ability to achieve high-bandwidth tasks with very little code.

Propeller Objects (see below) can be written entirely in Spin or can use various combinations of Spin and Propeller Assembly. It's possible to write objects almost entirely in Propeller Assembly as well, but at least two lines of Spin code are required to launch the final application.

Propeller Objects

The Propeller chip's Spin language is object-based and serves as the foundation for every Propeller Application.

What is an Object?

Objects are really just programs written in a way that: 1) create a self-contained entity, 2) perform a specific task, and 3) may be reused by many applications.

For example, the Keyboard object and Mouse object each come with the Propeller Tool software. The Keyboard object is a program that interfaces the Propeller chip to a standard PC-style keyboard. Similarly, the Mouse object interfaces to a standard computer mouse. Both of these objects are self-contained programs with carefully designed software interfaces that allow other objects, and developers, to use them easily.

By using existing objects, more sophisticated applications can be built very quickly. For instance, an application can include both the Keyboard and Mouse objects, and with just a few additional lines of code, a standard user interface is realized. Since the objects are self-contained and provide a concise software interface, application developers don't necessarily need to know exactly how an object achieves its task just to be able to use it. In a similar way, a driver of a car doesn't necessarily know how the engine works, but as long as that driver understands the interface (the ignition key, gas pedal, brakes, etc.) he or she can make the car accelerate and decelerate.

Well-written objects can be created by one developer and easily used by many different applications from many different developers.

Objects and Applications

A Propeller Object consists of Spin code and, optionally, Propeller Assembly code; see Figure 3-1. We'll simply call these "objects" from now on.

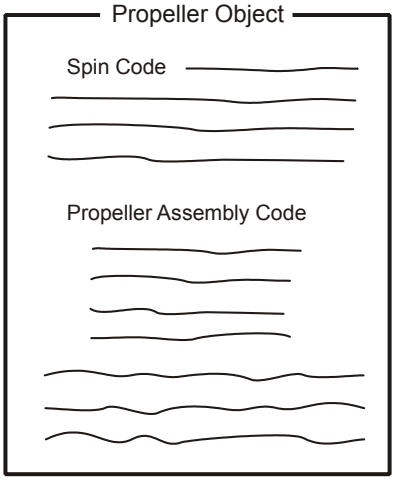


Figure 3-1: Propeller Object

Objects are stored on your computer as files with ".spin" extensions, therefore you should always think of each Spin file as an object.

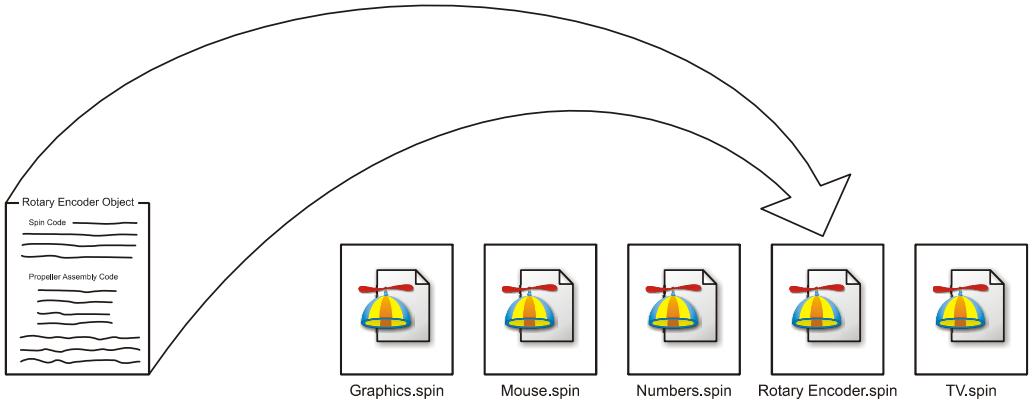


Figure 3-2: Object Files consist of Spin, and possibly Propeller Assembly, and are stored as ".spin" files on your computer's hard drive.

Each object can be thought of as a "building block" for an application. An object may choose to utilize one or more other objects in order to build a more sophisticated application. This is loosely called "referencing" or "including" another object. When an object references other objects it forms a hierarchy where it is the object at the top, as in Figure 3-3. The topmost object is referred to as the "Top Object File" and is the starting point for compiling a Propeller Application.

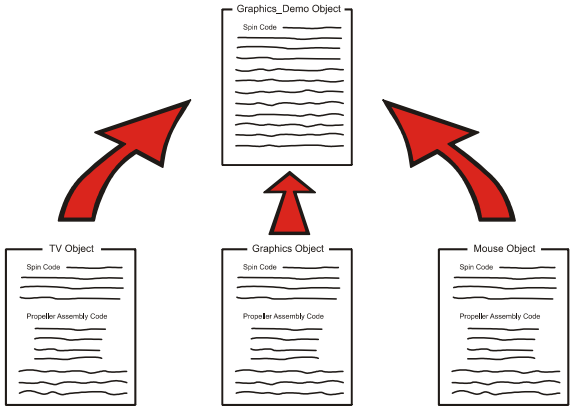


Figure 3-3: Object Hierarchy

When compiled, the Graphics Demo object is the "Top Object File" that references the other three objects shown below it.

In the above figure, the Graphics Demo object references three other objects: TV, Graphics, and Mouse. If the Graphics Demo object is compiled by the user, it is considered the Top

Object File and the other three objects are loaded and compiled with it resulting in a finished program called a Propeller Application, or “application” for short.

Applications are formed from one or more objects. The application is really a specially compiled binary stream that consists of executable code and data and can be run by the Propeller chip.

When downloaded, the application is stored in the Propeller chip’s Main RAM and optionally into an external EEPROM. At run time, the application is executed by one or more of the Propeller chip’s processors, called cogs, as directed by the application itself.

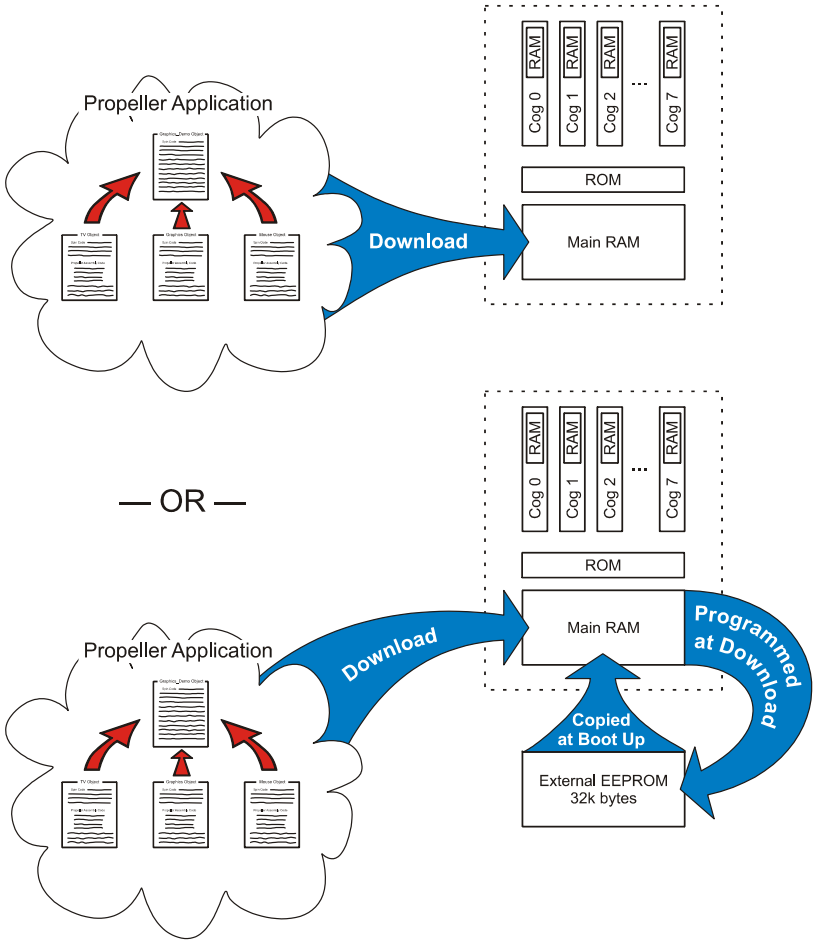


Figure 3-4: Downloading

Applications consisting of one or more objects are downloaded to the Propeller chip’s RAM, and optionally, its external EEPROM.

Connect for Downloading

In order to download a Propeller Application from the PC, you first need to connect the Propeller chip properly.

- If you have a Propeller Demo Board (Rev C or D), it includes the Propeller chip and all the necessary circuitry. Connect it to a power supply and the PC's USB cable and switch the power on. You may also need to install the USB drivers as directed by the Propeller Demo Board's documentation.
- If you do not have the Propeller Demo Board, we'll assume you have the Propeller chip and that you are experienced with wiring prototype circuits. Refer to Package Types on page 14 (showing the Propeller pinout) and Hardware Connections on page 17 for an example circuit showing the connections for power and programming. If you are using the Propeller Plug device, you may also need to install the USB drivers as directed by its documentation. The rest of this chapter relies heavily on circuitry similar to that of the Propeller Demo Board. In addition to the above power and programming connections, include the components and connections of the following schematic in your prototype circuit. You may also refer to the Propeller Demo Board's schematic; downloadable from the Parallax website.

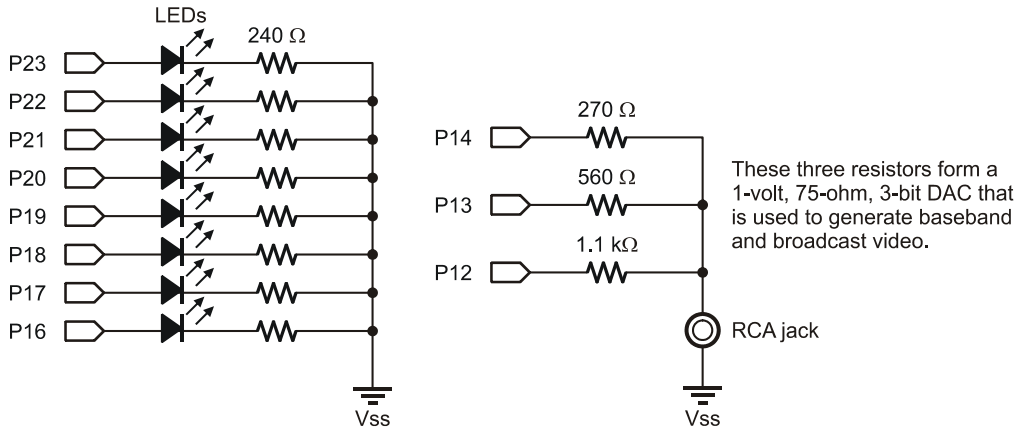


Figure 3-5: Propeller Tutorial Schematic

If you have made the connections suggested above, you should be able to verify and identify the Propeller chip via the Propeller Tool software. Start the Propeller Tool software (Version 1.0) and then press the F7 key (or select Run → Identify Hardware… from the menu). If the

Propeller chip is powered and connected to the PC properly, you should see an "Information" dialog similar to Figure 3-6.

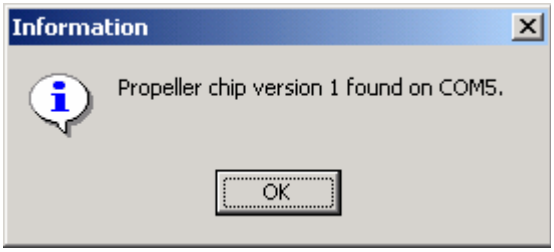


Figure 3-6: The Information Dialog

The port (COM5) may be different on your computer.

Quick Review: Intro

- The Propeller is programmed using two custom-designed languages: Spin and Propeller Assembly.
  - Spin is a high-level, object-based language interpreted at run time.
  - Propeller Assembly is a low-level, optimized assembly language which is executed directly at run time.
- Objects are programs that:
  - are self-contained.
  - perform a specific task.
  - may be reused by many applications.
- Well-written objects from one developer can easily be used by other developers and applications.
- A Propeller Object:
  - consists of two or more lines of Spin code and possibly Propeller Assembly code.
  - is stored on the computer as a file with a ".spin" extension.
  - may use one or more other objects to build a sophisticated application.
- Propeller Applications:
  - consist of one or more objects.
  - are compiled binary streams containing executable code and data.
  - are run by the Propeller chip in one or more cogs (processors) as directed by the application.
- The topmost object in a compiled application is called the "Top Object File."

Exercise 1: Output.spin – Our First Object

The following is a simple object, written in Spin, that will toggle an I/O pin high and low repeatedly. Start the Propeller Tool software and enter this program into the editor. We'll explain how it works in a moment. Make sure the "PUB" line begins in column 1 (the leftmost edge of the edit pane) and pay very close attention to each line's indention; it's important for proper operation.

```
PUB Toggle
  dira[16]~~
  repeat
    !outa[16]
    waitcnt(3_000_000 + cnt)
```

While indentation is critical, capitalization is not. Propeller code is not case-senitive. However, throughout this book, reserved words appear in bold all-captials, except in code snippets and excerpts, to help you become familiar with them.

After checking that you've typed it in properly, press the F10 key (or select Run → Compile Current → Load RAM + Run from the menu) to compile and download our example program. If the program you entered is syntactically correct and the Propeller chip is properly powered and connected to the PC, you should see a "Propeller Communication" dialog appear momentarily on the screen, like the one in Figure 3-7, and now the LED on I/O pin 16 of the Propeller chip should be blinking about twice per second. What we just accomplished is what is shown at the top of Figure 3-4: Downloading on page 89.

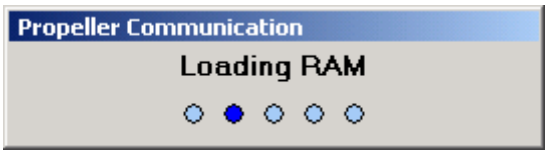


Figure 3-7: Propeller Communication Dialog

What really happened was probably too fast to see because the example program we entered is so small. When you pressed F10 it caused the Propeller Tool to compile the source code you entered and turn it into a Propeller Application. The Propeller Tool then searched for a Propeller chip connected to the PC and downloaded the application into its RAM. Finally, the Propeller started running the application from RAM, blinking the LED on I/O pin 16.

Downloading to RAM vs. EEPROM

Before we explain the code, let's take a closer look at the downloading process. Since our code was downloaded to RAM only, power cycling or resetting the Propeller will cause RAM contents to be lost and the program to stop permanently. Try pressing the reset button. The LED should turn off and never turn on again.

What if we don't want it to stop permanently? We could download to EEPROM instead of just RAM. Let's download again, but this time press the F11 key (or select Run → Compile Current → Load EEPROM + Run from the menu) to compile and download our example program to EEPROM. This is what is shown at the bottom of Figure 3-4: Downloading on page 89. As you may see from the figure, this actually downloaded to RAM first, then the Propeller chip programmed its external EEPROM, then started running the application from RAM, blinking the LED on I/O pin 16.

You probably noticed that the "Propeller Communication" dialog stayed on the screen much longer; EEPROMs take much longer to program than RAMs do.

Try pressing the reset button now. When you release the reset button, you'll notice a delay of about 1 ½ seconds and then the LED starts blinking. This is exactly what we wanted; a more permanent application in our Propeller chip.

Upon waking up from reset, the Propeller chip performed the boot-up procedure detailed on page 18. During that procedure, it determined it needed to boot up from the external EEPROM and then it took approximately 1½ seconds to completely copy the 32 Kbytes of content into its RAM and start running it.

Downloading only to RAM is convenient for development sessions because it is much faster. Downloading to both RAM and EEPROM to make the application more permanent is best done only when necessary because of the extra download time required.

A word of caution: If you download to EEPROM one or more times then revise your program and download to RAM only, when manually reset, the Propeller will boot up with your old program. This may make sense now, but that result can be very confusing when you're not paying attention. If things don't work right after a reset occurs, suspect the program in EEPROM first.

Exercise 1: Output.spin Explanation

Now for an explanation of the source code:

```
PUB Toggle
  dira[16]~~
  repeat
    !outa[16]
    waitcnt(3_000_000 + cnt)
```

The first line, `PUB Toggle`, declares that we're creating a "public" method called "`Toggle`." A method is the object-oriented term for "procedure" or "routine." We chose the name `Toggle` simply because it is descriptive of what the method does and we knew it is a unique symbol; it must be a unique symbol and conform to the Symbol Rules on page 159. We'll describe the term **PUB**, "public," in more detail later, but it's important to note that every object must contain at least one public (**PUB**) method.

The rest of the code is logically part of the `Toggle` method. We indented each line by two spaces from the **PUB**'s column to make that more clear; this indenting isn't required but is a good habit for clarity.

The first line of the `Toggle` method (second line of our example), `dira[16]~~`, sets the direction of I/O pin 16 to output. The **DIRA** symbol is the direction register for I/O pins P0 through P31; clearing or setting bits within it changes the corresponding I/O pin's direction to input or output. The `[16]` following `dira` indicates we want to access only the direction register's bit 16; the one that corresponds to I/O pin 16. Finally, the **~~** is the Post-Set operator that causes direction register bit 16 to be set to high (1); which makes I/O pin 16's direction an output. The Post-Set operator is a shorthand way of saying something like `dira[16] := 1` which may look familiar to you from other languages.

The next line, `repeat`, creates a loop consisting of the two lines of code below it. This **REPEAT** loop runs infinitely, toggling P16, waiting ¼ second, toggling P16, waiting ¼ second, etc.

The next line, `!outa[16]`, toggles the state of I/O pin 16 between high (VDD) and low (VSS). The **OUTA** symbol is the output state register for I/O pins P0 through P31. The `[16]` in `!outa[16]` indicates we want to access only the output register's bit 16; the one that corresponds to I/O pin 16. The **!** at the start of this statement is the Bitwise Not operator; it toggles the state of all bits specified to its right (the bit corresponding to I/O pin 16 in this case).

The last line, `waitcnt(3_000_000 + cnt)`, causes a delay of 3 million clock cycles. **WAITCNT** means "Wait for System Counter." The `cnt` symbol is the System Counter register; **CNT**

returns the current value of the System Counter, so this line means "wait for System Counter to equal 3 million plus its current value." In this code example, we didn't specify any clock settings for the Propeller chip, so by default it runs with its internal fast clock (about 12 MHz) meaning a delay of 3 million clock cycles is about ¼ second.

Remember how we said to pay close attention to each line's indenting? Here is where indenting *is* required: the Spin language uses levels of indention on lines following conditional or loop commands (`IF`, `CASE`, `REPEAT`, etc.) to determine which lines belong to that structure. In this case, since the two lines following `repeat` are indented to the right by at least one space beyond `repeat`'s column, those two lines are considered to be a part of the `repeat` loop. If you have trouble recognizing these structural groupings, the Propeller Tool can make them more visible on-screen through the Block-Group Indicators feature. Use Ctrl+I to toggle this feature on or off. Figure 3-8 shows our example code with these indicators visible.

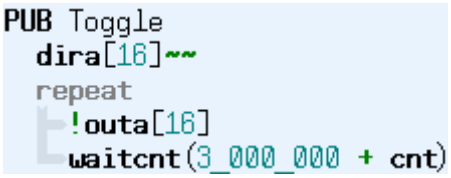


Figure 3-8: Block-Group Indicators

Ctrl+I toggles them on and off.

If you haven't saved this example object yet, you may do so by pressing Ctrl+S (or selecting File → Save from the menu). You may save it in a folder of your choosing but make sure to save it with the filename "Output.spin" since some exercises below rely on it.

Quick Review: Ex 1

- Applications are downloaded to either Propeller RAM only or RAM and EEPROM.
  - Those in RAM will not survive power cycling or resetting the Propeller chip.
  - Those in EEPROM are loaded into RAM on boot-up in approximately 1½ seconds.
  - To download the current object to:
    - RAM only: press F10 or select Run → Compile Current → Load RAM + Run.
    - RAM + EEPROM: press F11 or select Run → Compile Current → Load EEPROM + Run.
- Spin language:
  - Method means "procedure" or "routine."
  - `PUB` *Symbol* declares a public method called *Symbol*. Every object must contain at least one public (`PUB`) method. See `PUB` on page 287 and Symbol Rules on page 159.
  - `DIRA` is the direction register for I/O pins 0-31. Each bit sets the corresponding I/O pin's direction to input (0) or output (1). See `DIRA, DIRB` on page 212.
  - `OUTA` is the output state register for I/O pins 0-31. Each bit sets the corresponding I/O pin's output state to low (0) or high (1). See `OUTA, OUTB` on page 280.
  - Registers can use indexes, like [16], to access individual bits within them. See `DIRA, DIRB` on page 212 or `OUTA, OUTB` on page 280.
  - `~~` following a register/variable sets its bit(s) high. See Sign-Extend 15 or Post-Set '`~~`' on page 263 in the Operators section.
  - `!` preceding a value/register/variable sets its bit(s) opposite their current state. See Bitwise NOT '`!`'on page 272 in the Operators section.
  - `REPEAT` creates a loop structure. See `REPEAT` on page 293.
  - `WAITCNT` creates a delay. See `WAITCNT` on page 322.
  - Indention at the start of lines:
    - indicates they belong to the preceding structure; it is required for lines following conditional or loop commands (like `REPEAT`). (Indenting is *optional* after block indicators, such as `PUB`.)
    - Ctrl+I toggles visible block-group "structure" indicators on and off.

Cogs (Processors)

The Propeller has eight identical processors, called cogs. Each cog can be individually set to run or stop at any time as directed by the application it is running. Each cog can be programmed to run independent tasks or cooperative tasks with other cogs, as needed, and this can change as desired during the application's run time.

But we didn't specify which cog(s) to run in our Output.spin example, so how did it work? For a review, you could read Boot Up Procedure, page 18, and Run-Time Procedure, page 18, in Chapter 1, but we'll discuss it a bit more here.

For our example, upon power-up, the Propeller chip starts the first processor (Cog 0) and loads it with a built-in Boot Loader program. The Boot Loader program is copied from the Propeller chip's ROM into Cog 0's internal RAM memory. Cog 0 then runs the Boot Loader program in its internal memory and the Boot Loader soon determines it should copy user-code from the external EEPROM. So Cog 0 copies the entire 32 K byte EEPROM contents into the Propeller chip's 32 K byte main RAM memory (separate from the cog's internal memory). Then the Boot Loader program makes Cog 0 reload itself with the built-in Spin Interpreter; the Boot Loader program in Cog 0 halts at this point while it is being overwritten with the Spin Interpreter program.

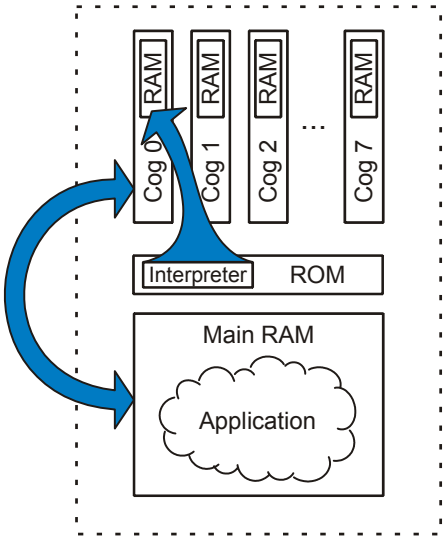


Figure 3-9: Running Output.spin

Notice that the Spin Interpreter, not the Spin Application, is loaded into Cog RAM. The Spin Application resides in Main RAM and is interpreted by the Spin Interpreter program that is running in the cog.

Now, Cog 0 is running the Spin Interpreter, which fetches and executes our application's code from main memory (RAM). This is shown in Figure 3-9. Since our application consists

entirely of interpreted Spin code, it continues to reside only in main memory while a cog running the Spin Interpreter (Cog 0 in this case) reads, interprets and effectively executes it. No other cogs were started during boot up or during our application's execution; the other seven cogs remain in a dormant state consuming virtually no current at all. Later, we'll change our application to start other cogs as well.

Exercise 2: Output.spin - Constants

Let's enhance our program a little. Suppose we want to make it easy to change the I/O pin and the length of the delay used. As it is written currently, we'd have to find and change the pin number in two places and the delay in yet another place. We can make it better by defining those items in a separate place that is easy to find and edit. Look at the following example and edit your code to match (we highlighted every new or modified element).

```
CON
  Pin   = 16
  Delay = 3_000_000

PUB Toggle
  dira[Pin]~~
  repeat
    !outa[Pin]
    waitcnt(Delay + cnt)
```

The new `CON` block at the top of the code defines global constants for the object (see **CON**, page 194.) In it, we created two symbols, `Pin` and `Delay`, and assigned the constant values 16 and 3,000,000 to them, respectively. We can now use the symbols `Pin` and `Delay` elsewhere in the code to represent our constant values 16 and 3,000,000. Notice that we used underscores (_) to separate the "thousands" groups in the number 3,000,000? Commas are not allowed there but underscores are allowed anywhere inside of constant values; this makes large numbers more readable.

In the `Toggle` method, we replaced both occurrences of 16 with the symbol `Pin`, and replaced the `3_000_000` with the symbol `Delay`. When compiled, the Propeller Tool will use the constant values in place of their respective symbols. This makes it easy, later on, to change the pin number or delay at will since we only have to change it up at the top of code in a place that is easy to find and understand.

Try changing the `Delay` constant from 3,000,000 to 500,000 and download again; the LED should now flicker at a rate of 12 blinks per second (24 toggles per second). You can also

change the Pin constant from 16 to 17 and download again to see a different LED blink. NOTE: you can try 18 through 23 as well, but on the Propeller Demo Board they are connected in pairs with resistors for the VGA driver circuit, so two LEDs will blink at once.

Block Designators

You may have noticed that the backgrounds of the CON and PUB blocks of code were colored differently when you entered them into the editor. This is the Propeller Tool's way to indicate these are distinct blocks of code.

Spin code is organized in blocks that have distinct purposes. CON and PUB are block designators that indicate the start of a "constant block" and "public method block", respectively. Every block designator must start in the first column of text (the leftmost edge of the edit pane) on a line. There are six types of blocks in the Spin language: CON, VAR, OBJ, PUB, PRI, and DAT. The following is a list of the block designators and their purpose:

- **CON** Global Constant Block. Defines symbolic constants that can be used anywhere within the object (and in some cases outside the object) wherever numeric values are allowed.
- **VAR** Global Variable Block. Defines symbolic variables that can be used anywhere within the object wherever variables are allowed.
- **OBJ** Object Reference Block. Defines symbolic references to other existing objects. These are used to access those objects and the methods and constants within them.
- **PUB** Public Method Block. Public methods are code routines that are accessible both inside and outside the object. Public routines provide the interface to the object; the way methods outside of the object interact with the object. There must be at least one PUB declaration in every object.
- **PRI** Private Method Block. Private methods are code routines that are accessible only inside the object. Since they are not visible from the outside, they provide a level of encapsulation to protect critical elements within the object and help maintain the integrity of the object's purpose.
- **DAT** Data Block. Defines data tables, memory buffers, and Propeller Assembly code. The data block's data can be assigned symbolic names and can be accessed via Spin code and assembly code.

There can be multiple occurrences of each block type, arranged in any order desired, but there must be at least one PUB block per object. Even though the number of blocks and their order

is quite flexible, typically there is only one occurrence of CON, VAR, OBJ and DAT blocks, multiple occurrences of PUB and PRI blocks, and the suggested order for typical programs is the order they are given in the list above.

The very first PUB block in the very first object (the Top Object File where compilation starts from) automatically becomes the Propeller Application's starting point; it is executed first when the application starts. No other public or private method is executed automatically, but rather they are executed only as determined by the natural flow of the application.

The Propeller Tool automatically colors the backgrounds of each block differently, even two consecutive occurrences of the same block type, in order to make it easy to identify the type, start, and end of each block. This in no way affects the actual source code itself, it is simply an indicator for on-screen use that is intended to solve a typical problem with source code; that is, as the code gets larger, it is harder to find a particular method quickly as you scroll up and down through the code unless you have some kind of separator between methods. The background color coding serves as an automatic separator that prevents you from having to waste time typing in text-based separators manually.

Exercise 3: Output.spin – Comments

Our Output object is better now, but it still could be more readable. How about adding some comments to the code to make it easier for other readers to understand? The next example functions the same as before, but with a number of comments (human-readable, non-executable text) above and to the right of our existing code.

These comments should help people figure out what it does. There are four types of comments supported by the Propeller Tool (all of which are shown in this example):

- `'...` – Single-line code comment (apostrophe).
- `''...` – Single-line document comment (two apostrophes, NOT a quotation mark).
- `{...}` – Multi-line code comment (curly braces).
- `{{...}}` – Multi-line document comment (two curly braces).

```
{{Output.spin

Toggles Pin with Delay clock cycles of high/low time.}}

CON
  Pin   = 16                  { I/O pin to toggle on/off }
  Delay = 3_000_000           { On/Off Delay, in clock cycles}

PUB Toggle
''Toggle Pin forever
{Toggles I/O pin given by Pin and waits Delay system clock cycles
in between each toggle.}

  dira[Pin]~~                 'Set I/O pin to output direction
  repeat                      'Repeat following endlessly
    !outa[Pin]                '  Toggle I/O Pin
    waitcnt(Delay + cnt)      '  Wait for Delay cycles
```

Single line comments begin with at least one apostrophe (') and continue until the end of the line. Executable code can be to the left of a single-line comment but not to the right of it since that would make it "commented out." The "'Set I/O pin..." and "'Repeat following..." comments are examples of single-line comments.

Multi-line comments begin with at least one open curly brace ({) and end with at least one close curly brace (}). Unlike single-line comments, executable code can be to the left and the right of multi-line comments. Multi-line comments can actually be entirely on one line, or can span across multiple lines. The "{On/Off Delay...}" and "{Toggles I/O pin given...}" comments are both examples of a multi-line comments.

If a comment begins with just one apostrophe (') or one open curly brace ({), it is a "code" comment. This is a comment meant to be read by code developers while reviewing the source code itself.

If a comment begins with either two apostrophes ('') or two open curly braces ({{), with no spaces in between, it is a "document" comment. This is a special type of comment that is visible within the code, but can also be extracted by the Propeller Tool into a document formatted for easier reading, containing no executable code.

As discussed in Chapter 2View Modes on page 61, the Propeller Tool's editor has a Documentation view mode. With the above code entered into the editor, if the Documentation view mode is selected, the code is compiled and the document comments are

shown along with some statistics about the compiled code. The following is what this looks like:

```
Output.spin

Toggles Pin with Delay clock cycles of high/low time.
Object "Output" Interface:

PUB  Toggle

Program:       8 Longs
Variable:      0 Longs

____________
PUB  Toggle

Toggle Pin forever
```

If you compare this to our code you should recognize all the text that came directly from our document comments. The section "`Object "Output" Interface:`" is created automatically by the Propeller Tool; it lists all the public methods (just `PUB Toggle` in this case) and shows that the program size is 8 longs (32 bytes) and no variables were used. Following that, it lists all the public methods again, with an overbar above each method, and the document comments that belong with them. This section shows the public `Toggle` method and our last document comment, "`Toggle Pin forever`," indicating what the `Toggle` method does.

Adding document comments to your code allows you to create just one file that contains both source code and documentation for an object. This is extremely convenient for developers since they can easily switch to Documentation view to learn how to use an object they are unfamiliar with. To support this further, the Propeller Tool's Parallax font has many special characters for including schematics, timing diagrams, and mathematical symbols right in the objects that they relate to, like those shown in Figure 2-1 on page 36.

Quick Review: Ex 2 & 3

- The Propeller has eight identical processors, called cogs.
  - Any number of cogs can be running or halted at any time as directed by the application.
  - Each cog can run independent or cooperative tasks.
  - At boot-up, Cog 0 runs the Spin Interpreter to execute the main memory-based Spin application.
- Spin language:
  - Organized in blocks that have distinct purposes.
    - `CON` – Defines global constants, see page 194.
    - `VAR` – Defines global variables, see page 315.
    - `OBJ` – Defines object references, see page 247.
    - `PUB` – Defines a public method, see page 287.
    - `PRI` – Defines a private method, see page 286.
    - `DAT` – Defines data, buffers, and assembly code, see page 208.
  - Block designators must be in column 1 of a line.
  - Each block type can occur multiple times and can be arranged in any order.
  - The very first `PUB` block in the very first object is the Propeller Application's starting point.
  - Underscores "_" in constants denote logical groupings, like thousands in decimal numbers.
  - Types of comments:
    - Code comments; visible in source code only. Great for notes to developers regarding function of specific code.
      - `'...` – Single-line; starts at apostrophe and continues to end of line.
      - `{...}` – Multi-line; starts and ends with single curly braces.
    - Document comments; visible in source code and documentation view. Great for object documentation. Can even include schematics, timing diagrams and other special symbols.
      - `''...` – Single-line; starts at double-apostrophe and continues to end of line.
      - `{{...}}` – Multi-line; starts and ends with double-curly braces.

Exercise 4: Output.spin – Parameters, Calls, and Finite Loops

Our current object from Exercise 3 is interesting, but still isn't very flexible; after all, the Toggle method only works with a specific pin and delay. Let's make the Toggle method more flexible and also give it the ability to toggle a specific, finite number of times. Look at the following example and edit your code to match. We've crossed out the elements that should be removed, and highlighted every new element.

```
{{Output.spin

Toggles Pin with Delay clock cycles of high/low time.}}
Toggles two pins, one after another.}}

CON
  Pin   = 16                    { I/O pin to toggle on/off }
  Delay = 3_000_000             { On/Off Delay, in clock cycles}

PUB Main
  Toggle(16, 3_000_000, 10)  'Toggle P16 ten times, 1/4 s each
  Toggle(17, 2_000_000, 20)  'Toggle P17 twenty times, 1/6 s each


PUB Toggle(Pin, Delay, Count)
''Toggle Pin forever
{Toggles I/O pin given by Pin and waits Delay system clock cycles
in between each toggle.}
{{Toggle Pin, Count times with Delay clock cycles in between.}}

  dira[Pin]~~                   'Set I/O pin to output direction
  repeat Count                  'Repeat for Count iterations
    !outa[Pin]                  '  Toggle I/O Pin
    waitcnt(Delay + cnt)        '  Wait for Delay cycles
```

Compile and download this application to see the results. The LED on pin 16 should blink five times (10 toggles) with 1/4th second durations and intervals, then it will stop and the LED on pin 17 will blink ten times (20 toggles) at 1/6th second durations and intervals.

In this example we removed the constant (CON) block, added a new method called Main, and made some minor modifications to the Toggle method. The Toggle method still performs the actual pin-toggling action, but the Main method tells it when and how to do so.

The Toggle Method

Let's look closely at the Toggle method first. In its declaration, we added (Pin, Delay, Count) immediately to the right of its name. This creates a "parameter list" for our Toggle method consisting of three parameters, Pin, Delay and Count. A parameter list is one or more symbols that must be filled with values when the method is called; more on that in a moment. Each parameter symbol is a long-sized (4-byte) variable that is local to the method; they are all accessible within the method but not outside of the method. Parameter variables can be modified within the method but those modifications do not affect anything outside the method.

Now, our Toggle method can be called by other methods and given unique values to use as its Pin, Delay and Count symbols; it is more flexible since we can adjust its operational parameters.

Inside of Toggle, nothing changed except the REPEAT command, which is now repeat Count. Remember, in our previous examples the REPEAT loop was an infinite loop; it never stopped. Well, if you immediately follow REPEAT with an expression, it becomes a finite loop that iterates the number of times indicated by the expression. In this case our REPEAT loop will execute Count times, then it will stop, and any lines of code below the end of the loop will begin to execute.

The Main Method

Now look at the Main method. Main's first line, Toggle(16, 3_000_000, 10), is a method call; it causes the Toggle method to execute using 16 for its Pin parameter, 3 million for its Delay parameter, and 10 for its Count parameter. The following line looks similar, Toggle(17, 2_000_000, 20), but it calls the Toggle method with different values: 17 for Pin, 2 million for Delay, and 20 for Count.

Notice that we put the Main method above Toggle? Remember that the first public method in the first object is automatically executed when the application is started by the Propeller. We are only using one object in this case, so Main is automatically executed after we download this application.

When Main's first line, Toggle(16, 3_000_000, 10), is executed, the Toggle method is called and it executes its function: blinking the LED on pin 16 five times with a delay of 1/4th second in between. Then, because Toggle has no more code to execute after the loop, it returns to the caller, Main, and execution continues at the next line of Main: Toggle(17, 2_000_000, 20). When that line executes, the Toggle method is called, and it blinks the LED on pin 17 ten times with a delay of 1/6th second in between. Finally, the Toggle method returns to Main again, but Main has no more code to execute so it exits and the application

terminates; the cog stops and the Propeller goes into a low-power mode until the next reset or power cycle.

Don't be confused by the look of the code. The two methods, `Main` and `Toggle`, are shown one right after another, but they are treated as distinct routines starting at their **PUB** block declarations and ending at the next block declaration or the end of the source code, whichever comes first. In other words, the Propeller knows that the `Toggle` method is not a part of the `Main` method's executable code.

Also note we've still used just one cog in our example, and the entire application is executed serially: first blink P16, then stop, blink P17, then stop. We'll begin to use multiple cogs in the next exercise.

Exercise 5: Output.spin – Parallel Processing

In exercises 1 through 4 we've used just one cog to process the application; it toggles P16 only, then stops and toggles P17 only, then terminates. This is called "serial processing."

Suppose, however, that we want to do things in parallel; simultaneously toggling pins 16 and 17, each at different rates and for different finite periods. Tasks like this can certainly be done with serial processing and clever programming but it is easier with parallel processing by having the Propeller activate two cogs. Look at the following example and edit your code to match. We added a variable block (**VAR**) and made a slight change to the `Main` method.

```
{{Output.spin
Toggle two pins, one after another simultaneously.}}

VAR
  long  Stack[9]              'Stack space for new cog

PUB Main
  cognew(Toggle(16, 3_000_000, 10), @Stack)      'Toggle P16 ten…
  Toggle(17, 2_000_000, 20)                      'Toggle P17 twenty…

PUB Toggle(Pin, Delay, Count)
{{Toggle Pin, Count times with Delay clock cycles in between.}}

  dira[Pin]~~                 'Set I/O pin to output direction
  repeat Count                'Repeat for Count iterations
    !outa[Pin]                '  Toggle I/O Pin
    waitcnt(Delay + cnt)      '  Wait for Delay cycles
```

The VAR Block

In the VAR block we defined an array of longs, Stack, which is 9 elements in length. This is used by the Main method.

The Main Method

We modified the Main method's first line such that its original code, the call to Toggle, is encased in a COGNEW command. The COGNEW command starts a new cog to run either Spin or Propeller Assembly code. In this case, we entered Toggle(16, 3_000_000, 10), for COGNEW's first parameter, and @Stack for its second parameter. This means COGNEW will start a new cog to run the Toggle method and will use the memory starting at the address of Stack for run-time stack space. The @ is the Symbol Address operator; it returns the actual address of the variable following it.

To run Spin code, the new cog needs some run-time workspace, called "stack space," where it can store temporary things like return addresses, return values, intermediate expression values, etc. We chose to reserve 9 longs of space (36 bytes), and passed the address of that space as COGNEW's second parameter, @Stack. How much stack space is needed? It varies depending on the Spin code being executed, but we'll discuss those details later. For now, rest assured that 9 longs of space is enough for our Toggle method.

Compile and download Output.spin. You should see that the LEDs on P16 and P17 now simultaneously blink at different rates, 5 times and 10 times, respectively. This is because we now have two cogs running simultaneously; one toggles P16 while the other toggles P17.

Here's how it works: Cog 0 starts executing our application's Main method. The first line of Main uses the COGNEW command to activate a new cog (Cog 1) to run the Toggle method with the parameters (16, 3_000_000, 10) passed to it. While Cog 1 is starting up, Cog 0 continues on with the second line of the Main method, the direct call to Toggle with the parameters (17, 2_000_000, 20) passed to it. Ultimately, Cog 0 is left executing Toggle on P17 while Cog 1 executes Toggle on P16, simultaneously. When their individual tasks have expired, they each terminate due to lack of code. Cog 1 terminates the moment it finishes Toggle. Cog 0 finishes Toggle, returns to Main and then terminates. Cog 1 happens to terminate earlier than Cog 0 in this case.

Figure 3-10 illustrates this. The Propeller loads the Spin Interpreter into Cog 0 to execute the application (the two leftmost arrows in the figure). Then the application requests a new cog to activate, via the `COGNEW` command, which causes the Propeller to load the Spin Interpreter into the next available cog, Cog 1, to execute a smaller portion of Spin code from the application, the `Toggle` method (the two rightmost arrows in the figure). Each cog executes its code completely independently of the other; true parallel processing. Note that towards the end of the application both cogs are executing the same piece of Spin code, the `Toggle` method, but each is using its own workspace and its own values for `Pin`, `Delay` and `Count`.

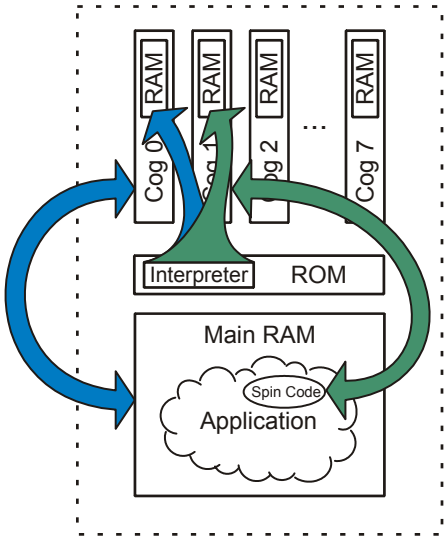


Figure 3-10: Two Cogs Running Output Application and Toggle method.

Notice that the Spin Interpreter is loaded into each Cog's RAM. The Spin Application, Output, resides in Main RAM interpreted by Cog 0 and it launches Cog 1 to run just the Toggle method.

Quick Review: Ex 4 & 5

- Spin language:
  - Methods:
    - To call methods in the same object, use *method* where method is the method's name, see PUB on page 287.
    - Methods automatically exit, returning to their caller, when they run out of code to execute.
    - When an application's first method exits, the application and the cog it is running in terminate.
  - Parameter Lists
    - Methods declare parameters in the form: *method*(*param1*, *param2*, etc.) , see PUB on page 287.
    - Parameters are long-sized, local variables that are accessible from within the method only.
      - They can be modified within the method but any corresponding variables used in the call are left unaffected.
  - REPEAT command:
    - Infinite loop: `repeat`
    - Finite loop: `repeat` *expression* where *expression* evaluates to the desired number of loops to iterate through, see REPEAT on page 293.
  - Arrays:
    - Arrays are defined with the form *symbol* [*count*] where *symbol* is the array's symbolic name and *count* is the number of elements in the array, see VAR on page 315.
  - COGNEW command:
    - Activates another cog (processor) to run either Spin or Propeller Assembly code, see COGNEW on page 189.
    - Allows for true parallel processing.
    - Requires an address to reserve run-time stack space for Spin code.
  - The Symbol Address operator (@) returns the address of the variable following it. See Symbol Address '@' on page 278.

Exercise 6: Output.spin & Blinker1.spin – Using Our Object

Now let's explore the power of objects. All of the preceding exercises created an application that contained only one object; the Output.spin object is the entire application. This is typical of how new objects begin their development. Suppose that the motivation behind all this work was really to create an object other developers could use to easily toggle one or more I/O pins. Yes, it may be silly to create an object for such a use, but let's have fun with it anyway!

It's time to make our Output object easily interface with other objects. Edit your code to look like the following:

Example Object: Output.spin

```
{{ Output.spin }}
Toggle two pins, one after another.}}

VAR
  long  Stack[9]              'Stack space for new cog

PUB Main
  cognew(Toggle(16, 3_000_000, 10), @Stack)      'Toggle P16 ten…
  Toggle(17, 2_000_000, 20)                      'Toggle P17 twenty…

PUB Start(Pin, Delay, Count)
{{Start new toggling process in a new cog.}}

  cognew(Toggle(Pin, Delay, Count), @Stack)

PUB Toggle(Pin, Delay, Count)
{{Toggle Pin, Count times with Delay clock cycles in between.}}

  dira[Pin]~~                 'Set I/O pin to output direction
  repeat Count                'Repeat for Count iterations
    !outa[Pin]                '  Toggle I/O Pin
    waitcnt(Delay + cnt)      '  Wait for Delay cycles
```

Be sure to save this object with the filename "Output.spin" for later use by our next object.

The Start Method

Here we replaced the Main method with a Start method. The Start method activates another cog to run the Toggle method independently and passes along the Pin, Delay, and Count parameters.

The interface to an object is made up of its public (PUB) methods, so our Output object now has two interface components, the Start method and the Toggle method.

Now our Output object can be used by other objects to toggle any pin at any rate for any number of times they want. They can also choose to do this serially, by calling Output's Toggle method, or in parallel with other tasks, by calling Output's Start method.

Let's create another object that uses Output. To create a new object, select File → New from the menu and a new edit tab will appear. In this new edit page, enter the following code. Pay attention to the bold items, as we will discuss them soon.

Example Object: Blinker1.spin

```
{{ Blinker1.spin }}

OBJ
  LED : "Output"

PUB Main
{Toggle pins at different rates, simultaneously}
  LED.Start(16, 3_000_000, 10)
  LED.Toggle(17, 2_000_000, 20)
```

Save this new object as "Blinker1.spin" in the same folder as you saved Output.spin. Now, with Blinker1's edit tab active, press F10 to compile and download. The LEDs should have blinked in the same way they did in Exercise 5, but a different technique was used by the code; Blinker1 used our Output object and simply called Output's Start and Toggle methods.

Here's how it worked. In Blinker1 we have an object block (OBJ) and a public method (PUB). The object block's LED : "Output" line declares that we're going to use another object called Output and that we'll refer to it as LED within this Blinker1 object.

The Object-Method Reference

In the public method, Main, we have two method calls. Remember how we learned in Exercise 4 that one method can call another just by referencing its name? That works for methods that are in the same object, but now we need to call a method that is in another object. To do this, we use the form *object*.*method* where *object* is the symbolic name we

gave the object in the `OBJ` block (`LED` in this case) and *method* is the name of that object's method. This is called an Object-Method reference. Blinker1 refers to the Output object as `LED`, so `LED.Start` calls Output's `Start` method, and `LED.Toggle` calls Output's `Toggle` method.

When Blinker1 is compiled, since it references Output, the two objects get compiled into one application. Figure 3-11 illustrates this. This structure is also shown in the Object View, which we'll learn about next.

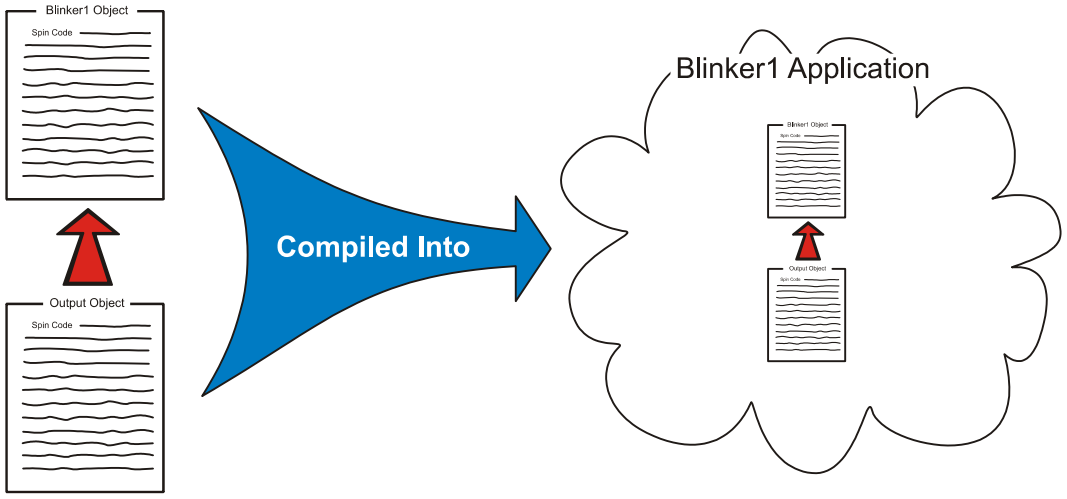


Figure 3-11: Blinker1 Hierarchy and Blinker1 Application

The Object View

When you compiled Blinker1, the Object View pane updated to indicate the application's structure. The Object View is in the upper left corner of the Propeller Tool if you have the Integrated Explorer pane open (see Pane 1: Object View Pane on page 39.) Figure 3-12 shows what it should look like now.

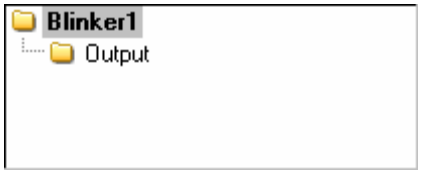


Figure 3-12: Blinker1 Object View

The Object View updates itself each time an application is successfully compiled to show you the logical structure of that application. The view shown in Figure 3-12 is the Object View's way of illustrating the logical structure in Figure 3-11. It is necessary to check the Object View once in a while to troubleshoot or to verify proper compilation.

Your entire application is displayed in the Object View, or at least what it looked like after the last successful compile. You can also use it to explore the application. For example, pointing the mouse at each object in the Object View gives you hint information about that object. Left-clicking each of those objects either opens them up or switches the active edit tab to that object. Right-clicking each of those objects does the same as left-clicking but it makes the object switch to Documentation view instead of Full Source view.

Top Object File

The object at the top of the Object View is always the "Top Object File" for that particular compilation. That means the compilation started from Blinker1, in this case. When we compile by using the F10 or F11 shortcut keys, or their corresponding menus, the Propeller Tool starts the compile operation using whatever edit tab is active at that moment. The active edit tab is the one that is highlighted differently than the rest; see Pane 4: Editor Pane on page 40 and Figure 2-4 on page 40 for an example.

If we had accidentally clicked on the Output object's tab first and then compiled with F10 or F11 the compile would have started from that object instead. This would not have resulted in the application we desired and the Object View would have shown only one object, Output, in its structure. This is all because the compile functions we've been using are the "Compile Current" options; meaning they compile from the currently active object, or edit tab.

There are other compile functions that can help us. Select the Run menu and look at the options. You should see a "Compile Current" and "Compile Top" flyout menu (Figure 3-13).

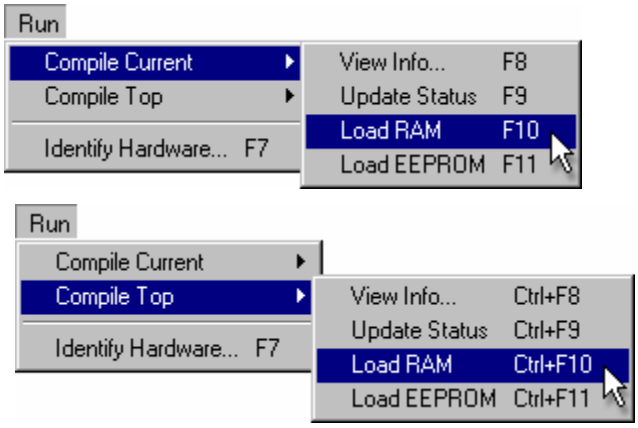


Figure 3-13: Compile Current Menu (top) and Compile Top Menu (bottom)

Each menu, Compile Current and Compile Top, has the same sub-options but they start their compilation from different places. Compile Current starts from the active edit tab and Compile Top starts from the designated Top Object File.

You can tell the Propeller Tool which object to treat as the "designated Top Object File" at any time. You do this by any one of the following methods:

1) Right-click the desired object's edit tab and select "Top Object File," or
2) Right-click the desired object from the File List (in the Integrated Explorer) and select "Top Object File," or
3) Choose the File → Select Top Object File… menu option and select the desired file from the browse window, or
4) Press Ctrl+T and select the desired file from the browse window.

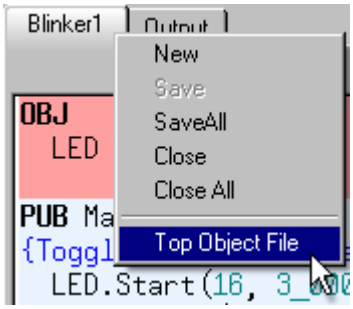
We used option #1 to select Blinker1 as the Top Object File, in the figure below. Note that afterwards, the Blinker1 tab's text is bold; see Figure 3-14b. The file the Propeller Tool knows as the Top Object File always appears in bold.

Now, if we use one of the Compile Top options, such as Ctrl+F10 or Ctrl+F11, regardless of which edit tab is active, the Propeller Tool will compile starting from the Top Object File. For example, in Figure 3-14b the Output object is the active edit tab. If we press Ctrl+F10, the application will be compiled starting with the Blinker1 object, however. If we had pressed F10 instead, the Output object would have been compiled.

Each of the shortcut keys for the Compile Current options, F8, F9, F10, etc., has a similar variation for the Compile Top options, Ctrl+F8, Ctrl+F9, Ctrl+10, etc.

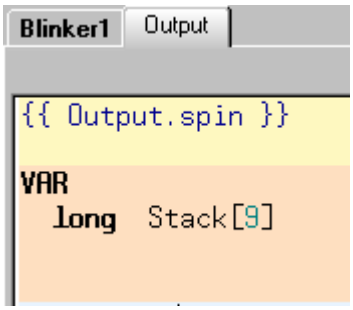
Figure 3-14: Setting Blinker1 to be the Top Object File

a.



One way to set the Top Object File is by right-clicking the desired edit tab, and choosing Top Object File.

b.



The Top Object File's name will show in bold on its Edit Tab.

Which Objects Were Compiled?

If there's ever a question of which object files were compiled in the last successful compile operation, use the mouse to explore the resulting application's structure in the Object View.

It's important to keep track of which file you've designated as the Top Object File and what compile option you chose; Current vs. Top. Only one file can be designated as the Top Object File at a time and the Propeller Tool remembers that file even between sessions.

Also, keep in mind that an object doesn't really need to be open in the Propeller Tool just to be compiled. If an object you compiled references another object, that object will be compiled whether or not it is currently open. Even the Top Object File can be compiled without it being open. For example, pressing Ctrl+F10 will compile the last designated Top Object File regardless of whether or not it even belongs to the current application you are working on.

Quick Review: Ex 6

- Spin language:
  - Methods:
    - To call methods in anoTher object, use *object*`.`*method* where *object* is the object's symbolic name (given to it in the `OBJ` block) and *method* is the method's name within that other object. See `OBJ` on page 247.
    - Public (`PUB`) methods are an object's interface; other objects call its public methods. See `PUB` on page 287.
- Object View
  - Illustrates the structure of the most recent successfully compiled application. See Object View, page 52.
  - Pointing the mouse at displayed objects displays hints about them.
  - Left-clicking a displayed object either opens it up or makes it the active edit tab.
  - Right-clicking a displayed object opens or switches to it in Documentation view.
- Compile Current – (F8 through F11) - compiles starting from the current object (active edit tab).
- Compile Top – (Ctrl+F8 through Ctrl+F11) - compiles starting from the Top Object File.
- Top Object File:
  - Appears with a bold name in the edit tab and File List.
  - Can be designated by one of the following (and compiled via Compile Top operation):
  1) Right-click object's edit tab and select "Top Object File," or
  2) Right-click object in the File List and select "Top Object File," or
  3) Choose File → Select Top Object File… menu and select object from browser, or
  4) Press Ctrl+T and select object from browser.
- Objects don't have to be open to be compiled; they may be compiled as the result of another object's compilation or as the result of a Compile Top operation.

Objects vs. Cogs

It's important to understand that there is no direct relationship between objects and cogs. Remember, Exercise 5 used just one object but two cogs and Exercise 6 used two objects and two cogs, but each of these exercises could have used only one cog if they wanted to process everything serially. When and how cogs are used is completely determined by the application and the developer(s) who wrote it.

Exercise 7: Output.spin – More Enhancements

Let's add some significant enhancements to our Output object. Currently the `Toggle` method can be called to toggle a pin serially, or the `Start` method can be called to launch the `Toggle` method as a separate process, to run in parallel. But we haven't provided a way to stop that process once it is going or even a way to determine if it's running in the first place. Also, it would be nice to have the option of toggling the pin endlessly, in addition to the finite count feature we already have.

Let's add a `Stop` method to stop the active process and an `Active` method to test whether a parallel process is currently running. In addition, we'll enhance our `Toggle` method as described above.

For objects like this one, it is a common and recommended convention to use the name "Start" for a method that activates a new cog and the name "Stop" for a method that deactivates a cog previously started by that object. This way, while scanning an object in summary or documentation view, other developers can more quickly understand how to use your object; when they see `Start` and `Stop` they can infer that the object activates/deactivates another cog. For objects that don't activate another cog but still need some kind of initialization, it is recommended to use the name "Init" for the key method.

This code is loaded with clever changes; be prepared, it will take a lot to explain it but the knowledge you'll gain is well worth it.

Here's the code; modify yours to match:

```
{{ Output.spin }}

VAR
  long  Stack[9]                        'Stack space for new cog
  byte  Cog                             'Hold ID of cog in use, if any


PUB Start(Pin, Delay, Count): Success
{{Start new blinking process in new cog; return TRUE if successful}}

  Stop
  Success := (Cog := cognew(Toggle(Pin, Delay, Count), @Stack) + 1)


PUB Stop
{{Stop toggling process, if any.}}

  if Cog
    cogstop(Cog~ - 1)


PUB Active: YesNo
{{Return TRUE if process is active, FALSE otherwise.}}

  YesNo := Cog > 0


PUB Toggle(Pin, Delay, Count)
{{Toggle Pin, Count times with Delay clock cycles in between.}}
  If Count = 0, toggle Pin forever.}}

  dira[Pin]~~                           'Set I/O pin to output…
  repeat Count                          'Repeat for Count iterations
  repeat                                'Repeat the following
    !outa[Pin]                          '  Toggle I/O Pin
    waitcnt(Delay + cnt)                '  Wait for Delay cycles
  while Count := --Count #> -1          'While not 0 (make min -1)
  Cog~                                  'Clear Cog ID variable
```

The VAR Block

In the VAR block we've added a byte-sized variable, Cog. This will be used to keep track of the ID of the cog started by the Start method, if any. Both Stack and Cog variables are global to the object; they can be used within any PUB or PRI block in the Output object. If they are modified by one method, other methods will see the new value when they are referenced.

The Start Method

For the Start method, we've decided it may be nice to know if it was successful or not. Since there are a limited number of cogs in the Propeller, the Start method may not be able to activate another cog every time it is called. For this reason, we'll make it return a Boolean (TRUE or FALSE) value as an indication of its outcome; the ": Success" in its declaration indicates it will return this value we chose to call Success. Each PUB and PRI method always returns a long value (4 bytes) whether or not it is specified to have one. When a method is designed to return a meaningful value, it is always good practice to declare a return value as we have done here. Our Success symbol becomes an alias for the method's built-in RESULT variable, so we can assign either Success or RESULT a value to have that value returned upon exit.

The body of the Start method now does two things: first it stops any existing process and then it starts a new process. It calls the Stop method first just in case Start has been called multiple times without first calling Stop from outside the object. Without that, a new cog would start up and overwrite another cog's workspace variables, such as Stack.

The next line is similar to our original but may seem a bit overwhelming because it is a compound expression. We'll dissect it a piece at a time from the inside out. The COGNEW portion of the line is exactly as it was before: cognew(Toggle(Pin, Delay, Count), @Stack). It activates another cog to run the Toggle method. What you may not have known is that COGNEW always returns the ID of the cog it started; 0 to 7, or -1 if no cog was available to start. In the prior version of the Output object, we simply ignored the return value. This time, however, we use COGNEW's return value in this expression and assign the result to a variable: Cog := cognew(Toggle(Pin, Delay, Count), @Stack) + 1. This expression says to execute COGNEW, take its returned value and add it to 1, then assign that result to the variable named Cog. The ':='is the assignment operator; similar to the equal sign '=' in other languages.

We'll use the Cog variable to remember the ID of the cog we started so we can later stop it if necessary. We'll explain why we added 1 to it in a moment.

We're not done with that line yet. To the left of the Cog := … part is the Success := assignment statement. So after the new cog's ID is returned, added to 1 and stored in Cog, that final value is also stored in the Success variable. Remember how Success is supposed to

be our Start method's Boolean return value? A Boolean result of FALSE is actually the numerical value 0 and TRUE is -1, but Boolean comparisons treat zero (0) as FALSE and any non-zero value (≠0) as TRUE. This is very convenient and is the reason we added 1 to COGNEW's return value; the range -1 to 7 becomes 0 to 8, and 0 (FALSE) means no cog was started while 1 to 8 (TRUE) means a cog was started.

So, in that single line of code we launched a new cog (hopefully), passed it the reference to the Toggle routine and stack space to use, stored the newly activated cog ID plus 1 in the variable Cog and used that final result to set Start's return value, the Success variable! This line demonstrates one of the most powerful features of the Spin language: compound expressions with assignable intermediate results.

The outer parentheses encasing the Cog :=… part are not required but we added them to help separate the two different variable assignments; Cog is assigned first then that result is assigned to Success. To assist you in studying complex expressions such as this one, the Propeller Tool temporarily bolds the matching pairs of parentheses that surround the current cursor position. Place the cursor in various positions on the line to see the effect. The figure below illustrates this; the star shows the cursor position, arrows show the bolded parentheses, and the shaded area is what is contained within those parentheses.

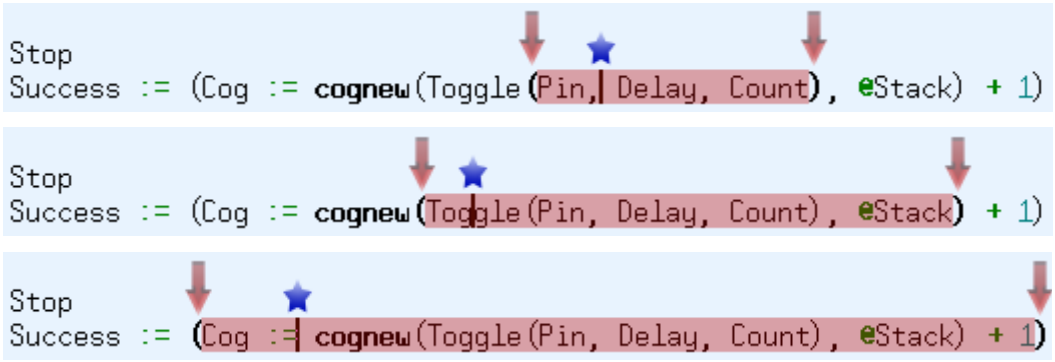


Figure 3-15: Matching Parentheses Bolded

Matching parentheses are temporarily displayed in bold for the expression group the cursor is currently within. Use this feature to study complex expressions.

The Stop Method

Our Stop method needs to stop the cog that was started by Start. The if Cog statement is a conditional structure meaning "if the Cog variable is TRUE execute the following indented block. " Remember, Cog was set to 0 if no cog was started, and set to 1 through 8 if a cog was started. Since 0 means FALSE and non-0 means TRUE, the IF statement is true only if we actually started a cog.

The COGSTOP statement is indented below the IF statement so it is executed only when the IF statement is true. The COGSTOP command deactivates the cog whose ID is indicated by its parameter: Cog~ - 1. This is another tricky but powerful expression in Spin. Remember the Post-Set operator, ~~, from earlier exercises? Well, a single ~ following a variable is the Post-Clear operator; it clears the variable preceding it to zero (0). These are called "post" operators because they perform their duty "after" the variable's original value is used by the expression that it is involved in. So Cog~ - 1 takes the value of Cog, subtracts 1, gives that value to the COGSTOP command, then clears Cog to zero (0). In effect, the cogstop(Cog~ - 1) statement stops the cog whose ID is Cog-1, then clears the Cog variable to 0 so future references to Cog reflect that there is no additional cog running.

The Active Method

The Active method is simple, it sets its return value, YesNo, to TRUE if Cog is greater than 0, FALSE otherwise. The > symbol is the Is Greater Than operator. Note that we could also have just set YesNo equal to Cog since zero is considered to be FALSE and non-zero is considered to be TRUE; that would have the additional advantage of being a true/false return value as well as the actual ID of the cog in use by this object.

The Toggle Method

We made a couple of minor but significant enhancements to the Toggle method. First, let's look at the last line, Cog~. Remember that if Start is called, it runs the Toggle method in another cog and stores the ID of that cog in the Cog variable. When Toggle terminates, that cog terminates as well, but the Cog variable would be left holding the ID of that cog, fooling the Active and Start methods into thinking its cog was still active. We put Cog~ at the end of Toggle to clear the Cog variable to zero (0) to maintain the code's integrity.

Remember we said we'd like to change Toggle to allow for an infinite loop as well as a finite loop? Our next change achieves that in a clever way. The Count parameter is the number of times to toggle the pin. That means it doesn't make sense to set Count equal to 0… who would want to toggle a pin zero times? So, we'll make 0 an exception case that means "toggle the pin infinitely."

We changed the loop from `repeat Count` to `repeat..while`. The `while` is at the end of the loop, three lines below `repeat`. This is another form of **REPEAT** loop structure called a "conditional one-to-many loop." It executes the statement block within it at least once, and iterates again and again as long as the "while" condition is true. In this case it repeats while `Count := --Count #> -1` is **TRUE** (ie: non-zero). This condition is another compound expression. The double-minus, '`--`' 'preceding `Count` is the Pre-Decrement operator; it decrements `Count` by 1 before its value is used by the expression. The `#>` is the Limit Minimum operator; it takes the value on its left and returns either that value, or the number on its right, whichever is greater. So each time this expression is evaluated, `Count` is decremented by 1, that result is limited to -1 or higher, and that final result is assigned back into `Count`. This has a clever effect that we'll explain next.

If `Toggle` was called with `Count` set to 2, the loop would execute two times, just like we want. After the first iteration, the `while Count := --Count #> -1` would decrement `Count`, making it 1, then would limit it to -1 or higher (still 1) and store that value in `Count`. Since the result, 1, is non-zero (**TRUE**) the loop would execute again. After the second iteration, the **WHILE** statement would decrement `Count`, making it 0, would limit that to -1 or higher (still 0) and store that in `Count`. Since 0 is **FALSE**, the **WHILE** condition terminates the loop.

That works for all normal `Count` values, but what about when `Toggle` is called with a `Count` of 0? After the first iteration, the `while Count := --Count #> -1` would decrement `Count`, making it -1, then would limit it to -1 or higher (still -1) and store that value in `Count`. Since the result, -1, is non-zero (**TRUE**) the loop would execute again. After the second iteration, the **WHILE** statement decrements `Count`, making it -2, limits that to -1 or higher (it is changed to -1) and stores that in `Count`. Once again, since the result, -1, is non-zero (**TRUE**) the loop would execute again.

So, if `Count` started out as 0, the loop iterates endlessly! If `Count` started out as greater than 0, it loops only that number of times!

Quick Review: Ex 7

- Objects:
  - Have no direct relationship with cogs.
  - Should call interface methods "Start" and "Stop" if they affect other cogs.
  - Should call interface method "Init" if it needs initialization.
- Spin language:
  - Variables defined in variable blocks are global to the object so modifications by one method are visible by other methods. See VAR, page 315.
  - Booleans: (See Constants (pre-defined), page 202 and Operators, page 249).
    - FALSE = 0
    - TRUE = -1; any non-zero (≠0) value is True for Boolean comparisons.
  - Compound expressions can include Intermediate Assignments, see page 253.
  - Operators:
    - "Pre"/"Post" operators perform their duty before/after the variable's value is used by the expression.
    - Assignment '**:=**' is similar to equal '**=**' in other languages, see Variable Assignment '**:=**', page 255.
    - Post-Clear '**~**' clears the variable preceding it to zero (0), see Sign-Extend 7 or Post-Clear '**~**', page 262.
    - Pre-Decrement '**--**' decrements the variable following it, giving the expression the result, see Decrement, pre- or post- '**- -**', page 257.
    - Is Greater Than '**>**' returns True if value on left-side is greater than that of right-side, see Boolean Is Greater Than '>', '>=', page 276.
    - Limit Minimum '**#>**' returns the greater of either the value on its left or its right, see Limit Minimum '#>', '#>=', page 260.
  - Methods: (See PUB, page 287).
    - Always return a long value (4 bytes) whether or not one is specified.
    - Contain a built-in local variable, RESULT, that holds its return value.
    - Return values are declared by following the method's name and parameters with a colon (:) and a descriptive return value name.
  - COGNEW returns the ID (0 to 7) of cog started; -1 if none, see COGNEW, page 189.
  - COGSTOP deactivates a cog by ID, see COGSTOP, page 193.
  - IF is a conditional structure that executes the indented block of code following it if the conditional statement is true, see IF, page 220.
  - REPEAT's conditional, one-to-many form: REPEAT WHILE *Condition* executes at least once and continue while *Condition* is true. See REPEAT, page 293.
- The Propeller Tool bolds matching parentheses pairs surrounding the cursor.

Exercise 8: Blinker2.spin – Many Objects, Many Cogs

Now let's make a new object that takes advantage of the enhancements to Output to use many cogs for many parallel processes. Here's the code:

Example Object: Blinker2.spin

```
{{ Blinker2.spin }}

CON
  MAXLEDS = 6                'Number of LED objects to use

OBJ
  LED[6] : "Output"


PUB Main
{Toggle pins at different rates, simultaneously}

  dira[16..23]~~                                  'Set pins to outputs
  LED[NextObject].Start(16, 3_000_000,   0)       'Blink LEDs
  LED[NextObject].Start(17, 2_000_000,   0)
  LED[NextObject].Start(18,   600_000, 300)
  LED[NextObject].Start(19, 6_000_000,  40)
  LED[NextObject].Start(20,   350_000, 300)
  LED[NextObject].Start(21, 1_250_000, 250)
  LED[NextObject].Start(22,   750_000, 200)       '<-Postponed
  LED[NextObject].Start(23,   400_000, 160)       '<-Postponed
  LED[0].Start(20, 12_000_000, 0)                 'Restart object 0
  repeat                                          'Loop endlessly


PUB NextObject : Index
{Scan LED objects and return index of next available LED object.
 Scanning continues until one is available.}

  repeat
    repeat Index from 0 to MAXLEDS-1
      if not LED[Index].Active
        quit
  while Index == MAXLEDS
```

Compile and download Blinker2. You should see six LEDs start blinking with different, independent, rates and periods. Look carefully, after about 8 seconds P20 will stop blinking and P22 will start. A few seconds later, P18 will stop and P23 will start, then P16 will stop and P20 will start again, but at a different rate. Eventually, all but P17 and P20 will cease. Can you figure out why it behaves this way? We'll explain it below.

The OBJ Block

In the object block we defined an array of Output objects, called LED, with six elements. This is so we can have six simultaneous processes running, each operating independently.

The Main Method

The first line of Main, dira[16..23]~~, sets I/O pins 16 through 23 to outputs. The I/O registers, DIRA, OUTA, and INA, can use this form to affect multiple contiguous pins. We are setting this group of I/O pins to outputs only to prevent confusing results due to the Propeller Demo Board's resistors between the I/O pairs in 18 to 23. If a cog is the only one making a particular pin an output, upon shutting down that pin becomes an input again which allows the resistor between it and its neighbor to affect the LED on it. We'll keep this application's cog active so results are clear.

The next nine lines, LED[…, call the Output object's Start method to activate a new cog and toggle different I/O pins at different rates. The lines in the form LED[NextObject].Start…, call the NextObject method to get an index value for the array. We'll explain the NextObject method in more detail soon, but simply put, it returns the index of the next available Output object in the LED array (i.e. the index of the first idle object) and pauses until one is available.

We only have six Output objects defined for the LED array, so the first six calls to Start are going to execute quickly, each one accessing LED indexes 0 through 5 and activating a total of 6 additional cogs. The first two have a Count parameter of 0, so they will toggle infinitely; the last four will terminate after the given number of toggles is performed.

The seventh line, LED[NextObject].Start(22, 750_000, 200) will first call NextObject to get the index of the next available object, but since all six objects are busy toggling pins, NextObject will wait and won't return to Main until it finds that an object has finished. As it turns out, the object at index 4 (I/O pin 20) finishes its task first and shuts down. The NextObject method then returns the number 4, allowing that object's Start method to execute, which will re-launch another cog to toggle pin 22. A similar process happens with the eighth line, LED[NextObject].Start(23, 400_000, 160); all objects are busy so NextObject postpones further operation until one becomes available, index 2 in this case.

Immediately after the eighth line is executed, the ninth line executes, LED[0].Start(20, 12_000_000, 0). This statement is unlike the previous in that it doesn't call NextObject, but

rather it uses a fixed index of 0. This means the LED object at index 0, which is busy toggling I/O pin 16 endlessly, suddenly has its Start method called again. This causes the cog that is toggling P16 to immediately stop and start again but with P20 instead.

The final line, repeat, is only there to keep the application's cog alive. It creates an endless loop that executes no additional code since there is nothing indented underneath it. If the application cog stopped, the I/O pins it directed to be outputs may switch back to inputs, causing strange-looking results due to the resistors between some pairs of LEDs on the Propeller Demo Board. If you are not using the Propeller Demo Board, the first line and last line of Main are not necessary.

The NextObject Method

We have six LED objects in this code and any number of them can be processing in parallel at any time. The point of the NextObject method is to tell us which one is available and to postpone future operations until one is available. To do this, it scans through all six LED objects looking for the first one that is not running as a parallel process and returns the LED-based index of that object. If all are currently running, it continues scanning until one becomes available. NextObject uses our Output object's Active method to assist with this.

There are two nested REPEAT loops. The outer loop, repeat..while Index == MAXLEDS iterates as long as Index equals MAXLEDS, 6 in this case. We learned how this type of REPEAT loop works in the previous exercise.

The inner REPEAT loop, repeat Index from 0 to MAXLEDS-1, is new to us, however. It is called a "counted loop" and repeats the indented block below it but for each iteration it sets the variable Index to a new value. Index is set to 0 for the first iteration, 1 for the second, etc., until the last iteration where Index equals MAXLEDS-1, or 5. This is an excellent way to adjust the operation within a loop based on how many times the loop has executed.

The next line, if not LED[Index].Active, is a conditional statement that executes the indented code below it if the LED object at Index is "not active." Since the inner loop changes the value of Index from 0 through 5 as it executes, this conditional statement calls the Active method of each of our LED objects, in order.

Once the condition is true (the LED object at Index is not active) the next line, quit, executes. The QUIT command is a special command for REPEAT loops only; it causes the REPEAT loop it is contained within to terminate immediately. When this happens, execution continues with the end of the outer REPEAT loop, the "while" condition.

If all LED objects are active, the inner loop will count, with Index, from 0 to 5, then Index will be 6 (MAXLEDS) when it exits, causing the outer loop to iterate again and the whole process starts over. If, however, an inactive LED object is found, the Index value will be less than

MAXLEDS, and the outer loop will terminate, causing the NextObject method to return the Index of the available object. That value is used by Main to select the right LED object to start.

Behind the Scenes

In the object block, we created an array of six Output objects. Each object that an application uses needs to be treated as its own individual entity with its critical data kept separate from that of any other object. So, since we needed the capabilities of six Output objects, we declared the need for six of them in the object block.

After compiling Blinker2, the Object View shows that there are six occurrences of the Output object in our application; the "[6]" that appears to the right of the Output object image. This is the Object View's way of illustrating the structure of our application, indicated by Figure 3-16.

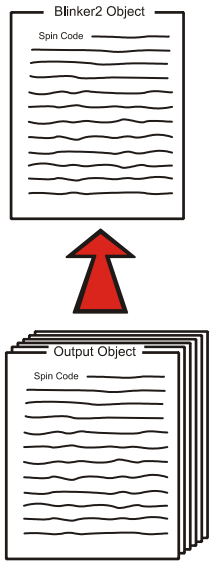


Figure 3-16: Blinker2 Application

There are six instances of the Output object. The application actually uses only one copy of its executable code and six copies of its global variable space.

Does this mean our application grew by the size of the Output object times six? Fortunately, the answer is no. The Propeller Tool optimizes the application's code such that, for every occurrence of an object only one copy of the object's code is included, but multiple copies of the object's global variables are created. This is because the code is considered to be static (unchanging) and exactly the same for each object. However, the object's global variables (defined in its VAR block), are not static; each object needs its own variable space in order to work independently without interference from other instances of itself.

Object Info Window

We can see this effect using the Propeller Tool's Object Info feature. First, change the object block in Blinker2 to specify only one instance of the Output object; `LED[1] : "Output"`. Don't run the code this way, it will not work, we're just experimenting for a moment.

Now, press the F8 key (or select Run → Compile Current → View Info…) to compile the application and display the Object Info window. Figure 3-17 shows how this should look.

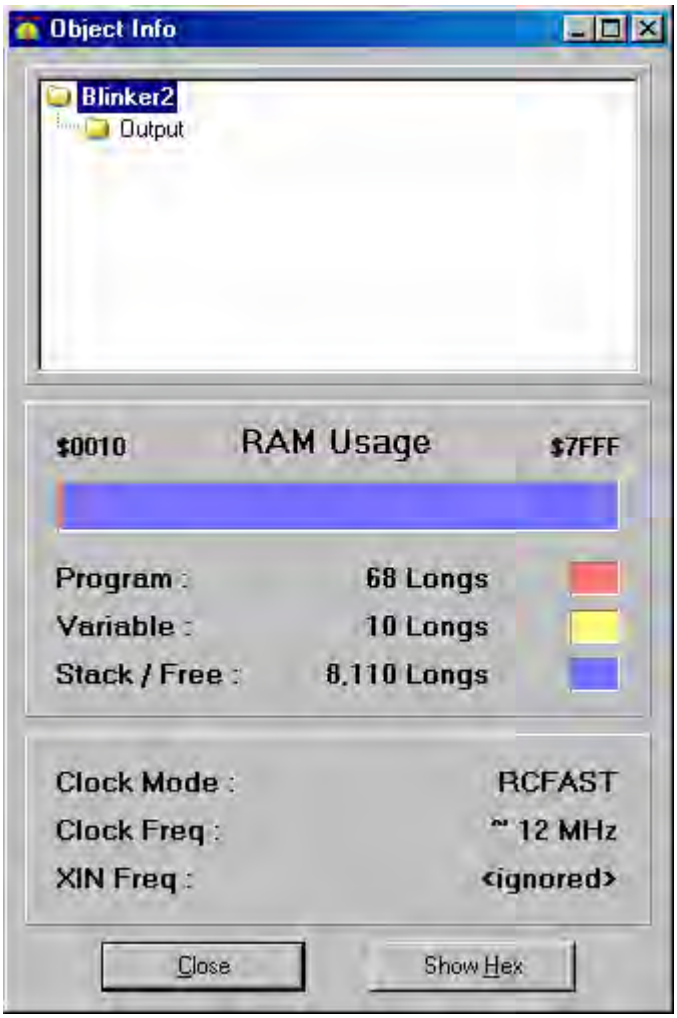


Figure 3-17: Object Info Window for Blinker2 Application

The top part of the window is the Info Object View; it is similar to the Object View. The center part of the window shows the application's RAM Usage. Notice that the "Program" (the compiled source code) itself consumes 68 longs of RAM and the "Variable" (the global variables) consumes 10 longs of space.

Now, close this window and change the object block back to the way it was, specifying six instances of the Output object; `LED[6] : "Output"`. Compile and view info again (F8 or Run → Compile Current → View Info…). Notice that now the program consumes 73 longs of RAM and the variable space consumes 60 longs. This is only 5 additional longs of program space but 50 additional longs of variable space. The extra program space is just overhead to deal with five additional objects but the variable space is six times its previous size; each object has its own global variable space. Our Blinker2 object doesn't define any global variable space, but Output defines nine longs for `Stack` and one byte for `Cog` for a total of 10 longs of space since an object's variable space is always long-aligned.

In the Object Info window, you can also click on the Output object to see how much space each individual instance of that object consumes. We can see that Output's program size is 21 longs and variable space is 10 longs.

Object Lifetime

When applications are compiled a binary image of the executable code is created. That binary image is what is actually downloaded into the Propeller and is usually what we are referring to when we say "application" or "Propeller Application."

The compiled code for each object used by an application is included within that binary image along with variable space for each instance of each of those objects.

During run time, the application may use any object for any amount of time; some may be used always and others only on occasion but all are consuming a static amount of memory for their code and variables.

For developers accustomed to programming with objects on a computer, this is an important concept to understand. On the Propeller an object's lifetime is static; whether or not it is actively in use at the time, it always requires a specific amount of memory in the application's binary image. On desktop/laptop computers, objects require a dynamic amount of memory because they are "created" and "destroyed" during the run-time process as is needed. On the Propeller, the objects are "created" at compile time and are never "created" or "destroyed" at run time because the act of doing so would fragment memory and cause indeterministic behaviour in real-time embedded systems.

This means that every instance of an object that is, or may be, required must be declared at compile time in the OBJ block, just as we did in Exercise 8 with the array of Output objects.

Quick Review: Ex 8

- Applications:
  - Use unique symbols, or elements of an array, for each distinct object in use.
  - Use one copy of an object's code and one or more copies of its global variables.
- Objects:
  - An object array may be created in object blocks similar to a variable array in variable blocks.
  - An object's lifetime is static, consuming a specific, static amount of memory regardless of whether or not it is active. This eliminates the possibility of fragmented memory during normal run-time use and ensures deterministic behavior in real-time systems.
- Spin language:
  - REPEAT command: (See REPEAT, page 293).
    - Finite, counted loop: REPEAT *Variable* FROM *Start* TO *Finish* where *Variable* is the variable to use as the counter and *Start* and *Finish* indicate the range.
    - The QUIT command works inside REPEAT loops only and causes the loop to terminate immediately, see QUIT, page 291.
  - I/O registers (DIR×, OUT×, and IN×) may use the form *reg*[*a*..*b*] to affect multiple contiguous pins; where *reg* is the register (DIR×, OUT×, or IN×) and *a* and *b* are I/O pin numbers, see DIRA, DIRB on page 212, OUTA, OUTB on page 280, and INA, INB on page 225.
- I/O pins are set to outputs only while a cog that set them that way remains active, see DIRA, DIRB, page 212.
- Compile & View Info: F8 key (or select Run → Compile Current → View Info…), see Object Info, page 55.

Exercise 9: Clock Settings

The Propeller chip's internal clock has two speeds, slow (≈ 20 KHz) and fast (≈ 12 MHz). Since we never specified any clock settings for our application, all previous exercises used the Propeller chip's default, internal RC clock in fast mode.

To specify the clock settings for the application, the top object file must set values for one or more special constants in a **CON** block. These constants are: **_CLKMODE**, **_CLKFREQ** and **_XINFREQ**.

We'll start with **_CLKMODE** first. Refer to Table 4-3: Clock Mode Setting Constants on page 180 for a listing of pre-defined symbolic values to set **_CLKMODE** to. For example, continuing with our Blinker2 object, changing the **CON** block as follows sets the clock mode to use the internal slow clock (only the **CON** block is shown here).

```
{{ Blinker2.spin }}

CON
  _CLKMODE = RCSLOW             'Set to internal slow clock
  MAXLEDS  = 6                  'Number of LED objects to use

<remaining code unchanged>
```

Try compiling and downloading Blinker2 now. Once the Propeller finishes the download/boot-up process, it switches to the **RCSLOW** clock mode and executes the application. Since the application is now running with a clock that is hundreds of times slower that before, the application will run much slower, taking more than 20 seconds for the fastest toggling pin, P20, to toggle off for the first time.

You can replace `_CLKMODE = RCSLOW` with `_CLKMODE = RCFAST` to have the application run with the internal fast clock (the default).

If you'd like to use an external clock, there are many more options for **_CLKMODE**. We'll assume you're using a 5 MHz external crystal, like the one that comes with the Propeller Demo Board.

Modify your code to match the following:

```
{{ Blinker2.spin }}

CON
  _CLKMODE = RCSLOW             'Set to internal slow clock
  _CLKMODE = XTAL1              'Set to ext. low-speed crystal
  _XINFREQ = 5_000_000          'Frequency on XIN pin is 5 MHz
  MAXLEDS  = 6                  'Number of LED objects to use

<remaining code unchanged>
```

Here we set **_CLKMODE** to **XTAL1** which configures the clock mode for an external low-speed crystal and configures the Propeller internal oscillator gain circuitry to drive a 4 MHz to 16 MHz crystal. Besides the crystal itself (which should be connected to the XI and XO pins), no other external circuitry is required for this clock configuration.

Whenever external crystals or clocks are used, either **_XINFREQ** or **_CLKFREQ** must be specified in addition to **_CLKMODE**. **_XINFREQ** specifies the frequency coming into the XI pin (Crystal Input pin). **_CLKFREQ** specifies the System Clock frequency. The two are related by PLL settings which we'll discuss later.

In this example we specified an **_XINFREQ** value of 5 million to indicate that the frequency on the XI pin is 5 MHz, since we have a 5 MHz crystal connected to XI and XO. Once that is specified, the **_CLKFREQ** value is automatically calculated and set by the Propeller Tool.

You could also have specified a **_CLKFREQ** of 5 MHz (instead of **_XINFREQ**) and the proper **_XINFREQ** value would automatically be set by the Propeller Tool. However, it is more typical to specify the **_XINFREQ** value since **_CLKFREQ** is directly affected by PLL settings. In our example, both **_XINFREQ** and **_CLKFREQ** end up with the same value, but a later example will show how they can typically differ.

If you compile and download Blinker2 now, you should see the LEDs toggle at slightly less than half the speed as in Exercise 8. Our settings specified an external 5 MHz crystal instead of the internal 12 MHz oscillator.

So why would anyone want to use an external crystal that is slower than the internal clock? Two reasons: 1) for accuracy; the internal clock is not very accurate from chip to chip or across voltage variances but external crystals or clock/oscillators are typically very accurate, and 2) the phase-locked loop (PLL) can only be used with external clock sources.

Try the following example:

```
{{ Blinker2.spin }}

CON
  _CLKMODE = XTAL1 + PLL4X      'Set to ext low-speed crystal, 4x PLL
  _XINFREQ = 5_000_000          'Frequency on XIN pin is 5 MHz
  MAXLEDS  = 6                  'Number of LED objects to use

<remaining code unchanged>
```

Here we changed the **_CLKMODE** setting slightly by adding the + PLL4X value. This configures the clock mode to use the internal phase-locked loop (PLL) to wind up the XIN frequency by four times, resulting in a System Clock frequency of 5 MHz * 4 = 20 MHz.

Try compiling and downloading Blinker2 with these settings. You should see the LEDs blink at a faster rate than you've seen before.

NOTE: Since we specified **_XINFREQ** here, **_CLKFREQ** is automatically calculated to be 20 MHz. If we had specified a **_CLKFREQ** value of 5 MHz instead, adding the **PLL4X** setting would have calculated an **_XINFREQ** value of 1.25 MHz, which doesn't match our external crystal's frequency. This is why it is more common to specify an XIN frequency (**_XINFREQ**) rather than a clock frequency (**_CLKFREQ**).

The Clock PLL circuit, when enabled, always winds up the frequency by 16 times, but you can select any of the 1x, 2x, …16x taps for the final System Clock frequency using the settings **PLL1X**, **PLL2X**, **PLL4X**, **PLL8X** and **PLL16X**.

Try changing **_CLKMODE** from XTAL1 + PLL4x to XTAL1 + PLL16x and download again. That configures the System Clock to be 5 MHz * 16 = 80 MHz! Most of the LEDs blink so quickly that they appear to be solidly on.

Exercise 10: Clock-Related Timing

The last exercise may have made you aware of something; our Output object is easily affected by the clock frequency. It relies on a specific, hard-coded time-base but subordinate objects (those that are not the top object) should never do that because they cannot predict what the clock frequency will be for the many applications they may be used for. Additionally, the Propeller application can change the System Clock frequency a number of times throughout its run time.

Suppose that we really intended to make an Output object that toggles pins at a specific rate that is essentially clock independent. This means that it must respond dynamically to the System Clock frequency. Below is the modified code; make sure to edit your code to match.

Example Object: Output.spin

```
{{ Output.spin }}

VAR
  long  Stack[9]                        'Stack space for new cog
  byte  Cog                             'Hold ID of cog in use, if any


PUB Start(Pin, DelayMS, Count): Success
{{Start new blinking process in new cog; return True if successful.}}

  Stop
  Success := (Cog := cognew(Toggle(Pin, DelayMS, Count), @Stack) + 1)


PUB Stop
{{Stop toggling process, if any.}}

  if Cog
    cogstop(Cog~ - 1)


PUB Active: YesNo
{{Return TRUE if process is active, FALSE otherwise.}}

  YesNo := Cog > 0


PUB Toggle(Pin, DelayMS, Count)
{{Toggle Pin, Count times with DelayMS milliseconds ~~clock cycles~~
  in between.  If Count = 0, toggle Pin forever.}}

  dira[Pin]~~                                   'Set I/O pin to output…
  repeat                                        'Repeat the following
    !outa[Pin]                                  '  Toggle I/O Pin
    waitcnt(clkfreq / 1000 * DelayMS + cnt)     '  Wait for DelayMS…
  while Count := --Count #> -1                  'While not 0 (make min…
  Cog~                                          'Clear Cog ID variable
```

We modified the Start and Toggle methods by changing the Delay parameter to DelayMS, meaning "delay in units of milliseconds." Then we modified the waitcnt… statement such that instead of waiting a fixed number of clock cycles, it calculates the number of clock cycles that there are in DelayMS milliseconds of time. CLKFREQ is a command that returns the current System Clock frequency in Hertz (cycles per second). Its value is set by the Propeller Tool at compile time and also by the CLKSET command at run time; see CLKSET on page 183. There are 1,000 milliseconds per second and CLKFREQ is the number of clock cycles per second, so clkfreq / 1000 * DelayMS is the number of clock cycles in DelayMS milliseconds of time.

With this modification, regardless of the application's start-up frequency, or how often the application changes the frequency during run time, the Output object will recalculate the proper delay each time through its loop.

Now, of course, we need to modify our Blinker2 object to adjust the DelayMS parameters appropriately. Enter the code modifications shown in the listing on page 136. Note that we entered the _CLKMODE and _XINFREQ settings just as we had left them from the last exercise.

Example Object: Blinker2.spin

```
{{ Blinker2.spin }}

CON
  _CLKMODE = XTAL1 + PLL4X      'Set to ext low-speed crystal, 4x PLL
  _XINFREQ = 5_000_000          'Frequency on XIN pin is 5 MHz
  MAXLEDS = 6                   'Number of LED objects to use


OBJ
  LED[6] : "Output"


PUB Main
{Toggle pins at different rates, simultaneously}

  dira[16..23]~~                                'Set pins to outputs
  LED[NextObject].Start(16, 250,   0)           'Blink LEDs
  LED[NextObject].Start(17, 500,   0)
  LED[NextObject].Start(18,  50, 300)
  LED[NextObject].Start(19, 500,  40)
  LED[NextObject].Start(20,  29, 300)
  LED[NextObject].Start(21, 104, 250)
  LED[NextObject].Start(22,  63, 200)           '<-Postponed
  LED[NextObject].Start(23,  33, 160)           '<-Postponed
  LED[0].Start(20, 1000, 0)                     'Restart object 0
  repeat                                        'Loop endlessly

PUB NextObject : Index
{Scan LED objects and return index of next available LED object.
 Scanning continues until one is available.}

  repeat
    repeat Index from 0 to MAXLEDS-1
      if not LED[Index].Active
        quit
  while Index == MAXLEDS
```

In Main, we adjusted the second parameter of the all the calls to Start from "delay in clock cycles" to "delay in milliseconds." Compile and download the Blinker2 object now. Notice that the rate at which each LED blinks is the same as it was when we used the internal fast clock. Try increasing the clock speed by changing _CLKMODE from XTAL1 + PLL4X to XTAL1 + PLL16X. You should not see any change in the blink rates even though we just multiplied the clock frequency by four!

Keep in mind that the accuracy of the internal clock on your particular Propeller chip can play a big role in the way this example looks, especially when using the RCSLOW mode.

There are two techniques for using the WAITCNT command but we only demonstrated one of them. For further tips regarding timing, see the WAITCNT command on page 322.

Quick Review: Ex 9 & 10

- Clock:
  - The internal clock has two speeds, slow (≈ 20 KHz) and fast (≈ 12 MHz).
  - To specify clock settings for an application, the top object file sets values for one or more special constants: _CLKMODE, _CLKFREQ and _XINFREQ.
  - Whenever external crystals or clocks are used, either _XINFREQ or _CLKFREQ must be specified in addition to _CLKMODE.
  - _CLKMODE specifies the clock mode: internal/external, oscillator gain, PLL settings, etc. See _CLKMODE, page 180.
  - _XINFREQ specifies the frequency coming into the XI pin (Crystal Input pin). See _XINFREQ, page 337.
  - _CLKFREQ specifies the System Clock frequency. See _CLKFREQ, page 177.
  - Use the internal clock for convenience where accuracy doesn't matter. Use an external clock for accuracy or when the phase-locked loop (PLL) is needed.
- Timing:
  - Subordinate objects can't rely on a specific, hard-coded time-base since applications which use them may change the clock frequency.
  - Use the CLKFREQ command to get the current System Clock frequency in Hertz for timing calculations. See CLKFREQ, page 175.

Exercise 11: Library Objects

The Propeller Tool comes with a library of objects created by Parallax engineers. These objects perform many useful functions such as serial communication, floating-point math, number-to-string and string-to-number conversion, and TV display generation, using standard PC-style keyboards, mice and monitors, etc.

The Propeller Object Library is simply a folder containing Propeller object files that are automatically created during the Propeller Tool software installation. You can get to the Propeller Library folder by selecting "Propeller Library" from the Recent Folders list; see Figure 3-18. After selecting the Propeller Library, the Files List will display all the available objects.

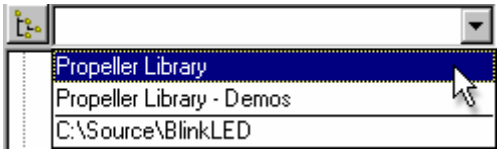


Figure 3-18: Propeller Library Browsing

Select "Propeller Library" from the Integrated Explorer's Recent Folders list to quickly browse to the library folder.

Let's use some of them now. Create a new file and enter the code below. The highlighted items are important for the discussion following the code.

Example Object: Display.spin

```
{{ Display.spin }}

CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

OBJ
  Num   :       "Numbers"
  TV    :       "TV_Terminal"

PUB Main | Temp
  Num.Init                                      'Initialize Numbers
  TV.Start(12)                                  'Start TV Terminal

  Temp := 900 * 45 + 401                        'Evaluate expression
  TV.Str(string("900 * 45 + 401 = "))           'then display it and
  TV.Str(Num.ToStr(Temp, Num#DDEC))             'its result in decimal
  TV.Out(13)
  TV.Str(string("In hexadecimal it's  = "))     'and in hexadecimal
  TV.Str(Num.ToStr(Temp, Num#IHEX))
  TV.Out(13)
  TV.Out(13)

  TV.Str(string("Counting by fives:"))          'Now count by fives
  TV.Out(13)
  repeat Temp from 5 to 30 step 5
    TV.Str(Num.ToStr(Temp, Num#DEC))
    if Temp < 30
      TV.Out(",")
```

Save this object as "Display.spin" in a folder of your choice; for this example we'll use the "C:\Source\" folder.

In this example we use two Propeller Library objects, Numbers and TV_Terminal, to convert numeric values to strings and display them on a TV. Compile and download this example object and connect a TV (NTSC) display to the composite output (RCA jack) on the Propeller Demo Board. The TV display should show the following text:

```
900 * 45 + 401 = 40,901
In hexadecimal it's  = $9FC5
Counting by fives:
 5, 10, 15, 20, 25, 30
```

Look at what we just achieved! Using just a few lines of our own code plus two existing library objects and three resistors (on the Propeller Demo Board) we converted numeric values to text strings and generated a TV-compatible signal to display that text in real time on a standard TV! In fact, while you are reading this, a cog is keeping busy constantly generating an NTSC signal at 60 frames per second that the TV can lock onto.

The TV_Terminal object provides a great display for debugging purposes. Since the Propeller has many processors and can run quite fast, a real-time display such as a TV monitor (CRT or LCD) used for debugging purposes goes a long way toward developing optimal source code. We recommend using this technique along with the usual debugging techniques to speed up development time.

Let's look at some important parts of our code now. The first new item in our code is the `| Temp` that appears in Main's declaration line. Don't be fooled, this may look like a return variable declaration, but it is not. The pipe symbol '|' indicates we are declaring local variables next. So, `| Temp` declares that `Temp` is a long-sized local variable for `Main`.

Next we have two very important statements, `Num.Init` and `TV.Start(12)`. These two statements initialize the Numbers object and start the TV_Terminal object (on pins 12, 13 and 14), respectively. Each of these objects requires some kind of initialization before using it. Numbers requires that its `Init` method is called to initialize some internal registers. TV_Terminal requires that its `Start` method is called to configure the proper output pins and to start two more cogs to generate the display signals. Objects typically indicate these requirements in their documentation, but it is common that they include an `Init` or a `Start` method if they require some initial setup before use.

The next line performs some arithmetic and sets our local variable, `Temp`, to the result. We'll use this result soon.

The next three statements create the first line of text on the TV display: `9 * 45 + 401 = 40,901`. The `TV.Str` method outputs a zero-terminated string to the display. Its parameter, `string("900 * 45 + 401 = ")` is new to us. **`STRING`** is a directive that creates a zero-terminated string of characters (multiple bytes of character data followed by a zero; sometimes called a z-string) and returns the address of that string. Most methods that deal with strings require just the address of the starting character and for the string to end with a byte equal to zero. `TV.Str` method's parameter requires exactly that, the address of a zero-

terminated string. So the line `TV.Str(string("900 * 45 + 401 = "))` causes the string "900 * 45 + 401 = " to be displayed on the TV.

The next statement, `TV.Str(Num.ToStr(Temp, Num#DDEC))` prints the "40,901" part of the line. The `Num.ToStr` method converts the numeric value in `Temp` into a string using delimited decimal format and returns the address of that string. `Temp`, of course, holds the long-sized result of our earlier expression: 40901. The `Num#DDEC` part is new to us, however. The `#` symbol when used this way is an Object-Constant reference; it is used to reference a constant that was defined in another object. In this case, `Num#DDEC` refers to the "format constant" `DDEC` that is declared within the Numbers object. As defined by Numbers, `DDEC` stands for Delimited Decimal and holds a value that indicates to the `ToStr` method that it should format the number with a thousands-group delimiter; a comma in this case. So, `ToStr` creates a z-string equal to "40,901" and returns the address of it. `TV.Str` then outputs that string onto the display. Read the documentation in the Numbers object for more information about this and other format constants.

`TV.Out(13)` outputs a single byte, 13, to the display. The 13 is the ASCII code for a carriage return (a non-visible character) and causes the TV_Terminal object to move to the next text line. We do this in preparation for the next string we'll print afterward.

Work and Library Folders

When our Display object is compiled, the Object View displays the structure shown below. This shows us that our Display object uses the Numbers and TV_Terminal objects and the TV_Terminal object uses the TV and Graphics objects.

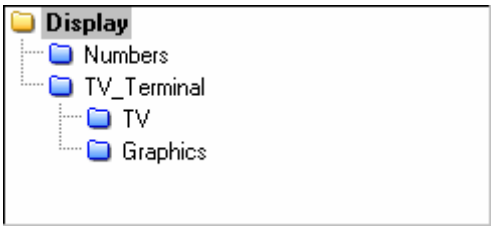


Figure 3-19: Object View of Display Application

Yellow folders indicate objects in the "work" folder. Blue folders indicate objects in the "library" folder.

The folder icons in front of each object are different colors to indicate their individual folder locations. Objects with yellow folders are in the "work" folder while those with blue folders

are in the "library" folder. From this display we can see that the Propeller Tool found the Numbers, TV_Terminal, TV and Graphics objects in the library folder and the Display object in the work folder.

Remember that we saved our Display object in the C:\Source folder? When an application is compiled, the folder that the top object file is stored within becomes known as the work folder. If that file refers to other objects, the work folder is the first place where the Propeller Tool looks for them. If the referenced object is not in the work folder, the library folder is searched next. If an object in the library folder refers to another object, the library folder is searched for that other object. An error occurs if referenced objects are not found in either the work folder or the library folder.

Due to this nature, it can be said that every application is composed entirely of files from as many as two folders; the work folder and/or the library folder. Keep this in mind while building your applications.

You can find out the location of each object, and the work and library folders, by pointing the mouse at each object in the Object View. In the figures below we see that Display is in C:\Source (the "work" folder) and Numbers is in C:\Program Files\Parallax Inc\Propeller Tool (the "library" folder).

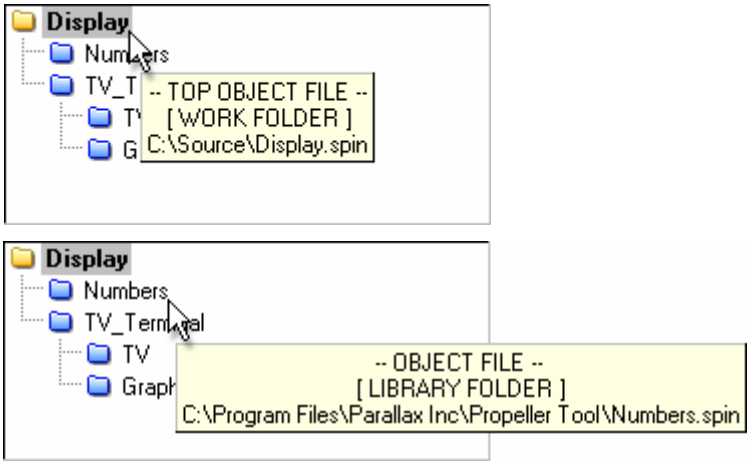


Figure 3-20:
Object View Hints for Display Application

The work and library folder paths can be seen in the hint messages

Exercise 12: Whole and Real Numbers

The Propeller is a 32-bit device and can naturally handle whole numbers as signed integers (-2,147,483,648 to 2,147,483,647) both in constants or in run-time math expressions. However, for real numbers (those with both integer and fraction components) the compiler supports floating-point format (single-precision, IEEE-754 compliant) for constants, and there are library objects that allow for run-time floating-point math operations.

Pseudo-Real Numbers

For handling real numbers, there are many possible techniques. One technique is to use integer math in a way that accommodates your real values as well as the run-time expressions involved. We call this pseudo-real numbers.

Having 32-bit integers built in to the Propeller provides us with a lot of "elbow room" for calculations. For example, perhaps we have an equation to multiply and divide values that have 2-digit fractions, like the following:

A = B * C / D

For our example, let's use A = 7.6 * 38.75 / 12.5 which evaluates to 23.56.

To solve this at run time, we can adjust all the equation's values upward by 2 digits to make them all integers, perform the math and then treat the rightmost 2 digits of the result as being the fractional portion. Multiplying each value by 100 achieves this. Here's the algebraic proof:

A = (B* 100) * (C * 100) / (D * 100)

A = (7.6 * 100) * (38.75 * 100) / (12.5 * 100)

A = 760 * 3875 / 1250

A = 2356

Since we multiplied all the original values by 100, we know that the final value is really 2356 / 100 = 23.56, but for most purposes we can keep it in integer form knowing that the rightmost two digits are really to the right of the decimal point.

The above solution works as long as each of the original values and each of the intermediate results never exceed the signed integer boundaries: -2,147,483,648 to 2,147,483,647.

The example presented next includes code that uses both the pseudo-real number technique as well as floating-point numbers.

Floating-Point Numbers

In many cases, expressions involving real numbers can be solved without using floating-point values and methods, such as with the pseudo-real number technique. Since solutions like the one above tend to execute much faster and consume less memory, it is recommended that you think carefully about whether or not you really need floating-point support before you actually use it. If you can afford the extra execution time and memory usage, floating-point support may be the best solution.

The Propeller Tool supports floating-point constants directly. The Propeller chip supports floating-point run-time expressions through the use of objects; ie: at run time the Spin Interpreter can only directly process integer-based expressions.

The next example object, RealNumbers.spin, demonstrates using integer constants (`iB`, `iC`, and `iD`) that are pre-translated to pseudo-real numbers, floating-point constants (`B`, `C`, and `D`) used in their native form by the FloatMath and FloatString library objects, and also those same floating-point constants translated to pseudo-real numbers at compile time.

Example Object: RealNumbers.spin

```
{{ RealNumbers.spin}}
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

  iB    =  760                  'Integer constants
  iC    = 3875
  iD    = 1250

  B     =  7.6                  'Floating-point constants
  C     = 38.75
  D     = 12.5

  K     = 100.0                 'Real-to-Pseudo-Real multiplier

OBJ
  Term  :       "TV_Terminal"
  F     :       "FloatMath"
  FS    :       "FloatString"

PUB Math
  Term.Start(12)

  {Integer constants (real numbers * 100) to do fast integer math}
  Term.Str(string("Pseudo-Real Number Result: "))
  Term.Dec(iB*iC/iD)

  {Floating-point constants using FloatMath and FloatString objects}
  Term.Out(13)
  Term.Str(string("Floating-Point Number Result: "))
  Term.Str(FS.FloatToString(F.FDiv(F.FMul(B, C), D)))

  {Floating-point constants translated to pseudo-real for fast math}
  Term.Out(13)
  Term.Str(string("Another Pseudo-Real Number Result: "))
  Term.Dec(trunc(B*K)*trunc(C*K)/trunc(D*K))
```

Compile and download RealNumbers.spin. It will display the following on a TV display:

```
Pseudo-Real Number Result: 2356
Floating-Point Number Result: 23.56
Another Pseudo-Real Number Result: 2356
```

The pseudo-real results, of course, each represent the value 23.56 but the entire value is shifted upwards by two digits to maintain integer math compatibility. With some additional code we could output it as 23.56 for display purposes.

The constants `iB`, `iC`, and `iD` are standard integer constants as we've seen before, but their values are really pseudo-real numbers representing the values in our example equation.

The constants `B`, `C`, `D`, and `K`, are floating-point constants (real numbers). The compiler automatically recognizes them as such and stores them in 32-bit single-precision floating-point format. They can be used in other compile-time floating-point expressions directly but at run time they should only be used with floating-point methods such as those found in the FloatMath and FloatString objects.

The statement `Term.Dec(iB*iC/iD)` uses the pre-translated pseudo-real constants as suggested by the Pseudo-Real Numbers technique, above. This is evaluated about 1.6 times faster than with the floating-point technique and takes much less code space.

The statement `Term.Str(FS.FloatToString(F.FDiv(F.FMul(B, C), D)))` calls FloatMath's `FMul` method to multiply the floating-point values `B` and `C`, then calls FloatMath's `FDiv` method to divide that result by the floating-point value `D`, translates the result to a string using FloatString's `FloatToString` method and displays that on the TV.

The statement `Term.Dec(trunc(B*K)*trunc(C*K)/trunc(D*K))` uses compile-time expressions inside of `TRUNC` directives to shift the floating-point constants `B`, `C`, and `D` upwards by two digits and truncate the values to integers. The resulting expression is equivalent to that of the first pseudo-real number equation `Term.Dec(iB*iC/iD)` but has the added benefit of allowing its component values to be defined in floating-point terms.

The **`TRUNC`** directive truncates fully resolved floating-point expressions to their integer form at compile time. It is required here since floating-point constant values can not be used directly by run-time expressions.

Context-Sensitive Compile Information

After an object has been compiled, the Propeller Tool displays context-sensitive compile information on the status bar (panel 5) about the source item the cursor is currently near or within. This is very useful in verifying and understanding the values of constants declared in an object. For example, compile this example by pressing F9 (or selecting the Run → Compile Current → Update Status menu option) and then place the cursor on the `iB` constant in the `CON` block. The status bar will temporarily highlight the context information and should look similar to the figure below.

Figure 3-21: Status Bar with Compile Information

After a compile operation, the status bar's panel 5 displays information about the source item nearest the cursor.

This tells us that our `iB` constant is defined by the `CON` block to be 760 decimal, or $2F8 hexadecimal.

Try placing the cursor on the `B` constant. The compile information should now read "CON B = 7.6 ($40F3_3333) Floating Point" to indicate this is a real number, in floating-point form, equal to 7.6 decimal ($40F3_3333 hexadecimal) This illustrates that floating-point values are encoded into 32 bits in a way that makes them incompatible with integer values.

In addition to symbols in `CON` and `DAT` blocks, the compile information displays shows the size, in bytes, of `PUB`/`PRI`/`DAT` blocks when the cursor is within that block. In our case, the `Math` method is 196 bytes long. This is a great feature to use when optimizing code for size; make small changes to code, press F9, check size against that of the previous code, and so on.

Quick Review: Ex 11 & 12

- Propeller Library:
  - Is a folder automatically created by the Propeller Tool installer.
  - Contains Parallax-made Propeller objects that perform useful functions.
  - The "Propeller Library" item in the Recent Folders list allows for quick access.
- Spin Language:
  - The pipe symbol '|' on method declaration lines declares a list of local variables for the method; see Parameters and Local Variables, page 289.
  - The STRING directive creates a zero-terminated string and returns its address; see STRING, page 310.
  - The # symbol forms an Object-Constant reference used to access constants defined in other objects; see Scope of Constants, page 199.
  - The TRUNC directive truncates floating-point constants to integers; see TRUNC, page 314.
- Work and Library Folders:
  - The Object View's folder icons indicate the object's location.
    - Objects with yellow folders are in the "work" folder.
    - Objects with blue folders are in the "library" folder.
  - Every application is composed entirely of files from as many as two folders; the work folder and/or the library folder.
- Integers and Real Numbers: (See CON, page 194, or Operators, page 249)
  - Integers are directly supported both in constants and in run-time expressions.
  - Real numbers, in floating-point format, are directly supported in constants and are indirectly supported at run time by special library objects.
  - In many cases, expressions involving real numbers can be solved without using floating-point values and methods.
- The Status Bar displays compile information about the source item nearest to the cursor. This includes CON/DAT block symbol's size/address and PUB/PRI/DAT block's size.

Where to go from here...

You should now have the knowledge you need to explore the Propeller chip on your own and develop your first applications. Use the rest of this manual as a reference to the Spin and Propeller Assembly languages, explore every existing library object that interests you and join the Propeller Forum to keep learning and sharing with other active Propeller chip users.