Propeller Manual

Version 1.1

WARRANTY

Parallax Inc. warrants its products against defects in materials and workmanship for a period of 90 days from receipt of product. If you discover a defect, Parallax Inc. will, at its option, repair or replace the merchandise, or refund the purchase price. Before returning the product to Parallax, call for a Return Merchandise Authorization (RMA) number. Write the RMA number on the outside of the box used to return the merchandise to Parallax. Please enclose the following along with the returned merchandise: your name, telephone number, shipping address, and a description of the problem. Parallax will return your product or its replacement using the same shipping method used to ship the product to Parallax.

14-DAY MONEY BACK GUARANTEE

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax Inc. will refund the purchase price of the product, excluding shipping/handling costs. This guarantee is void if the product has been altered or damaged. See the Warranty section above for instructions on returning a product to Parallax.

COPYRIGHTS AND TRADEMARKS

This text is available in printed format from Parallax Inc. Because we print the text in volume, the consumer price is often less than typical retail duplication charges.

ISBN 9-781928-982470

1.1.0-09.03.05-HKTP

DISCLAIMER OF LIABILITY

Parallax Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, or any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax Inc. is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your Propeller microcontroller application, no matter how life-threatening it may be.

INTERNET DISCUSSION LISTS

We maintain active web-based discussion forums for people interested in Parallax products. These forums are accessible at forums.parallax.com:

- Propeller chip – This list is specifically for our customers using Propeller chips and products.
- BASIC Stamp – This list is widely utilized by engineers, hobbyists and students who share their BASIC Stamp projects and ask questions.
- Stamps in Class® – Created for educators and students, subscribers discuss the use of the Stamps in Class series of tutorials in their courses. The list provides an opportunity for both students and educators to ask questions and get answers.
- HYDRA – for enthusiasts of the Propeller-powered HYDRA videogame development system.
- Parallax Educators – A private forum exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this group to obtain feedback on our curricula and to provide a place for educators to develop and obtain Teacher's Guides.
- Robotics – Designed for Parallax robots, this forum is intended to be an open dialogue for robotics enthusiasts. Topics include assembly, source code, expansion, and manual updates. The Boe-Bot®, Toddler®, SumoBot®, HexCrawler and QuadCrawler robots are discussed here.
- SX Microcontrollers and SX-Key – Discussion of programming the SX microcontroller with Parallax assembly language SX – Key® tools and 3rd party BASIC and C compilers.
- Javelin Stamp – Discussion of application and design using the Javelin Stamp, a Parallax module that is programmed using a subset of Sun Microsystems' Java® programming language.

ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. If you find an error, please let us know by sending an email to editor@parallax.com. We continually strive to improve all of our educational materials and documentation, and frequently revise our texts. Occasionally, an errata sheet with a list of known errors and corrections for a given text will be posted to our web site, www.parallax.com. Please check the individual product page's free downloads for an errata file.

SUPPORTED HARDWARE AND FIRMWARE

This manual is valid with the following hardware and firmware versions:

Hardware	Firmware
P8X32A-D40 P8X32A-Q44 P8X32A-M44	P8X32A v1.0

CREDITS

Authorship: Jeff Martin. Format & Editing, Stephanie Lindsay.
Cover Art: Jen Jacobs; Technical Graphics: Rich Allred; with many thanks to everyone at Parallax Inc.

Table of Contents

Preface

Thank you for purchasing a Propeller chip. You will be spinning your own programs in no time!

Propeller chips are incredibly capable multiprocessor microcontrollers; the much-anticipated result of over eight years of the intense efforts of Chip Gracey and the entire Parallax Engineering Team.

This book is intended to be a reference guide to Propeller chips and their native programming languages, Spin and Propeller Assembly. For a programming tutorial and Propeller Tool details, please refer to the on-line help that is installed with the Propeller Tool software. Have fun!

Despite our best efforts, there are bound to be questions unanswered by this manual alone. Check out our Propeller chip discussion forum – (accessible from www.parallax.com via the Support → Discussion Forums menu) – this is a group especially for Propeller users where you can post your questions or review discussions that may have already answered yours.

NEW

In addition to the forum, visit the Propeller Object Exchange (obex.parallax.com) for free access to hundreds of Propeller objects made by customers and Parallax Engineers. Besides being immediately useful for your own applications, Propeller objects written by various authors are a great resource for studying techniques and tricks employed by the very active Propeller community.

Editor's Note: About Version 1.1

The major content additions, corrections, and deletions that were made to Propeller Manual v1.0 to produce this edition are highlighted in the PDF version of this document of Propeller Manual v1.1. We recommend that if you previously read the original edition of the Propeller Manual. A complete record of changes can be found in the Propeller Manual Supplement and Errata v1.4. Both documents are available for download at www.parallax.com/Propeller.

Most significantly, the former *Chapter 2: Using the Propeller Tool*, was moved to the Propeller Tool's Online Help system where it can be updated frequently to stay in sync with the enhancements to the development software. Likewise, the former *Chapter 3: Propeller Programming Tutorial* was also moved to the Propeller Tool's Online Help system where it can be expanded more readily.

Additional important changes include:

- An additional Propeller Assembly instruction was added; TESTN (see page 347)
- The following sections were rewritten for clarity:
  - ADDSX on page 262
  - ADDX on page 264
  - CALL on page 268
  - CMPSX on page 277
  - CMPX on page 280
  - JMPRET on page 300
  - ORG on page 328
  - RES on page 339
  - RET on page 342
  - SUBSX on page 352
  - SUBX on page 354
- Extensive enhancements to the following sections were made to provide detail on previously undocumented features:
  - BYTE on page 51
  - COGINIT on page 76
  - COGNEW on page 78
  - DAT on page 99
  - LONG on page 128
  - WORD on page 227
- Extensive revisions were made to The Structure of Propeller Assembly section beginning on page 238, as well as to the start of the Assembly Language Elements section which begins on page 250.
- Concise Truth Tables have been added above the Explanation section for each Propeller Assembly instruction. Each of these truth tables include key value and flag combinations that reveal important aspects of the related instruction's nature.
- Individual Effects and Registers were given their own sections in the Assembly Language Reference for easier locating while scanning the manual.
- Multiplication, Division, and Square Root examples were added to Appendix B.
- Hundreds of important details were enhanced or corrected throughout. See the manual's PDF version, or the Supplement and Errata v1.4, for more information.

Chapter 1: Introducing the Propeller Chip

This chapter describes the Propeller chip hardware. To fully understand and use the Propeller effectively, it's important to first understand its hardware architecture. This chapter presents the details of the hardware such as package types, package sizes, pin descriptions, and functions.

Concept

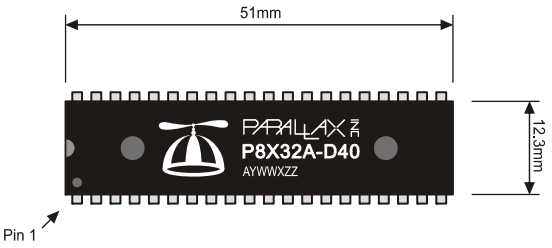
The Propeller chip is designed to provide high-speed processing for embedded systems while maintaining low current consumption and a small physical footprint. In addition to being fast, the Propeller provides flexibility and power through its eight processors, called cogs, that can perform simultaneous independent or cooperative tasks, all while maintaining a relatively simple architecture that is easy to learn and utilize.

The resulting design of the Propeller frees application developers from common complexities of embedded systems programming. For example:

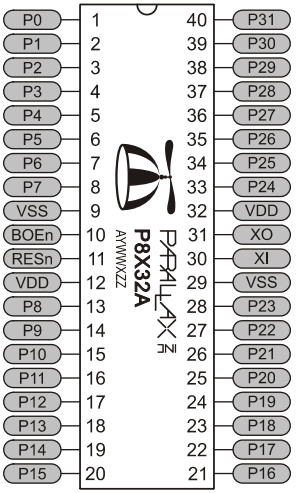
- The memory map is flat. There is no need for paging schemes with blocks of code, data or variables. This is a big time-saver during application development.
- Asynchronous events are easier to handle than they are with devices that use interrupts. The Propeller has no need for interrupts; just assign some cogs to individual, high-bandwidth tasks and keep other cogs free and unencumbered. The result is a more responsive application that is easier to maintain.
- The Propeller Assembly language features conditional execution and optional result writing for each individual instruction. This makes critical, multi-decision blocks of code more consistently timed; event handlers are less prone to jitter and developers spend less time padding, or squeezing, cycles here and there.

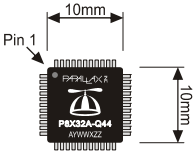
Package Types

The Propeller chip is available in the package types shown here.

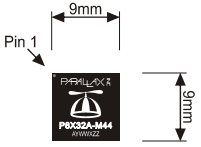


P8X32A-D40 ***40-pin DIP***

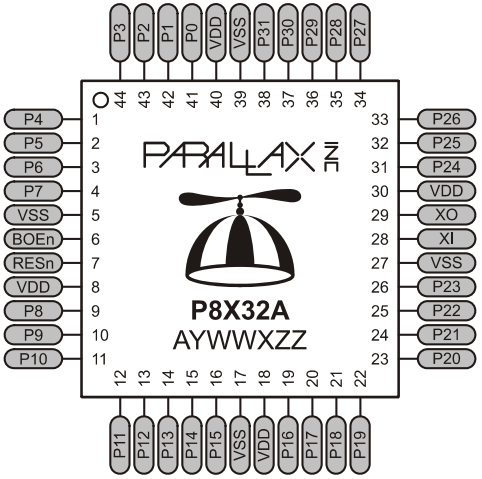




P8X32A-Q44
44-pin LQFP



P8X32A-M44
44-pin QFN



Pin Descriptions

IMPROVED

Table 1-1: Pin Descriptions		
Pin Name	**Direction**	**Description**
P0 – P31	I/O	General purpose I/O Port A. Can source/sink 40 mA each at 3.3 VDC. Logic threshold is ≈ ½ VDD; 1.65 VDC @ 3.3 VDC. The pins shown below have a special purpose upon power-up/reset but are general purpose I/O afterwards. P28 - I2C SCL connection to optional, external EEPROM. P29 - I2C SDA connection to optional, external EEPROM. P30 - Serial Tx to host. P31 - Serial Rx from host.
VDD	---	3.3 volt power (2.7 – 3.3 VDC).
VSS	---	Ground.
BOEn	I	Brown Out Enable (active low). Must be connected to either VDD or VSS. If low, RESn becomes a weak output (delivering VDD through 5 KΩ) for monitoring purposes but can still be driven low to cause reset. If high, RESn is CMOS input with Schmitt Trigger.
RESn	I/O	Reset (active low). When low, resets the Propeller chip: all cogs disabled and I/O pins floating. Propeller restarts 50 ms after RESn transitions from low to high.
XI	I	Crystal Input. Can be connected to output of crystal/oscillator pack (with XO left disconnected), or to one leg of crystal (with XO connected to other leg of crystal or resonator) depending on CLK Register settings. No external resistors or capacitors are required.
XO	O	Crystal Output. Provides feedback for an external crystal, or may be left disconnected depending on CLK Register settings. No external resistors or capacitors are required.

The Propeller (P8X32A) has 32 I/O pins (Port A, pins P0 through P31). Four of these I/O pins, P28-P31 have a special purpose upon power-up/reset. At power-up/reset, pins P30 and P31 communicate with a host for programming and P28 and P29 interface to an external 32 KB EEPROM (24LC256).

Specifications

Table 1-2: Specifications	
Model	P8X32A
IMPROVED Power Requirements	3.3 volts DC. (Max current draw must be limited to 300 mA).
External Clock Speed	DC to 80 MHz (4 MHz to 8 MHz with Clock PLL running)
System Clock Speed	DC to 80 MHz
Internal RC Oscillator	12 MHz or 20 kHz (approximate; may range from 8 MHz – 20 MHz, or 13 kHz – 33 kHz, respectively)
Main RAM/ROM	64 K bytes; 32 KB RAM + 32 KB ROM
Cog RAM	512 x 32 bits each
RAM/ROM Organization	Long (32-bit), Word (16-bit), or Byte (8-bit) addressable
I/O pins	32 CMOS signals with VDD/2 input threshold.
IMPROVED Current Source/Sink per I/O	40 mA
Current Draw @ 3.3 vdc, 70 °F	500 µA per MIPS (MIPS = Freq in MHz / 4 * Number of Active Cogs)

Hardware Connections

Figure 1-1 shows an example wiring diagram that provides host and EEPROM access to the Propeller chip. In this example the host access is achieved through the Propeller Plug device (a USB to TTL serial converter).

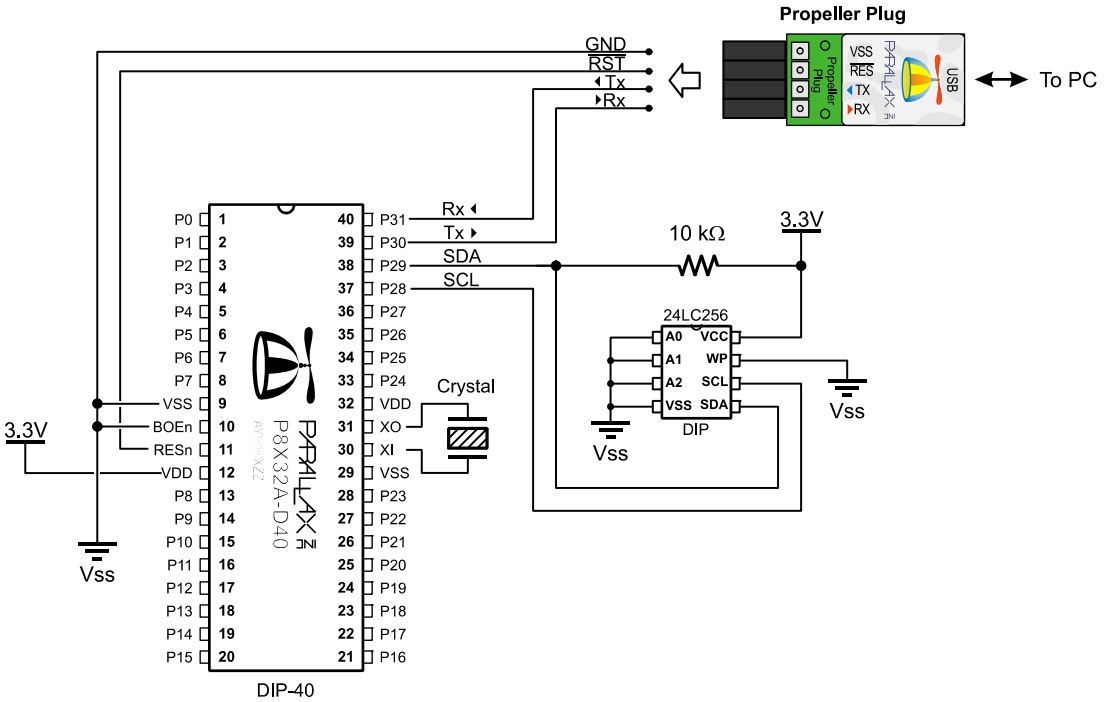


Figure 1-1: Example wiring diagram that allows for programming the Propeller chip and an external 32 Kbyte EEPROM, and running the Propeller with an external crystal.

Boot Up Procedure

Upon power-up (+ 100 ms), RESn low-to-high, or software reset:

1. The Propeller chip starts its internal clock in slow mode (≈ 20 kHz), delays for 50 ms (reset delay), switches the internal clock to fast mode (≈ 12 MHz), and then loads and runs the built-in Boot Loader program in the first processor (Cog 0).
2. The Boot Loader performs one or more of the following tasks, in order:
    a. Detects communication from a host, such as a PC, on pins P30 and P31. If communication from a host is detected, the Boot Loader converses with the host to identify the Propeller chip and possibly download a program into Main RAM and optionally into an external 32 KB EEPROM.
    b. If no host communication was detected, the Boot Loader looks for an external 32 KB EEPROM (24LC256) on pins P28 and P29. If an EEPROM is detected, the entire 32 KB data image is loaded into the Propeller chip's Main RAM.
    c. If no EEPROM was detected, the boot loader stops, Cog 0 is terminated, the Propeller chip goes into shutdown mode, and all I/O pins are set to inputs.
3. If either step 2a or 2b was successful in loading a program into the Main RAM, and a suspend command was not given by the host, then Cog 0 is reloaded with the built-in Spin Interpreter and the user code is run from Main RAM.

Run-Time Procedure

A Propeller Application is a user program compiled into its binary form and downloaded to the Propeller chip's RAM and, possibly, external EEPROM. The application consists of code written in the Propeller chip's Spin language (high-level code) with optional Propeller Assembly language components (low-level code). Code written in the Spin language is interpreted during run time by a cog running the Spin Interpreter while code written in Propeller Assembly is run in its pure form directly by a cog. Every Propeller Application consists of at least a little Spin code and may actually be written entirely in Spin or with various amounts of Spin and assembly. The Propeller chip's Spin Interpreter is started in Step 3 of the Boot Up Procedure, above, to get the application running.

Once the boot-up procedure is complete and an application is running in Cog 0, all further activity is defined by the application itself. The application has complete control over things like the internal clock speed, I/O pin usage, configuration registers, and when, what and how

many cogs are running at any given time. All of this is variable at run time, as controlled by the application, including the internal clock speed.

Shutdown Procedure

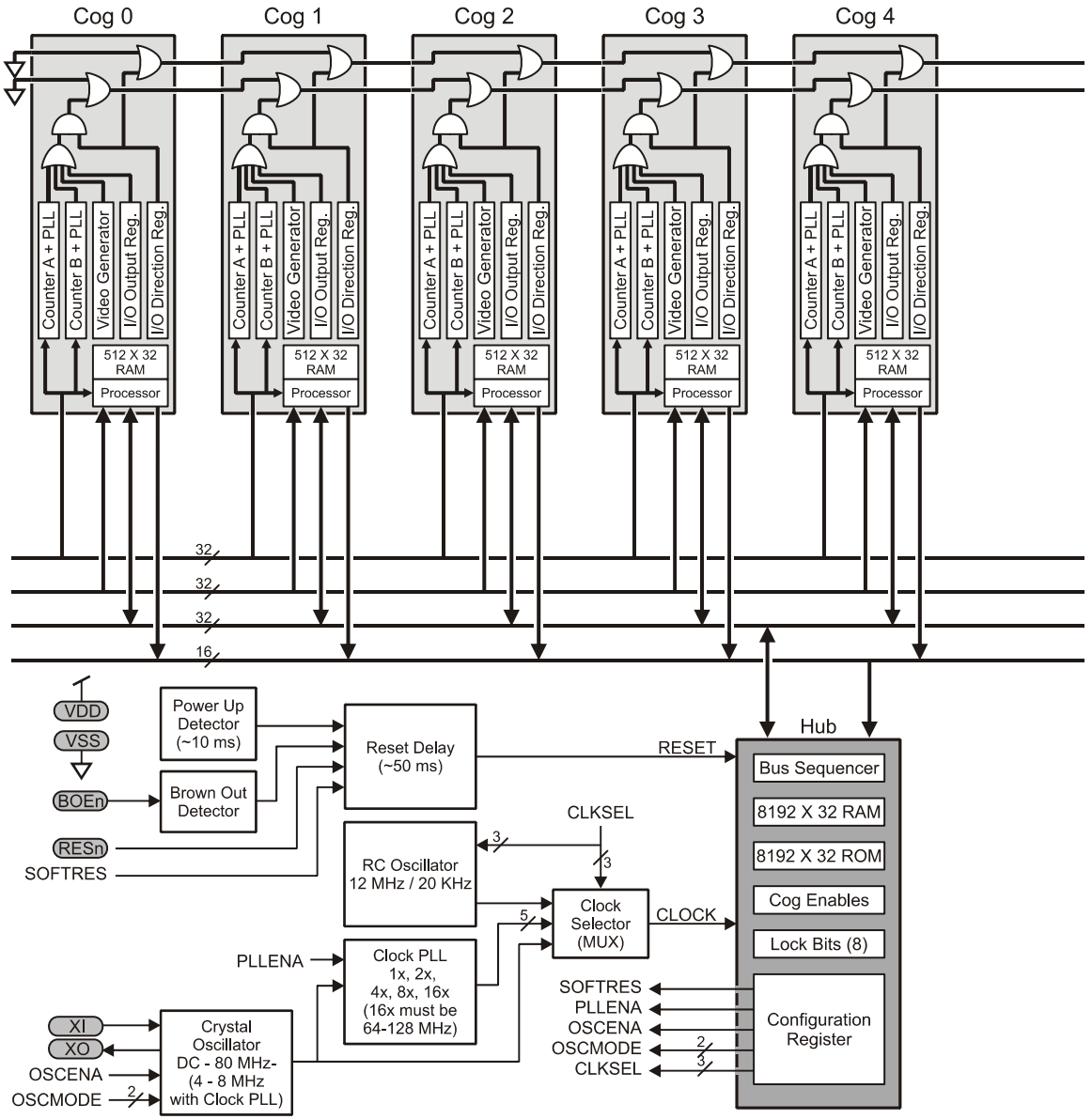
When the Propeller goes into shutdown mode, the internal clock is stopped causing all cogs to halt and all I/O pins are set to input direction (high impedance). Shutdown mode is triggered by one of the three following events:

1) VDD falling below the brown-out threshold (≈2.7 VDC), when the brown-out circuit is enabled,
2) the RESn pin going low, or
3) the application requesting a reboot (see the REBOOT command, page 187).

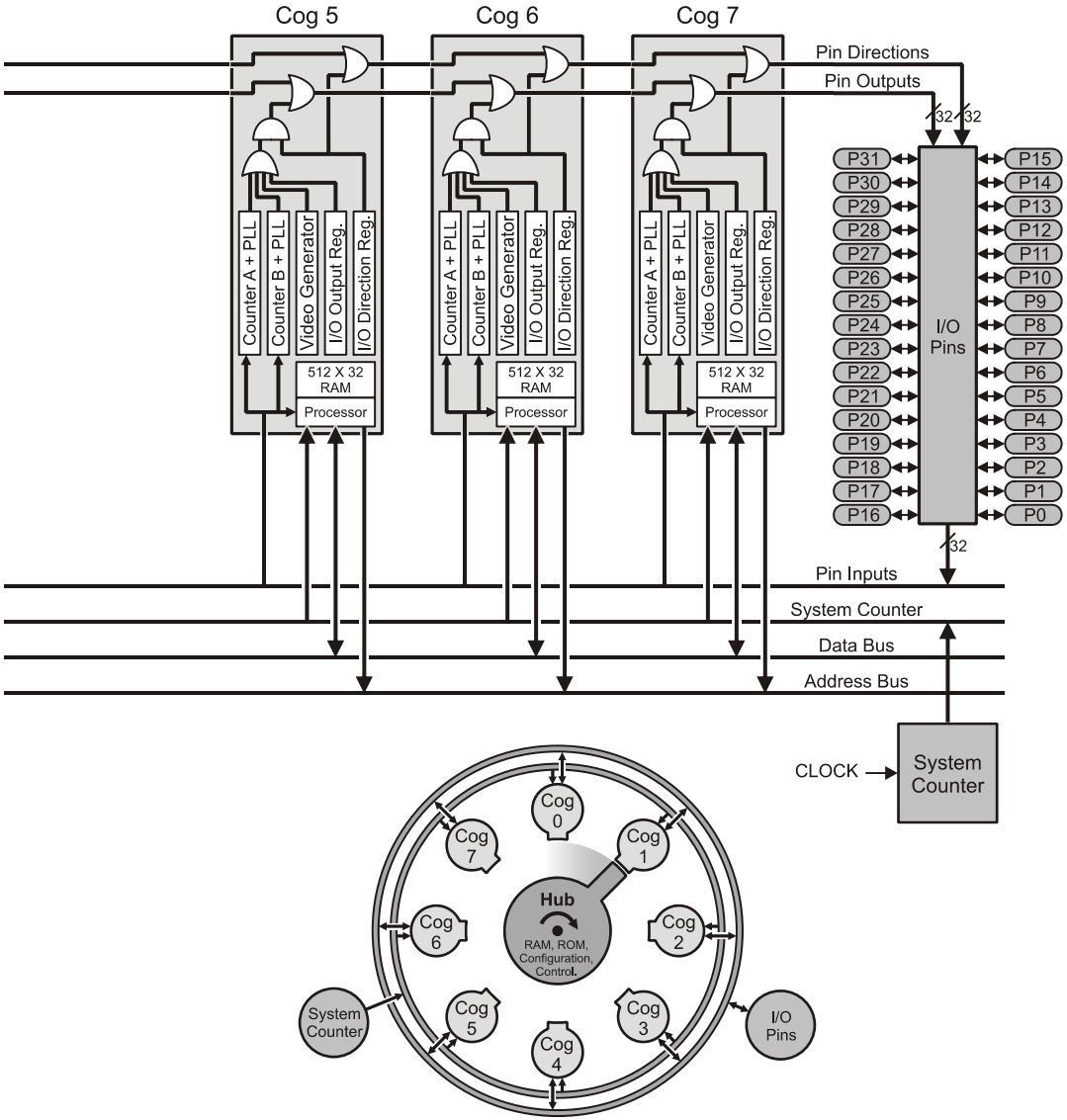
Shutdown mode is discontinued when the voltage level rises above the brown-out threshold and the RESn pin is high.

Block Diagram

Figure 1-2: Propeller Chip Block Diagram



Cog and Hub interaction is critical to the Propeller chip. The Hub controls which cog can access mutually exclusive resources, such as Main RAM/ROM, configuration registers, etc. The Hub gives exclusive access to every cog one at a time in a "round robin" fashion, regardless of how many cogs are running, in order to keep timing deterministic.



Hub and Cog Interaction

Shared Resources

There are two types of shared resources in the Propeller: 1) common, and 2) mutually exclusive. Common resources can be accessed at any time by any number of cogs. Mutually exclusive resources can also be accessed by all cogs, but only by one cog at a time. The common resources are the I/O pins and the System Counter. All other shared resources are mutually exclusive by nature and access to them is controlled by the Hub. See the Hub section on page 24.

System Clock

The System Clock (shown as "CLOCK" in Figure 1-2) is the central clock source for nearly every component of the Propeller chip. The System Clock's signal comes from one of three possible sources: 1) the Internal RC Oscillator, 2) the Clock Phase-Locked Loop (PLL), or 3) the Crystal Oscillator (an internal circuit that is fed by an external crystal or crystal/oscillator pack). The source is determined by the CLK register's settings, which is selectable at compile time or at run time. The only components that don't use the System Clock directly are the Hub and Bus; they divide the System Clock by two (2).

Cogs (processors)

The Propeller contains eight (8) processors, called cogs, numbered 0 to 7. Each cog contains the same components (see Figure 1-2): a Processor block, local 2 KB RAM configured as 512 longs (512 x 32 bits), two Counter Modules with PLLs, a Video Generator, I/O Output Register, I/O Direction Register, and other registers not shown in the diagram. See Table 1-3 for a complete list of cog registers. Each cog is designed exactly the same and can run tasks independently from the others.

All eight cogs are driven from the same clock source, the System Clock, so they each maintain the same time reference and all active cogs execute instructions simultaneously. See System Clock, above. They also all have access to the same shared resources, like I/O pins, Main RAM, and the System Counter. See Shared Resources, above.

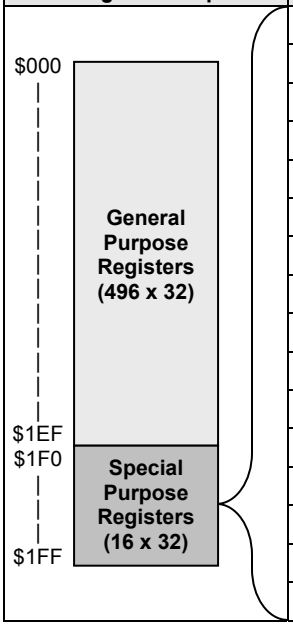
Cogs can be started and stopped at run time and can be programmed to perform tasks simultaneously, either independently or with coordination from other cogs through Main RAM. Regardless of the nature of their use, the Propeller application designer has full control over how and when each cog is employed; there is no compiler-driven or operating system-driven splitting of tasks between multiple cogs. This empowers the developer to deliver absolutely deterministic timing, power consumption, and response to the embedded application.

Each cog has its own RAM, called Cog RAM, which contains 512 registers of 32 bits each. The Cog RAM is all general purpose RAM except for the last 16 registers, which are special purpose registers, as described in Table 1-3. The Cog RAM is used for executable code, data, variables, and the last 16 locations serve as interfaces to the System Counter, I/O pins, and local cog peripherals.

When a cog is booted up, locations 0 ($000) through 495 ($1EF) are loaded sequentially from Main RAM / ROM and its special purpose locations, 496 ($1F0) through 511 ($1FF) are cleared to zero. After loading, the cog begins executing instructions, starting at location 0 of Cog RAM. It will continue to execute code until it is stopped or rebooted by either itself or another cog, or a reset occurs.

IMPROVED

Table 1-3: Cog RAM Special Purpose Registers

Cog RAM Map	Address	Name	Type	Description
	$1F0	PAR	Read-Only[1]	Boot Parameter, p. 178, 331
	$1F1	CNT	Read-Only[1]	System Counter, p. 73, 282
	$1F2	INA	Read-Only[1]	Input States for P31–P0, p. 118, 297
	$1F3	INB [3]	Read-Only[1]	Input States for P63–P32, p. 118, 297
	$1F4	OUTA	Read/Write	Output States for P3–P0, p. 175, 330
	$1F5	OUTB [3]	Read/Write	Output States for P63–P32, p. 175, 330
	$1F6	DIRA	Read/Write	Direction States for P31–P0, p. 104, 456
	$1F7	DIRB [3]	Read/Write	Direction States for P63–P32, p. 104, 456
	$1F8	CTRA	Read/Write	Counter A Control, p. 95, 288
	$1F9	CTRB	Read/Write	Counter B Control, p. 95, 288
	$1FA	FRQA	Read/Write	Counter A Frequency, p. 111, 293
	$1FB	FRQB	Read/Write	Counter B Frequency, p. 111, 293
	$1FC	PHSA	Read/Write[2]	Counter A Phase, p. 180, 332
	$1FD	PHSB	Read/Write[2]	Counter B Phase, p. 180, 332
	$1FE	VCFG	Read/Write	Video Configuration, p. 213, 366
	$1FF	VSCL	Read/Write	Video Scale, p. 216, 367

Note 1: For Propeller Assembly, only accessible as a source register (i.e., `mov` *dest*, *source*). See the Assembly language sections for PAR, page 331; CNT, page 282, and INA, INB, page 297.

Note 2: For Propeller Assembly, only readable as a source register (i.e., `mov` *dest, source*); read modify-write not possible as a destination register. See the Assembly language section for PHSA, PHSB on page 332.

Note 3: Reserved for future use.

Each Special Purpose Register may be accessed via:

1) its physical register address (Propeller Assembly),
2) its predefined name (Spin or Propeller Assembly), or
3) the register array variable (SPR) with an index of 0 to 15 (Spin).

The following are examples in Propeller Assembly:

```
MOV   $1F4, #$FF          'Set OUTA 7:0 high
MOV   OUTA, #$FF          'Same as above
```

The following are examples in Spin:

```
SPR[$4] := $FF            'Set OUTA 7:0 high
OUTA := $FF               'Same as above
```

Hub

To maintain system integrity, mutually exclusive resources must not be accessed by more than one cog at a time. The Hub maintains this integrity by controlling access to mutually exclusive resources, giving each cog a turn to access them in a "round robin" fashion from Cog 0 through Cog 7 and back to Cog 0 again. The Hub, and the bus it controls, runs at half the System Clock rate. This means that the Hub gives a cog access to mutually exclusive resources once every 16 System Clock cycles. Hub instructions, the Propeller Assembly instructions that access mutually exclusive resources, require 7 cycles to execute but they first need to be synchronized to the start of the hub access window. It takes up to 15 cycles (16 minus 1, if we just missed it) to synchronize to the hub access window plus 7 cycles to execute the hub instruction, so hub instructions take from 7 to 22 cycles to complete.

Figure 1-3 and Figure 1-4 show examples where Cog 0 has a hub instruction to execute. Figure 1-3 shows the best-case scenario; the hub instruction was ready right at the start of that cog's access window. The hub instruction executes immediately (7 cycles) leaving an additional 9 cycles for other instructions before the next hub access window arrives.

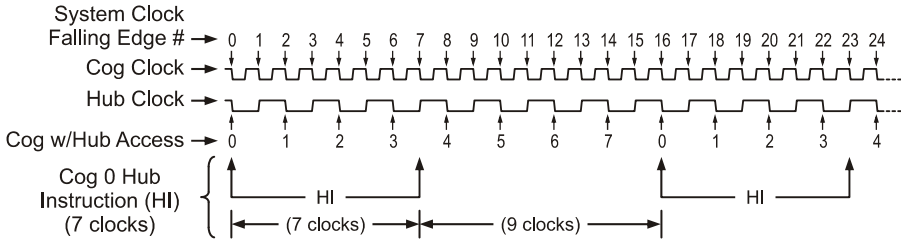


Figure 1-3: Cog-Hub Interaction – Best Case Scenario

Figure 1-4 shows the worst-case scenario; the hub instruction was ready on the cycle right after the start of Cog 0's access window; it just barely missed it. The cog waits until the next hub access window (15 cycles later) then the hub instruction executes (7 cycles) for a total of 22 cycles for that hub instruction. Again, there are 9 additional cycles after the hub instruction for other instructions to execute before the next hub access window arrives. To get the most efficiency out of Propeller Assembly routines that have to frequently access mutually exclusive resources, it can be beneficial to interleave non-hub instructions with hub instructions to lessen the number of cycles waiting for the next hub access window. Since most Propeller Assembly instructions take 4 clock cycles, two such instructions can be executed in between otherwise contiguous hub instructions.

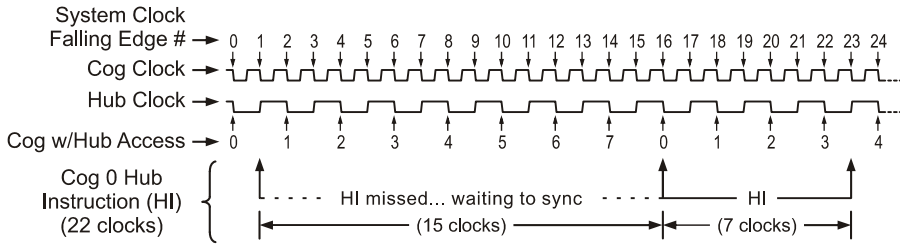


Figure 1-4: Cog-Hub Interaction – Worst Case Scenario

Keep in mind that a particular cog's hub instructions do not, in any way, interfere with other cogs' instructions because of the Hub mechanism. Cog 1, for example, may start a hub instruction during System Clock cycle 2, in both of these examples, possibly overlapping its execution with that of Cog 0 without any ill effects. Meanwhile, all other cogs can continue executing non-hub instructions, or awaiting their individual hub access windows regardless of what the others are doing.

I/O Pins

IMPROVED

The Propeller has 32 I/O pins, 28 of which are entirely general purpose. Four I/O pins (28 - 31) have a special purpose at Boot Up and are available for general purpose use afterwards; see the Boot Up Procedure section on page 18. After boot up, any I/O pins can be used by any cogs at any time since I/O pins are one of the common resources. It is up to the application developer to ensure that no two cogs try to use the same I/O pin for conflicting purposes during run-time.

For details of the I/O hardware, refer to the internals of the cogs in Figure 1-2 on page 20 while reading the following explanation.

Each cog has its own 32-bit I/O Direction Register and 32-bit I/O Output Register to influence the directions and output states of the Propeller chip's corresponding 32 I/O pins. A cog's desired I/O directions and output states is communicated through the entire cog collective to ultimately become what is called "Pin Directions" and "Pin Outputs" in the upper right corner of Figure 1-2 on page 20.

The cog collective determines Pin Directions and Pin Outputs as follows:

1. Pin Directions are the result of OR'ing the Direction Registers of the cogs together.
2. Pin Outputs are the result of OR'ing the output states of the cogs together. A cog's output state consists of the bits of its I/O modules (the Counters, the Video Generator, and the I/O Output Register) OR'd together then AND'd with the bits of its Direction Register.

In essence, each I/O pin's direction and output state is the "wired-OR" of the entire cog collective. This allows the cogs to access and influence the I/O pins simultaneously without the need for any resource arbiter and without any possibility of electrical contention between the cogs.

The result of this I/O pin wiring configuration can easily be described in the following simple rules:

A. A pin is an input only if no active cog sets it to an output.

B. A pin outputs low only if all active cogs that set it to output also set it to low.

C. A pin outputs high if any active cog sets it to an output and also sets it high.

Table 1-4 demonstrates a few possible combinations of the collective cogs' influence on a particular I/O pin, P12 in this example. For simplification, these examples assume that bit 12 of each cog's I/O hardware, other than its I/O Output Register, is cleared to zero (0).

Table 1-4: I/O Sharing Examples				
	Bit 12 of Cogs' I/O Direction Register	Bit 12 of Cogs' I/O Output Register	State of I/O Pin P12	Rule Followed
Cog ID	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7		
Example 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	Input	A
Example 2	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	Output Low	B
Example 3	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0	Output High	C
Example 4	1 0 0 0 0 0 0 0	0 1 0 0 0 0 0 0	Output Low	B
Example 5	1 1 0 0 0 0 0 0	0 1 0 0 0 0 0 0	Output High	C
Example 6	1 1 1 1 1 1 1 1	0 1 0 1 0 0 0 0	Output High	C
Example 7	1 1 1 1 1 1 1 1	0 0 0 1 0 0 0 0	Output High	C
Example 8	1 1 1 0 1 1 1 1	0 0 0 1 0 0 0 0	Output Low	B

Note: For the I/O Direction Register, a 1 in a bit location sets the corresponding I/O pin to the output direction while a 0 sets it to an input direction.

Any cog that is shut down has its Direction Register and output states cleared to zero, effectively removing it from influencing the final state of the I/O pins that the remaining active cogs are controlling.

Each cog also has its own 32-bit Input Register. This input register is really a pseudo-register; every time it is read, the actual states of the I/O pins are read, regardless of their input or output direction.

System Counter

The System Counter is a global, read-only, 32-bit counter that increments once every System Clock cycle. Cogs can read the System Counter (via their `CNT` register, page 73) to perform timing calculations and can use the `WAITCNT` command (page 218) to create effective delays within their processes. The System Counter is a common resource. Every cog can read it simultaneously. The System Counter is not cleared upon startup since its practical use is for differential timing. If a cog needs to keep track of time from a specific, fixed moment in time, it simply needs to read and save the initial counter value at that moment in time, and compare all of the later counter values against that initial value.

CLK Register

The CLK register is the System Clock configuration control; it determines the source of and the characteristics for the System Clock. More precisely, the CLK register configures the RC Oscillator, Clock PLL, Crystal Oscillator, and Clock Selector circuits. (See Figure 1-2: Propeller Chip Block Diagram on page 20.) It is configured at compile time by the _CLKMODE constant (page 68) and is writable at run time through the CLKSET Spin command (page 71) or the CLKSET assembly instruction (page 271). Whenever the CLK register is written, a global delay of ≈75 µs occurs as the clock source transitions.

Whenever this register is changed, a copy of the value written should be placed in the Clock Mode value location (which is BYTE[4] in Main RAM) and the resulting master clock frequency should be written to the Clock Frequency value location (which is LONG[0] in Main RAM) so that objects which reference this data will have current information for their timing calculations. (See CLKMODE, page 67, and CLKFREQ, page 63.) When possible, it is recommended to use Spin's CLKSET command (page 71), since it automatically updates all the above-mentioned locations with the proper information.

NEW

Only certain bit patterns in the CLK register are valid clock modes. See the _CLKMODE constant on page 68 and Table 2-4 on page 69 for more information. The Clock object in the Propeller Library may also be useful since it provides clock modification and timing methods.

Table 1-5: CLK Register Structure

Bit	7	6	5	4	3	2	1	0
Name	RESET	PLLENA	OSCENA	OSCM1	OSCM0	CLKSEL2	CLKSEL1	CLKSEL0

Table 1-6: CLK Register RESET (Bit 7)

Bit	Effect
0	Always write '0' here unless you intend to reset the chip.
1	Same as a hardware reset – reboots the chip. The Spin command REBOOT writes a '1' to the RESET bit.

Table 1-7: CLK Register PLLENA (Bit 6)	
Bit	**Effect**
0	Disables the PLL circuit. The RCFAST and RCSLOW settings of the _CLKMODE declaration configure PLLENA this way.
1	Enables the PLL circuit. Each of the PLLxx settings of the _CLKMODE declaration configures PLLENA this way at compile time. The Clock PLL internally multiplies the XIN pin frequency by 16. OSCENA must also be '1' to propagate the XIN signal to the Clock PLL. The Clock PLL's internal frequency must be kept within 64 MHz to 128 MHz – this translates to an XIN frequency range of 4 MHz to 8 MHz. Allow 100 µs for the Clock PLL to stabilize before switching to one of its outputs via the CLKSELx bits. Once the Crystal Oscillator and Clock PLL circuits are enabled and stabilized, you can switch freely among all clock sources by changing the CLKSELx bits.

Table 1-8: CLK Register OSCENA (Bit 5)	
Bit	**Effect**
0	Disables the Crystal Oscillator circuit. The RCFAST and RCSLOW settings of the _CLKMODE declaration configure OSCENA this way.
1	Enables the Crystal Oscillator circuit so that a clock signal can be input to XIN, or so that XIN and XOUT can function together as a feedback oscillator. The XINPUT and XTALx settings of the _CLKMODE declaration configure OSCENA this way. The OSCMx bits select the operating mode of the Crystal Oscillator circuit. Note that no external resistors or capacitors are required for crystals and resonators. Allow a crystal or resonator 10 ms to stabilize before switching to a Crystal Oscillator or Clock PLL output via the CLKSELx bits. When enabling the Crystal Oscillator circuit, the Clock PLL may be enabled at the same time so that they can share the stabilization period.

Table 1-9: CLK Register OSCMx (Bits 4:3)					
OSCMx		**_CLKMODE Setting**	**XOUT Resistance**	**XIN/XOUT Capacitance**	**Frequency Range**
1	**0**				
0	0	XINPUT	Infinite	6 pF (pad only)	DC to 80 MHz Input
0	1	XTAL1	2000 Ω	36 pF	4 to 16 MHz Crystal/Resonator
1	0	XTAL2	1000 Ω	26 pF	8 to 32 MHz Crystal/Resonator
1	1	XTAL3	500 Ω	16 pF	20 to 60 MHz Crystal/Resonator

IMPROVED

Table 1-10: CLK Register CLKSELx (Bits 2:0)						
CLKSELx			_CLKMODE Setting	Master Clock	Source	Notes
2	1	0				
0	0	0	RCFAST	~12 MHz	Internal	No external parts. May range from 8 MHz to 20 MHz.
0	0	1	RCSLOW	~20 kHz	Internal	Very low power. No external parts. May range from 13 kHz to 33 kHz.
0	1	0	XINPUT	XIN	OSC	OSCENA must be '1'.
0	1	1	XTALx and PLL1X	XIN x 1	OSC+PLL	OSCENA and PLLENA must be '1'.
1	0	0	XTALx and PLL2X	XIN x 2	OSC+PLL	OSCENA and PLLENA must be '1'.
1	0	1	XTALx and PLL4X	XIN x 4	OSC+PLL	OSCENA and PLLENA must be '1'.
1	1	0	XTALx and PLL8X	XIN x 8	OSC+PLL	OSCENA and PLLENA must be '1'.
1	1	1	XTALx and PLL16X	XIN x 16	OSC+PLL	OSCENA and PLLENA must be '1'.

Locks

There are eight lock bits (also known as semaphores) available to facilitate exclusive access to user-defined resources among multiple cogs. If a block of memory is to be used by two or more cogs at once and that block consists of more than one long (four bytes), the cogs will each have to perform multiple reads and writes to retrieve or update that memory block. This leads to the likely possibility of read/write contention on that memory block where one cog may be writing while another is reading, resulting in misreads and/or miswrites.

The locks are global bits accessed through the Hub via the hub instructions: LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR. Because locks are accessed only through the Hub, only one cog at a time can affect them, making this an effective control mechanism. The Hub maintains an inventory of which locks are in use and their current states, and cogs can check out, return, set, and clear locks as needed during run time. See LOCKNEW, 122; LOCKRET, 125; LOCKSET, 126; and LOCKCLR, 120 for more information.

Main Memory

The Main Memory is a block of 64 K bytes (16 K longs) that is accessible by all cogs as a mutually exclusive resource through the Hub. It consists of 32 KB of RAM and 32 KB of

ROM. The 32 KB of Main RAM is general purpose and is the destination of a Propeller Application either downloaded from a host or uploaded from the external 32 KB EEPROM. The 32 KB of Main ROM contains all the code and data resources vital to the Propeller chip's function: character definitions, log, anti-log and sine tables, and the Boot Loader and Spin Interpreter. The Main Memory organization is shown in Figure 1-5.

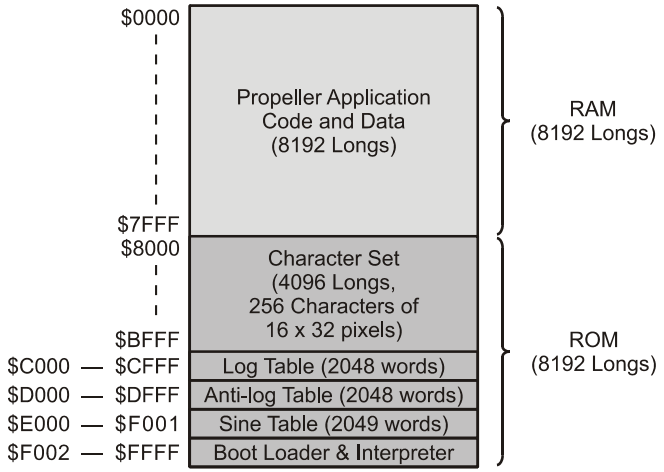


Figure 1-5: Main Memory Map

Main RAM

The first half of Main Memory is all RAM. This space is used for your program, data, variables and stack(s); otherwise known as your Propeller Application.

When a program is loaded into the chip, either from a host or from an external EEPROM, this entire memory space is written. The first 16 locations, $0000 – $000F, hold initialization data used by the Boot Loader and Interpreter. Your program's executable code and data will begin at $0010 and extend for some number of longs. The area after your executable code, extending to $7FFF, is used as variable and stack space.

There are two values stored in the initialization area that might be of interest to your program: a long at $0000 contains the initial master clock frequency, in Hertz, and a byte following it at $0004 contains the initial value written into the CLK register. These two values can be read/written using their physical addresses (`LONG[$0]` and `BYTE[$4]`) and can be read by using their predefined names (**`CLKFREQ`** and **`CLKMODE`**). If you change the CLK register without using the **`CLOCKSET`** command, you will also need to update these two locations so that objects which reference them will have current information.

Main ROM

The second half of Main Memory is all ROM. This space is used for character definitions, math functions, and the Boot Loader and Spin Interpreter.

Character Definitions

The first half of ROM is dedicated to a set of 256 character definitions. Each character definition is 16 pixels wide by 32 pixels tall. These character definitions can be used for video displays, graphical LCD's, printing, etc. The character set is based on a North American / Western European layout (Basic Latin and Latin-1 Supplement), with many specialized characters inserted. The special characters are connecting waveform and schematic building-blocks, Greek symbols commonly used in electronics, and several arrows and bullets.

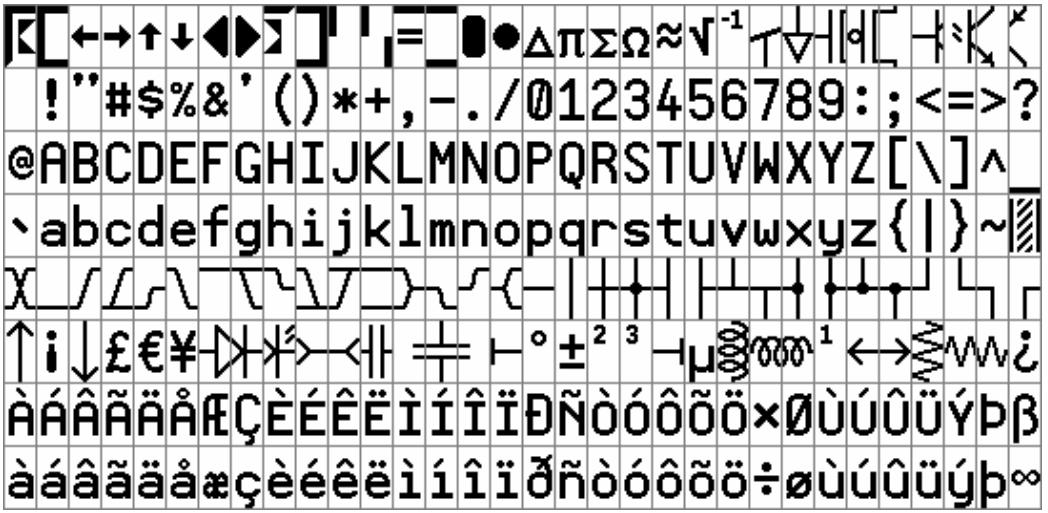


Figure 1-6: Propeller Font Characters

The character definitions are numbered 0 to 255 from left-to-right, top-to-bottom in Figure 1-6, above. In ROM, they are arranged with each pair of adjacent even-odd characters merged together to form 32 longs. The first character pair is located in bytes $8000-$807F. The second pair occupies bytes $8080-$80FF, and so on, until the last pair fills $BF80-$BFFF. The Propeller Tool includes an interactive character chart (Help → View Character Chart…) that has a ROM Bitmap view which shows where and how each character resides in ROM.

The character pairs are merged row-by-row such that each character's 16 horizontal pixels are spaced apart and interleaved with their neighbors' so that the even character takes bits 0, 2, 4, ...30, and the odd character takes bits 1, 3, 5, ...31. The leftmost pixels are in the lowest bits, while the rightmost pixels are in the highest bits, as shown in Figure 1-7. This forms a long (4 bytes) for each row of pixels in the character pair. 32 such longs, building from the character's top row down to the bottom, make up the complete merged-pair definition. The definitions are encoded in this manner so that a cog's video hardware can handle the merged longs directly, using color selection to display either the even or the odd character. It also has the advantage of allowing run-time character pairs (see next paragraph) that are four-color characters used to draw beveled buttons, lines and focus indicators.

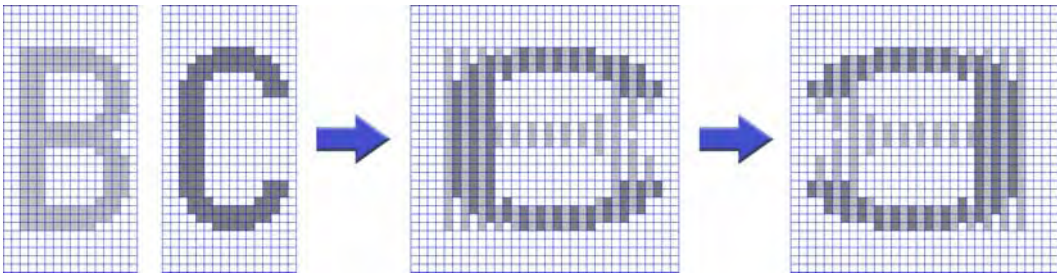


Figure 1-7: Propeller Character Interleaving

Some character codes have inescapable meanings, such as 9 for Tab, 10 for Line Feed, and 13 for Carriage Return. These character codes invoke actions and do not equate to static character definitions. For this reason, their character definitions have been used for special four-color characters. These four-color characters are used for drawing 3-D box edges at run time and are implemented as 16 x 16 pixel cells, as opposed to the normal 16 x 32 pixel cells. They occupy even-odd character pairs 0-1, 8-9, 10-11, and 12-13. Figure 1-8 shows an example of a button with 3D beveled edges made from some of these characters.

Figure 1-8: Button with 3-D Beveled Edges

The Propeller Tool includes, and uses, the Parallax True Type® font which follows the design of the Propeller Font embedded in the hardware. With this font, and the Propeller Tool, you

can include schematics, timing diagrams and other diagrams right in the source code for your application.

Log and Anti-Log Tables

The log and anti-log tables are useful for converting values between their number form and exponent form.

When numbers are encoded into exponent form, simple math operations take on more complex effects. For example ‘add’ and ‘subtract’ become ‘multiply’ and ‘divide.’ ‘Shift left’ becomes ‘square’ and ‘shift right’ becomes 'square-root.’ ‘Divide by 3’ will produce ‘cube root.’ Once the exponent is converted back to a number, the result will be apparent.

See Appendix B: Math Samples and Function Tables on page 380 for more information.

Sine Table

The sine table provides 2,049 unsigned 16-bit sine samples spanning from 0° to 90°, inclusively (0.0439° resolution). Sine values for all other quadrants covering > 90° to < 360° can be calculated from simple transformations on this single-quadrant sine table. The sine table can be used for calculations related to angular phenomena.

See Appendix B: Math Samples and Function Tables on page 380 for more information.

Boot Loader and Spin Interpreter

The last section in Main ROM contains the Propeller chip’s Boot Loader and Spin Interpreter programs.

The Boot Loader is responsible for initializing the Propeller upon power-up/reset. When a Boot Up procedure is started, the Boot Loader is loaded into Cog 0’s RAM and the cog executes the code starting at location 0. The Boot Loader program first checks the host and EEPROM communication pins for code/data to download/upload, processes that information accordingly and finally it either launches the Spin Interpreter program into Cog 0’s RAM (overwriting itself) to run the user’s Propeller Application, or it puts the Propeller into shutdown mode. See the Boot Up Procedure section on page 18.

The Spin Interpreter program fetches and executes the Propeller Application from Main RAM. This may lead to launching additional cogs to run more Spin code or Propeller Assembly code, as is requested by the application. See Run-Time Procedure, page 18.

Chapter 2: Spin Language Reference

This chapter describes all elements of the Propeller chip's Spin language and is best used as a reference for individual elements of the Spin language. For a tutorial on the use of the language refer to the Spin Language Tutorial in the Propeller Tool's on-line help, then return here for more details.

The Spin Language Reference is divided into three sections:

1) **The Structure of the Propeller Objects**. Propeller Objects consist of Spin code, optional Assembly Code, and data. An object's Spin code provides it with structure, consisting of special-purpose blocks. This section lists these blocks and the elements that may be used in each. Each listed element has a page reference for more information.

2) **The Categorical Listing of the Propeller Spin Language**. All elements, including operators and syntax symbols, are grouped by related function. This is a great way to quickly realize the breadth of the language and what features are available for specific uses. Each listed element has a page reference for more information. Some elements are marked with a superscript "a" indicating that they are also available in Propeller Assembly, though syntax may vary. Such marked elements are also included in Chapter 3: Assembly Language Reference.

3) **The Spin Language Elements**. Most elements have their own dedicated sub-section, alphabetically arranged to ease searching for them. Those individual elements without a dedicated sub-section, such as Operators, Symbols and some constants, are grouped within other related sub-sections but can be easily located by following their page reference from the Categorical Listing.

Structure of Propeller Objects/Spin

Each Propeller object is a Spin file that has an inherent structure consisting of up to six different special-purpose blocks: CON, VAR, OBJ, PUB, PRI, and DAT. These blocks are shown below (in the order that they typically appear in objects) along with the set of elements usable within each.

For detailed examples of the object (Spin) structure and usage, refer to the Propeller Programming Tutorial in the Propeller Tool Help.

CON: Constant blocks define global constants (page 84).

_CLKFREQ	p 65	NEGX	p 93	PLL16X	p 68	XINPUT	p 68
_CLKMODE	p 68	**Operators***	p 143	POSX	p 93	XTAL1	p 68
_FREE	p 110	PI	p 93	RCFAST	p 68	XTAL2	p 68
_STACK	p 202	PLL1X	p 68	RCSLOW	p 68	XTAL3	p 68
_XINFREQ	p 236	PLL2X	p 68	ROUND	p 198		
FALSE	p 93	PLL4X	p 68	TRUE	p 93		
FLOAT	p 108	PLL8X	p 68	TRUNC	p 209		

*Non-assignment operators only.

VAR: Variable blocks define global variables (page 210).

BYTE	p 51	LONG	p 128	ROUND	p 198	TRUNC	p 209
FLOAT	p 108	**Operators***	p 143	WORD	p 227		

*Non-assignment operators only.

OBJ: Object blocks define referenced objects (page 141).

FLOAT	p 108	**Operators***	p 143	ROUND	p 198	TRUNC	p 209

*Non-assignment operators only.

PUB/PRI: Public and Private method blocks define Spin routines (pages 182/181).

DAT: Data blocks define data and Propeller Assembly code (page 99).

*** Non-assignment operators only.**

Categorical Listing of Propeller Spin Language

Elements marked with a superscript "a" are also available in Propeller Assembly.

Block Designators

CON	Declare constant block; p 84.
VAR	Declare variable block; p 210.
OBJ	Declare object reference block; p 141.
PUB	Declare public method block; p 182.
PRI	Declare private method block; p 181.
DAT	Declare data block; p 99.

Configuration

CHIPVER	Propeller chip version number; p 62.
CLKMODE	Current clock mode setting; p 67.
_CLKMODE	Application-defined clock mode (read-only); p 68.
CLKFREQ	Current clock frequency; p 63.
_CLKFREQ	Application-defined clock frequency (read-only); p 65.
CLKSET[a]	Set clock mode and clock frequency; p 71.
_XINFREQ	Application-defined external clock frequency (read-only); p 236.
_STACK	Application-defined stack space to reserve (read-only); p 202.
_FREE	Application-defined free space to reserve (read-only); p 110.
RCFAST	Constant for _CLKMODE: internal fast oscillator; p 68.
RCSLOW	Constant for _CLKMODE: internal slow oscillator; p 68.
XINPUT	Constant for _CLKMODE: external clock/osc (XI pin); p 68.
XTAL1	Constant for _CLKMODE: external low-speed crystal; p 68.
XTAL2	Constant for _CLKMODE: external med-speed crystal; p 68.
XTAL3	Constant for _CLKMODE: external high-speed crystal; p 68.
PLL1X	Constant for _CLKMODE: external frequency times 1; p 68.
PLL2X	Constant for _CLKMODE: external frequency times 2; p 68.
PLL4X	Constant for _CLKMODE: external frequency times 4; p 68.

PLL8X	Constant for _CLKMODE: external frequency times 8; p 68.
PLL16X	Constant for _CLKMODE: external frequency times 16; p 68.

Cog Control

COGID[a]	Current cog's ID (0-7); p 75.
COGNEW	Start the next available cog; p 78.
COGINIT[a]	Start, or restart, a cog by ID; p 76.
COGSTOP[a]	Stop a cog by ID; p 83.
REBOOT	Reset the Propeller chip; p 187.

Process Control

LOCKNEW[a]	Check out a new lock; p 122.
LOCKRET[a]	Release a lock; p 125.
LOCKCLR[a]	Clear a lock by ID; p 120.
LOCKSET[a]	Set a lock by ID; p 126.
WAITCNT[a]	Wait for System Counter to reach a value; p 218.
WAITPEQ[a]	Wait for pin(s) to be equal to value; p 222.
WAITPNE[a]	Wait for pin(s) to be not equal to value; p 224.
WAITVID[a]	Wait for video sync and deliver next color/pixel group; p 225.

Flow Control

IF ...ELSEIF ...ELSEIFNOT ...ELSE	Conditionally execute one or more blocks of code; p 112.
IFNOT ...ELSEIF ...ELSEIFNOT ...ELSE	Conditionally execute one or more blocks of code; p 117.
CASE ...OTHER	Evaluate expression and execute block of code that satisfies a condition; p 59.

REPEAT ...FROM ...TO ...STEP ...UNTIL ...WHILE	Execute block of code repetitively an infinite or finite number of times with optional loop counter, intervals, exit and continue conditions; p 188.
NEXT	Skip rest of REPEAT block and jump to next loop iteration; p 140.
QUIT	Exit from REPEAT loop; p 186.
RETURN	Exit PUB/PRI with normal status and optional return value; p 196.
ABORT	Exit PUB/PRI with abort status and optional return value; p 47.

Memory

BYTE	Declare byte-sized symbol or access byte of main memory; p 51.
WORD	Declare word-sized symbol or access word of main memory; p 227.
LONG	Declare long-sized symbol or access long of main memory; p 128.
BYTEFILL	Fill bytes of main memory with a value; p 57.
WORDFILL	Fill words of main memory with a value; p 234.
LONGFILL	Fill longs of main memory with a value; p 134.
BYTEMOVE	Copy bytes from one region to another in main memory; p 58.
WORDMOVE	Copy words from one region to another in main memory; p 235.
LONGMOVE	Copy longs from one region to another in main memory; p 135.
LOOKUP	Get value at index (1..N) from a list; p 138.
LOOKUPZ	Get value at zero-based index (0..N−1) from a list; p 138.
LOOKDOWN	Get index (1..N) of a matching value from a list; p 136.
LOOKDOWNZ	Get zero-based index (0..N−1) of a matching value from a list; p 136.
STRSIZE	Get size of string in bytes; p 206.
STRCOMP	Compare a string of bytes against another string of bytes; p 203.

Directives

STRING	Declare in-line string expression; resolved at compile time; p 205.
CONSTANT	Declare in-line constant expression; resolved at compile time; p 91.
FLOAT	Declare floating-point expression; resolved at compile time; p 108.
ROUND	Round compile-time floating-point expression to integer; p 198.
TRUNC	Truncate compile-time floating-point expression at decimal; p 209.
FILE	Import data from an external file; p 107.

Registers

DIRA[a]	Direction Register for 32-bit port A; p 104.
DIRB[a]	Direction Register for 32-bit port B (future use); p 104.
INA[a]	Input Register for 32-bit port A (read only); p 117.
INB[a]	Input Register for 32-bit port B (read only) (future use); p 118.
OUTA[a]	Output Register for 32-bit port A; p 175.
OUTB[a]	Output Register for 32-bit port B (future use); p 177.
CNT[a]	32-bit System Counter Register (read only); p 73.
CTRA[a]	Counter A Control Register; p 95.
CTRB[a]	Counter B Control Register; p 95.
FRQA[a]	Counter A Frequency Register; p 111.
FRQB[a]	Counter B Frequency Register; p 111.
PHSA[a]	Counter A Phase-Locked Loop (PLL) Register; p 180.
PHSB[a]	Counter B Phase-Locked Loop (PLL) Register; p 180.
VCFG[a]	Video Configuration Register; p 213.
VSCL[a]	Video Scale Register; p 216.
PAR[a]	Cog Boot Parameter Register (read only); p 178.
SPR	Special-Purpose Register array; indirect cog register access; p 200.

Constants

TRUE[a]	Logical true: -1 ($FFFFFFFF); p 93.
FALSE[a]	Logical false: 0 ($00000000) ; p 93.
POSX[a]	Maximum positive integer: 2,147,483,647 ($7FFFFFFF); p 93.
NEGX[a]	Maximum negative integer: -2,147,483,648 ($80000000); p 93.
PI[a]	Floating-point value for PI: ~3.141593 ($40490FDB); p 93.

Variable

RESULT	Default result variable for PUB/PRI methods; p 194.

Unary Operators

+	Positive (+X); unary form of Add; p 150.
-	Negate (-X); unary form of Subtract; p 150.
--	Pre-decrement (--X) or post-decrement (X--) and assign; p 151.
++	Pre-increment (++X) or post-increment (X++) and assign; p 152.
^^	Square root; p 156.
||	Absolute Value; p 156.
~	Sign-extend from bit 7 (~X) or post-clear to 0 (X~); p 156.
~~	Sign-extend from bit 15 (~~X) or post-set to -1(X~~); p 157.
?	Random number forward (?X) or reverse (X?); p 159.
|<	Decode value (modulus of 32; 0-31) into single-high-bit long; p 160.
>|	Encode long into magnitude (0 - 32) as high-bit priority; p 160.
!	Bitwise: NOT; p 166.
NOT	Boolean: NOT (promotes non-0 to -1); p 168.
@	Symbol address; p 173.
@@	Object address plus symbol value; p 173.

Binary Operators

NOTE: All right-column operators are assignment operators.

=	--and--	=	Constant assignment (CON blocks); p 148.
:=	--and--	:=	Variable assignment (PUB/PRI blocks); p 149.
+	--or--	+=	Add; p 149.
-	--or--	-=	Subtract; p 150.
*	--or--	*=	Multiply and return lower 32 bits (signed); p 153.
**	--or--	**=	Multiply and return upper 32 bits (signed); p 153.
/	--or--	/=	Divide (signed); p 154.
//	--or--	//=	Modulus (signed); p 154.
#>	--or--	#>=	Limit minimum (signed); p 155.
<#	--or--	<#=	Limit maximum (signed); p 155.
~>	--or--	~>=	Shift arithmetic right; p 158.
<<	--or--	<<=	Bitwise: Shift left; p 161.
>>	--or--	>>=	Bitwise: Shift right; p 161.
<-	--or--	<-=	Bitwise: Rotate left; p 162.
->	--or--	->=	Bitwise: Rotate right; p 162.
><	--or--	>><=	Bitwise: Reverse; p 163.
&	--or--	&=	Bitwise: AND; p 164.
|	--or--	|=	Bitwise: OR; p 165.
^	--or--	^=	Bitwise: XOR; p 165.
AND	--or--	AND=	Boolean: AND (promotes non-0 to -1); p 167.
OR	--or--	OR=	Boolean: OR (promotes non-0 to -1); p 168.
==	--or--	===	Boolean: Is equal; p 169.
<>	--or--	<>=	Boolean: Is not equal; p 170.
<	--or--	<=	Boolean: Is less than (signed); p 170.
>	--or--	>=	Boolean: Is greater than (signed); p 171.
=<	--or--	=<=	Boolean: Is equal or less (signed); p 171.
=>	--or--	=>=	Boolean: Is equal or greater (signed); p 172.

Syntax Symbols

%	Binary number indicator, as in %1010; p 207.
%%	Quaternary number indicator, as in %%2130; p 207.
IMPROVED $	Hexadecimal indicator, as in $1AF or assembly 'here' indicator; p 207.
''	String designator ''Hello''; p 207.
_	Group delimiter in constant values, or underscore in symbols; p 207.
#	Object-Constant reference: obj#constant; p 207.
.	Object-Method reference: obj.method(param) or decimal point; p 207.
..	Range indicator, as in 0..7; p 207.
:	Return separator: PUB method : sym, or object assignment, etc.; p 207.
|	Local variable separator: PUB method | temp, str; p 208.
\	Abort trap, as in \method(parameters); p 208.
,	List delimiter, as in method(param1, param2, param3); p 208.
( )	Parameter list designators, as in method(parameters); p 208.
[ ]	Array index designators, as in INA[2]; p 208.
{ }	In-line/multi-line code comment designators; p 208.
{{ }}	In-line/multi-line document comment designators; p 208.
'	Code comment designator; p 208.
' '	Document comment designator; p 208.

Spin Language Elements

The remainder of this chapter describes the elements of the Spin Language, shown above, in alphabetical order. A few elements are explained within the context of others for clarity; use the page references from the categorical listing, above, to find those discussions. Many elements are available both in Spin and Propeller Assembly. Those elements are described in detail within this section, with references to them, and any differences, in the appropriate areas of Chapter 3: Assembly Language Reference beginning on page 238.

Symbol Rules

Symbols are case-insensitive, alphanumeric names either created by the compiler (reserved word) or by the code developer (user-defined word). They represent values (constants or variables) to make source code easier to understand and maintain. Symbols must fit the following rules:

1) Begins with a letter (a – z) or an underscore '_'.
2) Contains only letters, numbers, and underscores (a – z, 0 – 9, _); no spaces allowed.
3) Must be 30 characters or less. IMPROVED
4) Is unique to the object; not a reserved word (p. 379) or previous user-defined symbol.

Value Representations

Values can be entered in binary (base-2), quaternary (base-4), decimal (base-10), hexadecimal (base-16), or character formats. Numerical values can also use underscores, '_', as a group separator to clarify numbers. The following are examples of these formats.

Table 2-1: Value Representations

Base	Type of Value	Examples				
2	Binary	%1010	–or–	%11110000_10101100		
4	Quaternary	%%2130_3311	–or–	%%3311_2301_1012		
10	Decimal (integer)	1024	–or–	2_147_483_647	–or–	-25
10	Decimal (floating-point)	1e6	–or–	1.000_005	–or–	-0.70712
16	Hexadecimal	$1AF	–or–	$FFAF_126D_8755		
n/a	Character	"A"				

Separators can be used in place of commas (in decimal values) or to form logical groups, such as nibbles, bytes, words, etc.

Syntax Definitions

In addition to detailed descriptions, the following pages contain syntax definitions for many elements that describe, in short terms, all the options of that element. The syntax definitions use special symbols to indicate when and how certain element features are to be used.

BOLDCAPS	Items in bold uppercase should be typed in as shown.
Bold Italics	Items in bold italics should be replaced by user text; symbols, operators, expressions, etc.
. .. : , #	Periods, double-periods, colons, commas, pound signs,
| \ [ ] ( )	pipes, back slashes, square brackets and parentheses should be typed in where shown.
〈 〉	Angle bracket symbols enclose optional items. Enter the enclosed item if desired. Do not enter the angle brackets.
((¦))	Double parentheses symbols enclose mutually exclusive items, separated by a dash-bar. Enter one, and only one, of the encoded items. Do not enter the double parentheses or dash-bar.
…	Repetition symbol indicates that the previous item, or group, can be repeated numerous times. Repeat the last item(s) if desired. Do not enter the repetition symbol.
↳	New Line/Indent symbol indicates following items should appear on the next line, indented by at least one space.
→|	Indent symbol indicates following items should be intended by at least one space.
Single line	Separates various syntax structure options.
Double line	Separates instruction from the value it returns.

Since elements are limited to specific Spin blocks, all syntax definitions begin with an indication of the type of block required. For example, the following syntax indicates that the BYTEFILL command and its parameters must appear in either a PUB or PRI block, but it may be one of many commands within that block.

((PUB ¦ PRI))
  BYTEFILL (*StartAddress*, *Value*, *Count*)

ABORT

Command: Exit from PUB/PRI method using abort status with optional return *Value*.

((PUB ¦ PRI))
  ABORT 〈*Value*〉

Returns: Either the current RESULT value, or *Value* if provided.

- ***Value*** is an optional expression whose value is to be returned, with abort status, from the PUB or PRI method.

Explanation

ABORT is one of two commands (ABORT and RETURN) that terminate a PUB or PRI method's execution.

ABORT causes a return from a PUB or PRI method with abort status; meaning it pops the call stack repeatedly until either the call stack is empty or it reaches a caller with an Abort Trap, (\ ), and delivers a value in the process.

ABORT is useful for cases where a method needs to terminate and indicate an abnormal or elevated status to the immediate caller or one its previous callers. For example, an application may be involved in a complicated chain of events where any one of those events could lead to a different branch of the chain or a final action decision. It may be easier to write that application using small, specialized methods that are called in a nested fashion, each meant to deal with a specific sub-event in the chain. When one of the simple methods determines a course of action, it can issue an abort that completely collapses the nested call chain and prevents all the intermediate methods from continuing.

When ABORT appears without the optional *Value*, it returns the current value of the PUB/PRI's built-in RESULT variable. If the *Value* field was entered, however, the PUB or PRI aborts and returns that *Value* instead.

About the Call Stack

When methods are called simply by referring to them from other methods, there must be some mechanism in place to store where to return to once the called method is completed. This mechanism is a called a "stack" but we'll use the term "call stack" here. It is simply RAM memory used to store return addresses, return values, parameters and intermediate results. As more and more methods are called, the call stack logically gets longer. As more

and more methods are returned from (via RETURN or by reaching the end of the method) the call stack gets shorter. This is called "pushing" onto the stack and "popping" off of the stack, respectively.

The RETURN command pops the most recent data off the call stack to facilitate returning to the immediate caller; the one who directly called the method that just returned. The ABORT command, however, repetitively pops data off the call stack until it reaches a caller with an Abort Trap (see below); returning to some higher-level caller that may have just been one call, or many calls, up the nested chain of calls. Any return points along the way between an aborting method and an abort trapping method are ignored and essentially terminated. In this way, ABORT allows code to back way out of a very deep and potentially complicated series of logic to handle a serious issue at a high level.

Using ABORT

Any method can choose to issue an ABORT command. It's up to the higher-level code to check for an abort status and handle it. This higher-level code can be either that which called an aborting method directly, or via some other set of methods. To issue an ABORT command, use something like the following:

```
if <bad condition>
  abort                 'If bad condition detected, abort
```

—or—

```
if <bad condition>
  abort <value>         'If bad condition detected, abort with value
```

...where <bad condition> is a condition that determines the method should abort and <value> is a value to return upon aborting.

The Abort Trap (\ )

To trap an ABORT, the call to the method or method chain that could potentially abort must be preceded with the Abort Trap symbol, a backslash (\). For example, if a method named MayAbort could possibly abort, or if it calls other methods that may abort, a calling method could trap this with the following:

```
if \MayAbort            'Call MayAbort with abort trap
  abort <value>         'Process abort
```

The type of exit that MayAbort actually used, ABORT or RETURN, is not automatically known by the trapping call; it may have just happened to be the destination of a RETURN command. Therefore, the code must be written in a way to detect which type was used. Some possibilities are: 1) code may be designed such that a high-level method is the only place that traps an abort and other mid-level code processes things normally without allowing RETURNs to propagate higher, or 2) aborting methods may return a special value that can not occur in any normal circumstance, or 3) a global flag can be set by the aborting method prior to aborting.

Example Use Of Abort

The following is an example of a simple-minded robot application in which the robot is designed to move away from an object it senses with its four sensors (Left, Right, Front and Back). Assume that CheckSensors, Beep, and MotorStuck are methods defined elsewhere.

```
CON
  #0, None, Left, Right, Front, Back  'Direction Enumerations

PUB Main | Direction
  Direction := None
  repeat
    case CheckSensors                 'Get active sensor
      Left  : Direction := Right      'Object on left? Let's go right
      Right : Direction := Left       'Object on right? Let's go left
      Front : Direction := Back       'Object in front? Let's go back
      Back  : Direction := Front      'Object in back? Let's go front
      other : Direction := None       'Otherwise, stay still
    if not \Move(Direction)           'Move robot
      Beep                            'We're stuck?  Beep

PUB Move(Direction)
  result := TRUE                      'Assume success
  if Direction == None
    return                            'Return if no direction
  repeat 1000
    DriveMotors(Direction)            'Drive motor 1000 times

PUB DriveMotors(Direction)
  <code to drive motors>
  if MotorStuck
    abort FALSE                       'If motor is stuck, abort
```

```
<more code>
```

The above example shows three methods of various logical levels, Main ("high-level"), Move ("mid-level") and DriveMotors ("low-level"). The high-level method, Main, is the decision maker of the application; deciding how to respond to events like sensor activations and motor movements. The mid-level method, Move, is responsible for moving the robot a short distance. The low-level method, DriveMotors, handles the details of driving the motors properly and verifying that it is successful.

In an application like this, critical events could occur in low-level code that needs to be addressed by high-level code. The ABORT command can be instrumental in getting the message to the high-level code without requiring complicated message-passing code for all the mid-level code in-between. In this case, we have only one mid-level method but there could be many nested mid-level methods between the high-level and the low-level.

The Main method gets sensor inputs and decides what direction to move the robot via the CASE statement. It then calls Move in a special way, with the Abort Trap symbol, \ , preceding it. The Move method sets its RESULT to TRUE and then calls DriveMotors in a finite loop. If it successfully completes, Move returns TRUE. The DriveMotors method handles the complication of moving the robot's motors to achieve the desired direction, but if it determines the motors are stuck, it cannot move them further and it aborts with a FALSE value. Otherwise it simply returns normally.

If everything is fine, the DriveMotors method returns normally, the Move method carries on normally and eventually returns TRUE, and the Main method continues on normally. If, however, DriveMotors finds a problem, it ABORTs which causes the Propeller to pop the call stack all the way through the Move method and up to the Main method where the Abort Trap was found. The Move method is completely oblivious to this and is now effectively terminated. The Main method checks the value returned by its call to Move (which is now the FALSE value that was actually returned by the aborted DriveMotors method deep down the call stack) and it decides to Beep as a result of the detected failure.

If we had not put the Abort Trap, (\ ), in front of the call to Move, when DriveMotors aborted, the call stack would have been popped until it was empty and this application would have terminated immediately.

BYTE

Designator: Declare byte-sized symbol, byte aligned/sized data, or read/write a byte of main memory.

```
VAR
  BYTE Symbol {[Count]}
```

```
DAT
  {Symbol} BYTE Data {[Count]}
```

IMPROVED

```
((PUB | PRI))
  BYTE [BaseAddress] {[Offset]}
```

```
((PUB | PRI))
  Symbol.BYTE {[Offset]}
```

- ***Symbol*** is the desired name for the variable (Syntax 1) or data block (Syntax 2) or is the existing name of the variable (Syntax 4).
- ***Count*** is an optional expression indicating the number of byte-sized elements for *Symbol* (Syntax 1), or the number of byte-sized entries of *Data* (Syntax 2) to store in a data table.
- ***Data*** is a constant expression or comma-separated list of constant expressions. Quoted strings of characters are also allowed; they are treated as a comma-separated list of characters.
- ***BaseAddress*** is an expression describing the address of main memory to read or write. If *Offset* is omitted, *BaseAddress* is the actual address to operate on. If *Offset* is specified, *BaseAddress* + *Offset* is the actual address to operate on.
- ***Offset*** is an optional expression indicating the offset from *BaseAddress* to operate on, or the offset from byte 0 of *Symbol*.

Explanation

BYTE is one of three multi-purpose declarations (BYTE, WORD, and LONG) that declare or operate on memory. BYTE can be used to:

1) declare a byte-sized (8-bit) symbol or a multi-byte symbolic array in a VAR block, or
2) declare byte-aligned, and possibly byte-sized, data in a DAT block, or
3) read or write a byte of main memory at a base address with an optional offset, or
4) access a byte within a word-sized or long-sized variable.

NEW

Range of Byte

Memory that is byte-sized (8 bits) can contain a value that is one of 2^8 possible combinations of bits (i.e., one of 256 combinations). This gives byte-sized values a range of 0 to 255. Since the Spin language performs all mathematic operations using 32-bit signed math, any byte-sized values will be internally treated as positive long-sized values. However, the actual numeric value contained within a byte is subject to how a computer and user interpret it. For example, you may choose to use the Sign-Extend 7 operator (~), page 156, in a Spin expression to convert a byte value that you interpret as "signed" (-128 to +127) to a signed long value.

Byte Variable Declaration (Syntax 1)

In VAR blocks, syntax 1 of BYTE is used to declare global, symbolic variables that are either byte-sized, or are any array of bytes.

For example:

```
VAR
  byte  Temp                    'Temp is a byte
  byte  Str[25]                 'Str is a byte array
```

The above example declares two variables (symbols), Temp and Str. Temp is simply a single, byte-sized variable. The line under the Temp declaration uses the optional *Count* field to create an array of 25 byte-sized variable elements called Str. Both Temp and Str can be accessed from any PUB or PRI method within the same object that this VAR block was declared; they are global to the object. An example of this is below.

```
PUB SomeMethod
  Temp := 250                   'Set Temp to 250
  Str[0] := "A"                 'Set first element of Str to "A"
  Str[1] := "B"                 'Set second element of Str to "B"
  Str[24] := "C"                'Set last element of Str to "C"
```

For more information about using BYTE in this way, refer to the VAR section's Variable Declarations (Syntax 1) on page 210, and keep in mind that BYTE is used for the *Size* field in that description.

Byte Data Declaration (Syntax 2)

In DAT blocks, syntax 2 of BYTE is used to declare byte-aligned, and/or byte-sized data that is compiled as constant values in main memory. DAT blocks allow this declaration to have an

optional symbol preceding it, which can be used for later reference (See DAT, page 99). For example:

```
DAT
  MyData      byte  64, $AA, 55      'Byte-aligned and byte-sized data
  MyString    byte  "Hello",0        'A string of bytes (characters)
```

The above example declares two data symbols, MyData and MyString. Each data symbol points to the start of byte-aligned and byte-sized data in main memory. MyData's values, in main memory, are 64, $AA and 55, respectively. MyString's values, in main memory, are "H", "e", "l", "l", "o", and 0, respectively. This data is compiled into the object and resulting application as part of the executable code section and may be accessed using the read/write form, syntax 3, of BYTE (see below). For more information about using BYTE in this way, refer to the DAT section's Declaring Data(Syntax 1) on page 100, and keep in mind that BYTE is used for the *Size* field in that description.

NEW

Data items may be repeated by using the optional *Count* field. For example:

```
DAT
  MyData    byte 64. $AA[8], 55
```

The above example declares a byte-aligned, byte-sized data table, called MyData, consisting of the following ten values: 64, $AA, $AA, $AA, $AA, $AA, $AA, $AA, $AA, 55. There were eight occurrences of $AA due to the [8] in the declaration immediately after it.

IMPROVED

Reading/Writing Bytes of Main Memory (Syntax 3)

In PUB and PRI blocks, syntax 3 of BYTE is used to read or write byte-sized values of main memory. This is done by writing expressions that refer to main memory using the form: byte[*BaseAddress*][*Offset*]. Here's an example.

```
PUB MemTest | Temp
  Temp := byte[@MyData][1]           'Read byte value
  byte[@MyStr][0] := "M"             'Write byte value

DAT
  MyData   byte  64, $AA, 55         'Byte-sized/aligned data
  MyStr    byte  "Hello", 0          'A string of bytes (characters)
```

In this example, the DAT block (bottom of code) places its data in memory as shown in Figure 2-1. The first data element of MyData is placed at memory address $18. The last data element of MyData is placed at memory address $1A, with the first element of MyStr immediately

following it at $1B. Note that the starting address ($18) is arbitrary and is likely to change as the code is modified or the object itself is included in another application.

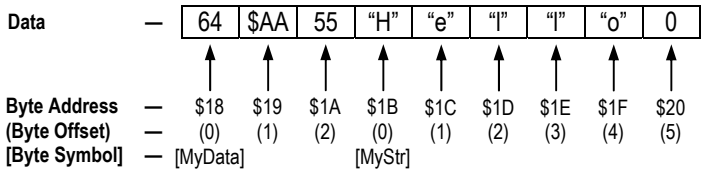


Figure 2-1: Main Memory Byte-Sized Data Structure and Addressing

Near the top of the code, the first executable line of the `MemTest` method, `Temp := byte[@MyData][1]`, reads a byte-sized value from main memory. It sets local variable `Temp` to $AA; the value read from main memory address $19. The address $19 was determined by the address of the symbol `MyData` ($18) plus byte offset 1. The following progressive simplification demonstrates this.

```
byte[@MyData][1] ➡ byte[$18][1] ➡ byte[$18 + 1] ➡ byte[$19]
```

The next line, `byte[@MyStr][0] := "M"`, writes a byte-sized value to main memory. It sets the value at main memory address $1B to the character "M." The address $1B was calculated from the address of the symbol `MyStr` ($1B) plus byte offset 0.

```
byte[@MyStr][0] ➡ byte[$1B][0] ➡ byte[$1B + 0] ➡ byte[$1B]
```

NEW

Addressing Main Memory

As Figure 2-1 shows, main memory is really just a set of contiguous bytes and the addresses are calculated in terms of bytes. This concept is a consistent theme for any commands that use addresses.

Main memory is ultimately addressed in terms of bytes regardless of the size of value you are accessing; byte, word, or long. This is advantageous when thinking about how bytes, words, and longs relate to each other, but it may prove problematic when thinking of multiple items of a single size, like words or longs. See the Syntax 3 discussions for **WORD** (page 229) and **LONG** (page 130) for examples of accessing words and longs.

For more explanation of how data is arranged in memory, see the **DAT** section's Declaring Data(Syntax 1) on page 100.

NEW

An Alternative Memory Reference

There is yet another way to access the data from the code example above; you could reference the data symbols directly. For example, this statement reads the first byte of the MyData list:

```
Temp := MyData[0]
```

And these statements read the second and third bytes of MyData:

```
Temp := MyData[1]
Temp := MyData[2]
```

NEW

Other Addressing Phenomena

Both the BYTE and direct symbol reference techniques demonstrated above can be used to access any location in main memory, regardless of how it relates to defined data. Here are some examples:

```
Temp := byte[@MyStr][-1]    'Read last byte of MyData (before MyStr)
Temp := byte[@MyData][3]    'Read first byte of MyStr (after MyData)
Temp := MyStr[-3]           'Read first byte of MyData
Temp := MyData[-2]          'Read byte that is two bytes before MyData
```

These examples read beyond the logical borders (start point or end point) of the lists of data they reference. This may be a useful trick, but more often it's done by mistake; be careful when addressing memory, especially if you're writing to that memory.

Accessing Bytes of Larger-Sized Symbols (Syntax 4)

In PUB and PRI blocks, syntax 4 of BYTE is used to read or write byte-sized components of word-sized or long-sized variables. For example:

```
VAR
  word  WordVar
  long  LongVar

PUB Main
  WordVar.byte := 0           'Set first byte of WordVar to 0
  WordVar.byte[0] := 0        'Same as above
  WordVar.byte[1] := 100      'Set second byte of WordVar to 100
  LongVar.byte := 25          'Set first byte of LongVar to 25
  LongVar.byte[0] := 25       'Same as above
  LongVar.byte[1] := 50       'Set second byte of LongVar to 50
```

```
  LongVar.byte[2] := 75       'Set third byte of LongVar to 75
  LongVar.byte[3] := 100      'Set fourth byte of LongVar to 100
```

This example accesses the byte-sized components of both WordVar and LongVar, individually. The comments indicate what each line is doing. At the end of the Main method, WordVar will equal 25,600 and LongVar will equal 1,682,649,625.

NEW

The same techniques can be used to reference byte-sized components of word-sized or long-sized data symbols.

```
PUB Main | Temp
  Temp := MyData.byte[0]              'Read low byte of MyData word 0
  Temp := MyData.byte[1]              'Read high byte of MyData word 0
  MyList.byte[3] := $12               'Write high byte of MyList long 0
  MyList.byte[4] := $FF               'Write low byte of MyList long 1

DAT
  MyData  word  $ABCD, $9988          'Word-sized/aligned data
  MyList  long  $FF998877, $EEEE      'Long-sized/aligned data
```

The first and second executable lines of Main read the values $CD and $AB, respectively, from MyData. The third line writes $12 to the high byte of the long in element 0 of MyList, resulting in a value of $12998877. The fourth line writes $FF to the byte at index 4 in MyList (the low byte of the long in element 1), resulting in a value of $EEFF.

BYTEFILL

Command: Fill bytes of main memory with a value.

((PUB ¦ PRI))
BYTEFILL (*StartAddress*, *Value*, *Count*)

- ***StartAddress*** is an expression indicating the location of the first byte of memory to fill with *Value*.
- ***Value*** is an expression indicating the value to fill bytes with.
- ***Count*** is an expression indicating the number of bytes to fill, starting with *StartAddress*.

Explanation

BYTEFILL is one of three commands (BYTEFILL, WORDFILL, and LONGFILL) used to fill blocks of main memory with a specific value. BYTEFILL fills *Count* bytes of main memory with *Value*, starting at location *StartAddress*.

Using BYTEFILL

BYTEFILL is a great way to clear large blocks of byte-sized memory. For example:

```
VAR
  byte   Buff[100]

PUB Main
  bytefill(@Buff, 0, 100)     'Clear Buff to 0
```

The first line of the Main method, above, clears the entire 100-byte Buff array to all zeros. BYTEFILL is faster at this task than a dedicated REPEAT loop is.

BYTEMOVE

Command: Copy bytes from one region to another in main memory.

((PUB ¦ PRI))
BYTEMOVE (*DestAddress*, *SrcAddress*, *Count*)

- ***DestAddress*** is an expression specifying the main memory location to copy the first byte of source to.
- ***SrcAddress*** is an expression specifying the main memory location of the first byte of source to copy.
- ***Count*** is an expression indicating the number of bytes of the source to copy to the destination.

Explanation

BYTEMOVE is one of three commands (BYTEMOVE, WORDMOVE, and LONGMOVE) used to copy blocks of main memory from one area to another. BYTEMOVE copies *Count* bytes of main memory starting from *SrcAddress* to main memory starting at *DestAddress*.

Using BYTEMOVE

BYTEMOVE is a great way to copy large blocks of byte-sized memory. For example:

```
VAR
  byte   Buff1[100]
  byte   Buff2[100]

PUB Main
  bytemove(@Buff2, @Buff1, 100)     'Copy Buff1 to Buff2
```

The first line of the Main method, above, copies the entire 100-byte Buff1 array to the Buff2 array. BYTEMOVE is faster at this task than a dedicated REPEAT loop.

CASE

Command: Compare expression against matching expression(s) and execute code block if match found.

((PUB ¦ PRI))
  CASE ***CaseExpression***
    →¦ ***MatchExpression*** :
      →¦ ***Statement(s)***
  〈→¦ ***MatchExpression*** :
      →¦ ***Statement(s)*** 〉
  〈→¦ OTHER :
      →¦ ***Statement(s)*** 〉

- ***CaseExpression*** is the expression to compare.
- ***MatchExpression*** is a singular or comma-delimited set of value- and/or range-expressions, to compare *CaseExpression* against. Each *MatchExpression* must be followed by a colon (:).
- ***Statement(s)*** is a block of one or more lines of code to execute when the *CaseExpression* matches the associated *MatchExpression*. The first, or only, statement in *Statement(s)* may appear to the right of the colon on the *MatchExpression* line, or below it and slightly indented from the *MatchExpression* itself.

Explanation

CASE is one of the three conditional commands (IF, IFNOT, and CASE) that conditionally executes a block of code. CASE is the preferred structure to use, as opposed to IF..ELSEIF..ELSE, when you need to compare the equality of *CaseExpression* to a number of different values.

CASE compares *CaseExpression* against the values of each *MatchExpression*, in order, and if a match is found, executes the associated *Statement(s)*. If no previous matches were found, the *Statement(s)* associated with the optional OTHER command are executed.

Indention is Critical

IMPORTANT: Indention is critical. The Spin language relies on indention (of one space or more) on lines following conditional commands to determine if they belong to that command or not. To have the Propeller Tool indicate these logically grouped blocks of code on-screen,

you can press Ctrl + I to turn on block-group indicators. Pressing Ctrl + I again will disable that feature. See the Propeller Tool Help for a complete list of shortcut keys.

Using CASE

CASE is handy where one of many actions needs to be performed depending on the value of an expression. The following example assumes A, X and Y are variables defined earlier.

```
case X+Y                        'Test X+Y
  10, 15: !outa[0]              'X+Y = 10 or 15? Toggle P0
  A*2   : !outa[1]              'X+Y = A*2? Toggle P1
  30..40: !outa[2]              'X+Y in 30 to 40? Toggle P2
X += 5                          'Add 5 to X
```

Since the *MatchExpression* lines are indented from the CASE line, they belong to the CASE structure and are executed based on the *CaseExpression* comparison results. The next line, X += 5, is not indented from CASE, so it is executed regardless of the CASE results.

This example compares the value of X + Y against 10 or 15, A*2 and the range 30 through 40. If X + Y equals 10 or 15, P0 is toggled. If X + Y equals A*2, P1 is toggled. If X + Y is in the range 30 through 40, inclusive, then P2 is toggled. Whether or not any match was found, the X += 5 line is executed next.

Using OTHER

The optional OTHER component of CASE is similar to the optional ELSE component of an IF structure. For example:

```
case X+Y                        'Test X+Y
  10, 15: !outa[0]              'X+Y = 10 or 15? Toggle P0
  25    : !outa[1]              'X+Y = 25? Toggle P1
  20..30: !outa[2]              'X+Y in 20 to 30? Toggle P2
  OTHER : !outa[3]              'Otherwise toggle P3
X += 5                          'Add 5 to X
```

This example is similar to the last one except that the third *MatchStatement* checks for the range 20 to 30 and there's an OTHER component. If X + Y does not equal 10, 15, 25, or is not in the range 20 to 30, the *Statement(s)* block following OTHER is executed. Following that, the X += 5 line is executed.

There is an important concept to note about this example. If X + Y is 10 or 15, P0 is toggled, or if X + Y is 25, P1 is toggled, or if X + Y is 20 to 30, P2 is toggled, etc. This is because the *MatchExpressions* are checked, one at a time, in the order they are listed and only the first

expression that is a match has its block of code executed; no further expressions are tested after that. This means that if we had rearranged the 25 and 20..30 lines, so that the range of 20..30 is checked first, we'd have a bug in our code. We did this below:

```
case X+Y                'Test X+Y
  10, 15: !outa[0]      'X+Y = 10 or 15? Toggle P0
  20..30: !outa[2]      'X+Y in 20 to 30? Toggle P2
  25    : !outa[1]      'X+Y = 25? Toggle P1  <-- THIS NEVER RUNS
```

The above example contains an error because, while `X + Y` could be equal to 25, that match expression would never be tested since the previous one, 20..30 would be tested first, and since it is true, its block is executed and no further match expressions are checked.

Variations of Statement(s)

The above examples only use one line per *Statement(s)* block, but each block can be many lines of course. Additionally, the *Statement(s)* block may also appear below, and slightly indented from, the *MatchExpression* itself. The following two examples show these variations.

```
case A                  'Test A
  4     : !outa[0]      'A = 4? Toggle P0
  Z+1   : !outa[1]      'A = Z+1? Toggle P1
          !outa[2]      'And toggle P2
  10..15: !outa[3]      'A in 10 to 15? Toggle P3


case A                  'Test A
  4:                    'A = 4?
    !outa[0]            'Toggle P0
  Z+1:                  'A = Z+1?
    !outa[1]            'Toggle P1
    !outa[2]            'And toggle P2
  10..15:               'A in 10 to 15?
    !outa[3]            'Toggle P3
```

CHIPVER

Command: Get the Propeller chip's version number.

((PUB ¦ PRI))
  CHIPVER

Returns: Version number of the Propeller chip.

Explanation

The CHIPVER command reads and returns the version number of the Propeller chip. For example:

```
V := chipver
```

This example sets V to the version number of the Propeller chip, 1 in this case. Future Propeller Applications can use this to determine the version and type of Propeller chip they are running on and make modifications to their operation as necessary.

CLKFREQ

Command: Current System Clock frequency; the frequency at which each cog is running.

((PUB ¦ PRI))
  CLKFREQ

Returns: Current System Clock frequency, in Hz.

Explanation

The value returned by CLKFREQ is the actual System Clock frequency as determined by the current clock mode (oscillator type, gain, and PLL settings) and the external XI pin frequency, if any. Objects use CLKFREQ to determine the proper time delays for time-sensitive operations. For example:

```
waitcnt(clkfreq / 10 + cnt)   'wait for .1 seconds (100 ms)
```

This statement divides CLKFREQ by 10 and adds the result to CNT (the current System Counter value) then waits (WAITCNT) until the System Counter reaches the result value. Since CLKFREQ is the number of cycles per second, a divide by 10 yields the number of clock cycles per 0.1 seconds, or 100 ms. So, disregarding the time it takes to process the expression, this statement pauses the cog's program execution for 100 ms. The table below shows more examples of System Clock tick verses Time calculations.

Table 2-2: System Clock Ticks vs. Time Calculations

Expression	Result
clkfreq / 10	Clock ticks per 0.1 seconds (100 ms)
clkfreq / 100	Clock ticks per 0.01 seconds (10 ms)
clkfreq / 1_000	Clock ticks per 0.001 seconds (1 ms)
clkfreq / 10_000	Clock ticks per 0.0001 seconds (100 µs)
clkfreq / 100_000	Clock ticks per 0.00001 seconds (10 µs)
clkfreq / 9600	Clock ticks per serial bit period at 9,600 baud (~ 104 µs)
clkfreq / 19200	Clock ticks per serial bit period at 19,200 baud (~ 52 µs)

The value that CLKFREQ returns is actually read from long 0 (the first location in RAM) and that value can change whenever the application changes the clock mode, either manually or via the CLKSET command. Objects that are time-sensitive should check CLKFREQ at strategic points in order to adjust to new settings automatically.

CLKFREQ vs. _CLKFREQ

CLKFREQ is related to, but not the same as, _CLKFREQ. CLKFREQ is command that returns the current System Clock frequency whereas _CLKFREQ is an application-defined constant that contains the application's System Clock frequency at startup. In other words, CLKFREQ is the current clock frequency and _CLKFREQ is the original clock frequency; they both may happen to be the same value but they certainly can be different.

_CLKFREQ

Constant: Pre-defined, one-time settable constant for specifying the System Clock frequency.

```
CON
  _CLKFREQ = Expression
```

- ***Expression*** is an integer expression that indicates the System Clock frequency upon application start-up.

Explanation

_CLKFREQ specifies the System Clock frequency for start-up. It is a pre-defined constant symbol whose value is determined by the top object file of an application. _CLKFREQ is either set directly by the application itself, or is set indirectly as the result of the _CLKMODE and _XINFREQ settings.

The top object file in an application (the one where compilation starts from) can specify a setting for _CLKFREQ in its CON block. This defines the initial System Clock frequency for the application and is the frequency that the System Clock will switch to as soon as the application is booted up and execution begins.

The application can specify either _CLKFREQ or _XINFREQ in the CON block; they are mutually exclusive and the non-specified one is automatically calculated and set as a result of specifying the other.

The following examples assume that they are contained within the top object file. Any _CLKFREQ settings in child objects are simply ignored by the compiler.

For example:

```
CON
  _CLKMODE = XTAL1 + PLL8X
  _CLKFREQ = 32_000_000
```

The first declaration in the above CON block sets the clock mode for an external low-speed crystal and a Clock PLL multiplier of 8. The second declaration sets the System Clock frequency to 32 MHz, which means the external crystal's frequency must be 4 MHz because 4 MHz * 8 = 32 MHz. The _XINFREQ value is automatically set to 4 MHz because of these declarations.

```
CON
  _CLKMODE = XTAL2
  _CLKFREQ = 10_000_000
```

These two declarations set the clock mode for an external medium-speed crystal, no Clock PLL multiplier, and a System Clock frequency of 10 MHz. The **_XINFREQ** value is automatically set to 10 MHz, as well, because of these declarations.

_CLKFREQ vs CLKFREQ

_CLKFREQ is related to, but not the same as, **CLKFREQ**. **_CLKFREQ** contains the application's System Clock frequency at startup whereas **CLKFREQ** is a command that returns the current System Clock frequency. In other words, **_CLKFREQ** is the original System Clock frequency and **CLKFREQ** is the current System Clock frequency; they both may happen to be the same value but they certainly can be different.

CLKMODE

Command: Current clock mode setting.

((PUB ¦ PRI))
  CLKMODE

Returns: Current clock mode.

Explanation

The clock mode setting is the byte-sized value, determined by the application at compile time, from the CLK register. See CLK Register, page 28, for explanation of the possible settings. For example:

```
Mode := clkmode
```

This statement can be used to set a variable, Mode, to the current clock mode setting. Many applications maintain a static clock mode setting; however, some applications will change the clock mode setting during run time for clock speed adjustments, low-power modes, etc. It may be necessary for some objects to pay attention to the potential for dynamic clock modes in order to maintain proper timing and functionality.

CLKMODE vs._CLKMODE

CLKMODE is related to, but not the same as, _CLKMODE. CLKMODE is a command that returns the current clock mode (in the form of the CLK register's bit pattern) whereas _CLKMODE is an application-defined constant containing the requested clock mode at startup (in the form of clock setting constants that are OR'd together). Both may describe the same logical clock mode but their values are not equivalent.

_CLKMODE

Constant: Pre-defined, one-time settable constant for specifying application-level clock mode settings.

```
CON
  _CLKMODE = Expression
```

- ***Expression*** is an integer expression made up of one or two Clock Mode Setting Constants shown in Table 2-3. This will be the clock mode upon application start-up.

Explanation

_CLKMODE is used to specify the desired nature of the System Clock. It is a pre-defined constant symbol whose value is determined by the top object file of an application. The clock mode setting is a byte whose value is described by a combination of the RCxxxx, XINPUT, XTALx and PLLxx constants at compile time. Table 2-3 illustrates the clock mode setting constants. Note that not every combination is valid; Table 2-4 shows all valid combinations.

Table 2-3: Clock Mode Setting Constants

Clock Mode Setting Constant[1]	XO Resistance[2]	XI/XO Capacitance[2]	Description
RCFAST	Infinite	n/a	Internal fast oscillator (~12 MHz). May be 8 MHz to 20 MHz. (Default)
RCSLOW	Infinite	n/a	Internal slow oscillator (~20 KHz). May be 13 KHz to 33 KHz.
XINPUT	Infinite	6 pF (pad only)	External clock-oscillator (DC to 80 MHz); XI pin only, XO disconnected
XTAL1	2 kΩ	36 pF	External low-speed crystal (4 MHz to 16 MHz)
XTAL2	1 kΩ	26 pF	External medium-speed crystal (8 MHz to 32 MHz)
XTAL3	500 Ω	16 pF	External high-speed crystal (20 MHz to 80 MHz)
PLL1X	n/a	n/a	Multiply external frequency times 1
PLL2X	n/a	n/a	Multiply external frequency times 2
PLL4X	n/a	n/a	Multiply external frequency times 4
PLL8X	n/a	n/a	Multiply external frequency times 8
PLL16X	n/a	n/a	Multiply external frequency times 16

1. All constants are also available in Propeller Assembly.
2. All necessary resistors/capacitors are included in Propeller chip.

Table 2-4: Valid Clock Mode Expressions and CLK Register Values			
Valid Expression	**CLK Register Value**	**Valid Expression**	**CLK Register Value**
RCFAST	0_0_0_00_000	XTAL1 + PLL1X	0_1_1_01_011
RCSLOW	0_0_0_00_001	XTAL1 + PLL2X	0_1_1_01_100
XINPUT	0_0_1_00_010	XTAL1 + PLL4X	0_1_1_01_101
		XTAL1 + PLL8X	0_1_1_01_110
		XTAL1 + PLL16X	0_1_1_01_111
XTAL1	0_0_1_01_010	XTAL2 + PLL1X	0_1_1_10_011
XTAL2	0_0_1_10_010	XTAL2 + PLL2X	0_1_1_10_100
XTAL3	0_0_1_11_010	XTAL2 + PLL4X	0_1_1_10_101
		XTAL2 + PLL8X	0_1_1_10_110
		XTAL2 + PLL16X	0_1_1_10_111
XINPUT + PLL1X	0_1_1_00_011	XTAL3 + PLL1X	0_1_1_11_011
XINPUT + PLL2X	0_1_1_00_100	XTAL3 + PLL2X	0_1_1_11_100
XINPUT + PLL4X	0_1_1_00_101	XTAL3 + PLL4X	0_1_1_11_101
XINPUT + PLL8X	0_1_1_00_110	XTAL3 + PLL8X	0_1_1_11_110
XINPUT + PLL16X	0_1_1_00_111	XTAL3 + PLL16X	0_1_1_11_111

IMPROVED

The top object file in an application (the one where compilation starts from) can specify a setting for **_CLKMODE** in its **CON** block. This defines the initial clock mode setting for the application and is the mode that the System Clock will switch to as soon as the application is booted up and execution begins. The following examples assume that they are contained within the top object file. Any **_CLKMODE** settings in child objects are simply ignored by the compiler. For example:

```
CON
  _CLKMODE = RCFAST
```

This sets the clock mode for the internal, fast RC Clock/Oscillator circuit. The System Clock would run at approximately 12 MHz with this setting. The **RCFAST** setting is the default setting, so if no **_CLKMODE** was actually defined, this is the setting that would be used. Note that the Clock PLL can not be used with the internal RC Clock/Oscillator. Here's an example with an external clock:

```
CON
  _CLKMODE = XTAL1 + PLL8X
```

This sets the clock mode for an external low-speed crystal (**XTAL1**), enables the Clock PLL circuit and sets the System Clock to use the 8x tap from the Clock PLL (**PLL8X**). If an external 4 MHz crystal was attached to XI and XO, for example, its signal would be

multiplied by 16 (the Clock PLL always multiplies by 16) but the 8x bit result would be used; the System Clock would be 4 MHz * 8 = 32 MHz.

```
CON
  _CLKMODE = XINPUT + PLL2X
```

This sets the clock mode for an external clock-oscillator, connected to XI only, and enables the Clock PLL circuit and sets the System Clock to use the 2x result. If an external clock-oscillator pack of 8 MHz was attached to XI, the System clock would run at 16 MHz; that's 8 MHz * 2.

Note that the Clock PLL is not required and can be disabled by simply not specifying any multiplier setting, for example:

```
CON
  _CLKMODE = XTAL1
```

This sets the clock mode for an external low-speed crystal but leaves the Clock PLL disabled; the System Clock will be equal to the external crystal's frequency.

The _CLKFREQ and _XINFREQ Settings

For simplicity, the examples above only show **_CLKMODE** settings, but either a **_CLKFREQ** or **_XINFREQ** setting is required to follow it so that objects can determine their actual System Clock's frequency. The following is the full version of the second example, above, with an external crystal frequency (**_XINFREQ**) of 4 MHz.

```
CON
  _CLKMODE = XTAL1 + PLL8X     'low-speed crystal x 8
  _XINFREQ = 4_000_000         'external crystal of 4 MHz
```

This example is exactly like the second example above, but **_XINFREQ** indicates that the frequency of the external crystal is 4 MHz. The Propeller chip uses this value along with the **_CLKMODE** setting to determine the System Clock frequency (as reported by the **CLKFREQ** command) so that objects can properly adjust their timing. See **_XINFREQ**, page 236.

_CLKMODE vs.CLKMODE

_CLKMODE is related to, but not the same as, **CLKMODE**. **_CLKMODE** is an application-defined constant containing the requested clock mode at startup (in the form of clock setting constants that are OR'd together) whereas **CLKMODE** is a command that returns the current clock mode (in the form of the CLK register's bit pattern). Both may describe the same logical clock mode but their values are not equivalent.

CLKSET

Command: Set both the clock mode and System Clock frequency at run time.

((PUB ¦ PRI))
CLKSET (*Mode*, *Frequency*)

- ***Mode*** is an integer expression that will be written to the CLK register to change the clock mode.
- ***Frequency*** is an integer expression that indicates the resulting System Clock frequency.

Explanation

One of the most powerful features of the Propeller chip is the ability to change the clock behavior at run time. An application can choose to toggle back and forth between a slow clock speed (for low-power consumption) and a fast clock speed (for high-bandwidth operations), for example. CLKSET is used to change the clock mode and frequency during run time. It is the run-time equivalent of the _CLKMODE and _CLKFREQ constants defined by the application at compile time. For example:

```
clkset(%01101100, 4_000_000)            'Set to XTAL1 + PLL2x
```

IMPROVED

This sets the clock mode to a low-speed external crystal and a Clock PLL multiplier of 2, and indicates the resulting System Clock frequency (CLKFREQ) is 4 MHz. After executing this command, the CLKMODE and CLKFREQ commands will report the updated settings for objects that use them.

In general, it is safe to switch between clock modes by using a single CLKSET command, however, if enabling the Crystal Oscillator circuit (CLK register's OSCENA bit) it is important to perform the clock mode switch as a three-stage process:

1) First set the CLK register's PLLENA, OSCENA, OSCM1 and OSCM0 bits as necessary. See CLK Register on page 28 for more information.
2) Wait for 10 ms to give the external crystal time to stabilize.
3) Set the CLK register's CLKSELx bits as necessary to switch the System Clock to the new source.

The above process is only necessary when switching the Crystal Oscillator circuit on. No other clock mode changes require this process if the Crystal Oscillator circuit is left in its

current state, either off or on. See the Clock object in the Propeller Library for clock modification and timing methods.

NOTE: It takes approximately 75 µs for the Propeller Chip to perform the clock source switching action.

CNT

Register: System Counter register.

((PUB ¦ PRI))
  CNT

Returns: Current 32-bit System Counter value.

Explanation

The CNT register contains the current value in the global 32-bit System Counter. The System Counter serves as the central time reference for all cogs; it increments its 32-bit value once every System Clock cycle.

Upon power-up/reset, the System Counter starts with an arbitrary value and counts upwards from there, incrementing with every System Clock cycle. Since the System Counter is a read-only resource, every cog can read it simultaneously and can use the returned value to synchronize events, count cycles and measure time.

IMPROVED

Using CNT

Read CNT to get the current System Counter value. The actual value itself does not matter for any particular purpose, but the difference in successive reads is very important. Most often, the CNT register is used to delay execution for a specific period or to synchronize an event to the start of a window of time. The next examples use the WAITCNT instruction to achieve this.

In Spin code, when using CNT inside of a WAITCNT command as shown above, make sure to write the expression in the form "offset + cnt" as opposed to "cnt + offset" and make sure offset is at least 381 to account for Spin Interpreter overhead and avoid unexpectedly long delays. See the WAITCNT command's Fixed Delays section on page 218 for more information.

```
waitcnt(3_000_000 + cnt)    'Wait for 3 million clock cycles
```

The above code is an example of a "fixed delay" It delays the cog's execution for 3 million system clock cycles (about ¼ second when running with the internal fast oscillator).

The next is an example of a "synchronized delay." It notes the current count at one place and performs an action (toggles a pin) every millisecond thereafter with accuracy as good as that of the oscillator driving the Propeller chip.

```
PUB Toggle | TimeBase, OneMS
  dira[0]~~                         'Set P0 to output
  OneMS := clkfreq / 1000           'Calculate cycles per 1 millisecond
  TimeBase := cnt                   'Get current count
  repeat                            'Loop endlessly
    waitcnt(TimeBase += OneMS)      '  Wait to start of next millisecond
    !outa[0]                        '  Toggle P0
```

Here, I/O pin 0 is set to output. Then the local variable `OneMS` is set equal to the current System Clock frequency divided by 1000; i.e., the number of System Clock cycles per 1 millisecond of time. Next, the local variable `TimeBase` is set to the current System Counter value. Finally, the last two lines of code repeat endlessly; each time waiting until the start of the next millisecond and then toggling the state of P0.

For more information, see the **`WAITCNT`** section's Fixed Delays on page 218 and Synchronized Delays on page 219.

NEW

The CNT register is read-only so in Spin it should not be assigned a value (i.e., should not be to the left of a `:=` or other assignment operator) and when used in Propeller Assembly it should only be accessed as a source (s-field) value (i.e., **`mov`** *dest*, *source*).

COGID

Command: Current cog's ID number (0-7).

((PUB ¦ PRI))
  COGID

Returns: The current cog's ID (0-7).

Explanation

The value returned by **COGID** is the ID of the cog that executed the command. Normally, the actual cog that code is running in does not matter, however, for some objects it might be important to keep track of it. For example:

```
PUB StopMyself
  'Stop cog this code is running in
  cogstop(cogid)
```

This example method, StopMyself, has one line of code that simply calls **COGSTOP** with **COGID** as the parameter. Since **COGID** returns the ID of the cog running that code, this routine causes the cog to terminate itself.

COGINIT

Command: Start or restart a cog by ID to run Spin code or Propeller Assembly code.

((PUB ¦ PRI))
COGINIT (*CogID*, *SpinMethod* 〈(*ParameterList*)〉, *StackPointer*)

((PUB ¦ PRI))
COGINIT (*CogID*, *AsmAddress*, *Parameter*)

- ***CogID*** is the ID (0 – 7) of the cog to start, or restart. A *CogID* above 7 results in the next available cog being started (if possible).
- ***SpinMethod*** is the PUB or PRI Spin method that the affected cog should run. Optionally, it can be followed by a parameter list enclosed in parentheses.
- ***ParameterList*** is an optional, comma-delimited list of one or more parameters for *SpinMethod*. It must be included only if *SpinMethod* requires parameters.
- ***StackPointer*** is a pointer to memory, such as a long array, reserved for stack space for the affected cog. The affected cog uses this space to store temporary data during further calls and expression evaluations. If insufficient space is allocated, either the application will fail to run or it will run with strange results.
- ***AsmAddress*** is the address of a Propeller Assembly routine from a DAT block.
- ***Parameter*** is used to optionally pass a value to the new cog. This value ends up in the affected cog's read-only Cog Boot Parameter (PAR) register. *Parameter* can be used to pass a either a single 14-bit value or the address of a block of memory to be used by the assembly routine. *Parameter* is required by COGINIT, but if not needed for your routine simply set it to an innocuous value like zero (0).

Explanation

COGINIT works exactly like COGNEW (page 78) with two exceptions: 1) it launches code into a specific cog whose ID is *CogID*, and 2) it does not return a value. Since COGINIT operates on a specific cog, as directed by the *CogID* parameter, it can be used to stop and restart an active cog in one step. This includes the current cog; i.e., a cog can use COGINIT to stop and restart itself to run, perhaps, completely different code.

NEW Note that every cog which runs Spin code must have its own stack space; they cannot share the same stack space.

NEW In addition, care must be taken when relaunching a cog with Spin code and specifying stack space that is currently in use by that cog. The Propeller always builds a cog's initial stack

frame in the specified stack space before launching the cog. Relaunching a cog with a COGINIT command specifying the same stack space that it is currently using will likely cause the new stack frame image to be clobbered before the cog is restarted. To prevent this from happening, first perform a COGSTOP on that cog, followed by the desired COGINIT.

Spin Code (Syntax 1)

To run a Spin method in a specific cog, the COGINIT command needs the cog ID, the method name, its parameters, and a pointer to some stack space. For example:

```
coginit(1, Square(@X), @SqStack)  'Launch Square in Cog 1
```

This example launches the Square method into Cog 1, passing the address of X into Square and the address of SqStack as COGINIT's stack pointer. See COGNEW, page 78, for more information.

Propeller Assembly Code (Syntax 2)

To run Propeller Assembly code in a specific cog, the COGINIT command needs the cog ID, the address of the assembly routine, and a value that can optionally be used by the assembly routine. For example:

IMPROVED

```
coginit(2, @Toggle, 0)
```

This example launches the Propeller Assembly routine, Toggle, into Cog 2 with a PAR parameter of 0. See the example in the assembly syntax description of COGNEW, page 78, for a more detailed example keeping in mind that the above COGINIT command can be used in place of COGNEW in that example.

NEW

The *Parameter* Field

It's important to note that the *Parameter* field is intended to pass a long address, so only 14 bits (bits 2 through 15) are passed into the cog's PAR register; the lower two bits are always cleared to zero to ensure it's a long-aligned address. A value other than a long address can still be passed via the *Parameter* field, but will have to be limited to 14 bits, must be shifted left by two bits (for the COGNEW/COGINIT command), and will have to be shifted right by two bits by the assembly program that receives it.

COGNEW

Command: Start the next available cog to run Spin code or Propeller Assembly code.

((PUB ¦ PRI))
COGNEW (*SpinMethod* 〈(*ParameterList*)〉, *StackPointer*)

((PUB ¦ PRI))
COGNEW (*AsmAddress*, *Parameter*)

Returns: The ID of the newly started cog (0-7) if successful, or -1 otherwise.

- ***SpinMethod*** is the PUB or PRI Spin method that the new cog should run. Optionally, it can be followed by a parameter list enclosed in parentheses.
- ***ParameterList*** is an optional, comma-delimited list of one or more parameters for *SpinMethod*. It must be included only if *SpinMethod* requires parameters.
- ***StackPointer*** is a pointer to memory, such as a long array, reserved for stack space for the new cog. The new cog uses this space to store temporary data during further calls and expression evaluations. If insufficient space is allocated, either the application will fail to run or it will run with strange results.
- ***AsmAddress*** is the address of a Propeller Assembly routine, usually from a DAT block.
- ***Parameter*** is used to optionally pass a value to the new cog. This value ends up in the new cog's read-only Cog Boot Parameter (PAR) register. *Parameter* can be used to pass a either a single 14-bit value or the address of a block of memory to be used by the assembly routine. *Parameter* is required by COGNEW, but if not needed for your routine, simply set it to an innocuous value like zero (0).

Explanation

IMPROVED

COGNEW starts a new cog and runs either a Spin method or a Propeller Assembly routine within it. If successful, COGNEW returns the ID of the newly started cog. If there were no more cogs available, COGNEW returns -1. COGNEW works exactly like COGINIT (page 76) with two exceptions: 1) it launches code into the next available cog, and 2) it returns the ID of the cog that it started, if any.

Spin Code (Syntax 1)

To run a Spin method in another cog, the **COGNEW** command needs the method name, its parameters, and a pointer to some stack space. For example:

```
VAR
  long SqStack[6]               'Stack space for Square cog


PUB Main | X
  X := 2                        'Initialize X
  cognew(Square(@X), @SqStack)  'Launch square cog
  <check X here>                'Loop here and check X


PUB Square(XAddr)
  'Square the value at XAddr
  repeat                        'Repeat the following endlessly
    long[XAddr] *= long[XAddr]  '  Square value, store back
    waitcnt(2_000_000 + cnt)    '  Wait 2 million cycles
```

This example shows two methods, `Main` and `Square`. `Main` starts another cog that runs `Square` endlessly, then `Main` can monitor the results in the `X` variable. `Square`, being run by another cog, takes the value of `XAddr`, squares it and stores the result back into `XAddr`, then waits for 2 million cycles before it does it again. More explanation follows, but the result is that `X` starts out as 2, and the second cog, running `Square`, iteratively sets `X` to 4, 16, 256, 65536 and then finally to 0 (it overflowed 32 bits), all independent of the first cog which may be checking the value of `X` or performing some other task.

The `Main` method declares a local variable, `X`, that is set to 2 in its first line. Then `Main` starts a new cog, with **COGNEW**, to run the `Square` method in a separate cog. **COGNEW**'s first parameter, `Square(@X)`, is the Spin method to run and its required parameter; in this case we pass it the address of the `X` variable. The second parameter of **COGNEW**, `@SqStack`, is the address of stack space reserved for the new cog. When a cog is started to run Spin code, it needs some stack space where it can store temporary data. This example only requires 6 longs of stack space for proper operation (see The Need for Stack Space, below, for more information).

After the **COGNEW** command is executed, two cogs are running; the first is still running the `Main` method and the second is starting to run the `Square` method. Despite the fact that they are using code from the same Spin object, they are running independently. The "`<check X here>`" line can be replaced with code that uses the value of `X` in some way.

NEW

The Need for Stack Space

A cog executing Spin code, unlike one executing Propeller Assembly code, needs some temporary workspace, called "stack space," to hold operational data such as call stacks, parameters and intermediate expression results. Without this, sophisticated features such as method calls, result values, and complex expressions would not be possible without severe limitations.

The Spin compiler automatically provides stack space for the Propeller Application's initial code, the top-level Spin code of the application. The "free space" following the application's memory image is used for this purpose. However, the compiler is not able to provide distinct blocks of stack space for Spin code that the application may launch on its own, therefore, the application must provide that stack space itself.

Typically, this stack space is provided in the form of a declared global variable meant only for that use, such as the `SqStack` variable in the example above. Unfortunately, it's difficult to determine just how much stack space should be provided, so when developing an object, it is suggested to initially provide a large amount of longs of memory (like 128 longs or more) and once the object is deemed complete, use an object like the Stack Length object in the Propeller Library to determine the optimal length. See the Stack Length object for further explanation.

NEW

Spin Code Can Only be Launched by its Containing Object

In the Spin language, by design, objects must intelligently manage their own data, the methods that operate on that data, the cogs that execute those methods, and the interface that other objects use to affect it. These are all aspects that serve to maintain the integrity of the object and increase its useful and reliable nature.

For these reasons, the object and its designer are notably the best equipped to provide the proper stack space that is required for Spin code being launched into another cog.

To enforce this principle, the **COGNEW** and **COGINIT** commands cannot launch Spin code outside of its containing object. This means that a statement like the following will not work as expected.

```
cognew(SomeObject.Method, @StackSpace)
```

Instead of launching `SomeObject.Method` into another cog, the Propeller will instead execute `SomeObject.Method` within the current cog and if that method returns a value, the Propeller will take that value and use it as the address of code to launch with the cognew command. This will not result in the code writer's intended effect.

If Method is determined to be code that is truly important to run in another cog, rather than write code like the example above, SomeObject should instead be rewritten similar to the example below.

```
VAR
  long StackSpace[8]                    'Stack space for new cog
  byte CogID                            'Stores the ID of new cog

PUB Start
  Stop                                  'Prevent multiple starts
  CogID := cognew(Method, @StackSpace)  'Launch method in another cog

PUB Stop
  if CogID > -1
    cogstop(CogID)                      'Stop previously launched cog

PRI Method
  <some code here>
```

The sample above includes two public interface methods, Start and Stop, which an outside object can use to properly launch the object's code into another cog. The important principle is that the object itself is providing this capability, and in doing so, is managing the stack memory required for proper operation. Also note that Method was changed to a private (**PRI**) method to discourage direct calling from the outside.

IMPROVED

Propeller Assembly Code (Syntax 2))

To run Propeller Assembly code in another cog, the **COGNEW** command needs the address of the assembly routine and a value that can optionally be used by the assembly routine. For example:

```
PUB Main
  cognew(@Toggle, 0)                    'Launch Toggle code

DAT
                org 0                   'Reset assembly pointer
Toggle          rdlong  Delay, #0       'Get clock frequency
                shr     Delay, #2       'Divide by 4
                mov     Time,  cnt      'Get current time
                add     Time,  Delay    'Adjust by 1/4 second
                mov     dira,  #1       'set pin 0 to output
```

```
Loop            waitcnt  Time,  Delay      'Wait for 1/4 second
                xor      outa,  #1          'toggle pin
                jmp      #Loop              'loop back

Delay   res      1
Time    res      1
```

The COGNEW instruction, in the Main method above, tells the Propeller chip to launch the Toggle assembly code into a new cog. The Propeller then finds an available cog and copies 496 longs of the DAT block's content, starting at Toggle, into the cog's RAM. Then the cog's PAR register is initialized, the remaining special purpose registers are cleared, and the cog starts executing the assembly code starting at Cog RAM location 0.

NEW

The *Parameter* Field

It's important to note that the *Parameter* field is intended to pass a long address, so only 14 bits (bits 2 through 15) are passed into the cog's PAR register; the lower two bits are always cleared to zero to ensure it's a long-aligned address. A value other than a long address can still be passed via the *Parameter* field, but will have to be limited to 14 bits, must be shifted left by two bits (for the COGNEW/COGINIT command), and will have to be shifted right by two bits by the assembly program that receives it.

COGSTOP

Command: Stop cog by its ID.

((PUB ¦ PRI))
  COGSTOP (*CogID*)

- ***CogID*** is the ID (0 – 7) of the cog to stop.

Explanation

COGSTOP stops a cog whose ID is *CogID* and places that cog into a dormant state. In the dormant state, the cog ceases to receive System Clock pulses so that power consumption is greatly reduced.

To stop a cog, issue the COGSTOP command with the ID of the cog to stop. For example:

```
VAR
  byte Cog    'Used to store ID of newly started cog


PUB Start(Pos) : Pass
  'Start a new cog to run Update with Pos,
  'return TRUE if successful
  Pass := (Cog := cognew(@Update, Pos) + 1) > 0


PUB Stop
  'Stop the cog we started earlier, if any.
  if Cog
    cogstop(Cog~ - 1)
```

This example, from the COGNEW description, uses COGSTOP in the public Stop method to stop the cog that was previously started by the Start method. See COGNEW, page 78, for more information about this example.

CON

Designator: Declare a Constant Block.

```
CON
   Symbol = Expression ⟨(( , ¦ ↳)) Symbol = Expression⟩...
```

```
CON
   #Expression (( , ¦ ↳)) Symbol ⟨[Offset]⟩ ⟨(( , ¦ ↳)) Symbol ⟨[Offset]⟩ ⟩...
```

IMPROVED

```
CON
   Symbol ⟨[Offset]⟩ ⟨(( , ¦ ↳)) Symbol ⟨[Offset]⟩ ⟩...
```

IMPROVED

- ***Symbol*** is the desired name for the constant.
- ***Expression*** is any valid integer, or floating-point, constant algebraic expression. Expression can include other constant symbols as long as they were defined previously.

NEW

- ***Offset*** is an optional expression by which to adjust the enumeration value for the *Symbol* following this one. If *Offset* is not specified, the default offset of 1 is applied. Use *Offset* to influence the next *Symbol's* enumerated value to something other than this *Symbol's* value plus 1.

Explanation

The Constant Block is a section of source code that declares global constant symbols and global Propeller configuration settings. This is one of six special declarations (`CON`, `VAR`, `OBJ`, `PUB`, `PRI`, and `DAT`) that provide inherent structure to the Spin language.

Constants are numerical values that cannot change during run time. They can be defined in terms of single values (1, $F, 65000, %1010, %%2310, "A", etc.) or as expressions, called constant expressions, (25 + 16 / 2, 1000 * 5, etc.) that always resolve to a specific number.

The Constant Block is an area of code specifically used for assigning symbols (useful names) to constants so that the symbols can be used anywhere in code where that constant value is needed. This makes code more readable and easier to maintain should you later have to change the value of a constant that appears in many places. These constants are global to the object so that any method within it can use them. There are many ways to define constants, described below.

Common Constant Declarations (Syntax 1)

The most common forms of constant declarations begin with **CON** on a line by itself followed by one or more declarations. **CON** must start in column 1 (the leftmost column) of the line it is on and we recommend the lines following be indented by at least one space. The expressions can be combinations of numbers, operators, parentheses, and quoted characters. See Operators, page 143, for examples of expressions.

Example:

```
CON
  Delay = 500
  Baud = 9600
  AChar = ''A''
```

—or—

```
CON
  Delay = 500, Baud = 9600, AChar = ''A''
```

Both of these examples create a symbol called `Delay` that is equal to 500, a symbol called `Baud` that is equal to 9600, and a symbol called `AChar` that is equal to the character "A". For the `Delay` declaration, for example, we could also have used an algebraic expression, such as:

```
  Delay = 250 * 2
```

The above statement results in `Delay` equaling 500, like before, but the expression may make the code easier to understand if the resulting number were not just an arbitrary value.

The **CON** block is also used for specifying global settings, such as system clock settings. The example below shows how to set the Clock Mode to low-speed crystal, the Clock PLL to 8x, and specify that the XIN pin frequency is 4 MHz.

```
CON
  _CLKMODE = XTAL1 + PLL8X
  _XINFREQ = 4_000_000
```

See **_CLKMODE**, page 68, and **_XINFREQ**, page 236, for detailed descriptions of these settings.

Floating-point values can also be defined as constants. Floating-point values are real numbers (with fractional components) and are encoded within 32 bits differently than integer constants. To specify a floating-point constant, you must give a clear indication that the

value is a floating-point value; the expression must either be a single floating-point value or be made up entirely of floating-point values (no integers).

Floating-point values must be written as:

1) decimal digits followed by a decimal point and at least one more decimal digit,
2) decimal digits followed by "e" (for exponent) and an integer exponent value, or,
3) a combination of 1 and 2.

The following are examples of valid constants:

`0.5`	floating-point value
`1.0`	floating-point value
`3.14`	floating-point value
`1e16`	floating-point value
`51.025e5`	floating-point value
`3 + 4`	integer expression
`3.0 + 4.0`	floating-point expression
`3.0 + 4`	invalid expression; causes compile error
`3.0 + FLOAT(4)`	floating-point expression

Here is an example declaring an integer constant and two floating-point constants.

```
CON
  Num1 = 20
  Num2 = 127.38
  Num3 = 32.05 * 18.1 - Num2 / float(Num1)
```

The above code sets `Num1`, `Num2` and `Num3` to 20, 127.38 and 573.736, respectively. Notice that the last expression required `Num1` to be enclosed in the **`FLOAT`** declaration so that the compiler treats it as a floating-point value.

The Propeller compiler handles floating-point constants as a single-precision real number as described by the IEEE-754 standard. Single-precision real numbers are stored in 32 bits, with a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa (the fractional part). This provides approximately 7.2 significant decimal digits.

For run-time floating-point operations, the FloatMath, FloatString, Float32, and Float32Full objects provide math functions compatible with single-precision numbers.

See FLOAT on page 108, ROUND on page 198, TRUNC on page 209, and the FloatMath, FloatString, Float32, and Float32Full objects for more information.

Enumerations (Syntax 2 and 3)

Constant Blocks can also declare enumerated constant symbols. Enumerations are logically grouped symbols which have incrementing integer constant values assigned to them that are each unique for the group. For example, an object may have the need for certain modes of operation. Each of these modes can be identified by a number, 0, 1, 2 and 3, for example. The numbers themselves don't really matter for our purposes; they just need to be unique within the context of the operation mode. Since the numbers themselves are not descriptive, it may be difficult to remember what mode 3 does, but it is a lot easier to remember what the mode means if it had a descriptive name instead. Look at the following example.

```
CON
  'Declare modes of operation
  RunTest    = 0
  RunVerbose = 1
  RunBrief   = 2
  RunFull    = 3
```

The above example would suffice for our purposes; now users of our object can indicate "RunFull" instead of "3" to specify the desired mode of operation. The problem is, defining a logical group of items this way may cause bugs and maintenance problems because if any value was changed (on purpose or by accident) without changing the rest accordingly, it may cause the program to fail. Also, imagine a case where there were 20 modes of operation. That would be a much longer set of constants and even more opportunities for maintenance issues.

Enumerations solve these problems by automatically incrementing values for symbols. We can rewrite the above example with enumeration syntax as follows:

```
CON    'Declare modes of operation
  #0, RunTest, RunVerbose, RunBrief, RunFull
```

Here, #0, tells the compiler to start counting from the number 0 and it sets the next symbol equal to that value. Then, any additional symbols that do not specify their own value (via an '= expression') are automatically assigned the previous value plus 1. The result is that RunTest equals 0, RunVerbose equals 1, RunBrief equals 2 and RunFull equals 3. For most

cases, the values themselves don't usually matter; all that matters is that they are each assigned a unique number. Defining enumerated values like this has the advantages of insuring that the assigned values are unique and contiguous within the group.

Using the example above, the methods that use them can do things like the following (assume `Mode` is a symbol set by a calling object):

```
  case Mode
    RunTest    : <test code here>
    RunVerbose : <verbose code here>
    RunBrief   : <brief code here>
    RunFull    : <full code here>
```

—or—

```
  if Mode > RunVerbose
    <brief and run mode code here>
```

Notice that these routines do not rely on the exact value of the mode, but rather they rely on the enumerated mode symbol itself for comparisons as well as the position of the symbol in relation to other symbols in the same enumeration. It is important to write code this way to decrease potentials for bugs introduced by future changes.

Enumerations don't have to consist of comma-separated items either. The following also works and leaves room for right-side comments about each mode.

```
CON    'Declare modes of operation
  #0
  RunTest          'Run in test mode
  RunVerbose       'Run in verbose mode
  RunBrief         'Run with brief prompts
  RunFull          'Run in full production mode
```

The above example does the same thing as the previous in-line example, but now we have convenient room to describe the purpose of each mode without losing the automatic incrementing advantage. Later on, if there's a need to add a fifth mode, simply add it to the list in whatever position is necessary. If there is a need for the list to begin at a certain value, simply change the `#0` to whatever you need: `#1`, `#20`, etc.

It is even possible to modify the enumerated value in the middle of the list.

```
CON
  'Declare modes of operation
  #1, RunTest, RunVerbose, #5, RunBrief, RunFull
```

Here, RunTest and RunVerbose are 1 and 2, respectively, and RunBrief and RunFull are 5 and 6, respectively. While this feature may be handy, to maintain good programming practices it should only be used in rare cases.

NEW

A more recommended way to achieve the previous example's result is to include the optional *Offset* field. The previous code could have been written as follows:

```
CON
  'Declare modes of operation
  #1, RunTest, RunVerbose[3], RunBrief, RunFull
```

Just as before, RunTest and RunVerbose are 1 and 2, respectively. The [3] immediately following RunVerbose causes the current enumeration value (2) to be incremented by 3 before the next enumerated symbol. The effect of this is also like before, RunBrief and RunFull are 5 and 6, respectively. The advantage of this technique, however, is that the enumerated symbols are all set relative to each other. Changing the line's starting value causes them all to change relatively. For example, changing the #1, to #4 causes RunTest and RunVerbose to be 4 and 5, respectively, and RunBrief and RunFull to be 8 and 9, respectively. In contrast, if the original example's #1 were changed to #4, both RunVerbose and RunBrief would be set to 5, possibly causing the code that relies on those symbols to misbehave.

The *Offset* value may be any signed value, but only affects the value immediately following it; the enumerated value is always incremented by 1 after a *Symbol* that doesn't specify an *Offset*. If overlapping values are desired, specifying an *Offset* of 0 or less can achieve that effect.

Syntax 3 is a variation of the enumeration syntax. It doesn't specify any starting value. Anything defined this way will always start with the first symbol equal to either 0 (for new **CON** blocks) or to the next enumerated value relative to the previous one (within the same **CON** block).

Scope of Constants

Symbolic constants defined in Constant Blocks are global to the object in which they are defined but not outside of that object. This means that constants can be accessed directly from anywhere within the object but their name will not conflict with symbols defined in other parent or child objects.

Symbolic constants can be indirectly accessed by parent objects, however, by using the constant reference syntax.

Example:

```
OBJ
  Num : "Numbers"

PUB SomeRoutine
  Format := Num#DEC     'Set Format to Number's Decimal constant
```

Here an object, "Numbers," is declared as the symbol Num. Later, a method refers to numbers' **DEC** constant with Num#DEC. Num is the object reference, # indicates we need to access that object's constants, and DEC is the constant within the object we need. This feature allows objects to define constants for use with themselves and for parent objects to access those constants freely without interfering with any symbols they created themselves.

CONSTANT

Directive: Declare in-line constant expression to be completely resolved at compile time.

((PUB ¦ PRI))
  CONSTANT (*ConstantExpression*)

Returns: Resolved value of constant expression.

- ***ConstantExpression*** is the desired constant expression.

Explanation

The CON block may be used to create constants from expressions that are referenced from multiple places in code, but there are occasions when a constant expression is needed for temporary, one-time purposes. The CONSTANT directive is used to fully resolve a method's in-line, constant expression at compile time. Without the use of the CONSTANT directive, a method's in-line expressions are always resolved at run time, even if the expression is always a constant value.

Using CONSTANT

The CONSTANT directive can create one-time-use constant expressions that save code space and speed up run-time execution. Note the two examples below:

Example 1, using standard run-time expressions:

```
CON
  X = 500
  Y = 2500

PUB Blink
  !outa[0]
  waitcnt(X+200 + cnt)                'Standard run-time expression
  !outa[0]
  waitcnt((X+Y)/2 + cnt)              'Standard run-time expression
```

Example 2, same as above, but with CONSTANT directive around constant, run-time expressions:

```
CON
  X = 500
  Y = 2500

PUB Blink
  !outa[0]
  waitcnt(constant(X+200) + cnt)   'exp w/compile & run-time parts
  !outa[0]
  waitcnt(constant((X+Y)/2) + cnt)'exp w/compile & run-time parts
```

The above two examples do exactly the same thing: their Blink methods toggle P0, wait for X+200 cycles, toggle P0 again and wait for (X+Y)/2 cycles before returning. While the **CON** block's X and Y symbols may need to be used in multiple places within the object, the **WAITCNT** expressions used in each example's Blink method might only need to be used in that one place. For this reason, it may not make sense to define additional constants in the **CON** block for things like X+200 and (X+Y)/2. There is nothing wrong with putting the expressions right in the run-time code, as in Example 1, but that entire expression is unfortunately evaluated at run time, requiring extra time and code space.

The **CONSTANT** directive is perfect for this situation, because it completely resolves each one-time-use constant expression to a single, static value, saving code space and speeding up execution. In Example 1, the Blink method consumes 33 bytes of code space while Example 2's Blink method, with the addition of the **CONSTANT** directives, only requires 23 bytes of space. Note that the "+ cnt" portion of the expressions are not included within the **CONSTANT** directive's parentheses; this is because the value of cnt is variable (cnt is the System Counter register; see **CNT**, page 73) so its value cannot be resolved at compile time.

If a constant needs to be used in more than one place in code, it is better to define it in the **CON** block so it is defined only once and the symbol representing it can be used multiple times.

Constants (pre-defined)

The following constants are pre-defined by the compiler:

TRUE	Logical true:	-1	($FFFFFFFF)
FALSE	Logical false:	0	($00000000)
POSX	Maximum positive integer:	2,147,483,647	($7FFFFFFF)
NEGX	Maximum negative integer:	-2,147,483,648	($80000000)
PI	Floating-point value for PI:	≈ 3.141593	($40490FDB)
RCFAST	Internal fast oscillator:	$00000001	(%00000000001)
RCSLOW	Internal slow oscillator:	$00000002	(%00000000010)
XINPUT	External clock/oscillator:	$00000004	(%00000000100)
XTAL1	External low-speed crystal:	$00000008	(%00000001000)
XTAL2	External medium-speed crystal:	$00000010	(%00000010000)
XTAL3	External high-speed crystal:	$00000020	(%00000100000)
PLL1X	External frequency times 1:	$00000040	(%00001000000)
PLL2X	External frequency times 2:	$00000080	(%00010000000)
PLL4X	External frequency times 4:	$00000100	(%00100000000)
PLL8X	External frequency times 8:	$00000200	(%01000000000)
PLL16X	External frequency times 16:	$00000400	(%10000000000)

(All of these constants are also available in Propeller Assembly.)

TRUE and FALSE

TRUE and FALSE are usually used for Boolean comparison purposes:

```
if (X = TRUE) or (Y = FALSE)
  <code to execute if total condition is true>
```

POSX and NEGX

POSX and NEGX are typically used for comparison purposes or as a flag for a specific event:

```
if Z > NEGX
  <code to execute if Z hasn't reached smallest negative>
```

—or—

```
PUB FindListItem(Item) : Index
  Index := NEGX                 'Default to "not found" response
  <code to find Item in list>
  if <item found>
    Index := <items index>
```

PI

PI can be used for floating-point calculations, either floating-point constants or floating-point variable values using the FloatMath and FloatString object.

RCFAST through PLL16X

RCFAST through PLL16X are Clock Mode Setting constants. They are explained in further detail in the _CLKMODE section beginning on page 68.

NEW

Note that they are enumerated constants and are not equivalent to the corresponding CLK register value. See CLK Register on page 28 for information regarding how each constant relates to the CLK register bits.

CTRA, CTRB

Register: Counter A and Counter B Control Registers.

((PUB ¦ PRI))
  CTRA

((PUB ¦ PRI))
  CTRB

Returns: Current value of Counter A or Counter B Control Register, if used as a source variable.

Explanation

CTRA and CTRB are two of six registers (CTRA, CTRB, FRQA, FRQB, PHSA, and PHSB) that affect the behavior of a cog's Counter Modules. Each cog has two identical counter modules (A and B) that can perform many repetitive tasks. The CTRA and CTRB registers contain the configuration settings of the Counter A and Counter B Modules, respectively.

The following discussion uses CTRx, FRQx and PHSx to refer to both the A and B pairs of each register.

Each of the two counter modules can control or monitor up to two I/O pins and perform conditional 32-bit accumulation of the value in the FRQx register into the PHSx register on every clock cycle. Each Counter Module has its own phase-locked loop (PLLx) which can be used to synthesize frequencies from 64 MHz to 128 MHz.

With just a little configuration and in some cases a little maintenance from the cog, the counter modules can be used for:

- Frequency synthesis
- Frequency measurement
- Pulse counting
- Pulse measurement
- Multi-pin state measurement
- Pulse-width modulation (PWM)
- Duty-cycle measurement
- Digital-to-analog conversion (DAC)
- Analog-to-digital conversion (ADC)
- And more.

For some of these operations the cog can set the counter's configuration, via CTRA or CTRB, and it will perform its task completely independently. For others, the cog may use WAITCNT to time-align the counter's reads and writes within a loop; creating the effect of a more complex

state machine. Since the counter's update period may be brief (12.5 ns at 80 MHz), very dynamic signal generation and measurement is possible.

Control Register Fields

The CTRA and CTRB registers each contain four fields shown in the table below.

Table 2-5: CTRA and CTRB Registers						
31	**30..26**	**25..23**	**22..15**	**14..9**	**8..6**	**5..0**
-	CTRMODE	PLLDIV	-	BPIN	-	APIN

APIN

The APIN field of CTRA selects a primary I/O pin for that counter. May be ignored if not used. %0xxxxx = Port A, %1xxxxx = Port B (reserved for future use). In Propeller Assembly, the APIN field can conveniently be written using the MOVS instruction.

Note that writing a zero to CTRA will immediately disable the Counter A and stop all related pin output and PHSA accumulation.

BPIN

The BPIN field of CTRx selects a secondary I/O pin for that counter. This field may be ignored if not used. %0xxxxx = Port A, %1xxxxx = Port B (reserved for future use). In Propeller Assembly, the BPIN field can conveniently be written using the MOVD instruction.

PLLDIV

The PLLDIV field of CTRx selects a PLLx output tap, see table below. This determines which power-of-two division of the VCO frequency will be used as the final PLLx output (a range of 500 KHz to 128 MHz). This field may be ignored if not used. In Propeller Assembly, the PLLDIV field can conveniently be written, along with CTRMODE, using the MOVI instruction.

Table 2-6: PLLDIV Field								
PLLDIV	%000	%001	%010	%011	%100	%101	%110	%111
Output	VCO ÷ 128	VCO ÷ 64	VCO ÷ 32	VCO ÷ 16	VCO ÷ 8	VCO ÷ 4	VCO ÷ 2	VCO ÷ 1

CTRMODE
The CTRMODE field of CTRA and CTRB selects one of 32 operating modes, shown in Table 2-7, for the corresponding Counter A or Counter B. In Propeller Assembly, the CTRMODE field can conveniently be written, along with PLLDIV, using the MOVI instruction.

The modes %00001 through %00011 cause FRQx-to-PHSx, accumulation to occur every clock cycle. This creates a numerically controlled oscillator (NCO) in PHSx[31], which feeds the PLLx's reference input. The PLLx will multiply this frequency by 16 using its voltage-controlled oscillator (VCO).

For stable operation, it is recommended that the VCO frequency be kept within 64 MHz to 128 MHz. This translates to an NCO frequency of 4 MHz to 8 MHz.

Using CTRA and CTRB

In Spin, CTRx can be read/written just like any other register or pre-defined variable. As soon as this register is written, the new operating mode goes into effect for the counter. For example:

```
CTRA := %00100 << 26
```

The above code sets CTRA's CTRMODE field to the NCO mode (%00100) and all other bits to zero.

Table 2-7: Counter Modes (CTRMODE Field Values)				
CTRMODE	**Description**	**Accumulate FRQx to PHSx**	**APIN Output***	**BPIN Output***
%00000	Counter disabled (off)	0 (never)	0 (none)	0 (none)
%00001 %00010 %00011	PLL internal (video mode) PLL single-ended PLL differential	1 (always) 1 1	0 PLLx PLLx	0 0 !PLLx
IMPROVED %00100 %00101	NCO single-ended NCO differential	1 1	PHSx[31] PHSx[31]	0 !PHSx[31]
%00110 %00111	DUTY single-ended DUTY differential	1 1	PHSx-Carry PHSx-Carry	0 !PHSx-Carry
%01000 %01001 %01010 %01011	POS detector POS detector with feedback POSEDGE detector POSEDGE detector w/ feedback	A^1 A^1 A^1 & $!A^2$ A^1 & $!A^2$	0 0 0 0	0 $!A^1$ 0 $!A^1$
%01100 %01101 %01110 %01111	NEG detector NEG detector with feedback NEGEDGE detector NEGEDGE detector w/ feedback	$!A^1$ $!A^1$ $!A^1$ & A^2 $!A^1$ & A^2	0 0 0 0	0 $!A^1$ 0 $!A^1$
%10000 %10001 %10010 %10011 %10100 %10101 %10110 %10111 %11000 IMPROVED %11001 %11010 %11011 %11100 %11101 %11110 %11111	LOGIC never LOGIC !A & !B LOGIC A & !B LOGIC !B LOGIC !A & B LOGIC !A LOGIC A <> B LOGIC !A | !B LOGIC A & B LOGIC A == B LOGIC A LOGIC A | !B LOGIC B LOGIC !A | B LOGIC A | B LOGIC always	0 $!A^1$ & $!B^1$ A^1 & $!B^1$ $!B^1$ $!A^1$ & B^1 $!A^1$ A^1 <> B^1 $!A^1$ | $!B^1$ A^1 & B^1 A^1 == B^1 A^1 A^1 | $!B^1$ B^1 $!A^1$ | B^1 A^1 | B^1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

***Must set corresponding DIR bit to affect pin**
A^1 = APIN input delayed by 1 clock
A^2 = APIN input delayed by 2 clocks
B^1 = BPIN input delayed by 1 clock

DAT

Designator: Declare a Data Block.

DAT

IMPROVED ⟨*Symbol*⟩ *Alignment* ⟨*Size*⟩ ⟨*Data*⟩ ⟨[*Count*]⟩ ⟨, ⟨*Size*⟩ *Data* ⟨[*Count*]⟩⟩...

DAT

IMPROVED ⟨*Symbol*⟩ ⟨*Condition*⟩ *Instruction Operands* ⟨*Effect(s)*⟩

- ***Symbol*** is an optional name for the data, reserved space, or instruction that follows.
- Alignment is the desired alignment and default size (BYTE, WORD, or LONG) of the data elements that follow.
- ***Size*** is the desired size (BYTE, WORD, or LONG) of the following data element immediately following it; alignment is unchanged.
- ***Data*** is a constant expression or comma-separated list of constant expressions. Quoted strings of characters are also allowed; they are treated as a comma-separated list of characters.
- NEW ***Count*** is an optional expression indicating the number of byte-, word-, or long-sized entries of *Data* to store in the data table.
- ***Condition*** is an assembly language condition, IF_C, IF_NC, IF_Z, etc.
- ***Instruction*** is an assembly language instruction, ADD, SUB, MOV, etc.
- NEW ***Operands*** is zero, one, or two operands as required by the *Instruction*.
- ***Effect(s)*** is/are one, two or three assembly language effects that cause the result of the instruction to be written or not, NR, WR, WC, or WZ.

Explanation

A DAT (Data) block is a section of source code that contains pre-defined data, memory reserved for run-time use and Propeller Assembly code. This is one of six special declarations (CON, VAR, OBJ, PUB, PRI, and DAT) that provide inherent structure to the Spin language.

Data blocks are multi-purpose sections of source code that are used for data tables, run-time workspace, and Propeller Assembly code. Assembly code and data can be intermixed, if necessary, so that data is loaded into a cog along with the assembly code.

Declaring Data(Syntax 1)

Data is declared with a specific alignment and size (BYTE, WORD, or LONG) to indicate how it should be stored in memory. The location where data is actually stored depends on the structure of the object and the application it is compiled into since data is included as part of the compiled code.

For example:

```
DAT
  byte 64, "A", "String", 0
  word $FFC2, 75000
  long $44332211, 32
```

The first thing on line two of this example, BYTE, indicates the data following it should be byte-aligned and byte-sized. At compile time, the data following BYTE, 64, "A", etc., is stored in program memory a byte at a time starting at the next available location. Line three specifies word-aligned and word-sized data. Its data, $FFC2 and 75000, will begin at the next word boundary position following the data that appeared before it; with any unused bytes from the previous data filled with zeros to pad up to the next word boundary. The fourth line specifies long-aligned and long-sized data; its data will be stored at the next long boundary following the word-aligned data that appeared before it, with zero-padded words leading up to that boundary. Table 2-8 shows what this looks like in memory (shown in hexadecimal).

Table 2-8: Example Data in Memory

L	0				1				2				3				4				5			
W	0		1		2		3		4		5		6		7		8		9		10		11	
B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
D	40	41	53	74	72	69	6E	67	00	00	C2	FF	F8	24	00	00	11	22	33	44	20	00	00	00

L = longs, W = words, B = bytes, D = data

The first nine bytes (0 – 8) are the byte data from line one; $40 = 64 (decimal), $41 = "A", $53 = "S", etc. Byte 9 is padded with zero to align the first word of word-aligned data, $FFC2, at byte 10. Bytes 10 and 11 (word 5) contain the first word-sized value, $FFC2, stored in low-byte-first format as $C2 and $FF. Bytes 12 and 13 (word 6) is the lowest word of 75000; more on this later. Bytes 14 and 15 (word 7) are zero padded to align the first long of long-aligned data, $44332211. Bytes 16 through 19 (long 5) contain that value in low-byte-first format. Finally, bytes 20 through 23 (long 6) contains the second long of data, 32, in low-byte-first format.

You may have noticed that the value 75000 was specified as a word-sized one. The number 75000 in hexadecimal is $124F8, but since that's larger than a word, only the lowest word ($24F8) of the value was stored. This resulted in word 6 (bytes 12 and 13) containing $F8 and $24, and word 7 (bytes 14 and 15) containing $00 and $00 due to the padding for the following long-aligned values.

This phenomenon, whether or not it is intentional, occurs for byte-aligned/byte-sized data as well, for example:

```
DAT
  byte $FFAA, $BB995511
```

...results in only the low bytes of each value, $AA and $11 being stored in consecutive locations.

Occasionally, however, it is desirable to store an entire large value as smaller elemental units that are not necessarily aligned according to the size of the value itself. To do this, specify the value's size just before the value itself.

```
DAT
  byte word $FFAA, long $BB995511
```

This example specifies byte-aligned data, but a word-sized value followed by a long-sized value. The result is that the memory contains $AA and $FF, consecutively, and following it, $11, $55, $99 and $BB.

If we modify line three of the first example above as follows:

```
  word $FFC2, long 75000
```

...then we'd end up with $F8, $24, $01, and $00 occupying bytes 12 through 15. Byte 15 is the upper byte of the value and it just happens to be immediately left of the next long boundary so no additional zero-padded bytes are needed for the next long-aligned data.

Optionally, the *Symbol* field of syntax 1 can be included to "name" the data. This makes referencing the data from a PUB or PRI block easy. For example:

```
DAT
  MyData    byte $FF, 25, %1010

PUB GetData | Temp
  Temp := MyData[0]             'Get first byte of data table
```

This example creates a data table called MyData that consists of bytes $FF, 25 and %1010. The public method, GetData, reads the first byte of MyData from main memory and stores it in its local variable, Temp.

You can also use the BYTE, WORD, and LONG declarations to read main memory locations. For example:

```
DAT
  MyData    byte $FF, 25, %1010

PUB GetData | Temp
  Temp := BYTE[@MyData][0]      'Get first byte of data table
```

This example is similar to the previous one except that it uses the BYTE declaration to read the value stored at the address of MyData. Refer to BYTE, page 51; WORD, page 227; and LONG, page 128, for more information on reading and writing main memory.

NEW

Declaring Repeating Data (Syntax 1)

Data items may be repeated by using the optional *Count* field. For example:

```
DAT
  MyData      byte  64, $AA[8], 55
```

The above example declares a byte-aligned, byte-sized data table, called MyData, consisting of the following ten values: 64, $AA, $AA, $AA, $AA, $AA, $AA, $AA, $AA, 55. There were eight occurrences of $AA due to the [8] in the declaration immediately after it.

IMPROVED

Writing Propeller Assembly Code (Syntax 2)

In addition to numeric and string data, the DAT block is also used for Propeller Assembly code. The following example toggles pin 0 every ¼ second.

```
DAT
                org 0                           'Reset assembly pointer
Toggle          rdlong  Delay, #0               'Get clock frequency
                shr     Delay, #2               'Divide by 4
                mov     Time,  cnt              'Get current time
                add     Time,  Delay            'Adjust by 1/4 second
                mov     dira,  #1               'set pin 0 to output
Loop            waitcnt Time,  Delay            'Wait for 1/4 second
                xor     outa,  #1               'toggle pin
                jmp     #Loop                   'loop back
```

```
Delay     res       1
Time      res       1
```

When a Propeller Application initially boots up, only Spin code is executed. At any time, however, that Spin code can choose to launch assembly code into a cog of its own. The **COGNEW** (page 78) and **COGINIT** (page 76) commands are used for this purpose. The following Spin code example launches the Toggle assembly code shown above.)

```
PUB Main
  cognew(@Toggle, 0)                          'Launch Toggle code
```

The **COGNEW** instruction, above, tells the Propeller chip to launch the Toggle assembly code into a new cog. The Propeller then finds an available cog, copies the code from the **DAT** block starting at Toggle into the cog's RAM, and then starts the cog which begins executing code from Cog RAM location 0.

A **DAT** block may contain multiple Propeller Assembly programs, or multiple **DAT** blocks may each contain individual assembly programs, but in both cases, each assembly program should begin with an **ORG** directive (page 328) to reset the assembly pointer properly.

NEW

Dual Commands

The Spin and Propeller Assembly languages share a number of like-named commands, called dual commands. These dual commands perform similar tasks but each has a different syntax structure that resembles the language in which it is written; Spin vs. Propeller Assembly. Any dual commands that are used in a **DAT** block are considered to be assembly instructions. Conversely, any dual commands that are used in **PUB** and **PRI** blocks are considered to be Spin commands.

DIRA, DIRB

Register: Direction Register for 32-bit Ports A and B.

((PUB ¦ PRI))
DIRA 〈[*Pin(s)*]〉

((PUB ¦ PRI))
DIRB 〈[*Pin(s)*]〉 (Reserved for future use)

Returns: Current value of direction bits for I/O *Pin(s)* in Ports A or B, if used as a source variable.

- ***Pin(s)*** is an optional expression, or a range-expression, that specifies the I/O pin, or pins, to access in Port A (0-31) or Port B (32-63). If given as a single expression, only the pin specified is accessed. If given as a range-expression (two expressions in a range format; x..y) the contiguous pins from the start to end expressions are accessed.

Explanation

DIRA and DIRB are one of six registers (DIRA, DIRB, INA, INB, OUTA and OUTB) that directly affect the I/O pins. The DIRA register holds the direction states for each of the 32 I/O pins in Port A; bits 0 through 31 correspond to P0 through P31. The DIRB register holds the direction states for each of the 32 I/O pins in Port B; bits 0 through 31 correspond to P32 through P63.

NOTE: DIRB is reserved for future use; the Propeller P8X32A does not include Port B I/O pins so only DIRA is discussed below.

DIRA is used to both set and get the current direction states of one or more I/O pins in Port A. A low (0) bit sets the corresponding I/O pin to an input direction. A high (1) bit sets the corresponding I/O pin to an output direction. All the DIRA register's bits default to zero (0) upon cog startup; all I/O pins are specified as inputs by that cog until the code instructs otherwise.

Each cog has access to all I/O pins at any given time. Essentially, all I/O pins are directly connected to each cog so that there is no hub-related mutually exclusive access involved. Each cog maintains its own DIRA register that gives it the ability to set any I/O pin's direction. Each cog's DIRA register is OR'd with that of the other cogs' DIRA registers and the resulting 32-bit value becomes the I/O directions of Port A pins P0 through P31. The result is that each I/O pin's direction state is the "wired-OR" of the entire cog collective. See I/O Pins on page 26 for more information.

This configuration can easily be described in the following simple rules:

A. A pin is an input only of no active cog sets it to an output.
B. A pin is an output if any active cog sets it to an output.

If a cog is disabled, its direction register is treated as if were cleared to 0, causing it to exert no influence on I/O pin directions and states.

Note that because of the "wired-OR" nature of the I/O pins, no electrical contention between cogs is possible, yet they can all still access I/O pins simultaneously. It is up to the application developer to ensure that no two cogs cause logical contention on the same I/O pin during run time.

Using DIRA

Set or clear bits in **DIRA** to affect the direction of I/O pins as desired. For example:

```
DIRA := %00000000_00000000_10110000_11110011
```

The above code sets the entire **DIRA** register (all 32 bits at once) to a value that makes I/O pins 15, 13, 12, 7, 6, 5, 4, 1 and 0 to outputs and the rest to inputs.

Using the post-clear (**~**) and post-set (**~~**) unary operators, the cog can set all I/O pins to inputs, or outputs, respectively; it's not usually desirable to set all I/O pins to outputs, however. For example:

```
DIRA~                   'Clear DIRA register (all I/Os are inputs)
```

—and—

```
DIRA~~                  'Set DIRA register (all I/Os are outputs)
```

The first example above clears the entire **DIRA** register (all 32 bits at once) to zero; all I/Os P0 through P31 to inputs. The second example above sets the entire **DIRA** register (all 32 bits at once) to ones; all I/Os P0 through P31 to outputs.

To affect only one I/O pin (one bit), include the optional *Pin(s)* field. This treats the **DIRA** register as an array of 32 bits.

```
DIRA[5]~~               'Set DIRA bit 5 (P5 to output)
```

This sets P5 to an output. All other bits of **DIRA** (and thus all other corresponding I/O pins) remain in their previous state.

The **DIRA** register supports a special form of expression, called a range-expression, which allows you to affect a group of I/O pins at once, without affecting others outside the specified range. To affect multiple, contiguous I/O pins at once, use a range expression (like x..y) in the *Pin(s)* field.

```
DIRA[5..3]~~            'Set DIRA bits 5 through 3 (P5-P3 to output)
```

This sets P5, P4 and P3 to outputs; all other bits of **DIRA** remain in their previous state. Here's another example:

```
DIRA[5..3] := %110      'Set P5 and P4 to output, P3 to input
```

The above example sets **DIRA** bits 5, 4 and 3 equal to 1, 1, and 0, respectively, leaving all other bits in their previous state. Consequently, P5 and P4 are now outputs and P3 is an input.

IMPORTANT: The order of the values in a range-expression affects how it is used. For example, the following swaps the order of the range-expression of the previous example.

```
DIRA[3..5] := %110      'Set P3 and P4 to output, P5 to input
```

Here, **DIRA** bits 3, 4 and 5 are set equal to 1, 1, and 0, respectively, making P3 and P4 outputs and P5 an input.

This is a powerful feature of range-expressions, but if care is not taken, it can also cause strange, unintentional results.

Normally **DIRA** is only written to but it can also be read from to retrieve the current I/O pin directions. The following assumes `Temp` is a variable created elsewhere:

```
Temp := DIRA[7..4]      'Get direction of P7 through P4
```

The above sets `Temp` equal to **DIRA** bits 7, 6, 5, and 4; i.e., the lower 4 bits of `Temp` are now equal to `DIRA7:4` and the other bits of `Temp` are cleared to zero.

FILE

Directive: Import external file as data.

```
DAT
  FILE "FileName"
```

- ***FileName*** is the name, without extension, of the desired data file. Upon compile, a file with this name is searched for in the editor tabs, the working directory and the library directory. *FileName* can contain any valid filename characters; disallowed characters are \, /, :, *, ?, ", <, >, and |.

Explanation

The FILE directive is used to import an external data file (usually a binary file) into the DAT block of an object. The data can then be accessed by the object just like any regular DAT block data.

Using FILE

FILE is used in DAT blocks similar to how BYTE would be used, except that following it is a filename in quotes instead of data values. For example:

```
DAT
  Str    byte  "This is a data string.", 0
  Data   file  "Datafile.dat"
```

In this example, the DAT block is made up of a byte string followed by the data from a file called Datafile.dat. Upon compile, the Propeller Tool will search through the editor tabs, the working directory or the library directory for a file called Datafile.dat and will load its data into the first byte following the zero-terminated string, Str. Methods can access the imported data using the BYTE, WORD or LONG declarations as they would normal data. For example:

```
PUB GetData | Index, Temp
  Index := 0
  repeat
    Temp := byte[Data][Index++] 'Read data into Temp 1 byte at a time
    <do something with Temp>    'Perform task with value in Temp
  while Temp > 0                'Loop until end found
```

This example will read the imported data, one byte at a time, until it finds a byte equal to 0.

FLOAT

Directive: Convert an integer constant expression to a compile-time floating-point value.

((CON ¦ VAR ¦ OBJ ¦ PUB ¦ PRI ¦ DAT))
  FLOAT (*IntegerConstant*)

Returns: Resolved value of integer constant expression as a floating-point number.

- ***IntegerConstant*** is the desired integer constant expression to be used as a constant floating-point value.

Explanation

FLOAT is one of three directives (FLOAT, ROUND and TRUNC) used for floating-point constant expressions. The FLOAT directive converts a constant integer value to a constant floating-point value.

Using FLOAT

While most constants are 32-bit integer values, the Propeller compiler supports 32-bit floating-point values and constant expressions for compile-time use. Note that this is for constant expressions only, not run-time variable expressions.

For typical floating-point constant declarations, the expression must be shown as a floating-point value in one of three ways: 1) as an integer value followed by a decimal point and at least one digit, 2) as an integer with an E followed by an exponent value, or 3) both 1 and 2. For example:

```
CON
  OneHalf = 0.5
  Ratio   = 2.0 / 5.0
  Miles   = 10e5
```

The above code creates three floating-point constants. OneHalf is equal to 0.5, Ratio is equal to 0.4 and Miles is equal to 1,000,000.

Notice that in the above example, every component of every expression is shown as a floating-point value. Now take a look at the following example:

```
CON
  Two     = 2
  Ratio   = Two / 5.0
```

Here, Two is defined as an integer constant and Ratio appears to be defined as a floating-point constant. This causes an error on the Ratio line because, for floating-point constant expressions, every value within the expression must be a floating-point value; you cannot mix integer and floating-point values like Ratio = 2 / 5.0.

You can, however, use the FLOAT directive to convert an integer value to a floating-point value, such as in the following:

```
CON
  Two     = 2
  Ratio   = float(Two) / 5.0
```

The FLOAT directive in this example converts the integer constant, Two, into the floating-point form of that value so that it can be used in the floating-point expression.

About Floating Point

The Propeller compiler handles floating-point constants as a single-precision real number as described by the IEEE-754 standard. Single-precision real numbers are stored in 32 bits, with a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa (the fractional part). This provides approximately 7.2 significant decimal digits.

Floating-point constant expressions can be defined and used for many compile-time purposes, but for run-time floating-point operations, the FloatMath, FloatString, Float32, and Float32Full objects provide math functions compatible with single-precision numbers.

See Constant Assignment '=' in the Operators section on page 148, ROUND on page 198, and TRUNC on page 209, as well as the FloatMath, FloatString, Float32, and Float32Full objects for more information.

_FREE

Constant: Pre-defined, one-time settable constant for specifying the size of an application's free space.

```
CON
  _FREE = Expression
```

- ***Expression*** is an integer expression that indicates the number of longs to reserve for free space.

Explanation

_FREE is a pre-defined, one-time settable optional constant that specifies the required free memory space of an application. This value is added to _STACK, if specified, to determine the total amount of free/stack memory space to reserve for a Propeller Application. Use _FREE if an application requires a minimum amount of free memory in order to run properly. If the resulting compiled application is too large to allow the specified free memory, an error message will be displayed. For example:

```
CON
  _FREE    = 1000
```

The _FREE declaration in the above CON block indicates that the application needs to have at least 1,000 longs of free memory left over after compilation. If the resulting compiled application does not have that much room left over, an error message will indicate by how much it was exceeded. This is a good way to prevent successful compiles of an application that will fail to run properly due to lack of memory.

Note that only the top object file can set the value of _FREE. Any child object's _FREE declarations will be ignored.

FRQA, FRQB

Register: Counter A and Counter B frequency registers.

((PUB ¦ PRI))
FRQA

((PUB ¦ PRI))
FRQB

Returns: Current value of Counter A or Counter B Frequency Register, if used as a source variable.

Explanation

FRQA and FRQB are two of six registers (CTRA, CTRB, FRQA, FRQB, PHSA, and PHSB) that affect the behavior of a cog's Counter Modules. Each cog has two identical counter modules (A and B) that can perform many repetitive tasks. The FRQA register contains the value that is accumulated into the PHSA register. The FRQB register contains the value that is accumulated into the PHSB register. See CTRA, CTRB on page 95 for more information.

Using FRQA and FRQB

FRQA and FRQB can be read/written just like any other register or pre-defined variable. For example:

```
FRQA := $00001AFF
```

The above code sets FRQA to $00001AFF. Depending on the CTRMODE field of the CTRA register, this value in FRQA may be added into the PHSA register at a frequency determined by the System Clock and the primary and/or secondary I/O pins. See CTRA, CTRB on page 95 for more information.

IF

Command: Test condition(s) and execute a block of code if valid (positive logic).

((PUB ¦ PRI))
IF ***Condition(s)***
→ ***IfStatement(s)***
〈 ELSEIF ***Condition(s)***
→ ***ElseIfStatement(s)*** 〉...
〈 ELSEIFNOT ***Condition(s)***
→ ***ElseIfNotStatement(s)*** 〉...
〈 ELSE
→ ***ElseStatement(s)*** 〉

- ***Condition(s)*** is one or more Boolean expressions to test.
- ***IfStatement(s)*** is a block of one or more lines of code to execute when the IF's *Condition(s)* is true.
- ***ElseIfStatement(s)*** is an optional block of one or more lines of code to execute when all the previous *Condition(s)* are invalid and the ELSEIF's *Condition(s)* is true.
- ***ElseIfNotStatement(s)*** is an optional block of one or more lines of code to execute when all the previous *Condition(s)* are invalid and the ELSEIFNOT's *Condition(s)* is false.
- ***ElseStatement(s)*** is an optional block of one or more lines of code to execute when all the previous *Condition(s*) are invalid.

Explanation

IF is one of the three major conditional commands (IF, IFNOT, and CASE) that conditionally executes a block of code. IF can optionally be combined with one or more ELSEIF commands, one or more ELSEIFNOT commands, and/or an ELSE command to form sophisticated conditional structures.

IF tests *Condition(s)* and, if true, executes *IfStatement(s)*. If *Condition(s)* is false, the following optional ELSEIF *Condition(s)*, and/or ELSEIFNOT *Condition(s),* are tested, in order, until a valid condition line is found, then the associated *ElseIfStatement(s)*, or *ElseIfNotStatement(s)*, block is executed. The optional *ElseStatement(s)* block is executed if no previous valid condition lines are found.

A "valid" condition is one that evaluates to TRUE for a positive conditional statement (IF or ELSEIF) or evaluates to FALSE for a negative conditional statement (ELSEIFNOT).

Indention is Critical

IMPORTANT: Indention is critical. The Spin language relies on indention (of one space or more) on lines following conditional commands to determine if they belong to that command or not. To have the Propeller Tool indicate these logically grouped blocks of code on-screen, you can press Ctrl + I to turn on block-group indicators. Pressing Ctrl + I again will disable that feature. See the Propeller Tool Help for a complete list of shortcut keys.

Simple IF Statement

The most common form of the **IF** conditional command performs an action if, and only if, a condition is true. This is written as an **IF** statement followed by one or more indented lines of code. For example:

```
if X > 10                   'If X is greater than 10
  !outa[0]                  'Toggle P0
!outa[1]                    'Toggle P1
```

This example tests if X is greater than 10; if it is, I/O pin 0 is toggled. Whether or not the **IF** condition was true, I/O pin P1 is toggled next.

Since the !outa[0] line is indented from the **IF** line, it belongs to the *IfStatement(s)* block and is executed only if the **IF** condition is true. The next line, !outa[1], is not indented from the **IF** line, so it is executed next whether or not the **IF**'s *Condition(s)* was true. Here's another version of the same example:

```
if X > 10                   'If X is greater than 10
  !outa[0]                  'Toggle P0
  !outa[1]                  'Toggle P1
waitcnt(2_000 + cnt)        'Wait for 2,000 cycles
```

This example is very similar to the first, except there are now two lines of code indented from the **IF** statement. In this case, if X is greater than 10, P0 is toggled then P1 is toggled and finally the waitcnt line is executed. If, however, X was not greater than 10, the !outa[0] and !outa[1] lines are skipped (since they are indented and part of the *IfStatement(s)* block) and the waitcnt line is executed (since it is not indented; it is not part of the *IfStatement(s)* block).

Combining Conditions

The *Condition(s)* field is evaluated as one single Boolean condition, but it can be made up of more than one Boolean expression by combining them with the **AND** and **OR** operators; see pages 167-168. For example:

```
if X > 10 AND X < 100     'If X greater than 10 and less than 100
```

This IF statement would be true if, and only if, X is greater than 10 and X is also less than 100. In other words, it's true if X is in the range 11 to 99. Sometimes statements like these can be a little difficult to read. To make it easier to read, parentheses can be used to group each sub-condition, such as with the following.

```
if (X > 10) AND (X < 100)'If X greater than 10 and less than 100
```

Using IF with ELSE

The second most common form of the IF conditional command performs an action if a condition is true or a different action if that condition is false. This is written as an IF statement followed by its *IfStatement(s)* block, then an ELSE followed by its *ElseStatement(s)* block, as shown below:

```
if X > 100                'If X is greater than 100
  !outa[0]                'Toggle P0
else                      'Else, X <= 100
  !outa[1]                'Toggle P1
```

Here, if X is greater than 100, I/O pin 0 is toggled, otherwise, X must be less than or equal to 100, and I/O pin 1 is toggled. This IF...ELSE construct, as written, always performs either a toggle on P0 or a toggle on P1; never both, and never neither.

Remember, the code that logically belongs to the *IfStatement(s)* or the *ElseStatement(s)* must be indented from the IF or the ELSE, respectively, by at least one space. Also note that the ELSE must be lined up horizontally with the IF statement; they must both begin on the same column or the compiler will not know that the ELSE goes with that IF.

For every IF statement, there can be zero or one ELSE component. ELSE must be the last component in an IF statement, appearing after any potential ELSEIFs.

Using IF with ELSEIF

The third form of the IF conditional command performs an action if a condition is true or a different action if that condition is false but another condition is true, etc. This is written as an IF statement followed by its *IfStatement(s)* block, then one or more ELSEIF statements followed by their respective *ElseIfStatement(s)* blocks. Here's an example:

```
if X > 100                'If X is greater than 100
  !outa[0]                'Toggle P0
elseif X == 90            'Else If X = 90
```

```
  !outa[1]                  'Toggle P1
```

Here, if X is greater than 100, I/O pin 0 is toggled, otherwise, if X equals 90, I/O pin 1 is toggled, and if neither of those conditions were true, neither P0 nor P1 is toggled. This is a slightly shorter way of writing the following code:

```
if X > 100                  'If X is greater than 100
  !outa[0]                  'Toggle P0
else                        'Otherwise,
  if X == 90                'If X = 90
    !outa[1]                'Toggle P1
```

Both of these examples perform the same actions, but the first is shorter and is usually considered easier to read. Note that the **ELSEIF**, just like the **ELSE**, must be lined up (start in the same column) as the **IF** that it is associated with.

Each **IF** conditional statement can have zero or more **ELSEIF** statements associated with it. Look at the following:

```
if X > 100                  'If X is greater than 100
  !outa[0]                  'Toggle P0
elseif X == 90              'Else If X = 90
  !outa[1]                  'Toggle P1
elseif X > 50               'Else If X > 50
  !outa[2]                  'Toggle P2
```

We have three conditions and three possible actions here. Just like the previous example, if X is greater than 100, P0 is toggled, otherwise, if X equals 90, P1 is toggled, but if neither of those conditions were true and X is greater than 50, P2 is toggled. If none of those conditions were true, then none of those actions would occur.

There is an important concept to note about this example. If X is 101 or higher, P0 is toggled, or if X is 90, P1 is toggled, or if X is 51 to 89, or 91 to 100, P2 is toggled. This is because the **IF** and **ELSEIF** conditions are tested, one at a time, in the order they are listed and only the first condition that is true has its block of code executed; no further conditions are tested after that. This means that if we had rearranged the two **ELSEIF**s so that the "X > 50" were checked first, we'd have a bug in our code.

We did this below:

```
if X > 100                  'If X is greater than 100
  !outa[0]                  'Toggle P0
elseif X > 50               'Else If X > 50
  !outa[2]                  'Toggle P2
elseif X == 90              'Else If X = 90 <-- ERROR, ABOVE COND.
  !outa[1]                  'Toggle P1      <-- SUPERSEDES THIS AND
                            '                   THIS CODE NEVER RUNS
```

The above example contains an error because, while `X` could be equal to 90, the `elseif X == 90` statement would never be tested because the previous one, `elseif X > 50`, would be tested first, and since it is true, its block is executed and no further conditions of that **IF** structure are tested. If `X` were 50 or less, the last **ELSEIF** condition is tested, but of course, it will never be true.

Using IF with ELSEIF and ELSE

Another form of the **IF** conditional command performs one of many different actions if one of many different conditions is true, or an alternate action if none of the previous conditions were true. This is written as with an **IF**, one or more **ELSEIFs**, and finally an **ELSE**. Here's an example:

```
if X > 100                  'If X is greater than 100
  !outa[0]                  'Toggle P0
elseif X == 90              'Else If X = 90
  !outa[1]                  'Toggle P1
elseif X > 50               'Else If X > 50
  !outa[2]                  'Toggle P2
else                        'Otherwise,
  !outa[3]                  'Toggle P3
```

This is just like the example above, except that if none of the **IF** or **ELSEIF** conditions are true, P3 is toggled.

The ELSEIFNOT Condition

The **ELSEIFNOT** condition behaves exactly like **ELSEIF** except that it uses negative logic; it executes its *ElseIfNotStatement(s)* block only if its *Condition(s)* expression evaluates to **FALSE**. Multiple **ELSEIFNOT** and **ELSEIF** conditions can be combined in a single **IF** conditional command, in any order, between the **IF** and the optional **ELSE**.

IFNOT

Command: Test condition(s) and execute a block of code if valid (negative logic).

((PUB ¦ PRI))
IFNOT ***Condition(s)***
→ ***IfNotStatement(s)***
〈 ELSEIF ***Condition(s)***
→ ***ElseIfStatement(s)*** 〉...
〈 ELSEIFNOT ***Condition(s)***
→ ***ElseIfNotStatement(s)*** 〉...
〈 ELSE
→ ***ElseStatement(s)*** 〉

- ***Condition(s)*** is one or more Boolean expressions to test.
- ***IfNotStatement(s)*** is a block of one or more lines of code to execute when the IFNOT's *Condition(s)* is false.
- ***ElseIfStatement(s)*** is an optional block of one or more lines of code to execute when all the previous *Condition(s)* are invalid and the ELSEIF's *Condition(s)* is true.
- ***ElseIfNotStatement(s)*** is an optional block of one or more lines of code to execute when all the previous *Condition(s)* are invalid and the ELSEIFNOT's *Condition(s)* is false.
- ***ElseStatement(s)*** is an optional block of one or more lines of code to execute when all the previous *Condition(s*) are invalid.

Explanation

IFNOT is one of the three major conditional commands (IF, IFNOT, and CASE) that conditionally executes a block of code. IFNOT is the complementary (negative) form of IF.

IFNOT tests *Condition(s)* and, if false, executes *IfNotStatement(s)*. If *Condition(s)* is true, the following optional ELSEIF *Condition(s)*, and/or ELSEIFNOT *Condition(s),* are tested, in order, until a valid condition line is found, then the associated *ElseIfStatement(s)*, or *ElseIfNotStatement(s)*, block is executed. The optional *ElseStatement(s)* block is executed if no previous valid condition lines are found.

A "valid" condition is one that evaluates to FALSE for a negative conditional statement (IFNOT, or ELSEIFNOT) or evaluates to TRUE for a positive conditional statement (ELSEIF).

See IF on page 112 for information on the optional components of IFNOT.

INA, INB

Register: Input Registers for 32-bit Ports A and B.

((PUB ¦ PRI))
INA 〈[*Pin(s)*]〉

((PUB ¦ PRI))
INB 〈[*Pin(s)*]〉 (Reserved for future use)

Returns: Current state of I/O *Pin(s)* for Port A or B.

- ***Pin(s)*** is an optional expression, or a range-expression, that specifies the I/O pin, or pins, to access in Port A (0-31) or Port B (32-63). If given as a single expression, only the pin specified is accessed. If given as a range-expression (two expressions in a range format; x..y) the contiguous pins from the start to end expressions are accessed.

Explanation

INA and INB are two of six registers (DIRA, DIRB, INA, INB, OUTA and OUTB) that directly affect the I/O pins. The INA register contains the current states for each of the 32 I/O pins in Port A; bits 0 through 31 correspond to P0 through P31. The INB register contains the current states for each of the 32 I/O pins in Port B; bits 0 through 31 correspond to P32 through P63.

NOTE: INB is reserved for future use; the Propeller P8X32A does not include Port B I/O pins so only INA is discussed below.

INA is read-only and is not really implemented as a register but rather is just an address that, when accessed as a source item in an expression, reads the Port A I/O pins directly at that moment. In the result, a low (0) bit indicates the corresponding I/O pin senses ground, and a high (1) bit indicates the corresponding I/O pin senses VDD (3.3 volts). Since the Propeller is a CMOS device, the I/O pins sense anything above ½ VDD to be high, so a high means the pin senses approximately 1.65 volts or higher.

Each cog has access to all I/O pins at any given time. Essentially, all I/O pins are directly connected to each cog so that there is no hub-related mutually exclusive access involved. Each cog has its own INA pseudo-register that gives it the ability to read the I/O pins states (low or high) at any time. The actual I/O pins' values are read, regardless of their designated input or output direction.

Note because of the "wired-OR" nature of the I/O pins, no electrical contention between cogs is possible, yet they can all still access I/O pins simultaneously. It is up to the application developer to ensure that no two cogs cause logical contention on the same I/O pin during run time. Since all cogs share all I/O pins, a cog could use INA to read pins it is using as well as the pins that are in use by one or more other cogs.

Using INA

Read INA to get the state of I/O pins at that moment. The following example assumes Temp was created elsewhere.

```
Temp := INA                 'Get state of P0 through P31
```

This example reads the states of all 32 I/O pins of Port A into Temp.

Using the optional *Pin(s)* field, the cog can read one I/O pin (one bit) at a time. For example:

```
Temp := INA[16]             'Get state of P16
```

The above line reads I/O pin 16 and stores its state (0 or 1) in the lowest bit of Temp; all other bits of Temp are cleared.

In Spin, the INA register supports a special form of expression, called a range-expression, which allows you to read a group of I/O pins at once, without reading others outside the specified range. To read multiple, contiguous I/O pins at once, use a range expression (like x..y) in the *Pin(s)* field.

```
Temp := INA[18..15]         'Get states of P18:P15
```

Here, the lowest four bits of Temp (3, 2, 1, and 0) are set to the states of I/O pins 18, 17, 16, and 15, respectively, and all other bits of Temp are cleared to 0.

IMPORTANT: The order of the values in a range-expression affects how it is used. For example, the following swaps the order of the range from the previous example.

```
Temp := INA[15..18]         'Get states of P15:P18
```

Here, Temp bits 3, 2, 1, and 0 are set to the states of I/O pins 15, 16, 17, and 18, respectively.

This is a powerful feature of range-expressions, but if care is not taken it can also cause strange, unintentional results.

LOCKCLR

Command: Clear lock to false and get its previous state.

((PUB ¦ PRI))
  LOCKCLR (*ID*)

Returns: Previous state of lock (TRUE or FALSE).

- *ID* is the ID (0 – 7) of the lock to clear to false.

Explanation

LOCKCLR is one of four lock commands (LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR) used to manage resources that are user-defined and deemed mutually exclusive. LOCKCLR clears lock ID to FALSE and retrieves the previous state of that lock (TRUE or FALSE).

See About Locks, page 122, and Suggested Rules for Locks, page 123 for information on the typical use of locks and the LOCKxxx commands.

The following assumes that a cog (either this one or another) has already checked out a lock using LOCKNEW and shared the ID with this cog, which saved it as SemID. It also assumes this cog has an array of longs called LocalData.

```
PUB ReadResource | Idx
  repeat until not lockset(SemID) 'wait until we lock the resource
  repeat Idx from 0 to 9          'read all 10 longs of resource
    LocalData[Idx] := long[Idx]
  lockclr(SemID)                  'unlock the resource

PUB WriteResource | Idx
  repeat until not lockset(SemID) 'wait until we lock the resource
  repeat Idx from 0 to 9          'write all 10 longs to resource
    long[Idx] := LocalData[Idx]
  lockclr(SemID)                  'unlock the resource
```

Both of these methods, ReadResource and WriteResource, follow the same rules before and after accessing the resource. First, they wait indefinitely at the first REPEAT loop until it has locked the resource; i.e., it has successfully "set" the associated lock. If LOCKSET returns TRUE, the condition "until not lockset…" is FALSE, meaning that some other cog is currently accessing the resource, so that first repeat loop tries again. If LOCKSET returns FALSE, the

condition "until not lockset…" is true, meaning we have "locked the resource" and the first repeat loop ends. The second REPEAT loop in each method reads or writes the resource, via the long[Idx] and LocalData[Idx] statements. The last line of each method, lockclr(SemID), clears the resource's associated lock to FALSE, logically unlocking or releasing the resource for others to use.

See LOCKNEW, page 122; LOCKRET, page 125; and LOCKSET, page 126 for more information.

LOCKNEW

Command: Check out a new lock and get its ID.

((PUB ¦ PRI))
  LOCKNEW

Returns: ID (0-7) of the lock checked out, or -1 if none were available.

Explanation

LOCKNEW is one of four lock commands (LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR) used to manage resources that are user-defined and deemed mutually exclusive. LOCKNEW checks out a unique lock, from the Hub, and retrieves the ID of that lock. If no locks were available, LOCKNEW returns -1.

About Locks

A lock is a semaphore mechanism that is used to communicate between two or more entities. In the Propeller chip, a lock is simply one of eight global bits in a protected register within the Hub. The Hub maintains an inventory of which locks are in use and their current states. Cogs can check out, set, clear, and return locks as needed during run time to indicate whether a custom shared item, such as a block of memory, is available or not. Since locks are managed by the Hub only one cog can affect them at a time, making this an effective control mechanism.

In applications where two or more cogs are sharing the same memory, a tool such as a lock may be required to prevent catastrophic collisions from occurring. The Hub prevents such collisions from occurring on elemental data (such a byte, word or long) at every moment in time, but it cannot prevent "logical" collisions on blocks of multiple elements (such as a block of bytes, words, longs or any combination of these). For example, if two or more cogs are sharing a single byte of main memory, each one is guaranteed exclusive access to that byte by nature of the Hub. But if those two cogs share multiple bytes of main memory, the Hub can not prevent one cog from writing a few of those bytes while another cog is reading all of them; all cogs' interactions with those bytes may be interleaved in time. In this case, the developer should design each process (in each cog that shares this memory) so that they cooperatively share the memory block in a non-destructive way. Locks serve as flags that notify each cog when a memory block is safe to manipulate or not.

Using LOCKNEW

A user-defined, mutually exclusive resource should be initially set up by a cog, then that same cog should use **LOCKNEW** to check out a unique lock in which to manage that resource and pass the ID of that lock to any other cogs that require it. For example:

```
VAR
  byte SemID

PUB SetupSharedResource
  <code to set up user-defined, shared resource here>
  if (SemID := locknew) == -1
    <error, no locks available>
  else
    <share SemID's value with other cogs>
```

The example above calls **LOCKNEW** and stores the result in SemID. If that result is -1, an error occurs. If the SemID is not -1, then a valid lock was checked out and that SemID needs to be shared with other cogs along with the address of the resource that SemID is used for. The method used to communicate the SemID and resource address depends on the application, but typically they are both passed as parameters to the Spin method that is launched into a cog, or as the **PAR** parameter when launching an assembly routine into a cog. See **COGNEW**, page 78.

Suggested Rules for Locks

The following are the suggested rules for using locks.

- Objects needing a lock to manage a user-defined, mutually exclusive resource should check out a lock using **LOCKNEW** and save the ID returned, we'll call it SemID here. Only one cog should check out this lock. The cog that checked out the lock must communicate SemID to all other cogs that will use the resource.
- Any cog that needs to access the resource must first successfully set the lock SemID. A successful "set" is when LOCKSET(SemID) returns **FALSE**; i.e., the lock was not already set. If **LOCKSET** returned **TRUE**, then another cog must be accessing the resource; you must wait and try again later to get a successful "set".
- The cog that has achieved a successful "set" can manipulate the resource as necessary. When done, it must clear the lock via LOCKCLR(SemID) so another cog can have access to the resource. In a well-behaved system, the result of **LOCKCLR** can be ignored here since this cog is the only one with the logical right to clear it.

- If a resource is no longer needed, or becomes non-exclusive, the associated lock should be returned to the lock pool via LOCKRET(SemID). Usually this is done by the same cog that checked out the lock originally.

Applications should be written such that locks are not accessed with LOCKSET or LOCKCLR unless they are currently checked out.

Note that user-defined resources are not actually locked by either the Hub or the checked-out lock. The lock feature only provides a means for objects to cooperatively lock those resources. It's up to the objects themselves to decide on, and abide by, the rules of lock use and what resource(s) will be governed by them. Additionally, the Hub does not directly assign a lock to the cog that called LOCKNEW, rather it simply marks it as being "checked out" by a cog; any other cog can "return" locks to the pool of available locks. Also, any cog can access any lock through the LOCKCLR and LOCKSET commands even if those locks were never checked out. Doing such things is generally not recommended because of the havoc it can cause with other, well-behaved objects in the application.

See LOCKRET, page 125; LOCKCLR, page 120; and LOCKSET, page 126 for more information.

LOCKRET

Command: Release lock back to lock pool, making it available for future LOCKNEW requests.

((PUB ¦ PRI))
  LOCKRET (*ID*)

- *ID* is the ID (0 – 7) of the lock to return to the lock pool.

Explanation

LOCKRET is one of four lock commands (LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR) used to manage resources that are user-defined and deemed mutually exclusive. LOCKRET returns a lock, by *ID*, back to the Hub's lock pool so that it may be reused by other cogs at a later time. For example:

```
LOCKRET (2)
```

This example returns Lock 2 back to the Hub. This doesn't prevent cogs from accessing Lock 2 afterwards, it only allows the Hub to reassign it to cogs that call LOCKNEW in the future. Applications should be written such that locks are not accessed with LOCKSET or LOCKCLR unless they are currently checked out.

See About Locks, page 122, and Suggested Rules for Locks, page 123 for information on the typical use of locks and the LOCKxxx commands.

Note that user-defined resources are not actually locked by either the Hub or the checked-out lock. The lock feature only provides a means for objects to cooperatively lock those resources. It's up to the objects themselves to decide on, and abide by, the rules of lock use and what resource(s) will be governed by them. Additionally, the Hub does not directly assign a lock to the cog that called LOCKNEW, rather it simply marks it as being "checked out" by a cog; any other cog can "return" locks to the pool of available locks. Also, any cog can access any lock through the LOCKCLR and LOCKSET commands even if those locks were never checked out. Doing such things is generally not recommended because of the havoc it can cause with other, well-behaved objects in the application.

See LOCKNEW, page 122; LOCKCLR, page 120; and LOCKSET, page126 for more information.

LOCKSET

Command: Set lock to true and get its previous state.

((PUB ¦ PRI))
  LOCKSET (*ID*)

Returns: Previous state of lock (TRUE or FALSE).

- *ID* is the ID (0 – 7) of the lock to set to TRUE.

Explanation

LOCKSET is one of four lock commands (LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR) used to manage resources that are user-defined and deemed mutually exclusive. LOCKSET sets lock *ID* to TRUE and retrieves the previous state of that lock (TRUE or FALSE).

See About Locks, page 122, and Suggested Rules for Locks, page 123 for information on the typical use of locks and the LOCKxxx commands.

The following assumes that a cog (either this one or another) has already checked out a lock using LOCKNEW and shared the *ID* with this cog, which saved it as SemID. It also assumes this cog has an array of longs called LocalData.

```
PUB ReadResource | Idx
  repeat until not lockset(SemID) 'wait until we lock the resource
  repeat Idx from 0 to 9          'read all 10 longs of resource
    LocalData[Idx] := long[Idx]
  lockclr(SemID)                  'unlock the resource

PUB WriteResource | Idx
  repeat until not lockset(SemID) 'wait until we lock the resource
  repeat Idx from 0 to 9          'write all 10 longs to resource
    long[Idx] := LocalData[Idx]
  lockclr(SemID)                  'unlock the resource
```

Both of these methods, ReadResource and WriteResource, follow the same rules before and after accessing the resource. First, they wait indefinitely at the first REPEAT loop until it has locked the resource; i.e., it has successfully "set" the associated lock. If LOCKSET returns TRUE, the condition "until not lockset…" is false, meaning that some other cog is currently accessing the resource, so that first REPEAT loop tries again. If LOCKSET returns FALSE, the

condition "until not lockset…" is true, meaning we have "locked the resource" and the first REPEAT loop ends. The second REPEAT loop in each method reads or writes the resource, via the long[Idx] and LocalData[Idx] statements. The last line of each method, lockclr(SemID), clears the resource's associated lock to FALSE, logically unlocking or releasing the resource for others to use.

See LOCKNEW, page 122; LOCKRET, page 125; and LOCKCLR, page 120 for more information.

LONG

Designator: Declare long-sized symbol, long aligned/sized data, or read/write a long of main memory.

```
VAR
  LONG Symbol {[Count]}
```

```
DAT
  {Symbol} LONG Data {[Count]}
```

```
((PUB | PRI))
  LONG [BaseAddress] {[Offset]}
```

- ***Symbol*** is the desired name for the variable (Syntax 1) or data block (Syntax 2).
- ***Count*** is an optional expression indicating the number of long-sized elements for *Symbol* (Syntax 1) or the number of long-sized entries of *Data* (Syntax 2) to store in a data table.
- ***Data*** is a constant expression or comma-separated list of constant expressions.
- ***BaseAddress*** is an expression describing the long-aligned address of main memory to read or write. If *Offset* is omitted, *BaseAddress* is the actual address to operate on. If *Offset* is specified, *BaseAddress* + *Offset* * 4 is the actual address to operate on.
- ***Offset*** is an optional expression indicating the offset from *BaseAddress* to operate on. *Offset* is in units of longs.

Explanation

LONG is one of three multi-purpose declarations (BYTE, WORD, and LONG) that declares or operates on memory. LONG can be used to:

1) declare a long-sized (32-bit) symbol or a multi-long symbolic array in a VAR block, or
2) declare long-aligned, and/or long-sized, data in a DAT block, or
3) read or write a long of main memory at a base address with an optional offset.

Range of Long

Memory that is long-sized (32 bits) can contain a value that is one of 2^{32} possible combinations of bits (i.e., one of 4,294,967,296 combinations). The Spin language performs all mathematic operations using 32-bit signed math, meaning every long value is considered to be in the range -2,147,483,648 to +2,147,483,647. However, the actual numeric value contained within a long is subject to how a computer and user interpret it. In Propeller Assembly a long value can be treated as both signed and unsigned.

Long Variable Declaration (Syntax 1)

In VAR blocks, syntax 1 of LONG is used to declare global, symbolic variables that are either long-sized, or are any array of longs. For example:

```
VAR
  long  Temp                  'Temp is a long (2 words, 4 bytes)
  long  List[25]              'List is a long array
```

The above example declares two variables (symbols), Temp and List. Temp is simply a single, long-sized variable. The line under the Temp declaration uses the optional *Count* field to create an array of 25 long-sized variable elements called List. Both Temp and List can be accessed from any PUB or PRI method within the same object that this VAR block was declared; they are global to the object. An example of this is below.

```
PUB SomeMethod
  Temp := 25_000_000          'Set Temp to 25,000,000
  List[0] := 500_000          'Set first element of List to 500,000
  List[1] := 9_000            'Set second element of List to 9,000
  List[24] := 60              'Set last element of List to 60
```

For more information about using LONG in this way, refer to the VAR section's Variable Declarations (Syntax 1) on page 210, and keep in mind that LONG is used for the *Size* field in that description.

Long Data Declaration (Syntax 2)

In DAT blocks, syntax 2 of LONG is used to declare long-aligned, and/or long-sized data that is compiled as constant values in main memory. DAT blocks allow this declaration to have an optional symbol preceding it, which can be used for later reference (see DAT, page 99). For example:

```
DAT
  MyData  long  640_000, $BB50              'Long-aligned/sized data
  MyList  byte  long $FF995544, long 1_000  'Byte-aligned/long-sized
```

The above example declares two data symbols, MyData and MyList. MyData points to the start of long-aligned and long-sized data in main memory. MyData's values, in main memory, are 640,000 and $0000BB50, respectively. MyList uses a special DAT block syntax of LONG that creates a byte-aligned but long-sized set of data in main memory. MyList's values, in main memory, are $FF995544 and 1,000, respectively. When accessed a byte at a time, MyList

contains $44, $55, $99, $FF, 232 and 3, 0 and 0 since the data is stored in little-endian format.

Note: MyList could have been defined as word-aligned, long-sized data if the "byte" reference were replaced with "word".

This data is compiled into the object and resulting application as part of the executable code section and may be accessed using the read/write form, syntax 3, of LONG (see below). For more information about using LONG in this way, refer to the DAT section'sDeclaring Data(Syntax 1) on page 100, and keep in mind that LONG is used for the *Size* field in that description.

NEW Data items may be repeated by using the optional *Count* field. For example:

```
DAT
  MyData      long  640_000, $BB50[3]
```

The above example declares a long-aligned, long-sized data table, called MyData, consisting of the following four values: 640000, $BB50, $BB50, $BB50. There were three occurrences of $BB50 due to the [3] in the declaration immediately after it.

IMPROVED

Reading/Writing Longs of Main Memory (Syntax 3)

In PUB and PRI blocks, syntax 3 of LONG is used to read or write long-sized values of main memory. This is done by writing expressions that refer to main memory using the form: long[*BaseAddress*][*Offset*]. Here's an example.

```
PUB MemTest | Temp
  Temp := LONG[@MyData][1]                  'Read long value
  long[@MyList][0] := Temp + $01234567      'Write long value

DAT
  MyData  long 640_000, $BB50               'Long-sized/aligned data
  MyList  byte long $FF995544, long 1_000   'Byte-sized/aligned
                                            'long data
```

In this example, the DAT block (bottom of code) places its data in memory as shown in Figure 2-2. The first data element of MyData is placed at memory address $18. The last data element of MyData is placed at memory address $1C, with the first element of MyList immediately following it at $20. Note that the starting address ($18) is arbitrary and is likely to change as the code is modified or the object itself is included in another application.

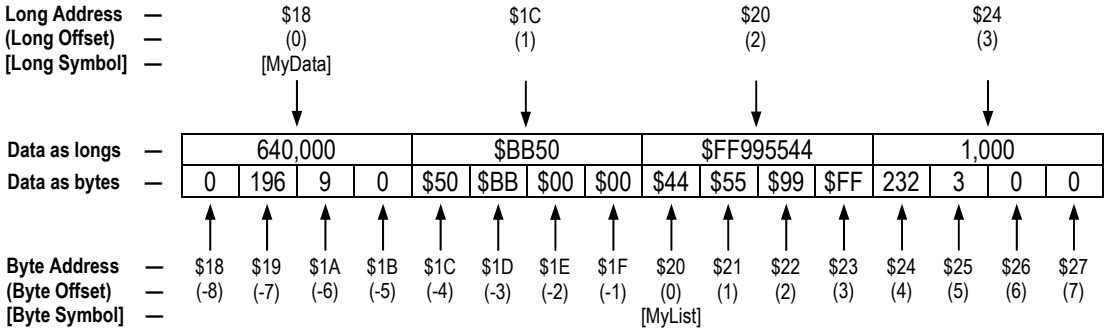


Figure 2-2: Main Memory Long-Sized Data Structure and Addressing

Near the top of the code, the first executable line of the `MemTest` method, `Temp := long[@MyData][1]`, reads a long-sized value from main memory. It sets local variable `Temp` to $BB50; the value read from main memory address $1C. The address $1C was determined by the address of the symbol `MyData` ($18) plus long offset 1 (4 bytes). The following progressive simplification demonstrates this.

```
long[@MyData][1] ➡ long[$18][1] ➡ long[$18 + (1*4)] ➡ long[$1C]
```

The next line, long[@MyList][0] := Temp + $01234567, writes a long-sized value to main memory. It sets the value at main memory address $20 to $012400BC. The address $20 was calculated from the address of the symbol MyList ($20) plus long offset 0 (0 bytes).

```
long[@MyList][0] ➡ long[$20][0] ➡ long[$20 + (0*4)] ➡ long[$20]
```

The value $012400BC was derived from the current value of `Temp` plus $012400BC; $BB50 + $01234567 equals $012400BC.

NEW

Addressing Main Memory

As Figure 2-2 suggests, main memory is really just a set of contiguous bytes (see "data as bytes" row) that can also be read as longs (4-byte sets) when done properly. In fact, the above example shows that even the addresses are calculated in terms of bytes. This concept is a consistent theme for any commands that use addresses.

Main memory is ultimately addressed in terms of bytes regardless of the size of value you are accessing; byte, word, or long. This is advantageous when thinking about how bytes, words,

and longs relate to each other, but it may prove problematic when thinking of multiple items of a single size, like longs.

For this reason, the LONG designator has a very handy feature to facilitate addressing from a long-centric perspective. Its *BaseAddress* field when combined with the optional *Offset* field operates in a base-aware fashion.

Imagine accessing longs of memory from a known starting point (the *BaseAddress*). You may naturally think of the next long or longs as being a certain distance from that point (the *Offset*). While those longs are indeed a certain number of "bytes" beyond a given point, it's easier to think of them as a number of "longs" beyond a point (i.e., the 4th long, rather than the long that starts beyond the 12th byte). The LONG designator treats it properly by taking the *Offset* value (units of longs), multiplies it by 4 (number of bytes per long), and adds that result to the *BaseAddress* to determine the correct long of memory to read. It also clears the lowest two bits of *BaseAddress* to ensure the address referenced is a long-aligned one.

So, when reading values from the MyData list, long[@MyData][0] reads the first long value and long[@MyData][1] reads the second.

If the *Offset* field were not used, the above statements would have to be something like long[@MyData], and long[@MyData+4], respectively. The result is the same, but the way it's written may not be as clear.

For more explanation of how data is arranged in memory, see the DAT section's Declaring Data(Syntax 1) on page 100.

NEW

An Alternative Memory Reference

There is yet another way to access the data from the code example above; you could reference the data symbols directly. For example, these statements read the first two longs of the MyData list:

```
Temp := MyData[0]
Temp := MyData[1]
```

So why wouldn't you just use direct symbol references all the time? Consider the following case:

```
Temp := MyList[0]
Temp := MyList[1]
```

Referring back to the example code above Figure 2-2 you might expect these two statements to read the first and second longs of MyList; $FF995544 and 1000, respectively. Instead, it reads the first and second "bytes" of MyList, $44 and $55, respectively.

What happened? Unlike MyData, the MyList entry is defined in the code as byte-sized and byte-aligned data. The data does indeed consist of long-sized values, because each element is preceded by LONG, but since the symbol for the list is declared as byte-sized, all direct references to it will return individual bytes.

However, the LONG designator can be used instead, since the list also happens to be long-aligned because of its position following MyData.

```
Temp := long[@MyList][0]
Temp := long[@MyList][1]
```

The above reads the first long, $FF995544, followed by the second long, 1000, of MyList. This feature is very handy should a list of data need to be accessed as both bytes and longs at various times in an application.

NEW

Other Addressing Phenomena

Both the LONG and direct symbol reference techniques demonstrated above can be used to access any location in main memory, regardless of how it relates to defined data. Here are some examples:

```
Temp := long[@MyList][-1]  'Read last long of MyData (before MyList)
Temp := long[@MyData][2]   'Read first long of MyList (after MyData)
Temp := MyList[-8]         'Read first byte of MyData
Temp := MyData[-2]         'Read long that is two longs before MyData
```

These examples read beyond the logical borders (start point or end point) of the lists of data they reference. This may be a useful trick, but more often it's done by mistake; be careful when addressing memory, especially if you're writing to that memory.

LONGFILL

Command: Fill longs of main memory with a value.

((PUB ¦ PRI))
  LONGFILL (*StartAddress*, *Value*, *Count*)

- ***StartAddress*** is an expression indicating the location of the first long of memory to fill with *Value*.
- ***Value*** is an expression indicating the value to fill longs with.
- ***Count*** is an expression indicating the number of longs to fill, starting with *StartAddress*.

Explanation

LONGFILL is one of three commands (BYTEFILL, WORDFILL, and LONGFILL) used to fill blocks of main memory with a specific value. LONGFILL fills *Count* longs of main memory with *Value*, starting at location *StartAddress*.

Using LONGFILL

LONGFILL is a great way to clear large blocks of long-sized memory. For example:

```
VAR
  long   Buff[100]

PUB Main
  longfill(@Buff, 0, 100)     'Clear Buff to 0
```

The first line of the Main method, above, clears the entire 100-long (400-byte) Buff array to all zeros. LONGFILL is faster at this task than a dedicated REPEAT loop is.

LONGMOVE

Command: Copy longs from one region to another in main memory.

((PUB ¦ PRI))
  LONGMOVE (*DestAddress*, *SrcAddress*, *Count*)

- ***DestAddress*** is an expression specifying the main memory location to copy the first long of source to.
- ***SrcAddress*** is an expression specifying the main memory location of the first long of source to copy.
- ***Count*** is an expression indicating the number of longs of the source to copy to the destination.

Explanation

LONGMOVE is one of three commands (BYTEMOVE, WORDMOVE, and LONGMOVE) used to copy blocks of main memory from one area to another. LONGMOVE copies *Count* longs of main memory starting from *SrcAddress* to main memory starting at *DestAddress*.

Using LONGMOVE

LONGMOVE is a great way to copy large blocks of long-sized memory. For example:

```
VAR
  long   Buff1[100]
  long   Buff2[100]

PUB Main
  longmove(@Buff2, @Buff1, 100)     'Copy Buff1 to Buff2
```

The first line of the Main method, above, copies the entire 100-long (400-byte) Buff1 array to the Buff2 array. LONGMOVE is faster at this task than a dedicated REPEAT loop is.

LOOKDOWN, LOOKDOWNZ

Command: Get the index of a value in a list.

((PUB ¦ PRI))
  LOOKDOWN (*Value* : *ExpressionList*)

((PUB ¦ PRI))
  LOOKDOWNZ (*Value* : *ExpressionList*)

Returns: One-based index position (LOOKDOWN) or a zero-based index position (LOOKDOWNZ) of *Value* in *ExpressionList*, or 0 if *Value* not found.

- ***Value*** is an expression indicating the value to find in *ExpressionList*.
- ***ExpressionList*** is a comma-separated list of expressions. Quoted strings of characters are also allowed; they are treated as a comma-separated list of characters.

Explanation

LOOKDOWN and LOOKDOWNZ are commands that retrieve indexes of values from a list of values. LOOKDOWN returns the one-based index position (1..N) of *Value* from *ExpressionList*. LOOKDOWNZ is just like LOOKDOWN except it returns the zero-based index position (0..N−1). For both commands, if *Value* is not found in *ExpressionList* then 0 is returned.

Using LOOKDOWN or LOOKDOWNZ

LOOKDOWN and LOOKDOWNZ are useful for mapping a set of non-contiguous numbers (25, -103, 18, etc.) to a set of contiguous numbers (1, 2, 3, etc. –or– 0, 1, 2, etc.) where no algebraic expression can be found to do so concisely. The following example assumes Print is a method created elsewhere.

```
PUB ShowList | Index
  Print(GetIndex(25))
  Print(GetIndex(300))
  Print(GetIndex(2510))
  Print(GetIndex(163))
  Print(GetIndex(17))
  Print(GetIndex(8000))
  Print(GetIndex(3))

PUB GetIndex(Value): Index
  Index := lookdown(Value: 25, 300, 2_510, 163, 17, 8_000, 3)
```

The GetIndex method in this example uses LOOKDOWN to find Value and returns the index where it was found in the *ExpressionList*, or 0 if not found. The ShowList method calls GetIndex repeatedly with different values and prints the resulting index on a display. Assuming Print is a method that displays a value, this example will print 1, 2, 3, 4, 5, 6 and 7 on a display.

If LOOKDOWNZ were used instead of LOOKDOWN this example would print 0, 1, 2, 3, 4, 5, and 6 on a display.

If *Value* is not found, LOOKDOWN, or LOOKDOWNZ, returns 0. So if one of the lines of the ShowList method was, Print(GetIndex(50)), the display would show 0 at the time it was executed.

If using LOOKDOWNZ, keep in mind that it may return 0 if either *Value* was not found or *Value* is at index 0. Make sure this will not cause an error in your code or use LOOKDOWN instead.

LOOKUP, LOOKUPZ

Command: Get value from an indexed position within a list.

((PUB ¦ PRI))
LOOKUP (***Index*** : ***ExpressionList***)

((PUB ¦ PRI))
LOOKUPZ (***Index*** : ***ExpressionList***)

Returns: Value at the one-based *Index* position (LOOKUP) or zero-based *Index* position (LOOKUPZ) of *ExpressionList*, or 0 if out-of-range.

- ***Index*** is an expression indicating the position of the desired value in *ExpressionList*. For LOOKUP, *Index* is one-based (1..N). For LOOKUPZ, *Index* is zero-based (0..N-1).
- ***ExpressionList*** is a comma-separated list of expressions. Quoted strings of characters are also allowed; they are treated as a comma-separated list of characters.

Explanation

LOOKUP and LOOKUPZ are commands that retrieve entries from a list of values. LOOKUP returns the value from *ExpressionList* that is located in the one-based position (1..N) given by *Index*. LOOKUPZ is just like LOOKUP except it uses a zero-based *Index* (0..N-1). For both commands, if *Index* is out of range then 0 is returned.

Using LOOKUP or LOOKUPZ

LOOKUP and LOOKUPZ are useful for mapping a contiguous set of numbers (1, 2, 3, etc. –or– 0, 1, 2, etc.) to a set of non-contiguous numbers (45, -103, 18, etc.) where no algebraic expression can be found to do so concisely. The following example assumes Print is a method created elsewhere.

```
PUB ShowList | Index, Temp
  repeat Index from 1 to 7
    Temp := lookup(Index: 25, 300, 2_510, 163, 17, 8_000, 3)
    Print(Temp)
```

This example looks up all the values in LOOKUP's *ExpressionList* and displays them. The REPEAT loop counts with Index from 1 to 7. Each iteration of the loop, LOOKUP uses Index to retrieve a value from its list. If Index equals 1, the value 25 is returned. If Index equals 2, the value 300 is returned. Assuming Print is a method that displays the value of Temp, this example will print 25, 300, 2510, 163, 17, 8000 and 3 on a display.

If LOOKUPZ is used, the list is zero-based (0..N-1) instead of one-based; an Index of 0 returns 25, Index of 1 returns 300, etc.

If *Index* is out of range 0 is returned. So, for LOOKUP, if the REPEAT statement went from 0 to 8, instead of 1 to 7, this example would print 0, 25, 300, 2510, 163, 17, 8000, 3 and 0 on a display.

NEXT

Command: Skip remaining statements of REPEAT loop and continue with the next loop iteration.

((PUB ¦ PRI))
  NEXT

Explanation

NEXTis one of two commands (NEXT and QUIT) that affect REPEAT loops. NEXT causes any further statements in the REPEAT loop to be skipped and the next iteration of the loop to be started thereafter.

Using NEXT

NEXT is typically used as an exception case, in a conditional statement, in REPEAT loops to move immediately to the next iteration of the loop. For example, assume that X is a variable created earlier and Print() is a method created elsewhere that prints a value on a display:

```
repeat X from 0 to 9          'Repeat 10 times
  if X == 4
    next                      'Skip if X = 4
  byte[$7000][X] := 0         'Clear RAM locations
  Print(X)                    'Print X on screen
```

The above code iteratively clears RAM locations and prints the value of X on a display, but with one exception. If X equals 4, the IF statement executes the NEXT command which causes the loop to skip remaining lines and go right to the next iteration. This has the effect of clearing RAM locations $7000 through $7003 and locations $7005 through $7009 and printing 0, 1, 2, 3, 5, 6, 7, 8, 9 on the display.

The NEXT command can only be used within a REPEAT loop; an error will occur otherwise.

OBJ

Designator: Declare an Object Block.

OBJ
Symbol ⟨[*Count*]⟩: "*ObjectName*" ⟨ ↳ *Symbol* ⟨[*Count*]⟩: "*ObjectName*"⟩...

- ***Symbol*** is the desired name for the object symbol.
- ***Count*** is an optional expression, enclosed in brackets, that indicates this is an array of objects, with *Count* number of elements. When later referencing these elements, they begin with element 0 and end with element *Count*-1.
- ***ObjectName*** is the filename, without extension, of the desired object. Upon compile, an object with this filename is searched for in the editor tabs, the working directory and the library directory. The object name can contain any valid filename characters; disallowed characters are \, /, :, *, ?, ", <, >, and |.

Explanation

The Object Block is a section of source code that declares which objects are used and the object symbols that represent them. This is one of six special declarations (CON, VAR, OBJ, PUB, PRI, and DAT) that provide inherent structure to the Spin language.

Object declarations begin with OBJ on a line by itself followed by one or more declarations. OBJ must start in column 1 (the leftmost column) of the line it is on and we recommend the lines following be indented by at least one space. For example:

```
OBJ
  Num  : "Numbers"
  Term : "TV_Terminal"
```

This example defines Num as an object symbol of type "Numbers" and Term as an object symbol of type "TV_Terminal". Public and Private methods can then refer to these objects using the object symbols as in the following example.

```
PUB Print | S
  S := Num.ToStr(LongVal, Num#DEC)
  Term.Str(@S)
```

This public method, Print, calls the Numbers' ToStr method and also the TV_Terminal's Str method. It does this by using the Num and Term object symbols followed by the Object-

Method reference symbol (a period '.') and finally the name of the method to call. Num.ToStr, for instance, calls the Numbers object's public ToStr method. Term.Str calls the TV_Terminal's public Str method. In this case the Num.ToStr has two parameters, in parentheses, and Term.Str has one parameter.

Also notice that the second parameter of the Num.ToStr call is Num#DEC. The # symbol is the Object-Constant reference symbol; it gives access to an object's constants. In this case, Num#DEC refers to the DEC (decimal format) constant in the Numbers object.

See Object-Method Reference '.' and Object-Constant Reference '#'> in Table 2-16: Symbols on page 207 for more information.

Multiple instances of an object can be declared with the same object symbol using array syntax and can be accessed similar to arrays as well. For example:

```
OBJ
  PWM[2] : "PWM"
PUB GenPWM
  PWM[0].Start
  PWM[1].Start
```

This example declares PWM as an array of two objects (two instances of the same object). The object itself just happens to be called "PWM" as well. The public method, GenPWM, calls the Start method of each instance using indexes 0 and 1 with the object symbol array, PWM.

Both instances of the PWM object are compiled into the application such that there is one copy of its program code (PUBs, PRIs, and DATs) and two copies of its variable blocks (VARs). This is because, for each instance, the code is the same but each instance needs its own variable space so it can operate independent of the other.

An important point to consider with multiple instances of an object is that there is only one copy of its DAT block because it may contain Propeller Assembly code. DAT blocks can also contain initialized data and regions set aside for workspace purposes, all with symbolic names. Since there is only one copy of it for multiple instances of an object, that area is shared among all instances. This provides a convenient way to create shared memory between multiple instances of a particular object.

Scope of Object Symbols

Object symbols defined in Object Blocks are global to the object in which they are defined but are not available outside of that object. This means that these object symbols can be accessed directly from anywhere within the object but their name will not conflict with symbols defined in other parent or child objects.

Operators

The Propeller chip features a powerful set of math and logic operators. A subset of these operators is supported by the Propeller Assembly language; however, since the Spin language has a use for every form of operator supported by the Propeller, this section describes every operator in detail. Please see the Operators section on page 325 for a list of operators available in Propeller Assembly.

Expression Workspace

The Propeller is a 32-bit device and, unless otherwise noted, expressions are always evaluated using 32-bit, signed integer math. This includes intermediate results as well. If any intermediate result overflows a 32-bit signed integer (above 2,147,483,647 or below -2,147,483,648), the final result of the expression will not be as expected. A workspace of 32 bits provides lots of room for intermediate results but it is still wise to keep overflow possibilities in mind.

If mathematic truncation is an issue, or if an expression requires real numbers rather than integers, floating-point support can help. The compiler supports 32-bit floating-point values and constant expressions with many of the same math operators as it does for integer constant expressions. Note that this is for constant expressions only, not run time variable expressions. For floating-point run-time expressions, the Propeller chip provides support through the FloatMath object supplied with the software installation. See Constant Assignment '=', page 148; FLOAT, page 108; ROUND, page 198; and TRUNC, page 209, as well as the FloatMath and FloatString objects for more information

Operator Attributes

The operators have the following important attributes, each of which is shown in the following two tables and further explained afterwards:

- Unary / Binary
- Normal / Assignment
- Constant and/or Variable Expression
- Level of Precedence

Table 2-9: Math and Logic Operators					
Operator	Assignment Usage	Constant Expressions[1]		Is Unary	Description, Page Number
		Integer	Float		
=	always	n/a[1]	n/a[1]		Constant assignment (CON blocks only), 148
:=	always	n/a[1]	n/a[1]		Variable assignment (PUB/PRI blocks only), 149
+	+=	✓	✓		Add, 149
+	never	✓	✓	✓	Positive (+X); unary form of Add, 150
-	-=	✓	✓		Subtract, 150
-	if solo	✓	✓	✓	Negate (-X); unary form of Subtract, 150
--	always			✓	Pre-decrement (--X) or post-decrement (X--), 151
++	always			✓	Pre-increment (++X) or post-increment (X++), 152
*	*=	✓	✓		Multiply and return lower 32 bits (signed), 153
**	**=	✓			Multiply and return upper 32 bits (signed), 153
/	/=	✓	✓		Divide (signed), 154
//	//=	✓			Modulus (signed), 154
#>	#>=	✓	✓		Limit minimum (signed), 155
<#	<#=	✓	✓		Limit maximum (signed), 155
^^	if solo	✓	✓	✓	Square root, 156
||	if solo	✓	✓	✓	Absolute value, 156
~	always			✓	Sign-extend from bit 7 (~X) or post-clear to 0 (X~); all bits low, 156
~~	always			✓	Sign-extend from bit 15 (~~X) or post-set to -1 (X~~); all bits high, 157
~>	~>=	✓			Shift arithmetic right, 158
?	always			✓	Random number forward (?X) or reverse (X?), 159
|<	if solo	✓		✓	Bitwise: Decode value (0 - 31) into single-high-bit long, 160
>|	if solo	✓		✓	Bitwise: Encode long into value (0 - 32) as high-bit priority, 160
<<	<<=	✓			Bitwise: Shift left, 161
>>	>>=	✓			Bitwise: Shift right, 161
<-	<-=	✓			Bitwise: Rotate left, 162
->	->=	✓			Bitwise: Rotate right, 162
><	><=	✓			Bitwise: Reverse, 163
&	&=	✓			Bitwise: AND, 164
|	|=	✓			Bitwise: OR, 165
^	^=	✓			Bitwise: XOR, 165
!	if solo	✓		✓	Bitwise: NOT, 166
AND	AND=	✓	✓		Boolean: AND (promotes non-0 to -1), 167
OR	OR=	✓	✓		Boolean: OR (promotes non-0 to -1), 168
NOT	if solo	✓	✓	✓	Boolean: NOT (promotes non-0 to -1), 168
==	===	✓	✓		Boolean: Is equal, 169
<>	<>=	✓	✓		Boolean: Is not equal, 170
<	<=	✓	✓		Boolean: Is less than (signed), 171
>	>=	✓	✓		Boolean: Is greater than (signed), 171
=<	=<=	✓	✓		Boolean: Is equal or less (signed), 171
=>	=>=	✓	✓		Boolean: Is equal or greater (signed), 172
@	never	✓		✓	Symbol address, 173
@@	never			✓	Object address plus symbol, 173

[1] **Assignment forms of operators are not allowed in constant expressions.**

Table 2-10: Operator Precedence Levels			
Level	**Notes**	**Operators**	**Operator Names**
Highest (0)	Unary	--, ++, ~, ~~, ?, @, @@	Inc/Decrement, Clear, Set, Random, Symbol/Object Address
1	Unary	+, -, ^^, ||, |<, >|, !	Positive, Negate, Square Root, Absolute, Decode, Encode, Bitwise NOT
2		->, <-, >>, <<, ~>, ><	Rotate Right/Left, Shift Right/Left, Shift Arithmetic Right, Reverse
3		&	Bitwise AND
4		|, ^	Bitwise OR, Bitwise XOR
5		*, **, /, //	Multiply-Low, Multiply-High, Divide, Modulus
6		+, -	Add, Subtract
7		#>, <#	Limit Minimum/Maximum
8		<, >, <>, ==, =<, =>	Boolean: Less/Greater Than, Not Equal, Equal, Equal or Less/Greater
9	Unary	NOT	Boolean NOT
10		AND	Boolean AND
11		OR	Boolean OR
Lowest (12)		=, :=, all other assignments	Constant/Variable Assignment, assignment forms of Binary Operators

Unary / Binary

Each operator is either unary or binary in nature. Unary operators are those that operate on only one operand. For example:

```
!Flag     ' bitwise NOT of Flag
^^Total   ' square root of Total
```

Binary operators are those that operate on two operands. For example:

```
X + Y     ' add X and Y
Num << 4  ' shift Num left 4 bits
```

Note that the term "binary operator" means "two operands," and has nothing to do with binary digits. To distinguish operators whose function relates to binary digits, we'll use the term "bitwise" instead.

Normal / Assignment

Normal operators, like Add '**+**' and Shift Left '**<<**', operate on their operand(s) and provide the result for use by the rest of the expression, without affecting the operand or operands themselves. Those that are assignment operators, however, write their result to either the variable they operated on (unary), or to the variable to their immediate left (binary), in addition to providing the result for use by the rest of the expression.

Here are assignment operator examples:

```
Count++       ' (Unary) evaluate Count + 1
              ' and write result to Count
Data >>= 3    ' (Binary) shift Data right 3 bits
              ' and write result to Data
```

Binary operators have special forms that end in equal '=' to make them assignment operators. Unary operators do not have a special assignment form; some always assign while others assign only in special situations. See Table 2-9 above and the operator's explanation, for more information.

NEW

Most assignment operators can only be used within methods (PUB and PRI blocks). The only exception is the constant assignment operator '=' which can only be used in CON blocks.

Constant and/or Variable Expression

Operators which have the integer-constant-expression attribute can be used both at run time in variable expressions, and at compile time in constant expressions. Operators that have the float-constant-expression attribute can be used in compile-time constant expressions. Operators without either of the constant-expression attributes can only be used at run time in variable expressions. Most operators have a normal, non-assignment form that allows them to be used in both constant and variable expressions.

Level of Precedence

Each operator has an assigned level of precedence that determines when it will take action in relation to other operators within the same expression. For example, it is commonly known that Algebraic rules require multiply and divide operations to be performed before add and subtract operations. The multiply and divide operators are said to have a "higher level of precedence" than add and subtract. Additionally, multiply and divide are commutable; both are on the same precedence level, so their operations result in the same value regardless of the order it is performed (multiply first, then divide, or vice versa). Commutative operators are always evaluated left to right except where parentheses override that rule.

The Propeller chip applies the order-of-operations rules as does Algebra: expressions are evaluated left-to-right, except where parentheses and differing levels of precedence exist.

Following these rules, the Propeller will evaluate:

$$X = 20 + 8 * 4 - 6 / 2$$

...to be equal to 49; that is, 8 * 4 = 32, 6 / 2 = 3, and 20 + 32 – 3 = 49. If you wish the expression to be evaluated differently, use parentheses to enclose the necessary portions of the expression.

For example:

X = (20 + 8) * 4 – 6 / 2

This will evaluate the expression in parentheses first, the 20 + 8, causing the expression to now result in 109, instead of 49.

Table 2-10 indicates each operator's level of precedence from highest (level 0) to lowest (level 12). Operators with a higher precedence are performed before operators of a lower precedence; multiply before add, absolute before multiply, etc. The only exception is if parentheses are included; they override every precedence level.

Intermediate Assignments

The Propeller chip's expression engine allows for, and processes, assignment operators at intermediate stages. This is called "intermediate assignments" and it can be used to perform complex calculations in less code. For example, the following equation relies heavily on X, and X + 1.

```
X := X - 3 * (X + 1) / ||(X + 1)
```

The same statement could be rewritten, taking advantage of the intermediate assignment property of the increment operator:

```
X := X++ - 3 * X / ||X
```

Assuming `X` started out at -5, both of these statements evaluate to -2, and both store that value in `X` when done. The second statement, however, does it by relying on an intermediate assignment (the `X++` part) in order to simplify the rest of the statement. The Increment operator '**++**' is evaluated first (highest precedence) and increments X's -5 to -4. Since this is a "post increment" (see Increment, pre- or post- '++', page 152) it first returns `X`'s original value, -5, to the expression and then writes the new value, -4, to `X`. So, the "`X++ - 3`…" part of the expression becomes "-5 – 3…" Then the absolute, multiply, and divide operators are evaluated, but the value of `X` has been changed, so they use the new value, -4, for their operations:

-5 – 3 * -4 / ||-4 → -5 – 3 * -4 / 4 → -5 – 3 * -1 → -5 – -3 = -2

Occasionally, the use of intermediate assignments can compress multiple lines of expressions into a single expression, resulting in slightly smaller code size and slightly faster execution.

The remaining pages of this section further explain each math and logic operator shown in Table 2-9 in the same order shown.

Constant Assignment '='

The Constant Assignment operator is used only within CON blocks, to declare compile-time constants. For example,

```
CON
  _xinfreq = 4096000
  WakeUp   = %00110000
```

This code sets the symbol _xinfreq to 4,096,000 and the symbol WakeUp to %00110000. Throughout the rest of the program the compiler will use these numbers in place of their respective symbols. See CON, page 84.

These declarations are constant expressions, so many of the normal operators can be used to calculate a final constant value at compile time. For example, it may be clearer to rewrite the above example as follows:

```
CON
  _xinfreq   = 4096000
  Reset      = %00100000
  Initialize = %00010000
  WakeUp     = Reset & Initialize
```

Here, WakeUp is still set to %00110000 at compile time, but it is now more obvious to future readers that the WakeUp symbol contains the binary codes for a Reset and an Initialize sequence for that particular application.

The above examples create 32-bit signed integer constants; however, it is also possible to create 32-bit floating-point constants. To do so, the expression must be expressed as a floating-point value in one of three ways: 1) as an integer value followed by a decimal point and at least one digit, 2) as an integer with an E followed by an exponent value, or 3) both 1 and 2.

For example:

```
CON
  OneHalf = 0.5
  Ratio   = 2.0 / 5.0
  Miles   = 10e5
```

The above code creates three floating-point constants. OneHalf is equal to 0.5, Ratio is equal to 0.4 and Miles is equal to 1,000,000. Note that if Ratio were defined as 2 / 5 instead of 2.0 / 5.0, the expression would be treated as an integer constant and the result would be an integer constant equal to 0. For floating-point constant expressions, every value within the expression must be a floating-point value; you cannot mix integer and floating-point values like Ratio = 2 / 5.0. You can, however, use the FLOAT declaration to convert an integer value to a floating-point value, such as Ratio = FLOAT(2) / 5.0.

The Propeller compiler handles floating-point constants as a single-precision real number as described by the IEEE-754 standard. Single-precision real numbers are stored in 32 bits, with a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa (the fractional part). This provides approximately 7.2 significant decimal digits.

For run-time floating-point operations, the FloatMath and FloatString objects provide math functions compatible with single-precision numbers.

See FLOAT, page 108; ROUND, page 198; TRUNC, page 209, as well as the FloatMath and FloatString objects for more information.

Variable Assignment ':='

The Variable Assignment operator is used only within methods (PUB and PRI blocks), to assign a value to a variable. For example,

```
Temp := 21
Triple := Temp * 3
```

At run time this code would set the Temp variable equal to 21 and set Triple to 21 * 3, which is 63.

As with other assignment operators, the Variable Assignment operator can be used within expressions to assign intermediate results, such as:

```
Triple := 1 + (Temp := 21) * 3
```

This example first sets Temp to 21, then multiplies Temp by 3 and adds 1, finally assigning the result, 64, to Triple.

Add '+', '+='

The Add operator adds two values together. Add can be used in both variable and constant expressions. Example:

```
X := Y + 5
```

Add has an assignment form, +=, that uses the variable to its left as both the first operand and the result destination.

For example:

```
X += 10  'Short form of X := X + 10
```

Here, the value of X is added to 10 and the result is stored back in X. The assignment form of Add may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Positive '+' (unary form of Add)

Positive is the unary form of Add and can be used similar to Negate except that it is never an assignment operator. Positive is essentially ignored by the compiler, but is handy when the sign of operands is important to emphasize. For example:

```
Val := +2 - A
```

Subtract '-', '-='

The Subtract operator subtracts two values. Subtract can be used in both variable and constant expressions. Example:

```
X := Y - 5
```

Subtract has an assignment form, -=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X -= 10  'Short form of X := X - 10
```

Here, 10 is subtracted from the value of X and the result is stored back in X. The assignment form of subtract may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Negate '-' (unary form of Subtract)

Negate is the unary form of Subtract. Negate toggles the sign of the value on its right; a positive value becomes negative and a negative value becomes positive. For example:

```
Val := -2 + A
```

Negate becomes an assignment operator when it is the sole operator to the left of a variable on a line by itself. For example:

```
-A
```

This would negate the value of A and store the result back to A.

Decrement, pre- or post- '--'

The Decrement operator is a special, immediate operator that decrements a variable by one and assigns the new value to that same variable. It can only be used in run-time variable expressions. Decrement has two forms, pre-decrement and post-decrement, depending on which side of the variable it appears on. The pre-decrement form appears to the left of a variable and the post-decrement form appears to the right of a variable. This is extremely useful in programming since there are many situations that call for the decrementing of a variable right before or right after the use of that variable's value. For example:

```
Y := --X + 2
```

The above shows the pre-decrement form; it means "decrement before providing the value for the next operation". It decrements the value of X by one, writes that result to X and provides that result to the rest of the expression. If X started out as 5 in this example, --X would store 4 in X, then the expression, 4 + 2 is evaluated, finally writing the result, 6, into Y. After this statement, X equals 4 and Y equals 6.

```
Y := X-- + 2
```

The above shows the post-decrement form; it means "decrement after providing the value for the next operation". It provides the current value of X for the next operation in the expression, then decrements the value of X by one and writes that result to X. If X began as 5 in this example, X-- would provide the current value for the expression (5 + 2) to be evaluated later, then would store 4 in X. The expression 5 + 2 is then evaluated and the result, 7, is stored into Y. After this statement, X equals 4 and Y equals 7.

Since Decrement is always an assignment operator, the rules of Intermediate Assignments (see page 147) apply here. Assume X started out as 5 for the following examples.

```
Y := --X + X
```

Here, X would first be set to 4, then 4 + 4 is evaluated and Y is set to 8.

```
Y := X-- + X
```

Here, X's current value, 5, is saved for the next operation (the Add) and X itself is decremented to 4, then 5 + 4 is evaluated and Y is set to 9.

Increment, pre- or post- '++'

The Increment operator is a special, immediate operator that increments a variable by one and assigns the new value to that same variable. It can only be used in run-time variable expressions. Increment has two forms, pre-increment and post-increment, depending on which side of the variable it appears on. The pre-increment form appears to the left of a variable and the post-increment form appears to the right of a variable. This is extremely useful in programming since there are many situations that call for the incrementing of a variable right before or right after the use of that variable's value. For example:

```
Y := ++X - 4
```

The above shows the pre-increment form; it means "increment before providing the value for the next operation". It increments the value of X by one, writes that result to X and provides that result to the rest of the expression. If X started out as 5 in this example, ++X would store 6 in X, then the expression, 6 - 4 is evaluated, finally writing the result, 2, into Y. After this statement, X equals 6 and Y equals 2.

```
Y := X++ - 4
```

The above shows the post-increment form; it means "increment after providing the value for the next operation". It provides the current value of X for the next operation in the expression, then increments the value of X by one and writes that result to X. If X started out as 5 in this example, X++ would provide the current value for the expression (5 - 4) to be evaluated later, then would store 6 in X. The expression 5 - 4 is then evaluated and the result, 1, is stored into Y. After this statement, X equals 6 and Y equals 1.

Since Increment is always an assignment operator, the rules of Intermediate Assignments (see page 147) apply here. Assume X started out as 5 for the following examples.

```
Y := ++X + X
```

Here, X would first be set to 6, then 6 + 6 is evaluated and Y is set to 12.

```
Y := X++ + X
```

Here, X's current value, 5, is saved for the next operation (the Add) and X itself is incremented to 6, then 5 + 6 is evaluated and Y is set to 11.

Multiply, Return Low '*', '*='

This operator is also called Multiply-Low, or simply Multiply. It can be used in both variable and constant expressions. When used with variable expressions or integer constant expressions, Multiply Low multiplies two values together and returns the lower 32 bits of the 64-bit result. When used with floating-point constant expressions, Multiply Low multiplies two values together and returns the 32-bit single-precision floating-point result. Example:

```
X := Y * 8
```

Multiply-Low has an assignment form, ***=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X *= 20  'Short form of X := X * 20
```

Here, the value of X is multiplied by 20 and the lowest 32 bits of the result is stored back in X. The assignment form of Multiply-Low may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Multiply, Return High '**', '**='

This operator is also called Multiply-High. It can be used in both variable and integer constant expressions, but not in floating-point constant expressions. Multiply High multiplies two values together and returns the upper 32 bits of the 64-bit result. Example:

```
X := Y ** 8
```

If Y started out as 536,870,912 (2^{29}) then Y ** 8 equals 1; the value in the upper 32 bits of the result.

Multiply-High has an assignment form, ****=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X **= 20 'Short form of X := X ** 20
```

Here, the value of X is multiplied by 20 and the upper 32 bits of the result is stored back in X. The assignment form of Multiply-High may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Divide '/', '/='

Divide can be used in both variable and constant expressions. When used with variable expressions or integer constant expressions, it divides one value by another and returns the 32-bit integer result. When used with floating-point constant expressions, it divides one value by another and returns the 32-bit single-precision floating-point result. Example:

```
X := Y / 4
```

Divide has an assignment form, **/=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X /= 20  'Short form of X := X / 20
```

Here, the value of X is divided by 20 and the integer result is stored back in X. The assignment form of Divide may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Modulus '//', '//='

Modulus can be used in both variable and integer constant expressions, but not in floating-point constant expressions. Modulus divides one value by another and returns the 32-bit integer remainder. Example:

```
X := Y // 4
```

If Y started out as 5 then Y // 4 equals 1, meaning the division of 5 by 4 results in a real number whose fractional component equals ¼, or .25.

Modulus has an assignment form, **//=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X //= 20 'Short form of X := X // 20
```

Here, the value of X is divided by 20 and the 32-bit integer remainder is stored back in X. The assignment form of Modulus may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Limit Minimum '#>', '#>='

The Limit Minimum operator compares two values and returns the highest value. Limit Minimum can be used in both variable and constant expressions. Example:

```
X := Y - 5 #> 100
```

The above example subtracts 5 from Y and limits the result to a minimum value to 100. If Y is 120 then 120 – 5 = 115; it is greater than 100 so X is set to 115. If Y is 102 then 102 – 5 = 97; it is less than 100 so X is set to 100 instead.

Limit Minimum has an assignment form, **#>=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X #>= 50 'Short form of X := X #> 50
```

Here, the value of X is limited to a minimum value of 50 and the result is stored back in X. The assignment form of Limit Minimum may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Limit Maximum '<#', '<#='

The Limit Maximum operator compares two values and returns the lowest value. Limit Maximum can be used in both variable and constant expressions. Example:

```
X := Y + 21 <# 250
```

The above example adds 21 to Y and limits the result to a maximum value to 250. If Y is 200 then 200 + 21 = 221; it is less than 250 so X is set to 221. If Y is 240 then 240 + 21 = 261; it is greater than 250 so X is set to 250 instead.

Limit Maximum has an assignment form, **<#=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X <#= 50 'Short form of X := X <# 50
```

Here, the value of X is limited to a maximum value of 50 and the result is stored back in X. The assignment form of Limit Minimum may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Square Root '^^'

The Square Root operator returns the square root of a value. Square Root can be used in both variable and constant expressions. When used with variable expressions or integer constant expressions, Square Root returns the 32-bit truncated integer result. When used with floating-point constant expressions, Square Root returns the 32-bit single-precision floating-point result. Example:

```
X := ^^Y
```

Square Root becomes an assignment operator when it is the sole operator to the left of a variable on a line by itself. For example:

```
^^Y
```

This would store the square root of the value of Y back into Y.

Absolute Value '||'

The Absolute Value operator, also called Absolute, returns the absolute value (the positive form) of a number. Absolute Value can be used in both variable and constant expressions. When used with variable expressions or integer constant expressions, Absolute Value returns the 32-bit integer result. When used with floating-point constant expressions, Absolute Value returns the 32-bit single-precision floating-point result. Example:

```
X := ||Y
```

If Y is -15, the absolute value, 15, would be stored into X.

Absolute Value becomes an assignment operator when it is the sole operator to the left of a variable on a line by itself. For example:

```
||Y
```

This would store the absolute value of Y back into Y.

Sign-Extend 7 or Post-Clear '~'

This operator is a special, immediate operator that has a dual purpose depending on which side of the variable it appears on. It can only be used in run-time variable expressions. The Sign-Extend 7 form of the operator appears to the left of a variable and the Post-Clear form appears to the right of a variable.

The following is an example of the Sign-Extend 7 operator form:

```
Y := ~X + 25
```

The Sign-Extend 7 operator in this example extends the sign of the value, X in this case, from bit 7 up to bit 31. A 32-bit signed integer is stored in twos-complement form and the most significant bit (31) indicates the sign of the value (positive or negative). There may be times where calculations on simple data result in byte-sized values that should be treated as a signed integer in the range of -128 to +127. When you need to perform further calculations with those byte-sized values, use the Sign-Extend 7 operator to convert the number into the proper 32-bit signed integer form. In the above example, assume X represents the value -20, which in 8-bit twos-complement form is actually the value 236 (%11101100). The ~X portion of the expression extends the sign bit from bit 7 all the way up to bit 31, converting the number to the proper 32-bit twos-complement form of -20 (%11111111 11111111 11111111 11101100). Adding that sign-extended value to 25 results in 5, the intended result, whereas it would have resulted in 261 without the proper sign extension.

The following is an example of the Post-Clear operator form.

```
Y := X~ + 2
```

The Post-Clear operator in this example clears the variable to 0 (all bits low) after providing its current value for the next operation. In this example if X started out as 5, X~ would provide the current value for the expression (5 + 2) to be evaluated later, then would store 0 in X. The expression 5 + 2 is then evaluated and the result, 7, is stored into Y. After this statement, X equals 0 and Y equals 7.

Since Sign-Extend 7 and Post-Clear are always assignment operators, the rules of Intermediate Assignments apply to them (see page 147).

Sign-Extend 15 or Post-Set '~~'

This operator is a special, immediate operator that has a dual purpose depending on which side of the variable it appears on. It can only be used in run-time variable expressions. The Sign-Extend 15 form of the operator appears to the left of a variable and the Post-Set form appears to the right of a variable.

The following is an example of the Sign-Extend 15 operator form:

```
Y := ~~X + 50
```

The Sign-Extend 15 operator in this example extends the sign of the value, X in this case, from bit 15 up to bit 31. A 32-bit signed integer is stored in twos-complement form and the most significant bit (31) indicates the sign of the value (positive or negative). There may be

times where calculations on simple data result in word-sized values that should be treated as a signed integer in the range of -32768 to +32767. When you need to perform further calculations with those word-sized values, use the Sign-Extend 15 operator to convert the number into the proper 32-bit signed integer form. In the above example, assume X represents the value -300, which in 16-bit twos-complement form is actually the value 65,236 (%11111110 11010100). The ~~X portion of the expression extends the sign bit from bit 15 all the way up to bit 31, converting the number to the proper 32-bit twos-complement form of -300 (%11111111 11111111 11111110 11010100). Adding that sign-extended value to 50 results in -250, the intended result, whereas it would have resulted in 65,286 without the proper sign extension.

The following is an example of the Post-Set operator form.

```
Y := X~~ + 2
```

The Post-Set operator in this example sets the variable to -1 (all bits high) after providing its current value for the next operation. In this example if X started out as 6, X~~ would provide the current value for the expression (6 + 2) to be evaluated later, then would store -1 in X. The expression 6 + 2 is then evaluated and the result, 8, is stored into Y. After this statement, X equals -1 and Y equals 8.

Since Sign-Extend 15 and Post-Set are always assignment operators, the rules of Intermediate Assignments apply to them (see page 147).

Shift Arithmetic Right '~>', '~>='

The Shift Arithmetic Right operator is just like the Shift Right operator except that it maintains the sign, like a divide by 2, 4, 8, etc on a signed value. Shift Arithmetic Right can be used in variable and integer constant expressions, but not in floating-point constant expressions. Example:

```
X := Y ~> 4
```

The above example shifts Y right by 4 bits, maintaining the sign. If Y is -3200 (%11111111 11111111 11110011 10000000) then -3200 ~> 4 = -200 (%11111111 11111111 11111111 00111000). If the same operation had been done with the Shift Right operator instead, the result would have been 268,435,256 (%00001111 11111111 11111111 00111000).

Shift Arithmetic Right has an assignment form, **~>=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X ~>= 2  'Short form of X := X ~> 2
```

Here, the value of X is shifted right 2 bits, maintaining the sign, and the result is stored back in X. The assignment form of Shift Arithmetic Right may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Random '?'

The Random operator is a special, immediate operator that uses a variable's value as a seed to create a pseudo random number and assigns that number to the same variable. It can only be used in run-time variable expressions. Random has two forms, forward and reverse, depending on which side of the variable it appears on. The forward form appears to the left of a variable and the reverse form appears to the right of a variable.

Random generates pseudo-random numbers ranging from -2,147,483,648 to +2,147,483,647. It's called "pseudo-random" because the numbers appear random, but are really generated by a logic operation that uses a "seed" value as a tap into a sequence of over 4 billion essentially random numbers. If the same seed value is used again, the same sequence of numbers is generated. The Propeller chip's Random output is reversible; in fact, specifically it is a 32-bit maximum-length, four-tap LFSR (Linear Feedback Shift Register) with taps in both the LSB (Least Significant Bit, rightmost bit) and the MSB (Most Significant Bit, leftmost bit) allowing for bi-directional operation.

Think of the pseudo-random sequence it generates as simply a static list of over 4 billion numbers. Starting with a particular seed value and moving forward results in a list of a specific set of numbers. If, however, you took that last number generated and used it as the first seed value moving backward, you would end up with a list of the same numbers as before, but in the reverse order. This is handy in many applications.

Here's an example:

```
?X
```

The above shows the Random forward form; it uses X's current value to retrieve the next pseudo-random number in the forward direction and stores that number back in X. Executing ?X again results in yet a different number, again stored back into X.

```
X?
```

The above shows the Random reverse form; it uses X's current value to retrieve the next pseudo-random number in the reverse direction and stores that number back in X. Executing X? again results in yet a different number, again stored back into X.

Since Random is always an assignment operator, the rules of Intermediate Assignments apply to it (see page 147).

Bitwise Decode '|<'

The Bitwise Decode operator decodes a value (0 – 31) into a 32-bit long value with a single bit set high corresponding to the bit position of the original value. Bitwise Decode can be used in variable and integer constant expressions, but not in floating-point constant expressions. Example:

```
Pin := |<PinNum
```

The above example sets Pin equal to the 32-bit value whose single high-bit corresponds to the position indicated by PinNum.

If PinNum is 3, Pin is set equal to %00000000 00000000 00000000 00001000.

If PinNum is 31, Pin is set equal to %10000000 00000000 00000000 00000000.

There are many uses for Bitwise Decode, but one of the most useful is to convert from an I/O pin number to the 32-bit pattern that describes that pin number in relation to the I/O registers. For example, Bitwise Decode is very handy for the mask parameter of the **WAITPEQ** and **WAITPNE** commands.

Bitwise Decode becomes an assignment operator when it is the sole operator to the left of a variable on a line by itself. For example:

```
|<PinNum
```

This would store the decoded value of PinNum back into PinNum.

Bitwise Encode '>|'

The Bitwise Encode operator encodes a 32-bit long value into the value (0 – 32) that represents the highest bit set, plus 1. Bitwise Encode can be used in variable and integer constant expressions, but not in floating-point constant expressions. Example:

```
PinNum := >|Pin
```

The above example sets PinNum equal to the number of the highest bit set in Pin, plus 1.

If Pin is %00000000 00000000 00000000 00000000, PinNum is set equal to 0; no bits are set.

If Pin is %00000000 00000000 00000000 10000000, PinNum is set equal to 8; bit 7 is set.

If Pin is %10000000 00000000 00000000 00000000, PinNum is set equal to 32; bit 31 is set.

If Pin is %00000000 00010011 00010010 00100000, PinNum is set equal to 21; bit 20 is the highest bit set.

Bitwise Shift Left '<<', '<<='

The Bitwise Shift Left operator shifts the bits of the first operand left by the number of bits indicated in the second operand. The original MSBs (leftmost bits) drop off and the new LSBs (rightmost bits) are set to zero. Bitwise Shift Left can be used in both variable and integer constant expressions, but not in floating-point constant expressions. Example:

```
X := Y << 2
```

If Y started out as:

%10000000 01110000 11111111 00110101

...the Bitwise Shift Left operator would shift that value left by two bits, setting X to:

%00000001 11000011 11111100 11010100

Since the nature of binary is base-2, shifting a value left is like multiplying that value by powers of two, 2^b, where b is the number of bits shifted.

Bitwise Shift Left has an assignment form, **<<=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X <<= 4   'Short form of X := X << 4
```

Here, the value of X is shifted left four bits and is stored back in X. The assignment form of Bitwise Shift Left may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Bitwise Shift Right '>>', '>>='

The Bitwise Shift Right operator shifts the bits of the first operand right by the number of bits indicated in the second operand. The original LSBs (rightmost bits) drop off and the new MSBs (leftmost bits) are set to zero. Bitwise Shift Right can be used in both variable and integer constant expressions, but not in floating-point constant expressions. Example:

```
X := Y >> 3
```

If Y started out as:

%10000000 01110000 11111111 00110101

...the Bitwise Shift Right operator would shift that value right by three bits, setting X to:

%00010000 00001110 00011111 11100110

Since the nature of binary is base-2, shifting a value right is like performing an integer divide of that value by powers of two, 2^b, where b is the number of bits shifted.

Bitwise Shift Right has an assignment form, **>>=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X >>= 2  'Short form of X := X >> 2
```

Here, the value of X is shifted right two bits and is stored back in X. The assignment form of Bitwise Shift Right may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Bitwise Rotate Left '<-', '<-='

The Bitwise Rotate Left operator is similar to the Bitwise Shift Left operator, except that the MSBs (leftmost bits) are rotated back around to the LSBs (rightmost bits). Bitwise Rotate Left can be used in both variable and integer constant expressions, but not in floating-point constant expressions. Example:

```
X := Y <- 4
```

If Y started out as:

%10000000 01110000 11111111 00110101

the Bitwise Rotate Left operator would rotate that value left by four bits, moving the original four MSBs to the four new LSBs, and setting X to:

%00000111 00001111 11110011 01011000

Bitwise Rotate Left has an assignment form, **<-=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X <-= 1  'Short form of X := X <- 1
```

Here, the value of X is rotated left one bit and is stored back in X. The assignment form of Bitwise Rotate Left may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Bitwise Rotate Right '->', '->='

The Bitwise Rotate Right operator is similar to the Bitwise Shift Right operator, except that the LSBs (rightmost bits) are rotated back around to the MSBs (leftmost bits). Bitwise Rotate Right can be used in both variable and integer constant expressions, but not in floating-point constant expressions. Example:

```
X := Y -> 5
```

If Y started out as:

%10000000 01110000 11111111 00110101

...the Bitwise Rotate Right operator would rotate that value right by five bits, moving the original five LSBs to the five new MSBs, and setting X to:

%10101100 00000011 10000111 11111001

Bitwise Rotate Right has an assignment form, **->=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X ->= 3   'Short form of X := X -> 3
```

Here, the value of X is rotated right three bits and is stored back in X. The assignment form of Bitwise Rotate Right may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Bitwise Reverse '><', '><='

IMPROVED

The Bitwise Reverse operator returns least-significant bits from the first operand in their reverse order. The total number of least-significant bits to be reversed is indicated by the second operand. All other bits to the left of the reversed bits are zeros in the result. Bitwise Reverse can be used in both variable and integer constant expressions, but not in floating-point constant expressions. Example:

```
X := Y >< 6
```

If Y started out as:

%10000000 01110000 11111111 00110101

...the Bitwise Reverse operator would return the six LSBs in reverse order with all other bits zero, setting X to:

%00000000 00000000 00000000 00101011

Bitwise Reverse has an assignment form, **><=**, that uses the variable to its left as both the first operand and the result destination. For example,

```
X ><= 8   'Short form of X := X >< 8
```

Here, the eight LSBs of the value of X are reversed, all other bits are set to zero and the result is stored back in X. The assignment form of Bitwise Reverse may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

NEW Note that specifying 0 as the number of bits to reverse is the same as specifying 32.

Bitwise AND '&', '&='

The Bitwise AND operator performs a bitwise AND of the bits of the first operand with the bits of the second operand. Bitwise AND can be used in both variable and integer constant expressions, but not in floating-point constant expressions.

Each bit of the two operands is subject to the following logic:

Table 2-11: Bitwise AND Truth Table

Bit States		Result
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```
X := %00101100 & %00001111
```

The above example ANDs %00101100 with %00001111 and writes the result, %00001100, to X.

Bitwise AND has an assignment form, &=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X &= $F   'Short form of X := X & $F
```

Here, the value of X is ANDed with $F and the result is stored back in X. The assignment form of Bitwise AND may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Be careful not to get Bitwise AND '&' confused with Boolean AND 'AND'. Bitwise AND is for bit manipulation while Boolean AND is for comparison purposes (see page 167).

Bitwise OR '|', '|='

The Bitwise OR operator performs a bitwise OR of the bits of the first operand with the bits of the second operand. Bitwise OR can be used in both variable and integer constant expressions, but not in floating-point constant expressions. Each bit of the two operands is subject to the following logic:

Table 2-12: Bitwise OR Truth Table

Bit States		Result
0	0	0
0	1	1
1	0	1
1	1	1

Example:

```
X := %00101100 | %00001111
```

The above example ORs %00101100 with %00001111 and writes the result, %00101111, to X.

Bitwise OR has an assignment form, |=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X |= $F   'Short form of X := X | $F
```

Here, the value of X is ORed with $F and the result is stored back in X. The assignment form of Bitwise OR may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Be careful not to get Bitwise OR '|'confused with Boolean OR 'OR'. Bitwise OR is for bit manipulation while Boolean OR is for comparison purposes (see page 168).

Bitwise XOR '^', '^='

The Bitwise XOR operator performs a bitwise XOR of the bits of the first operand with the bits of the second operand. Bitwise XOR can be used in both variable and integer constant expressions, but not in floating-point constant expressions.

Each bit of the two operands is subject to the following logic:

Table 2-13: Bitwise XOR Truth Table		
Bit States		Result
0	0	0
0	1	1
1	0	1
1	1	0

Example:

IMPROVED

```
X := %00101100 ^ %00001111
```

The above example XORs %00101100 with %00001111 and writes the result, %00100011, to X.

Bitwise XOR has an assignment form, ^=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X ^= $F   'Short form of X := X ^ $F
```

Here, the value of X is XORed with $F and the result is stored back in X. The assignment form of Bitwise XOR may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Bitwise NOT '!'

The Bitwise NOT '!' operator performs a bitwise NOT (inverse, or one's-complement) of the bits of the operand that follows it. Bitwise NOT can be used in both variable and integer constant expressions, but not in floating-point constant expressions.

Each bit of the two operands is subject to the following logic:

Table 2-14: Bitwise NOT Truth Table	
Bit State	Result
0	1
1	0

Example:

```
X := !%00101100
```

The above example NOTs %00101100 and writes the result, %11010011, to X.

Bitwise NOT becomes an assignment operator when it is the sole operator to the left of a variable on a line by itself. For example:

```
!Flag
```

This would store the inverted value Flag back into Flag.

Be careful not to get Bitwise NOT '!'confused with Boolean NOT 'NOT'. Bitwise NOT is for bit manipulation while Boolean NOT is for comparison purposes (see page 168).

Boolean AND 'AND', 'AND='

The Boolean AND 'AND' operator compares two operands and returns TRUE (-1) if both values are TRUE (non-zero), or returns FALSE (0) if one or both operands are FALSE (0). Boolean AND can be used in both variable and constant expressions.

Example:

```
X := Y AND Z
```

The above example compares the value of Y with the value of Z and sets X to either: TRUE (-1) if both Y and Z are non-zero, or FALSE (0) if either Y or Z is zero. During the comparison, it promotes each of the two values to -1 if they are non-zero, making any value, other than 0, a -1, so that the comparison becomes: "If Y is true and Z is true…"

Quite often this operator is used in combination with other comparison operators, such as in the following example.

```
IF (Y == 20) AND (Z == 100)
```

This example evaluates the result of Y == 20 against that of Z == 100, and if both are true, the Boolean AND operator returns TRUE (-1).

Boolean AND has an assignment form, AND=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X AND= True     'Short form of X := X AND True
```

Here, the value of X is promoted to TRUE if it is non-zero, then is compared with TRUE and the Boolean result (TRUE / FALSE, -1 / 0) is stored back in X. The assignment form of Boolean AND may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Be careful not to get Boolean AND 'AND' confused with Bitwise AND '&'. Boolean AND is for comparison purposes while Bitwise AND is for bit manipulation (see page 164).

Boolean OR 'OR', 'OR='

The Boolean OR 'OR' operator compares two operands and returns TRUE (-1) if either value is TRUE (non-zero), or returns FALSE (0) if both operands are FALSE (0). Boolean OR can be used in both variable and constant expressions. Example:

```
X := Y OR Z
```

The above example compares the value of Y with the value of Z and sets X to either: TRUE (-1) if either Y or Z is non-zero, or FALSE (0) if both Y and Z are zero. During the comparison, it promotes each of the two values to -1 if they are non-zero, making any value, other than 0, a -1, so that the comparison becomes: "If Y is true or Z is true…"

Quite often this operator is used in combination with other comparison operators, such as in the following example.

```
IF (Y == 1) OR (Z > 50)
```

This example evaluates the result of Y == 1 against that of Z > 50, and if either are true, the Boolean OR operator returns TRUE (-1).

Boolean OR has an assignment form, OR=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X OR= Y      'Short form of X := X OR Y
```

Here, the value of X is promoted to TRUE if it is non-zero, then is compared with Y (also promoted to TRUE if non-zero) and the Boolean result (TRUE / FALSE, -1 / 0) is stored back in X. The assignment form of Boolean OR may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Be careful not to get Boolean OR 'OR' confused with Bitwise OR'|'. Boolean OR is for comparison purposes while Bitwise OR is for bit manipulation (see page 165).

Boolean NOT 'NOT'

The Boolean NOT 'NOT' operator returns TRUE (-1) if the operand is FALSE (0), or returns FALSE (0) if the operand is TRUE (non-zero). Boolean NOT can be used in both variable and constant expressions. Example:

```
X := NOT Y
```

The above example returns the Boolean opposite of Y; TRUE (-1) if Y is zero, or FALSE (0) if Y is non-zero. During the comparison, it promotes the value of Y to -1 if it is non-zero, making any value, other than 0, a -1, so that the comparison becomes: "If NOT true" or "If NOT false"

Quite often this operator is used in combination with other comparison operators, such as in the following example.

```
IF NOT ( (Y > 9) AND (Y < 21) )
```

This example evaluates the result of (Y > 9 AND Y < 21), and returns the Boolean opposite of the result; TRUE (-1) if Y is in the range 10 to 20, in this case.

Boolean NOT becomes an assignment operator when it is the sole operator to the left of a variable on a line by itself. For example:

```
NOT Flag
```

This would store the Boolean opposite of Flag back into Flag.

Be careful not to get Boolean NOT '**NOT**' confused with Bitwise NOT'**!**'. Boolean NOT is for comparison purposes while Bitwise NOT is for bit manipulation (see page 166).

Boolean Is Equal '==', '==='

The Boolean operator Is Equal compares two operands and returns TRUE (-1) if both values are the same, or returns FALSE (0), otherwise. Is Equal can be used in both variable and constant expressions. Example:

```
X := Y == Z
```

The above example compares the value of Y with the value of Z and sets X to either: TRUE (-1) if Y is the same value as Z, or FALSE (0) if Y is not the same value as Z.

This operator is often used in conditional expressions, such as in the following example.

```
IF (Y == 1)
```

Here, the Is Equal operator returns TRUE if Y equals 1.

Is Equal has an assignment form, ===, that uses the variable to its left as both the first operand and the result destination. For example,

```
X === Y      'Short form of X := X == Y
```

Here, X is compared with Y, and if they are equal, X is set to TRUE (-1), otherwise X is set to FALSE (0). The assignment form of Is Equal may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Boolean Is Not Equal '<>', '<>='

The Boolean operator Is Not Equal compares two operands and returns True (-1) if the values are not the same, or returns FALSE (0), otherwise. Is Not Equal can be used in both variable and constant expressions. Example:

```
X := Y <> Z
```

The above example compares the value of Y with the value of Z and sets X to either: TRUE (-1) if Y is not the same value as Z, or FALSE (0) if Y is the same value as Z.

This operator is often used in conditional expressions, such as in the following example.

```
IF (Y <> 25)
```

Here, the Is Not Equal operator returns TRUE if Y is not 25.

Is Not Equal has an assignment form, <>=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X <>= Y      'Short form of X := X <> Y
```

Here, X is compared with Y, and if they are not equal, X is set to TRUE (-1), otherwise X is set to FALSE (0). The assignment form of Is Not Equal may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Boolean Is Less Than '<', '<='

The Boolean operator Is Less Than compares two operands and returns TRUE (-1) if the first value is less than the second value, or returns FALSE (0), otherwise. Is Less Than can be used in both variable and constant expressions. Example:

```
X := Y < Z
```

The above example compares the value of Y with the value of Z and sets X to either: TRUE (-1) if Y is less than the value of Z, or FALSE (0) if Y is equal to or greater than the value of Z.

This operator is often used in conditional expressions, such as in the following example.

```
IF (Y < 32)
```

Here, the Is Less Than operator returns TRUE if Y is less than 32.

Is Less Than has an assignment form, <=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X <= Y        'Short form of X := X < Y
```

Here, X is compared with Y, and if X is less than Y, X is set to TRUE (-1), otherwise X is set to FALSE (0). The assignment form of Is Less Than may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Boolean Is Greater Than '>', '>='

The Boolean operator Is Greater Than compares two operands and returns TRUE (-1) if the first value is greater than the second value, or returns FALSE (0), otherwise. Is Greater Than can be used in both variable and constant expressions. Example:

```
X := Y > Z
```

The above example compares the value of Y with the value of Z and sets X to either: TRUE (-1) if Y is greater than the value of Z, or FALSE (0) if Y is equal to or less than the value of Z.

This operator is often used in conditional expressions, such as in the following example.

```
IF (Y > 50)
```

Here, the Is Greater Than operator returns TRUE if Y is greater than 50.

Is Greater Than has an assignment form, >=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X >= Y        'Short form of X := X > Y
```

Here, X is compared with Y, and if X is greater than Y, X is set to TRUE (-1), otherwise X is set to FALSE (0). The assignment form of Is Greater Than may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Boolean Is Equal or Less '=<', '=<='

The Boolean operator Is Equal or Less compares two operands and returns TRUE (-1) if the first value is equal to or less than the second value, or returns FALSE (0), otherwise. Is Equal or Less can be used in both variable and constant expressions. Example:

```
X := Y =< Z
```

The above example compares the value of Y with the value of Z and sets X to either: TRUE (-1) if Y is equal to or less than the value of Z, or FALSE (0) if Y is greater than the value of Z.

This operator is often used in conditional expressions, such as in the following example.

```
IF (Y =< 75)
```

Here, the Is Equal or Less operator returns TRUE if Y is equal to or less than 75.

Is Equal or Less has an assignment form, =<=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X =<= Y      'Short form of X := X =< Y
```

Here, X is compared with Y, and if X is equal to or less than Y, X is set to TRUE (-1), otherwise X is set to FALSE (0). The assignment form of Is Equal or Less may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Boolean Is Equal or Greater '=>', '=>='

The Boolean operator Is Equal or Greater compares two operands and returns TRUE (-1) if the first value is equal to greater than the second value, or returns FALSE (0), otherwise. Is Equal or Greater can be used in both variable and constant expressions. Example:

```
X := Y => Z
```

The above example compares the value of Y with the value of Z and sets X to either: TRUE (-1) if Y is equal to or greater than the value of Z, or FALSE (0) if Y is less than the value of Z.

This operator is often used in conditional expressions, such as in the following example.

```
IF (Y => 100)
```

Here, the Is Equal or Greater operator returns TRUE if Y is equal to or greater than 100.

Is Equal or Greater has an assignment form, =>=, that uses the variable to its left as both the first operand and the result destination. For example,

```
X =>= Y      'Short form of X := X => Y
```

Here, X is compared with Y, and if X is equal to or greater than Y, X is set to TRUE (-1), otherwise X is set to FALSE (0). The assignment form of Is Equal or Greater may also be used within expressions for intermediate results; see Intermediate Assignments, page 147.

Symbol Address '@'

The Symbol Address operator returns the address of the symbol following it. Symbol Address can be used in variable and integer constant expressions, but not in floating-point constant expressions. Example:

```
BYTE[@Str] := "A"
```

In the above example, the Symbol Address operator returns the address of the Str symbol, which is then used by the BYTE memory array reference to store the character "A" at that address.

Symbol Address is often used to pass the address of strings and data structures, defined in a DAT block, to methods that operate on them.

It is important to note that this is a special operator that behaves differently in variable expressions than it does in constant expressions. At run time, like our example above shows, it returns the absolute address of the symbol following it. This run-time, absolute address consists of the object's program base address plus the symbol's offset address.

In constant expressions, it only returns the symbol's offset within the object. It cannot return the absolute address, effective at run time, because that address changes depending on the object's actual address at run time. To properly use the Symbol Address in a constant, such as a table of data, see the Object Address Plus Symbol operator, below.

Object Address Plus Symbol '@@'

The Object Address Plus Symbol operator returns the value of the symbol following it plus the current object's program base address. Object Address Plus Symbol can only be used in variable expressions.

This operator is useful when creating a table of offset addresses, then at run time, using those offsets to reference the absolute run-time addresses they represent. For example, a DAT block may contain a number of strings to which you want both direct and indirect access. Here's an example DAT block containing strings.

```
DAT
  Str1  byte  "Hello.", 0
  Str2  byte  "This is an example", 0
  Str3  byte  "of strings in a DAT block.",0
```

At run time we can access those strings directly, using @Str1, @Str2, and @Str3, but accessing them indirectly is troublesome because each string is of a different length; making it difficult to use any of them as a base for indirect address calculations.

The solution might seem to be within reach by simply making another table of the addresses themselves, as in:

```
DAT
  StrAddr  word  @Str1, @Str2, @Str3
```

This creates a table of words, starting at StrAddr, where each word contains the address of a unique string. Unfortunately, for compile-time constants (like those of the StrAddr table), the address returned by @ is only the compile-time offset address, rather than the run-time absolute address, of the symbol. To get the true, run-time address, we need to add the object's program base address to the symbol's offset address. That is what the Object Address Plus Symbol operator does. Example:

```
REPEAT Idx FROM 0 TO 2
  PrintStr(@@StrAddr[Idx])
```

The above example increments Idx from 0 through 2. The StrAddr[Idx] statement retrieves the compile-time offset of the string stored in element Idx of the StrAddr table. The @@ operator in front of the StrAddr[Idx] statement adds the object's base address to the compile-time offset value that was retrieved, resulting in a valid run-time address of the string. The PrintStr method, whose code is not shown in this example, can use that address to process each character of the string.

OUTA, OUTB

Register: Output registers for 32-bit Ports A and B.

((PUB ¦ PRI))
OUTA 〈[*Pin(s)*]〉

((PUB ¦ PRI))
OUTB 〈[*Pin(s)*]〉 (Reserved for future use)

Returns: Current value of output *Pin(s)* for Port A or B, if used as a source variable.

- ***Pin(s)*** is an optional expression, or a range-expression, that specifies the I/O pin, or pins, to access in Port A (0-31) or Port B (32-63). If given as a single expression, only the pin specified is accessed. If given as a range-expression (two expressions in a range format; x..y) the contiguous pins from the start to end expressions are accessed.

Explanation

OUTA and OUTB are two of six registers (DIRA, DIRB, INA, INB, OUTA and OUTB) that directly affect the I/O pins. The OUTA register holds the output states for each of the 32 I/O pins in Port A; bits 0 through 31 correspond to P0 through P31. The OUTB register holds the output states for each of the 32 I/O pins in Port B; bits 0 through 31 correspond to P32 through P63.

NOTE: OUTB is reserved for future use; the Propeller P8X32A does not include Port B I/O pins so only OUTA is discussed below.

OUTA is used to both set and get the current output states of one or more I/O pins in Port A. A low (0) bit sets the corresponding I/O pin to ground. A high (1) bit sets the corresponding I/O pin VDD (3.3 volts). All the OUTA register's bits default to zero (0) upon cog startup.

Each cog has access to all I/O pins at any given time. Essentially, all I/O pins are directly connected to each cog so that there is no hub-related mutually exclusive access involved. Each cog maintains its own OUTA register that gives it the ability to set any I/O pin's output state (low or high). Each cog's output states is OR'd with that of the other cogs' output states and the resulting 32-bit value becomes the output states of Port A pins P0 through P31. The result is that each I/O pin's output state is the "wired-OR" of the entire cog collective. See I/O Pins on page 26 for more information.

Note that each cog's output states are made up of the OR'd states of its internal I/O hardware (Output Register, Video Generator, etc.) and that is all AND'd with its Direction Register's states.

An I/O pin actually outputs low or high, as specified by the cog's output states, if, and only if, that pin's bit in that same cog's direction register (DIRA) is high (1). Otherwise, that cog specifies the pin to be an input and its output state is ignored.

This configuration can easily be described in the following simple rules:

A. A pin outputs low only if all active cogs that set it to output also set it to low.
B. A pin outputs high if any active cog sets it to an output and also sets it high.

If a cog is disabled, its direction register is treated as if were cleared to 0, causing it to exert no influence on I/O pin directions and states.

Note because of the "wired-OR" nature of the I/O pins, no electrical contention between cogs is possible, yet they can all still access I/O pins simultaneously. It is up to the application developer to ensure that no two cogs cause logical contention on the same I/O pin during run time.

Using OUTA

Set or clear bits in OUTA to affect the output state of I/O pins as desired. Make sure to also set the corresponding bits of DIRA to make that pin an output. For example:

```
DIRA := %00000100_00110000_00000001_11110000
OUTA := %01000100_00110000_00000001_10010000
```

The DIRA line above sets the I/O pins 26, 21, 20, 8, 7, 6, 5 and 4 to outputs and the rest to inputs. The OUTA line sets I/O pins 30, 26, 21, 20, 8, 7, and 4 to high, the rest to low. The result is that I/O pins 26, 21, 20, 8, 7, and 4 output high and I/O pins 6 and 5 output low. I/O pin 30 is set to an input direction (according to DIRA) so the high in bit 30 of OUTA is ignored and the pin remains an input according to this cog.

Using the optional *Pin(s)* field, and the post-clear (~) and post-set (~~) unary operators, the cog can affect one I/O pin (one bit) at a time. The *Pin(s)* field treats the I/O pin registers as an array of 32 bits. For example:

```
DIRA[10]~~                  'Set P10 to output
OUTA[10]~                   'Make P10 low
OUTA[10]~~                  'Make P10 high
```

The first line in the code above sets I/O pin 10 to output. The second line clears P10's output latch bit, making P10 output low (ground). The third line sets P10's output latch bit, making P10 output high (VDD).

In Spin, the **OUTA** register supports a special form of expression, called a range-expression, which allows you to affect a group of I/O pins at once, without affecting others outside the specified range. To affect multiple, contiguous I/O pins at once, use a range expression (like x..y) in the *Pin(s)* field.

```
DIRA[12..8]~~               'Set DIRA12:8 (P12-P8 to output)
OUTA[12..8] := %11001       'Set P12:8 to 1, 1, 0, 0, and 1
```

The first line, "DIRA…," sets P12, P11, P10, P9 and P8 to outputs; all other pins remain in their previous state. The second line, "OUTA…," sets P12, P11, and P8 to output high, and P10 and P9 to output low.

IMPORTANT: The order of the values in a range-expression affects how it is used. For example, the following swaps the order of the range from the previous example.

```
DIRA[8..12]~~               'Set DIRA8:12 (P8-P12 to output)
OUTA[8..12] := %11001       'Set OUTA8:12 to 1, 1, 0, 0, and 1
```

Here, **DIRA** bits 8 through 12 are set to high (like before) but **OUTA** bits 8, 9, 10, 11 and 12 are set equal to 1, 1, 0, 0, and 1, respectively, making P8, P9 and P12 output high and P10 and P11 output low.

This is a powerful feature of range-expressions, but if care is not taken it can also cause strange, unintentional results.

Normally **OUTA** is only written to but it can also be read from to retrieve the current I/O pin output latch states. This is ONLY the cog's output latch states, not necessarily the actual output states of the Propeller chip's I/O pins, as they can be further affected by other cogs or even this cog's other I/O hardware (Video Generator, Count A, etc.). The following assumes Temp is a variable created elsewhere:

```
Temp := OUTA[15..13]        'Get output latch state of P15 to P13
```

The above sets Temp equal to **OUTA** bits 15, 14, and 13; i.e.: the lower 3 bits of Temp are now equal to OUTA15:13 and the other bits of Temp are cleared to zero.

PAR

Register: Cog Boot Parameter register.

((PUB ¦ PRI))
  PAR

IMPROVED

Returns: Address value passed during launch of assembly code with COGINIT or COGNEW.

Explanation

The PAR register contains the address value passed into the *Parameter* field of a COGINIT or COGNEW command; see COGINIT, page 76 and COGNEW, page 78. The PAR register's contents are used by Propeller Assembly code to locate and operate on memory shared between Spin code and assembly code.

Since the PAR register is intended to contain an address upon cog launch, the value stored into it via COGINIT and COGNEW is limited to 14 bits: a 16-bit word with the lower two bits cleared to zero.

Using PAR

PAR is affected by Spin code and is used by assembly code as a memory pointer mechanism to point to shared main memory between the two. Either the COGINIT or COGNEW command, when launching Propeller Assembly into a cog, affects the PAR register. For example:

```
VAR
  long   Shared                         'Shared variable (Spin & Assy)

PUB Main | Temp
  cognew(@Process, @Shared)             'Launch assy, pass Shared addr
  repeat
    <do something with Shared vars>

DAT
                  org 0
Process           mov Mem, PAR          'Retrieve shared memory addr
:loop             <do something>
                  wrlong ValReg, Mem    'Move ValReg value to Shared
                  jmp #:loop
Mem       res  1
ValReg    res  1
```

In the example above, the Main method launches the Process assembly routine into a new cog with COGNEW. The second parameter of COGNEW is used by Main to pass the address of a variable, Shared. The assembly routine, Process, retrieves that address value from its PAR register and stores it locally in Mem. Then it performs some task, updating its local ValReg register (created at the end of the DAT block) and finally updates the Shared variable via wrlong ValReg, Mem.

NEW

In Propeller Assembly, the PAR register is read-only so it should only be accessed as a source (s-field) value (i.e., mov *dest*, *source*).

PHSA, PHSB

Register: Counter A and Counter B Phase Registers.

((PUB ¦ PRI))
PHSA

((PUB ¦ PRI))
PHSB

Returns: Current value of Counter A or Counter B Phase Register, if used as a source variable.

Explanation

PHSA and PHSB are two of six registers (CTRA, CTRB, FRQA, FRQB, PHSA, and PHSB) that affect the behavior of a cog's Counter Modules. Each cog has two identical counter modules (A and B) that can perform many repetitive tasks. The PHSA and PHSB registers contain values that can be directly read or written by the cog, but may also be accumulating with the value of FRQA and FRQB, respectively, on potentially every System Clock cycle. See CTRA on page 95 for more information.

Using PHSA and PHSB

PHSA and PHSB can be read/written like other registers or pre-defined variables. For example:

```
PHSA := $1FFFFFFF
```

The above code sets PHSA to $1FFFFFFF. Depending on the CTRMODE field of the CTRA register, this value may remain the same, or may automatically increment by the value in FRQA at a frequency determined by the System Clock and the primary and/or secondary I/O pins. See CTRA, CTRB on page 95 for more information.

NEW Note that in Propeller Assembly, the PHSA and PHSB registers cannot be used in a read-modify-write operation in the destination field of an instruction. Instead, separate read, modify, and write operations (instructions) must be performed.

Keep in mind that writing to PHSA or PHSB directly overrides both the current accumulated value and any potential accumulation scheduled for the same moment the write is performed.

PRI

Designator: Declare a Private Method Block.

((PUB ¦ PRI))
PRI *Name* 〈 (*Param* 〈 , *Param*〉...) 〉 〈 : *RValue*〉 〈 | *LocalVar* 〈 [*Count*] 〉〉 〈 , *LocalVar* 〈 [*Count*] 〉〉...
SourceCodeStatements

- ***Name*** is the desired name for the private method.
- ***Param*** is a parameter name (optional). Methods can contain zero or more comma-delimited parameters, enclosed in parentheses. *Param* must be globally unique, but other methods may also use the same symbol name. Each parameter is essentially a long variable and can be treated as such.
- ***RValue*** is a name for the return value of the method (optional). This becomes an alias to the method's built-in RESULT variable. *RValue* must be globally unique, but other methods may also use the same symbol name. The *RValue* (and/or RESULT variable) is initialized to zero (0) upon each call to the method.
- ***LocalVar*** is a name for a local variable (optional). *LocalVar* must be globally unique, but other methods may also use the same symbol name. All local variables are of size long (four bytes) and are left uninitialized upon each call to the method. Methods can contain zero or more comma-delimited local variables.
- ***Count*** is an optional expression, enclosed in brackets, that indicates this is a local array variable, with *Count* number of elements; each being a long in size. When later referencing these elements, they begin with element 0 and end with element *Count*-1.
- ***SourceCodeStatements*** is one or more lines of executable source code, indented by at least one space, that perform the function of the method.

Explanation

PRI is the Private Method Block declaration. A Private Method is a section of source code that performs a specific function then returns a result value. This is one of six special declarations (CON, VAR, OBJ, PUB, PRI, and DAT) that provide inherent structure to the Spin language.

Every object can contain a number of private (PRI) and public (PUB) methods. Private methods can only be accessed from inside of the object and serve to perform vital, protected functions, for the object. Private methods are like Public methods in every way except that they are declared with PRI, instead of PUB, and are not accessible from outside the object. Please see PUB, Page 182, for more information.

PUB

Designator: Declare a Public Method Block.

((PUB ¦ PRI))
PUB ***Name*** 〈 (***Param*** 〈 , ***Param***〉...) 〉 〈 : ***RValue***〉 〈 | ***LocalVar*** 〈 [***Count***] 〉〉 〈 , ***LocalVar*** 〈 [***Count***] 〉〉...
SourceCodeStatements

- ***Name*** is the desired name for the public method.
- ***Param*** is a parameter name (optional). Methods can contain zero or more comma-delimited parameters, enclosed in parentheses. *Param* must be globally unique, but other methods may also use the same symbol name. Each parameter is essentially a long variable and can be treated as such.
- ***RValue*** is a name for the return value of the method (optional). This becomes an alias to the method's built-in RESULT variable. *RValue* must be globally unique, but other methods may also use the same symbol name. The *RValue* (and/or RESULT variable) is initialized to zero (0) upon each call to the method.
- ***LocalVar*** is a name for a local variable (optional). *LocalVar* must be globally unique, but other methods may also use the same symbol name. All local variables are of size long (four bytes) and are left uninitialized upon each call to the method. Methods can contain zero or more comma-delimited local variables.
- ***Count*** is an optional expression, enclosed in brackets, that indicates this is a local array variable, with *Count* number of elements; each being a long in size. When later referencing these elements, they begin with element 0 and end with element *Count*-1.
- ***SourceCodeStatements*** is one or more lines of executable source code, indented by at least one space, that perform the function of the method.

Explanation

PUB is the Public Method Block declaration. A Public Method is a section of source code that performs a specific function then returns a result value. This is one of six special declarations (CON, VAR, OBJ, PUB, PRI, and DAT) that provide inherent structure to the Spin language.

Every object can contain a number of public (PUB) and private (PRI) methods. Public methods can be accessed outside of the object itself and serve to make up the interface to the object.

The PUB and PRI declarations don't return a value themselves, but the public and private methods they represent always return a value when called from elsewhere in the code.

Public Method Declaration

Public Method declarations begin with **PUB**, in column 1 of a line, followed a unique name and an optional set of parameters, a result variable, and local variables.

Example:

```
PUB Init
  <initialization code>

PUB MotorPos : Position
  Position := <code to retrieve motor position>

PUB MoveMotor(Position, Speed) : Success | PosIndex
  <code that moves motor to Position at Speed and returns True/False>
```

This example contains three public methods, `Init`, `MotorPos` and `MoveMotor`. The `Init` method has no parameters and declares no return value or local variables. The `MotorPos` method has no parameters but declares a return value called `Position`. The `MoveMotor` method has two parameters, `Position` and `Speed`, a return value, `Success`, and a local variable, `PosIndex`.

All executable statements that belong to a **PUB** method appear underneath its declaration, indented by at least one space.

The Return Value

Whether or not a **PUB** declaration specifies an *RValue* , there is always an implied return value that defaults to zero (0). There is a pre-defined name for this return value within every **PUB** method, called **RESULT**. At any time within a method, **RESULT** can be updated like any other variable and, upon exiting the method, the current value of **RESULT** will be passed back to the caller. In addition, if a **RESULT** is declared for the method, that name can be used interchangeably with the built-in **RESULT** variable. For instance, the `MotorPos` method above sets "`Position := …`" and could also have used "`Result := …`" for the same effect. Despite this, it is considered good practice to give a descriptive name to the return value (in the `PUB` declaration) for any method whose return value is significant. Likewise, it is good practice to leave the return value unnamed (in the **PUB** declaration) for any method whose return value is unimportant and unused.

Parameters and Local Variables

Parameters and local variables are all longs (four bytes). In fact, parameters are really just variables that are initialized to the corresponding values specified by the caller of the method. Local variables, however, are not initialized; they contain random data whenever the method is called.

All parameters are passed into a method by value, not by reference, so any changes to the parameters themselves are not reflected outside of the method. For example, if we called MoveMotor using a variable called Pos for the first parameter, it may look something like this:

```
Pos := 250
MoveMotor(Pos, 100)
```

When the MoveMotor method is executed, it receives the value of Pos in its Position parameter, and the value 100 in its Speed parameter. Inside the MoveMotor method, it can change Position and Speed at any time, but the value of Pos (the caller's variable) remains at 250.

If a variable must be altered by a routine, the caller must pass the variable by reference; meaning it must pass the address of the variable instead of the value of the variable, and the routine must treat that parameter as the address of a memory location in which to operate on. The address of a variable, or other register-based symbol, can be retrieved by using the Symbol Address operator, '**@**'. For example,

```
Pos := 250
MoveMotor(@Pos, 100)
```

The caller passed the address of Pos for the first parameter to MoveMotor. What MoveMotor receives in its Position parameter is the address of the caller's Pos variable. The address is just a number, like any other, so the MoveMotor method must be designed to treat it as an address, rather than a value. The MoveMotor method then must use something like:

```
PosIndex := LONG[Position]
```

...to retrieve the value of the caller's Pos variable, and something like:

```
LONG[Position] := <some expression>
```

...to modify the caller's Pos variable, if necessary.

Passing a value by reference with the Symbol Address operator is commonly used when providing a string variable to a method. Since string variables are really just byte arrays,

there is no way to pass them to a method by value; doing so would result in the method receiving only the first character. Even if a method does not need to modify a string, or other logical array, the array in question still needs to be passed by reference because there are multiple elements to be accessed.

NEW

Optimized Addressing

In the compiled Propeller Application, the first eight (8) longs that make up the parameters, the RESULT variable, and the local variables are addressed using an optimized encoding scheme. This means accessing the first eight longs (parameters, RESULT, and local variables) takes slightly less time than accessing the ninth, or later, longs. To optimize execution speed, ensure that all local longs used by the method's most repetitive routines are among the first eight. A similar mechanism applies to global variables; see the VAR section, page 212, for more information.

Exiting a Method

A method is exited either when execution reaches the last statement within the method or when it reaches a RETURN or ABORT command. A method may have only one exit point (the last executable statement), or may have many exit points (any number of RETURN or ABORT commands in addition to the last executable statement). The RETURN and ABORT commands can also be used to set the RESULT variable upon exit; for more information see RETURN, page 196, and ABORT, page 47.

QUIT

Command: Exit from REPEAT loop immediately.

((PUB ¦ PRI))
  QUIT

Explanation

QUIT is one of two commands (NEXT and QUIT) that affect REPEAT loops. QUIT causes a REPEAT loop to terminate immediately.

Using QUIT

QUIT is typically used as an exception case, in a conditional statement, in REPEAT loops to terminate the loop prematurely. For example, assume that DoMore and SystemOkay are methods created elsewhere that each return Boolean values:

```
repeat while DoMore          'Repeat while more to do
  !outa[0]                   'Toggle status light
  <do something>             'Perform some task
  if !SystemOkay
    quit                     'If system failure, exit
  <more code here>           'Perform other tasks
```

The above code toggles a status light on P0 and performs other tasks while the DoMore method returns TRUE. However, if the SystemOkay method returns FALSE partway through the loop, the IF statement executes the QUIT command which causes the loop to terminate immediately.

The QUIT command can only be used within a REPEAT loop; an error will occur otherwise.

REBOOT

Command: Reset the Propeller chip.

((PUB ¦ PRI))
  REBOOT

Explanation

This is a software controlled reset, but acts like just like a hardware reset via the RESn pin.

Use REBOOT if you want to reset the Propeller chip to its power-up state. All the same hardware-based, power-up/reset delays, as well as the boot-up processes, are applied as if the Propeller had been reset via the RESn pin or a power cycle.

REPEAT

Command: Execute code block repetitively.

((PUB ¦ PRI))
  REPEAT ⟨*Count*⟩
  →| *Statement(s)*

((PUB ¦ PRI))
  REPEAT *Variable* FROM *Start* TO *Finish* ⟨STEP *Delta*⟩
  →| *Statement(s)*

((PUB ¦ PRI))
  REPEAT ((UNTIL ¦ WHILE)) *Condition(s)*
  →| *Statement(s)*

((PUB ¦ PRI))
  REPEAT
  →| *Statement(s)*
  ((UNTIL ¦ WHILE)) *Condition(s)*

- ***Count*** is an optional expression indicating the finite number of times to execute *Statement(s)*. If *Count* is omitted, syntax 1 creates an infinite loop made up of *Statement(s)*.
- ***Statement(s)*** is an optional block of one or more lines of code to execute repeatedly. Omitting *Statement(s)* is rare, but may be useful in syntax 3 and 4 if *Condition(s)* achieves the needed effects.
- ***Variable*** is a variable, usually user-defined, that will be iterated from *Start* to *Finish*, optionally by *Delta* units per iteration. *Variable* can be used in *Statement(s)* to determine or utilize the iteration count.
- ***Start*** is an expression that determines the starting value of *Variable* in syntax 2. If *Start* is less than *Finish*, *Variable* will be incremented each iteration; it will be decremented otherwise.
- ***Finish*** is an expression that determines the ending value of *Variable* in syntax 2. If *Finish* is greater than *Start, Variable* will be incremented each iteration; it will be decremented otherwise.
- ***Delta*** is an optional expression that determines the units in which to increment/decrement *Variable* each iteration (syntax 2). If omitted, *Variable* is incremented/decremented by 1 each iteration.

- ***Condition(s)*** is one or more Boolean expression(s) used by syntax 3 and 4 to continue or terminate the loop. When preceded by UNTIL, *Condition(s)* terminates the loop when true. When preceded by WHILE, *Conditions(s)* terminates the loop when FALSE.

Explanation

REPEAT is the very flexible looping structure for Spin code. It can be used to create any type of loop, including: infinite, finite, with/without loop counter, and conditional zero-to-many/one-to-many loops.

Indention is Critical

IMPORTANT: Indention is critical. The Spin language relies on indention (of one space or more) on lines following conditional commands to determine if they belong to that command or not. To have the Propeller Tool indicate these logically grouped blocks of code on-screen, you can press Ctrl + I to turn on block-group indicators. Pressing Ctrl + I again will disable that feature. See the Propeller Tool Help for a complete list of shortcut keys.

Infinite Loops (Syntax 1)

Truthfully, any of the four forms of REPEAT can be made into infinite loops, but the form used most often for this purpose is syntax 1 without the *Count* field. For example:

```
repeat                      'Repeat endlessly
  !outa[25]                 'Toggle P25
  waitcnt(2_000 + cnt)      'Pause for 2,000 cycles
```

This code repeats the `!outa[25]` and `waitcnt(2_000 + cnt)` lines endlessly. Both lines are indented from the REPEAT so they belong to the REPEAT loop.

Since *Statement(s)* is really an optional part of REPEAT, the REPEAT command by itself can be used as an endless loop that does nothing but keep the cog active. This can be intentional, but sometimes is unintentional due to improper indention. For example:

```
repeat                      'Repeat endlessly
!outa[25]                   'Toggle P25       <-- This is never run
```

The above example is erroneous; the last line is never executed because the REPEAT above it is an endless loop that has no *Statement(s)*; there is nothing indented immediately below it, so the cog simply sits in an endless loop at the REPEAT line that does nothing but keep the cog active and consuming power.

Simple Finite Loops (Syntax 1)

Most loops are finite in nature; they execute a limited number of iterations only. The simplest form is syntax 1 with the *Count* field included.

For example:

```
repeat 10                       'Repeat 10 times
  !outa[25]                     'Toggle P25
byte[$7000]++                   'Increment RAM location $7000
```

The above code toggles P25 ten times, then increments the value in RAM location $7000.

NEW

Note that the *Count* field may be any numeric expression but the expression is evaluated only once, the first time the loop is entered. This means that any changes to the expression's variables within the loop will not affect the number of iterations of the loop. The next example assumes the variable Index was created previously.

```
Index := 10                     'Set loop to repeat 10 times
repeat Index                    'Repeat Index times
  !outa[25]                     'Toggle P25
  Index := 20                   'Change Index to 20
```

In the above example, Index is 10 upon entering the REPEAT loop the first time. Each time through the loop, however, Index is set to 20, but the loop continues to execute only 10 times.

Counted Finite Loops (Syntax 2)

Quite often it is necessary to count the loop iterations so the loop's code can perform differently based on that count. The REPEAT command makes it easy to do this with syntax 2. The next example assumes the variable Index was created previously.

```
repeat Index from 0 to 9        'Repeat 10 times
  byte[$7000][Index]++          'Increment RAM locations $7000 to $7009
```

Like the previous example, the code above loops 10 times, but each time it adjusts the variable Index. The first time through the loop, Index will be 0 (as indicated by the "from 0") and each iteration afterwards Index will be 1 higher than the previous (as indicated by the "to 9"): ..1, 2, 3…9. After the tenth iteration, Index will be incremented to 10 and the loop will terminate, causing the next code following the REPEAT loop structure to execute, if any exists. The code in the loop uses Index as an offset to affect memory, byte[$7000][Index]++; in this case it is incrementing each of the byte-sized values in RAM locations $7000 to $7009 by 1, one at a time.

The REPEAT command automatically determines whether the range suggested by *Start* and *Finish* is increasing or decreasing. Since the above example used 0 to 9, the range is an increasing range; adjusting Index by +1 every time. To get the count to go backwards, simply reverse the *Start* and *Finish* values, as in:

```
repeat Index from 9 to 0    'Repeat 10 times
  byte[$7000][Index]++      'Increment RAM $7009 down through $7000
```

This example also loops 10 times, but counts with Index from 9 down to 0; adjusting Index by -1 each time. The contents of the loop still increments the values in RAM, but from locations $7009 down to $7000. After the tenth iteration, Index will equal -1.

Since the *Start* and *Finish* fields can be expressions, they can contain variables. The next example assumes that S and F are variables created previously.

```
S := 0
F := 9
repeat 2                        'Repeat twice
  repeat Index from S to F      'Repeat 10 times
    byte[$7000][Index]++        'Increment RAM locations 7000..$7009
  S := 9
  F := 0
```

The above example uses a nested loop. The outer loop (the first one) repeats 2 times. The inner loop repeats with Index from S to F, which were previously set to 0 and 9, respectively. The inner loop increments the values in RAM locations $7000 to $7009, in that order, because the inner loop is counting iterations from 0 to 9. Then, the inner loop terminates (with Index being set to 10) and the last two lines set S to 9 and F to 0, effectively swapping the *Start* and *Finish* values. Since this is still inside the outer loop, the outer loop then executes its contents again (for the second time) causing the inner loop to repeat with Index from 9 down to 0. The inner loop increments the values in RAM locations $7009 to $7000, in that order (reverse of the previous time) and terminates with Index equaling -1. The last two lines set S and F again, but the outer loop does not repeat a third time.

REPEAT loops don't have to be limited to incrementing or decrementing by 1 either. If the REPEAT command uses the optional STEP *Delta* syntax, it will increment or decrement the *Variable* by the *Delta* amount. In the syntax 2 form, REPEAT is actually always using a *Delta* value, but when the "STEP *Delta*" component is omitted, it uses either +1 or -1 by default, depending on the range of *Start* and *Finish*. The following example includes the optional *Delta* value to increment by 2.

```
repeat Index from 0 to 8 step 2 'Repeat 5 times
```

```
  byte[$7000][Index]++          'Increment even RAM $7000 to $7008
```

Here, REPEAT loops five times, with Index set to 0, 2, 4, 6, and 8, respectively. This code effectively increments every other RAM location (the even numbered locations) from $7000 to $7008 and terminates with Index equaling 10.

The *Delta* field can be positive or negative, regardless of the natural ascending/descending range of the *Start* and *Finish* values, and can even be adjusted within the loop to achieve interesting effects. For example, assuming Index and D are previously defined variables, the following code sets Index to the following sequence: 5, 6, 6, 5, 3.

```
D := 2
repeat Index from 5 to 10 step D
  --D
```

This loop started out with Index at 5 and a *Delta* (D) of +2. But each iteration of the loop decrements D by one, so at the end of iteration 1, Index = 5 and D = +1. Iteration 2 has Index = 6 and D = 0. Iteration 3 has Index = 6 and D = -1. Iteration 4 has Index = 5 and D = -2. Iteration 5 has Index = 3 and D = -3. The loop then terminates because Index plus *Delta* (3 + -3) is outside the range of *Start* to *Finish* (5 to 10).

Conditional Loops (Syntax 3 and 4)

The final forms of REPEAT, syntax 3 and 4, are finite loops with conditional exits and have flexible options allowing for the use of either positive or negative logic and the creation of zero-to-many or one-to-many iteration loops. These two forms of REPEAT are usually referred to as "repeat while" or "repeat until" loops.

Let's look at the REPEAT form described by syntax 3. It consists of the REPEAT command followed immediately by either WHILE or UNTIL then *Condition(s)* and finally, on the lines below it, optional *Statement(s)*. Since this form tests *Condition(s)* at the start of every iteration, it creates a zero-to-many loop; the *Statement(s)* block will execute zero or more times, depending on the *Condition(s)*. For example, assume that X is a variable created earlier:

```
X := 0
repeat while X < 10         'Repeat while X is less than 10
  byte[$7000][X] := 0       'Increment RAM value
  X++                       'Increment X
```

This example first sets X to 0, then repeats the loop while X is less than 10. The code inside the loop clears RAM locations based on X (starting at location $7000) and increments X.

After the 10th iteration of the loop, X equals 10, making the condition while X < 10 false and the loop terminates.

This loop is said to use "positive" logic because it continues "WHILE" a condition is true. It could also be written with "negative" logic using UNTIL, instead. Such as:

```
X := 0
repeat until X > 9          'Repeat until X is greater than 9
  byte[$7000][X] := 0       'Increment RAM value
  X++                       'Increment X
```

The above example performs the same way as the previous, but the REPEAT loop uses negative logic because it continues "UNTIL" a condition is true; i.e: it continues while a condition is false.

In either example, if X was equal to 10 or higher before the first iteration of the REPEAT loop, the condition would cause the loop to never execute at all, which is why we call it a zero-to-many loop.

The REPEAT form described by syntax 4 is very similar to syntax 3, but the condition is tested at the end of every iteration, making it a one-to-many loop. For example:

```
X := 0
repeat
  byte[$7000][X] := 0       'Increment RAM value
  X++                       'Increment X
while X < 10                'Repeat while X is less than 10
```

This works the same as the previous examples, looping 10 times, except that the condition is not tested until the end of each iteration. However, unlike the previous examples, even if X was equal to 10 or higher before the first iteration, the loop would run once then terminate, which is why we call it a one-to-many loop.

Other REPEAT Options

There are two other commands that affect the behavior of REPEAT loops: NEXT and QUIT. See the NEXT (page 140) and QUIT (page 186) commands for more information.

RESULT

Variable: The return value variable for methods.

((PUB ¦ PRI))
  RESULT

Explanation

The RESULT variable is a pre-defined local variable for each PUB and PRI method. RESULT holds the method's return value; the value passed back to the caller of the method, when the method is terminated.

When a public or private method is called, its built-in RESULT variable is initialized to zero (0). If that method does not alter RESULT, or does not call RETURN or ABORT with a value specified, then zero will be the return value upon that method's termination.

Using RESULT

In the example below, the DoSomething method sets RESULT equal to 100 at its end. The Main method calls DoSomething and sets its local variable, Temp, equal to the result; so that when DoSomething exits, Temp will be set to 100

```
PUB Main | Temp
  Temp := DoSomething     'Call DoSomething, set Temp to return value

PUB DoSomething
  <do something here>
  result := 100           'Set result to 100
```

You can also provide an alias name for a method's RESULT variable in order to make it more clear what the method returns. This is highly recommended since it makes a method's intent more easily discerned. For example:

```
PUB GetChar : Char
  <do something>
  Char := <retrieved character>   'Set Char (result) to the character
```

The above method, GetChar, declares Char as an alias for its built-in RESULT variable; see PUB, page 182 or PRI, page 181, for more information. The GetChar method then performs some task to get a character then it sets Char to the value of the retrieved character. It could have

also used "`result := …`" to set the return value since either statement affects the method's return value.

Either the **RESULT** variable, or the alias provided for it, may be modified multiple times within the method before exiting since they both affect **RESULT** and only the last value of **RESULT** will be used upon exiting.

RETURN

Command: Exit from PUB/PRI method with optional return *Value*.

((PUB ¦ PRI))
  RETURN ⟨*Value*⟩

Returns: Either the current RESULT value, or *Value* if provided.

- ***Value*** is an optional expression whose value is to be returned from the PUB or PRI method.

Explanation

RETURN is one of two commands (ABORT and RETURN) that terminate a PUB or PRI method's execution. RETURN causes a return from a PUB or PRI method with normal status; meaning it pops the call stack once and returns to the caller of this method, delivering a value in the process.

Every PUB or PRI method has an implied RETURN at its end, but RETURN can also be manually entered in one or more places within the method to create multiple exit points.

When RETURN appears without the optional *Value*, it returns the current value of the PUB/PRI's built-in RESULT variable. If the *Value* field was entered, however, the PUB or PRI returns with that *Value* instead.

About the Call Stack

When methods are called, simply by referring to them from other methods, there must be some mechanism in place to store where to return to once the called method is completed. This mechanism is a called a "stack" but we'll use the term "call stack" here. It is simply RAM memory used to store return addresses, return values, parameters and intermediate results. As more and more methods are called, the call stack logically gets longer. As more and more methods are returned from (via RETURN or by reaching the end of the method) the call stack gets shorter. This is called "pushing" onto the stack and "popping" off of the stack, respectively.

The RETURN command pops the most recent data off the call stack to facilitate returning to the immediate caller; the one who directly called the method that just returned.

Using RETURN

The following example demonstrates two uses of **RETURN**. Assume that DisplayDivByZeroError is a method defined elsewhere.

```
PUB Add(Num1, Num2)
  Result := Num1 + Num2                 'Add Num1 + Num2
  return

PUB Divide(Dividend, Divisor)
  if Divisor == 0                       'Check if Divisor = 0
    DisplayDivByZeroError               'If so, display error
    return 0                            'and return with 0
  return Dividend / Divisor             'Otherwise return quotient
```

The Add method sets its built-in **RESULT** variable equal to Num1 plus Num2, then executes **RETURN**. The **RETURN** causes Add to return the value of **RESULT** to the caller. Note that this **RETURN** was not really required because the Propeller Tool Compiler will automatically put it in at the end of any methods that don't have one.

The Divide method checks the Divisor value. If Divisor equals zero, it calls a DisplayDivByZeroError method and then executes return 0, which immediately causes the method to return with the value 0. If, however, the Divisor was not equal to zero, it executes return Dividend / Divisor, which causes the method to return with the result of the division. This is an example where the last **RETURN** was used to perform the calculation and return the result all in one step rather than separately affecting the built-in **RESULT** variable beforehand.

ROUND

Directive: Round a floating-point constant to the nearest integer.

((CON ¦ VAR ¦ OBJ ¦ PUB ¦ PRI ¦ DAT))
ROUND (*FloatConstant*)

Returns: Nearest integer to original floating-point constant value.

- ***FloatConstant*** is the floating-point constant expression to be rounded to the nearest integer.

Explanation

ROUND is one of three directives (FLOAT, ROUND and TRUNC) used for floating-point constant expressions. ROUND returns an integer constant that is the closest integer value to the given floating-point constant expression. Fractional values of ½ (.5) or higher are rounded up to the nearest whole number while lower fractions are rounded down.

Using ROUND

ROUND can be used to round floating-point constants up or down to the nearest integer value. Note that this is for compile-time constant expressions only, not run-time variable expressions. For example:

```
CON
  OneHalf = 0.5
  Smaller = 0.4999
  Rnd1    = round(OneHalf)
  Rnd2    = round(Smaller)
  Rnd3    = round(Smaller * 10.0) + 4
```

The above code creates two floating-point constants, OneHalf and Smaller, equal to 0.5 and 0.4999, respectively. The next three constants, Rnd1, Rnd2 and Rnd3, are integer constants that are based on OneHalf and Smaller using the ROUND directive. Rnd1 = 1, Rnd2 = 0, and Rnd3 = 9.

About Floating-Point Constants

The Propeller compiler handles floating-point constants as a single-precision real number as described by the IEEE-754 standard. Single-precision real numbers are stored in 32 bits, with

a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa (the fractional part). This provides approximately 7.2 significant decimal digits.

Floating-point constant expressions can be defined and used for many compile-time purposes, but for run-time floating-point operations, the FloatMath and FloatString objects provide math functions compatible with single-precision numbers.

See the Constant Assignment '=' in the Operators section on page 148, FLOAT on page 108, and TRUNC on page 209, as well as the FloatMath and FloatString objects for more information.

SPR

Register: Special Purpose Register array; provides indirect access to cog's special registers.

((PUB ¦ PRI))
SPR [*Index*]

Returns: Value in special purpose register at *Index*.

- ***Index*** is an expression that specifies the index (0-15) of the special purpose register to access (PAR through VSCL).

Explanation

SPR is an array of the 16 special purpose registers in the cog. Element 0 is the PAR register and element 15 is the VSCL register. See Table 2-15 below. SPR provides an indirect method of accessing the cog's special purpose registers.

Table 2-15: Cog RAM Special Purpose Registers

Name	Index	Type	Description
PAR	0	Read-Only	Boot Parameter
CNT	1	Read-Only	System Counter
INA	2	Read-Only	Input States for P31 - P0
INB	3	Read-Only	Input States for P63- P32[1]
OUTA	4	Read/Write	Output States for P31 - P0
OUTB	5	Read/Write	Output States for P63 – P32[1]
DIRA	6	Read/Write	Direction States for P31 - P0
DIRB	7	Read/Write	Direction States for P63 - P32[1]
CTRA	8	Read/Write	Counter A Control
CTRB	9	Read/Write	Counter B Control
FRQA	10	Read/Write	Counter A Frequency
FRQB	11	Read/Write	Counter B Frequency
PHSA	12	Read/Write	Counter A Phase
PHSB	13	Read/Write	Counter B Phase
VCFG	14	Read/Write	Video Configuration
VSCL	15	Read/Write	Video Scale

Note 1: Reserved for future use

Using SPR

SPR can be used like any other long-sized array. The following assumes Temp is a variable defined elsewhere.

```
spr[4] := %11001010          'Set outa register
Temp := spr[2]               'Get ina value
```

This example sets the OUTA register (index 4 of SPR) to %11001010 and then sets Temp equal to the INA register (index 2 of SPR).

_STACK

Constant: Pre-defined, one-time settable constant for specifying the size of an application's stack space.

```
CON
  _STACK = Expression
```

- ***Expression*** is an integer expression that indicates the number of longs to reserve for stack space.

Explanation

_STACK is a pre-defined, one-time settable optional constant that specifies the required stack space of an application. This value is added to _FREE if specified, to determine the total amount of stack/free memory space to reserve for a Propeller application. Use _STACK if an application requires a minimum amount of stack space in order to run properly. If the resulting compiled application is too large to allow the specified stack space, an error message will be displayed. For example:

```
CON
  _STACK   = 3000
```

The _STACK declaration in the above CON block indicates that the application needs to have at least 3,000 longs of stack space left over after compilation. If the resulting compiled application does not have that much room left over, an error message will indicate by how much it was exceeded. This is a good way to prevent successful compiles of an application that will fail to run properly due to lack of memory.

Note that only the top object file can set the value of _STACK. Any child object's _STACK declarations will be ignored. The stack space reserved by this constant is used by the application's main cog to store temporary data such as call stacks, parameters, and intermediate expression results.

STRCOMP

Command: Compare two strings for equality.

((PUB ¦ PRI))
STRCOMP (*StringAddress1*, *StringAddress2*)

Returns: TRUE if both strings are equal, FALSE otherwise.

- ***StringAddress1*** is an expression specifying the starting address of the first string to compare.
- ***StringAddress2*** is an expression specifying the starting address of the second string to compare.

Explanation

STRCOMP is one of two commands (STRCOMP and STRSIZE) that retrieve information about a string. STRCOMP compares the contents of the string at *StringAddress1* to the contents of the string at *StringAddress2*, up to the zero-terminator of each string, and returns TRUE if both strings are equivalent, FALSE otherwise. This comparison is case-sensitive.

Using STRCOMP

The following example assumes PrintStr is a method created elsewhere.

```
PUB Main
  if strcomp(@Str1, @Str2)
    PrintStr(string("Str1 and Str2 are equal"))
  else
    PrintStr(string("Str1 and Str2 are different"))

DAT
  Str1  byte  "Hello World", 0
  Str2  byte  "Testing.", 0
```

The above example has two zero-terminated strings in the DAT block, Str1 and Str2. The Main method calls STRCOMP to compare the contents of each string. Assuming PrintStr is a method that displays a string, this example prints "Str1 and Str2 are different" on the display.

Zero-Terminated Strings

The STRCOMP command requires the strings being compared to be zero-terminated; a byte equal to 0 must immediately follow each string. This practice is quite common and is recommended since most string-handling methods rely on zero terminators.

STRING

Directive: Declare in-line string constant and get its address.

((PUB ¦ PRI))
  STRING (*StringExpression*)

Returns: Address of in-line string constant.

- ***StringExpression*** is the desired string expression to be used for temporary, in-line purposes.

Explanation

The DAT block is used often to create strings or string buffers that are reusable for various purposes, but there are occasions when a string is needed for temporary purposes like debugging or one-time uses in an object. The STRING directive is meant for those one-time uses; it compiles an in-line, zero-terminated string into memory and returns the address of that string.

Using STRING

The STRING directive is very good for creating one-time-use strings and passing the address of that string to other methods. For example, assuming PrintStr is a method created elsewhere.

```
PrintStr(string(''This is a test string.''))
```

The above example uses the STRING directive to compile a string, "This is a test string.", into memory and return the address of that string as the parameter for the fictitious PrintStr method.

If a string needs to be used in more than one place in code, it is better to define it in the DAT block so the address can be used multiple times.

STRSIZE

Command: Get size of string.

((PUB ¦ PRI))
STRSIZE (***StringAddress***)

Returns: Size (in bytes) of zero-terminated string.

- ***StringAddress*** is an expression specifying the starting address of the string to measure.

Explanation

STRSIZE is one of two commands (STRCOMP and STRSIZE) that retrieve information about a string. STRSIZE measures the length of a string at *StringAddress*, in bytes, up to, but not including, a zero-termination byte.

Using STRSIZE

The following example assumes Print is a method created elsewhere.

```
PUB Main
  Print(strsize(@Str1))
  Print(strsize(@Str2))

DAT
  Str1  byte  "Hello World", 0
  Str2  byte  "Testing.", 0
```

The above example has two zero-terminated strings in the DAT block, Str1 and Str2. The Main method calls STRSIZE to get the length of each string. Assuming Print is a method that displays a value, this example prints 11 and 8 on the display.

Zero-Terminated Strings

The STRSIZE command requires the string being measured to be zero-terminated; a byte equal to 0 must immediately follow the string. This practice is quite common and is recommended since most string-handling methods rely on zero terminators.

Symbols

IMPROVED

The symbols in Table 2-16 below serve one or more special purposes in Spin code. For Propeller Assembly symbols, see Symbols, page 360. Each symbol's purpose is described briefly with references to other sections that describe it directly or use it in examples.

Table 2-16: Symbols

Symbol	Purpose(s)
%	Binary indicator: used to indicate that a value is being expressed in binary (base-2). See Value Representations on page 45.
%%	Quaternary indicator: used to indicate a value is being expressed in quaternary (base-4). See Value Representations on page 45.
$	Hexadecimal indicator: used to indicate a value is being expressed in hexadecimal (base-16). See Value Representations on page 45.
''	String designator: used to begin and end a string of text characters. Usually used in Object blocks (page 141), Data blocks (page 99), or in Public/Private blocks with the STRING directive (page 205).
NEW @	Symbol Address Indicator: used immediately before a symbol to indicate the address of that symbol is to be used, rather than the value at that symbol's location. See Symbol Address '@' on page 173.
NEW @@	Object Address Plus Symbol Indicator: used immediately before a symbol to indicate the value of that symbol should be added to the object's base address. See Object Address Plus Symbol '@@' on page 173.
_	1) Delimiter: used as a group delimiter in constant values (where a comma ',' or period '.' may normally be used as a number group delimiter). See Value Representations on page 45. 2) Underscore: used as part of a symbol. See Symbol Rules on page 45.
#	1) Object-Constant reference: used to reference a sub-object's constants. See the CON section's Scope of Constants, page 89, and OBJ, page 141. 2) Enumeration Set: used in a CON block to set the start of an enumerated set of symbols. See the CON section's Enumerations (Syntax 2 and 3) on page 87.
.	1) Object-Method reference: used to reference a sub-object's methods. See OBJ, page 141. 2) Decimal point: used in floating-point constant expressions. See CON, page 84.
..	Range indicator: indicates a range from one expression to another for CASE statements or an I/O register index. See OUTA, OUTB on page 175, INA, INB on page 118, and DIRA, DIRB on page 104.

(Table continues on next page.)

Table 2-16: Symbols (continued)	
Symbol	**Purpose(s)**
:	1) Return value separator: appears immediately before a symbolic return value on a PUB or PRI declaration. See PUB on page 182, PRI on page 181, and RESULT on page 194. 2) Object assignment: appears in an object reference declaration in an OBJ block. See OBJ, page 141. 3) Case statement separator: appears immediately after the match expressions in a CASE structure. See CASE, page 59.
|	1) Local variable separator: appears immediately before a list of local variables on a PUB or PRI declaration. See PUB, page 182 and PRI, page 181. 2) Bitwise OR: used in expressions. See Bitwise OR '|', '|=' on page 165.
\	Abort trap: appears immediately before a method call that could potentially abort. See ABORT on page 47.
,	List delimiter: used to separate items in lists. See LOOKUP, LOOKUPZ on page 138, LOOKDOWN, LOOKDOWNZ on page 136, and the DAT section's Declaring Data(Syntax 1) on page 100.
( )	Parameter list designators: used to surround method parameters. See PUB, page 182 and PRI, page 181.
[ ]	Array index designators: used to surround indexes on variable arrays or main memory references. See VAR, page 210; BYTE, page 51; WORD, page 227; and LONG, page 128.
'	Code comment designator: used to enter single-line code comments (non-compiled text) for code viewing purposes. See "Using the Propeller Tool" in the software's Help file.
''	Document comment designator: used to enter single-line document comments (non-compiled text) for documentation viewing purposes. See "Using the Propeller Tool" in the software's Help file.
{ }	In-line/multi-line code comment designators: used to enter multi-line code comments (non-compiled text) for code viewing purposes.
{{ }}	In-line/multi-line document comment designators: used to enter multi-line document comments (non-compiled text) for documentation viewing purposes. See "Using the Propeller Tool" in the software's Help file.

TRUNC

Directive: Remove, "truncate," the fractional portion from a floating-point constant.

((CON ¦ VAR ¦ OBJ ¦ PUB ¦ PRI ¦ DAT))
  TRUNC (*FloatConstant*)

Returns: Integer that is the given floating-point constant value truncated at the decimal point.

- ***FloatConstant*** is the floating-point constant expression to be truncated to an integer.

Explanation

TRUNC is one of three directives (FLOAT, ROUND and TRUNC) used for floating-point constant expressions. TRUNC returns an integer constant that is the given floating-point constant expression with the fractional portion removed.

Using TRUNC

TRUNC can be used to retrieve the integer portion of a floating-point constant. For example:

```
CON
  OneHalf = 0.5
  Bigger  = 1.4999
  Int1    = trunc(OneHalf)
  Int2    = trunc(Bigger)
  Int3    = trunc(Bigger * 10.0) + 4
```

The above code creates two floating-point constants, OneHalf and Bigger, equal to 0.5 and 1.4999, respectively. The next three constants, Int1, Int2 and Int3, are integer constants that are based on OneHalf and Bigger using the TRUNC directive. Int1 = 0, Int2 = 1, and Int3 = 18.

About Floating-Point Constants

The Propeller compiler handles floating-point constants as a single-precision real number as described by the IEEE-754 standard. Single-precision real numbers are stored in 32 bits, with a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa (the fractional part). This provides approximately 7.2 significant decimal digits.

Floating-point constant expressions can be defined and used for many compile-time purposes, but for run-time floating-point operations, the FloatMath and FloatString objects provide math functions compatible with single-precision numbers. See the Constant Assignment '=' in the Operators section on page 148, FLOAT on page 108, and ROUND on page 198, as well the FloatMath and FloatString objects for more information.

VAR

Designator: Declare a Variable Block.

VAR
  Size Symbol ⟨[*Count*]⟩ ⟨ ↳ *Size Symbol* ⟨ [*Count*]⟩⟩...

VAR
  Size Symbol ⟨[*Count*]⟩ ⟨ , *Symbol* ⟨ [*Count*]⟩⟩...

- ***Size*** is the desired size of the variable, BYTE, WORD or LONG.
- ***Symbol*** is the desired name for the variable.
- ***Count*** is an optional expression, enclosed in brackets, that indicates this is an array variable, with *Count* number of elements; each being of size byte, word or long. When later referencing these elements, they begin with element 0 and end with element *Count*-1.

Explanation

VAR is the Variable Block declaration. The Variable Block is a section of source code that declares global variable symbols. This is one of six special declarations (CON, VAR, OBJ, PUB, PRI, and DAT) that provide inherent structure to the Spin language.

Variable Declarations (Syntax 1)

The most common form of variable declarations begins with VAR on a line by itself followed by one or more declarations. VAR must start in column 1 (the leftmost column) of the line it is on and the lines following it must be indented by at least one space.

```
VAR
  byte  Str[10]
  word  Code
  long  LargeNumber
```

This example defines Str as a byte array of 10 elements, Code as a word (two bytes) and LargeNumber as a long (four bytes). Public and Private methods in the same object can refer to these variables in ways similar to the following:

```
PUB SomeMethod
  Code := 60000
  LargeNumber := Code * 250
  GetString(@Str)
  if Str[0] == "A"
    <more code here>
```

Notice that Code and LargeNumber are used directly by expressions. The Str reference in the GetString method's parameter list looks different; it has an @, the Symbol Address operator, preceding it. This is because our fictitious GetString method needs to write back to the Str variable. If we had said GetString(Str), then the first byte of Str, element 0, would have been passed to GetString. By using the Symbol Address operator, @, we caused the address of Str to be passed to GetString instead; GetString can use that address to write to Str's elements. Lastly, we use Str[0] in the condition of an IF statement to see if the first byte is equal to the character "A". Remember, the first element of an array is always zero (0).

Variable Declarations (Syntax 2)

A variation on Syntax 1 allows for comma-delimited variables of the same size. The following is, similar to the above example, but we declare two words, Code and Index.

```
VAR
  byte  Str[10]
  word  Code, Index
  long  LargeNumber
```

NEW

Range of Variables

The range and nature of each type of variable is as follows:

- BYTE – 8 bits (unsigned); 0 to 255.
- WORD – 16 bits (unsigned); 0 to 65,535.
- LONG – 32 bits (signed); -2,147,483,648 to +2,147,483,647.

For more details regarding their range and usage, see BYTE on page 51, WORD on page 227, and LONG on page 128.

NEW

Organization of Variables

During compilation of an object, all declarations in its collective VAR blocks are grouped together by type. The variables in RAM are arranged with all the longs first, followed by all words, and finally by all bytes. This is done so that RAM space is allocated efficiently without unnecessary gaps. Keep this in mind when writing code that accesses variables indirectly based on relative positions to each other.

NEW

Optimized Addressing

In the compiled Propeller Application, the first eight (8) global long-sized variables are addressed using an optimized encoding scheme. This means accessing the first eight global long variables takes slightly less time than accessing the ninth, or later, long variables. Word and byte variables do not have an optimized addressing scheme. To optimize execution speed, ensure that all global variables used by the application's most repetitive loops are among the first eight longs. A similar mechanism applies to local variables; see the PUB section, page 185, for more information.

Scope of Variables

IMPROVED

Symbolic variables defined in VAR blocks are global to the object in which they are defined but not outside of that object. This means that these variables can be accessed directly by any public and private method within the object but those variable names will not conflict with symbols defined in other parent or child objects.

Note that public and private methods also have the ability to declare their own local variables. See PUB, page 182, and PRI, page 181.

Global variables are not accessible outside of an object unless the address of that variable is passed into, or back to, another object through a method call.

NEW

Scope Extends Beyond a Single Cog

The scope of global variables is not limited to a single cog either. An object's public and private methods naturally have access to its global variables regardless of what cog they are running on. If the object launches some of its methods into multiple cogs, each of those methods, and thus the cogs they are running on, have access to those variables.

This feature allows a single object to easily manage aspects of multiple processes in a centralized way. For example, a sophisticated object may take advantage of this to instantly affect "global" settings used by every multi-processed code entity that it created.

Of course, care should be used to ensure that multiple processing on the same block of variables does not create undesirable situations. See LOCKNEW on page 122 for examples.

VCFG

Register: Video Configuration Register.

((PUB ¦ PRI))
  VCFG

Returns: Current value of cog's Video Configuration Register, if used as a source variable.

Explanation

VCFG is one of two registers (VCFG and VSCL) that affect the behavior of a cog's Video Generator. Each cog has a video generator module that facilitates transmitting video image data at a constant rate. The VCFG register contains the configuration settings of the video generator, as shown in Table 2-17.

Table 2-17: VCFG Register

VCFG Bits									
31	**30..29**	**28**	**27**	**26**	**25..23**	**22..12**	**11..9**	**8**	**7..0**
-	VMode	CMode	Chroma1	Chroma0	AuralSub	-	VGroup	-	VPins

In Propeller Assembly, the VMode field through AuralSub fields can conveniently be written using the MOVI instruction, the VGroup field can be written with the MOVD instruction, and the VPins field can be written with the MOVS instruction.

VMode

The 2-bit VMode (video mode) field selects the type and orientation of video output, if any, according to Table 2-18.

Table 2-18: The Video Mode Field

VMode	Video Mode
00	Disabled, no video generated.
01	VGA mode; 8-bit parallel output on VPins7:0
10	Composite Mode 1; broadcast on VPins 7:4, baseband on VPins 3:0
11	Composite Mode 2; baseband on VPins 7:4, broadcast on VPins 3:0

CMode

The CMode (color mode) field selects two or four color mode. 0 = two-color mode; pixel data is 32 bits by 1 bit and only colors 0 or 1 are used. 1 = four-color mode; pixel data is 16 bits by 2 bits, and colors 0 through 3 are used.

Chroma1

The Chroma1 (broadcast chroma) bit enables or disables chroma (color) on the broadcast signal. 0 = disabled, 1 = enabled.

Chroma0

The Chroma0 (baseband chroma) bit enables or disables chroma (color) on the baseband signal. 0 = disabled, 1 = enabled.

AuralSub

The AuralSub (aural sub-carrier) field selects the source of the FM aural (audio) sub-carrier frequency to be modulated on. The source is the PLLA of one of the cogs, identified by AuralSub's value.

Table 2-19: The AuralSub Field

AuralSub	Sub-Carrier Frequency Source
000	Cog 0's PLLA
001	Cog 1's PLLA
010	Cog 2's PLLA
011	Cog 3's PLLA
100	Cog 4's PLLA
101	Cog 5's PLLA
110	Cog 6's PLLA
111	Cog 7's PLLA

VGroup

The VGroup (video output pin group) field selects which group of eight I/O pins to output video on.

Table 2-20: The VGroup Field	
VGroup	**Pin Group**
000	Group 0: P7..P0
001	Group 1: P15..P8
010	Group 2: P23..P16
011	Group 3: P31..P24
100-111	<reserved for future use>

VPins

The VPins (video output pins) field is a mask applied to the pins of VGroup that indicates which pins to output video signals on.

Table 2-21: The VPins Field	
VPins	**Effect**
00001111	Drive Video on lower 4 pins only; composite
11110000	Drive Video on upper 4 pins only; composite
11111111	Drive video on all 8 pins; VGA

Using VCFG

VCFG can be read/written like other registers or pre-defined variables. For example:

```
VCFG := %0_10_1_0_1_000_00000000000_001_0_00001111
```

This sets the video configuration register to enable video in composite mode 1 with 4 colors, baseband chroma (color) enabled, on pin group 1, lower 4 pins (which is pins P11:8).

VSCL

Register: Video Scale Register.

((PUB ¦ PRI))
  VSCL

Returns: Current value of cog's Video Scale Register, if used as a source variable.

Explanation

VSCL is one of two registers (VCFG and VSCL) that affect the behavior of a cog's Video Generator. Each cog has a video generator module that facilitates transmitting video image data at a constant rate. The VSCL register sets the rate at which video data is generated.

Table 2-22: VSCL Register		
VSCL Bits		
31..20	19..12	11..0
–	PixelClocks	FrameClocks

PixelClocks

The 8-bit PixelClocks field indicates the number of clocks per pixel; the number of clocks that should elapse before each pixel is shifted out by the video generator module. These clocks are the PLLA clocks, not the System Clock.

FrameClocks

The 12-bit FrameClocks field indicates the number of clocks per frame; the number of clocks that should elapse before each frame is shifted out by the video generator module. These clocks are the PLLA clocks, not the System Clock. A frame is one long of pixel data (delivered via the WAITVID command). Since the pixel data is either 16 bits by 2 bits, or 32 bits by 1 bit (meaning 16 pixels wide with 4 colors, or 32 pixels wide with 2 colors, respectively), the FrameClocks is typically 16 or 32 times that of the PixelClocks value.

Using VSCL

VSCL can be read/written like other registers or pre-defined variables. For example:

```
VSCL := %000000000000_10100000_101000000000
```

This sets the video scale register for 160 PixelClocks and 2,560 FrameClocks (for a 16-pixel by 2-bit color frame). Of course, the actual rate at which pixels clock out depends on the frequency of PLLA in combination with this scale factor.

WAITCNT

Command: Pause a cog's execution temporarily.

((PUB ¦ PRI))
  WAITCNT (*Value*)

- ***Value*** is the desired 32-bit System Counter value to wait for.

Explanation

WAITCNT, "Wait for System Counter," is one of four wait commands (WAITCNT, WAITPEQ, WAITPNE, and WAITVID) used to pause execution of a cog until a condition is met. WAITCNT pauses the cog until the global System Counter equals *Value*.

When executed, WAITCNT activates special "wait" hardware in the cog that prevents the System Clock from causing further code execution within the cog until the moment the System Counter equals *Value*. The wait hardware checks the System Counter every System Clock cycle and the cog's power consumption is reduced by approximately 7/8^{ths} during this time. In normal applications, WAITCNT may be used strategically to reduce power consumption anywhere in the program where time is wasted waiting for low-bandwidth events.

There are two types of delays WAITCNT can be used for: fixed delays and synchronized delays. Both are explained below.

Fixed Delays

Fixed delays are those that are all unrelated to one specific point in time and only serve the purpose of pausing execution for a fixed amount of time. A fixed delay, for example, may be used to wait for 10 milliseconds after an event occurs, before proceeding with another action. For example:

```
CON
  _clkfreq = xtal1                'Set for slow crystal
  _xinfreq = 5_000_000            'Use 5 MHz accurate crystal

   repeat
    !outa[0]                      'Toggle pin 0
    waitcnt(50_000 + cnt)         'Wait for 10 ms
```

This code toggles the state of I/O pin P0 and waits for 50,000 system clock cycles before repeating the loop again. Remember, the *Value* parameter must be the desired 32-bit value to match against the System Clock's value. Since the System Clock is a global resource that changes every clock cycle, to delay for a certain number of cycles from "now" we need a value that is added to the current System Counter value. The `cnt` in "`50_000 + cnt`" is the System Counter Register variable; it returns the value of the System Counter at that moment in time. So our code says to wait for 50,000 cycles plus the current value of the System Counter; i.e.: wait 50,000 cycles from now. Assuming that an external 5 MHz crystal is being used, 50,000 cycles is about 10 ms ($1/100^{th}$ second) of time.

IMPORTANT: Since **WAITCNT** pauses the cog until the System Counter matches the given value, care must be taken to ensure that the given value was not already surpassed by the System Counter. If the System Counter already passed the given value before the wait hardware activated then the cog will appear to have halted permanently when, in fact, it is waiting for the counter to exceed 32 bits and wrap around to the given value. Even at 80 MHz, it takes over 53 seconds for the 32-bit System Counter to wrap around!

Related to this, when using **WAITCNT** in Spin code as shown above, make sure to write the *Value* expression the same way we did: in the form "`offset + cnt`" as opposed to "`cnt + offset`." This is because the Spin interpreter will evaluate this expression from left to right, and each intermediate evaluation within an expression takes time to perform. If `cnt` were at the start of the expression, the System Counter would be read first then the rest of the expression would be evaluated, taking an unknown amount of cycles and making our `cnt` value quite old by the time the final result is calculated. However, having `cnt` as the last value in the **WAITCNT** expression ensures a fixed amount of overhead (cycles) between reading the System Counter and activating the wait hardware. In fact, the interpreter takes 381 cycles of final overhead when the command is written in the form `waitcnt(offset + cnt)`. **This means the value of `offset` must always be at least 381 to avoid unexpectedly long delays.**

Synchronized Delays

Synchronized delays are those that are all directly related to one specific point in time, a "base" time, and serve the purpose of "time-aligning" future events relative to that point. A synchronized delay, for example, may be used to output or input a signal at a specific interval, despite the unknown amounts of overhead associated with the code itself. To understand how this is different from the Fixed Delay example, let's look at that example's timing diagram.

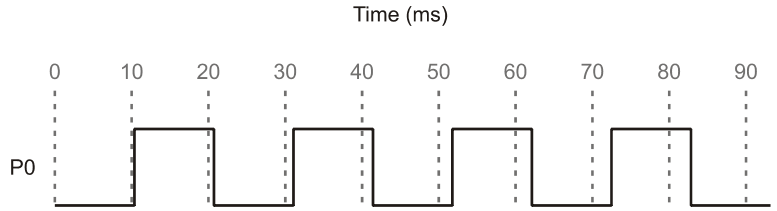


Figure 2-3: Fixed Delay Timing

Figure 2-3 shows the output of our previous example, the fixed delay example. Notice how the I/O pin P0 toggles roughly every 10 milliseconds, but not exactly? In fact, there's a cumulative error that makes successive state changes further and further out-of-sync in relation to our start time, 0 ms. The delay is 10 ms in length, but the error occurs because that delay doesn't compensate for the length of the rest of the loop. The `repeat, !outa[0]` and **WAITCNT** statements each take a little time to execute, and all that extra time is in addition to the 10 ms delay that **WAITCNT** specified.

Using **WAITCNT** a slightly different way, for a synchronized delay, will eliminate this timing error. The following example assumes we're using a 5 MHz external crystal.

```
CON
  _clkfreq = xtal1                'Set for slow crystal
  _xinfreq = 5_000_000            'Use 5 MHz accurate crystal

PUB Toggle | Time
  Time := cnt                     'Get current system counter value
  repeat
    waitcnt(Time += 50_000)       'Wait for 10 ms
    !outa[0]                      'Toggle pin 0
```

This code first retrieves the value of the System Counter, `Time := cnt`, then starts the `repeat` loop where it waits for the System Counter to reach `Time` + 50,000, toggles the state of I/O pin P0 and repeats the loop again. The statement `Time += 50_000` is actually an assignment statement; it adds the value of `Time` to 50,000, stores that result back into `Time` and then executes the **WAITCNT** command using that result. Notice that we retrieved the System Counter's value only once, at the start of the example; that is our base time. Then we wait for the System Counter to equal that original base time plus 50,000 and perform the actions in the loop. Each successive iteration through the loop, we wait for the System Counter to equal another multiple of 50,000 from the base time. This method automatically compensates for

the overhead time consumed by the loop statements: repeat, !outa[0] and waitcnt. The resulting output looks like Figure 2-4.

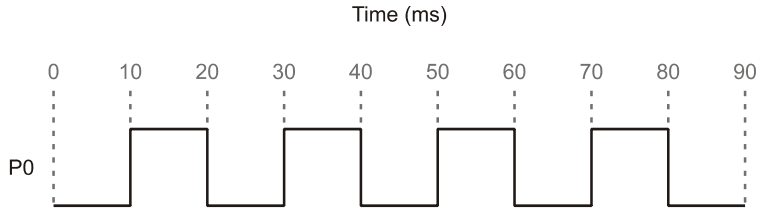


Figure 2-4: Synchronized Delay Timing

Using the synchronized delay method, our output signal is always perfectly aligned to the time base plus a multiple of our interval. This will work as long as the time base (an external crystal) is accurate and the overhead in the loop does not exceed the time interval itself. Note that we waited, with WAITCNT, before the first toggle so that the time between the very first toggle and the second matches that of all the rest.

Calculating Time

An object can delay a specific amount of time even if the application changes the System Clock frequency occasionally. To do this, use WAITCNT combined with an expression that includes the current System Clock frequency (CLKFREQ). For example, without you knowing what the actual clock frequency will be for applications using your object, the following line can be used to delay the cog for 1 millisecond; as long as the clock frequency is fast enough.

```
waitcnt(clkfreq / 1000 + cnt)        'delay cog 1 millisecond
```

For more information, see CLKFREQ on page 63.

WAITPEQ

Command: Pause a cog's execution until I/O pin(s) match designated state(s).

((PUB ¦ PRI))
  WAITPEQ (*State*, *Mask*, *Port*)

- ***State*** is the logic state(s) to compare the pin(s) against. It is a 32-bit value that indicates the high or low states of up to 32 I/O pins. *State* is compared against either (INA & *Mask*), or (INB & *Mask*), depending on *Port*.
- ***Mask*** is the desired pin(s) to monitor. *Mask* is a 32-bit value that contains high (1) bits for every I/O pin that should be monitored; low (0) bits indicate pins that should be ignored. Mask is bitwised-ANDed with the 32-bit port's input states and the resulting value is compared against the entire *State* value.
- ***Port*** is a 1-bit value indicating the I/O port to monitor; 0 = Port A, 1 = Port B. Only Port A exists on current (P8X32A) Propeller chips.

Explanation

WAITPEQ, "Wait for Pin(s) to Equal," is one of four wait commands (WAITCNT, WAITPEQ, WAITPNE, and WAITVID) used to pause execution of a cog until a condition is met. WAITPEQ pauses the cog until the value of *Port's* I/O pin states, bitwised-ANDed with *Mask*, matches that of *State*.

When executed, WAITPEQ activates special "wait" hardware in the cog that prevents the System Clock from causing further code execution within the cog until the moment the designated pin, or group of pins, equals the indicated state(s). The wait hardware checks the I/O pins every System Clock cycle and the cog's power consumption is reduced by approximately 7/8^{ths} during this time.

Using WAITPEQ

WAITPEQ is a great way to synchronize code to external events. For example:

```
waitpeq(%0100, %1100, 0)       'Wait for P3 & P2 to be low & high
outa[0] := 1                   'Set P0 high
```

The above code pauses the cog until I/O pin 3 is low and I/O pin 2 is high, then sets I/O pin 0 high.

Using Variable Pin Numbers

For Propeller objects, quite often it is necessary to monitor a single pin whose pin number is specified outside the object itself. An easy way to translate that pin number into the proper 32-bit *State* and *Mask* value is by using the Bitwise Decode operator "|<" (See 160 for more information). For example, if the pin number was specified by the variable Pin, and we needed to wait until that pin is high, we could use the following code:

```
waitpeq(|< Pin, |< Pin, 0)   'Wait for Pin to go high
```

The *Mask* parameter, |< Pin, evaluates to a long value where only one bit is high; the bit that corresponds to the pin number given by Pin.

Waiting for Transitions

If we needed to wait for a transition from one state to another (high-to-low, for example) we could use the following code:

IMPROVED

```
waitpeq(%100000, |< 5, 0)    'Wait for Pin 5 to go high
waitpeq(%000000, |< 5, 0)    'Then wait for Pin 5 to go low
```

This example first waits for P5 to go high, then waits for it to go low; a high-to-low transition. If we had used the second line of code without the first, the cog would not have paused at all if P5 had been low to start with.

WAITPNE

Command: Pause a cog's execution until I/O pin(s) do not match designated state(s).

((PUB ¦ PRI))
  WAITPNE (*State*, *Mask*, *Port*)

- ***State*** is the logic state(s) to compare the pins against. It is a 32-bit value that indicates the high or low states of up to 32 I/O pins. *State* is compared against either (INA & *Mask*), or (INB & *Mask*), depending on *Port*.
- ***Mask*** is the desired pin(s) to monitor. *Mask* is a 32-bit value that contains high (1) bits for every I/O pin that should be monitored; low (0) bits indicate pins that should be ignored. Mask is bitwised-ANDed with the 32-bit port's input states and the resulting value is compared against the entire *State* value.
- ***Port*** is a 1-bit value indicating the I/O port to monitor; 0 = Port A, 1 = Port B. Only Port A exists on current (P8X32A) Propeller chips.

Explanation

WAITPNE, "Wait for Pin(s) to Not Equal," is one of four wait commands (WAITCNT, WAITPEQ, WAITPNE, and WAITVID) used to pause execution of a cog until a condition is met. WAITPNE is the complimentary form of WAITPEQ; it pauses the cog until the value of *Port's* I/O pin states, bitwised-ANDed with *Mask*, does not match that of *State*.

When executed, WAITPNE activates special "wait" hardware in the cog that prevents the System Clock from causing further code execution within the cog until the moment the designated pin, or group of pins, does not equal the designated state(s). The wait hardware checks the I/O pins every System Clock cycle and the cog's power consumption is reduced by approximately 7/8^{ths} during this time.

Using WAITPNE

WAITPNE is a great way to synchronize code to external events. For example:

```
waitpeq(%0100, %1100, 0)  'Wait for P3 & P2 to be low & high
waitpne(%0100, %1100, 0)  'Wait for P3 & P2 to not match prev. state
outa[0] := 1              'Set P0 high
```

The above code pauses the cog until P3 is low and P2 is high, then pauses the cog again until one or both of those pins changes states, then it sets P0 high.

WAITVID

Command: Pause a cog's execution until its Video Generator is available to take pixel data.

((PUB ¦ PRI))
  WAITVID (*Colors*, *Pixels*)

- ***Colors*** is a long containing four byte-sized color values, each describing the four possible colors of the pixel patterns in *Pixels*.
- ***Pixels*** is the next 16-pixel by 2-bit (or 32-pixel by 1-bit) pixel pattern to display.

Explanation

WAITVID, "Wait for Video Generator," is one of four wait commands (WAITCNT, WAITPEQ, WAITPNE, and WAITVID) used to pause execution of a cog until a condition is met. WAITVID pauses the cog until its Video Generator hardware is ready for the next pixel data, then the Video Generator accepts that data and the cog continues execution with the next line of code.

When executed, WAITVID activates special "wait" hardware in the cog that prevents the System Clock from causing further code execution within the cog until the moment the Video Generator is ready. The wait hardware checks the Video Generator's status every System Clock cycle and the cog's power consumption is reduced significantly during this time.

Using WAITVID

WAITVID is simply a delivery mechanism for data to the cog's Video Generator hardware. Since the Video Generator works independently from the cog itself, the two must synchronize each time data is needed for the display device. The frequency at which this occurs depends on the display device and the corresponding settings for the Video Generator, but in every case, the cog must have new data available the moment the Video Generator is ready for it. The cog uses the WAITVID command to wait for the right time and then "hand off" this data to the Video Generator.

The *Colors* parameter is a 32-bit value containing either four 8-bit color values (for 4-color mode) or two 8-bit color values in the lower 16 bits (for 2-color mode). For VGA, each color value's upper 6-bits is the 2-bit red, 2-bit green, and 2-bit blue color components describing the desired color; the lower 2-bits are "don't care" bits. Each of the color values corresponds to one of the four possible colors per 2-bit pixel (when *Pixels* is used as a 16x2 bit pixel pattern) or as one of the two possible colors per 1-bit pixel (when *Pixels* is used at a 32x1 bit pixel pattern).

Pixels describes the pixel pattern to display, either 16 pixels or 32 pixels depending on the color-depth configuration of the Video Generator.

Review the TV and VGA objects for examples of how WAITVID is used.

Make sure to start the cog's Video Generator module and Counter A before executing the WAITVID command or it will wait forever. See VCFG on page 213, VSCL on page 216, and CTRA, CTRB on page 95.

WORD

Designator: Declare word-sized symbol, word aligned/sized data, or read/write a word of main memory.

```
VAR
  WORD Symbol {[Count]}
```

```
DAT
  {Symbol} WORD Data {[Count]}
```

IMPROVED

```
((PUB | PRI))
  WORD [BaseAddress] {[Offset]}
```

```
((PUB | PRI))
  Symbol.WORD {[Offset]}
```

IMPROVED

- ***Symbol*** is the desired name for the variable (Syntax 1) or data block (Syntax 2) or is the existing name of the variable (Syntax 4).

IMPROVED

- ***Count*** is an optional expression indicating the number of word-sized elements for *Symbol* (Syntax 1), or the number of word-sized entries of *Data* (Syntax 2) to store in a data table.
- ***Data*** is a constant expression or comma-separated list of constant expressions.

IMPROVED

- ***BaseAddress*** is an expression describing the word-aligned address of main memory to read or write. If *Offset* is omitted, *BaseAddress* is the actual address to operate on. If *Offset* is specified, *BaseAddress* + *Offset* * 2 is the actual address to operate on.

IMPROVED

- ***Offset*** is an optional expression indicating the offset from *BaseAddress* to operate on, or the offset from word 0 of *Symbol*. *Offset* is in units of words.

Explanation

WORD is one of three multi-purpose declarations (BYTE, WORD, and LONG) that declare or operate on memory. WORD can be used to:

1) declare a word-sized (16-bit) symbol or a multi-word symbolic array in a VAR block, or
2) declare word-aligned, and/or word-sized, data in a DAT block, or
3) read or write a word of main memory at a base address with an optional offset, or
4) access a word within a long-sized variable.

NEW

Range of Word

Memory that is word-sized (16 bits) can contain a value that is one of 2^{16} possible combinations of bits (i.e., one of 65,536 combinations). This gives word-sized values a range of 0 to 65,535. Since the Spin language performs all mathematic operations using 32-bit signed math, any word-sized values will be internally treated as positive long-sized values. However, the actual numeric value contained within a word is subject to how a computer and user interpret it. For example, you may choose to use the Sign-Extend 15 operator (**~~**), page 157, in a Spin expression to convert a word value that you interpret as "signed" (-32,768 to +32,767) to a signed long value.

Word Variable Declaration (Syntax 1)

In **VAR** blocks, syntax 1 of **WORD** is used to declare global, symbolic variables that are either word-sized, or are any array of words. For example:

```
VAR
  word  Temp                    'Temp is a word (2 bytes)
  word  List[25]                'List is a word array
```

The above example declares two variables (symbols), `Temp` and `List`. `Temp` is simply a single, word-sized variable. The line under the `Temp` declaration uses the optional *Count* field to create an array of 25 word-sized variable elements called `List`. Both `Temp` and `List` can be accessed from any **PUB** or **PRI** method within the same object that this **VAR** block was declared; they are global to the object. An example of this is below.

```
PUB SomeMethod
  Temp := 25_000                'Set Temp to 25,000
  List[0] := 500                'Set first element of List to 500
  List[1] := 9_000              'Set second element of List to 9,000
  List[24] := 60_000            'Set last element of List to 60,000
```

For more information about using **WORD** in this way, refer to the **VAR** section's Variable Declarations (Syntax 1) on page 210, and keep in mind that **WORD** is used for the *Size* field in that description.

Word Data Declaration (Syntax 2)

In DAT blocks, syntax 2 of WORD is used to declare word-aligned, and/or word-sized data that is compiled as constant values in main memory. DAT blocks allow this declaration to have an optional symbol preceding it, which can be used for later reference. See DAT, page 99. For example:

```
DAT
  MyData  word  640, $AAAA, 5_500       'Word-aligned/word-sized data
  MyList  byte  word $FF99, word 1_000  'Byte-aligned/word-sized data
```

The above example declares two data symbols, MyData and MyList. MyData points to the start of word-aligned and word-sized data in main memory. MyData's values, in main memory, are 640, $AAAA and 5,500, respectively. MyList uses a special DAT block syntax of WORD that creates a byte-aligned but word-sized set of data in main memory. MyList's values, in main memory, are $FF99 and 1,000, respectively. When accessed a byte at a time, MyList contains $99, $FF, 232 and 3 since the data is stored in little-endian format.

This data is compiled into the object and resulting application as part of the executable code section and may be accessed using the read/write form, syntax 3, of WORD (see below). For more information about using WORD in this way, refer to the DAT section's Declaring Data(Syntax 1) on page 100 and keep in mind that WORD is used for the *Size* field in that description.

NEW

Data items may be repeated by using the optional *Count* field. For example:

```
DAT
  MyData      word  640, $AAAA[4], 5_500
```

The above example declares a word-aligned, word-sized data table, called MyData, consisting of the following six values: 640, $AAAA, $AAAA, $AAAA, $AAAA, 5500. There were four occurrences of $AAAA due to the [4] in the declaration immediately after it.

IMPROVED

Reading/Writing Words of Main Memory (Syntax 3)

In PUB and PRI blocks, syntax 3 of WORD is used to read or write word-sized values of main memory. This is done by writing expressions that refer to main memory using the form: word[*BaseAddress*][*Offset*]. Here's an example.

```
PUB MemTest | Temp
  Temp := word[@MyData][1]              'Read word value
  word[@MyList][0] := Temp + $0123      'Write word value
```

```
DAT
  MyData  word  640, $AAAA, 5_500       'Word-sized/aligned data
  MyList  byte  word $FF99, word 1_000  'Byte-sized/aligned word data
```

In this example, the DAT block (bottom of code) places its data in memory as shown in Figure 2-5. The first data element of MyData is placed at memory address $18. The last data element of MyData is placed at memory address $1C, with the first element of MyList immediately following it at $1E. Note that the starting address ($18) is arbitrary and is likely to change as the code is modified or the object itself is included in another application.

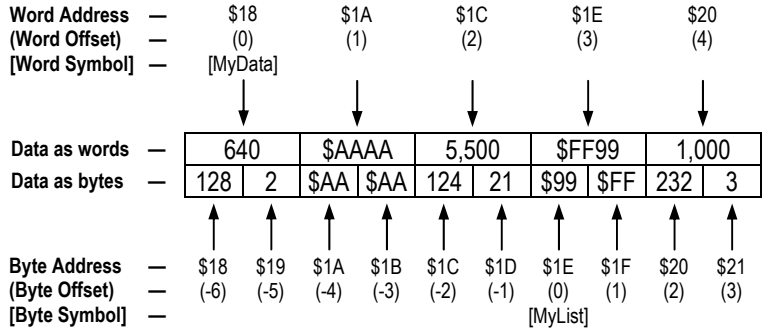


Figure 2-5: Main Memory Word-Sized Data Structure and Addressing

Near the top of the code, the first executable line of the MemTest method, Temp := word[@MyData][1], reads a word-sized value from main memory. It sets local variable Temp to $AAAA; the value read from main memory address $1A. The address $1A was determined by the address of the symbol MyData ($18) plus word offset 1 (2 bytes). The following progressive simplification demonstrates this.

```
word[@MyData][1] ➡ word[$18][1] ➡ word[$18 + (1*2)] ➡ word[$1A]
```

The next line, word[@MyList][0] := Temp + $0123, writes a word-sized value to main memory. It sets the value at main memory address $1E to $ABCD. The address $1E was calculated from the address of the symbol MyList ($1E) plus word offset 0 (0 bytes).

```
word[@MyList][0] ➡ word[$1E][0] ➡ word[$1E + (0*2)] ➡ word[$1E]
```

The value $ABCD was derived from the current value of Temp plus $0123; $AAAA + $0123 equals $ABCD.

NEW

Addressing Main Memory

As Figure 2-5 suggests, main memory is really just a set of contiguous bytes (see "data as bytes" row) that can also be read as words (2-byte pairs) when done properly. In fact, the above example shows that even the addresses are calculated in terms of bytes. This concept is a consistent theme for any commands that use addresses.

Main memory is ultimately addressed in terms of bytes regardless of the size of value you are accessing; byte, word, or long. This is advantageous when thinking about how bytes, words, and longs relate to each other, but it may prove problematic when thinking of multiple items of a single size, like words.

For this reason, the WORD designator has a very handy feature to facilitate addressing from a word-centric perspective. Its *BaseAddress* field when combined with the optional *Offset* field operates in a base-aware fashion.

Imagine accessing words of memory from a known starting point (the *BaseAddress*). You may naturally think of the next word or words as being a certain distance from that point (the *Offset*). While those words are indeed a certain number of "bytes" beyond a given point, it's easier to think of them as a number of "words" beyond a point (i.e., the 4th word, rather than the word that starts beyond the 6th byte). The WORD designator treats it properly by taking the *Offset* value (units of words), multiplies it by 2 (number of bytes per word), and adds that result to the *BaseAddress* to determine the correct word of memory to read. It also clears the lowest bit of *BaseAddress* to ensure the address referenced is a word-aligned one.

So, when reading values from the MyData list, word[@MyData][0] reads the first word value, word[@MyData][1] reads the second word value, and word[@MyData][2] reads the third.

If the *Offset* field were not used, the above statements would have to be something like word[@MyData], word[@MyData+2], and word[@MyData+4], respectively. The result is the same, but the way it's written may not be as clear.

For more explanation of how data is arranged in memory, see the DAT section's Declaring Data(Syntax 1) on page 100.

NEW

An Alternative Memory Reference

There is yet another way to access the data from the code example above; you could reference the data symbols directly. For example, this statement reads the first word of the MyData list:

```
Temp := MyData[0]
```

...and these statements read the second and third words of MyData:

```
Temp := MyData[1]
Temp := MyData[2]
```

So why wouldn't you just use direct symbol references all the time? Consider the following case:

```
Temp := MyList[0]
Temp := MyList[1]
```

Referring back to the example code above Figure 2-5, you might expect these two statements to read the first and second words of MyList; $FF99 and 1000, respectively. Instead, it reads the first and second "bytes" of MyList, $99 and $FF, respectively.

What happened? Unlike MyData, the MyList entry is defined in the code as byte-sized and byte-aligned data. The data does indeed consist of word-sized values, because each element is preceded by WORD, but since the symbol for the list is declared as byte-sized, all direct references to it will return individual bytes.

However, the WORD designator can be used instead, since the list also happens to be word-aligned because of its position following MyData.

```
Temp := word[@MyList][0]
Temp := word[@MyList][1]
```

The above reads the first word, $FF99, followed by the second word, 1000, of MyList. This feature is very handy should a list of data need to be accessed as both bytes and words at various times in an application.

NEW

Other Addressing Phenomena

Both the WORD and direct symbol reference techniques demonstrated above can be used to access any location in main memory, regardless of how it relates to defined data. Here are some examples:

```
Temp := word[@MyList][-1]  'Read last word of MyData (before MyList)
Temp := word[@MyData][3]   'Read first word of MyList (after MyData)
Temp := MyList[-6]         'Read first byte of MyData
Temp := MyData[-2]         'Read word that is two words before MyData
```

These examples read beyond the logical borders (start point or end point) of the lists of data they reference. This may be a useful trick, but more often it's done by mistake; be careful when addressing memory, especially if you're writing to that memory.

Accessing Words of Larger-Sized Symbols (Syntax 4)

In PUB and PRI blocks, syntax 4 of WORD is used to read or write word-sized components of long-sized variables. For example:

```
VAR
  long  LongVar

PUB Main
  LongVar.word := 65000      'Set first word of LongVar to 65000
  LongVar.word[0] := 65000   'Same as above
  LongVar.word[1] := 1       'Set second word of LongVar to 1
```

This example accesses the word-sized components of LongVar, individually. The comments indicate what each line is doing. At the end of the Main method LongVar will equal 130,536.

NEW

The same techniques can be used to reference word-sized components of long-sized data symbols.

```
PUB Main | Temp
  Temp := MyList.word[0]           'Read low word of MyList long 0
  Temp := MyList.word[1]           'Read high word of MyList long 0
  MyList.word[1] := $1234          'Write high word of MyList long 0
  MyList.word[2] := $FFEE          'Write low word of MyList long 1

DAT
  MyList  long  $FF998877, $DDDDEEEE  'Long-sized/aligned data
```

The first and second executable lines of Main read the values $8877 and $FF99, respectively, from MyList. The third line writes $1234 to the high word of the long in element 0 of MyList, resulting in a value of $12348877. The fourth line writes $FFEE to the word at index 2 in MyList (the low word of the long in element 1), resulting in a value of $DDDDFFEE.

WORDFILL

Command: Fill words of main memory with a value.

((PUB ¦ PRI))
  WORDFILL (*StartAddress*, *Value*, *Count*)

- ***StartAddress*** is an expression indicating the location of the first word of memory to fill with *Value*.
- ***Value*** is an expression indicating the value to fill words with.
- ***Count*** is an expression indicating the number of words to fill, starting with *StartAddress*.

Explanation

WORDFILL is one of three commands (BYTEFILL, WORDFILL, and LONGFILL) used to fill blocks of main memory with a specific value. WORDFILL fills *Count* words of main memory with *Value*, starting at location *StartAddress*.

Using WORDFILL

WORDFILL is a great way to clear large blocks of word-sized memory. For example:

```
VAR
  word   Buff[100]

PUB Main
  wordfill(@Buff, 0, 100)     'Clear Buff to 0
```

The first line of the Main method, above, clears the entire 100-word (200-byte) Buff array to all zeros. WORDFILL is faster at this task than a dedicated REPEAT loop is.

WORDMOVE

Command: Copy words from one region to another in main memory.

((PUB ¦ PRI))
  WORDMOVE (*DestAddress*, *SrcAddress*, *Count*)

- ***DestAddress*** is an expression specifying the main memory location to copy the first word of source to.
- ***SrcAddress*** is an expression specifying the main memory location of the first word of source to copy.
- ***Count*** is an expression indicating the number of words of the source to copy to the destination.

Explanation

WORDMOVE is one of three commands (BYTEMOVE, WORDMOVE, and LONGMOVE) used to copy blocks of main memory from one area to another. WORDMOVE copies *Count* words of main memory starting from *SrcAddress* to main memory starting at *DestAddress*.

Using WORDMOVE

WORDMOVE is a great way to copy large blocks of word-sized memory. For example:

```
VAR
  word   Buff1[100]
  word   Buff2[100]

PUB Main
  wordmove(@Buff2, @Buff1, 100)     'Copy Buff1 to Buff2
```

The first line of the Main method, above, copies the entire 100-word (200-byte) Buff1 array to the Buff2 array. WORDMOVE is faster at this task than a dedicated REPEAT loop.

_XINFREQ

Constant: Pre-defined, one-time settable constant for specifying the external crystal frequency.

```
CON
  _XINFREQ = Expression
```

- ***Expression*** is an integer expression that indicates the external crystal frequency; the frequency on the XI pin. This value is used for application start-up.

Explanation

_XINFREQ specifies the external crystal frequency, which is used along with the clock mode to determine the System Clock frequency at start-up. It is a pre-defined constant symbol whose value is determined by the top object file of an application. **_XINFREQ** is either set directly by the application itself, or is set indirectly as the result of the **_CLKMODE** and **_CLKFREQ** settings.

The top object file in an application (the one where compilation starts from) can specify a setting for **_XINFREQ** in its **CON** block. This, along with the clock mode, defines the frequency that the System Clock will switch to as soon as the application is booted up and execution begins.

The application can specify either **_XINFREQ** or **_CLKFREQ** in the **CON** block; they are mutually exclusive and the non-specified one is automatically calculated and set as a result of specifying the other.

The following examples assume that they are contained within the top object file. Any **_XINFREQ** settings in child objects are simply ignored by the compiler.

For example:

```
CON
  _CLKMODE = XTAL1 + PLL8X
  _XINFREQ = 4_000_000
```

The first declaration in the above **CON** block sets the clock mode for an external low-speed crystal and a Clock PLL multiplier of 8. The second declaration indicates the external crystal frequency is 4 MHz, which means the System Clock's frequency will be 32 MHz because 4 MHz * 8 = 32 MHz. The **_CLKFREQ** value is automatically set to 32 MHz because of these declarations.

```
CON
  _CLKMODE = XTAL2
  _XINFREQ = 10_000_000
```

These two declarations set the clock mode for an external medium-speed crystal, no Clock PLL multiplier, and an external crystal frequency of 10 MHz. The **_CLKFREQ** value, and thus the System Clock frequency, is automatically set to 10 MHz, as well, because of these declarations.

Chapter 3: Assembly Language Reference

This chapter describes all elements of the Propeller chip's Assembly language and is best used as a reference for individual elements of the assembly language. Many instructions have corresponding Spin commands so referring to the Spin Language Reference is also recommended.

The Assembly Language Reference is divided into three main sections:

1) **The Structure of Propeller Assembly**. Propeller Assembly code is an optional part of Propeller Objects. This section describes the general structure of Propeller Assembly code and how it fits within objects.

2) **The Categorical Listing of the Propeller Assembly Language.** All elements, including operators, are grouped by related function. This is a great way to quickly realize the breadth of the language and what features are available for specific uses. Each listed element has a page reference for more information. Some elements are marked with a superscript "s" indicating that they are also available in Propeller Spin, though syntax may vary.

3) **The Assembly Language Elements.** All instructions are listed in a Master Table at the start, and most elements have their own dedicated sub-section, alphabetically arranged to ease searching for them. Those individual elements without a dedicated sub-section, such as Operators, are grouped within other related sub-sections but can be easily located by following their page references from the Categorical Listing.

The Structure of Propeller Assembly

Every Propeller Object consists of Spin code plus optional assembly code and data. An object's Spin code provides it with structure, consisting of special-purpose blocks. Data and Propeller Assembly code are located in the DAT block; see DAT on page 99.

Spin code is executed from Main RAM by a cog running the Spin Interpreter, however, Propeller Assembly code is executed directly from within a cog itself. Because of this nature, Propeller Assembly code and any data belonging to it must be loaded (in its entirety) into a cog in order to execute it. In this way, both assembly code and data are treated the same during the cog loading process.

Here's an example Propeller object. Its Spin code in the PUB block, Main, launches another cog to run the DAT block's Propeller Assembly routine, Toggle.

```
{{ AssemblyToggle.spin }}

CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


PUB Main
{Launch cog to toggle P16 endlessly}

  cognew(@Toggle, 0)                            'Launch new cog


DAT
{Toggle P16}
                    org     0                   'Begin at Cog RAM addr 0
Toggle              mov     dira, Pin           'Set Pin to output
                    mov     Time, cnt           'Calculate delay time
                    add     Time, #9            'Set minimum delay here
:loop               waitcnt Time, Delay         'Wait
                    xor     outa, Pin           'Toggle Pin
                    jmp     #:loop              'Loop endlessly

Pin     long        |< 16                      'Pin number
Delay   long        6_000_000                   'Clock cycles to delay
Time                res  1                      'System Counter Workspace
```

NEW

When the Main method's COGNEW command is executed, a new cog begins filling its Cog RAM with 496 consecutive longs from Main Memory, starting with the instruction at the address of Toggle. Afterwards, the new cog initializes its special purpose registers and begins executing the code starting at Cog RAM register 0.

IMPROVED

Both assembly and data may be intermixed within this DAT block but care should be taken to arrange it such that all critical elements are loaded into the cog in the proper order for execution. It is recommended to write it in the following order: 1) assembly code, 2) initialized symbolic data (i.e.: LONGs), 3) reserved symbolic memory (i.e.: RESs). This causes the cog to load up the assembly code first, followed immediately by initialized data, and any

application data after that, whether or not it is required by the code. See the sections discussing ORG (page 328), RES (page 339), and DAT (page 99) for more information.

NEW

Cog Memory

Cog RAM is similar to Main RAM in the following ways:

- Each can contain program instruction(s) and/or data.
- Each can be modified at run-time (example: variables occupy RAM locations).

Cog RAM is different from Main RAM in the following ways:

- Cog RAM is smaller and faster than Main RAM.
- Cog RAM is a set of "registers" addressable only as longs (four bytes) while Main RAM is a set of "locations" addressable as bytes, words, or longs.
- Propeller Assembly executes right from Cog RAM while Spin code is fetched and executed from Main RAM.
- Cog RAM is available only to its own cog while Main RAM is shared by all cogs.

Once assembly code is loaded into Cog RAM, the cog executes it by reading a long (32-bit) opcode from a register (starting with register 0), resolving its destination and source data, executing the instruction (possibly writing the result to another register) and then moving on to the next register address to repeat the process. Cog RAM registers may contain instructions or pure data, and each may be modified as the result of the execution of another instruction.

NEW

Where Does an Instruction Get Its Data?

Most instructions have two data operands; a destination value and a source value. For example, the format for an ADD instruction is:

```
add   destination, {#}source
```

The *destination* operand is the 9-bit address of a register containing the desired value to operate on. The *source* operand is either a 9-bit literal value (constant) or a 9-bit address of a register containing the desired value. The meaning of the source operand depends on whether or not the literal indicator "#" was specified. For example:

```
   add  X, #25           'Add 25 to X
   add  X, Y             'Add Y to X

X  long  50
Y  long  10
```

The first instruction adds the literal value 25 to the value stored in the register X. The second instruction adds the value stored in register Y to the value stored in the register X. In both cases, the result of the addition is stored back into register X.

The last two lines define data symbols X and Y as long values 50 and 10, respectively. Since launching assembly code into the cog caused this data to enter Cog RAM right after the instructions, X naturally is a symbol that points to the register containing 50, and Y is a symbol that points to the register containing 10.

Thus, the result of the first ADD instruction is 75 (i.e.: X + 25 → 50 + 25 = 75) and that value, 75, is stored back in the X register. Similarly, the result of the second ADD instruction is 85 (i.e.: X + Y → 75 + 10 = 85) and so X is set to 85.

NEW

Don't Forget the Literal Indicator '#'

Make sure to enter the literal indicator, #, when a literal value (a.k.a. immediate value) is intended. Modifying the first line of the above example by omitting the # character (ex: ADD X, 25) causes the value in register 25 to be added to X instead of the value 25 being added to X.

Another possible mistake is to omit the # on branching instructions like JMP and DJNZ. If the intended branch destination is a label named MyRoutine, a JMP instruction should normally look like JMP #MyRoutine rather than JMP MyRoutine. The latter causes the value stored in the MyRoutine register to be used as the address to jump to; that's handy for indirect jumping but it is usually not the intention of the developer.

NEW

Literals Must Fit in 9 Bits

The source operand is only 9 bits wide; it can hold a value from 0 to 511 ($000 to $1FF). Keep this in mind when specifying literal values. If a value is too big to fit in 9 bits, it must be stored in a register and accessed via the register's address. For example:

```
   add  X, BigValue      'Add BigValue to X

X         long  50
BigValue  long  1024
```

NEW

Global and Local Labels

To give names to special routines, Propeller Assembly code can make use of two types of labels: global and local.

Global labels look just like other symbols and follow the same rules as symbols; they begin with an underscore '_' or a letter and are followed by more letters, underscores, and/or numbers. See Symbol Rules, page 45, for more information.

Local labels are similar to global labels except they start with a colon ':' and must be separated from other same-named local labels by at least one global label. Here's an example:

```
Addition            mov     Count, #9       'Set up 'Add' loop counter
:loop               add     Temp, X         'Iteratively do Temp+X
                    djnz    Count, #:loop   'Dec counter, loop back

Subtraction         mov     Count, #15      'Set up 'Sub' loop counter
:loop               sub     Temp, Y         'Iteratively do Temp-Y
                    djnz    Count, #:loop   'Dec counter, loop back

                    jmp     #Addition       'Go add more
```

This example has two global labels, `Addition` and `Subtraction`, and two local labels, both named `:loop`. The local labels can have the exact same name, `:loop`, because at least one global label separates them. In fact, this is the point of local labels; they indicate common, generic things like loops without requiring unique names for each of them.

The two **DJNZ** instructions are exactly the same, but they each jump to different places. The `Addition` routine's **DJNZ** jumps back to the local label `:loop` within `Addition`, and the `Subtraction` routine's **DJNZ** jumps back to the local label `:loop` within `Subtraction`.

Note that the **DJNZ** instructions use the literal indicator, `#`, and the exact name of the local label, including the colon. Without the `#` the code would execute improperly (jumping indirectly rather than directly), and without the colon the code would result in a compile error. For more Propeller Assembly format information, see Common Syntax Elements, page 250.

Categorical Listing of Propeller Assembly Language

Directives

ORG	Adjust compile-time cog address pointer; p 328.
FIT	Validate that previous instructions/data fit entirely in cog; p 292.
RES	Reserve next long(s) for symbol; p 338.

Configuration

CLKSET[s]	Set clock mode at run time; p 271.

Cog Control

COGID[s]	Get current cog's ID; p 283.
COGINIT[s]	Start, or restart, a cog by ID; p 284.
COGSTOP[s]	Stop a cog by ID; p 286.

Process Control

LOCKNEW[s]	Check out a new lock; p 304.
LOCKRET[s]	Return a lock; p 305.
LOCKCLR[s]	Clear a lock by ID; p 303.
LOCKSET[s]	Set a lock by ID; p 306.
WAITCNT[s]	Pause execution temporarily; p 368.
WAITPEQ[s]	Pause execution until pin(s) match designated state(s); p 369.
WAITPNE[s]	Pause execution until pin(s) do not match designated state(s); p 370.
WAITVID[s]	Pause execution until Video Generator is available for pixel data; p 371.

Conditions

IF_ALWAYS	Always; p 296.
IF_NEVER	Never; p 296.
IF_E	If equal (Z = 1); p 296.
IF_NE	If not equal (Z = 0); p 296.
IF_A	If above (!C & !Z = 1); p 296.

IF_B	If below (C = 1); p 296.
IF_AE	If above or equal (C = 0); p 296.
IF_BE	If below or equal (C | Z = 1); p 296.
IF_C	If C set; p 296.
IF_NC	If C clear; p 296.
IF_Z	If Z set; p296.
IF_NZ	If Z clear; p 296.
IF_C_EQ_Z	If C equal to Z; p 296.
IF_C_NE_Z	If C not equal to Z; p 296.
IF_C_AND_Z	If C set and Z set; p 296.
IF_C_AND_NZ	If C set and Z clear; p 296.
IF_NC_AND_Z	If C clear and Z set; p 296.
IF_NC_AND_NZ	If C clear and Z clear; p 296.
IF_C_OR_Z	If C set or Z set; p 296.
IF_C_OR_NZ	If C set or Z clear; p 296.
IF_NC_OR_Z	If C clear or Z set; p 296.
IF_NC_OR_NZ	If C clear or Z clear; p 296.
IF_Z_EQ_C	If Z equal to C; p 296.
IF_Z_NE_C	If Z not equal to C; p 296.
IF_Z_AND_C	If Z set and C set; p 296.
IF_Z_AND_NC	If Z set and C clear; p 296.
IF_NZ_AND_C	If Z clear and C set; p 296.
IF_NZ_AND_NC	If Z clear and C clear; p 296.
IF_Z_OR_C	If Z set or C set; p 296.
IF_Z_OR_NC	If Z set or C clear; p 296.
IF_NZ_OR_C	If Z clear or C set; p 296.
IF_NZ_OR_NC	If Z clear or C clear; p 296.

Flow Control

CALL	Jump to address with intention to return to next instruction; p 268.
DJNZ	Decrement value and jump to address if not zero; p 290.
JMP	Jump to address unconditionally; p 298.
JMPRET	Jump to address with intention to "return" to another address; p 300.
TJNZ	Test value and jump to address if not zero; p 362.
TJZ	Test value and jump to address if zero; p 365.
RET	Return to stored address; p 342.

Effects

NR	No result (don't write result); p 291.
WR	Write result; p 291.
WC	Write C status; p 291.
WZ	Write Z status; p 291.

Main Memory Access

RDBYTE	Read byte of main memory; p 335.
RDWORD	Read word of main memory; p 337.
RDLONG	Read long of main memory; p 336.
WRBYTE	Write a byte to main memory; p 374.
WRWORD	Write a word to main memory; p 376.
WRLONG	Write a long to main memory; p 375.

Common Operations

ABS	Get absolute value of a number; p 257.
ABSNEG	Get negative of number's absolute value; p 258.
NEG	Get negative of a number; p 319.
NEGC	Get a value, or its additive inverse, based on C; p 320.
NEGNC	Get a value or its additive inverse, based on !C; p 321.
NEGZ	Get a value, or its additive inverse, based on Z; p 323.

NEGNZ	Get a value, or its additive inverse, based on !Z; p 322.
MIN	Limit minimum of unsigned value to another unsigned value; p 309.
MINS	Limit minimum of signed value to another signed value; p 310.
MAX	Limit maximum of unsigned value to another unsigned value; p 307.
MAXS	Limit maximum of signed value to another signed value; p 308.
ADD	Add two unsigned values; p 259.
ADDABS	Add absolute value to another value; p 260.
ADDS	Add two signed values; p 262.
ADDX	Add two unsigned values plus C; p 264.
ADDSX	Add two signed values plus C; p 262.
SUB	Subtract two unsigned values; p 349.
SUBABS	Subtract an absolute value from another value; p 350.
SUBS	Subtract two signed values; p 351.
SUBX	Subtract unsigned value plus C from another unsigned value; p 354.
SUBSX	Subtract signed value plus C from another signed value; p 352.
SUMC	Sum signed value with another of C-affected sign; p 356.
SUMNC	Sum signed value with another of !C-affected sign; p 357.
SUMZ	Sum signed value with another Z-affected sign; p 359.
SUMNZ	Sum signed value with another of !Z-affected sign; p 358.
MUL	<reserved for future use>
MULS	<reserved for future use>
AND	Bitwise AND two values; p 266.
ANDN	Bitwise AND value with NOT of another; p 267.
OR	Bitwise OR two values; p 327.
XOR	Bitwise XOR two values; p 378.
ONES	<reserved for future use>
ENC	<reserved for future use>
RCL	Rotate C left into value by specified number of bits; p 333.
RCR	Rotate C right into value by specified number of bits; p 334.
REV	Reverse LSBs of value and zero-extend; p 343.
ROL	Rotate value left by specified number of bits; p 344.

ROR	Rotate value right by specified number of bits; p 345.
SHL	Shift value left by specified number of bits; p 347.
SHR	Shift value right by specified number of bits; p 348.
SAR	Shift value arithmetically right by specified number of bits; p 346.
CMP	Compare two unsigned values; p 272.
CMPS	Compare two signed values; p 274.
CMPX	Compare two unsigned values plus C; p 280.
CMPSX	Compare two signed values plus C; p 277.
CMPSUB	Compare unsigned values, subtract second if lesser or equal; p 276.
TEST	Bitwise AND two values to affect flags only; p 362.
NEW TESTN	Bitwise AND a value with NOT of another to affect flags only; p 363.
MOV	Set a register to a value; p 311.
MOVS	Set a register's source field to a value; p 313.
MOVD	Set a register's destination field to a value; p 312.
MOVI	Set a register's instruction field to a value; p 312.
MUXC	Set discrete bits of a value to the state of C; p 315.
MUXNC	Set discrete bits of a value to the state of !C; p 316.
MUXZ	Set discrete bits of a value to the state of Z; p 318.
MUXNZ	Set discrete bits of a value to the state of !Z; p 317.
HUBOP	Perform a hub operation; p 294.
NOP	No operation, just elapse four cycles; p 324.

Constants

NOTE: Refer to Constants (pre-defined) in Chapter 2: Spin Language Reference.

TRUE[s]	Logical true: -1 ($FFFFFFFF); p 93.
FALSE[s]	Logical false: 0 ($00000000); p 93.
POSX[s]	Maximum positive integer: 2,147,483,647 ($7FFFFFFF); p 94.
NEGX[s]	Maximum negative integer: -2,147,483,648 ($80000000); p 94.
PI[s]	Floating-point value for PI: ~3.141593 ($40490FDB); p 94.

Registers

DIRA[s]	Direction Register for 32-bit port A; p 338.
DIRB[s]	Direction Register for 32-bit port B (future use); p 338.
INA[s]	Input Register for 32-bit port A (read only); p 338.
INB[s]	Input Register for 32-bit port B (read only) (future use); p 338.
OUTA[s]	Output Register for 32-bit port A; p 338.
OUTB[s]	Output Register for 32-bit port B (future use); p 338.
CNT[s]	32-bit System Counter Register (read only); p 338.
CTRA[s]	Counter A Control Register; p 338.
CTRB[s]	Counter B Control Register; p 338.
FRQA[s]	Counter A Frequency Register; p 338.
FRQB[s]	Counter B Frequency Register; p 338.
PHSA[s]	Counter A Phase Lock Loop (PLL) Register; p 338.
PHSB[s]	Counter B Phase Lock Loop (PLL) Register; p 338.
VCFG[s]	Video Configuration Register; p 338.
VSCL[s]	Video Scale Register; p 338.
PAR[s]	Cog Boot Parameter Register (read only); p 338.

Unary Operators

NOTE: All operators shown are constant-expression operators.

+	Positive (+X) unary form of Add; p 326.
-	Negate (−X); unary form of Subtract; p 326.
^^	Square root; p 326.
||	Absolute Value; p 326.
|<	Decode value (0-31) into single-high-bit long; p 326.
>|	Encode long into value (0 - 32) as high-bit priority; p 326.
!	Bitwise: NOT; p 326.
@	Address of symbol; p 326.

Binary Operators

NOTE: All operators shown are constant expression operators.

+	Add; p 326.
-	Subtract; p 326.
*	Multiply and return lower 32 bits (signed); p 326.
**	Multiply and return upper 32 bits (signed); p 326.
/	Divide and return quotient (signed); p 326.
//	Divide and return remainder (signed); p 326.
#>	Limit minimum (signed); p 326.
<#	Limit maximum (signed); p 326.
~>	Shift arithmetic right; p 326.
<<	Bitwise: Shift left; p 326.
>>	Bitwise: Shift right; p 326.
<-	Bitwise: Rotate left; p 326.
->	Bitwise: Rotate right; p 326.
><	Bitwise: Reverse; p 326.
&	Bitwise: AND; p 326.
|	Bitwise: OR; p 326.
^	Bitwise: XOR; p 326.
AND	Boolean: AND (promotes non-0 to -1); p 326.
OR	Boolean: OR (promotes non-0 to -1); p 326.
==	Boolean: Is equal; p 326.
<>	Boolean: Is not equal; p 326.
<	Boolean: Is less than (signed); p 326.
>	Boolean: Is greater than (signed); p 326.
=<	Boolean: Is equal or less (signed); p 326.
=>	Boolean: Is equal or greater (signed); p 326.

Assembly Language Elements

Syntax Definitions

In addition to detailed descriptions, the following pages contain syntax definitions for many elements that describe, in short terms, all the options of that element. The syntax definitions use special symbols to indicate when and how certain element features are to be used.

BOLDCAPS	Items in bold uppercase should be typed in exactly as shown.
Bold Italics	Items in bold italics should be replaced by user text; symbols, operators, expressions, etc.
. : , #	Periods, colons, commas, and pound signs should be typed in where shown.
〈 〉	Angle bracket symbols enclose optional items. Enter the enclosed item if desired. Do not enter the angle brackets.
Double line	Separates instruction from the result value.

Common Syntax Elements

When reading the syntax definitions in this chapter, keep in mind that all Propeller Assembly instructions have three common, optional elements: a label, a condition, and effects. Each Propeller Assembly instruction has the following basic syntax:

IMPROVED 〈***Label***〉 〈***Condition***〉 ***Instruction Operands*** 〈***Effects***〉

- ***Label*** — an optional statement label. *Label* can be global (starting with an underscore '_' or a letter) or can be local (starting with a colon ':'). Local *Labels* must be separated from other same-named local labels by at least one global label. *Label* is used by instructions like JMP, CALL and COGINIT to designate the target destination. See Global and Local Labels on page 242 for more information.
- ***Condition*** — an optional execution condition (IF_C, IF_Z, etc.) that causes *Instruction* to be executed or not. See IF_x (Conditions) on page 295 for more information.
- IMPROVED ***Instruction*** and ***Operands*** — a Propeller Assembly instruction (MOV, ADD, COGINIT, etc.) and its zero, one, or two operands as required by the *Instruction*.
- ***Effects*** — an optional list of one to three execution effects (WZ, WC, WR, and NR) to apply to the instruction, if executed. They cause the *Instruction* to modify the Z flag, C

flag, and to write, or not write, the instruction's result value to the destination register, respectively. See Effects on page 291 for more information.

Since every instruction can include these three optional fields (*Label*, *Condition*, and *Effects*), for simplicity those common fields are intentionally left out of the instruction's syntax description.

So, when you read a syntax description such as this:

WAITCNT ***Target***, ⟨#⟩ ***Delta***

...remember that the true syntax is this:

⟨***Label***⟩ ⟨***Condition***⟩ **WAITCNT** ***Target***, ⟨#⟩ ***Delta*** ⟨***Effects***⟩

This rule applies only to Propeller Assembly instructions; it does not apply to Propeller Assembly directives.

IMPROVED

Syntax declarations always give descriptive names to the instruction's operands, such as **WAITCNT**'s *Target* and *Delta* operands in the example above. The detailed descriptions refer to operands by these names, however, the opcode tables and truth tables always use the generic names (D, DEST, Destination, and S, SRC, Source) to refer to the instruction's bits that store the respective values.

NEW

Opcodes and Opcode Tables

Most syntax definitions include an opcode table similar to the one below. This table lists the instruction's 32-bit opcode, outputs and number of clock cycles.

The opcode table's first column contains the Propeller Assembly Instruction opcode, consisting of the following fields:

- **INSTR** (bits 31:26) - Indicates the instruction being executed.
- **ZCRI** (bits 25:22) - Indicates instruction's effect status and **SRC** field meaning.
- **CON** (bits 21:18) - Indicates the condition in which to execute the instruction.
- **DEST** (bits 17:9) - Contains the destination register address.
- **SRC** (bits 8:0) - Contains the source register address or 9-bit literal value.

The bits of the **ZCRI** field each contain a 1 or 0 to indicate whether or not the 'Z' flag, 'C' flag, and 'R'esult should be written, and whether or not the **SRC** field contains an 'I'mmediate value (rather than a register address). The Z and C bits of the **ZCRI** field are clear (0) by default and

are set (1) if the instruction was specified with a WZ and/or WC effect. See Effects on page 291. The R bit's default state depends on the type of instruction, but is also affected if the instruction was specified with the WR or NR effect. The I field's default state depends on the type of instruction and is affected by the inclusion, or lack of, the literal indicator (#) in the instruction's source field.

The bits of the CON field usually default to all ones (1111) but are affected if the instruction was specified with a condition. See IF_x (Conditions) on page 295.

The last four columns of the opcode table indicate the meaning of the instruction's output Z and C flags, the default behavior for writing or not writing the result value, and the number of clocks the instruction requires for execution.

CLKSET Opcode Table:

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
000011	0001	1111	ddddddddd	------000	---	---	Not Written	7..22

NEW

Concise Truth Tables

After the opcode table, there is a concise truth table. The concise truth table demonstrates sample inputs and resulting outputs for the corresponding instruction. Rather than showing every possible input/output case, the instruction's concise truth table focuses on exploiting numerical or logical boundaries that result in flag activity and notable destination output. This information can aid in learning or verifying the instruction's intrinsic function and behavior.

Generally, the concise truth tables should be read carefully from the top row towards the bottom row. When multiple boundary cases are possible, the related rows are grouped together for emphasis and separated from other groups by a thick horizontal line.

The following conventions are used:

$FFFF_FFFE; -2	Numbers are values in hexadecimal (left of ';') and decimal (right of ';').
%0_00000011; 3	Numbers are values in binary (left of ';') and decimal (right of ';').
0 –or– 1	Individual zero (0) or one (1) means binary 0 or 1.
wr, wz, wc	Assembly effects indicate execution state; write-result, write-z, write-c.
x	Lower case "x" indicates items where every possible value applies.
---	Hyphens indicate items that are not applicable or not important.

A good example is the truth table for the ADDS instruction:

ADDS Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$FFFF_FFFF; -1	$0000_0001; 1	-	-	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0002; 2	-	-	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$FFFF_FFFF; -1	-	-	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFE; -2	-	-	wz wc	$FFFF_FFFF; -1	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0001; 1	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0002; 2	-	-	wz wc	$8000_0000; -2,147,483,648	0	1
$8000_0001; -2,147,483,647	$FFFF_FFFF; -1	-	-	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$FFFF_FFFE; -2	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

In the ADDS truth table there are eight data rows grouped into four pairs. Each group exploits a different boundary condition. The first five columns of each row indicate the inputs to the instruction and the last three columns show the resulting outputs.

- The first pair of data rows demonstrates a simple signed addition (-1 + 1) that results in zero (z flag set) and also an example (-1 + 2) that results in non-zero (Z flag clear).
- The second pair of rows shows the same concept but with inverted signs on the values; (1 + -1) and (1 + -2).
- The third pair of rows demonstrates an addition near the highest signed integer boundary (2,147,482,646 + 1) followed by another that crosses that boundary (2,147,482,646 + 2) resulting in a signed overflow (C flag set).
- The fourth pair of rows shows the same concept but approaching and crossing the signed integer boundary from the negative side, also resulting in a signed overflow (C flag set).

Note that an instruction's destination field actually contains the address to a register that holds the desired operand value, and the source field is often encoded similarly, but the truth tables always simplify this detail by only showing the desired operand value itself for each source and destination.

Propeller Assembly Instruction Master Table

A master table for all Propeller Assembly instructions is provided on the next two pages. In this table, D and S refer to the instructions' destination and source fields, also known as d-field and s-field, respectively. Please be sure to read the notes on the page that follows the table.

Instruction		-INSTR- ZCRI -CON- -DEST- -SRC-	Z Result	C Result	Result	Clocks
ABS	D, S	101010 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4
ABSNEG	D, S	101011 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4
IMPROVED ADD	D, S	100000 001i 1111 ddddddddd sssssssss	D + S = 0	Unsigned Carry	Written	4
IMPROVED ADDABS	D, S	100010 001i 1111 ddddddddd sssssssss	D + |S| = 0	Unsigned Carry [3]	Written	4
IMPROVED ADDS	D, S	110100 001i 1111 ddddddddd sssssssss	D + S = 0	Signed Overflow	Written	4
IMPROVED ADDSX	D, S	110110 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Signed Overflow	Written	4
IMPROVED ADDX	D, S	110010 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Unsigned Carry	Written	4
AND	D, S	011000 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4
ANDN	D, S	011001 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4
CALL	#S	010111 0011 1111 ????????? sssssssss	Result = 0	---	Written	4
CLKSET	D	000011 0001 1111 ddddddddd ------000	---	---	Not Written	7..22 [1]
IMPROVED CMP	D, S	100001 000i 1111 ddddddddd sssssssss	D = S	Unsigned (D < S)	Not Written	4
IMPROVED CMPS	D, S	110000 000i 1111 ddddddddd sssssssss	D = S	Signed (D < S)	Not Written	4
IMPROVED CMPSUB	D, S	111000 001i 1111 ddddddddd sssssssss	D = S	Unsigned (D => S)	Written	4
IMPROVED CMPSX	D, S	110001 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	Signed (D < S+C)	Not Written	4
IMPROVED CMPX	D, S	110011 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	Unsigned (D < S+C)	Not Written	4
IMPROVED COGID	D	000011 0011 1111 ddddddddd ------001	ID = 0	0	Written	7..22 [1]
IMPROVED COGINIT	D	000011 0001 1111 ddddddddd ------010	ID = 0	No Cog Free	Not Written	7..22 [1]
IMPROVED COGSTOP	D	000011 0001 1111 ddddddddd ------011	Stopped ID = 0	No Cog Free	Not Written	7..22 [1]
DJNZ	D, S	111001 001i 1111 ddddddddd sssssssss	Result = 0	Unsigned Borrow	Written	4 or 8 [2]
HUBOP	D, S	000011 000i 1111 ddddddddd sssssssss	Result = 0	---	Not Written	7..22 [1]
JMP	S	010111 000i 1111 --------- sssssssss	Result = 0	---	Not Written	4
JMPRET	D, S	010111 001i 1111 ddddddddd sssssssss	Result = 0	---	Written	4
IMPROVED LOCKCLR	D	000011 0001 1111 ddddddddd ------111	ID = 0	Prior Lock State	Not Written	7..22 [1]
IMPROVED LOCKNEW	D	000011 0011 1111 ddddddddd ------100	ID = 0	No Lock Free	Written	7..22 [1]
IMPROVED LOCKRET	D	000011 0001 1111 ddddddddd ------101	ID = 0	No Lock Free	Not Written	7..22 [1]
IMPROVED LOCKSET	D	000011 0001 1111 ddddddddd ------110	ID = 0	Prior Lock State	Not Written	7..22 [1]
MAX	D, S	010011 001i 1111 ddddddddd sssssssss	S = 0	Unsigned (D < S)	Written	4
MAXS	D, S	010001 001i 1111 ddddddddd sssssssss	S = 0	Signed (D < S)	Written	4
MIN	D, S	010010 001i 1111 ddddddddd sssssssss	S = 0	Unsigned (D < S)	Written	4
MINS	D, S	010000 001i 1111 ddddddddd sssssssss	S = 0	Signed (D < S)	Written	4
MOV	D, S	101000 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4
MOVD	D, S	010101 001i 1111 ddddddddd sssssssss	Result = 0	---	Written	4
MOVI	D, S	010110 001i 1111 ddddddddd sssssssss	Result = 0	---	Written	4
MOVS	D, S	010100 001i 1111 ddddddddd sssssssss	Result = 0	---	Written	4
MUXC	D, S	011100 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4
MUXNC	D, S	011101 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4
MUXNZ	D, S	011111 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4
MUXZ	D, S	011110 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4
NEG	D, S	101001 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4

Instruction	-INSTR- ZCRI -CON- -DEST- -SRC-	Z Result	C Result	Result	Clocks
NEGC D, S	101100 001i 1111 dddddddddd ssssssssss	Result = 0	S[31]	Written	4
NEGNC D, S	101101 001i 1111 dddddddddd ssssssssss	Result = 0	S[31]	Written	4
NEGNZ D, S	101111 001i 1111 dddddddddd ssssssssss	Result = 0	S[31]	Written	4
NEGZ D, S	101110 001i 1111 dddddddddd ssssssssss	Result = 0	S[31]	Written	4
NOP	------ ---- 0000 ---------- ----------	---	---	---	4
OR D, S	011010 001i 1111 dddddddddd ssssssssss	Result = 0	Parity of Result	Written	4
RCL D, S	001101 001i 1111 dddddddddd ssssssssss	Result = 0	D[31]	Written	4
RCR D, S	001100 001i 1111 dddddddddd ssssssssss	Result = 0	D[0]	Written	4
RDBYTE D, S	000000 001i 1111 dddddddddd ssssssssss	Result = 0	---	Written	7..22 [1]
RDLONG D, S	000010 001i 1111 dddddddddd ssssssssss	Result = 0	---	Written	7..22 [1]
RDWORD D, S	000001 001i 1111 dddddddddd ssssssssss	Result = 0	---	Written	7..22 [1]
RET	010111 0001 1111 ---------- ----------	Result = 0	---	Not Written	4
REV D, S	001111 001i 1111 dddddddddd ssssssssss	Result = 0	D[0]	Written	4
ROL D, S	001001 001i 1111 dddddddddd ssssssssss	Result = 0	D[31]	Written	4
ROR D, S	001000 001i 1111 dddddddddd ssssssssss	Result = 0	D[0]	Written	4
SAR D, S	001110 001i 1111 dddddddddd ssssssssss	Result = 0	D[0]	Written	4
SHL D, S	001011 001i 1111 dddddddddd ssssssssss	Result = 0	D[31]	Written	4
SHR D, S	001010 001i 1111 dddddddddd ssssssssss	Result = 0	D[0]	Written	4
IMPROVED SUB D, S	100001 001i 1111 dddddddddd ssssssssss	D - S = 0	Unsigned Borrow	Written	4
IMPROVED SUBABS D, S	100011 001i 1111 dddddddddd ssssssssss	D - |S| = 0	Unsigned Borrow [4]	Written	4
IMPROVED SUBS D, S	110101 001i 1111 dddddddddd ssssssssss	D - S = 0	Signed Overflow	Written	4
IMPROVED SUBSX D, S	110111 001i 1111 dddddddddd ssssssssss	Z & (D-(S+C) = 0)	Signed Overflow	Written	4
IMPROVED SUBX D, S	110011 001i 1111 dddddddddd ssssssssss	Z & (D-(S+C) = 0)	Unsigned Borrow	Written	4
IMPROVED SUMC D, S	100100 001i 1111 dddddddddd ssssssssss	D ± S = 0	Signed Overflow	Written	4
IMPROVED SUMNC D, S	100101 001i 1111 dddddddddd ssssssssss	D ± S = 0	Signed Overflow	Written	4
IMPROVED SUMNZ D, S	100111 001i 1111 dddddddddd ssssssssss	D ± S = 0	Signed Overflow	Written	4
IMPROVED SUMZ D, S	100110 001i 1111 dddddddddd ssssssssss	D ± S = 0	Signed Overflow	Written	4
IMPROVED TEST D, S	011000 000i 1111 dddddddddd ssssssssss	D = 0	Parity of Result	Not Written	4
NEW TESTN D, S	011001 000i 1111 dddddddddd ssssssssss	D = 0	Parity of Result	Not Written	4
IMPROVED TJNZ D, S	111010 000i 1111 dddddddddd ssssssssss	D = 0	0	Not Written	4 or 8 [2]
IMPROVED TJZ D, S	111011 000i 1111 dddddddddd ssssssssss	D = 0	0	Not Written	4 or 8 [2]
WAITCNT D, S	111110 001i 1111 dddddddddd ssssssssss	Result = 0	Unsigned Carry	Written	5+
WAITPEQ D, S	111100 000i 1111 dddddddddd ssssssssss	Result = 0	---	Not Written	5+
WAITPNE D, S	111101 000i 1111 dddddddddd ssssssssss	Result = 0	---	Not Written	5+
WAITVID D, S	111111 000i 1111 dddddddddd ssssssssss	Result = 0	---	Not Written	5+
WRBYTE D, S	000000 000i 1111 dddddddddd ssssssssss	---	---	Not Written	7..22 [1]
WRLONG D, S	000010 000i 1111 dddddddddd ssssssssss	---	---	Not Written	7..22 [1]
WRWORD D, S	000001 000i 1111 dddddddddd ssssssssss	---	---	Not Written	7..22 [1]
XOR D, S	011011 001i 1111 dddddddddd ssssssssss	Result = 0	Parity of Result	Written	4

Notes for Master Table

Note 1: Clock Cycles for Hub Instructions

Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. The Hub provides a hub access window to each cog every 16 clocks. Because each cog runs independently of the Hub, it must sync to the Hub when executing a hub instruction. The first hub instruction in a sequence will take from 0 to 15 clocks to sync up to the hub access window, and 7 clocks afterwards to execute; thus the 7 to 22 (15 + 7) clock cycles to execute. After the first hub instruction, there will be 9 (16 – 7) free clocks before a subsequent hub access window arrives for that cog; enough time to execute two 4-clock instructions without missing the next hub access window. To minimize clock waste, you can insert two normal instructions between any two otherwise-contiguous hub instructions without any increase in execution time. Beware that hub instructions can cause execution timing to appear indeterminate; particularly the first hub instruction in a sequence.

Note 2: Clock Cycles for Modify-Branch Instructions

Instructions that modify a value and possibly jump, based on the result, require a different amount of clock cycles depending on whether or not a jump is required. These instructions take 4 clock cycles if a jump is required and 8 clock cycles if no jump is required. Since loops utilizing these instructions typically need to be fast, they are optimized in this way for speed.

NEW **Note 3:** `ADDABS` C out: If S is negative, C = the inverse of unsigned *borrow* (for D-S).

NEW **Note 4**: `SUBABS` C out: If S is negative, C = the inverse of unsigned *carry* (for D+S).

ABS

Instruction: Get the absolute value of a number.

ABS *AValue*, ⟨#⟩ *SValue*

Result: Absolute *SValue* is stored in *AValue*.

- ***AValue*** (d-field) is the register in which to write the absolute of *SValue*.
- ***SValue*** (s-field) is a register or a 9-bit literal whose absolute value will be written to *AValue*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
101010 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$----_----; -	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	-	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

[1] The smallest negative number (-2,147,483,648) has no corresponding positive value in 32-bit two's-complement math.

Explanation

ABS takes the absolute value of *SValue* and writes the result into *AValue*.

If the WZ effect is specified, the Z flag is set (1) if *SValue* is zero. If the WC effect is specified, the C flag is set (1) if *SValue* is negative, or cleared (0) if *SValue* is positive. The result is written to *AValue* unless the NR effect is specified.

Literal *SValues* are zero-extended, so ABS is really best used with register *SValues*.

ABSNEG

Instruction: Get the negative of a number's absolute value.

ABSNEG *NValue*, ⟨#⟩ *SValue*

Result: Absolute negative of *SValue* is stored in *NValue*.

- ***NValue*** (d-field) is the register in which to write the negative of *SValue*'s absolute value.
- ***SValue*** (s-field) is a register or a 9-bit literal whose absolute negative value will be written to *NValue*.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
101011 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$----_----; -	$0000_0001; 1	-	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	-	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	-	wz wc	$8000_0000; -2,147,483,648	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	-	wz wc	$8000_0001; -2,147,483,647	0	1

Explanation

ABSNEG negates the absolute value of *SValue* and writes the result into *NValue*.

If the WZ effect is specified, the Z flag is set (1) if *SValue* is zero. If the WC effect is specified, the C flag is set (1) if *SValue* is negative, or cleared (0) if *SValue* is positive. The result is written to *NValue* unless the NR effect is specified.

Literal *SValues* are zero-extended, so ABS is really best used with register *SValues*.

ADD

Instruction: Add two unsigned values.

ADD *Value1*, ⟨#⟩ *Value2*

Result: Sum of unsigned *Value1* and unsigned *Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to add to *Value2* and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is added into *Value1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
100000 001i 1111 ddddddddd sssssssss	D + S = 0	Unsigned Carry	Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source[1]	Z	C	Effects	Destination	Z	C
$FFFF_FFFE; 4,294,967,294	$0000_0001; 1	–	–	wz wc	$FFFF_FFFF; 4,294,967,295	0	0
$FFFF_FFFE; 4,294,967,294	$0000_0002; 2	–	–	wz wc	$0000_0000; 0	1	1
$FFFF_FFFE; 4,294,967,294	$0000_0003; 3	–	–	wz wc	$0000_0001; 1	0	1

[1] **Both Source and Destination are treated as unsigned values.**

Explanation

ADD sums the two unsigned values of *Value1* and *Value2* together and stores the result into the *Value1* register.

If the WZ effect is specified, the Z flag is set (1) if *Value1* + *Value2* equals zero. If the WC effect is specified, the C flag is set (1) if the summation resulted in an unsigned carry (32-bit overflow). The result is written to *Value1* unless the NR effect is specified.

NEW

To add unsigned, multi-long values, use ADD followed by ADDX. See ADDX on page 264 for more information.

ADDABS

Instruction: Add an absolute value to another value.

ADDABS *Value*, ⟨#⟩ *SValue*

Result: Sum of *Value* and absolute of signed *SValue* is stored in *Value*.

- ***Value*** (d-field) is the register containing the value to add to the absolute of *SValue* and is the destination in which to write the result.
- ***SValue*** (s-field) is a register or a 9-bit literal whose absolute value is added into *Value*. Literal *SValues* are zero-extended (always positive values) so ADDABS is best used with register *SValues*.

Opcode Table:

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
100010	001i	1111	ddddddddd	sssssssss	D + |S| = 0	Unsigned Carry[1]	Written	4

[1] If S is negative, C Result is the inverse of unsigned *borrow* (for D - S).

Concise Truth Table:

In					Out		
Destination[1]	Source	Z	C	Effects	Destination	Z	C
$FFFF_FFFD; 4,294,967,293	$0000_0004; 4	–	–	wz wc	$0000_0001; 1	0	1
$FFFF_FFFD; 4,294,967,293	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	1
$FFFF_FFFD; 4,294,967,293	$0000_0002; 2	–	–	wz wc	$FFFF_FFFF; 4,294,967,295	0	0
$FFFF_FFFD; 4,294,967,293	$FFFF_FFFF; -1	–	–	wz wc	$FFFF_FFFE; 4,294,967,294	0	1
$FFFF_FFFD; 4,294,967,293	$FFFF_FFFE; -2	–	–	wz wc	$FFFF_FFFF; 4,294,967,295	0	1
$FFFF_FFFD; 4,294,967,293	$FFFF_FFFD; -3	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFD; 4,294,967,293	$FFFF_FFFC; -4	–	–	wz wc	$0000_0001; 1	0	0

[1] Destination is treated as an unsigned value.

Explanation

ADDABS sums *Value* and the absolute of *SValue* together and stores the result into the *Value* register.

If the WZ effect is specified, the Z flag is set (1) if *Value* + |*SValue*| equals zero. If the WC effect is specified, the C flag is set (1) if the summation resulted in an unsigned carry (32-bit overflow). The result is written to *Value* unless the NR effect is specified.

ADDS

Instruction: Add two signed values.

ADDS *SValue1*, ⟨#⟩ *SValue2*

Result: Sum of signed *SValue1* and signed *SValue2* is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to add to *SValue2* and is the destination in which to write the result.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is added into *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
110100 001i 1111 ddddddddd sssssssss	D + S = 0	Signed Overflow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$FFFF_FFFF; -1	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$FFFF_FFFF; -1	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFE; -2	–	–	wz wc	$FFFF_FFFF; -1	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0001; 1	–	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0002; 2	–	–	wz wc	$8000_0000; -2,147,483,648	0	1
$8000_0001; -2,147,483,647	$FFFF_FFFF; -1	–	–	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$FFFF_FFFE; -2	–	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

Explanation

ADDS sums the two signed values of *SValue1* and *SValue2* together and stores the result into the *SValue1* register.

If the WZ effect is specified, the Z flag is set (1) if *SValue1* + *SValue2* equals zero. If the WC effect is specified, the C flag is set (1) if the summation resulted in a signed overflow. The result is written to *SValue1* unless the NR effect is specified.

NEW

To add signed, multi-long values, use ADD, possibly ADDX, and finally ADDSX. See ADDSX on page 262 for more information.

ADDSX

Instruction: Add two signed values plus C.

ADDSX *SValue1*, {#} *SValue2*

Result: Sum of signed *SValue1* and signed *SValue2* plus C flag is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to add to *SValue2* plus C, and is the destination in which to write the result.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value plus C is added into *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
110110 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Signed Overflow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$FFFF_FFFE; -2	$0000_0001; 1	x	0	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFE; -2	$0000_0001; 1	0	1	wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; -2	$0000_0001; 1	1	1	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFE; -2	x	0	wz wc	$FFFF_FFFF; -1	0	0
$0000_0001; 1	$FFFF_FFFE; -2	0	1	wz wc	$0000_0000; 0	0	0
$0000_0001; 1	$FFFF_FFFE; -2	1	1	wz wc	$0000_0000; 0	1	0
$7FFF_FFFE; 2,147,483,646	$0000_0001; 1	x	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0001; 1	x	1	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFE; 2,147,483,646	$0000_0002; 2	x	0	wz wc	$8000_0000; -2,147,483,648	0	1
$8000_0001; -2,147,483,647	$FFFF_FFFF; -1	x	0	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$FFFF_FFFE; -2	x	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0001; -2,147,483,647	$FFFF_FFFE; -2	x	1	wz wc	$8000_0000; -2,147,483,648	0	0

NEW

Explanation

ADDSX (Add Signed, Extended) sums the two signed values of *SValue1* and *SValue2* plus C, and stores the result into the *SValue1* register. The ADDSX instruction is used to perform signed multi-long addition; 64-bit additions, for example.

In a signed multi-long operation, the first instruction is unsigned (ex: ADD), any middle instructions are unsigned, extended (ex: ADDX), and the last instruction is signed, extended (ex: ADDSX). Make sure to use the WC, and optionally WZ, effect on the leading ADD and ADDX instructions.

For example, a signed double-long (64-bit) addition may look like this:

```
add     XLow,  YLow   wc wz   'Add low longs together; save C and Z
addsx   XHigh, YHigh          'Add high longs together
```

After executing the above, the double-long (64-bit) result is in the long registers *XHigh:XLow*. If *XHigh:XLow* started out as $0000_0001:0000_0000 (4,294,967,296) and *YHigh:YLow* was $FFFF_FFFF:FFFF_FFFF (-1) the result in *XHigh:XLow* would be $0000_0000:FFFF_FFFF (4,294,967,295). This is demonstrated below.

```
                          Hexadecimal              Decimal
                        (high)     (low)
   (XHigh:XLow)      $0000_0001:0000_0000       4,294,967,296
 + (YHigh:YLow)    + $FFFF_FFFF:FFFF_FFFF     +            -1
                   ----------------------     ---------------
                   = $0000_0000:FFFF_FFFF     = 4,294,967,295
```

A signed triple-long (96-bit) addition would look similar but with an ADDX instruction inserted between the ADD and ADDSX instructions:

```
add     XLow,  YLow   wc wz   'Add low longs; save C and Z
addx    XMid,  YMid   wc wz   'Add middle longs; save C and Z
addsx   XHigh, YHigh          'Add high longs
```

Of course, it may be necessary to specify the WC and WZ effects on the final instruction, ADDSX, in order to watch for a result of zero or signed overflow condition. Note that during this multi-step operation the Z flag always indicates if the result is turning out to be zero, but the C flag indicates unsigned carries until the final instruction, ADDSX, in which it indicates signed overflow.

For ADDSX, if the WZ effect is specified, the Z flag is set (1) if Z was previously set and *SValue1* + *SValue2* + C equals zero (use WC and WZ on preceding ADD and ADDX instructions). If the WC effect is specified, the C flag is set (1) if the summation resulted in a signed overflow. The result is written to *SValue1* unless the NR effect is specified.

ADDX

Instruction: Add two unsigned values plus C.

ADDX *Value1*, {#} *Value2*

Result: Sum of unsigned *Value1* and unsigned *Value2* plus C flag is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to add to *Value2* plus C, and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value plus C is added into *Value1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
110010 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Unsigned Carry	Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source[1]	Z	C	Effects	Destination	Z	C
$FFFF_FFFE; 4,294,967,294	$0000_0001; 1	x	0	wz wc	$FFFF_FFFF; 4,294,967,295	0	0
$FFFF_FFFE; 4,294,967,294	$0000_0001; 1	0	1	wz wc	$0000_0000; 0	0	1
$FFFF_FFFE; 4,294,967,294	$0000_0001; 1	1	1	wz wc	$0000_0000; 0	1	1

[1] **Both Source and Destination are treated as unsigned values.**

NEW

Explanation

ADDX (Add Extended) sums the two unsigned values of *Value1* and *Value2* plus C, and stores the result into the *Value1* register. The ADDX instruction is used to perform multi long addition; 64-bit additions, for example.

In a multi-long operation, the first instruction is unsigned (ex: ADD), any middle instructions are unsigned, extended (ex: ADDX), and the last instruction is unsigned, extended (ADDX) or signed, extended (ADDSX) depending on the nature of the original multi-long values. We'll discuss unsigned multi-long values here; see ADDSX on page 262 for examples with signed, multi-long values. Make sure to use the WC, and optionally WZ, effect on the leading ADD and ADDX instructions.

For example, an unsigned double-long (64-bit) addition may look like this:

```
add     XLow,  YLow   wc wz    'Add low longs together; save C and Z
addx    XHigh, YHigh           'Add high longs together
```

After executing the above, the double-long (64-bit) result is in the long registers *XHigh:XLow*. If *XHigh:XLow* started out as $0000_0000:FFFF_FFFF (4,294,967,295) and *YHigh:YLow* was $0000_0000:0000_0001 (1) the result in *XHigh:XLow* would be $0000_0001:0000_0000 (4,294,967,296). This is demonstrated below.

```
                          Hexadecimal                 Decimal
                        (high)     (low)
   (XHigh:XLow)       $0000_0000:FFFF_FFFF        4,294,967,295
 + (YHigh:YLow)     + $0000_0000:0000_0001      +             1
                    ----------------------      ---------------
                    = $0000_0001:0000_0000      = 4,294,967,296
```

Of course, it may be necessary to specify the **WC** and **WZ** effects on the final instruction, **ADDX**, in order to watch for a result of zero or an unsigned overflow condition.

For **ADDX**, if the **WZ** effect is specified, the Z flag is set (1) if Z was previously set and *Value1* + *Value2* + C equals zero (use **WC** and **WZ** on preceding **ADD** and **ADDX** instructions). If the **WC** effect is specified, the C flag is set (1) if the summation resulted in an unsigned carry (32-bit overflow). The result is written to *Value1* unless the **NR** effect is specified.

AND

Instruction: Bitwise AND two values.

AND *Value1*, {#} *Value2*

Result: *Value1* AND *Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to bitwise AND with *Value2* and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is bitwise ANDed with *Value1*.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
011000 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_000A; 10	$0000_0005; 5	–	–	wz wc	$0000_0000; 0	1	0
$0000_000A; 10	$0000_0007; 7	–	–	wz wc	$0000_0002; 2	0	1
$0000_000A; 10	$0000_000F; 15	–	–	wz wc	$0000_000A; 10	0	0

Explanation

AND (bitwise AND) performs a bitwise AND of the value in *Value2* into that of *Value1*.

If the WZ effect is specified, the Z flag is set (1) if *Value1* AND *Value2* equals zero. If the WC effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits. The result is written to *Value1* unless the NR effect is specified.

ANDN

Instruction: Bitwise AND a value with the NOT of another.

ANDN *Value1*, {#} *Value2*

Result: *Value1* AND !*Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to bitwise AND with !*Value2* and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is inverted (bitwise NOT) and bitwise ANDed with *Value1*.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
011001 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$F731_125A; -147,778,982	$FFFF_FFFA; -6	-	-	wz wc	$0000_0000; 0	1	0
$F731_125A; -147,778,982	$FFFF_FFF8; -8	-	-	wz wc	$0000_0002; 2	0	1
$F731_125A; -147,778,982	$FFFF_FFF0; -16	-	-	wz wc	$0000_000A; 10	0	0

Explanation

ANDN (bitwise AND NOT) performs a bitwise AND of the inverted value (bitwise NOT) of *Value2* into that of *Value1*.

If the WZ effect is specified, the Z flag is set (1) if *Value1* AND !*Value2* equals zero. If the WC effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits. The result is written to *Value1* unless the NR effect is specified.

CALL

Instruction: Jump to address with intention to return to next instruction.

CALL #*Symbol*

Result: PC + 1 is written to the s-field of the register indicated by the d-field.

IMPROVED

- *Symbol* (s-field) is a 9-bit literal whose value is the address to jump to. This field must contain a DAT symbol specified as a literal (#symbol) and the corresponding code should eventually execute a RET instruction labeled with the same symbol plus a suffix of "_ret" (*Symbol*_ret RET).

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
010111 0011 1111 ????????? sssssssss	Result = 0	---	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C[2]
$----_----; -	$----_----; -	-	-	wz wc	31:9 unchanged, 8:0 = PC+1	0	1

[1] The Destination register's s-field (lowest 9 bits) are overwritten with the return address (PC+1) at run-time.
[2] The C flag is set (1) unless PC+1 equals 0; very unlikely since it would require the CALL to be executed from the top of cog RAM ($1FF; special purpose register VSCL).

NEW

Explanation

CALL records the address of the next instruction (PC + 1) then jumps to *Symbol*. The routine at *Symbol* should eventually execute a RET instruction to return to the recorded address (PC+1; the instruction following the CALL). For the CALL to compile and run properly, the *Symbol* routine's RET instruction must be labeled in the form *Symbol* with "_ret" appended to it. The reason for this is explained below.

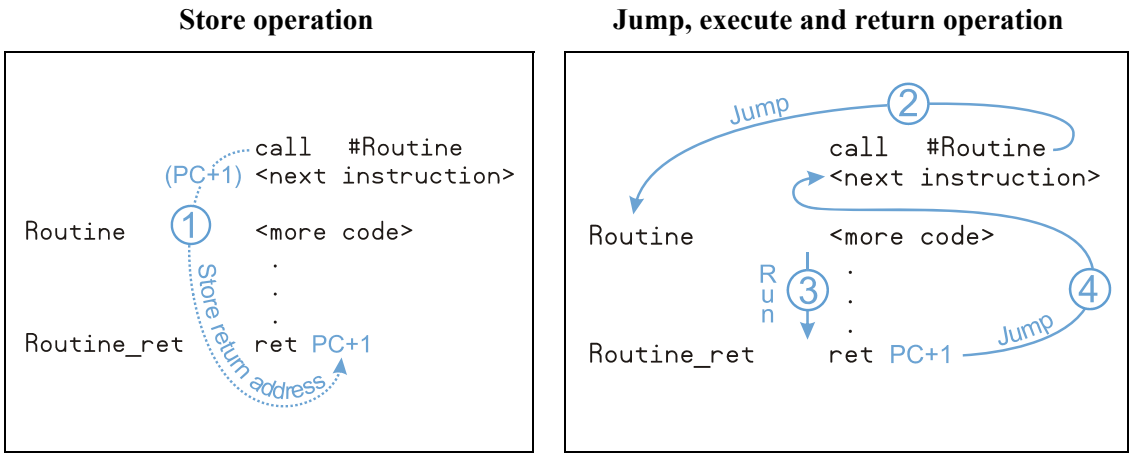
The Propeller hardware does not use a call stack, so the return address must be stored in a different manner. At compile time the assembler locates the destination routine as well as its RET instruction (labeled *Symbol* and *Symbol*_ret, respectively) and encodes those addresses into the CALL instruction's s-field and d-field. This provides the CALL instruction with the knowledge of both where it's going to jump to and exactly where it will return from.

At run time the first thing the CALL instruction does is store the return address (PC+1) into the location where it will return from; the "*Symbol*_ret RET" instruction location. The RET

instruction is really just a JMP instruction without a hard-coded destination address, and this run-time action provides it with the "return" address to jump back to. After storing the return address, CALL jumps to the destination address; *Symbol*.

The diagram below uses a short program example to demonstrate the CALL instruction's run-time behavior; the store operation (left) and the jump-execute-return operation (right).

Figure 3-1: Run-time CALL Procedure



In this example, the following occurs when the CALL instruction is reached at run time:

① The cog stores the return address (PC+1; that of <next instruction>) into the source (s-field) of the register at Routine_ret (see left image).

② The cog jumps to Routine (see right image).

③ Routine's instructions are executed, eventually leading to the Routine_ret line.

④ Since the Routine_ret location contains a RET instruction with an updated source (s-field), which is the return address written by step 1, it returns, or jumps, back to the <next instruction> line.

This nature of the CALL instruction dictates the following:

- The referenced routine must have only one RET instruction associated with it. If a routine needs more than one exit point, make one of those exit points the RET instruction and make all other exit points branch (i.e., JMP) to that RET instruction.
- The referenced routine can not be recursive. Making a nested call to the routine will overwrite the return address of the previous call.

CALL is really a subset of the JMPRET instruction; in fact, it is the same opcode as JMPRET but with the i-field set (since CALL uses an immediate value only) and the d-field set by the assembler to the address of the label named *Symbol*_ret.

The return address (PC + 1) is written to the source (s-field) of the *Symbol*_ret register unless the NR effect is specified. Of course, specifying NR is not recommended for the CALL instruction since that turns it into a JMP, or RET, instruction.

CLKSET

Instruction: Set the clock mode at run time.

CLKSET *Mode*

- ***Mode*** (d-field) is the register containing the 8-bit pattern to write to the CLK register.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
000011 0001 1111 ddddddddd ------000	---	---	Not Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source[1]	Z	C	Effects	Destination[2]	Z	C
$0000_006F; 111	%0_00000000; 0	-	-	wr wz wc	$0000_0007; 7	0	0

[1] The Source is automatically set to immediate value 0 by the assembler to indicate that this is the CLKSET hub instruction.
[2] Destination is not written unless the WR effect is given.

Explanation

CLKSET changes the System Clock mode during run time. The CLKSET instruction behaves similar to the Spin command of the same name (see CLKSET on page 71) except that it only sets the clock mode, not the frequency.

After issuing a CLKSET instruction, it is important to update the System Clock Frequency value by writing to its location in Main RAM (long 0): WRLONG freqaddr, #0. If the System Clock Frequency value is not updated, other objects will misbehave due to invalid clock frequency data.

CLKSET is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

CMP

Instruction: Compare two unsigned values.

CMP *Value1*, ⟨#⟩ *Value2*

Result: Optionally, equality and greater/lesser status is written to the Z and C flags.

- ***Value1*** (d-field) is the register containing the value to compare with that of *Value2*.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is compared with *Value1*.

Opcode Table:

IMPROVED

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
100001	000i	1111	ddddddddd	sssssssss	D = S	Unsigned (D < S)	Not Written	4

NEW

Concise Truth Table:

In					Out		
Destination [1]	Source [1]	Z	C	Effects	Destination [2]	Z	C
$0000_0003; 3	$0000_0002; 2	–	–	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0003; 3	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0004; 4	–	–	wr wz wc	$FFFF_FFFF; -1 [3]	0	1
$8000_0000; 2,147,483,648	$7FFF_FFFF; 2,147,483,647	–	–	wr wz wc	$0000_0001; 1	0	0 [4]
$7FFF_FFFF; 2,147,483,647	$8000_0000; 2,147,483,648	–	–	wr wz wc	$FFFF_FFFF; -1 [3]	0	1 [4]
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFF; 4,294,967,295	–	–	wr wz wc	$FFFF_FFFF; -1 [3]	0	1
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFE; 4,294,967,294	–	–	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFD; 4,294,967,293	–	–	wr wz wc	$0000_0001; 1	0	0

[1] Both Source and Destination are treated as unsigned values.
[2] Destination is not written unless the WR effect is given.
[3] Destination Out (written Destination) may be thought of as either unsigned or signed; it is shown here as signed for demonstration purposes only.
[4] The C flag result of CMP (Compare Unsigned) may differ from CMPS (Compare Signed) where the "interpreted sign" of Source and Destination are opposite. The first example in the second group, above, shows that CMP clears C because unsigned $8000_0000 (2,147,483,648) is not less than unsigned $7FFF_FFFF (2,147,483,647). CMPS, however, would have set C because signed $8000_0000 (-2,147,483,648) is less than signed $7FFF_FFFF (2,147,483,647). The second example is the complementary case where the Source and Destination values are switched.

Explanation

CMP (Compare Unsigned) compares the unsigned values of *Value1* and *Value2*. The Z and C flags, if written, indicate the relative equal, and greater or lesser relationship between the two.

If the WZ effect is specified, the Z flag is set (1) if *Value1* equals *Value2*. If the WC effect is specified, the C flag is set (1) if *Value1* is less than *Value2*.

NEW

The result is not written unless the WR effect is specified. If WR is specified, CMP becomes exactly like a SUB instruction.

NEW

To compare unsigned, multi-long values, use CMP followed by CMPX. See CMPX on page 280 for more information..

CMPS

Instruction: Compare two signed values.

CMPS *SValue1*, ⟨#⟩ *SValue2*

Result: Optionally, equality and greater/lesser status is written to the Z and C flags.

- ***SValue1*** (d-field) is the register containing the value to compare with that of *SValue2*.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is compared with *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
110000 000i 1111 ddddddddd sssssssss	D = S	Signed (D < S)	Not Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C
$0000_0003; 3	$0000_0002; 2	–	–	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0003; 3	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0004; 4	–	–	wr wz wc	$FFFF_FFFF; -1	0	1
$8000_0000; -2,147,483,648	$7FFF_FFFF; 2,147,483,647	–	–	wr wz wc	$0000_0001; 1	0	1[2]
$7FFF_FFFF; 2,147,483,647	$8000_0000; -2,147,483,648	–	–	wr wz wc	$FFFF_FFFF; -1	0	0[2]
$8000_0000; -2,147,483,648	$0000_0001; 1	–	–	wr wz wc	$7FFF_FFFF; 2,147,483,647[3]	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	–	–	wr wz wc	$8000_0000; -2,147,483,648[3]	0	0
$FFFF_FFFE; -2	$FFFF_FFFF; -1	–	–	wr wz wc	$FFFF_FFFF; -1	0	1
$FFFF_FFFE; -2	$FFFF_FFFE; -2	–	–	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; -2	$FFFF_FFFD; -3	–	–	wr wz wc	$0000_0001; 1	0	0

[1] Destination is not written unless the WR effect is given.

[2] The C flag result of CMPS (Compare Signed) may differ from CMP (Compare Unsigned) where the "interpreted sign" of Source and Destination are opposite. The first example in the second group, above, shows that CMPS sets C because signed $8000_0000 (-2,147,483,648) is less than signed $7FFF_FFFF (2,147,483,647). CMP, however, would have cleared C because unsigned $8000_0000 (2,147,483,648) is not less than unsigned $7FFF_FFFF (2,147,483,647). The second example is the complementary case where the Source and Destination values are switched.

[3] The examples of the third group, above, demonstrate cases where the comparison is properly reflected in the flags but the Destination Out has crossed the signed border (signed overflow error) in either the negative or positive direction. This signed overflow condition can not be reflected in the flags. If this condition is important to an application, perform a CMPS without a WR effect, note C's status (signed borrow), then perform a SUBS and note C's status (signed overflow).

Explanation

CMPS (Compare Signed) compares the signed values of *SValue1* and *SValue2*. The Z and C flags, if written, indicate the relative equal, and greater or lesser relationship between the two.

IMPROVED

If the WZ effect is specified, the Z flag is set (1) if *SValue1* equals *SValue2*. If the WC effect is specified, the C flag is set (1) if *SValue1* is less than *SValue2*.

NEW

To compare signed, multi-long values, instead of using CMPS, use CMP, possibly CMPX, and finally CMPSX. See CMPSX on page 277 for more information.

CMPSUB

Instruction: Compare two unsigned values and subtract the second if it is lesser or equal.

CMPSUB *Value1*, ⟨#⟩ *Value2*

Result: Optionally, *Value1* = *Value1* – *Value2*, and Z and C flags = comparison results.

- ***Value1*** (d-field) is the register containing the value to compare with that of *Value2* and is the destination in which to write the result if a subtraction is performed.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is compared with and possibly subtracted from *Value1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
111000 001i 1111 ddddddddd sssssssss	D = S	Unsigned (D => S)	Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source[1]	Z	C	Effects	Destination	Z	C
$0000_0003; 3	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	1
$0000_0003; 3	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	1
$0000_0003; 3	$0000_0004; 4	–	–	wz wc	$0000_0003; 3	0	0

[1] Both Source and Destination are treated as unsigned values.

IMPROVED

Explanation

CMPSUB compares the unsigned values of *Value1* and *Value2*, and if *Value2* is equal to or less than *Value1* then it is subtracted from *Value1*.

If the WZ effect is specified, the Z flag is set (1) if *Value1* equals *Value2*. If the WC effect is specified, the C flag is set (1) if a subtraction is possible (*Value1* is equal to or greater than *Value2*). The result, if any, is written to *Value1* unless the NR effect is specified.

CMPSX

Instruction: Compare two signed values plus C.

CMPSX *SValue1*, ⟨#⟩ *SValue2*

Result: Optionally, equality and greater/lesser status is written to the Z and C flags.

- ***SValue1*** (d-field) is the register containing the value to compare with that of *SValue2*.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is compared with *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
110001 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	Signed (D < S+C)	Not Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C
$0000_0003; 3	$0000_0002; 2	x	0	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0002; 2	0	1	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0002; 2	1	1	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	0	0	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0003; 3	1	0	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	x	1	wr wz wc	$FFFF_FFFF; -1	0	1
$0000_0003; 3	$0000_0004; 4	x	0	wr wz wc	$FFFF_FFFF; -1	0	1
$0000_0003; 3	$0000_0004; 4	x	1	wr wz wc	$FFFF_FFFE; -2	0	1
$8000_0000; -2,147,483,648	$7FFF_FFFF; 2,147,483,647	0	0	wr wz wc	$0000_0001; 1	0	1[2]
$7FFF_FFFF; 2,147,483,647	$8000_0000; -2,147,483,648	0	0	wr wz wc	$FFFF_FFFF; -1	0	0[2]
$8000_0000; -2,147,483,648	$0000_0001; 1	0	0	wr wz wc	$7FFF_FFFF; 2,147,483,647[3]	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	0	0	wr wz wc	$8000_0000; -2,147,483,648[3]	0	0
$FFFF_FFFE; -2	$FFFF_FFFF; -1	x	0	wr wz wc	$FFFF_FFFF; -1	0	1
$FFFF_FFFE; -2	$FFFF_FFFF; -1	x	1	wr wz wc	$FFFF_FFFE; -2	0	1
$FFFF_FFFE; -2	$FFFF_FFFE; -2	0	0	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; -2	$FFFF_FFFE; -2	1	0	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; -2	$FFFF_FFFE; -2	x	1	wr wz wc	$FFFF_FFFF; -1	0	1
$FFFF_FFFE; -2	$FFFF_FFFD; -3	x	0	wr wz wc	$0000_0001; 1	0	0
$FFFF_FFFE; -2	$FFFF_FFFD; -3	0	1	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; -2	$FFFF_FFFD; -3	1	1	wr wz wc	$0000_0000; 0	1	0

[1] Destination is not written unless the WR effect is given.

[2] The C flag result of CMPSX (Compare Signed, Extended) may differ from CMPX (Compare Unsigned, Extended) where the "interpreted sign" of Source and Destination are opposite. The first example in the second group, above, shows that CMPSX sets C because signed $8000_0000 (-2,147,483,648) is less than signed $7FFF_FFFF (2,147,483,647). CMPX, however, would have cleared C because unsigned $8000_0000 (2,147,483,648) is not less than unsigned $7FFF_FFFF (2,147,483,647). The second example is the complementary case where the Source and Destination values are switched. Note that examples with differing Z and C are not shown but have expected effects similar to the other examples.

[3] The examples of the third group, above, demonstrate cases where the comparison is properly reflected in the flags but the Destination Out has crossed the signed border (signed overflow error) in either the negative or positive direction. This signed overflow condition can not be reflected in the flags. If this condition is important to an application, it must detect it through other means.

NEW

Explanation

CMPSX (Compare Signed, Extended) compares the signed values of *SValue1* and *SValue2* plus C. The Z and C flags, if written, indicate the relative equal, and greater or lesser relationship between the two. The **CMPSX** instruction is used to perform signed multi-long comparison; 64-bit comparisons, for example.

In a signed multi-long operation, the first instruction is unsigned (ex: **CMP**), any middle instructions are unsigned, extended (ex: **CMPX**), and the last instruction is signed, extended (ex: **CMPSX**). Make sure to use the **WC**, and optionally **WZ**, effect on all the instructions in the comparison operation.

For example, a signed double-long (64-bit) comparison may look like this:

```
cmp     XLow,  YLow   wc wz   'Compare low longs; save C and Z
cmpsx   XHigh, YHigh  wc wz   'Compare high longs; save C and Z
```

After executing the above, the C and Z flags will indicate the relationship between the two double-long (64-bit) values. If *XHigh:XLow* started out as $FFFF_FFFF:FFFF_FFFF (-1) and *YHigh:YLow* was $0000_0000:0000_0001 (1) the resulting flags would be: Z = 0 and C = 1; (Value1 < Value2). This is demonstrated below. Note that the comparison is really just a subtraction with the result not written; the Z and C flag result is important, however.

```
                        Hexadecimal          Decimal        Flags
                      (high)     (low)
  (XHigh:XLow)     $FFFF_FFFF:FFFF_FFFF              -1      n/a
- (YHigh:YLow)   - $0000_0000:0000_0001   -             1      n/a
                  ---------------------   ---------------   --------
                  = $FFFF_FFFF:FFFF_FFFE   =            -2   Z=0, C=1
```

A signed triple-long (96-bit) comparison would look similar but with a **CMPX** instruction inserted between the **CMP** and **CMPSX** instructions:

```
cmp     XLow,  YLow   wc wz   'Compare low longs; save C and Z
cmpx    XMid,  YMid   wc wz   'Compare middle longs; save C and Z
cmpsx   XHigh, YHigh  wc wz   'Compare high longs; save C and Z
```

For **CMPSX**, if the **WZ** effect is specified, the Z flag is set (1) if Z was previously set and *SValue1* equals *SValue2* + C (use **WC** and **WZ** on preceding **CMP** and **CMPX** instructions). If the **WC** effect is specified, the C flag is set (1) if *SValue1* is less than *SValue2* (as multi-long values).

CMPX

Instruction: Compare two unsigned values plus C.

CMPX *Value1*, ⟨#⟩ *Value2*

Result: Optionally, equality and greater/lesser status is written to the Z and C flags.

- ***Value1*** (d-field) is the register containing the value to compare with that of *Value2*.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is compared with *Value1*.

Opcode Table:

IMPROVED

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
110011	000i	1111	ddddddddd	sssssssss	Z & (D = S+C)	Unsigned (D < S+C)	Not Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source[1]	Z	C	Effects	Destination[2]	Z	C
$0000_0003; 3	$0000_0002; 2	x	0	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0002; 2	0	1	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0002; 2	1	1	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	0	0	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0003; 3	1	0	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	x	1	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$0000_0003; 3	$0000_0004; 4	x	0	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$0000_0003; 3	$0000_0004; 4	x	1	wr wz wc	$FFFF_FFFE; -2[3]	0	1
$8000_0000; 2,147,483,648	$7FFF_FFFF; 2,147,483,647	0	0	wr wz wc	$0000_0001; 1	0	0[4]
$7FFF_FFFF; 2,147,483,647	$8000_0000; 2,147,483,648	0	0	wr wz wc	$FFFF_FFFF; -1[3]	0	1[4]
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFF; 4,294,967,295	x	0	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFF; 4,294,967,295	x	1	wr wz wc	$FFFF_FFFE; -2[3]	0	1
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFE; 4,294,967,294	0	0	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFE; 4,294,967,294	1	0	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFE; 4,294,967,294	x	1	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFD; 4,294,967,293	x	0	wr wz wc	$0000_0001; 1	0	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFD; 4,294,967,293	0	1	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFD; 4,294,967,293	1	1	wr wz wc	$0000_0000; 0	1	0

[1] Both Source and Destination are treated as unsigned values.
[2] Destination is not written unless the WR effect is given.
[3] Destination Out (written Destination) may be thought of as either unsigned or signed; it is shown here as signed for demonstration purposes only.

[4] The C flag result of CMPX (Compare Unsigned, Extended) may differ from CMPSX (Compare Signed, Extended) where the "interpreted sign" of Source and Destination are opposite. The first example in the second group, above, shows that CMPX clears C because unsigned $8000_0000 (2,147,483,648) is not less than unsigned $7FFF_FFFF (2,147,483,647). CMPSX, however, would have set C because signed $8000_0000 (2,147,483,648) is less than signed $7FFF_FFFF (2,147,483,647). The second example is the complementary case where the Source and Destination values are switched. Note that examples with differing Z and C are not shown but have expected effects similar to the other examples.

NEW

Explanation

CMPX (Compare Extended) compares the unsigned values of *Value1* and *Value2* plus C. The Z and C flags, if written, indicate the relative equal, and greater or lesser relationship between the two. The CMPX instruction is used to perform multi-long comparison; 64-bit comparisons, for example.

In a multi-long operation, the first instruction is unsigned (ex: CMP), any middle instructions are unsigned, extended (ex: CMPX), and the last instruction is unsigned, extended (CMPX) or signed, extended (CMPSX) depending on the nature of the original multi-long values. We'll discuss unsigned multi-long values here; see CMPSX on page 277 for examples with signed, multi-long values. Make sure to use the WC, and optionally WZ, effect on all the instructions in the comparison operation.

For example, an unsigned double-long (64-bit) comparison may look like this:

```
cmp     XLow,  YLow   wc wz   'Compare low longs; save C and Z
cmpx    XHigh, YHigh  wc wz   'Compare high longs
```

After executing the above, the C and Z flags will indicate the relationship between the two double-long (64-bit) values. If *XHigh:XLow* started out as $0000_0001:0000_0000 (4,294,967,296) and *YHigh:YLow* was $0000_0000:0000_0001 (1) the resulting flags would be: Z = 0 and C = 0; (Value1 > Value2). This is demonstrated below. Note that the comparison is really just a subtraction with the result not written; the Z and C flag result is important, however.

```
                      Hexadecimal                 Decimal          Flags
                    (high)     (low)
  (XHigh:XLow)    $0000_0001:0000_0000      4,294,967,296          n/a
- (YHigh:YLow)  - $0000_0000:0000_0001    -             1          n/a
                 ---------------------    ---------------    --------
                 = $0000_0000:FFFF_FFFF   = 4,294,967,295    Z=0, C=0
```

For CMPX, if the WZ effect is specified, the Z flag is set (1) if Z was previously set and *Value1* equals *Value2* + C (use WC and WZ on preceding CMP and CMPX instructions). If the WC effect is specified, the C flag is set (1) if *Value1* is less than *Value2* (as multi-long values).

NEW

CNT

Register: System Counter register.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, CNT ⟨*Effects*⟩

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. CNT is a read-only register and thus should only be used in the instruction's source operand.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, using the value of CNT in the instruction's source operand.

Explanation

The CNT register contains the current value in the global 32-bit System Counter. The System Counter serves as the central time reference for all cogs; it increments its 32-bit value once every System Clock cycle.

CNT is a read-only pseudo-register; when used as an instruction's source operand, it reads the current value of the System Counter. Do not use CNT as the destination operand; that only results in reading and modifying the shadow register whose address CNT occupies.

CNT is often used to determine an initial target value for a WAITCNT-based delay. The following code performs an operation in a loop every ¼ second. See Registers, page 338, and the Spin language CNT section, page 73, for more information.

```
DAT
                org 0                   'Reset assembly pointer
AsmCode         rdlong  Delay, #0       'Get clock frequency
                shr     Delay, #2       'Divide by 4
                mov     Time,  cnt      'Get current time
                add     Time,  Delay    'Adjust by 1/4 second
Loop            waitcnt Time,  Delay    'Wait for 1/4 second
                '<more code here>       'Perform operation
                jmp     #Loop           'loop back

Delay   res     1
Time    res     1
```

COGID

Instruction: Get current cog's ID.

COGID *Destination*

Result: The current cog's ID (0-7) is written to *Destination*.

- ***Destination*** (d-field) is the register to write the cog's ID into.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
000011 0011 1111 ddddddddd ------001	ID = 0	0	Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source[1]	Z	C	Effects	Destination[2]	Z	C
$----_----; -	%0_00000001; 1	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	%0_00000001; 1	-	-	wz wc	$1; 1 .. $7; 7	0	0

[1] Source is automatically set to immediate value 1 by the assembler to indicate that this is the COGID hub instruction.
[2] Destination Out (written Destination) will be 0 through 7, depending on which cog executed the instruction.

Explanation

COGID returns the ID of the cog that executed the command. The COGID instruction behaves similar to the Spin command of the same name; see COGID on page 75.

If the WZ effect is specified, the Z flag is set if the cog ID is zero. The result is written to *Destination* unless the NR effect is specified.

COGID is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

COGINIT

Instruction: Start or restart a cog, optionally by ID, to run Propeller Assembly or Spin code.

COGINIT *Destination*

Result: Optionally, the started/restarted cog's ID (0-7) is written to *Destination*.

- *Destination* (d-field) is the register containing startup information for the target cog and optionally becomes the destination of the started cog's ID if a new cog is started.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
000011 0001 1111 ddddddddd ------010	ID = 0	No Cog Free	Not Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source[2]	Z	C	Effects	Destination[3]	Z	C
$0000_0000; 0	%0_00000010; 2	–	–	wr wz wc	$0000_0000; 0[4]	1	0
$0000_0001; 1	%0_00000010; 2	–	–	wr wz wc	$0000_0001; 1[4]	0	0
$0000_0008; 8	%0_00000010; 2	–	–	wr wz wc	$0000_0000; 0[5]	1	0
$0000_0008; 8	%0_00000010; 2	–	–	wr wz wc	$1; 1 .. $7; 7[5]	0	0
$0000_0008; 8	%0_00000010; 2	–	–	wr wz wc	$0000_0007; 7[5]	0	1

[1] Destination In must be a PAR value (bits 31:18), an assembly code address (bits 17:4), and a new cog / cog ID indicator (bits 3:0).
[2] Source is automatically set to immediate value 2 by the assembler to indicate that this is the COGINIT hub instruction.
[3] Destination is not written unless the WR effect is given.
[4] When Destination In indicates to start a specific cog, Destination Out (written Destination) indicates that cog's ID; it is always started or restarted.
[5] When Destination In indicates to start a new (next available) cog, Destination Out (written Destination) indicates the ID of the cog that was started, or 7 (with C set) if no cog available.

Explanation

The COGINIT instruction behaves similar to two Spin commands, COGNEW and COGINIT, put together. Propeller Assembly's COGINIT instruction can be used to start a new cog or restart an active cog. The *Destination* register has four fields that determine which cog is started, where its program begins in main memory, and what its PAR register will contain. The table below describes these fields.

Table 3-1: Destination Register Fields			
31:18	**17:4**	**3**	**2:0**
14-bit Long address for PAR Register	14-bit Long address of code to load	New	Cog ID

The first field, bits 31:18, will be written to the started cog's PAR register bits 15:2. This is 14 bits total that are intended to be the upper bits of a 16-bit long address. Similar to the *Parameter* field of Spin's version of COGINIT, this first field of *Destination* is used to pass the 14-bit address of an agreed-upon memory location or structure to the started cog.

The second field, bits 17:4, holds the upper 14-bits of a 16-bit long address pointing to the desired assembly program to load into the cog. Cog registers $000 through $1EF will be loaded sequentially starting at this address, the special purpose registers will be cleared to zero (0), and the cog will start executing the code at register $000.

The third field, bit 3, should be set (1) if a new cog should be started, or cleared (0) if a specific cog should be started or restarted.

If the third field bit is set (1), the Hub will start the next available (lowest-numbered inactive) cog and return that cog's ID in *Destination* (if the WR effect is specified).

If the third field bit is clear (0), the Hub will start or restart the cog identified by *Destination*'s fourth field, bits 2:0.

If the WZ effect is specified, the Z flag will be set (1) if the cog ID returned is 0. If the WC effect is specified, the C flag will be set (1) if no cog was available. If the WR effect is specified, *Destination* is written with the ID of the cog that the Hub started, or would have started, if you let it pick one.

It is not practical to launch Spin code from a user's Propeller Assembly code; we recommend launching only assembly code with this instruction.

COGINIT is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

COGSTOP

Instruction: Start a cog by its ID.

COGSTOP *CogID*

- ***CogID*** (d-field) is the register containing the ID (0 – 7) of the cog to stop.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
000011 0001 1111 ddddddddd ------011	Stopped ID = 0	No Cog Free	Not Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source[1]	Z	C	Effects	Destination[2,3]	Z	C[4]
$0000_0000; 0	%0_00000011; 3	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0005; 5	%0_00000011; 3	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0008; 8[5]	%0_00000011; 3	–	–	wr wz wc	$0000_0000; 0	1	0

[1] Source is automatically set to immediate value 3 by the assembler to indicate that this is the COGSTOP hub instruction.
[2] Destination is not written unless the WR effect is given.
[3] Destination Out (written Destination) indicates the cog that was stopped.
[4] The C flag will be set (1) if all cogs were running prior to executing the COGSTOP instruction.
[5] Only the lowest 3 bits of Destination In are utilized, so a value of 8 is seen as cog 0.

Explanation

The COGSTOP instruction stops a cog whose ID is in the register *CogID,* placing that cog into a dormant state. In the dormant state, the cog ceases to receive System Clock pulses so that power consumption is greatly reduced.

COGSTOP is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

NEW

If the WZ effect is specified, the Z flag will be set (1) if the ID of the cog that was stopped is zero (0). If the WC effect is specified, the C flag will be set (1) if all cogs were running prior to executing this instruction. If the WR effect is specified, *Destination* is written with the ID of the cog that was stopped.

Conditions (IF_x)

Every Propeller Assembly instruction has an optional "condition" field that is used to dynamically determine whether or not it executes when it is reached at run time. The basic syntax for Propeller Assembly instructions is:

⟨***Label***⟩ ⟨***Condition***⟩ ***Instruction Operands*** ⟨***Effects***⟩

The optional *Condition* field can contain one of 32 conditions (see **IF_x** (Conditions), page 295) and defaults to **IF_ALWAYS** when no condition is specified. During compilation, the 4-bit **Value** representing the condition is used in place of the **–CON–** field's default bits in the instruction's opcode.

This feature, along with proper use of instructions' optional *Effects* field, makes Propeller Assembly very powerful. For example, the C and Z flags can be affected at will and later instructions can be conditionally executed based on those results.

When an instruction's condition evaluates to **FALSE**, the instruction dynamically becomes a **NOP**, elapsing 4 clock cycles but affecting no flags or registers. This makes the timing of multi-decision code very deterministic since the same path of execution (same execution time) can be used and yet can achieve one of many possible outcomes.

See **IF_x** (Conditions) on page 295 for more information.

NEW

CTRA, CTRB

Register: Counter A and Counter B control registers.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* CTRA, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, CTRA ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* CTRB, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, CTRB ⟨*Effects*⟩

Result: Optionally, the counter control register is updated.

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. CTRA or CTRB may be used in either the assembly instruction's *DestOperand* or *SrcOperand* fields.
- ***SrcOperand*** is a constant expression used by *Instruction* to operate on, and optionally write to, the CTRA or CTRB register in *DestOperand*.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, using the value of CTRA or CTRB in *SrcOperand*.

Explanation

CTRA and CTRB are two of six registers (CTRA, CTRB, FRQA, FRQB, PHSA, and PHSB) that affect the behavior of a cog's Counter Modules. Each cog has two identical counter modules (A and B) that can perform many repetitive tasks. The CTRA and CTRB registers contain the configuration settings of the Counter A and Counter B Modules, respectively.

CTRA and CTRB are read/write registers and can be used in either the *DestOperand* or *SrcOperand* fields of an assembly instruction. The following code sets Counter A to NCO mode on I/O pin 2. See Registers, page 338, and the Spin language CTRA, CTRB section, page 95, for more information.

```
                mov     ctra, CtrCfg

CtrCfg          long    %0_00100_000_0000000_000000_000_000010
```

NEW

DIRA, DIRB

Register: Direction registers for 32-bit ports A and B.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* DIRA, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, DIRA ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* DIRB, *SrcOperand* ⟨*Effects*⟩ (Reserved for future use)

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, DIRB ⟨*Effects*⟩ (Reserved for future use)

Result: Optionally, the direction register is updated.

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. DIRA or DIRB may be used in either the assembly instruction's *DestOperand* or *SrcOperand* fields.
- ***SrcOperand*** is a constant expression used by *Instruction* to operate on, and optionally write to, the DIRA or DIRB register in *DestOperand*.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, using the value of DIRA or DIRB in *SrcOperand*.

Explanation

DIRA and DIRB are one of six special purpose registers (DIRA, DIRB, INA, INB, OUTA and OUTB) that directly affect the I/O pins. The DIRA and DIRB register's bits indicate the direction states for each of the 32 I/O pins in Port A and Port B, respectively. DIRB is reserved for future use; the Propeller P8X32A does not include Port B I/O pins so only DIRA is discussed below.

DIRA is a read/write register and can be used in either the *DestOperand* or *SrcOperand* fields of an assembly instruction. A low (0) bit sets the corresponding I/O pin to an input direction, and a high (1) bit sets it to an output direction. The following code sets I/O pins P0 through P3 to outputs.

```
mov     dira, #$0F
```

See Registers, page 338, and the Spin language DIRA, DIRB section, page 104, for more information. Keep in mind that in Propeller Assembly, unlike in Spin, all 32 bits of DIRA are accessed at once unless the MUXx instructions are used.

DJNZ

Instruction: Decrement value and jump to address if not zero.

DJNZ *Value*, ⟨#⟩ *Address*

Result: *Value*-1 is written to *Value*.

- ***Value*** (d-field) is the register to decrement and test.
- ***Address*** (s-field) is the register or a 9-bit literal whose value is the address to jump to when the decremented *Value* is not zero.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
111001 001i 1111 ddddddddd sssssssss	Result = 0	Unsigned Borrow	Written	4 or 8

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0002; 2	$----_----; -	-	-	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$----_----; -	-	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$----_----; -	-	-	wz wc	$FFFF_FFFF; -1	0	1

Explanation

DJNZ decrements the *Value* register and jumps to *Address* if the result is not zero.

When the WZ effect is specified, the Z flag is set (1) if the decremented *Value* is zero. When the WC effect is specified, the C flag is set (1) if the decrement results in an unsigned borrow (32-bit overflow). The decremented result is written to *Value* unless the NR effect is specified.

DJNZ requires a different amount of clock cycles depending on whether or not it has to jump. If it must jump it takes 4 clock cycles, if no jump occurs it takes 8 clock cycles. Since loops utilizing DJNZ need to be fast, it is optimized in this way for speed.

Effects (WC, WZ, WR, NR)

Every Propeller Assembly instruction has an optional "effects" field that causes it to modify a flag or register when it executes. The basic syntax for Propeller Assembly instructions is:

⟨*Label*⟩ ⟨*Condition*⟩ *Instruction Operands* ⟨*Effects*⟩

The optional *Effects* field can contain one or more of the four items shown below. For any effect not specified next to the instruction in code, the default behavior remains as indicated by the corresponding bit (Z, C, or R) in the **ZCRI** field of the instruction's opcode.

IMPROVED

Table 3-2: Effects	
Effect	**Description**
WC	Updates C Flag. See **WC**, page 372.
WZ	Updates Z Flag. See **WZ**, page 377.
WR	Updates Destination Register. See **WR**, page 373.
NR	Preserves Destination Register. See **NR**, page 325.

Follow an instruction with one to three comma-delimited *Effects* to cause that instruction to affect the indicated item. For example:

```
                and     temp1, #$20       wc
                andn    temp2, #$38       wz, nr
NEW   if_c_and_z  jmp     #MoreCode
```

IMPROVED

The first instruction performs a bitwise AND of the value in the `temp1` register with $20, stores the result in `temp1` and modifies with C flag to indicate the parity of the result. The second instruction performs a bitwise AND NOT of the value in the `temp2` register with $38, modifies the Z flag according to whether or not the result is zero, and does not write the result to `temp2`. During the execution of the first instruction, the Z flag is not altered. During the execution of the second instruction, the C flag is not altered. If these instructions did not include the **WC** and **WZ** effects, those flags would not be altered at all. The third instruction, which specifies a *Condition*, jumps to the `MoreCode` label (not shown) but only if both the C and Z flags are set; otherwise, the **JMP** instruction acts like a **NOP** instruction.

Using *Effects* on instructions, along with *Conditions* on later instructions, enables code to be much more powerful than what is possible with typical assembly languages. See **IF_x** (Conditions) on page 295 for more information.

FIT

Directive: Validate that previous instructions/data fit entirely below a specific address.

FIT ⟨*Address*⟩

Result: Compile-time error if previous instructions/data exceed *Address*-1.

- ***Address*** is an optional Cog RAM address (0-$1F0) for which prior assembly code should not reach. If *Address* is not given, the value $1F0 is used (the address of the first special purpose register).

Explanation

The FIT directive checks the current compile-time cog address pointer and generates an error if it is beyond *Address*-1 or if it is beyond $1EF (the end of general purpose Cog RAM). This directive can be used to ensure that the previous instructions and data fit within Cog RAM, or a limited region of Cog RAM. Note: any instructions that do not fit in Cog RAM will be left out when the assembly code is launched into the cog. Consider the following example:

```
DAT
                ORG     492
Toggle          mov     dira, Pin
:Loop           mov     outa, Pin
                mov     outa, #0
                jmp     #:Loop

Pin     long    $1000

                FIT
```

This code was artificially pushed into upper Cog RAM space by the ORG statement, causing the code to overlap the first special purpose register ($1F0) and causing the FIT directive to cause a compile-time error when the code is compiled.

NEW

FRQA, FRQB

Register: Counter A and Counter B frequency registers.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* FRQA, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, FRQA ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* FRQB, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, FRQB ⟨*Effects*⟩

Result: Optionally, the counter frequency register is updated.

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. FRQA or FRQB may be used in either the assembly instruction's *DestOperand* or *SrcOperand* fields.
- ***SrcOperand*** is a constant expression used by *Instruction* to operate on, and optionally write to, the FRQA or FRQB register in *DestOperand*.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, using the value of FRQA or FRQB in *SrcOperand*.

Explanation

FRQA and FRQB are two of six registers (CTRA, CTRB, FRQA, FRQB, PHSA, and PHSB) that affect the behavior of a cog's Counter Modules. Each cog has two identical counter modules (A and B) that can perform many repetitive tasks. The FRQA and FRQB registers contain the value that is accumulated into the PHSA and PHSB registers, respectively, according to the corresponding counter's mode and input stimulus. See the Spin language CTRA, CTRB section, page 95, for more information.

FRQA and FRQB are read/write registers and can be used in either the *DestOperand* or *SrcOperand* fields of an assembly instruction. The following code sets FRQA to $F. See Registers, page 338, and the Spin language FRQA, FRQB section, page 111, for more information.

```
mov     frqa, #$F
```

HUBOP

Instruction: Perform a hub operation.

HUBOP ***Destination***, ⟨#⟩ ***Operation***

Result: Varies depending on the operation performed.

- ***Destination*** (d-field) is the register containing a value to use in the *Operation*.
- ***Operation*** (s-field) is a register or a 3-bit literal that indicates the hub operation to perform.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
000011 000i 1111	ddddddddd	sssssssss	Result = 0	---	Not Written	7..22

Concise Truth Table:
(Not specified because it varies with each hub operation. See CLKSET, page 271; COGID, page 283; COGINIT, page 284; COGSTOP, page 286; LOCKNEW, page 304; LOCKRET, page 305; LOCKSET, page 306, and LOCKCLR, page 303.)

Explanation

HUBOP is the template for every hub operation instruction in the Propeller chip: CLKSET, COGID, COGINIT, COGSTOP, LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR. The instructions that perform hub operations set the *Operation* field (s-field of the opcode) to the 3-bit immediate value that represents the desired operation (see the opcode of each hub instruction's syntax description for more information). The HUBOP instruction itself should rarely be used, but may be handy for special situations.

HUBOP is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

NEW

IF_x (Conditions)

Every Propeller Assembly instruction has an optional "condition" field that is used to dynamically determine whether or not it executes when it is reached at run time. The basic syntax for Propeller Assembly instructions is:

⟨***Label***⟩ ⟨***Condition***⟩ ***Instruction Operands*** ⟨***Effects***⟩

The optional *Condition* field can contain one of 32 conditions (see Table 3-3) and defaults to **IF_ALWAYS** when no condition is specified. The 4-bit **Value** shown for each condition is the value used for the **–CON–** field in the instruction's opcode.

This feature, along with proper use of instructions' optional *Effects* field, makes Propeller Assembly very powerful. Flags can be affected at will and later instructions can be conditionally executed based on the results. Here's an example:

```
            test   _pins, #$20      wc
            and    _pins, #$38
            shl    t1, _pins
            shr    _pins, #3
            movd   vcfg, _pins
if_nc       mov    dira, t1
if_nc       mov    dirb, #0
if_c        mov    dira, #0
if_c        mov    dirb, t1
```

The first instruction, `test _pins, #$20 wc`, performs its operation and adjusts the state of the C flag because the **WC** effect was specified. The next four instructions perform operations that could affect the C flag, but they do not affect it because no **WC** effect was specified. This means that the state of the C flag is preserved since it was last modified by the first instruction. The last four instructions are conditionally executed based on the state of the C flag that was set five instructions prior. Among the last four instructions, the first two `mov` instructions have `if_nc` conditions, causing them to execute only "if not C" (if C = 0). The last two `mov` instructions have `if_c` conditions, causing them to execute only "if C" (if C = 1). In this case, the two pairs of `mov` instructions are executed in a mutually exclusive fashion.

When an instruction's condition evaluates to **FALSE**, the instruction dynamically becomes a **NOP**, elapsing 4 clock cycles but affecting no flags or registers. This makes the timing of multi-decision code very deterministic.

Table 3-3: Conditions			
Condition	**Instruction Executes**	**Value**	**Synonyms**
IF_ALWAYS	always	1111	
IF_NEVER	never	0000	
IF_E	if equal (Z = 1)	1010	IF_Z
IF_NE	if not equal (Z = 0)	0101	IF_NZ
IF_A	if above (!C & !Z = 1)	0001	IF_NC_AND_NZ –and– IF_NZ_AND_NC
IF_B	if below (C = 1)	1100	IF_C
IF_AE	if above or equal (C = 0)	0011	IF_NC
IF_BE	if below or equal (C | Z = 1)	1110	IF_C_OR_Z –and– IF_Z_OR_C
IF_C	if C set	1100	IF_B
IF_NC	if C clear	0011	IF_AE
IF_Z	if Z set	1010	IF_E
IF_NZ	if Z clear	0101	IF_NE
IF_C_EQ_Z	if C equal to Z	1001	IF_Z_EQ_C
IF_C_NE_Z	if C not equal to Z	0110	IF_Z_NE_C
IF_C_AND_Z	if C set and Z set	1000	IF_Z_AND_C
IF_C_AND_NZ	if C set and Z clear	0100	IF_NZ_AND_C
IF_NC_AND_Z	if C clear and Z set	0010	IF_Z_AND_NC
IF_NC_AND_NZ	if C clear and Z clear	0001	IF_A –and– IF_NZ_AND_NC
IF_C_OR_Z	if C set or Z set	1110	IF_BE –and– IF_Z_OR_C
IF_C_OR_NZ	if C set or Z clear	1101	IF_NZ_OR_C
IF_NC_OR_Z	if C clear or Z set	1011	IF_Z_OR_NC
IF_NC_OR_NZ	if C clear or Z clear	0111	IF_NZ_OR_NC
IF_Z_EQ_C	if Z equal to C	1001	IF_C_EQ_Z
IF_Z_NE_C	if Z not equal to C	0110	IF_C_NE_Z
IF_Z_AND_C	if Z set and C set	1000	IF_C_AND_Z
IF_Z_AND_NC	if Z set and C clear	0010	IF_NC_AND_Z
IF_NZ_AND_C	if Z clear and C set	0100	IF_C_AND_NZ
IF_NZ_AND_NC	if Z clear and C clear	0001	IF_A –and– IF_NC_AND_NZ
IF_Z_OR_C	if Z set or C set	1110	IF_BE –and– IF_C_OR_Z
IF_Z_OR_NC	if Z set or C clear	1011	IF_NC_OR_Z
IF_NZ_OR_C	if Z clear or C set	1101	IF_C_OR_NZ
IF_NZ_OR_NC	if Z clear or C clear	0111	IF_NC_OR_NZ

NEW

INA, INB

Register: Input registers for 32-bit ports A and B.

```
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction  DestOperand, INA ⟨Effects⟩
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction  DestOperand, INB ⟨Effects⟩  (Reserved for future use)
```

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. INA and INB are read-only registers and thus should only be used in the instruction's source operand.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, using the value of the instruction's source operand; INA or INB.

Explanation

INA and INB are one of six special purpose registers (DIRA, DIRB, INA, INB, OUTA and OUTB) that directly affect the I/O pins. The INA and INB register's bits indicate the current logical states of each of the 32 I/O pins in Port A and Port B, respectively. INB is reserved for future use; the Propeller P8X32A does not include Port B I/O pins so only INA is discussed below.

INA is a read-only pseudo-register; when used as an instruction's source operand, it reads the current logic state of the corresponding I/O pins. Do not use INA as the destination operand; that only results in reading and modifying the shadow register whose address INA occupies.

In INA, a low (0) bit indicates the corresponding I/O pin senses ground, and a high (1) bit indicates it senses VDD (3.3 volts). The following code writes the current state of I/O pins P0 through P31 into a register named Temp.

```
mov     Temp, ina
```

See Registers, page 338, and the Spin language INA, INB section, page 118, for more information. Keep in mind that in Propeller Assembly, unlike in Spin, all 32 bits of INA are accessed at once unless the MUXx instructions are used.

JMP

Instruction: Jump to address.

JMP ⟨#⟩ *Address*

- ***Address*** (s-field) is the register or a 9-bit literal whose value is the address to jump to.

Opcode Table:

–INSTR– ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
010111 000i	1111	---------	sssssssss	Result = 0	---	Not Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source	Z	C	Effects	Destination[2]	Z	C[3]
$----_----; -	$----_----; -	-	-	wr wz wc	31:9 unchanged, 8:0 = PC+1	0	1

[1] Destination is normally ignored for typical JMP usage, however if the WR effect is given, the JMP instruction becomes a JMPRET instruction and Destination's s-field (lowest 9 bits) are overwritten with the return address (PC+1).
[2] Destination is not written unless the WR effect is given.
[3] The C flag is set (1) unless PC+1 equals 0; very unlikely since it would require the JMP to be executed from the top of cog RAM ($1FF; special purpose register VSCL).

IMPROVED

Explanation

JMP sets the Program Counter (PC) to *Address* causing execution to jump to that location in Cog RAM. JMP is closely related to the CALL, JMPRET, and RET commands; in fact, they are all the same opcode but with different r-field and i-field values and varying assembler-driven and user-driven d-field and s-field values.

NEW

Conditional Jumps

Conditional jumps, and conditional execution of any instruction, is achieved by preceding the instruction with a condition in the form: IF_x. See IF_x (Conditions) on page 295 for more information.

NEW

The Here Symbol '$'

The 'Here' Symbol, $, represents the current address. During development and debugging, the Here Symbol '$' is often used in the JMP instruction's *Address* field (i.e., JMP #$) to cause the cog to endlessly loop in place. It may also be used as a relative jump back or forward a number of instructions, for example:

```
Toggle          mov     dira, Pin       'Set I/O pin to output
                xor     outa, Pin       'Toggle I/O pin's state
                jmp     #$-1            'Loop back endlessly
```

The last instruction, `JMP #$-1`, causes execution to jump back to the second-to-last instruction (i.e., 'here' minus 1).

JMPRET

Instruction: Jump to address with intention to "return" to another address.

JMPRET *RetInstAddr*, ⟨#⟩ *DestAddress*

Result: PC + 1 is written to the s-field of the register indicated by the d-field.

- ***RetInstAddr*** (d-field) is the register in which to store the return address (PC + 1); often it is the address of an appropriate RET or JMP instruction executed by the *DestAddress* routine.
- ***DestAddress*** (s-field) is the register or 9-bit literal whose value is the address of the routine to temporarily execute.

Opcode Table:

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
010111	001i	1111	ddddddddd	sssssssss	Result = 0	---	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C[2]
$----_----; -	$----_----; -	-	-	WZ WC	31:9 unchanged, 8:0 = PC+1	0	1

[1] The Destination register's s-field (lowest 9 bits) are overwritten with the return address (PC+1) at run-time.

[2] The C flag is set (1) unless PC+1 equals 0; very unlikely since it would require the JMPRET to be executed from the top of cog RAM ($1FF; special purpose register VSCL).

NEW

Explanation

JMPRET (jump and return) provides a mechanism to "call" other routines and eventually return to the instruction that follows the JMPRET. For normal subroutine calls, use the CALL instruction instead since it serves a function similar to its namesake in other processors. The JMPRET instruction provides additional power beyond simple "calling" to execute multiple routines in a task switching manner.

The Propeller hardware does not use a call stack, so the return address of a call-type operation must be stored in a different manner. At run time the JMPRET instruction stores the address of the next instruction (PC + 1) into the source (s-field) of the register at *RetInstAddr*, then jumps to *DestAddress*.

If the *RetInstAddr* register contains a RET or JMP instruction and it is eventually executed by the *DestAddress* routine, the behavior is similar to a CALL instruction; the return address is

stored, the *DestAddress* routine is executed, and finally control returns to the instruction following the JMPRET. See CALL on page 268 for more information.

When used a little differently, the JMPRET instruction can aid in single-process multi-tasking. This is done by defining a set of registers to hold various destination and return addresses and specifying those registers for the *RetInstAddr* and *DestAddress* fields. For example:

```
Initialize      mov     Task2, #SecondTask      'Initialize 1st Dest.

FirstTask       <start of first task>
                ...
                jmpret  Task1, Task2            'Give 2nd task cycles
                <more first task code>
                ...
                jmpret  Task1, Task2            'Give 2nd task cycles
                jmp       #FirstTask            'Loop first task

SecondTask      <start of second task>
                ...
                jmpret  Task2, Task1            'Give 1st task cycles
                <more second task code>
                ...
                jmpret  Task2, Task1            'Give 1st task cycles
                jmp     #SecondTask             'Loop second task

Task1           res     1                       'Declare task address
Task2           res     1                       'storage space
```

In this example there are two routines, FirstTask and SecondTask, which serve as separate tasks in the cog process. The function each task performs is relatively irrelevant; they may do similar or dissimilar operations. Task1 and Task2 are longs, declared at the end of code, used to hold the destination and return addresses that facilitate the switching of execution between the two tasks.

The first instruction, mov Task2,#SecondTask, stores the address of SecondTask into the Task2 register. This primes the task registers for the first task-switch event.

Once FirstTask starts, it performs some operations denoted by "…" and reaches the first JMPRET instruction, jmpret Task1,Task2. First, JMPRET saves the return address (PC + 1, the address of <more first task code>) into the s-field of the Task1 register, then it jumps to the

address indicated by Task2. Since we initialized Task2 to point to SecondTask, the second task will now be executed.

SecondTask performs some operations denoted by "…" and reaches another JMPRET instruction, jmpret Task2,Task1. Note that this is similar to FirstTask's JMPRET instruction except the order of Task1 and Task2 is reversed. This JMPRET instruction saves the return address (PC + 1, the address of <more second task code>) into the s-field of the Task2 register, then it jumps to the address indicated by Task1. Since Task1 contains the address of <more first task code>, as written by the previous JMPRET instruction, execution now switches back to FirstTask starting with the <more first task code> line.

Execution continues to switch back and forth between FirstTask and SecondTask wherever the JMPRET instruction exists, faithfully returning to where the previous task left off last time. Each JMPRET instruction overwrites the previously used destination address with the new return address and then jumps to the new destination; the return address from last time.

This multitasking concept can be applied in a number of ways. For example, removing one of the JMPRET instructions from SecondTask will cause FirstTask to receive fewer cog cycles per unit of time. Techniques like this may be used allocate more cog cycles to time-sensitive tasks, or to time-sensitive portions of tasks. It's also possible to introduce more Task registers to multitask between three or more routines in the same cog. The processing time of each task is always determinate and based upon where the JMPRET instructions are placed and which tasks they refer to.

Note that the state of the flags, C and Z, are unchanged and are not stored between these logical task-switching events. For this reason, it is important to switch between tasks only when the flags are no longer needed or are not in danger of changing states before execution returns.

JMPRET is a superset of the CALL instruction; in fact, it is the same opcode as CALL but with the i-field and d-field configured by the developer, rather than the assembler. See CALL on page 268 for more information.

The return address (PC + 1) is written to the source (s-field) of the *RetInstAddr* register unless the NR effect is specified. Of course, specifying NR is not recommended for the JMPRET instruction since that turns it into a JMP, or RET, instruction.

LOCKCLR

Instruction: Clear lock to false and get its previous state.

LOCKCLR *ID*

Result: Optionally, previous state of lock is written to C flag.

- ***ID*** (d-field) is the register containing the ID (0 – 7) of the lock to clear.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
000011 0001 1111 ddddddddd ------111	ID = 0	Prior Lock State	Not Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source[1]	Z	C	Effects	Destination[2,3]	Z	C
$0000_0005; 5	%0_00000111; 7	–	–	wr wz wc	$0000_0005; 5	0	1[4]
$0000_0005; 5	%0_00000111; 7	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0000; 0	%0_00000111; 7	–	–	wr wz wc	$0000_0000; 0	1	1[4]
$0000_0000; 0	%0_00000111; 7	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0008; 8[5]	%0_00000111; 7	–	–	wr wz wc	$0000_0000; 0	1	0

[1] Source is automatically set to immediate value 7 by the assembler to indicate that this is the LOCKCLR hub instruction.
[2] Destination is not written unless the WR effect is given.
[3] Destination Out (written Destination) indicates the ID of the lock bit that was cleared.
[4] The C flag indicates the previous state of the lock bit; in these cases the lock bit was previously set by a formerly executed LOCKSET instruction (not shown). The next example clears the C flag because the lock bit was just cleared by the previous example.
[5] Only the lowest 3 bits of Destination In are utilized, so a value of 8 is seen as lock bit ID 0.

Explanation

LOCKCLR is one of four lock instructions (LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR) used to manage resources that are user-defined and deemed mutually exclusive. LOCKCLR clears the lock described by the register *ID* to zero (0) and returns the previous state of that lock in the C flag; if the WC effect is specified. The LOCKCLR instruction behaves similar to Spin's LOCKCLR command; see LOCKCLR on page 120.

NEW

If the WZ effect is specified, the Z flag is set (1) if the ID of the cleared lock is zero (0). If the WC effect is specified, the C flag is set equal to the previous state of the lock. If the WR effect is specified, the ID of the cleared lock is written to *ID*.

LOCKCLR is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

LOCKNEW

Instruction: Check out a new lock and get its ID.

LOCKNEW *NewID*

Result: The new lock's ID (0-7) is written to *NewID*.

- ***NewID*** (d-field) is the register where the newly checked-out lock's ID is written.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
000011 0011 1111 dddddddd ------100	ID = 0	No Lock Free	Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source[1]	Z	C	Effects	Destination	Z	C
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0001; 1	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0002; 2	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0003; 3	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0004; 4	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0005; 5	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0006; 6	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0007; 7	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0007; 7	0	1

[1] Source is automatically set to immediate value 4 by the assembler to indicate that this is the LOCKNEW hub instruction.

Explanation

LOCKNEW is one of four lock instructions (LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR) used to manage resources that are user-defined and deemed mutually exclusive. LOCKNEW checks out a unique lock, from the hub, and retrieves the ID of that lock. The LOCKNEW instruction behaves similar to Spin's LOCKNEW command; see LOCKNEW on page 122.

If the WZ effect is specified, the Z flag is set (1) if the returned ID is zero (0). If the WC effect is specified, the C flag is set (1) if no lock was available for checking out. The ID of the newly checked-out lock is written to *NewID* unless the NR effect is specified.

LOCKNEW is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

LOCKRET

Instruction: Release lock back for future "new lock" requests.

LOCKRET *ID*

- ***ID*** (d-field) is the register containing the ID (0 – 7) of the lock to return to the lock pool.

Opcode Table:

IMPROVED

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
000011	0001	1111	dddddddd	------101	ID = 0	No Lock Free	Not Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source[1]	Z	C	Effects	Destination[2]	Z	C[3]
$0000_0000; 0	%0_00000101; 5	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0005; 5	%0_00000101; 5	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0008; 8[4]	%0_00000101; 5	–	–	wr wz wc	$0000_0000; 0	1	0

[1] Source is automatically set to immediate value 5 by the assembler to indicate that this is the LOCKRET hub instruction.
[2] Destination is not written unless the WR effect is given.
[3] The C flag will be set (1) if all lock bits were allocated prior to executing the LOCKRET instruction.
[4] Only the lowest 3 bits of Destination In are utilized, so a value of 8 is seen as lock bit ID 0.

Explanation

LOCKRET is one of four lock instructions (LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR) used to manage resources that are user-defined and deemed mutually exclusive. LOCKRET returns a lock, by *ID*, back to the Hub's lock pool so that it may be reused by other cogs at a later time. The LOCKRET instruction behaves similar to Spin's LOCKRET command; see LOCKRET on page 125.

NEW

If the WZ effect is specified, the Z flag is set (1) if the ID of the returned lock is zero (0). If the WC effect is specified, the C flag is set (1) if all lock bits were allocated prior to executing this instruction. If the WR effect is specified, the ID of the returned lock is written to *ID*.

LOCKRET is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

LOCKSET

Instruction: Set lock to true and get its previous state.

LOCKSET *ID*

Result: Optionally, previous state of lock is written to C flag.

- ***ID*** (d-field) is the register containing the ID (0 – 7) of the lock to set.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
000011 0001 1111 ddddddddd ------110	ID = 0	Prior Lock State	Not Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source[1]	Z	C	Effects	Destination[2,3]	Z	C
$0000_0005; 5	%0_00000110; 6	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0005; 5	%0_00000110; 6	–	–	wr wz wc	$0000_0005; 5	0	1[4]
$0000_0000; 0	%0_00000110; 6	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0000; 0	%0_00000110; 6	–	–	wr wz wc	$0000_0000; 0	1	1[4]
$0000_0008; 8[5]	%0_00000110; 6	–	–	wr wz wc	$0000_0000; 0	1	1[4]

[1] Source is automatically set to immediate value 6 by the assembler to indicate that this is the LOCKSET hub instruction.
[2] Destination is not written unless the WR effect is given.
[3] Destination Out (written Destination) indicates the ID of the lock bit that was set.
[4] The C flag indicates the previous state of the lock bit; in these cases the lock bit was already set from the previous example.
[5] Only the lowest 3 bits of Destination In are utilized, so a value of 8 is seen as lock bit ID 0.

Explanation

LOCKSET is one of four lock instructions (LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR) used to manage resources that are user-defined and deemed mutually exclusive. LOCKSET sets the lock described by the register *ID* to one (1) and returns the previous state of that lock in the C flag; if the WC effect is specified. The LOCKSET instruction behaves similar to Spin's LOCKSET command; see LOCKSET on page 126.

NEW

If the WZ effect is specified, the Z flag is set (1) if the ID of the set lock is zero (0). If the WC effect is specified, the C flag is set equal to the previous state of the lock. If the WR effect is specified, the ID of the set lock is written to *ID*.

LOCKSET is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

MAX

Instruction: Limit maximum of unsigned value to another unsigned value.

MAX *Value1*, ⟨#⟩ *Value2*

Result: Lesser of unsigned *Value1* and unsigned *Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to compare against *Value2* and is the destination in which to write the lesser of the two.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is compared against *Value1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
010011 001i 1111 ddddddddd sssssssss	S = 0	Unsigned (D < S)	Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source[1]	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0

[1] Both Source and Destination are treated as unsigned values.

Explanation

MAX compares the unsigned values of *Value1* and *Value2* and stores the lesser of the two into the *Value1* register, effectively limiting *Value1* to a maximum of *Value2*.

IMPROVED

If the WZ effect is specified, the Z flag is set (1) if *Value2* is zero (0). If the WC effect is specified, the C flag is set (1) if the unsigned *Value1* is less than the unsigned *Value2*. The lesser of the two values is written to *Value1* unless the NR effect is specified.

MAXS

Instruction: Limit maximum of signed value to another signed value.

MAXS *SValue1*, ⟨#⟩ *SValue2*

Result: Lesser of signed *SValue1* and signed *SValue2* is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to compare against *SValue2* and is the destination in which to write the lesser of the two.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is compared against *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
010001 001i 1111	dddddddd	ssssssss	S = 0	Signed (D < S)	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$FFFF_FFFF; -1	–	–	wz wc	$FFFF_FFFF; -1	0	0
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	1
$FFFF_FFFF; -1	$0000_0001; 1	–	–	wz wc	$FFFF_FFFF; -1	0	1
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0

Explanation

MAXS compares the signed values of *SValue1* and *SValue2* and stores the lesser of the two into the *SValue1* register, effectively limiting *SValue1* to a maximum of *SValue2*.

IMPROVED

If the WZ effect is specified, the Z flag is set (1) if *SValue2* is zero (0). If the WC effect is specified, the C flag is set (1) if the signed *SValue1* is less than the signed *SValue2*. The lesser of the two values is written to *SValue1* unless the NR effect is specified.

MIN

Instruction: Limit minimum of unsigned value to another unsigned value.

MIN *Value1*, ⟨#⟩ *Value2*

Result: Greater of unsigned *Value1* and unsigned *Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to compare against *Value2* and is the destination in which to write the greater of the two.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is compared against *Value1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
010010 001i 1111 ddddddddd sssssssss	S = 0	Unsigned (D < S)	Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source[1]	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0001; 2	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0001; 1	1	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1

[1] Both Source and Destination are treated as unsigned values.

Explanation

MIN compares the unsigned values of *Value1* and *Value2* and stores the greater of the two into the *Value1* register, effectively limiting *Value1* to a minimum of *Value2*.

IMPROVED

If the WZ effect is specified, the Z flag is set (1) if *Value2* is zero (0). If the WC effect is specified, the C flag is set (1) if the unsigned *Value1* is less than the unsigned *Value2*. The greater of the two values is written to *Value1* unless the NR effect is specified.

MINS

Instruction: Limit minimum of signed value to another signed value.

MINS *SValue1*, ⟨#⟩ *SValue2*

Result: Greater of signed *SValue1* and signed *SValue2* is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to compare against *SValue2* and is the destination in which to write the greater of the two.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is compared against *SValue1*.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
010000 001i 1111 ddddddddd sssssssss	S = 0	Signed (D < S)	Written	4

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0002; 2	-	-	wz wc	$0000_0002; 2	0	1
$0000_0001; 1	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0000; 0	-	-	wz wc	$0000_0001; 1	1	0
$0000_0001; 1	$FFFF_FFFF; -1	-	-	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0001; 1	-	-	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0
$0000_0000; 0	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	1
$FFFF_FFFF; -1	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	1

Explanation

MINS compares the signed values of *SValue1* and *SValue2* and stores the greater of the two into the *SValue1* register, effectively limiting *SValue1* to a minimum of *SValue2*.

If the WZ effect is specified, the Z flag is set (1) if *SValue2* is zero (0). If the WC effect is specified, the C flag is set (1) if the signed *SValue1* is less than the signed *SValue2*. The greater of the two values is written to *SValue1* unless the NR effect is specified.

MOV

Instruction: Set a register to a value.

MOV *Destination*, ⟨#⟩ *Value*

Result: *Value* is stored in *Destination*.

- ***Destination*** (d-field) is the register in which to store *Value*.
- ***Value*** (s-field) is a register or a 9-bit literal whose value is stored into *Destination*.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
101000 001i 1111	ddddddddd	sssssssss	Result = 0	S[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0

Explanation

MOV copies, or stores, the number in *Value* into *Destination*.

If the WZ effect is specified, the Z flag is set (1) if *Value* equals zero. If the WC effect is specified, the C flag is set to *Value's* MSB. The result is written to *Destination* unless the NR effect is specified.

MOVD

Instruction: Set a register's destination field to a value.

MOVD *Destination*, ⟨#⟩ *Value*

Result: *Value* is stored in *Destination's* d-field (bits 17..9).

- ***Destination*** (d-field) is the register whose destination field (bits 17..9) is set to *Value's* value.
- ***Value*** (s-field) is a register or a 9-bit literal whose value is stored into *Destination's* d-field.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
010101 001i 1111	ddddddddd	sssssssss	Result = 0	---	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_01FF; 511	-	-	wz wc	$0003_FE00; 261,632	0	1
$FFFF_FFFF; -1	$0000_01FF; 511	-	-	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$0000_0000; 0	-	-	wz wc	$FFFC_01FF; -261,633	0	0

Explanation

MOVD copies the 9-bit value of *Value* into *Destination's* d-field (destination field) bits 17..9. *Destination's* other bits are left unchanged. This instruction is handy for setting certain registers like CTRA and VCFG, and for updating the destination field of instructions in self-modifying code.

NEW

For self-modifying code, ensure that at least one other instruction is executed between a MOVD instruction and the target instruction that MOVD modifies. This gives the cog time to write MOVD's result before it fetches the target instruction for execution; otherwise, the as-yet-unmodified version of the target instruction would be fetched and executed.

If the WZ effect is specified, the Z flag is set (1) if *Value* equals zero. The result is written to *Destination* unless the NR effect is specified.

MOVI

Instruction: Set a register's instruction and effects fields to a value.

MOVI *Destination*, {#} *Value*

Result: *Value* is stored in *Destination's* i-field and effects-field (bits 31..23).

- ***Destination*** (d-field) is the register whose instruction and effects fields (bits 31..23) are set to *Value's* value.
- ***Value*** (s-field) is a register or a 9-bit literal whose value is stored into *Destination's* instruction and effects field.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
010110 001i 1111 ddddddddd sssssssss	Result = 0	---	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_01FF; 511	–	–	wz wc	$FF80_0000; -8,388,608	0	1
$FFFF_FFFF; -1	$0000_01FF; 511	–	–	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$0000_0000; 0	–	–	wz wc	$007F_FFFF; 8,388,607	0	0

Explanation

MOVI copies the 9-bit value of *Value* into *Destination's* instruction and effects fields bits 31..23. *Destination's* other bits are left unchanged. This instruction is handy for setting certain registers like CTRA and VCFG, and for updating the instruction and effects fields of instructions in self-modifying code.

NEW

For self-modifying code, ensure that at least one other instruction is executed between a MOVI instruction and the target instruction that MOVI modifies. This gives the cog time to write MOVI's result before it fetches the target instruction for execution; otherwise, the as-yet-unmodified version of the target instruction would be fetched and executed.

If the WZ effect is specified, the Z flag is set (1) if *Value* equals zero. The result is written to *Destination* unless the NR effect is specified.

MOVS

Instruction: Set a register's source field to a value.

MOVS *Destination*, ⟨#⟩ *Value*

Result: *Value* is stored in *Destination'* s-field (bits 8..0).

- ***Destination*** (d-field) is the register whose source field (bits 8..0) is set to *Value's* value.
- ***Value*** (s-field) is a register or a 9-bit literal whose value is stored into *Destination's* source field.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
010100 001i 1111	ddddddddd	sssssssss	Result = 0	---	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_01FF; 511	-	-	wz wc	$0000_01FF; 511	0	1
$FFFF_FFFF; -1	$0000_01FF; 511	-	-	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$0000_0000; 0	-	-	wz wc	$FFFF_FE00; -512	0	0

Explanation

MOVS copies the 9-bit value of *Value* into *Destination's* source field (s-field) bits 8..0. *Destination's* other bits are left unchanged. This instruction is handy for setting certain registers like CTRA and VCFG, and for updating the source field of instructions in self-modifying code.

NEW

For self-modifying code, ensure that at least one other instruction is executed between a MOVS instruction and the target instruction that MOVS modifies. This gives the cog time to write MOVS's result before it fetches the target instruction for execution; otherwise, the as-yet-unmodified version of the target instruction would be fetched and executed.

If the WZ effect is specified, the Z flag is set (1) if *Value* equals zero. The result is written to *Destination* unless the NR effect is specified.

MUXC

Instruction: Set discrete bits of a value to the state of C.

MUXC *Destination*, ⟨#⟩ *Mask*

Result: *Destination's* bits, indicated by *Mask*, are set to the state of C.

- ***Destination*** (d-field) is the register whose bits described by *Mask* are affected by C.
- ***Mask*** (s-field) is a register or a 9-bit literal whose value contains high (1) bits for every bit in *Destination* to set to the C flag's state.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
011100 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0001; 1	-	0	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	-	1	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0003; 3	-	0	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	-	1	wz wc	$0000_0003; 3	0	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	-	0	wz wc	$A841_2000; -1,472,126,976	0	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	-	1	wz wc	$BA75_7678; -1,166,707,080	0	1
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	-	0	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	-	1	wz wc	$FFFF_FFFF; -1	0	0

Explanation

MUXC sets each bit of the value in *Destination*, which corresponds to *Mask's* high (1) bits, to the C state. All bits of *Destination* that are not targeted by high (1) bits of *Mask* are unaffected. This instruction is handy for setting or clearing discrete bits, or groups of bits, in an existing value.

If the WZ effect is specified, the Z flag is set (1) if *Destination's* final value is 0. If the WC effect is specified, the C flag is set (1) if the resulting *Destination* contains an odd number of high (1) bits. The result is written to *Destination* unless the NR effect is specified.

MUXNC

Instruction: Set discrete bits of a value to the state of !C.

MUXNC *Destination*, ⟨#⟩ *Mask*

Result: *Destination's* bits, indicated by *Mask*, are set to the state of !C.

- ***Destination*** (d-field) is the register whose bits described by *Mask* are affected by !C.
- ***Mask*** (s-field) is a register or a 9-bit literal whose value contains high (1) bits for every bit in *Destination* to set to the inverse of the C flag's state.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
011101 001i 1111	ddddddddd	sssssssss	Result = 0	Parity of Result	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0001; 1	-	0	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0001; 1	-	1	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	-	0	wz wc	$0000_0003; 3	0	0
$0000_0000; 0	$0000_0003; 3	-	1	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	-	0	wz wc	$BA75_7678; -1,166,707,080	0	1
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	-	1	wz wc	$A841_2000; -1,472,126,976	0	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	-	0	wz wc	$FFFF_FFFF; -1	0	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	-	1	wz wc	$0000_0000; 0	1	0

Explanation

MUXNC sets each bit of the value in *Destination*, which corresponds to *Mask's* high (1) bits, to the !C state. All bits of *Destination* that are not targeted by high (1) bits of *Mask* are unaffected. This instruction is handy for setting or clearing discrete bits, or groups of bits, in an existing value.

If the WZ effect is specified, the Z flag is set (1) if *Destination's* final value is 0. If the WC effect is specified, the C flag is set (1) if the resulting *Destination* contains an odd number of high (1) bits. The result is written to *Destination* unless the NR effect is specified.

MUXNZ

Instruction: Set discrete bits of a value to the state of !Z.

MUXNZ *Destination*, ⟨#⟩ *Mask*

Result: *Destination's* bits, indicated by *Mask*, are set to the state of !Z.

- ***Destination*** (d-field) is the register whose bits described by *Mask* are affected by !Z.
- ***Mask*** (s-field) is a register or a 9-bit literal whose value contains high (1) bits for every bit in *Destination* to set to the inverse of the Z flag's state.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
011111 001i 1111	dddddddd	ssssssss	Result = 0	Parity of Result	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0001; 1	0	-	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0001; 1	1	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	0	-	wz wc	$0000_0003; 3	0	0
$0000_0000; 0	$0000_0003; 3	1	-	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	0	-	wz wc	$BA75_7678; -1,166,707,080	0	1
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	1	-	wz wc	$A841_2000; -1,472,126,976	0	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	0	-	wz wc	$FFFF_FFFF; -1	0	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	1	-	wz wc	$0000_0000; 0	1	0

Explanation

MUXNZ sets each bit of the value in *Destination*, which corresponds to *Mask's* high (1) bits, to the !Z state. All bits of *Destination* that are not targeted by high (1) bits of *Mask* are unaffected. This instruction is handy for setting or clearing discrete bits, or groups of bits, in an existing value.

If the WZ effect is specified, the Z flag is set (1) if *Destination's* final value is 0. If the WC effect is specified, the C flag is set (1) if the resulting *Destination* contains an odd number of high (1) bits. The result is written to *Destination* unless the NR effect is specified.

MUXZ

Instruction: Set discrete bits of a value to the state of Z.

MUXZ *Destination*, ⟨#⟩ *Mask*

Result: *Destination's* bits, indicated by *Mask*, are set to the state of Z.

- ***Destination*** (d-field) is the register whose bits described by *Mask* are affected by Z.
- ***Mask*** (s-field) is a register or a 9-bit literal whose value contains high (1) bits for every bit in *Destination* to set to the Z flag's state.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
011110 001i 1111	ddddddddd	sssssssss	Result = 0	Parity of Result	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0001; 1	0	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	1	–	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0003; 3	0	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	1	–	wz wc	$0000_0003; 3	0	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	0	–	wz wc	$A841_2000; -1,472,126,976	0	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	1	–	wz wc	$BA75_7678; -1,166,707,080	0	1
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	0	–	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	1	–	wz wc	$FFFF_FFFF; -1	0	0

Explanation

MUXZ sets each bit of the value in *Destination*, which corresponds to *Mask's* high (1) bits, to the Z state. All bits of *Destination* that are not targeted by high (1) bits of *Mask* are unaffected. This instruction is handy for setting or clearing discrete bits, or groups of bits, in an existing value.

If the WZ effect is specified, the Z flag is set (1) if *Destination's* final value is 0. If the WC effect is specified, the C flag is set (1) if the resulting *Destination* contains an odd number of high (1) bits. The result is written to *Destination* unless the NR effect is specified.

NEG

Instruction: Get the negative of a number.

NEG *NValue*, ⟨#⟩ *SValue*

Result: –*SValue* is stored in *NValue*.

- ***NValue*** (d-field) is the register in which to write the negative of *SValue*.
- ***SValue*** (s-field) is a register or a 9-bit literal whose negative value will be written to *NValue*.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
101001 001i 1111	ddddddddd	sssssssss	Result = 0	S[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	-	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	-	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

[1] The smallest negative number (-2,147,483,648) has no corresponding positive value in 32-bit two's-complement math.

Explanation

NEG stores negative *SValue* into *NValue*.

If the WZ effect is specified, the Z flag is set (1) if *SValue* is zero. If the WC effect is specified, the C flag is set (1) if *SValue* is negative or cleared (0) if *SValue* is positive. The result is written to *NValue* unless the NR effect is specified.

NEGC

Instruction: Get a value, or its additive inverse, based on C.

NEGC *RValue*, ⟨#⟩ *Value*

Result: *Value* or –*Value* is stored in *RValue*.

- ***RValue*** (d-field) is the register in which to write *Value* or –*Value*.
- ***Value*** (s-field) is a register or a 9-bit literal whose value (if C = 0) or additive inverse value (if C = 1) will be written to *RValue*.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
101100 001i 1111	ddddddddd	sssssssss	Result = 0	S[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$----_----; -	$FFFF_FFFF; -1	-	0	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$FFFF_FFFF; -1	-	1	wz wc	$0000_0001; 1	0	1
$----_----; -	$0000_0000; 0	-	x	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	0	wz wc	$0000_0001; 1	0	0
$----_----; -	$0000_0001; 1	-	1	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	1	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	x	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	0	wz wc	$8000_0001; -2,147,483,647	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

[1] **The smallest negative number (-2,147,483,648) has no corresponding positive value in 32-bit two's-complement math.**

Explanation

NEGC stores *Value* (if C = 0) or –*Value* (if C = 1) into *RValue*.

If the WZ effect is specified, the Z flag is set (1) if *Value* is zero. If the WC effect is specified, the C flag is set (1) if *Value* is negative or cleared (0) if *Value* is positive. The result is written to *RValue* unless the NR effect is specified.

NEGNC

Instruction: Get a value, or its additive inverse, based on !C.

NEGNC *RValue*, ⟨#⟩ *Value*

Result: –*Value* or *Value* is stored in *RValue*.

- ***RValue*** (d-field) is the register in which to write –*Value* or Value.
- ***Value*** (s-field) is a register or a 9-bit literal whose additive inverse value (if C = 0) or value (if C = 1) will be written to *RValue*.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
101101 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4

NEW

Concise Truth Table:

					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$----_----; -	$FFFF_FFFF; -1	-	0	wz wc	$0000_0001; 1	0	1
$----_----; -	$FFFF_FFFF; -1	-	1	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$0000_0000; 0	-	x	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	0	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$0000_0001; 1	-	1	wz wc	$0000_0001; 1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	0	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	x	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	1	wz wc	$8000_0001; -2,147,483,647	0	1

[1] The smallest negative number (-2,147,483,648) has no corresponding positive value in 32-bit two's-complement math.

Explanation

NEGNC stores –*Value* (if C = 0) or *Value* (if C = 1) into *RValue*.

If the WZ effect is specified, the Z flag is set (1) if *Value* is zero. If the WC effect is specified, the C flag is set (1) if *Value* is negative or cleared (0) if *Value* is positive. The result is written to *RValue* unless the NR effect is specified.

NEGNZ

Instruction: Get a value, or its additive inverse, based on !Z.

NEGNZ *RValue*, ⟨#⟩ *Value*

Result: –*Value* or *Value* is stored in *RValue*.

- ***RValue*** (d-field) is the register in which to write –*Value* or *Value*.
- ***Value*** (s-field) is a register or a 9-bit literal whose additive inverse value (if Z = 0) or value (if Z = 1) will be written to *RValue*.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
101111 001i 1111	ddddddddd	sssssssss	Result = 0	S[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$----_----; -	$FFFF_FFFF; -1	0	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$FFFF_FFFF; -1	1	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$0000_0000; 0	x	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	0	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$0000_0001; 1	1	-	wz wc	$0000_0001; 1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	0	-	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	1	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	x	-	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	0	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$----_----; -	$8000_0001; -2,147,483,647	1	-	wz wc	$8000_0001; -2,147,483,647	0	1

[1] **The smallest negative number (-2,147,483,648) has no corresponding positive value in 32-bit two's-complement math.**

Explanation

NEGNZ stores –*Value* (if Z = 0) or *Value* (if Z = 1) into *RValue*.

If the WZ effect is specified, the Z flag is set (1) if *Value* is zero. If the WC effect is specified, the C flag is set (1) if *Value* is negative or cleared (0) if *Value* is positive. The result is written to *RValue* unless the NR effect is specified.

NEGZ

Instruction: Get a value, or its additive inverse, based on Z.

NEGZ *RValue*, ⟨#⟩ *Value*

Result: *Value* or –*Value* is stored in *RValue*.

- ***RValue*** (d-field) is the register in which to write *Value* or –*Value*.
- ***Value*** (s-field) is a register or a 9-bit literal whose value (if Z = 0) or additive inverse value (if Z = 1) will be written to *RValue*.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
101110 001i 1111	ddddddddd	sssssssss	Result = 0	S[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$----_----; -	$FFFF_FFFF; -1	0	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$FFFF_FFFF; -1	1	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$0000_0000; 0	x	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	0	-	wz wc	$0000_0001; 1	0	0
$----_----; -	$0000_0001; 1	1	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	0	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	1	-	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	x	-	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	0	-	wz wc	$8000_0001; -2,147,483,647	0	1
$----_----; -	$8000_0001; -2,147,483,647	1	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

[1] The smallest negative number (-2,147,483,648) has no corresponding positive value in 32-bit two's-complement math.

Explanation

NEGZ stores *Value* (if Z = 0) or –*Value* (if Z = 1) into *RValue*.

If the WZ effect is specified, the Z flag is set (1) if *Value* is zero. If the WC effect is specified, the C flag is set (1) if *Value* is negative or cleared (0) if *Value* is positive. The result is written to *RValue* unless the NR effect is specified.

NOP

Instruction: No operation, just elapse four clock cycles.

```
NOP
```

Opcode Table:

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
000000	----	0000	---------	---------	---	---	---	4

Concise Truth Table:
(Not specified because NOP performs no action.)

Explanation

IMPROVED

NOP performs no operation but consumes 4 clock cycles. NOP has its –CON– field set to all zeros, the NEVER condition; effectively, every instruction with a NEVER condition is a NOP instruction. Because of this, the NOP instruction can never be preceded by a *Condition*, such as IF_Z or IF_C_AND_Z, since it can never be conditionally executed.

NEW

NR

Effect: Prevent assembly instruction from writing a result.

⟨***Label***⟩ ⟨***Condition***⟩ ***Instruction Operands*** NR

Result: *Instruction*'s destination register is left unaffected.

- ***Label*** is an optional statement label. See Common Syntax Elements on page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements on page 250.
- ***Instruction*** is the desired assembly instruction.
- ***Operands*** is zero, one, or two operands as required by the *Instruction*.

Explanation

NR (No Result) is one of four optional effects (WC, WZ, WR, and NR) that influence the behavior of assembly instructions. NR causes an executing assembly instruction to leave the destination register's value unaffected.

For example, by default the SHL (Shift Left) instruction shifts the destination value left a number of bits, writes the result back into the destination register, and optionally indicates status via the C and Z flags. If all that you really need is the SHL instruction's C flag status, simply specify it with both the WC and NR effects:

```
shl     value, #1     WC, NR     'Put value's MSB in C
```

The above example effectively sets the C flag to the state of value's high bit (bit 31) without affecting the final contents of value.

See Effects on page 291 for more information.

Operators

Propeller Assembly code can contain expressions using any operators that are allowed in constant expressions. Table 3-4 summarizes all the operators allowed in Propeller Assembly code. Please refer to the Spin Language Reference Operators section on page 143 for detailed descriptions of their functions; page numbers are given in the table.

Table 3-4: Constant Expression Math/Logic Operators

Normal Operator	Is Unary	Description, Page Number
+		Add, 149
+	✓	Positive (+X); unary form of Add, 150
-		Subtract, 150
-	✓	Negate (-X); unary form of Subtract, 150
*		Multiply and return lower 32 bits (signed), 153
**		Multiply and return upper 32 bits (signed), 153
/		Divide (signed), 154
//		Modulus (signed), 154
#>		Limit minimum (signed), 155
<#		Limit maximum (signed), 155
^^	✓	Square root, 156
||	✓	Absolute value, 156
~>		Shift arithmetic right, 158
|<	✓	Bitwise: Decode value (0-31) into single-high-bit long, 160
>|	✓	Bitwise: Encode long into value (0 - 32) as high-bit priority, 160
<<		Bitwise: Shift left, 161
>>		Bitwise: Shift right, 161
<-		Bitwise: Rotate left, 162
->		Bitwise: Rotate right, 162
><		Bitwise: Reverse, 163
&		Bitwise: AND, 164
|		Bitwise: OR, 165
^		Bitwise: XOR, 165
!	✓	Bitwise: NOT, 166
AND		Boolean: AND (promotes non-0 to -1), 167
OR		Boolean: OR (promotes non-0 to -1), 168
NOT	✓	Boolean: NOT (promotes non-0 to -1), 168
==		Boolean: Is equal, 169
<>		Boolean: Is not equal, 170
<		Boolean: Is less than (signed), 170
>		Boolean: Is greater than (signed), 171
=<		Boolean: Is equal or less (signed), 171
=>		Boolean: Is equal or greater (signed), 172
@	✓	Symbol address, 173

OR

Instruction: Bitwise OR two values.

OR *Value1*, ⟨#⟩ *Value2*

Result: *Value1* OR *Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to bitwise OR with *Value2* and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is bitwise ORed with *Value1*.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
011010 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1
$0000_000A; 10	$0000_0005; 5	–	–	wz wc	$0000_000F; 15	0	0

Explanation

OR (bitwise inclusive OR) performs a bitwise OR of the value in *Value2* into that of *Value1*.

If the WZ effect is specified, the Z flag is set (1) if *Value1* OR *Value2* equals zero. If the WC effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits. The result is written to *Value1* unless the NR effect is specified.

ORG

Directive: Adjust compile-time assembly pointer.

ORG ⟨*Address*⟩

- ***Address*** is an optional Cog RAM address (0-495) to assemble the following assembly code with. If *Address* is not given, the value 0 is used.

NEW

Explanation

The **ORG** (origin) directive sets the Propeller Tool's assembly pointer to a new value for use in address references within the assembly code to follow. **ORG** typically appears as ORG 0, or just ORG, at the start of any new assembly code intended to be launched into a separate cog.

ORG only affects symbol references; it does not affect the position of assembly code in the cog itself. When assembly code is launched via a **COGNEW** or **COGINIT** command, the destination cog always loads the code into its RAM starting at address 0.

Even though assembly code is always loaded in this way, the compiler/assembler does not know which part of the code constitutes its beginning since developers are free to launch any code starting from any address.

To solve this, the assembler uses a reference point (the assembly pointer value) to calculate the absolute address of referenced symbols. Those absolute addresses are encoded into the assembly instruction's destination (d-field) or source (s-field) in place of the symbolic reference. For example:

```
DAT
                org     0               'Start at Cog RAM 0
Toggle          mov     dira, Pin       'Set I/O direction to output
:Loop           xor     outa, Pin       'Toggle output pin state
                jmp     #:Loop          'Loop endlessly

Pin             long    $0000_0010      'Use I/O pin 4 ($10 or %1_0000)
```

The **ORG** statement in this example sets the assembly pointer to zero (0) so the code following it is assembled with that reference point in mind. Because of this, to the assembler the Toggle symbol is logically at Cog RAM location 0, the :Loop symbol is at Cog RAM location 1, and the Pin symbol is at Cog RAM location 3. The assembler will replace each reference to these symbols with their respective hard-coded location.

When the Toggle code is launched with COGNEW(@Toggle, 0), for example, the code will properly execute starting with Cog RAM address 0 since all symbol addresses were calculated from that point. If the ORG statement had been ORG 1 and the Toggle code was launched, it would not execute properly because the symbol addresses were calculated from the wrong reference (1 instead of 0).

A Propeller object may contain multiple instances of the ORG directive, each placed immediately before a launchable portion of assembly code. However, it is not common to use ORG with values other than zero (0) though that may be handy when creating run-time swappable portions of assembly code.

NEW

OUTA, OUTB

Register: Output registers for 32-bit ports A and B.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* OUTA, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, OUTA ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* OUTB, *SrcOperand* ⟨*Effects*⟩ (Reserved for future use)

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, OUTB ⟨*Effects*⟩ (Reserved for future use)

Result: Optionally, the output register is updated.

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. OUTA or OUTB may be used in either the assembly instruction's *DestOperand* or *SrcOperand* fields.
- ***SrcOperand*** is a constant expression used by *Instruction* to operate on, and optionally write to, the OUTA or OUTB register in *DestOperand*.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, using the value of OUTA or OUTB in *SrcOperand*.

Explanation

OUTA and OUTB are one of six special purpose registers (DIRA, DIRB, INA, INB, OUTA and OUTB) that directly affect the I/O pins. The OUTA and OUTB register's bits indicate the output states of each of the 32 I/O pins in Port A and Port B, respectively. OUTB is reserved for future use; the Propeller P8X32A does not include Port B I/O pins so only OUTA is discussed below.

OUTA is a read/write register usable in an instruction's *DestOperand* or *SrcOperand* fields. If the I/O pin is set to output, a low (0) bit in OUTA causes it to output ground, and a high (1) bit causes it to output VDD (3.3 volts). The following code sets I/O pins P0 through P3 to output high.

```
mov     dira, #$0F
mov     outa, #$0F
```

See Registers, page 338, and the Spin language OUTA, OUTB section, page 175, for more information. Keep in mind that in Propeller Assembly, unlike in Spin, all 32 bits of OUTA are accessed at once unless the MUXx instructions are used.

NEW

PAR

Register: Cog boot parameter register.

DAT
〈*Label*〉 〈*Condition*〉 *Instruction* *DestOperand*, PAR 〈*Effects*〉

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. PAR is a read-only register and thus should only be used in the instruction's source operand.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, by the value of PAR in the instruction's source operand.

Explanation

The PAR register contains the address value passed into the *Parameter* field of a Spin-based COGINIT or COGNEW command, or into the upper bits of the *Destination* field of an assembly-based COGINIT command. When the cog starts up, its Propeller Assembly code can use the PAR register's contents to locate and operate on main memory shared between it and the calling code.

It's important to note that the value passed into PAR is intended to be a long address, so only 14-bits (bits 2 through 15) are retained; the lower two bits are cleared to zero to ensure that it's a long-aligned address. Values other than long addresses can still be passed, but must be 14 bits or less and shifted left and right appropriately by both caller and newly started cog.

PAR is a read-only pseudo-register; when used as an instruction's source operand, it reads the value passed to the cog upon launch. Do not use PAR as the destination operand; that only results in reading and modifying the shadow register whose address PAR occupies.

The following code moves the value of PAR into the register Addr for further use later in code. See Registers, page 338, and the Spin language PAR section, page 178, for more information.

```
DAT
                org 0                       'Reset assembly pointer
AsmCode         mov     Addr,  par          'Get shared address
                '<more code here>           'Perform operation

Addr    res     1
```

NEW

PHSA, PHSB

Register: Counter A and Counter B phase registers.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* PHSA, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, PHSA ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* PHSB, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, PHSB ⟨*Effects*⟩

Result: Optionally, the counter phase register is updated.

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. PHSA and PHSB are read/write pseudo-registers that act differently in the *SrcOperand* than they do in the *DestOperand*.
- ***SrcOperand*** is a constant expression used by *Instruction* to operate on, and optionally write to, the PHSA or PHSB register in *DestOperand*.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, using the value of PHSA or PHSB in *SrcOperand*.

Explanation

PHSA and PHSB are two of six registers (CTRA, CTRB, FRQA, FRQB, PHSA, and PHSB) that affect the behavior of a cog's Counter Modules. Each cog has two identical counter modules (A and B) that can perform many repetitive tasks. The PHSA and PHSB registers contain the accumulations of the FRQA and FRQB registers' value, respectively, according to the corresponding counter's mode and input stimulus. See the Spin language CTRA, CTRB section on page 95 for more information.

PHSA and PHSB are read/write pseudo-registers. In the *SrcOperand* they read the counter's accumulator value. In the *DestOperand* they read the shadow register whose address PHSA or PHSB occupies, but modifications affect both the shadow and accumulator registers.

The following code moves the value of PHSA into Result. See Registers, page 338, and the Spin language PHSA, PHSB section, page 180, for more information.

```
        mov     Result, phsa    'Get current phase value
```

RCL

Instruction: Rotate C left into value by specified number of bits.

RCL *Value*, ⟨#⟩ *Bits*

Result: *Value* has *Bits* copies of C rotated left into it.

- ***Value*** (d-field) is the register in which to rotate C leftwards.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits of *Value* to rotate C leftwards into.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
001101 001i 1111	dddddddd	ssssssss	Result = 0	D[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$8000_0000; -2,147,483,648	$0000_0000; 0	-	x	wz wc	$8000_0000; -2,147,483,648	0	1
$8000_0000; -2,147,483,648	$0000_0001; 1	-	0	wz wc	$0000_0000; 0	1	1
$8000_0000; -2,147,483,648	$0000_0001; 1	-	1	wz wc	$0000_0001; 1	0	1
$2108_4048; 554,188,872	$0000_0002; 2	-	0	wz wc	$8421_0120; -2,078,211,808	0	0
$2108_4048; 554,188,872	$0000_0002; 2	-	1	wz wc	$8421_0123; -2,078,211,805	0	0
$8765_4321; -2,023,406,815	$0000_0004; 4	-	0	wz wc	$7654_3210; 1,985,229,328	0	1
$8765_4321; -2,023,406,815	$0000_0004; 4	-	1	wz wc	$7654_321F; 1,985,229,343	0	1

Explanation

RCL (Rotate Carry Left) performs a rotate left of *Value*, *Bits* times, using the C flag's original value for each of the LSBs affected.

If the WZ effect is specified, the Z flag is set (1) if the resulting *Value* equals zero. If the WC effect is specified, at the end of the operation, the C flag is set equal to *Value's* original bit 31. The result is written to *Value* unless the NR effect is specified.

RCR

Instruction: Rotate C right into value by specified number of bits.

RCR Value, ⟨#⟩ Bits

Result: *Value* has *Bits* copies of C rotated right into it.

- ***Value*** (d-field) is the register in which to rotate C rightwards.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits of *Value* to rotate C rightwards into.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
001100 001i 1111	ddddddddd	sssssssss	Result = 0	D[0]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0000; 0	–	×	wz wc	$0000_0001; 1	0	1
$0000_0001; 1	$0000_0001; 1	–	0	wz wc	$0000_0000; 0	1	1
$0000_0001; 1	$0000_0001; 1	–	1	wz wc	$8000_0000; -2,147,483,648	0	1
$18C2_1084; 415,371,396	$0000_0002; 2	–	0	wz wc	$0630_8421; 103,842,849	0	0
$18C2_1084; 415,371,396	$0000_0002; 2	–	1	wz wc	$C630_8421; -969,898,975	0	0
$8765_4321; -2,023,406,815	$0000_0004; 4	–	0	wz wc	$0876_5432; 141,972,530	0	1
$8765_4321; -2,023,406,815	$0000_0004; 4	–	1	wz wc	$F876_5432; -126,462,926	0	1

Explanation

RCR (Rotate Carry Right) performs a rotate right of *Value*, *Bits* times, using the C flag's original value for each of the MSBs affected.

If the WZ effect is specified, the Z flag is set (1) if the resulting *Value* equals zero. If the WC effect is specified, at the end of the operation, the C flag is set equal to *Value's* original bit 0. The result is written to *Value* unless the NR effect is specified.

RDBYTE

Instruction: Read byte of main memory.

RDBYTE *Value*, ⟨#⟩ *Address*

Result: Zero-extended byte is stored in *Value*.

- ***Value*** (d-field) is the register to store the zero-extended byte value into.
- ***Address*** (s-field) is a register or a 9-bit literal whose value is the main memory address to read from.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
000000 001i 1111 ddddddddd sssssssss	Result = 0	---	Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination [1]	Z [2]	C
$----_----; -	$----_----; -	-	-	wz wc	31:8 = 0, 7:0 = byte value	0	0

[1] Destination Out is the zero-extended byte value read from main memory and is always generated since including an NR effect would turn RDBYTE into a WRBYTE instruction.
[2] The Z flag is cleared (0) unless Destination Out equals 0.

Explanation

RDBYTE syncs to the Hub, reads the byte of main memory at *Address*, zero-extends it, and stores it into the *Value* register.

IMPROVED

If the WZ effect is specified, the Z flag will be set (1) if the value read from main memory is zero. The NR effect can not be used with RDBYTE as that would change it to a WRBYTE instruction.

RDBYTE is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

RDLONG

Instruction: Read long of main memory.

RDLONG *Value*, ⟨#⟩ *Address*

Result: Long is stored in *Value*.

- ***Value*** (d-field) is the register to store the long value into.
- ***Address*** (s-field) is a register or a 9-bit literal whose value is the main memory address to read from.

Opcode Table:

–INSTR– ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
000010 001i	1111	ddddddddd	sssssssss	Result = 0	---	Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination [1]	Z [2]	C
$----_----; -	$----_----; -	-	-	wz wc	long value	0	0

[1] Destination Out is always generated since including an NR effect would turn RDLONG into a WRLONG instruction.
[2] The Z flag is cleared (0) unless Destination Out equals 0.

IMPROVED

Explanation

RDLONG syncs to the Hub, reads the long of main memory at *Address*, and stores it into the *Value* register. *Address* can point to any byte within the desired long; the address' lower two bits will be cleared to zero resulting in an address pointing to a long boundary.

If the WZ effect is specified, the Z flag will be set (1) if the value read from main memory is zero. The NR effect can not be used with RDLONG as that would change it to a WRLONG instruction.

RDLONG is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

RDWORD

Instruction: Read word of main memory.

RDWORD *Value*, ⟨#⟩ *Address*

Result: Zero-extended word is stored in *Value*.

- ***Value*** (d-field) is the register to store the zero-extended word value into.
- ***Address*** (s-field) is a register or a 9-bit literal whose value is the main memory address to read from.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
000001 001i 1111	ddddddddd	sssssssss	Result = 0	---	Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	31:16 = 0, 15:0 = word value	0	0

[1] Destination Out is the zero-extended word value read from main memory and is always generated since including an NR effect would turn RDWORD into a WRWORD instruction.

[2] The Z flag is cleared (0) unless Destination Out equals 0.

IMPROVED

Explanation

RDWORD syncs to the Hub, reads the word of main memory at *Address*, zero-extends it, and stores it into the *Value* register *Address* can point to any byte within the desired word; the address' lower bit will be cleared to zero resulting in an address pointing to a word boundary.

If the WZ effect is specified, the Z flag will be set (1) if the value read from main memory is zero. The NR effect can not be used with RDWORD as that would change it to a WRWORD instruction.

RDWORD is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

Registers

IMPROVED

Each cog contains 16 special purpose registers for accessing I/O pins, the built-in counters and video generator, and the parameter passed at the moment the cog is launched. All of these registers are explained in the Spin Language Reference and most of the information applies to both Spin and Propeller Assembly. The following table illustrates the 16 special purpose registers, indicates where to find information, and states what details, if any, do not apply to Propeller Assembly. Each of these registers can be accessed just like any other register in the destination or source fields of instructions, except for those that are designated with a footnote of 1 or 2. These special registers can only be read via the Source field of an instruction; (1) they are not writable, or (2) they can not be used in the Destination field for a read-modify-write operation.

IMPROVED

Table 3-5: Registers

Register(s)	Description
DIRA, DIRB [3]	Direction Registers for 32-bit port A and 32-bit port B, see pages 289 and 104. The optional "[*Pin(s)*]" parameter does not apply to Propeller Assembly; all bits of the entire register are read/written at once, unless using the MUXx instructions.
INA [1], INB [1,3]	Input Registers for 32-bit port A and 32-bit port B (Read-Only), see pages 297 and 118. The optional "[*Pin(s)*]" parameter does not apply to Propeller Assembly; all bits of the entire register are read at once.
OUTA, OUTB [3]	Output Registers for 32-bit port A and 32-bit port B, see pages 330 and 175. The optional "[*Pin(s)*]" parameter does not apply to Propeller Assembly; all bits of the entire register are read/written at once, unless using the MUXx instructions.
CNT [1]	32-bit System Counter Register (Read-Only), see pages 282 and 73.
CTRA, CTRB	Counter A and Counter B Control Registers, see pages 288 and 95.
FRQA, FRQB	Counter A and Counter B Frequency Registers, see pages 293 and 111.
PHSA [2], PHSB [2]	Counter A and Counter B Phase-Locked Loop Registers, see pages 332 and 180.
VCFG	Video Configuration Register, see pages 366 and 213.
VSCL	Video Scale Register, see pages 367 and 216.
PAR [1]	Cog Boot Parameter Register (Read Only), see pages 331 and 178.

Note 1: For Propeller Assembly, only accessible as a source register (i.e., mov *dest*, *source*). See the Assembly language sections for PAR, page 331; CNT, page 282, and INA, INB, page 297.

Note 2: For Propeller Assembly, only readable as a source register (i.e., mov *dest, source*); read modify-write not possible as a destination register. See the Assembly language section for PHSA, PHSB on page 332.

Note 3: Reserved for future use.

RES

Directive: Reserve next long(s) for symbol.

⟨***Symbol***⟩ RES ⟨***Count***⟩

- ***Symbol*** is an optional name for the reserved long in Cog RAM.
- ***Count*** is the optional number of longs to reserve for *Symbol*. If not specified, RES reserves one long.

NEW

Explanation

The RES (reserve) directive effectively reserves one or more longs of Cog RAM by incrementing the compile-time assembly pointer by *Count*. Normally this is used to reserve memory for an assembly symbol that does not need initialization to any specific value. For example:

```
DAT
                ORG     0
AsmCode         mov     Time, cnt           'Get system counter
                add     Time, Delay         'Add delay
:Loop           waitcnt Time, Delay         'Wait for time window
                nop                         'Do something useful
                jmp     #:Loop              'Loop endlessly

Delay   long    6_000_000                   'Time window size
Time    RES     1                           'Time window workspace
```

The last line of the AsmCode example, above, reserves one long of Cog RAM for the symbol Time without defining an initial value. AsmCode uses Time as a long variable to wait for the start of a time window of 6 million clock cycles. When AsmCode is launched into a cog, it is loaded into Cog RAM as shown below

Symbol	Address	Instruction/Data
AsmCode	0	mov Time, cnt
	1	add Time, Delay
:Loop	2	waitcnt Time, Delay
	3	nop
	4	jmp #:Loop
Delay	5	6_000_000
Time	6	?

RES simply increments the compile-time assembly pointer that affects further symbol references (Cog RAM); it does not consume space in the object (Main RAM). This distinction is important in how it impacts the object code, initialized symbols, and affects run-time operation.

- Since it increments the compile-time assembly pointer only, the computed address of all symbols following a RES statement are affected accordingly.
- No space is actually consumed in the object/application (Main RAM). This is an advantage if defining buffers that should only exist in Cog RAM but not Main RAM.

NEW

Caution: Use RES Only After Instructions and Data

It's important to use RES only after the final instructions and data in a logical assembly program. Placing RES in a prior location could have unintended results as explained below.

Remember that assembly instructions and data are placed in the application memory image in the exact order entered in source, independent of the assembly pointer. This is because launched assembly code must be loaded in order, starting with the designated routine label. However, since RES does not generate any data (or code), it has absolutely no effect on the memory image of the application; RES only adjusts the value of the assembly pointer.

By nature of the RES directive, any data or code appearing after a RES statement will be placed immediately after the last non-RES entity, in the same logical space as the RES entities themselves. Consider the following example where the code, from above, has the order of the Time and Delay declarations reversed.

```
DAT
                ORG     0
AsmCode         mov     Time, cnt         'Get system counter
                add     Time, Delay       'Add delay
```

```
:Loop           waitcnt Time, Delay       'Wait for time window
                nop                       'Do something useful
                jmp     #:Loop            'Loop endlessly

Time    RES     1                         'Time window workspace
Delay   long    6_000_000                 'Time window size
```

This example would be launched into a cog as follows:

Symbol	Address	Instruction/Data
AsmCode	0	mov Time, cnt
	1	add Time, Delay
:Loop	2	waitcnt Time, Delay
	3	nop
	4	jmp #:Loop
Time	5	6_000_000
Delay	6	?

Notice how Time and Delay are reversed with respect to the previous example but their data is not? Here's what happened:

- First, the assembler placed everything in the object's memory image exactly as it did before up to and including the JMP instruction.
- The assembler reached the Time symbol, which is declared with a RES directive, so it equated Time to address 5 (the current assembly pointer value) and then incremented the assembly pointer by 1. No data was placed in the application memory image due to this step.
- The assembler reached the Delay symbol, which is declared as a LONG of data, so it equated Delay to address 6 (the current assembly pointer value), incremented the assembly pointer by 1, then placed the data, 6_000_000, into the next available location in the memory image right after the JMP instruction which happens to be where Time is logically pointing.

The effect when launched into a cog is that the Time symbol occupies the same Cog RAM space that Delay's initial value does, and Delay exists in the next register that contains unknown data. The code will fail to run as intended.

For this reason, it is best to place RES statements after the last instruction and after the last defined data that your assembly code relies upon, as shown in the first example.

RET

Instruction: Return to previously recorded address.

RET

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
010111 0001 1111 --------- ---------	Result = 0	---	Not Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source	Z	C	Effects	Destination[2]	Z	C[3]
s----_----; -	s----_----; -	-	-	wr wz wc	31:9 unchanged, 8:0 = PC+1	0	1

[1] Destination is normally ignored for RET usage, however if the WR effect is given, the RET instruction becomes a CALL instruction and Destination's s-field (lowest 9 bits) are overwritten with the return address (PC+1).
[2] Destination is not written unless the WR effect is given.
[3] The C flag is set (1) unless PC+1 equals 0; very unlikely since it would require the RET to be executed from the top of cog RAM ($1FF; special purpose register VSCL).

IMPROVED

Explanation

RET returns execution to a previously recorded address by setting the Program Counter (PC) to that address. The RET instruction is meant to be used along with a label in the form "label_ret" and a CALL instruction that targets RET's routine, denoted by "label." See CALL on page 268 for more information.

RET is a subset of the JMP instruction but with the i-field set and the s-field unspecified. It is also closely related to the CALL and JMPRET commands; in fact, they are all the same opcode but with different r-field and i-field values and varying assembler-driven and user-driven d-field and s-field values.

REV

Instruction: Reverse LSBs of value and zero-extend.

REV *Value*, ⟨#⟩ *Bits*

Result: *Value* has lower 32 - *Bits* of its LSBs reversed and upper bits cleared.

- ***Value*** (d-field) is the register containing the value whose bits are reversed.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value subtracted from 32, (32 - *Bits*), is the number of *Value's* LSBs to reverse. The upper *Bits* MSBs of *Value* are cleared.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
001111 001i 1111 ddddddddd sssssssss	Result = 0	D[0]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$8421_DECA; -2,078,155,062	$0000_001F; 31	–	–	wz wc	$0000_0000; 0	1	0
$8421_DECA; -2,078,155,062	$0000_001C; 28	–	–	wz wc	$0000_0005; 5	0	0
$8421_DECA; -2,078,155,062	$0000_0018; 24	–	–	wz wc	$0000_0053; 83	0	0
$8421_DECA; -2,078,155,062	$0000_0010; 16	–	–	wz wc	$0000_537B; 21,371	0	0
$8421_DECA; -2,078,155,062	$0000_0000; 0	–	–	wz wc	$537B_8421; 1,400,603,681	0	0
$4321_8765; 1,126,270,821	$0000_001C; 28	–	–	wz wc	$0000_000A; 10	0	1
$4321_8765; 1,126,270,821	$0000_0018; 24	–	–	wz wc	$0000_00A6; 166	0	1
$4321_8765; 1,126,270,821	$0000_0010; 16	–	–	wz wc	$0000_A6E1; 42,721	0	1
$4321_8765; 1,126,270,821	$0000_0000; 0	–	–	wz wc	$A6E1_84C2; -1,495,169,854	0	1

Explanation

REV (Reverse) reverses the lower (32 - *Bits*) of *Value's* LSB and clears the upper *Bits* of *Value's* MSBs.

If the WZ effect is specified, the Z flag is set (1) if the resulting *Value* equals zero. If the WC effect is specified, the C flag is set equal to *Value's* original bit 0. The result is written to *Value* unless the NR effect is specified.

ROL

Instruction: Rotate value left by specified number of bits.

ROL *Value*, ⟨#⟩ *Bits*

Result: *Value* is rotated left by *Bits*.

- ***Value*** (d-field) is the register to rotate left.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits to rotate left.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
001001 001i 1111	dddddddd	ssssssss	Result = 0	D[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$8765_4321; -2,023,406,815	$0000_0004; 4	–	–	wz wc	$7654_3218; 1,985,229,336	0	1
$7654_3218; 1,985,229,336	$0000_000C; 12	–	–	wz wc	$4321_8765; 1,126,270,821	0	0
$4321_8765; 1,126,270,821	$0000_0010; 16	–	–	wz wc	$8765_4321; -2,023,406,815	0	0

Explanation

ROL (Rotate Left) rotates *Value* left, *Bits* times. The MSBs rotated out of *Value* are rotated into its LSBs.

If the WZ effect is specified, the Z flag is set (1) if the resulting *Value* equals zero. If the WC effect is specified, at the end of the operation, the C flag is set equal to *Value's* original bit 31. The result is written to *Value* unless the NR effect is specified.

ROR

Instruction: Rotate value right by specified number of bits.

ROR *Value*, ⟨#⟩ *Bits*

Result: *Value* is rotated right by *Bits*.

- ***Value*** (d-field) is the register to rotate right.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits to rotate right.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
001000 001i 1111	ddddddddd	sssssssss	Result = 0	D[0]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$1234_5678; 305,419,896	$0000_0004; 4	–	–	wz wc	$8123_4567; -2,128,394,905	0	0
$8123_4567; -2,128,394,905	$0000_000C; 12	–	–	wz wc	$5678_1234; 1,450,709,556	0	1
$5678_1234; 1,450,709,556	$0000_0010; 16	–	–	wz wc	$1234_5678; 305,419,896	0	0

Explanation

ROR (Rotate Right) rotates *Value* right, *Bits* times. The LSBs rotated out of *Value* are rotated into its MSBs.

If the WZ effect is specified, the Z flag is set (1) if the resulting *Value* equals zero. If the WC effect is specified, at the end of the operation, the C flag is set equal to *Value's* original bit 0. The result is written to *Value* unless the NR effect is specified.

SAR

Instruction: Shift value arithmetically right by specified number of bits.

SAR *Value*, ⟨#⟩ *Bits*

Result: *Value* is shifted arithmetically right by *Bits*.

- ***Value*** (d-field) is the register to shift arithmetically right.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits to shift arithmetically right.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
001110 001i 1111	ddddddddd	sssssssss	Result = 0	D[0]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$FFFF_FF9C; -100	$0000_0001; 1	–	–	wz wc	$FFFF_FFCE; -50	0	0
$FFFF_FF9C; -100	$0000_0002; 2	–	–	wz wc	$FFFF_FFE7; -25	0	0
$FFFF_FF9C; -100	$0000_0003; 3	–	–	wz wc	$FFFF_FFF3; -13	0	0
$FFFF_FFF3; -13	$0000_0001; 1	–	–	wz wc	$FFFF_FFF9; -7	0	1
$FFFF_FFF9; -7	$0000_0001; 1	–	–	wz wc	$FFFF_FFFC; -4	0	1
$FFFF_FFFC; -4	$0000_0001; 1	–	–	wz wc	$FFFF_FFFE; -2	0	0
$0000_0006; 6	$0000_0001; 1	–	–	wz wc	$0000_0003; 3	0	0
$0000_0006; 6	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	0
$0000_0006; 6	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	0

Explanation

IMPROVED

SAR (Shift Arithmetic Right) shifts *Value* right by *Bits* places, extending the MSB along the way. This has the effect of preserving the sign in a signed value, thus SAR is a quick divide-by-power-of-two for signed integer values.

If the WZ effect is specified, the Z flag is set (1) if the resulting *Value* equals zero. If the WC effect is specified, the C flag is set equal to *Value's* original bit 0. The result is written to *Value* unless the NR effect is specified.

SHL

Instruction: Shift value left by specified number of bits.

SHL *Value*, ⟨#⟩ *Bits*

Result: *Value* is shifted left by *Bits*.

- ***Value*** (d-field) is the register to shift left.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits to shift left.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
001011 001i 1111	dddddddd	ssssssss	Result = 0	D[31]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$8765_4321; -2,023,406,815	$0000_0004; 4	–	–	wz wc	$7654_3210; 1,985,229,328	0	1
$7654_3210; 1,985,229,328	$0000_000C; 12	–	–	wz wc	$4321_0000; 1,126,236,160	0	0
$4321_0000; 1,126,236,160	$0000_0010; 16	–	–	wz wc	$0000_0000; 0	1	0

Explanation

IMPROVED

SHL (Shift Left) shifts *Value* left by *Bits* places and sets the new LSBs to 0.

If the WZ effect is specified, the Z flag is set (1) if the resulting *Value* equals zero. If the WC effect is specified, the C flag is set equal to *Value's* original bit 31. The result is written to *Value* unless the NR effect is specified.

SHR

Instruction: Shift value right by specified number of bits.

SHR *Value*, ⟨#⟩ *Bits*

Result: *Value* is shifted right by *Bits*.

- ***Value*** (d-field) is the register to shift right.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits to shift right.

Opcode Table:

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
001010	001i	1111	ddddddddd	sssssssss	Result = 0	D[0]	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$1234_5678; 305,419,896	$0000_0004; 4	–	–	wz wc	$0123_4567; 19,088,743	0	0
$0123_4567; 19,088,743	$0000_000C; 12	–	–	wz wc	$0000_1234; 4,660	0	1
$0000_1234; 4,660	$0000_0010; 16	–	–	wz wc	$0000_0000; 0	1	0

Explanation

IMPROVED

SHR (Shift Right) shifts *Value* right by *Bits* places and sets the new MSBs to 0.

If the WZ effect is specified, the Z flag is set (1) if the resulting *Value* equals zero. If the WC effect is specified, the C flag is set equal to *Value's* original bit 0. The result is written to *Value* unless the NR effect is specified.

SUB

Instruction: Subtract two unsigned values.

SUB *Value1*, {#} *Value2*

Result: Difference of unsigned *Value1* and unsigned *Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to subtract *Value2* from, and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is subtracted from *Value1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
100001 001i 1111 dddddddd ssssssss	D - S = 0	Unsigned Borrow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source	Z	C	Effects	Destination	Z	C
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0002; 2	–	–	wz wc	$0000_0000; 0	1	0
$0000_0002; 2	$0000_0003; 3	–	–	wz wc	$FFFF_FFFF; 4,294,967,295	0	1

[1] Both Source and Destination are treated as unsigned values.

Explanation

SUB subtracts the unsigned *Value2* from the unsigned *Value1* and stores the result into the *Value1* register.

If the WZ effect is specified, the Z flag is set (1) if *Value1* − *Value2* equals zero. If the WC effect is specified, the C flag is set (1) if the subtraction resulted in an unsigned borrow (32-bit overflow). The result is written to *Value1* unless the NR effect is specified.

NEW

To subtract unsigned, multi-long values, use SUB followed by SUBX. See SUBX on page 354 for more information.

SUBABS

Instruction: *Subtract* an absolute value from another value.

SUBABS *Value*, ⟨#⟩ *SValue*

Result: Difference of *Value* and absolute of signed *SValue* is stored in *Value*.

- ***Value*** (d-field) is the register containing the value to subtract the absolute of *SValue* from, and is the destination in which to write the result.
- ***SValue*** (s-field) is a register or a 9-bit literal whose absolute value is subtracted from *Value*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
100011 001i 1111 ddddddddd sssssssss	D - |S| = 0	Unsigned Borrow[1]	Written	4

[1] If S is negative, C Result is the inverse of unsigned *carry* (for D + S).

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source	Z	C	Effects	Destination	Z	C
$0000_0003; 3	$FFFF_FFFC; -4	–	–	wz wc	$FFFF_FFFF; 4,294,967,295	0	0
$0000_0003; 3	$FFFF_FFFD; -3	–	–	wz wc	$0000_0000; 0	1	1
$0000_0003; 3	$FFFF_FFFE; -2	–	–	wz wc	$0000_0001; 1	0	1
$0000_0003; 3	$FFFF_FFFF; -1	–	–	wz wc	$0000_0002; 2	0	1
$0000_0003; 3	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0004; 4	–	–	wz wc	$FFFF_FFFF; 4,294,967,295	0	1

[1] Destination is treated as an unsigned value.

Explanation

SUBABS subtracts the absolute of *SValue* from *Value* and stores the result into the *Value* register.

If the WZ effect is specified, the Z flag is set (1) if *Value* − |*SValue*| equals zero. If the WC effect is specified, the C flag is set (1) if the subtraction resulted in an unsigned borrow (32-bit overflow). The result is written to *Value* unless the NR effect is specified.

SUBS

Instruction: Subtract two signed values.

SUBS *SValue1*, ⟨#⟩ *SValue2*

Result: Difference of signed *SValue1* and signed *SValue2* is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to subtract *SValue2* from, and is the destination in which to write the result.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is subtracted from *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
110101 001i 1111	dddddddd	ssssssss	D - S = 0	Signed Overflow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFE; -2	–	–	wz wc	$0000_0001; 1	0	0
$8000_0001; -2,147,483,647	$0000_0001; 1	–	–	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$0000_0002; 2	–	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFE; 2,147,483,646	$FFFF_FFFF; -1	–	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$7FFF_FFFE; 2,147,483,646	$FFFF_FFFE; -2	–	–	wz wc	$8000_0000; -2,147,483,648	0	1

Explanation

SUBS subtracts the signed *SValue2* from the signed *SValue1* and stores the result into the *SValue1* register.

If the WZ effect is specified, the Z flag is set (1) if *SValue1* − *SValue2* equals zero. If the WC effect is specified, the C flag is set (1) if the subtraction resulted in a signed overflow. The result is written to *SValue1* unless the NR effect is specified.

NEW

To subtract signed, multi-long values, use SUB, possibly SUBX, and finally SUBSX. See SUBSX on page 352 for more information.

SUBSX

Instruction: Subtract signed value plus C from another signed value.

SUBSX *SValue1*, ⟨#⟩ *SValue2*

Result: Difference of signed *SValue1*, and signed *SValue2* plus C flag, is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to subtract *SValue2* plus C from, and is the destination in which to write the result.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value plus C is subtracted from *SValue1*.

Opcode Table:

IMPROVED

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
110111	001i	1111	ddddddddd	sssssssss	Z & (D–(S+C) = 0)	Signed Overflow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0001; 1	0	0	wz wc	$0000_0000; 0	0	0
$0000_0001; 1	$0000_0001; 1	1	0	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	x	1	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	0	0	wz wc	$0000_0000; 0	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	1	0	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	x	1	wz wc	$FFFF_FFFF; -1	0	0
$8000_0001; -2,147,483,647	$0000_0001; 1	x	0	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$0000_0001; 1	x	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	x	0	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	x	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	0

NEW

Explanation

SUBSX (Subtract Signed, Extended) subtracts the signed value of *SValue2* plus C from *SValue1*, and stores the result into the *SValue1* register. The SUBSX instruction is used to perform signed multi-long subtraction; 64-bit subtractions, for example.

In a signed multi-long operation, the first instruction is unsigned (ex: SUB), any middle instructions are unsigned, extended (ex: SUBX), and the last instruction is signed, extended

(ex: SUBSX). Make sure to use the WC, and optionally WZ, effect on the leading SUB and SUBX instructions.

For example, a signed double-long (64-bit) subtraction may look like this:

```
sub     XLow,  YLow   wc wz   'Subtract low longs; save C and Z
subsx   XHigh, YHigh          'Subtract high longs
```

After executing the above, the double-long (64-bit) result is in the long registers *XHigh:XLow*. If *XHigh:XLow* started out as $0000_0000:0000_0001 (1) and *YHigh:YLow* was $0000_0000:0000_0002 (2) the result in *XHigh:XLow* would be $FFFF_FFFF:FFFF_FFFF (-1). This is demonstrated below.

```
                        Hexadecimal              Decimal
                       (high)     (low)
   (XHigh:XLow)      $0000_0000:0000_0001                  1
 - (YHigh:YLow)    - $0000_0000:0000_0002      -           2
                   ----------------------      ---------------
                   = $FFFF_FFFF:FFFF_FFFF      =          -1
```

A signed triple-long (96-bit) subtraction would look similar but with a SUBX instruction inserted between the SUB and SUBSX instructions:

```
sub     XLow,  YLow   wc wz   'Subtract low longs; save C and Z
subx    XMid,  YMid   wc wz   'Subtract middle longs; save C and Z
subsx   XHigh, YHigh          'Subtract high longs
```

Of course, it may be necessary to specify the WC and WZ effects on the final instruction, SUBSX, in order to watch for a result of zero or signed overflow condition. Note that during this multi-step operation the Z flag always indicates if the result is turning out to be zero, but the C flag indicates unsigned borrows until the final instruction, SUBSX, in which it indicates signed overflow.

For SUBSX, if the WZ effect is specified, the Z flag is set (1) if Z was previously set and *SValue1* − (*SValue2* + C) equals zero (use WC and WZ on preceding SUB and SUBX instructions). If the WC effect is specified, the C flag is set (1) if the subtraction resulted in a signed overflow. The result is written to *SValue1* unless the NR effect is specified).

SUBX

Instruction: Subtract unsigned value plus C from another unsigned value.

SUBX *Value1*, ⟨#⟩ *Value2*

Result: Difference of unsigned *Value1*, and unsigned *Value2* plus C flag, is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to subtract *Value2* plus C from, and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value plus C is subtracted from *Value1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
110011 001i 1111 ddddddddd sssssssss	Z & (D–(S+C) = 0)	Unsigned Borrow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination[1]	Source[1]	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0001; 1	0	0	wz wc	$0000_0000; 0	0	0
$0000_0001; 1	$0000_0001; 1	1	0	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	×	1	wz wc	$FFFF_FFFF; 4,294,967,295	0	1

[1] Both Source and Destination are treated as unsigned values.

NEW

Explanation

SUBX (Subtract Extended) subtracts the unsigned value of *Value2* plus C from the unsigned *Value1* and stores the result into the *Value1* register. The SUBX instruction is used to perform multi long subtraction; 64-bit subtractions, for example.

In a multi-long operation, the first instruction is unsigned (ex: SUB), any middle instructions are unsigned, extended (ex: SUBX), and the last instruction is unsigned, extended (SUBX) or signed, extended (SUBSX) depending on the nature of the original multi-long values. We'll discuss unsigned multi-long values here; see SUBSX on page 352 for examples with signed, multi-long values. Make sure to use the WC, and optionally WZ, effect on the leading SUB and SUBX instructions.

For example, an unsigned double-long (64-bit) subtraction may look like this:

```
sub     XLow,  YLow   wc wz   'Subtract low longs; save C and Z
subx    XHigh, YHigh          'Subtract high longs
```

After executing the above, the double-long (64-bit) result is in the long registers *XHigh:XLow*. If *XHigh:XLow* started out as $0000_0001:0000_0000 (4,294,967,296) and *YHigh:YLow* was $0000_0000:0000_0001 (1) the result in *XHigh:XLow* would be $0000_0000:FFFF_FFFF (4,294,967,295). This is demonstrated below.

```
                          Hexadecimal              Decimal
                        (high)     (low)
   (XHigh:XLow)       $0000_0001:0000_0000       4,294,967,296
 - (YHigh:YLow)     - $0000_0000:0000_0001     -             1
                    ----------------------     ---------------
                    = $0000_0000:FFFF_FFFF     = 4,294,967,295
```

Of course, it may be necessary to specify the **WC** and **WZ** effects on the final instruction, **SUBX**, in order to watch for a result of zero or an unsigned borrow condition.

For **SUBX**, if the **WZ** effect is specified, the Z flag is set (1) if Z was previously set and *Value1* - (*Value2* + C) equals zero (use **WC** and **WZ** on preceding **SUB** and **SUBX** instructions). If the **WC** effect is specified, the C flag is set (1) if the subtraction resulted in an unsigned borrow (32-bit overflow). The result is written to *Value1* unless the **NR** effect is specified.

SUMC

Instruction: Sum a signed value with another whose sign is inverted depending on C.

SUMC *SValue1*, ⟨#⟩ *SValue2*

Result: Sum of signed *SValue1* and ±*SValue2* is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to sum with either –*SValue2* or *SValue2*, and is the destination in which to write the result.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is sign-affected by C and summed into *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
100100 001i 1111 ddddddddd sssssssss	D ± S = 0	Signed Overflow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0001; 1	–	0	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	1	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFF; -1	–	0	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	0	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	1	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0001; 1	–	0	wz wc	$0000_0000; 0	1	0
$8000_0000; -2,147,483,648	$0000_0001; 1	–	0	wz wc	$8000_0001; -2,147,483,647	0	0
$8000_0000; -2,147,483,648	$0000_0001; 1	–	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0000; -2,147,483,648	$FFFF_FFFF; -1	–	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	–	0	wz wc	$7FFF_FFFE; 2,147,483,646	0	0
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	–	1	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$0000_0001; 1	–	0	wz wc	$8000_0000; -2,147,483,648	0	1

Explanation

SUMC (Sum with C-affected sign) adds the signed value of *SValue1* to –*SValue2* (if C = 1) or to *SValue2* (if C = 0) and stores the result into the *SValue1* register.

If the WZ effect is specified, the Z flag is set (1) if *SValue1* ± *SValue2* equals zero. If the WC effect is specified, the C flag is set (1) if the summation resulted in a signed overflow. The result is written to *SValue1* unless the NR effect is specified.

SUMNC

Instruction: Sum a signed value with another whose sign is inverted depending on !C.

SUMNC *SValue1*, ⟨#⟩ *SValue2*

Result: Sum of signed *SValue1* and ±*SValue2* is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to sum with either *SValue2* or -*SValue2*, and is the destination in which to write the result.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is sign-affected by !C and summed into *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
100101 001i 1111 ddddddddd sssssssss	D ± S = 0	Signed Overflow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0001; 1	-	0	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	-	1	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$FFFF_FFFF; -1	-	1	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	-	0	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	-	1	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$0000_0001; 1	-	1	wz wc	$0000_0000; 0	1	0
$8000_0000; -2,147,483,648	$0000_0001; 1	-	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0000; -2,147,483,648	$0000_0001; 1	-	1	wz wc	$8000_0001; -2,147,483,647	0	0
$8000_0000; -2,147,483,648	$FFFF_FFFF; -1	-	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	-	0	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	-	1	wz wc	$7FFF_FFFE; 2,147,483,646	0	0
$7FFF_FFFF; 2,147,483,647	$0000_0001; 1	-	1	wz wc	$8000_0000; -2,147,483,648	0	1

Explanation

SUMNC (Sum with !C-affected sign) adds the signed value of *SValue1* to *SValue2* (if C = 1) or to –*SValue2* (if C = 0) and stores the result into the *SValue1* register.

If the WZ effect is specified, the Z flag is set (1) if *SValue1* ± *SValue2* equals zero. If the WC effect is specified, the C flag is set (1) if the summation resulted in a signed overflow. The result is written to *SValue1* unless the NR effect is specified.

SUMNZ

Instruction: Sum a signed value with another whose sign is inverted depending on !Z.

SUMNZ *SValue1*, ⟨#⟩ *SValue2*

Result: Sum of signed *SValue1* and ±*SValue2* is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to sum with either *SValue2* or -*SValue2*, and is the destination in which to write the result.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is sign-affected by !Z and summed into *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
100111 001i 1111 ddddddddd sssssssss	D ± S = 0	Signed Overflow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0001; 1	0	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	1	–	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$FFFF_FFFF; -1	1	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	0	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	1	–	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$0000_0001; 1	1	–	wz wc	$0000_0000; 0	1	0
$8000_0000; -2,147,483,648	$0000_0001; 1	0	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0000; -2,147,483,648	$0000_0001; 1	1	–	wz wc	$8000_0001; -2,147,483,647	0	0
$8000_0000; -2,147,483,648	$FFFF_FFFF; -1	1	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	0	–	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	1	–	wz wc	$7FFF_FFFE; 2,147,483,646	0	0
$7FFF_FFFF; 2,147,483,647	$0000_0001; 1	1	–	wz wc	$8000_0000; -2,147,483,648	0	1

Explanation

SUMNZ (Sum with !Z-affected sign) adds the signed value of *SValue1* to *SValue2* (if Z = 1) or to –*SValue2* (if Z = 0) and stores the result into the *SValue1* register.

If the WZ effect is specified, the Z flag is set (1) if *SValue1* ± *SValue2* equals zero. If the WC effect is specified, the C flag is set (1) if the summation resulted in a signed overflow. The result is written to *SValue1* unless the NR effect is specified.

SUMZ

Instruction: Sum a signed value with another whose sign is inverted depending on Z.

SUMZ *SValue1*, ⟨#⟩ *SValue2*

Result: Sum of signed *SValue1* and ±*SValue2* is stored in *SValue1*.

- ***SValue1*** (d-field) is the register containing the value to sum with either –*SValue2* or *SValue2*, and is the destination in which to write the result.
- ***SValue2*** (s-field) is a register or a 9-bit literal whose value is sign-affected by Z and summed into *SValue1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
100110 001i 1111 ddddddddd sssssssss	D ± S = 0	Signed Overflow	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0001; 1	$0000_0001; 1	0	–	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	1	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFF; -1	0	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	0	–	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	1	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0001; 1	0	–	wz wc	$0000_0000; 0	1	0
$8000_0000; -2,147,483,648	$0000_0001; 1	0	–	wz wc	$8000_0001; -2,147,483,647	0	0
$8000_0000; -2,147,483,648	$0000_0001; 1	1	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0000; -2,147,483,648	$FFFF_FFFF; -1	0	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	0	–	wz wc	$7FFF_FFFE; 2,147,483,646	0	0
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	1	–	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$0000_0001; 1	0	–	wz wc	$8000_0000; -2,147,483,648	0	1

Explanation

SUMZ (Sum with Z-affected sign) adds the signed value of *SValue1* to –*SValue2* (if Z = 1) or to *SValue2* (if Z = 0) and stores the result into the *SValue1* register.

If the WZ effect is specified, the Z flag is set (1) if *SValue1* ± *SValue2* equals zero. If the WC effect is specified, the C flag is set (1) if the summation resulted in a signed overflow. The result is written to *SValue1* unless the NR effect is specified.

NEW

Symbols

The symbols in Table 3-6 below serve one or more special purposes in Propeller Assembly code. For Spin symbols, see Symbols on page 207. Each symbol's purpose is described briefly with references to other sections that describe it directly or use it in examples.

Table 3-6: Symbols

Symbol	Purpose(s)
%	Binary indicator: used to indicate that a value is being expressed in binary (base-2). See Value Representations on page 45.
%%	Quaternary indicator: used to indicate a value is being expressed in quaternary (base-4). See Value Representations on page 45.
$	1) Hexadecimal indicator: used to indicate a value is being expressed in hexadecimal (base-16). See Value Representations on page 45. 2) Assembly Here indicator: used to indicate current address in assembly instructions. See **JMP** on page 298.
''	String designator: used to begin and end a string of text characters. See Data blocks on page 99.
_	1) Delimiter: used as a group delimiter in constant values (where a comma ',' or period '.' may normally be used as a number group delimiter). See Value Representations on page 45. 2) Underscore: used as part of a symbol. See Symbol Rules on page 45.
#	Assembly Literal indicator: used to indicate an expression or symbol is a literal value rather than a register address. See Where Does an Instruction Get Its Data? on page 240.

Table 3-6: Symbols (continued)	
Symbol	**Purpose(s)**
:	Assembly local label indicator: appears immediately before a local label. See Global and Local Labels on page 242.
,	List delimiter: used to separate items in lists. See the **DAT** section's Declaring Data(Syntax 1) on page 100.
'	Code comment designator: used to enter single-line code comments (non-compiled text) for code viewing purposes. See "Using the Propeller Tool" in the software's Help file.
''	Document comment designator: used to enter single-line document comments (non-compiled text) for documentation viewing purposes.
{ }	In-line/multi-line code comment designators: used to enter multi-line code comments (non-compiled text) for code viewing purposes.
{{ }}	In-line/multi-line document comment designators: used to enter multi-line document comments (non-compiled text) for documentation viewing purposes. See "Using the Propeller Tool" in the software's Help file.

TEST

Instruction: Bitwise AND two values to affect flags only.

TEST *Value1*, ⟨#⟩ *Value2*

Result: Optionally, zero-result and parity of result is written to the Z and C flags.

- ***Value1*** (d-field) is the register containing the value to bitwise AND with *Value2*.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is bitwise ANDed with *Value1*.

Opcode Table:

IMPROVED

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
011000 000i 1111 ddddddddd sssssssss	D = 0	Parity of Result	Not Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C
$0000_000A; 10	$0000_0005; 5	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_000A; 10	$0000_0007; 7	–	–	wr wz wc	$0000_0002; 2	0	1
$0000_000A; 10	$0000_000F; 15	–	–	wr wz wc	$0000_000A; 10	0	0

[1] Destination is not written unless the WR effect is given. NOTE: the TEST instruction with a WR effect is an AND instruction.

Explanation

TEST is similar to AND except it doesn't write a result to *Value1*; it performs a bitwise AND of the values in *Value1* and *Value2* and optionally stores the zero-result and parity of the result in the Z and C flags.

If the WZ effect is specified, the Z flag is set (1) if *Value1* AND *Value2* equals zero. If the WC effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits.

NEW

TESTN

Instruction: Bitwise AND a value with the NOT of another to affect flags only.

TESTN *Value1*, ⟨#⟩ *Value2*

Result: Optionally, zero-result and parity of result is written to the Z and C flags.

- ***Value1*** (d-field) is the register containing the value to bitwise AND with !*Value2*.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is inverted (bitwise NOT) and bitwise ANDed with *Value1*.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
011001 000i 1111 ddddddddd sssssssss	D = 0	Parity of Result	Not Written	4

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C
$F731_125A; -147,778,982	$FFFF_FFFA; -6	-	-	wr wz wc	$0000_0000; 0	1	0
$F731_125A; -147,778,982	$FFFF_FFF8; -8	-	-	wr wz wc	$0000_0002; 2	0	1
$F731_125A; -147,778,982	$FFFF_FFF0; -16	-	-	wr wz wc	$0000_000A; 10	0	0

[1] Destination is not written unless the WR effect is given. NOTE: the TESTN instruction with a WR effect is an ANDN instruction.

Explanation

TESTN is similar to ANDN except it doesn't write a result to *Value1*; it performs a bitwise AND NOT of *Value2* into *Value1* and optionally stores the zero-result and parity of the result in the Z and C flags.

If the WZ effect is specified, the Z flag is set (1) if *Value1* AND NOT *Value2* equals zero. If the WC effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits.

TJNZ

Instruction: Test value and jump to address if not zero.

TJNZ ***Value***, ⟨#⟩ ***Address***

- ***Value*** (d-field) is the register to test.
- ***Address*** (s-field) is the register or a 9-bit literal whose value is the address to jump to when *Value* contains a non-zero number.

Opcode Table:

IMPROVED

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
111010	000i	1111	ddddddddd	sssssssss	D = 0	0	Not Written	4 or 8

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C
s0000_0000; 0	s----_----; -	-	-	wr wz wc	s0000_0000; 0	1	0
s0000_0001; 1	s----_----; -	-	-	wr wz wc	s0000_0001; 1	0	0

[1] Destination is not written unless the WR effect is given.

Explanation

TJNZ tests the *Value* register and jumps to *Address* if it contains a non-zero number.

When the WZ effect is specified, the Z flag is set (1) if the *Value* register contains zero.

TJNZ requires a different amount of clock cycles depending on whether or not it has to jump. If it must jump it takes 4 clock cycles, if no jump occurs it takes 8 clock cycles. Since loops utilizing TJNZ need to be fast, it is optimized in this way for speed.

TJZ

Instruction: Test value and jump to address if zero.

TJZ ***Value***, ⟨#⟩ ***Address***

- ***Value*** (d-field) is the register to test.
- ***Address*** (s-field) is the register or a 9-bit literal whose value is the address to jump to when *Value* contains zero.

Opcode Table:

IMPROVED

–INSTR– ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
111011 000i	1111	dddddddd	ssssssss	D = 0	0	Not Written	4 or 8

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C
$0000_0000; 0	$----_----; -	-	-	wr wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$----_----; -	-	-	wr wz wc	$0000_0001; 1	0	0

[1] Destination is not written unless the WR effect is given.

Explanation

TJZ tests the *Value* register and jumps to *Address* if it contains zero.

When the WZ effect is specified, the Z flag is set (1) if the *Value* register contains zero.

TJZ requires a different amount of clock cycles depending on whether or not it has to jump. If it must jump it takes 4 clock cycles, if no jump occurs it takes 8 clock cycles.

NEW

VCFG

Register: Video configuration register.

```
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction  VCFG, SrcOperand ⟨Effects⟩
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction  DestOperand, VCFG ⟨Effects⟩
```

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. `VCFG` may be used in either the assembly instruction's *DestOperand* or *SrcOperand* fields.
- ***SrcOperand*** is a constant expression used by *Instruction* to operate on, and optionally write to, the `VCFG` register in *DestOperand*.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, using the value of `VCFG` in *SrcOperand*.

Explanation

`VCFG` is one of two registers (`VCFG` and `VSCL`) that affect the behavior of a cog's Video Generator. Each cog has a video generator module that facilitates transmitting video image data at a constant rate. The `VCFG` register contains the configuration settings of the video generator.

The following code sets the Video Configuration register to enable video in composite mode 1 with four colors, baseband chroma (color) enabled, on pin group 1, lower 4 pins (which is pins P11:8).

```
                    mov       vcfg,  VidCfg

VidCfg     long     %0_10_1_0_1_000_00000000000_001_0_00001111
```

See Registers, page 338, and the Spin language `VCFG` section, page 213, for more information.

NEW

VSCL

Register: Video scale register

```
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction  VSCL, SrcOperand ⟨Effects⟩
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction  DestOperand, VSCL ⟨Effects⟩
```

- ***Label*** is an optional statement label. See Common Syntax Elements, page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements, page 250.
- ***Instruction*** is the desired assembly instruction. VSCL may be used in either the assembly instruction's *DestOperand* or *SrcOperand* fields.
- ***SrcOperand*** is a constant expression used by *Instruction* to operate on, and optionally write to, the VSCL register in *DestOperand*.
- ***DestOperand*** is a constant expression indicating the register that is operated on, and optionally written to, using the value of VSCL in *SrcOperand*.

Explanation

VSCL is one of two registers (VCFG and VSCL) that affect the behavior of a cog's Video Generator. Each cog has a video generator module that facilitates transmitting video image data at a constant rate. The VSCL register sets the rate at which video data is generated.

The following code sets the Video Scale register for 160 pixel clocks and 2,560 frame clocks (for a 16-pixel by 2-bit color frame). Of course, the actual rate at which pixels clock out depends on the frequency of PLLA in combination with this scale factor.

```
                mov     vcfg,  VscCfg

VscCfg    long  %000000000000_10100000_101000000000
```

See Registers, page 338, and the Spin language VSCL section, page 216, for more information.

WAITCNT

Instruction: Pause a cog's execution temporarily.

WAITCNT *Target*, ⟨#⟩ *Delta*

Result: *Target* + *Delta* is stored in *Target*.

- ***Target*** (d-field) is the register with the target value to compare against the System Counter (CNT). When the System Counter has reached *Target's* value, *Delta* is added to *Target* and execution continues at the next instruction.
- ***Delta*** (s-field) is the register or a 9-bit literal whose value is added to *Target's* value in preparation for the next WAITCNT instruction. This creates a synchronized delay window.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
111110 001i 1111	ddddddddd	sssssssss	Result = 0	Unsigned Carry	Written	5+

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; 4,294,967,295	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	1
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0

Explanation

WAITCNT, "Wait for System Counter," is one of four wait instructions (WAITCNT, WAITPEQ, WAITPNE, and WAITVID) used to pause execution of a cog until a condition is met. The WAITCNT instruction pauses the cog until the global System Counter equals the value in the *Target* register, then it adds *Delta* to *Target* and execution continues at the next instruction. The WAITCNT instruction behaves similar to Spin's WAITCNT command for Synchronized Delays; see WAITCNT on page 218.

If the WZ effect is specified, the Z flag will be set (1) if the sum of *Target* and *Delta* is zero. If the WC effect is specified, the C flag will be set (1) if the sum of *Target* and *Delta* resulted in a 32-bit carry (overflow). The result will be written to *Target* unless the NR effect is specified.

WAITPEQ

Instruction: Pause a cog's execution until I/O pin(s) match designated state(s).

WAITPEQ ***State***, ⟨#⟩ ***Mask***

- ***State*** (d-field) is the register with the target state(s) to compare against INx ANDed with *Mask*.
- ***Mask*** (s-field) is the register or a 9-bit literal whose value is bitwise ANDed with INx before the comparison with *State*.

Opcode Table:

IMPROVED

–INSTR– ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
111100 000i	1111	ddddddddd	sssssssss	---	---	Not Written	5+

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination [1]	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	–	–	wr wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0001; 1	–	–	wr wz wc	$0000_0002; 2	0	0
$0000_0000; 0	$0000_0002; 2	–	–	wr wz wc	$0000_0002; 2	0	0
$0000_0002; 2	$0000_0002; 2	–	–	wr wz wc	$0000_0004; 4	0	0

[1] Destination is not written unless the WR effect is given.

Explanation

WAITPEQ, "Wait for Pin(s) to Equal," is one of four wait instructions (WAITCNT, WAITPEQ, WAITPNE, and WAITVID) used to pause execution of a cog until a condition is met. The WAITPEQ instruction pauses the cog until the result of INx ANDed with *Mask* equals the value in the *State* register. INx is either INA or INB depending on the value of the C flag upon execution; INA if C = 0, INB if C = 1 (the P8X32A is an exception to this rule; it always tests INA).

The WAITPEQ instruction behaves similar to Spin's WAITPEQ command; see WAITPEQ on page 222.

WAITPNE

Instruction: Pause a cog's execution until I/O pin(s) do not match designated state(s).

WAITPNE ***State***, ⟨#⟩ ***Mask***

- ***State*** (d-field) is the register with the target state(s) to compare against INx ANDed with *Mask*.
- ***Mask*** (s-field) is the register or a 9-bit literal whose value is bitwise ANDed with INx before the comparison with *State*.

Opcode Table:

IMPROVED

–INSTR– ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
111101 000i	1111	ddddddddd	sssssssss	---	---	Not Written	5+

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C
%0000_0000; 0	%0000_0000; 0	–	–	wr wz wc	%0000_0000; 0	1	1
%0000_0000; 0	%0000_0001; 1	–	–	wr wz wc	%0000_0002; 2	0	0
%0000_0001; 1	%0000_0001; 1	–	–	wr wz wc	%0000_0003; 3	0	0
%0000_0000; 0	%0000_0002; 2	–	–	wr wz wc	%0000_0003; 3	0	0
%0000_0002; 2	%0000_0002; 2	–	–	wr wz wc	%0000_0002; 2	0	0

[1] Destination is not written unless the WR effect is given.

Explanation

WAITPNE, "Wait for Pin(s) to Not Equal," is one of four wait instructions (WAITCNT, WAITPEQ, WAITPNE, and WAITVID) used to pause execution of a cog until a condition is met. The WAITPNE instruction pauses the cog until the result of INx ANDed with *Mask* does not match the value in the *State* register. INx is either INA or INB depending on the value of the C flag upon execution; INA if C = 0, INB if C = 1 (the P8X32A is an exception to this rule; it always tests INA). The WAITPNE instruction behaves similar to Spin's WAITPNE command; see WAITPNE on page 224.

WAITVID

Instruction: Pause a cog's execution until its Video Generator is available to take pixel data.

WAITVID ***Colors***, ⟨#⟩ ***Pixels***

- ***Colors*** (d-field) is the register with four byte-sized color values, each describing the four possible colors of the pixel patterns in *Pixels*.
- ***Pixels*** (s-field) is the register or a 9-bit literal whose value is the next 16-pixel by 2-bit (or 32-pixel by 1-bit) pixel pattern to display.

IMPROVED

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
111111 000i 1111 ddddddddd sssssssss	D + S = 0	Unsigned overflow	Not Written	5+

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z	C
$0000_0002; 2	$FFFF_FFFD; -3	–	–	wr wz wc	$FFFF_FFFF; -1	0	0
$0000_0002; 2	$FFFF_FFFE; -2	–	–	wr wz wc	$0000_0000; 0	1	1
$0000_0002; 2	$FFFF_FFFF; -1	–	–	wr wz wc	$0000_0001; 1	0	1
$0000_0002; 2	$0000_0000; 0	–	–	wr wz wc	$0000_0002; 2	0	0

[1] Destination is not written unless the WR effect is given.

Explanation

WAITVID, "Wait for Video Generator," is one of four wait instructions (WAITCNT, WAITPEQ, WAITPNE, and WAITVID) used to pause execution of a cog until a condition is met. The WAITVID instruction pauses the cog until its Video Generator hardware is ready for the next pixel data, then the Video Generator accepts that data (*Colors* and *Pixels*) and the cog continues execution with the next instruction. The WAITVID instruction behaves similar to Spin's WAITVID command; see WAITVID on page 225.

If the WZ effect is specified, the Z flag will be set (1) if the *Colors* and *Pixels* are equal.

Make sure to start the cog's Video Generator module and Counter A before executing the WAITVID command or it will wait forever. See VCFG on page 213, VSCL on page 216, and CTRA, CTRB on page 95.

NEW

WC

Effect: Cause assembly instruction to modify the C flag.

⟨Label⟩ ⟨Condition⟩ Instruction Operands WC

Result: C flag is updated with status from the *Instruction*'s execution.

- ***Label*** is an optional statement label. See Common Syntax Elements on page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements on page 250.
- ***Instruction*** is the desired assembly instruction.
- ***Operands*** is zero, one, or two operands as required by the *Instruction*.

Explanation

WC (Write C flag) is one of four optional effects (NR, WR, WZ, and WC) that influence the behavior of assembly instructions. WC causes an executing assembly instruction to modify the C flag in accordance with its results.

For example, the CMP (Compare Unsigned) instruction compares two values (destination and source) but does not automatically write the results to the C and Z flags. You can determine if the destination value is less than the source value by using the CMP instruction with the WC effect:

```
cmp     value1, value2   WC      'C = 1 if value1 < value2
```

The above CMP instruction compares value1 with value2 and sets C high (1) if value1 is less than value2.

See Effects on page 291 for more information.

NEW

WR

Effect: Cause assembly instruction to write a result.

⟨*Label*⟩ ⟨*Condition*⟩ *Instruction Operands* WR

Result: *Instruction*'s destination register is changed to the result value.

- ***Label*** is an optional statement label. See Common Syntax Elements on page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements on page 250.
- ***Instruction*** is the desired assembly instruction.
- ***Operands*** is zero, one, or two operands as required by the *Instruction*.

Explanation

WR (Write Result) is one of four optional effects (WC, WZ, NR, and WR) that influence the behavior of assembly instructions. WR causes an executing assembly instruction to write its result value to the destination register.

For example, by default the COGINIT (Cog Initialize) instruction does not write a result to the destination register. This is fine when wanting to start a cog with a specific ID, but when asking to start the next available cog (i.e., destination register value bit 3 is high) you may like to know what cog was actually started, by ID. You can get the new cog's ID from the COGINIT instruction by using the WR effect:

```
coginit   launch_value   WR     'Launch new cog, get ID back
```

Assuming launch_value points to a register that contained a high (1) in bit 3, the COGINIT instruction starts the next available cog and writes its ID back to the launch_value register.

See Effects on page 291 for more information.

WRBYTE

Instruction: Write a byte to main memory.

WRBYTE *Value*, ⟨#⟩ *Address*

- ***Value*** (d-field) is the register containing the 8-bit value to write to main memory.
- ***Address*** (s-field) is a register or a 9-bit literal whose value is the main memory address to write to.

Opcode Table:

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
000000	000i	1111	ddddddddd	sssssssss	---	---	Not Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	n/a	1	0

[1] Destination Out doesn't exist since including a WR effect would turn WRBYTE into a RDBYTE instruction.
[2] The Z flag is set (1) unless the main memory address is on a long boundary.

Explanation

WRBYTE synchronizes to the Hub and writes the lowest byte in *Value* to main memory at *Address*.

NEW

The WR effect can not be used with WRBYTE as that would change it to a RDBYTE instruction.

WRBYTE is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

WRLONG

Instruction: Write a long to main memory.

WRLONG ***Value***, ⟨#⟩ ***Address***

- ***Value*** (d-field) is the register containing the 32-bit value to write to main memory.
- ***Address*** (s-field) is a register or a 9-bit literal whose value is the main memory address to write to.

Opcode Table:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
000010 000i 1111	ddddddddd	sssssssss	---	---	Not Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	WZ WC	n/a	0	0

[1] Destination Out doesn't exist since including a WR effect would turn WRLONG into a RDLONG instruction.
[2] The Z flag is always cleared (0) since the main memory address (bits 13:2) is always on a long boundary.

Explanation

WRLONG synchronizes to the Hub and writes the long in *Value* to main memory at *Address*.

NEW

The WR effect can not be used with WRLONG as that would change it to a RDLONG instruction.

WRLONG is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. Hub on page 24 for more information.

WRWORD

Instruction: Write a word to main memory.

WRWORD *Value*, ⟨#⟩ *Address*

- ***Value*** (d-field) is the register containing the 16-bit value to write to main memory.
- ***Address*** (s-field) is a register or a 9-bit literal whose value is the main memory address to write to.

Opcode Table:

–INSTR–	ZCRI	–CON–	–DEST–	–SRC–	Z Result	C Result	Result	Clocks
000001	000i	1111	ddddddddd	sssssssss	---	---	Not Written	7..22

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	n/a	1	0

[1] Destination Out doesn't exist since including a WR effect would turn WRWORD into a RDWORD instruction.
[2] The Z flag is set (1) unless the main memory address (bits 13:1) is on a long boundary.

Explanation

WRWORD synchronizes to the Hub and writes the lowest word in *Value* to main memory at *Address*.

NEW

The WR effect can not be used with WRWORD as that would change it to a RDWORD instruction.

WRWORD is a hub instruction. Hub instructions require 7 to 22 clock cycles to execute depending on the relation between the cog's hub access window and the instruction's moment of execution. See Hub on page 24 for more information.

NEW

WZ

Effect: Cause assembly instruction to modify the Z flag.

⟨Label⟩ ⟨Condition⟩ Instruction Operands **WZ**

Result: Z flag is updated with status from the *Instruction*'s execution.

- ***Label*** is an optional statement label. See Common Syntax Elements on page 250.
- ***Condition*** is an optional execution condition. See Common Syntax Elements on page 250.
- ***Instruction*** is the desired assembly instruction.
- ***Operands*** is zero, one, or two operands as required by the *Instruction*.

Explanation

WZ (Write Z flag) is one of four optional effects (**NR**, **WR**, **WC**, and **WZ**) that influence the behavior of assembly instructions. **WZ** causes an executing assembly instruction to modify the Z flag in accordance with its results.

For example, the **CMP** (Compare Unsigned) instruction compares two values (destination and source) but does not automatically write the results to the C and Z flags. You can determine if the destination value is equal to the source value by using the **CMP** instruction with the **WZ** effect:

```
cmp     value1, value2   WZ      'Z = 1 if value1 = value2
```

The above **CMP** instruction compares `value1` with `value2` and sets Z high (1) if `value1` is equal to `value2`.

See Effects on page 291 for more information.

XOR

Instruction: Bitwise XOR two values.

XOR *Value1*, ⟨#⟩ *Value2*

Result: *Value1* XOR *Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to bitwise XOR with *Value2* and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is bitwise XORed with *Value1*.

Opcode Table:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Result	C Result	Result	Clocks
011011 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

NEW

Concise Truth Table:

In					Out		
Destination	Source	Z	C	Effects	Destination	Z	C
$0000_000A; 10	$0000_0005; 5	–	–	wz wc	$0000_000F; 15	0	0
$0000_000A; 10	$0000_0007; 7	–	–	wz wc	$0000_000D; 13	0	1
$0000_000A; 10	$0000_000A; 10	–	–	wz wc	$0000_0000; 0	1	0
$0000_000A; 10	$0000_000D; 13	–	–	wz wc	$0000_0007; 7	0	1
$0000_000A; 10	$0000_000F; 15	–	–	wz wc	$0000_0005; 5	0	0

Explanation

XOR (bitwise exclusive OR) performs a bitwise XOR of the value in *Value2* into that of *Value1*.

If the WZ effect is specified, the Z flag is set (1) if *Value1* XOR *Value2* equals zero. If the WC effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits. The result is written to *Value1* unless the NR effect is specified.

Appendix A: Reserved Word List

These words are always reserved, whether programming in Spin or Propeller Assembly.

IMPROVED

Table A-0-1: Propeller Reserved Word List

_CLKFREQ[s]	CONSTANT[s]	IF_NC_AND_NZ[a]	MIN[a]	PLL4X[s]	SUBSX[a]
_CLKMODE[s]	CTRA[d]	IF_NC_AND_Z[a]	MINS[a]	PLL8X[s]	SUBX[a]
_FREE[s]	CTRB[d]	IF_NC_OR_NZ[a]	MOV[a]	PLL16X[s]	SUMC[a]
_STACK[s]	DAT[s]	IF_NC_OR_Z[a]	MOVD[a]	POSX[d]	SUMNC[a]
_XINFREQ[s]	DIRA[d]	IF_NE[a]	MOVI[a]	PRI[s]	SUMNZ[a]
ABORT[s]	DIRB[d#]	IF_NEVER[a]	MOVS[a]	PUB[s]	SUMZ[a]
ABS[a]	DJNZ[a]	IF_NZ[a]	MUL[a#]	QUIT[s]	TEST[a]
ABSNEG[a]	ELSE[s]	IF_NZ_AND_C[a]	MULS[a#]	RCFAST[s]	TESTN[a]
ADD[a]	ELSEIF[s]	IF_NZ_AND_NC[a]	MUXC[a]	RCL[a]	TJNZ[a]
ADDABS[a]	ELSEIFNOT[s]	IF_NZ_OR_C[a]	MUXNC[a]	RCR[a]	TJZ[a]
ADDS[a]	ENC[a]	IF_NZ_OR_NC[a]	MUXNZ[a]	RCSLOW[s]	TO[s]
ADDSX[a]	FALSE[d]	IF_Z[a]	MUXZ[a]	RDBYTE[a]	TRUE[d]
ADDX[a]	FILE[s]	IF_Z_AND_C[a]	NEG[a]	RDLONG[a]	TRUNC[s]
AND[d]	FIT[a]	IF_Z_AND_NC[a]	NEGC[a]	RDWORD[a]	UNTIL[s]
ANDN[a]	FLOAT[s]	IF_Z_EQ_C[a]	NEGNC[a]	REBOOT[s]	VAR[s]
BYTE[s]	FROM[s]	IF_Z_NE_C[a]	NEGNZ[a]	REPEAT[s]	VCFG[d]
BYTEFILL[s]	FRQA[d]	IF_Z_OR_C[a]	NEGX[d]	RES[a]	VSCL[d]
BYTEMOVE[s]	FRQB[d]	IF_Z_OR_NC[a]	NEGZ[a]	RESULT[s]	WAITCNT[d]
CALL[a]	HUBOP[a]	INA[d]	NEXT[s]	RET[a]	WAITPEQ[d]
CASE[s]	IF[s]	INB[d#]	NOP[a]	RETURN[s]	WAITPNE[d]
CHIPVER[s]	IFNOT[s]	JMP[a]	NOT[s]	REV[a]	WAITVID[d]
CLKFREQ[s]	IF_A[a]	JMPRET[a]	NR[a]	ROL[a]	WC[a]
CLKMODE[s]	IF_AE[a]	LOCKCLR[d]	OBJ[s]	ROR[a]	WHILE[s]
CLKSET[d]	IF_ALWAYS[a]	LOCKNEW[d]	ONES[a#]	ROUND[s]	WORD[s]
CMP[a]	IF_B[a]	LOCKRET[d]	OR[d]	SAR[a]	WORDFILL[s]
CMPS[a]	IF_BE[a]	LOCKSET[d]	ORG[a]	SHL[a]	WORDMOVE[s]
CMPSUB[a]	IF_C[a]	LONG[s]	OTHER[s]	SHR[a]	WR[a]
CMPSX[a]	IF_C_AND_NZ[a]	LONGFILL[s]	OUTA[d]	SPR[s]	WRBYTE[a]
CMPX[a]	IF_C_AND_Z[a]	LONGMOVE[s]	OUTB[d#]	STEP[s]	WRLONG[a]
CNT[d]	IF_C_EQ_Z[a]	LOOKDOWN[s]	PAR[d]	STRCOMP[s]	WRWORD[a]
COGID[d]	IF_C_NE_Z[a]	LOOKDOWNZ[s]	PHSA[d]	STRING[s]	WZ[a]
COGINIT[d]	IF_C_OR_NZ[a]	LOOKUP[s]	PHSB[d]	STRSIZE[s]	XINPUT[s]
COGNEW[s]	IF_C_OR_Z[a]	LOOKUPZ[s]	PI[d]	SUB[a]	XOR[a]
COGSTOP[d]	IF_E[a]	MAX[a]	PLL1X[s]	SUBABS[a]	XTAL1[s]
CON[s]	IF_NC[a]	MAXS[a]	PLL2X[s]	SUBS[a]	XTAL2[s]
					XTAL3[s]

a = Assembly element; s = Spin element; d = dual (available in both languages); # = reserved for future use

Appendix B: Math Samples and Function Tables

NEW

Multiplication, Division, and Square Root

Multiplication, division, and square root can be computed by using add, subtract, and shift instructions. Here is an unsigned multiplier routine that multiplies two 16-bit values to yield a 32-bit product:

```
'' Multiply x[15..0] by y[15..0] (y[31..16] must be 0)
' on exit, product in y[31..0]
'
multiply        shl     x,#16           'get multiplicand into x[31..16]
                mov     t,#16           'ready for 16 multiplier bits
                shr     y,#1      wc    'get initial multiplier bit into c
:loop  if_c     add     y,x       wc    'if c set, add multiplicand to product
                rcr     y,#1      wc    'put next multiplier in c, shift prod.
                djnz    t,#:loop        'loop until done
multiply_ret    ret                     'return with product in y[31..0]
```

The above routine's execution time could be cut by ~1/3 if the loop was unrolled, repeating the **ADD** / **RCR** sequence and getting rid of the **DJNZ** instruction.

Division is like multiplication, but backwards. It is potentially more complex, though, because a comparison test must be performed before a subtraction can take place. To remedy this, there don't is a special **CMPSUB** *D*, *S* instructions which tests to see if a subtraction can be performed without causing an underflow. If no underflow would occur, the subtraction takes place and the C output is 1. If an underflow would occur, D is left alone and the C output is 0.

Here is an unsigned divider routine that divides a 32-bit value by a 16-bit value to yield a 16-bit quotient and a 16-bit remainder:

```
' Divide x[31..0] by y[15..0] (y[16] must be 0)
' on exit, quotient is in x[15..0] and remainder is in x[31..16]
'
divide          shl     y,#15           'get divisor into y[30..15]
                mov     t,#16           'ready for 16 quotient bits
:loop           cmpsub  x,y       wc    'y =< x? Subtract it, quotient bit in c
                rcl     x,#1            'rotate c into quotient, shift dividend
                djnz    t,#:loop        'loop until done
divide_ret      ret                     'quotient in x[15..0],
                                        'remainder in x[31..16]
```

Like the multiplier routine, this divider routine could be recoded with a sequence of 16 CMPSUB + RCL instruction pairs to get rid of the DJNZ and cut execution time by ~1/3. By making such changes, speed can often be gained at the expense of code size.

Here is a square-root routine that uses the CMPSUB instruction:

```
'
' Compute square-root of y[31..0] into x[15..0]
'
root            mov     a,#0            'reset accumulator
                mov     x,#0            'reset root
                mov     t,#16           'ready for 16 root bits
:loop           shl     y,#1    wc      'rotate top two bits of y to accumulator
                rcl     a,#1
                shl     y,#1    wc
                rcl     a,#1
                shl     x,#2            'determine next bit of root
                or      x,#1
                cmpsub  a,x     wc
                shr     x,#2
                rcl     x,#1
                djnz    t,#:loop        'loop until done
root_ret        ret                     'square root in x[15..0]
```

Many complex math functions can be realized by additions, subtractions, and shifts. Though specific examples were given here, these types of algorithms may be coded in many different ways to best suit the application.

Log and Anti-Log Tables ($C000-DFFF)

The log and anti-log tables are useful for converting values between their number form and exponent form.

When numbers are encoded into exponent form, simple math operations take on more complex effects. For example ‘add’ and ‘subtract’ become ‘multiply’ and ‘divide,’ ‘shift-left’ becomes ‘square’ and ‘shift-right’ becomes ‘square-root,’ and ‘divide by 3’ will produce ‘cube root.’ Once the exponent is converted back to a number, the result will be apparent. This process is imperfect, but quite fast.

For applications where many multiplies and divides must be performed in the absence of many additions and subtractions, exponential encoding can greatly speed things up. Exponential encoding is also useful for compressing numbers into fewer bits – sacrificing resolution at higher magnitude. In many applications, such as audio synthesis, the nature of signals is logarithmic in both frequency and magnitude. Processing such data in exponent

form is quite natural and efficient, as it lends a 'linear' simplicity to what is actually logarithmic.

The code examples given below use each tables' samples verbatim. Higher resolution could be achieved by linearly interpolating between table samples, since the slope change is very slight from sample to sample. The cost, though, would be larger code and lower execution speed.

Log Table ($C000-$CFFF)

The log table contains data used to convert unsigned numbers into base-2 exponents.

The log table is comprised of 2,048 unsigned words which make up the base-2 fractional exponents of numbers. To use this table, you must first determine the integer portion of the exponent of the number you are converting. This is simply the leading bit position. For $60000000 this would be 30 ($1E). This integer portion will always fit within 5 bits. Isolate these 5 bits into the result so that they occupy bit positions 20..16. In our case of $60000000, we would now have a partial result of $001E0000. Next, top-justify and isolate the first 11 bits below the leading bit into positions 11..1. This would be $0800 for our example. Add $C000 for the log table base and you now have the actual word address of the fractional exponent. By reading the word at $C800, we get the value $95C0. Adding this into the partial result yields $001E95C0 – that's $60000000 in exponent form. Note that bits 20..16 make up the integer portion of the exponent, while bits 15..0 make up the fractional portion, with bit 15 being the ½, bit 14 being the ¼, and so on, down to bit 0. The exponent can now be manipulated by adding, subtracting, and shifting. Always insure that your math operations will never drive the exponent below 0 or cause it to overflow bit 20. Otherwise, it may not convert back to a number correctly.

Here is a routine that will convert an unsigned number into its base-2 exponent using the log table:

```
' Convert number to exponent
'
' on entry: num holds 32-bit unsigned value
' on exit:  exp holds 21-bit exponent with 5 integer bits and 16 fractional bits
'
numexp          mov     exp,#0              'clear exponent

                test    num,num4        wz  'get integer portion of exponent
                muxnz   exp,exp4            'while top-justifying number
    if_z        shl     num,#16
                test    num,num3        wz
                muxnz   exp,exp3
    if_z        shl     num,#8
```

```
                test    num,num2        wz
                muxnz   exp,exp2
    if_z        shl     num,#4
                test    num,num1        wz
                muxnz   exp,exp1
    if_z        shl     num,#2
                test    num,num0        wz
                muxnz   exp,exp0
    if_z        shl     num,#1

                shr     num,#30-11          'justify sub-leading bits as word
offset
                and     num,table_mask      'isolate table offset bits
                add     num,table_log       'add log table address
                rdword  num,num             'read fractional portion of exponent
                or      exp,num             'combine fractional & integer portions

numexp_ret      ret                         '91..106 clocks
                                            '(variance due to HUB sync on RDWORD)

num4            long    $FFFF0000
num3            long    $FF000000
num2            long    $F0000000
num1            long    $C0000000
num0            long    $80000000
exp4            long    $00100000
exp3            long    $00080000
exp2            long    $00040000
exp1            long    $00020000
exp0            long    $00010000
table_mask      long    $0FFE               'table offset mask
table_log       long    $C000               'log table base

num             long    0                   'input
exp             long    0                   'output
```

Anti-Log Table ($D000-$DFFF)

The anti-log table contains data used to convert base-2 exponents into unsigned numbers.

The anti-log table is comprised of 2,048 unsigned words which are each the lower 16-bits of a 17-bit mantissa (the 17th bit is implied and must be set separately). To use this table, shift the top 11 bits of the exponent fraction (bits 15..5) into bits 11..1 and isolate. Add $D000 for the anti-log table base. Read the word at that location into the result – this is the mantissa. Next, shift the mantissa left to bits 30..15 and set bit 31 – the missing 17th bit of the mantissa. The last step is to shift the result right by 31 minus the exponent integer in bits 20..16. The exponent is now converted to an unsigned number.

Here is a routine that will convert a base-2 exponent into an unsigned number using the anti-log table:

```
' Convert exponent to number
'
' on entry: exp holds 21-bit exponent with 5 integer bits and 16 fraction bits
' on exit:  num holds 32-bit unsigned value
'
expnum          mov     num,exp             'get exponent into number
                shr     num,#15-11          'justify exponent fraction as word
offset
                and     num,table_mask      'isolate table offset bits
                or      num,table_antilog   'add anti-log table address
                rdword  num,num             'read mantissa word into number
                shl     num,#15             'shift mantissa into bits 30..15
                or      num,num0            'set top bit (17th bit of mantissa)
                shr     exp,#20-4           'shift exponent integer into bits 4..0
                xor     exp,#$1F            'inverse bits to get shift count
                shr     num,exp             'shift number into final position

expnum_ret      ret                         '47..62 clocks
                                            '(variance is due to HUB sync on
RDWORD)

num0            long    $80000000           '17th bit of the mantissa
table_mask      long    $0FFE               'table offset mask
table_antilog   long    $C000               'anti-log table base

exp             long    0                   'input
num             long    0                   'output
```

Sine Table ($E000-$F001)

The sine table provides 2,049 unsigned 16-bit sine samples spanning from 0° to 90°, inclusively (0.0439° resolution).

A small amount of assembly code can mirror and flip the sine table samples to create a full-cycle sine/cosine lookup routine that has 13-bit angle resolution and 17-bit sample resolution:

```
' Get sine/cosine
'
'
'       quadrant:  1             2             3             4
'          angle:  $0000..$07FF  $0800..$0FFF  $1000..$17FF  $1800..$1FFF
'    table index:  $0000..$07FF  $0800..$0001  $0000..$07FF  $0800..$0001
'         mirror:  +offset       -offset       +offset       -offset
'           flip:  +sample       +sample       -sample       -sample
'
' on entry: sin[12..0] holds angle (0° to just under 360°)
' on exit:  sin holds signed value ranging from $0000FFFF ('1') to
' $FFFF0001 ('-1')
'
getcos          add     sin,sin_90          'for cosine, add 90°
getsin          test    sin,sin_90      wc  'get quadrant 2|4 into c
                test    sin_sin_180     wz  'get quadrant 3|4 into nz
                negc    sin,sin             'if quadrant 2|4, negate offset
                or      sin,sin_table       'or in sin table address >> 1
                shl     sin,#1              'shift left to get final word address
                rdword  sin,sin             'read word sample from $E000 to $F000
                negnz   sin,sin                 'if quadrant 3|4, negate sample
getsin_ret
getcos_ret      ret                         '39..54 clocks
                                            '(variance due to HUB sync on RDWORD)


sin_90          long    $0800
sin_180         long    $1000
sin_table       long    $E000 >> 1          'sine table base shifted right

sin             long    0
```

As with the log and anti-log tables, linear interpolation could be applied to the sine table to achieve higher resolution.

Index

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z