WRD william robert drury

bob\_drury@hotmail.com

Abstract

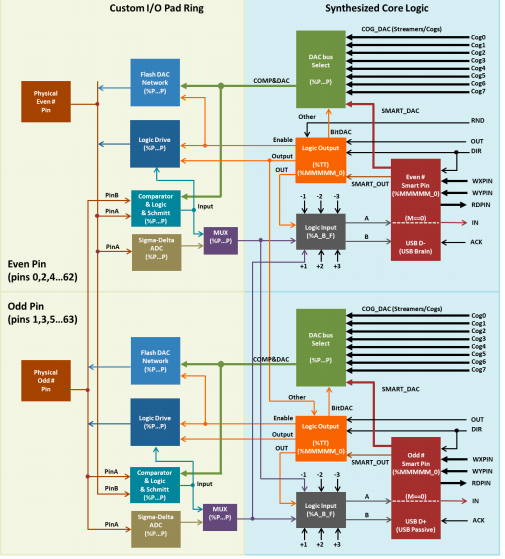
Spin2 Example Programs

HELLO Propeller II   
eXAMPLES

User Notes

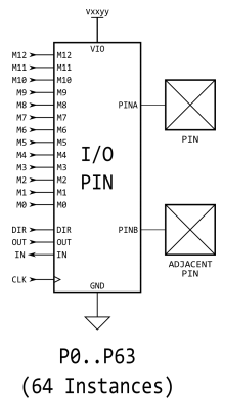
# Smart Pin Block Diagram

Each of the 64 I/O pins in a Propeller-2 microcontroller can operate as a Smart Pin. In brief, every Smart Pin provides access to internal functions such as analog-to-digital converters, digital-to-analog converters, signal generators, PWM controllers, and so on. The Propeller-2 architecture lets these functions operate independent of the cogs so they don't rely on software interactions to "micromanagement" their control and operation. Normally, a DIR bit controls an I/O pin's output enable, while the IN bit returns the pin's state. In Smart Pin modes, though, these bits serve different purposes. The DIR bit controls an active-low (logic-0) reset signal for the selected Smart Pin's circuitry, while a configuration bit controls the pin's output enable state. In some modes, the Smart-Pin circuit directly controls the pin's output state, in which case the OUT bit gets ignored. The IN bit serves as a "finished" flag that indicate to a cog(s) that the Smart Pin has completed some function, or an event has occurred. Depending on the operation, software might need to acknowledge a set IN flag (and reset it?). The block diagram below shows the main functions for a Smart Pin. At first this information might seem complicated, but later explanations of the functions, registers, and instructions clarify their use.



## 1.1) Smart Pin Schematic

Every I/O pin features versatile digital and analog capabilities as well as autonomous state machine functions that would otherwise require processor time to perform. The combination provides adept functionality for application design, increasing the Propeller 2 potential beyond what multi-core architecture alone provides. There are 24 low-level 'pin' modes and 34 high-level 'smart' modes. Pin Modes Each I/O pin has 13 low-level pin mode configuration bits which determine the operation of its 3.3 V circuit.



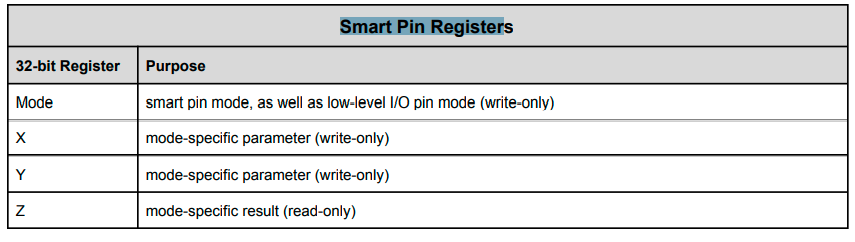
The pin mode is set using the WRPIN instruction, where the 13 %MMMMMMMMMMMMM bits within the instruction's D operand go directly to these bits. Note though that in some smart pin modes, these bits are partially overwritten to set things like DAC values.

Note: Upon startup or reset, all I/O pins default to input (high impedance), meaning each cog's direction registers are initialized to zero. Each cog's output registers are initialized to zero as well, but this low (ground) state is not reflected on the pin until the pin is set to the output direction (via the direction register).  
**Pins to avoid for starndard configuration:**

SP58: MISO (connection to SPI Flash Data Out pin or Micro SD MISO pin)   
P59: MOSI (connection to SPI Flash Data In pin or Micro SD MOSI pin)   
P60: CLK / CS (connection to SPI Flash CLK pin or Micro SD CS pin)   
P61: CS / CLK (connection to SPI Flash CS pin or Micro SD CLK pin)   
P62: Serial Tx (connection to host's Serial Rx)   
P63: Serial Rx (connection to host's Serial Tx)

## 1.2) Smart Pin WRPIN,WXPIN and WYPIN

Smart Modes Each I/O pin has built-in 'smart pin' circuitry which, when enabled, performs an autonomous function on the pin. Smart pins free the cogs from the need to micromanage many I/O operations by providing high-bandwidth concurrent hardware functions that cogs could otherwise not perform as well through I/O pin manipulating instructions. In normal operation, an I/O pin's output enable is controlled by its DIR bit, its output state is controlled by its OUT bit, and its IN bit returns the pin's read state. With smart pin mode enabled, its DIR bit is used as an active-low reset signal to the smart pin circuitry, while the output enable state is controlled by a configuration bit. In some modes, the smart pin circuit takes over driving the output state, in which case the OUT bit gets ignored. Its IN bit serves as a flag to indicate to the cog(s) that the smart pin has completed some function or an event has occurred, and acknowledgment is perhaps needed. To configure a smart pin, first set its DIR bit to low (holding it in reset) then use WRPIN, WXPIN, and WYPIN to establish the mode and related parameters. Once configured, DIR can be raised high and the smart pin will begin operating. After that, depending on the mode, you may feed it new data via WXPIN/WYPIN or retrieve results using RDPIN/RQPIN. These activities are usually coordinated with the IN signal going high; explained later. Note that while a smart pin is configured, the %TT bits (of the WRPIN instruction's D operand) will govern the pin's output enable, regardless of the DIR state. Smart pins have four 32-bit registers inside of them:



Note: S/# indicates a literal 9-bit pin number (0..63) or a symbol such as LED\_pin, you defined earlier. D/# indicates a literal 9-bit value or a symbol such as sensor\_12A, you defined earlier. {WC} indicates the operation affects the carry flag.

These four registers are written and read via the following **PASM 2-clock instructions**, in which S/# is used to select the pin number (0..63) and D/# is the 32-bit data conduit:

**WRPIN D/#,S/# - Set smart pin S/# mode to D/#, ack pin**

**WXPIN D/#,S/# - Set smart pin S/# parameter X to D/#, ack pin**

**WYPIN D/#,S/# - Set smart pin S/# parameter Y to D/#, ack pin**

**RDPIN D,S/# {WC} - Get smart pin S/# result Z into D, flag into C, ack pin**

**RQPIN D,S/# {WC} - Get smart pin S/# result Z into D, flag into C, don't ack pin**

**AKPIN S/# - Acknowledge pin S/#**

The format of the D (pin setup) operand value is:

**D = %AAAA\_BBBB\_FFF\_MMMMMMMMMMMMM\_TT\_SSSSS\_0**

● A = PINA input selector

● B = PINB input selector

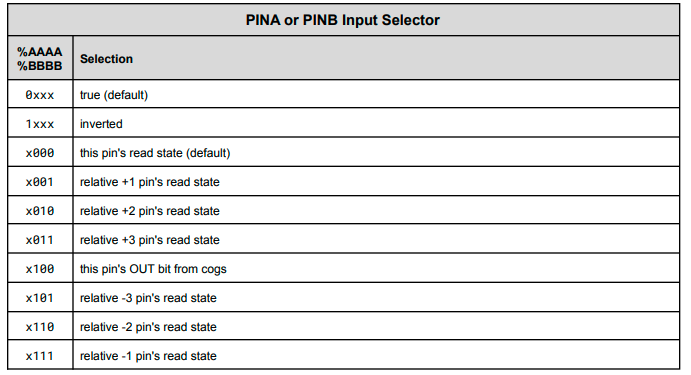
● F = PINA and PINB input logic/filtering (after PINA and PINB input selectors)

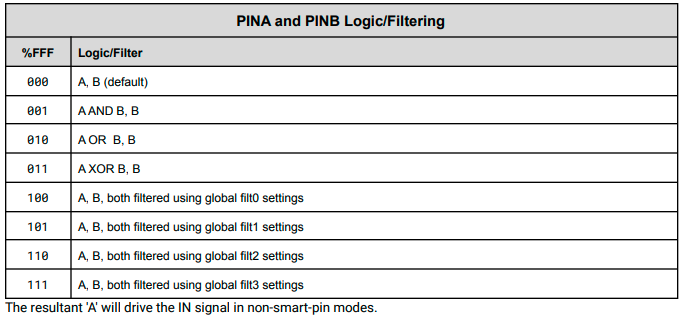
● M = pin mode

● T = pin DIR/OUT control (default = %00)

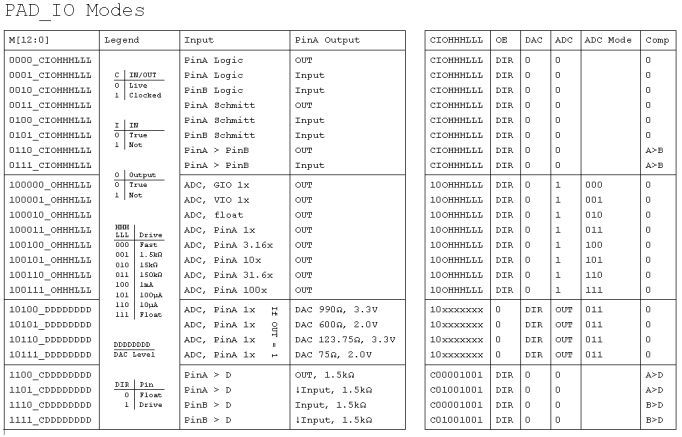
● S = smart mode

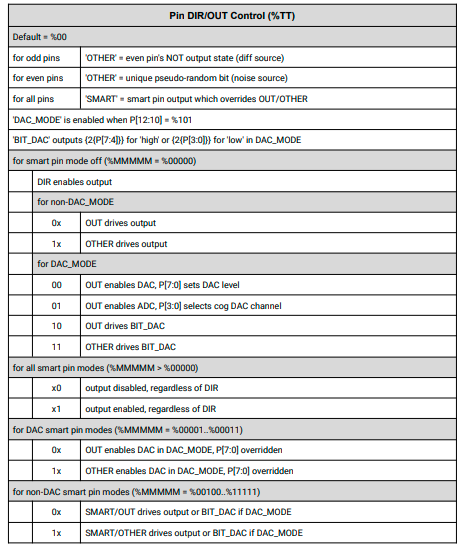
Each smart pin has a 34-bit input bus and a 33-bit output bus that connect it to the cogs. To configure and control smart pins, each cog writes data and acknowledgement signals to the smart pin input bus. Each smart pin OR's all incoming 34-bit buses from the collective of cogs in the same way DIR and OUT bits are OR'd before going to the pins. Therefore, if you intend to have multiple cogs execute WRPIN / WXPIN / WYPIN / RDPIN / AKPIN instructions on the same smart pin, you must be sure that they do so at different times, in order to avoid clobbering each other's bus data. Reading a smart pin with RDPIN can cause the same conflict; however, any number of cogs can read a smart pin simultaneously without bus conflict by using RQPIN ('read quiet'), since it does not utilize the smart pin input bus for acknowledgement signalling (like RDPIN does). Each smart pin writes to it's output bus to convey its Z result and a special flag. The RDPIN and RQPIN multiplex and read these buses, so that a pin's Z result is read into D and its special flag can be read into C. C will be either a mode-related flag or the MSB of the Z result. When a mode-related event occurs in a smart pin, it raises its IN signal to alert the cog(s) that new data is ready, new data can be loaded, or some process has finished. A cog can test for this signal via the TESTP instruction and can acknowledge a smart pin by executing a WRPIN, WXPIN, WYPIN, RDPIN, or AKPIN instruction for it. This acknowledgement causes the smart pin to lower its IN signal so that it can be raised again on the next event. After a WRPIN/WXPIN/WYPIN/RDPIN/AKPIN, it takes two clocks for IN to drop, before it can be polled again. A smart pin can be reset at any time, without the need to reconfigure it, by clearing and then setting its DIR bit. To return a pin to normal mode, do a 'WRPIN #0,pin'



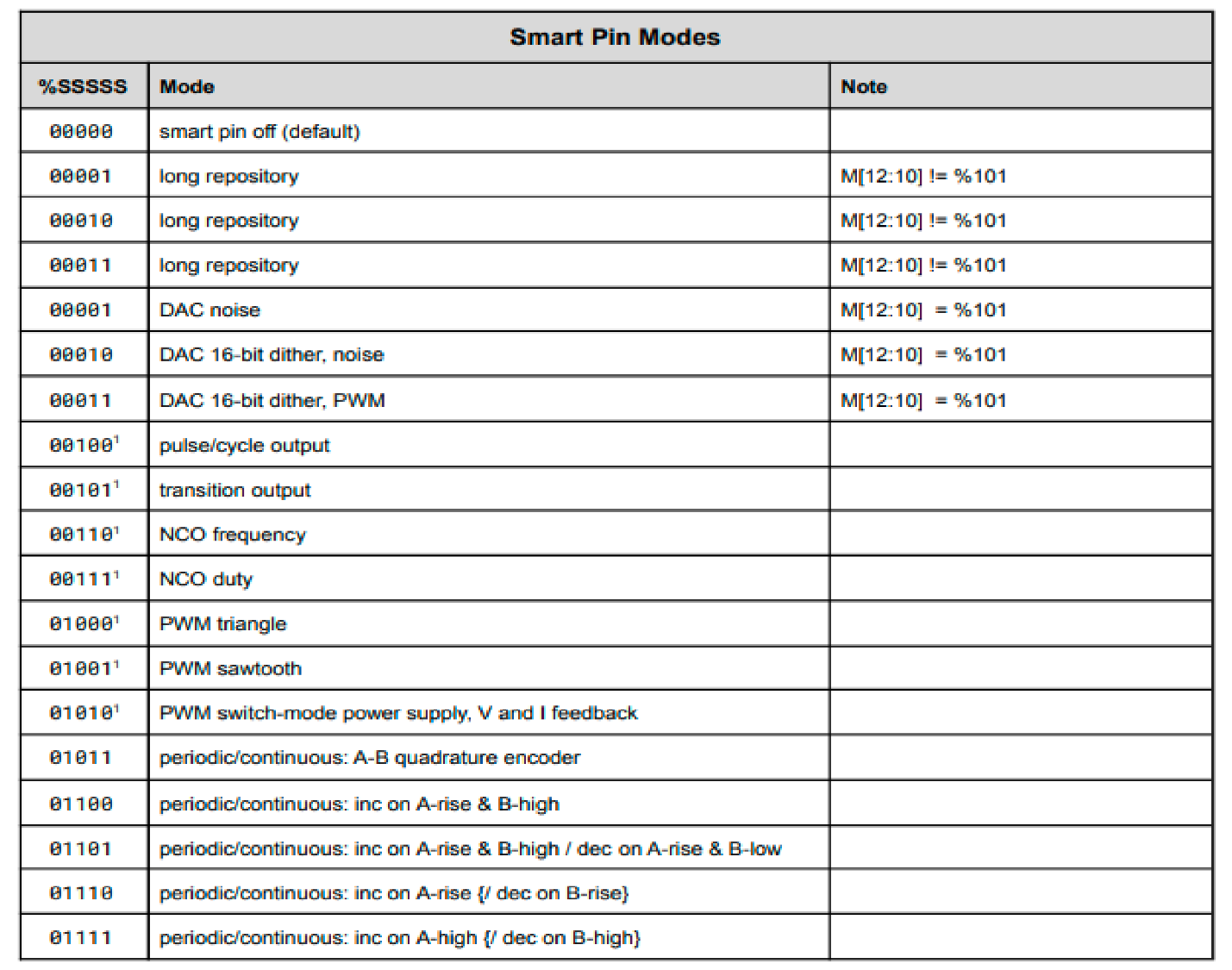


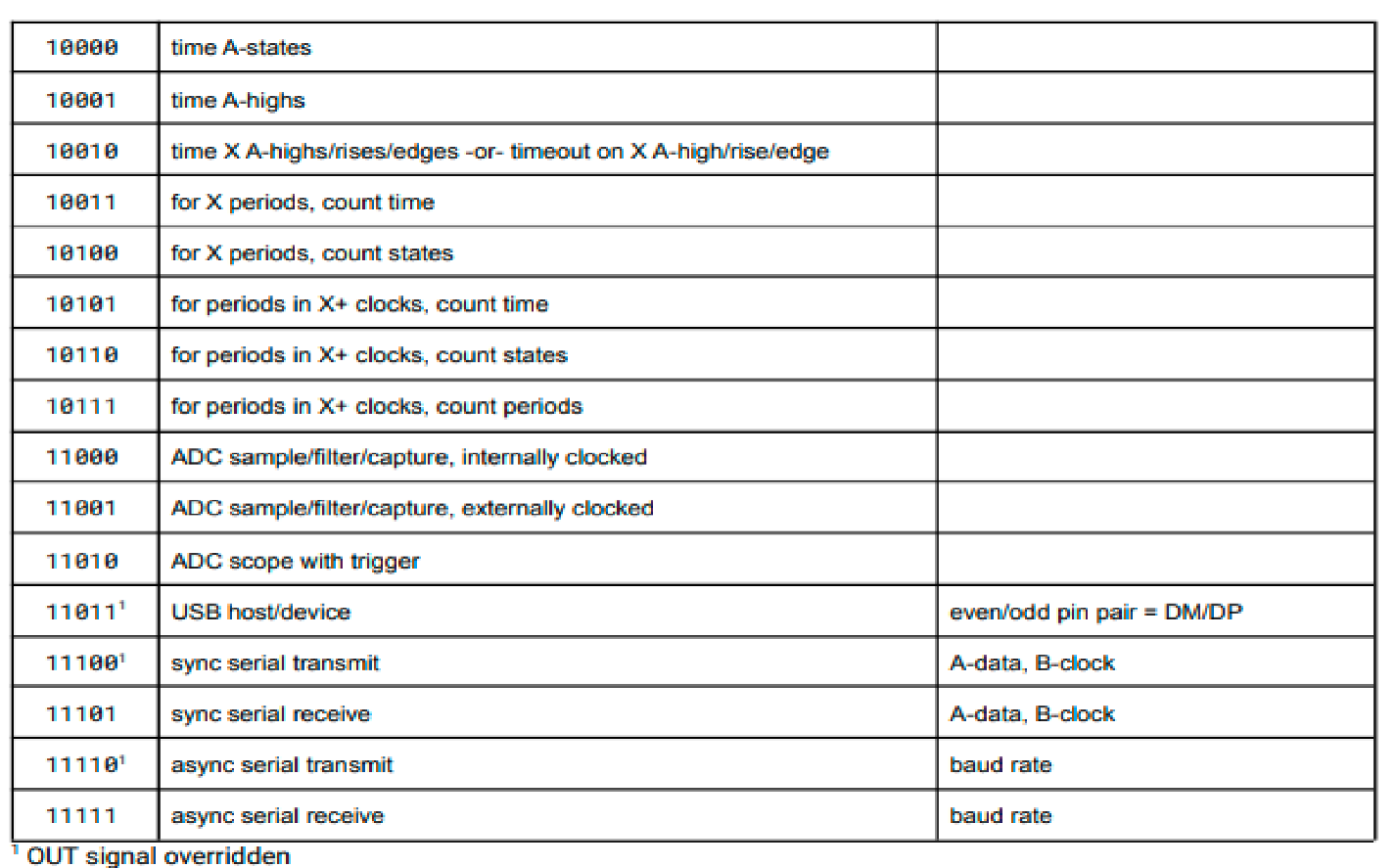
Pin Modes corresponding to the 13-bit M field are described by this table (?)





## 1.3) SSSSS Smart Pin Mode Setting

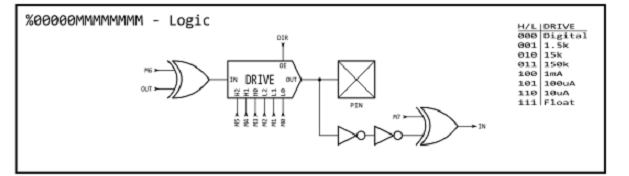




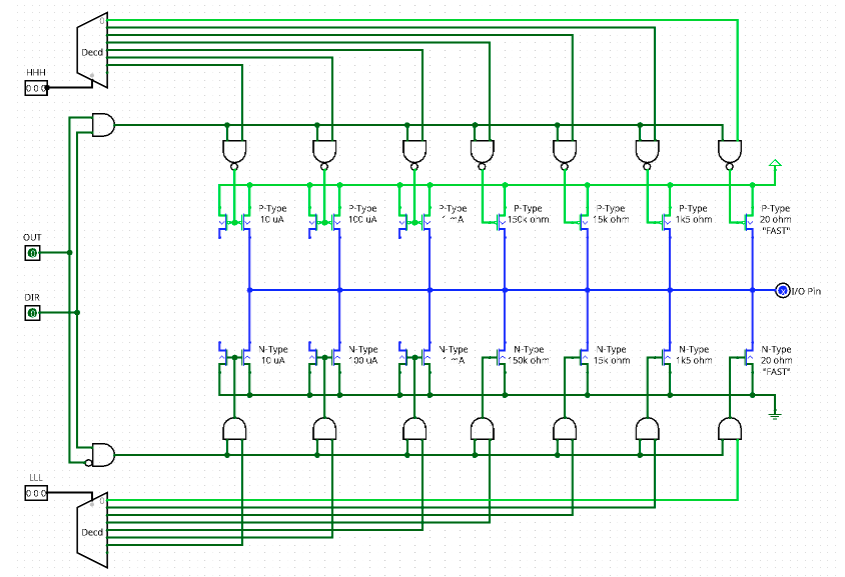
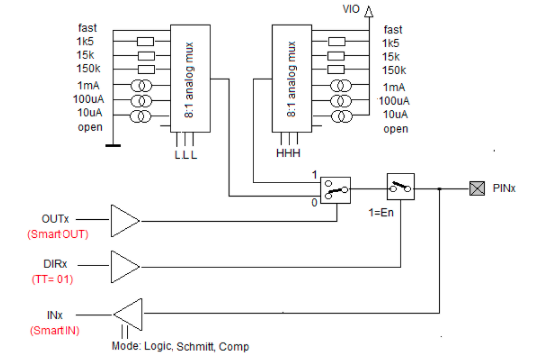
## 1.4) Smart Pin Symbol Names

|  |  |  |
| --- | --- | --- |
| **Smart Pin Symbol Value** | **Symbol Name** | **Details** |
| **A Input Polarity** | **(pick one)** |  |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_TRUE\_A (default) | True A input |
| %1000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_INVERT\_A | Invert A input |
| **A Input Selection** | **(pick one)** |  |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_LOCAL\_A (default) | Select local pin for A input |
| %0001\_0000\_000\_0000000000000\_00\_00000\_0 | P\_PLUS1\_A | Select pin+1 for A input |
| %0010\_0000\_000\_0000000000000\_00\_00000\_0 | P\_PLUS2\_A | Select pin+2 for A input |
| %0011\_0000\_000\_0000000000000\_00\_00000\_0 | P\_PLUS3\_A | Select pin+3 for A input |
| %0100\_0000\_000\_0000000000000\_00\_00000\_0 | P\_OUTBIT\_A | Select OUT bit for A input |
| %0101\_0000\_000\_0000000000000\_00\_00000\_0 | P\_MINUS3\_A | Select pin-3 for A input |
| %0110\_0000\_000\_0000000000000\_00\_00000\_0 | P\_MINUS2\_A | Select pin-2 for A input |
| %0111\_0000\_000\_0000000000000\_00\_00000\_0 | P\_MINUS1\_A | Select pin-1 for A input |
| **B Input Polarity** | **(pick one)** |  |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_TRUE\_B (default) | True B input |
| %0000\_1000\_000\_0000000000000\_00\_00000\_0 | P\_INVERT\_B | Invert B input |
| **B Input Selection** | **(pick one)** |  |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_LOCAL\_B (default) | Select local pin for B input |
| %0000\_0001\_000\_0000000000000\_00\_00000\_0 | P\_PLUS1\_B | Select pin+1 for B input |
| %0000\_0010\_000\_0000000000000\_00\_00000\_0 | P\_PLUS2\_B | Select pin+2 for B input |
| %0000\_0011\_000\_0000000000000\_00\_00000\_0 | P\_PLUS3\_B | Select pin+3 for B input |
| %0000\_0100\_000\_0000000000000\_00\_00000\_0 | P\_OUTBIT\_B | Select OUT bit for B input |
| %0000\_0101\_000\_0000000000000\_00\_00000\_0 | P\_MINUS3\_B | Select pin-3 for B input |
| %0000\_0110\_000\_0000000000000\_00\_00000\_0 | P\_MINUS2\_B | Select pin-2 for B input |
| %0000\_0111\_000\_0000000000000\_00\_00000\_0 | P\_MINUS1\_B | Select pin-1 for B input |
| **A, B Input Logic** | **(pick one)** |  |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_PASS\_AB (default) | Select A, B |
| %0000\_0000\_001\_0000000000000\_00\_00000\_0 | P\_AND\_AB | Select A & B, B |
| %0000\_0000\_010\_0000000000000\_00\_00000\_0 | P\_OR\_AB | Select A | B, B |
| %0000\_0000\_011\_0000000000000\_00\_00000\_0 | P\_XOR\_AB | Select A ^ B, B |
| %0000\_0000\_100\_0000000000000\_00\_00000\_0 | P\_FILT0\_AB | Select FILT0 settings for A, B |
| %0000\_0000\_101\_0000000000000\_00\_00000\_0 | P\_FILT1\_AB | Select FILT1 settings for A, B |
| %0000\_0000\_110\_0000000000000\_00\_00000\_0 | P\_FILT2\_AB | Select FILT2 settings for A, B |
| %0000\_0000\_111\_0000000000000\_00\_00000\_0 | P\_FILT3\_AB | Select FILT3 settings for A, B |
| **Low-Level Pin Modes** | **(pick one)** |  |
| **Logic/Schmitt/Comparator Input Modes** |  |  |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_LOGIC\_A (default) | Logic level A → IN, output OUT |
| %0000\_0000\_000\_0001000000000\_00\_00000\_0 | P\_LOGIC\_A\_FB | Logic level A → IN, output feedback |
| %0000\_0000\_000\_0010000000000\_00\_00000\_0 | P\_LOGIC\_B\_FB | Logic level B → IN, output feedback |
| %0000\_0000\_000\_0011000000000\_00\_00000\_0 | P\_SCHMITT\_A | Schmitt trigger A → IN, output OUT |
| %0000\_0000\_000\_0100000000000\_00\_00000\_0 | P\_SCHMITT\_A\_FB | Schmitt trigger A → IN, output feedback |
| %0000\_0000\_000\_0101000000000\_00\_00000\_0 | P\_SCHMITT\_B\_FB | Schmitt trigger B → IN, output feedback |
| %0000\_0000\_000\_0110000000000\_00\_00000\_0 | P\_COMPARE\_AB | A > B → IN, output OUT |
| %0000\_0000\_000\_0111000000000\_00\_00000\_0 | P\_COMPARE\_AB\_FB | A > B → IN, output feedback |
| %xxxx\_xxxx\_xxx\_xxxxSIOHHHLLL\_xx\_xxxxx\_x |  | Sync mode, IN/output polarity, high/low drive |
| **ADC Input Modes** |  |  |
| %0000\_0000\_000\_1000000000000\_00\_00000\_0 | P\_ADC\_GIO | ADC GIO → IN, output OUT |
| %0000\_0000\_000\_1000010000000\_00\_00000\_0 | P\_ADC\_VIO | ADC VIO → IN, output OUT |
| %0000\_0000\_000\_1000100000000\_00\_00000\_0 | P\_ADC\_FLOAT | ADC FLOAT → IN, output OUT |
| %0000\_0000\_000\_1000110000000\_00\_00000\_0 | P\_ADC\_1X | ADC 1x → IN, output OUT |
| %0000\_0000\_000\_1001000000000\_00\_00000\_0 | P\_ADC\_3X | ADC 3.16x → IN, output OUT |
| %0000\_0000\_000\_1001010000000\_00\_00000\_0 | P\_ADC\_10X | ADC 10x → IN, output OUT |
| %0000\_0000\_000\_1001100000000\_00\_00000\_0 | P\_ADC\_30X | ADC 31.6x → IN, output OUT |
| %0000\_0000\_000\_1001110000000\_00\_00000\_0 | P\_ADC\_100X | ADC 100x → IN, output OUT |
| %xxxx\_xxxx\_xxx\_xxxxxxOHHHLLL\_xx\_xxxxx\_x |  | O = output polarity, HHH/LLL = high/low drive |
| **DAC Output Modes** |  | **DIR enables output, OUT enables ADC** |
| %0000\_0000\_000\_1010000000000\_00\_00000\_0 | P\_DAC\_990R\_3V | DAC 990Ω, 3.3V peak, ADC 1x → IN |
| %0000\_0000\_000\_1010100000000\_00\_00000\_0 | P\_DAC\_600R\_2V | DAC 600Ω, 2.0V peak, ADC 1x → IN |
| %0000\_0000\_000\_1011000000000\_00\_00000\_0 | P\_DAC\_124R\_3V | DAC 123.75Ω, 3.3V peak, ADC 1x → IN |
| %0000\_0000\_000\_1011100000000\_00\_00000\_0 | P\_DAC\_75R\_2V | DAC 75Ω, 2.0V peak, ADC 1x → IN |
| %xxxx\_xxxx\_xxx\_xxxxxDDDDDDDD\_xx\_xxxxx\_x |  | DDDDDDDD = 8-bit DAC value |
| **Level-Comparison Modes** |  | **DIR enables output (1.5kΩ drive)** |
| %0000\_0000\_000\_1100000000000\_00\_00000\_0 | P\_LEVEL\_A | A > Level → IN, output OUT |
| %0000\_0000\_000\_1101000000000\_00\_00000\_0 | P\_LEVEL\_A\_FBN | A > Level → IN, output negative feedback |
| %0000\_0000\_000\_1110000000000\_00\_00000\_0 | P\_LEVEL\_B\_FBP | B > Level → IN, output positive feedback |
| %0000\_0000\_000\_1111000000000\_00\_00000\_0 | P\_LEVEL\_B\_FBN | B > Level → IN, output negative feedback |
| %xxxx\_xxxx\_xxx\_xxxxSLLLLLLLL\_xx\_xxxxx\_x |  | S = Synchronous, LLLLLLLL = 8-bit Level |
| **Low-Level Pin Sub-Modes** |  |  |
| **Sync Mode** | (pick one) | (for Logic/Schmitt/Comparator/Level modes) |
| %xxxx\_xxxx\_xxx\_xxxxSxxxxxxxx\_xx\_xxxxx\_x |  | Sync mode bit |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_ASYNC\_IO (default) | Select asynchronous I/O |
| %0000\_0000\_000\_0000100000000\_00\_00000\_0 | P\_SYNC\_IO | Select synchronous I/O |
| **IN Polarity** | (pick one) | (for Logic/Schmitt/Comparator modes) |
| %xxxx\_xxxx\_xxx\_xxxxxIxxxxxxx\_xx\_xxxxx\_x |  | IN polarity bit |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_TRUE\_IN (default) | True IN bit |
| %0000\_0000\_000\_0000010000000\_00\_00000\_0 | P\_INVERT\_IN | Invert IN bit |
| **Output Polarity** | (pick one) | (for Logic/Schmitt/Comparator/ADC modes) |
| %xxxx\_xxxx\_xxx\_xxxxxxOxxxxxx\_xx\_xxxxx\_x |  | Output polarity bit |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_TRUE\_OUTPUT (default) | Select true output |
| %0000\_0000\_000\_0000001000000\_00\_00000\_0 | P\_INVERT\_OUTPUT | Select inverted output |
| **Drive-High Strength** | (pick one) | (for Logic/Schmitt/Comparator/ADC modes) |
| %xxxx\_xxxx\_xxx\_xxxxxxxHHHxxx\_xx\_xxxxx\_x |  | Drive-high selector bits |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_HIGH\_FAST (default) | Drive high fast (30mA) |
| %0000\_0000\_000\_0000000001000\_00\_00000\_0 | P\_HIGH\_1K5 | Drive high 1.5kΩ |
| %0000\_0000\_000\_0000000010000\_00\_00000\_0 | P\_HIGH\_15K | Drive high 15kΩ |
| %0000\_0000\_000\_0000000011000\_00\_00000\_0 | P\_HIGH\_150K | Drive high 150kΩ |
| %0000\_0000\_000\_0000000100000\_00\_00000\_0 | P\_HIGH\_1MA | Drive high 1mA |
| %0000\_0000\_000\_0000000101000\_00\_00000\_0 | P\_HIGH\_100UA | Drive high 100μA |
| %0000\_0000\_000\_0000000110000\_00\_00000\_0 | P\_HIGH\_10UA | Drive high 10μA |
| %0000\_0000\_000\_0000000111000\_00\_00000\_0 | P\_HIGH\_FLOAT | Float high |
| **Drive-Low Strength** | (pick one) | (for Logic/Schmitt/Comparator/ADC modes) |
| %xxxx\_xxxx\_xxx\_xxxxxxxxxxLLL\_xx\_xxxxx\_x |  | Drive-low selector bits |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_LOW\_FAST (default) | Drive low fast (30mA) |
| %0000\_0000\_000\_0000000000001\_00\_00000\_0 | P\_LOW\_1K5 | Drive low 1.5kΩ |
| %0000\_0000\_000\_0000000000010\_00\_00000\_0 | P\_LOW\_15K | Drive low 15kΩ |
| %0000\_0000\_000\_0000000000011\_00\_00000\_0 | P\_LOW\_150K | Drive low 150kΩ |
| %0000\_0000\_000\_0000000000100\_00\_00000\_0 | P\_LOW\_1MA | Drive low 1mA |
| %0000\_0000\_000\_0000000000101\_00\_00000\_0 | P\_LOW\_100UA | Drive low 100μA |
| %0000\_0000\_000\_0000000000110\_00\_00000\_0 | P\_LOW\_10UA | Drive low 10μA |
| %0000\_0000\_000\_0000000000111\_00\_00000\_0 | P\_LOW\_FLOAT | Float low |
| **DIR/OUT Control** | **(pick one)** |  |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_TT\_00 (default) | TT = %00 |
| %0000\_0000\_000\_0000000000000\_01\_00000\_0 | P\_TT\_01 | TT = %01 |
| %0000\_0000\_000\_0000000000000\_10\_00000\_0 | P\_TT\_10 | TT = %10 |
| %0000\_0000\_000\_0000000000000\_11\_00000\_0 | P\_TT\_11 | TT = %11 |
| %0000\_0000\_000\_0000000000000\_01\_00000\_0 | P\_OE | Enable output in smart pin mode |
| %0000\_0000\_000\_0000000000000\_01\_00000\_0 | P\_CHANNEL | Enable DAC channel in non-smart pin DAC mode |
| %0000\_0000\_000\_0000000000000\_10\_00000\_0 | P\_BITDAC | Enable BITDAC for non-smart pin DAC mode |
| **Smart Pin Modes** | **(pick one)** |  |
| %0000\_0000\_000\_0000000000000\_00\_00000\_0 | P\_NORMAL (default) | Normal mode (not smart pin mode) |
| %0000\_0000\_000\_0000000000000\_00\_00001\_0 | P\_REPOSITORY | Long repository (non-DAC mode) |
| %0000\_0000\_000\_0000000000000\_00\_00001\_0 | P\_DAC\_NOISE | DAC Noise (DAC mode) |
| %0000\_0000\_000\_0000000000000\_00\_00010\_0 | P\_DAC\_DITHER\_RND | DAC 16-bit random dither (DAC mode) |
| %0000\_0000\_000\_0000000000000\_00\_00011\_0 | P\_DAC\_DITHER\_PWM | DAC 16-bit PWM dither (DAC mode) |
| %0000\_0000\_000\_0000000000000\_00\_00100\_0 | P\_PULSE | Pulse/cycle output |
| %0000\_0000\_000\_0000000000000\_00\_00101\_0 | P\_TRANSITION | Transition output |
| %0000\_0000\_000\_0000000000000\_00\_00110\_0 | P\_NCO\_FREQ | NCO frequency output |
| %0000\_0000\_000\_0000000000000\_00\_00111\_0 | P\_NCO\_DUTY | NCO duty output |
| %0000\_0000\_000\_0000000000000\_00\_01000\_0 | P\_PWM\_TRIANGLE | PWM triangle output |
| %0000\_0000\_000\_0000000000000\_00\_01001\_0 | P\_PWM\_SAWTOOTH | PWM sawtooth output |
| %0000\_0000\_000\_0000000000000\_00\_01010\_0 | P\_PWM\_SMPS | PWM switch-mode power supply I/O |
| %0000\_0000\_000\_0000000000000\_00\_01011\_0 | P\_QUADRATURE | A-B quadrature encoder input |
| %0000\_0000\_000\_0000000000000\_00\_01100\_0 | P\_REG\_UP | Inc on A-rise when B-high |
| %0000\_0000\_000\_0000000000000\_00\_01101\_0 | P\_REG\_UP\_DOWN | Inc on A-rise when B-high, dec on A-rise when B-low |
| %0000\_0000\_000\_0000000000000\_00\_01110\_0 | P\_COUNT\_RISES | Inc on A-rise, optionally dec on B-rise |
| %0000\_0000\_000\_0000000000000\_00\_01111\_0 | P\_COUNT\_HIGHS | Inc on A-high, optionally dec on B-high |
| %0000\_0000\_000\_0000000000000\_00\_10000\_0 | P\_STATE\_TICKS | For A-low and A-high states, count ticks |
| %0000\_0000\_000\_0000000000000\_00\_10001\_0 | P\_HIGH\_TICKS | For A-high states, count ticks |
| %0000\_0000\_000\_0000000000000\_00\_10010\_0 | P\_EVENTS\_TICKS | For X A-highs/rises/edges,  count ticks /  Timeout on X ticks of no A-high/rise/edge |
| %0000\_0000\_000\_0000000000000\_00\_10011\_0 | P\_PERIODS\_TICKS | For X periods of A, count ticks |
| %0000\_0000\_000\_0000000000000\_00\_10100\_0 | P\_PERIODS\_HIGHS | For X periods of A, count highs |
| %0000\_0000\_000\_0000000000000\_00\_10101\_0 | P\_COUNTER\_TICKS | For periods of A in X+ ticks, count ticks |
| %0000\_0000\_000\_0000000000000\_00\_10110\_0 | P\_COUNTER\_HIGHS | For periods of A in X+ ticks, count highs |
| %0000\_0000\_000\_0000000000000\_00\_10111\_0 | P\_COUNTER\_PERIODS | For periods of A in X+ ticks, count periods |
| %0000\_0000\_000\_0000000000000\_00\_11000\_0 | P\_ADC | ADC sample/filter/capture, internally clocked |
| %0000\_0000\_000\_0000000000000\_00\_11001\_0 | P\_ADC\_EXT | ADC sample/filter/capture, externally clocked |
| %0000\_0000\_000\_0000000000000\_00\_11010\_0 | P\_ADC\_SCOPE | ADC scope with trigger |
| %0000\_0000\_000\_0000000000000\_00\_11011\_0 | P\_USB\_PAIR | USB pin pair |
| %0000\_0000\_000\_0000000000000\_00\_11100\_0 | P\_SYNC\_TX | Synchronous serial transmit |
| %0000\_0000\_000\_0000000000000\_00\_11101\_0 | P\_SYNC\_RX | Synchronous serial receive |
| %0000\_0000\_000\_0000000000000\_00\_11110\_0 | P\_ASYNC\_TX | Asynchronous serial transmit |
| %0000\_0000\_000\_0000000000000\_00\_11111\_0 | P\_ASYNC\_RX | Asynchronous serial receive |
|  |  |  |

# Digitial Pin Operation (Smart Pin Off)



The “Drive” Logic block is broken into the following components:



The resistors/current sources are a kind of drive strength. To get a pullup, you need to set the pin to output with a drive strength of 1.5k or 15k or 150k Ohm for HHH and you need to output a High (OUTx=1). Also if the pin is set to output, you still can read the input.  
If the pin is in smartmode, the red labels are valid, DIRx and OUTx do no longer control the IO signals.

**D = %AAAA\_BBBB\_FFF\_MMMMMMMMMMMMM\_TT\_SSSSS\_0**

AAAA = 0000 “x000 this pin read state”

BBBB = 0000 “x000 this pin read state”

FFF = 000 “A,B default filter

MMMMMMMMMMMMM = 000000000000 “Fast”

TT = 00 “can’t figure out what table says

SSSSS = 00000 “smart pin off”

By default Propeller II starts with ( 'WRPIN #0,pin') Fast Logic Mode

**D = %0000\_0000\_000\_0000000000000\_00\_00000\_0**

If you're familiar with the assembly-language input-output instructions for the Propeller-1 microcontroller you will recognize the following six instructions a Propeller-2 program also may use these registers. These registers give you direct access to I/O pins:

DIRA direction register pins P0..P31, 1= output, 0 = disable output

DIRB direction register pins P63-P32, 1= output, 0 = disable output

OUTA output register bits for pins P0..P31

OUTB output register bits for pins P32..P63

INA input register bits for pins P0..P31

INB input register bits for pins P32..P63

Propeller II does not have instructions to above registers as propeller 1 DIRA[0]~~ but the registers can be accessed with:

mov dira,#0  
mov reg1,ina  
mov outa,#1

Definition of PinField

**{#}D = PinField** = 11 bits %LLLLL\_PPPPPP %extrapins\_basepins

The above direction and output registers can be affected using special instructions which operate on 1 to 32 bits within each register.

In the following lists, {#}D denotes an 11-bit value,(PinField) with the 6 lower bits pointing to a base pin and the next upper 5 bits expressing an additional number of pins within the same I/O register.

**PinField** = 11 bits %LLLLL\_PPPPPP %extrapins\_basepins

11 bits for PinField LLLLL 5bits for number of additional pins PPPPPP 6 bits for base pin number

%00011\_000101 base pin 5 plus 3 pins P3 P4 P5 P6 total 4 pins

%11111\_0010000 base pin 8 plus 31 pins wrapping occurs P8-P31 plus P0-P7 (P0-P31)

PinField ≔ BasePin addpins Add\_pins eg. PinField ≔ 0 addpins 7 ‘ P0-P7 assigned

In these instructions, bit 5 of {#}D selects between DIRA/DIRB or OUTA/OUTB.

10-09-08-07-06\_\_05-04-03-02-01-00

16 08 04 02 01 \_\_32 16 08 04 02 01

The ADDPINS operator can be used to set the additional-bits field in {#}D as follows:

DIRH #8 ‘Drive P8 high

DIRH #10 ADDPINS 7 'Drive P10..P17 high {#}D = 00111\_001010 = LLLLL\_PPPPPP

Each cog has its own pairs of 32-bit I/O Direction Registers (DIRA & DIRB) and 32-bit I/O Output Registers (OUTA & OUTB) to influence the directions and output states of the Propeller 2’s 64 I/O pins. A cog's desired I/O directions and output states are communicated through the entire cog collective to ultimately become what is applied to the I/O pins.  
  
The result of this I/O pin wiring configuration can easily be described in the following simple rules:  
\* A pin is an input only if no active cog sets it to an output.   
\* A pin outputs low only if all active cogs that set it to output also set it to low.   
\* A pin outputs high if any active cog sets it to an output and also sets it high.

The Propeller 2 is a CMOS device, so the I/O pin digital logic threshold is approximately 1/2 Vdd.  
With the Propeller 2's I/O pins powered by 3.3 V (via the corresponding Vxxyy pins), the I/O pin digital logic threshold is about 1.65 V.   
An input pin will interpret a voltage below 1.65 V as a digital logic level low, and will interpret a voltage above 1.65 V as a digital logic level high.   
An output pin will produce 0 V for digital low and 3.3 V for digital high.

DIRL {#}D Set direction bit(s) to logic 0 (input)

DIRH {#}D Set direction bit(s) to logic 1 (output)

DIRC {#}D Set direction bit(s) to Carry flag

DIRNC {#}D Set direction bit(s) to inverse of Carry flag

DIRZ {#}D Set direction bit(s) to Zero flag

DIRNZ {#}D Set direction bit(s) to inverse of Zero flag

DIRRND {#}D Set direction bit(s) to random state(s)

DIRNOT {#}D Invert direction bit(s)

Example: DIRL #20 'Set P20 as an input pin

## 2.1) PASM Pin Digital Commands

### 2.1.1)Pin-Output Instructions

These Instructions change the associated OUT bit(s)

OUTL {#}D Set output bit(s) to logic 0

OUTH {#}D Set output bit(s) to logic 1

OUTC {#}D Set output bit(s) to Carry flag

OUTNC {#}D Set output bit(s) to inverse of Carry flag

OUTZ {#}D Set output bit(s) to Zero flag

OUTNZ {#}D Set output bit(s) to inverse of Zero flag

OUTRND {#}D Set output bit(s) to random state(s)

OUTNOT {#}D Invert output bit(s)

Example: OUTNOT $20 'Invert the logic state of the P20 output

### 2.1.2)Pin-Float Instructions

These instructions change the asscoiaed DIR bit(s) to logic-0(input float)

FLTL {#}D Set output bit(s) to logic 0

FLTH {#}D Set output bit(s) to logic 1

FLTC {#}D Set output bit(s) to Carry flag

FLTNC {#}D Set output bit(s) to inverse of Carry flag

FLTZ {#}D Set output bit(s) to Zero flag

FLTNZ {#}D Set output bit(s) to inverse of Zero flag

FLTRND {#}D Set output bit(s) to random state(s)

FLTNOT {#}D Invert output bit(s)

Example: FLTC #20 'Make P20 input with its output bit set to C.

### 2.1.3) Pin-Drive Instructions

These instructions change the associated DIR bit(s) to logic-1 (output).

DRVL {#}D Set output bit(s) to logic-0

DRVH {#}D Set output bit(s) to logic-1

DRVC {#}D Set output bit(s) to Carry flag value

DRVNC {#}D Set output bit(s) to inverse of Carry flag

DRVZ {#}D Set output bit(s) to Zero flag

DRVNZ {#}D Set output bit(s) to inverse of Zero flag

DRVRND {#}D Set output bit(s) to random state(s)

DRVNOT {#}D Invert output bit(s)

Example: DRVZ #20 'Make P20 output the Z-flag state.

### 2.1.4)Input-Pin Instructions

Two instructions, TESTP and TESTPN can read the state of a single bit within an INA/INB register and either write that bit to the Carry (C) or Zero (Z) flag, or perform a logic operation on the flag. Again, {#}D (pinfield)represents a pin number.

TESTP {#}D WC/WZ Get a pin's state and write it into the C or Z flag.

TESTP {#}D ANDC/ANDZ Get a pin's state and AND it into the C or Z flag.

TESTP {#}D ORC/ORZ Get a pin's state and OR it into the C or Z flag.

TESTP {#}D XORC/XORZ Get a pin's state and XOR it into the C or Z flag.

TESTPN {#}D WC/WZ Get a pin's NOT-state and write it into the C or Z flag.

TESTPN {#}D ANDC/ANDZ Get a pin's NOT-state and AND it into the C or Z flag.

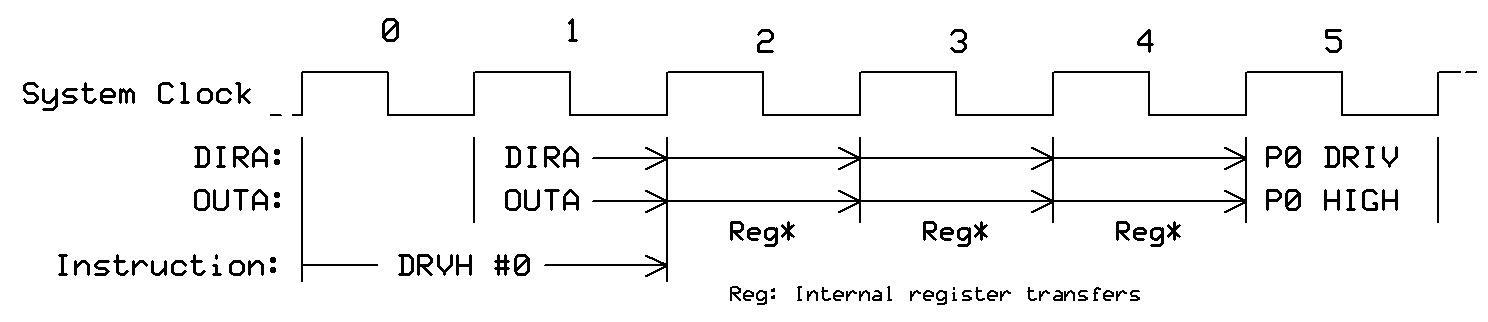
TESTPN {#}D ORC/ORZ Get a pin's NOT-state and OR it into the C or Z flag.

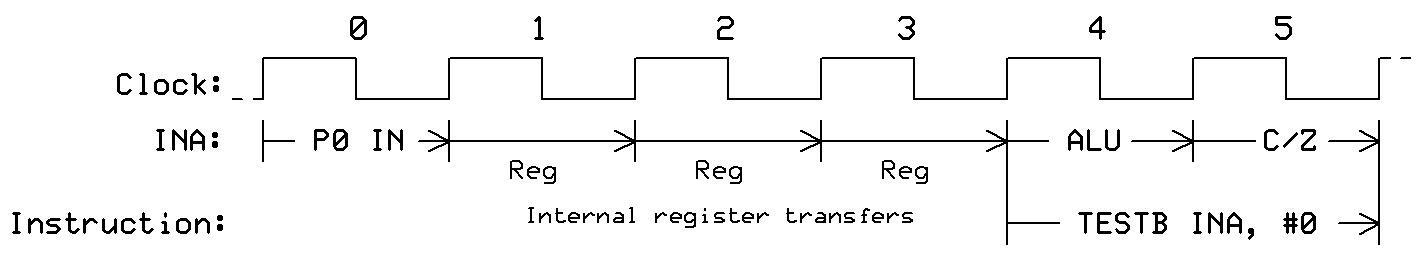
TESTPN {#}D XORC/XORZ Get a pin's NOT-state and XOR it into the C or Z flag.

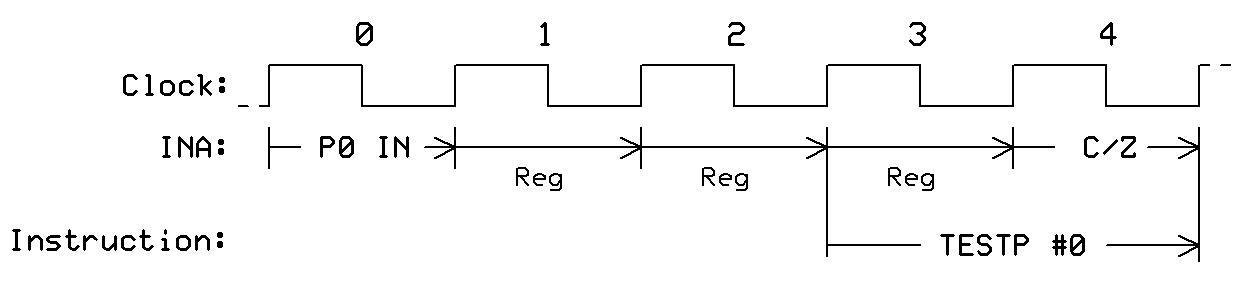
Example: TESTP #10 ORZ 'Read P10 and or its state into Z.

### 2.1.5)Input-Output-Bit Timing

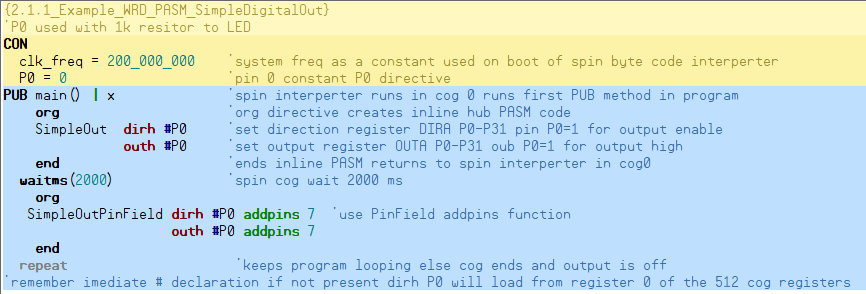
When an instruction changes a DIRx or OUTx bit, the processor needs three (3) additional system-clock cycles *after the instruction* before the pin starts to transition to its new state. The figure below shows the delay for a DRVH instruction:

When an instruction reads the contents of the IN register associated with a pin, the processor receives the state of the pins as they existed three (3) system-clock cycles *before* the start of the instruction. The figure below shows the timing for a the TESTB INA,#0 operation:



When a program uses a TESTP or TESTPN instruction to read the state of a pin, the processor receives the state of the pins as they existed two (2) system-clock cycles *before* the start of the instruction. So, the TESTP and TESTPN gather "fresher" INx data than is available via the INx registers. The figure below shows the timing for a TESTP instruction:

### 2.1.1\_Example\_WRD\_PASM\_SimpleDigitalOut.spin2



### 2.1.2\_Example\_Wrd\_PASM\_DRVH\_DRVL.spin2

{2.1.2\_Example\_WRD\_PASM\_SimpleDigitalOut}

'P0-P7 used with 1k resitor to LED

CON

clk\_freq = 200\_000\_000 'system freq as a constant used on boot of spin byte code interperter

P0 = 0 'pin 0 constant P0 directive

PUB main() | x 'spin interperter runs in cog 0 runs first PUB method in program

org 'org directive creates inline hub PASM code

SimpleDRVH drvh #P0 'set direction register DIRA P0 pin P0=1 and OUTA P0 = 1

end 'ends inline PASM returns to spin interperter in cog0

waitms(2000) 'spin cog wait 2000 ms

org

SimpleDRVHPinField drvh #P0 addpins 7

'set direction DIRA register P0-P7 %11111111 and OUTA P0-P7 %11111111

end

repeat 'keeps program looping else cog ends and output is off

'remember imediate # declaration if not present dirh P0 will load from register 0 of the 512 cog registers

'DrivePin DRVH command sets DIRA\DIRB output enable 1 and the OUTA\OUTB output to command L/H ‘according to addpins

### 2.1.3\_Example\_WRD\_PASM\_Cog\_Assembly

{{2.1.3\_Exammple\_WRD\_PASM\_Cog\_Assembly}}

''Debug must be enabled in propeller tool

{===================================================================================}

CON {Processor Timing}

\_clkfreq = 200\_000\_000

P0 = 0

PUB main()

COGINIT(COGEXEC\_NEW,@blink01,0) 'returns cog number started 1 PTRA set to 0

repeat 'processor clock speed

DAT ORG 0 'Cog1 Blink

blink01 OR DIRA, #$FF 'set the direction of the first 8 pins to Output

OR OUTA, #P0 + 1 'set P0 bit 0 equal 1 for High

GETCT cog1CountValue 'the counter value is now in in cog1CountValue

ADDCT1 cog1CountValue,cog1WaitTime 'add counter tick for delay

'the cog1CountValue + cog1WaitTime = CT1 result is placed in the CT1 event register

Loop01 WAITCT1 'wait for cog1 program counter to reach CT1

ADDCT1 cog1CountValue,cog1WaitTime 'add counter tick for delay

XOR OUTA, #1 'toggle bit 1 exclusive or

AND cog1MskINA,INA 'mask of bits not being used

debug(ubin(cog1MskINA)) ‘debug interrupt window to display value

MOV cog1MskINA,#%00000000\_00000000\_00000000\_00000001

JMP #Loop01

'------------------------------------------

cog1WaitTime long 150\_000\_000

cog1CountValue long 200\_000\_000

cog1MskINA long %00000000\_00000000\_00000000\_00000001

### 2.1.4\_Example\_WRD\_PASM\_Cog\_Assembly

{{2.1.4\_Example\_WRD\_PASM\_Cog\_Assembly}}

''Debug must be enabled in propeller tool

{===================================================================================}

CON {Processor Timing}

\_clkfreq = 200\_000\_000

P0 = 0

P1 = 1

PUB main()

COGINIT(COGEXEC\_NEW,@blink00,0) 'returns cog number started 1 PTRA set to 0

COGINIT(COGEXEC\_NEW,@blink01,0) 'returns cog number started 1 PTRA set to 0

repeat

DAT ORG 0 'Cog1 Blink

blink00 OR DIRA, #$FF 'Set the direction of the first 8 pins to Output

MOV OUTA, #%01 'set P0 bit 0 equal 1 for High

GETCT cog1CountValue 'the counter value is now in in cog1CountValue

ADDCT1 cog1CountValue,cog1WaitTime1 'add counter tick for delay

'the cog1CountValue + cog1WaitTime = CT1 result is placed in the CT1 event register

Loop01 WAITCT1 'wait for cog1 program counter to reach CT1

ADDCT1 cog1CountValue,cog1WaitTime1 'add counter tick for delay

XOR OUTA, #%01 'toggle bit 1 exclusive or

JMP #Loop01

'------------------------------------------

cog1WaitTime1 long 100\_000\_000

cog1CountValue long 200\_000\_000

'------------------------------------------

DAT ORG 0 'Cog1 Blink

blink01 OR DIRA, #$FF 'Set the direction of the first 8 pins to Output

MOV OUTA, #%10 'set P0 bit 0 equal 1 for High

GETCT cog2CountValue 'get cog counter value

'the counter value is now in in cog1CountValue

ADDCT1 cog2CountValue,cog2WaitTime1 'add counter tick for delay

'the cog1CountValue + cog1WaitTime = CT1 result is placed in the CT1 event register

Loop02 WAITCT1 'wait for cog1 program counter to reach CT1

ADDCT1 cog2CountValue,cog2WaitTime1 'add counter tick for delay

XOR OUTA, #%10 'toggle bit 1 exclusive or

JMP #Loop02

'------------------------------------------

cog2WaitTime1 long 200\_000\_000

cog2CountValue long 200\_000\_000

## 2.2) I/O Digital Spin Methods

**PinField = 11 bits %LLLLL\_PPPPPP %extrapins\_basepins**

11 bits for PinField LLLLL 5bits for number of additional pins PPPPPP 6 bits for base pin number

%00011\_000101 base pin 5 plus 3 pins P3 P4 P5 P6 total 4 pins

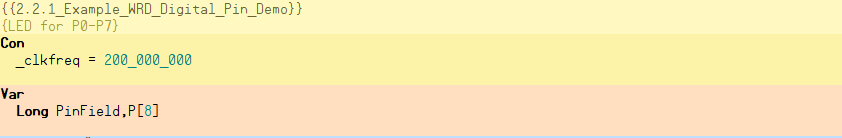
%11111\_0010000 base pin 8 plus 31 pins wrapping occurs P8-P31 plus P0-P7 (P0-P31)

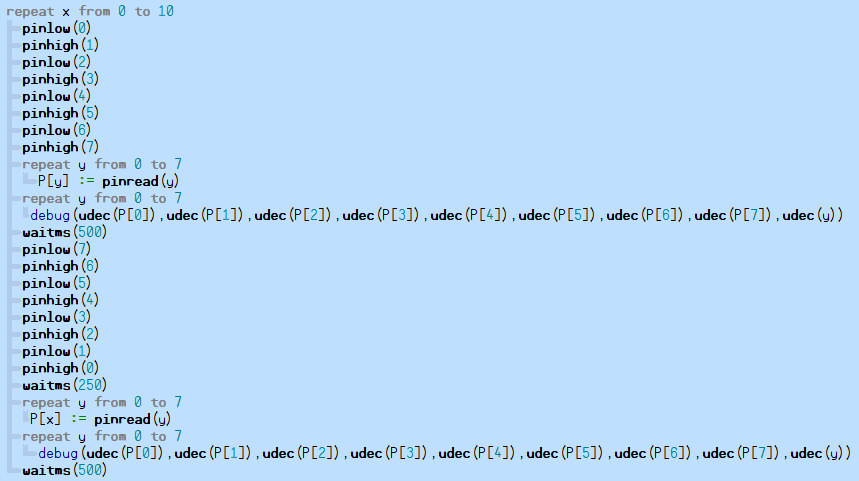
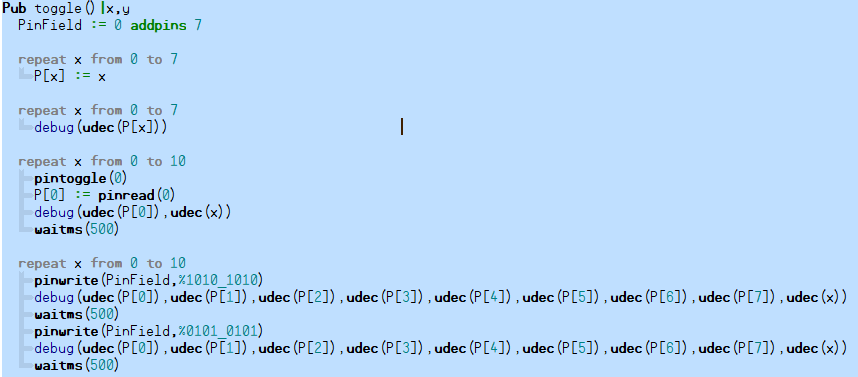
PinField ≔ BasePin addpins Add\_pins eg. PinField ≔ 0 addpins 7 ‘ P0-P7 assigned

**The following are spin 2 commands:**

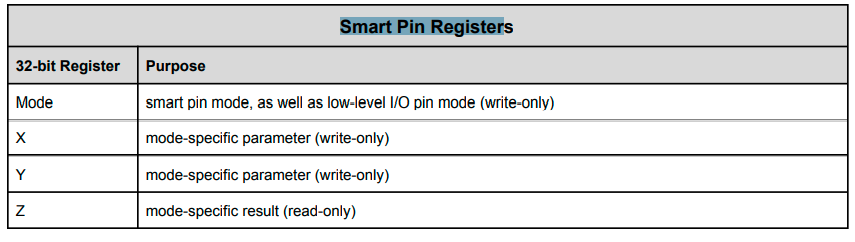
|  |  |
| --- | --- |
| **Pin Methods** | **Details** |
| **PINW | PINWRITE(PinField, Data)** | Drive PinField pin(s) with Data |
| **PINL | PINLOW(PinField)** | Drive PinField pin(s) low |
| **PINH | PINHIGH(PinField)** | Drive PinField pin(s) high |
| **PINT | PINTOGGLE(PinField)** | Drive and toggle PinField pin(s) |
| **PINF | PINFLOAT(PinField)** | Float PinField pin(s) |
| **PINR | PINREAD(PinField) : PinStates** | Read PinField pin(s) |
| **PINSTART(PinField, Mode, Xval, Yval)** | Start PinField smart pin(s): DIR=0, then WRPIN=Mode, WXPIN=Xval, WYPIN=Yval, then DIR=1 |
| **PINCLEAR(PinField)** | Clear PinField smart pin(s): DIR=0, then WRPIN=0 |
| **WRPIN(PinField, Data)** | Write 'mode' register(s) of PinField smart pin(s) with Data |
| **WXPIN(PinField, Data)** | Write 'X' register(s) of PinField smart pin(s) with Data |
| **WYPIN(PinField, Data)** | Write 'Y' register(s) of PinField smart pin(s) with Data |
| **AKPIN(PinField)** | Acknowledge PinField smart pin(s) |
| **RDPIN(Pin) : Zval** | Read Pin smart pin and acknowledge, Zval[31] = C flag from RDPIN, other bits are RDPIN data |
| **RQPIN(Pin) : Zval** | Read Pin smart pin without acknowledge, Zval[31] = C flag from RQPIN, other bits are RQPIN data |

### 2.2.1\_Example\_WRD\_Digital\_Pin.spin2



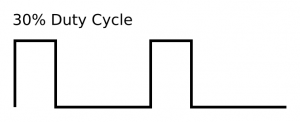


# PWM Pulse Width Modulation with Smart Pin



The mechanism typically used to control the brightness of an LED is called PWM (Pulse Width Modulation). In our blink example the LED was either always on or always off. If we want an intermediate brightness, we need to have it partially on; this is the purpose of PWM.

In this figure the on-time portion of the waveform is 30% of the entire cycle (on-time plus off-time). The ratio of on-time to cycle-time is called the duty cycle.

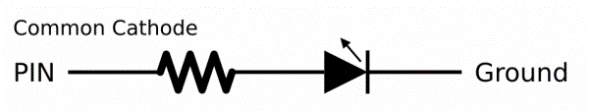
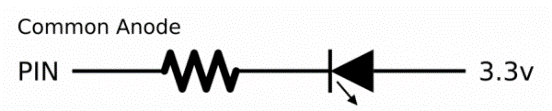


**pinstart(led, m, x, y)**

m := P\_PWM\_SAWTOOTH | P\_OE

|  |  |  |
| --- | --- | --- |
| %0000\_0000\_000\_0000000000000\_00\_01001\_0 | P\_PWM\_SAWTOOTH | PWM sawtooth output |
| %0000\_0000\_000\_0000000000000\_01\_00000\_0 | P\_OE | Enable output in smart pin mode |

The first step is to select the PWM mode (sawtooth is the easiest to implement) and to make the smart pin an output with the P\_OE constant. The output enable flag is required because in smart pin mode, the pin direction bit is used to enable or disable the smart pin.



If the Common Cathode connection is used the output being high will turn on the LED if the Common Anode connection is used the output being high will turn off the LED.

The output can be inverted by setting the P\_INVERT\_OUTPUT bit in the mode register:

m |= P\_INVERT\_OUTPUT

|  |  |  |
| --- | --- | --- |
| %0000\_0000\_000\_0000001000000\_00\_00000\_0 | P\_INVERT\_OUTPUT | Select inverted output |

x.word[1] := 255

The high word of the smart pin X register holds the value that will set the output to 100% duty cycle. As discussed, we will use 255.

x.word[0] := 1 #> ((clkfreq / hz) / 255) <# $FFFF

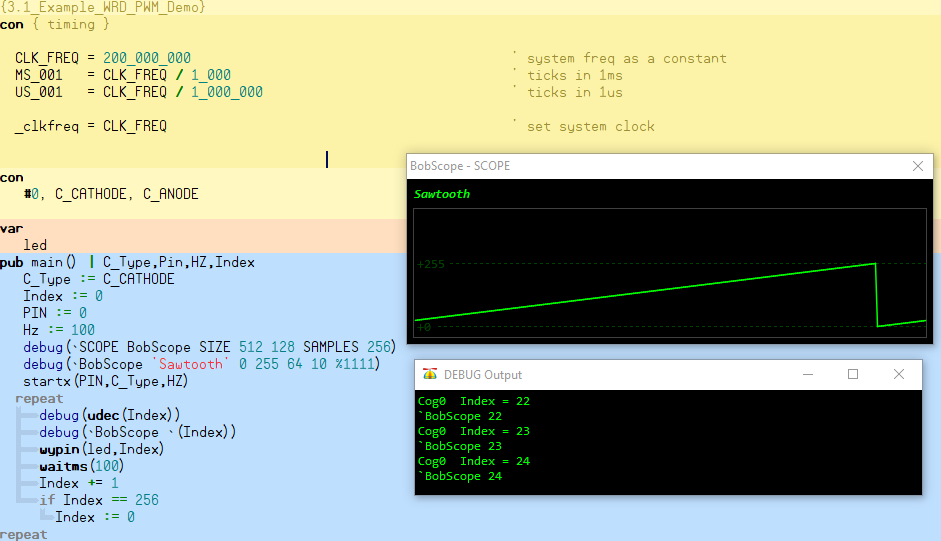
Finally, the low word of the smart pin X register holds the number of system ticks in one unit for the desired PWM frequency. This takes a little bit of math, but, again, is fairly straightforward.

It works out like this: the system clock frequency (clkfreq) is divided by the desired PWM frequency (hz); this gives us the number of system ticks in one PWM period. That is divided by the number of units in 100% (255) to get the number of system ticks in one unit. The #> and <# operators constrain the value to a legal 16-bit number for the low word of X.

Y register which holds the current level; in our setup this will be 0 (0%) to 255 (100%). To change the LED brightness at any time we can write to the smart pin Y register like this:

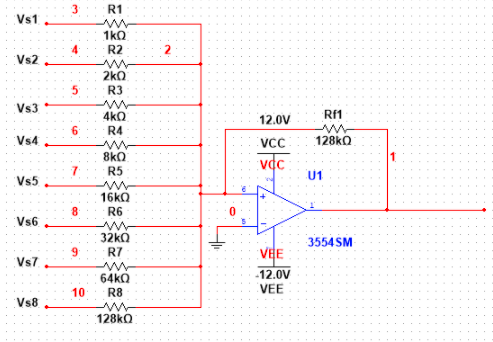
wypin(LED, 128) this sets 50% duty cycle

## 3.1\_Example\_ WRD\_PWM\_Demo.spin2

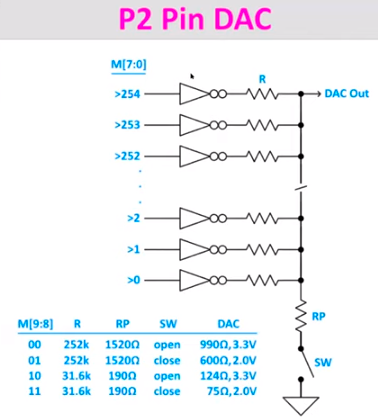


# Analog Out Smart Pin(DAC)

## 4.1) DAC Digital to Analog Conversion



The above Schematic is the standard Method for Dac P2 Uses a Voltage Divider approach:



The above schematic is how the propeller II implements a Dac using a voltage divider

Propeller smart pins each have 2 Dac’s , one DAC has 255 resistors of 252K and the other DAC has 255 resistors of 31.6k. The resistors are pulled high or low dependant on the magnitude of the value essentially an 8 bit dac is created. If all resistors are pulled high the Voltage is 100% or value 255 (3.3v) if the value is 128 ($80) the voltage is 50% half resistors high half low. If 0 is used all resistors pulled low. 255 resistors in parallel would be 252K/255 Plus the driver circuit impedance aprox 990 ohm. For video on DAC see: <https://www.youtube.com/c/ParallaxInc/playlists> The essence of this dac is a voltage divider network that is set by a clocked Flip Flops.

1) Variable “pin” may be a PinField need to make sure only 1 pin is configured.

PinField = lllll\_pppppp lllll = 5 bits for addpins pppppp = 6 bits for P0-P63 0-63

let pin = lllll\_pppppp

0000\_0000\_0000\_0000\_0000\_0lll\_llpp\_pppp = pin

0000\_0000\_0000\_0000\_0000\_0000\_ 0011\_1111 = $3F

**pin &= $3F ‘include this instruction to clear llll upper addpins**

0000\_0000\_0000\_0000\_0000\_0000\_00pp\_pppp = pin

2) Disable analog smart pin if previously configured

**pinclear(Pin) ' disable smart pin**

3) Following Spin method sets Pin for Digital output

pinstart(pin, P\_DAC\_DITHER\_PWM | P\_DAC\_990R\_3V | P\_OE, 256, 0)' 16-bit dac

-- <https://docs.google.com/document/d/16qVkmA6Co5fUNKJHF6pBfGfDupuRwDtf-wyieh_fbqw/edit#heading=h.1h0sz9w9bl25>

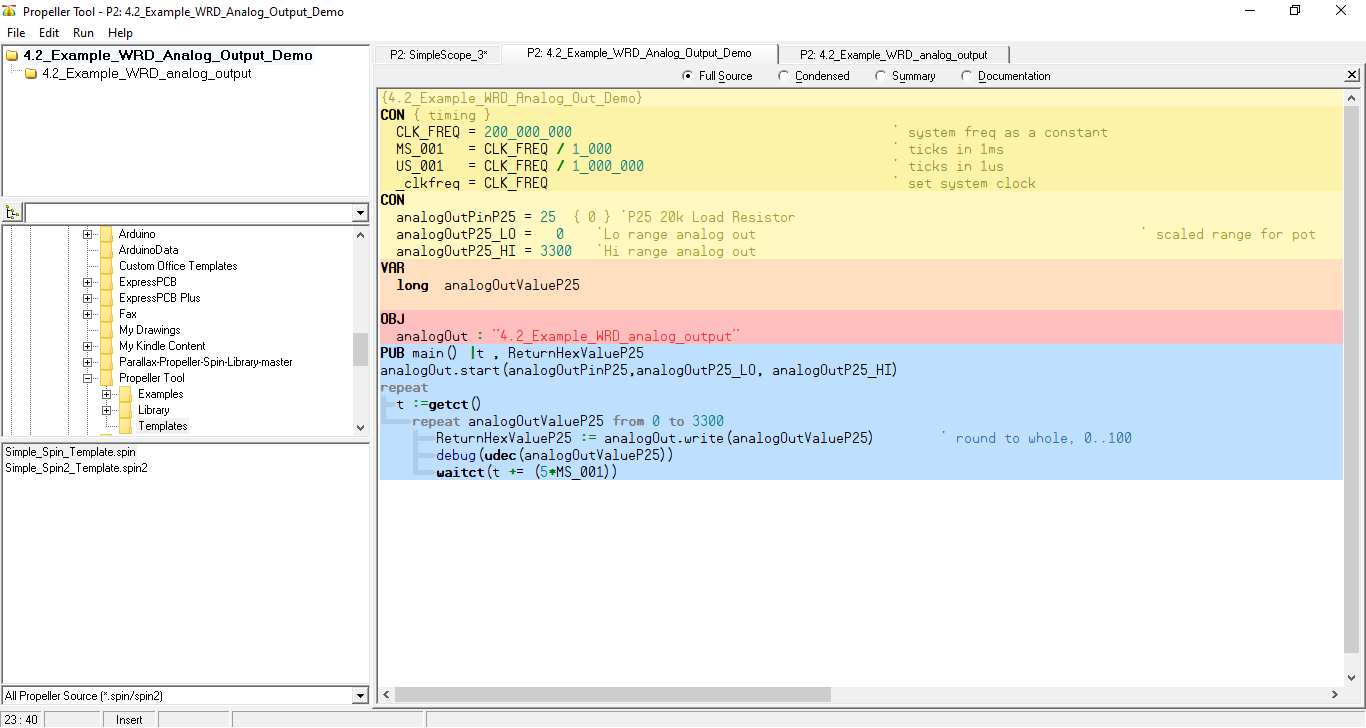
**Built-In Symbols for Smart Pin Configuration**; use Ctrl+F for searching strings

|  |  |  |
| --- | --- | --- |
| %0000\_0000\_000\_0000000000000\_00\_00011\_0 | P\_DAC\_DITHER\_PWM | DAC 16-bit PWM dither (DAC mode) |

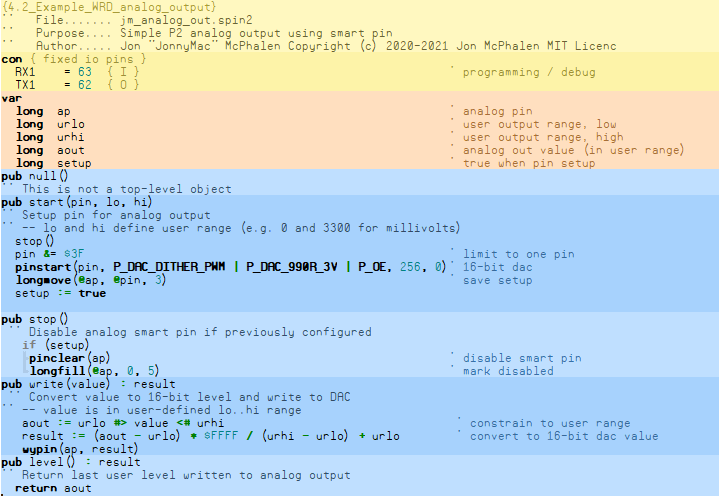
|  |  |  |
| --- | --- | --- |
| %0000\_0000\_000\_1010000000000\_00\_00000\_0 | P\_DAC\_990R\_3V | DAC 990Ω, 3.3V peak, ADC 1x → IN |

|  |  |  |
| --- | --- | --- |
| %0000\_0000\_000\_0000000000000\_01\_00000\_0 | P\_OE | Enable output in smart pin mode |

## 4.2\_Example\_WRD\_Analog\_Out\_Demo



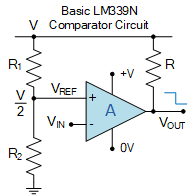
## 4.2\_Example\_WRD\_Analog\_Output



# Analog Input Smart Pin (ADC)

## 5.1) ADC Analog Digital Conversion

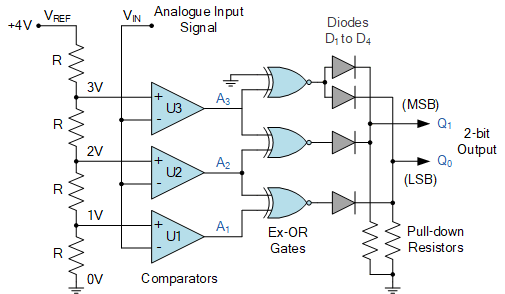
**Comparator Circuit**



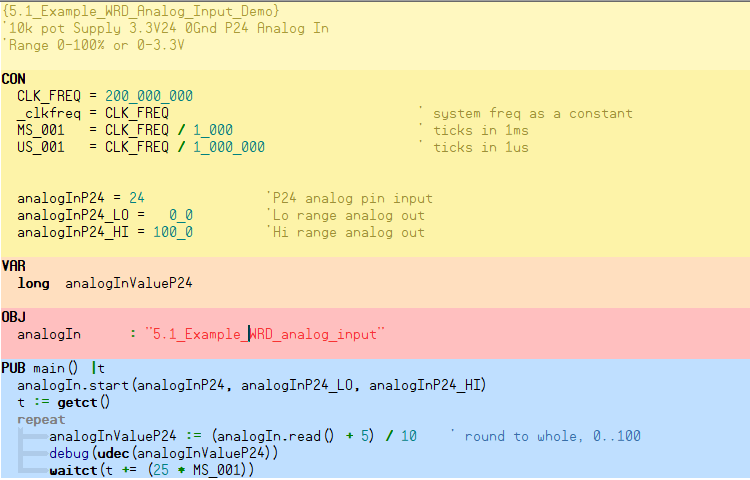
An analogue comparator such as the LM339N which has two analogue inputs, one positive and one negative, and which can be used to compare the magnitudes of two different voltage levels. A voltage input, (VIN) signal is applied to one input of the comparator, while a reference voltage, (VREF) to the other. A comparison of the two voltage levels at the comparator’s input is made to determine the comparators digital logic output state, either a “1” or a “0”.

The reference voltage, VREF is compared against the input voltage, VIN applied to the other input. For an LM339 comparator, if the input voltage is less than the reference voltage, (VIN < VREF) the output is “OFF”, and if it is greater than the reference voltage, (VIN > VREF) the output will be “ON”. Thus a comparator compares two voltage levels and determines which one of the two is higher.

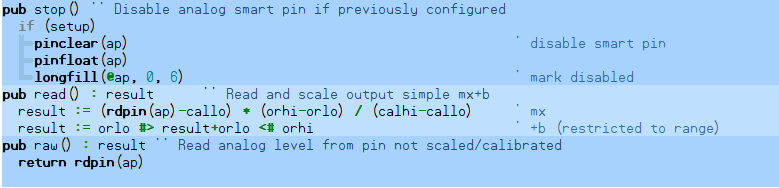
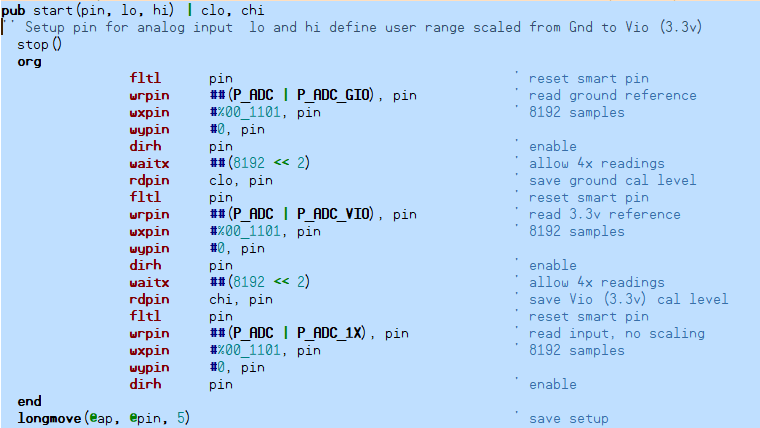
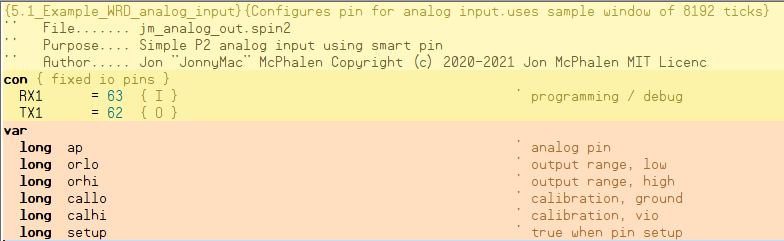
2-bit ADC Using Diodes



## 5.1\_Example\_WRD\_Analog\_Input\_Demo



## 5.2\_Example\_WRD\_Analog\_Input



**PINCLEAR(PINFIELD)** Clear PinField smart pin(s): DIR=0, then WRPIN=0

**PINFLOAT(PinField)** Float PinField pin(s)

**LONGMOVE(Dest, Source, Count)** Move Count longs from Source to Dest

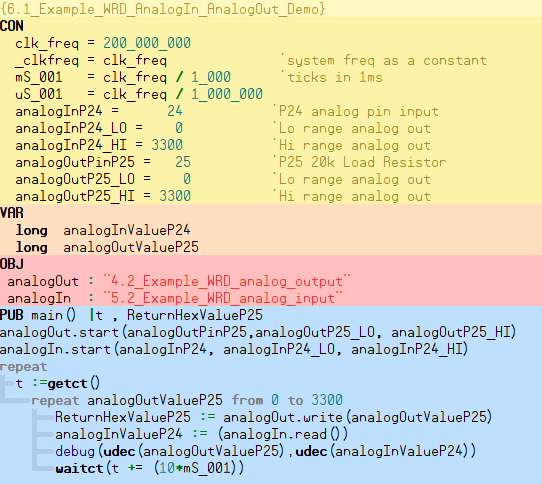
**LONGFILL(Dest, Value, Count)** Fill Count longs at Dest with Value

**RDPIN(Pin) :Zval** Read Pin smart pin and acknowledge, Zval[31] = C flag from RDPIN, other bits are RDPIN Data

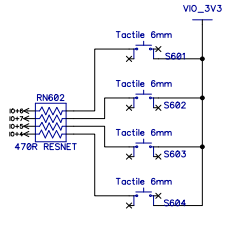
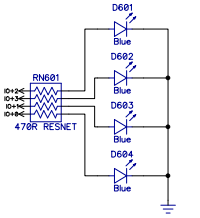
# AnalogIn and AnalogOut Demo

Using P24 as an analog Input fed from P25 as an analog output a 25 K load resistor is tied to Pins and ground.

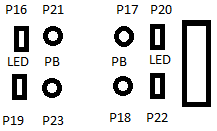
## 6.1\_Example\_WRD\_AnalogIn\_AnalogOut\_Demo



# 7.0) P2 Eval PB/LED Control Add-on Board (64006-ES)



Led\_P16 Led\_P17 Led\_P18 Led\_P19 Pb\_P20 Pb\_21 Pb\_22 Pb\_23 Pb-24

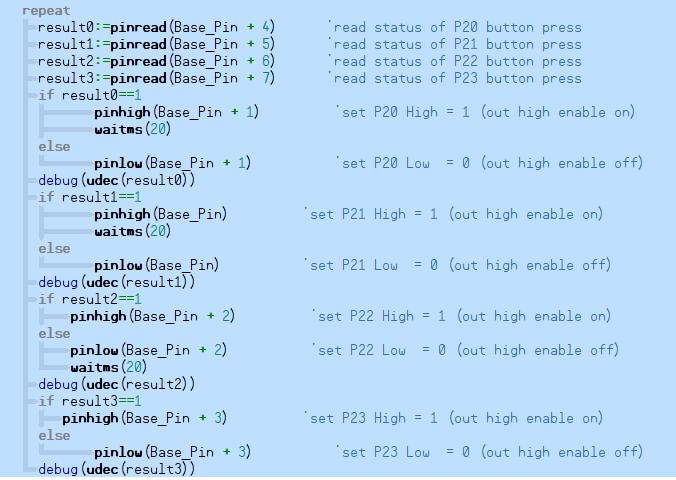
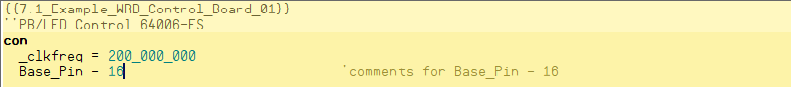


wrpin (20 , P\_LOW\_15K) 'select P20 pull-down enable sets pin as a low

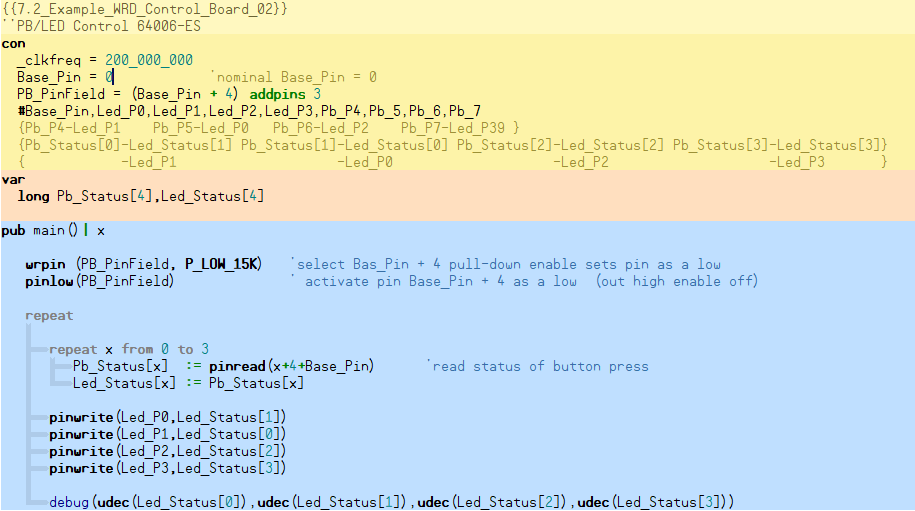
result0:=pinread(20) 'read status of P20 button press

pinhigh(17) 'set P17 led High = 1 (out high enable on)

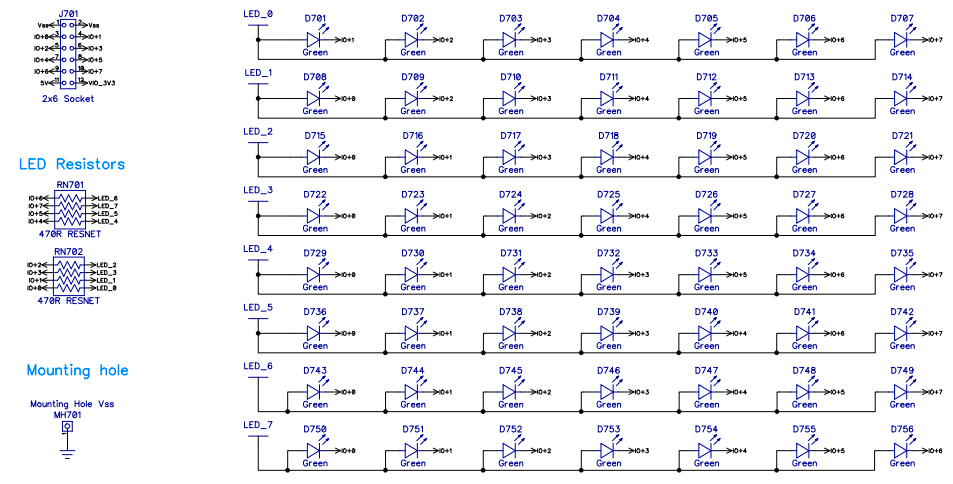
## 7.1\_Ecample\_WRD\_Control\_Board\_01



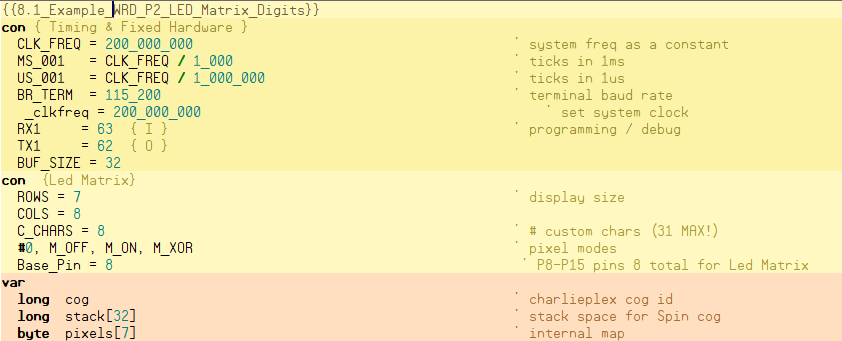
## 7.2\_Example\_WRD\_Control\_Board\_02

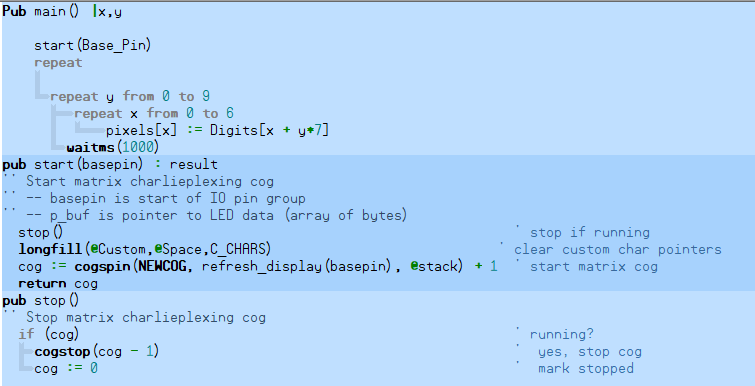


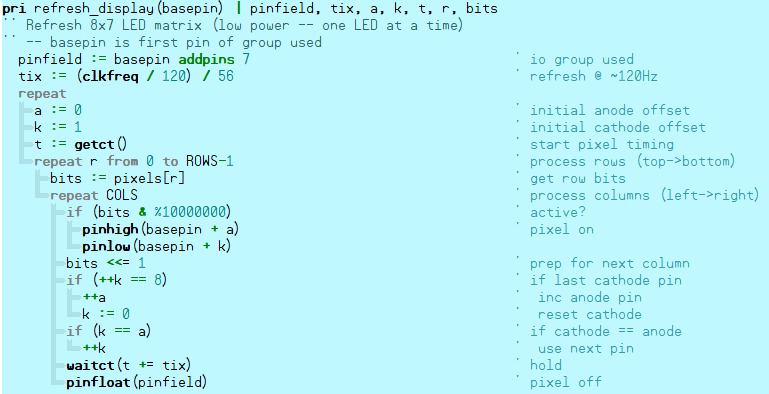
# 8.0) P2 Eval LED Matrix Add-on Board (#64006C)

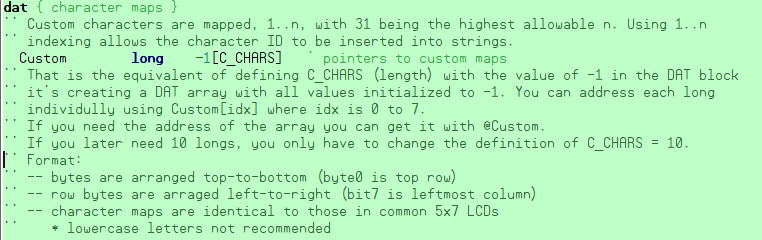


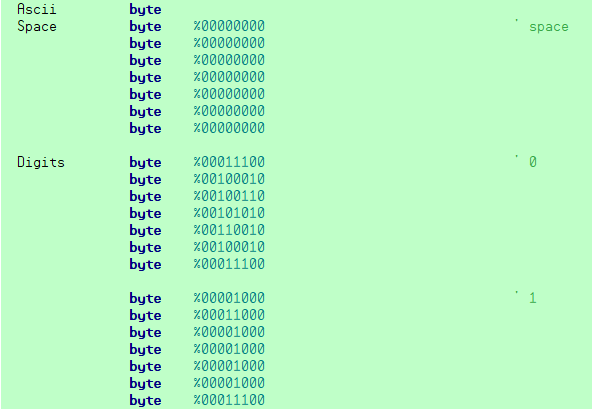
## 8.1\_Example\_WRD\_P2\_LED\_Matrix\_Digits

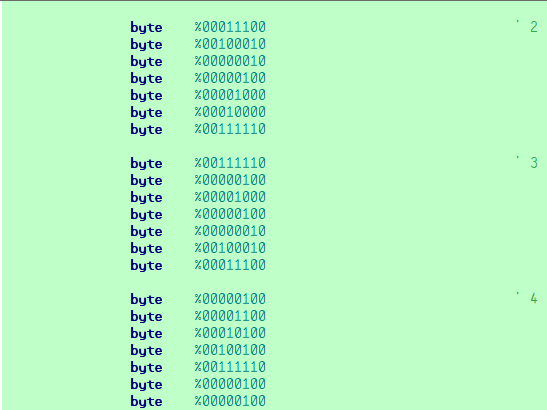


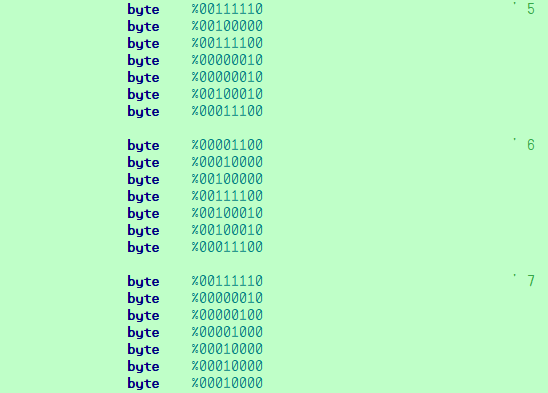


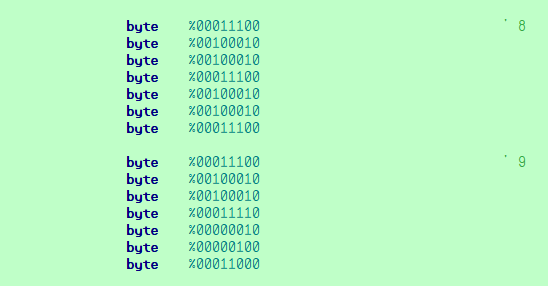






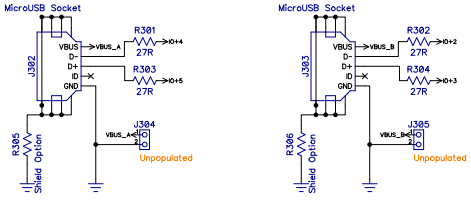


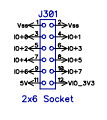




# 9.0) P2 Eval Serial Device Add-on Board SKU 64006F





Two user controlled activity LEDs (red and blue) are located beside each microUSB-type socket. Copyright © Parallax Inc. P2 Eval Add-on Boards (#64006 Series)

**Function**

0 Blue LED with 1 kΩ series resistor. Assert high to light.

1 Red LED with 1 kΩ series resistor. Assert high to light.

2 Serial channel 1 : Data D-

3 Serial channel 1 : Data D+

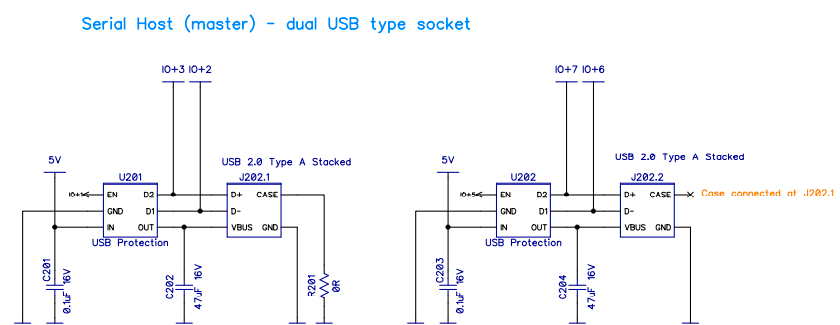
4 Serial channel 2 : Data D-

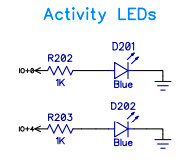
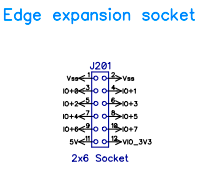
5 Serial channel 2 : Data D+

6 Blue LED with 1 kΩ series resistor. Assert high to light.

7 Red LED with 1 kΩ series resistor. Assert high to light

# 10.0) P2 Eval Serial Host Add-on Board SKU 64006B





## 10.1) USB host overview

The Universal **Serial** Bus, or **USB**, is an external **port** that interfaces between external **devices** and a computer.

When a port is in USB host mode, it powers the bus, and enumerates connected USB devices.

* ***Host:***   The host is the computer or item that acts as the main element or controller for the USB system. The host has a hub contained within it and this is called the Root Hub.
* ***Hub:***   The hub is a device that effectively expands the number of ports available - it will have one connection to the upstream connection, and several downstream. It is possible to plug one hub into another to expand the capability and connectivity further.
* ***Port:***   This is the socket through which access to the USB network is gained. It can be on a host, or a hub.
* ***Function:***   These are the peripherals or items to which the USB link is connected. Mice, keyboards, Flash memories, etc, etc.
* ***Device:***   This term is collectively used for hubs and functions.

## 10.2) Selecting USB function destination

With data for all devices beings sent along the bus, it is necessary for the USB operation that the data is only accepted by the required function.

To achieve this, when a device is attached to the bus it is assigned a unique number or address by the host for the time it is connected.

In addition to the address, the device also contains endpoints. These are the actual sources and destinations for communications between the host and the device.

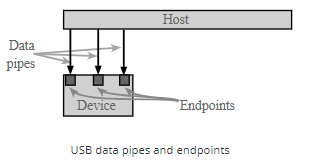
Endpoints can only operate in one direction, i.e. input or output, but not both, and devices can have up to 16, of which one each for the input and output must be reserved as the 'Zero Endpoint' for that direction. Although each device can have sixteen input and sixteen output endpoints, it is very rare for them all to be used.

The zero endpoints are used for a variety of activities including auto-detection and configuration of the device on the bus and the two zero endpoints are the only ones accessible until the device is properly connected on the bus.

## 10.3) USB data pipes

The communication within USB is based around the concept of using data pipes. These can be considered as being logical channels within the data flow on the bus.

In reality, a USB data pipe is a connection from the host controller to a logical entity within a device, i.e. the endpoint. Because pipes correspond to endpoints, the terms are sometimes used interchangeably.



The host then uses the concept of a data pipe to ensure the data to and from a device is correctly directed or the source is known. The data pipe uses a combination of the address, endpoint and also the direction to define it.

To communicate with the zero endpoints a special form of data pipe is needed because it needs to be used to establish the initial communication. It is called the Default Control Pipe and it can be used when the initial physical connection is made.

There are two types of USB pipe:

* ***Message Pipe :***   This type is a bi-directional USB pipe and it is used for control data. Message pipes are typically used for short, simple commands to the device, and for status responses from the device. They can be used by the bus control pipe number 0.
* ***Stream Pipe:***   This form of USB pipe is uni-directional and it is connected to a uni-directional endpoint that transfers data using an isochronous, interrupt, or bulk transfers (see below).
* h

## 10.4) USB signalling and data transfer basics

For USB 1 and 2 a four wire system is employed. As detailed elsewhere, the cables carry: power, ground and then there is a twisted pair for the differential data transfer.

The lines are designated Data+, D+ and Data-, D- for USB 1 and USB 2. For USB 3, new lines were introduced. For each port there are TX1+ & TX1- and TX2+ & TX2- to cover the transmitted data and then for the received data the lines are RX1+ & RX1- and RX2+ & RX2-.

The use of twisted pairs and differential signaling reduces the effects of external interference that may picked up. It also reduces the effect of any hum loops, etc that could cause issues. As it is not related to ground, but the difference between the two lines, the effects of hum are significantly reduced

The data uses an NRZI system, i.e. non-return to zero.In terms of operation, when the USB host powers up, it polls each of the slave devices in turn.

The USB host has address 0, and then assigns addresses to each device as well as discovering the slave device capabilities in a process called enumeration. [Enumeration takes place when a new device is connected].

Transactions between the host and device comprise a number of packets. As there are several different types of data that can be sent, a token indicating the type is required, and sometimes an acknowledgement is also returned.

Each packet that is sent is preceded by a sync field and followed by an end of packet marker. This defines the start and end of the packet and also enables the receiving node to synchronize properly so that the various date elements fall into place.

There are four basic types of data transaction that can be made within USB.

***Control:***   This type of data transaction within the overall USB protocol is used by the host to send commands or query parameters. The packet lengths are defined within the protocol as 8 bytes for Low speed, 8-64 bytes for Full, and 64 bytes for High Speed devices.

***Interrupt:***   The USB protocol defines an interrupt message. This is often used by devices sending small amounts of data, e.g. mice or keyboards. It is a polled message from the host which has to request specific data of the remote device

***Bulk:***   This USB protocol message is used by devices like printers for which much larger amounts of data are required. In this form of data transfer, variable length blocks of data are sent or requested by the Host. The maximum length is 64-byte for full speed Devices or 512 bytes for high speed ones. The data integrity is verified using cyclic redundancy checking, CRC and an acknowledgement is sent. This USB data transfer mechanism is not used by time critical peripherals because it utilises bandwidth not used by the other mechanisms.

***Isochronous:***   This form of data transfer is used to stream real time data and is used for applications like live audio channels, etc. It does not use and data checking, as there is not time to resend any data packets with errors - lost data can be accommodated better than the delays incurred by resending data. Packet sizes can be up to 1024 bytes.

The data transfer methodology and protocol for USB provides an effective method of transferring the data across the interface in an effective and reliable manner.

## 10.5) USB data packets

Within the USB system, there are four different types of data packets each used for different types of data transfer.

***Token Packets:***   Essentially a Token USB data packet indicates the type of transaction is to follow.

***Data Packets:***   The USB data packets carry the payload data, carrying the data as required.

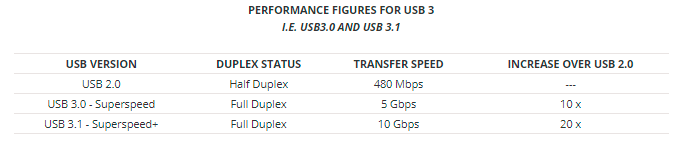
***Handshake Packets:***   The handshake packets are used acknowledging data packets received or for reporting errors, etc.

***Start of Frame Packets:***   The Start of Frame packets used to indicate the start of a new frame of data.

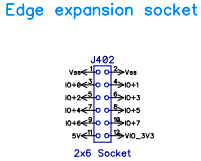
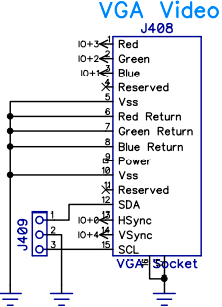
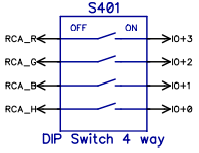
Although USB has developed from USB 1 through USB 2 to USB 3 and now USB 4, it still utilizes the same basic approach to data transfer. There are many USB connectors and leads available, and these leads now have many more wires for higher rate data transfer. Accordingly the data transfer speeds have increased many fold over the first USB specification that was released and the devices that were available.

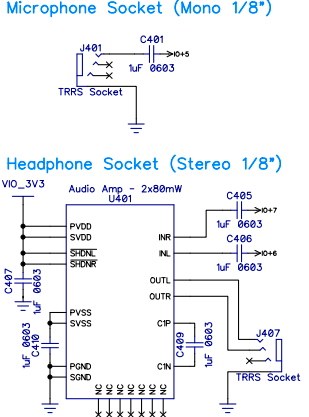
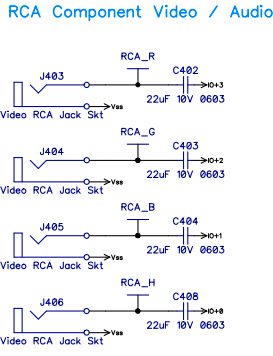
## 10.6) USB 3 capabilities

The USB 3.0, Superspeed and 3.1, Superspeed+ specifications enable much higher rates of data transfer. This is in keeping with requirements for downloading video and many other applications.



# 11.0) P2 Eval A/V (Audio/Video) Breakout Add-on Board (#64006H)





# 12.0 COG Initiation

Any cog can start or stop any other cog, or restart or stop itself. Each of the eight cogs has a unique three-bit ID which can be used to start or stop it. It's also possible to start free (stopped or never started) cogs, without needing to know their ID's. This way, entire applications can be written which simply start free cogs, as needed, and as those cogs retire by stopping themselves or getting stopped by others, they return to the pool of free cogs and become available, again, for restarting.

In **Spin2** the Cogspin(CogNum,Spin\_Method(<(parameters)>,@stack) instruction is used to start cog running a **“spin method”**

In **Spin2** the Coginit(CogID,AsmAddress,Paramater) instruction is used to start cog to run a **“PASM”** program

|  |  |
| --- | --- |
| **COGSPIN(CogNum, Method({Pars}), StkAddr)** | Start Spin2 method in a cog, returns cog's ID if used as an expression element, -1 = no cog free |
| **COGINIT(CogNum, PASMaddr, PTRAvalue)** | Start PASM code in a cog, returns cog's ID if used as an expression element, -1 = no cog free |
| **COGSTOP(CogNum)** | Stop cog CogNum |
| **COGID() : CogNum** | Get this cog's ID |
| **COGCHK(CogNum) : Running** | Check if cog CogNum is running, returns -1 if running or 0 if not |

PASM code can also initiate a cog to rurn.

RET {WC/WZ/WCZ} Return by popping stack (K). C = K[31], Z = K[30], PC = K[19:0].

REGLOAD and REGEXEC

The Spin2 instructions **REGLOAD(HubAddress)** and **REGEXEC(HubAddress)** are used to load or load-and-execute PASM code and/or data chunks from hub RAM into cog registers.

The chunk of PASM code and/or data must be preceded with two words which provide the starting register and the number of registers (longs) to load, minus 1.

|  |
| --- |
| PUB go()    REGLOAD(@chunk)    'load self-defined chunk from hub into registers    REPEAT      CALL(#start)     'call program within chunk at register address      WAITMS(100)  DAT  chunk   WORD    start,finish-start-1  'define chunk start and size-1          ORG     $120                  'org can be $000..$130-size  start   DRVRND  #56 ADDPINS 7         'some code   \_RET\_  DRVNOT  #0                    'more code + return  finish |

REGEXEC works like REGLOAD, but it also CALLs to the start register of the chunk after loading it.

In the example below, REGEXEC launches a chunk of code in upper register memory which sets up a timer interrupt and then returns to Spin2. Meanwhile, as the Spin2 method repeatedly randomizes pins 60..63 every 100ms, the chunk of code loaded into upper register memory perpetuates the timer interrupt and toggles pins 56..59 every 500ms. Note that registers $000..$127 are still free for other code chunks and interrupts 2 and 3 are still unused.

|  |
| --- |
| PUB go()    REGEXEC(@chunk)                       'load self-defined chunk and execute it                                          'chunk starts timer interrupt and returns    REPEAT      PINWRITE(60 ADDPINS 3, GETRND())    'randomize pins 60..63      WAITMS(100)                         'pins 56..59 toggle via interrupt  DAT  chunk   WORD    start,finish-start-1    'define chunk start and size-1          ORG     $128                    'org can be $000..$130-size  start   MOV     IJMP1,#isr              'set int1 vector          SETINT1 #1                      'set int1 to ct-passed-ct1 event          GETCT   PR0                     'get ct   \_ret\_  ADDCT1  PR0,bigwait             'set initial ct1 target, return to Spin2  isr     DRVNOT  #56 ADDPINS 3           'interrupt service routine, toggle 56..59          ADDCT1  PR0,bigwait             'set next ct1 target          RETI1                           'return from interrupt  bigwait LONG    20\_000\_000 / 2          '500ms second on RCFAST  finish |

## 12.1)Built-In Symbols for COGINIT Usage

|  |  |  |
| --- | --- | --- |
| **COGINIT Symbol Value** | **Symbol Name** | **Details** |
| %00\_0000 | COGEXEC (default) | Use "COGEXEC + CogNumber" to start a cog in cogexec mode |
| %10\_0000 | HUBEXEC | Use "HUBEXEC + CogNumber" to start a cog in hubexec mode |
| %01\_0000 | COGEXEC\_NEW | Starts an available cog in cogexec mode |
| %11\_0000 | HUBEXEC\_NEW | Starts an available cog in hubexec mode |
| %01\_0001 | COGEXEC\_NEW\_PAIR | Starts an available eve/odd pair of cogs in cogexec mode, useful for LUT sharing |
| %11\_0001 | HUBEXEC\_NEW\_PAIR | Starts an available eve/odd pair of cogs in hubexec mode, useful for LUT sharing |

## 12.2) Built-In Symbol for COGSPIN Usage

|  |  |  |
| --- | --- | --- |
| **COGINIT Symbol Value** | **Symbol Name** | **Details** |
| %01\_0000 | NEWCOG | Starts an available cog |

## 12.3) PASM Propeller Assembly Machine Language

PASM stands for propeller assembly language program. PASM can be inline with spin2 code or called and loaded separately. There are two different languages PASM for propeller 1 (32 I/O) and PASM for propeller 2 (64 I/O). Most of the instructions are similar but are not 100% equivalent.

The boot procedure requires spin code to initiated. The spin interpreter then can be used to launch PASM code. The propeller 1 and propeller 2 do not operate in the same manner conderning assembly code and hub access.

## 12.4) In Line PASM Code

Spin2 methods can execute in-line PASM code by preceding the PASM code with an **'ORG** {$000..$12F}' and terminating it with an **END**.

|  |
| --- |
| PUB go() | x    repeat      org          getrnd  wc      'rotate a random bit into x          rcl     x,#1      end      pinwrite(56 addpins 7, x)     'output x to the P2 Eval board's LEDs      waitms(100) |
|  |

Your PASM code will be assembled with a **RET** instruction added at the end to ensure that it returns to Spin2, in case no early \_RET\_ or RET executes.

Here's the internal Spin2 procedure for executing in-line PASM code:

* Save the current streamer address for restoration after the PASM code executes.
* Copy the method's first 16 long variables, including any parameters, return values, and local variables, from hub RAM to cog registers $1E0..$1EF.
* Copy the in-line PASM-code longs from hub RAM into cog registers, starting at the ORG address (default is $000).
* CALL the PASM code.
* Restore the 16 longs in cog registers $1E0..$1EF back to hub RAM, in order to update any modified method variables.
* Restore the streamer address and resume Spin2 bytecode execution.

Within your in-line PASM code, you can do all these things:

* Read and write the following register areas:
  + $000..$12F, which your PASM code loads into. You can even load different PASM programs at different addresses within this range and CALL them from Spin2.
  + **$1D8..$1DF**, which are general-purpose registers, **named PR0..PR7**, available to both PASM and Spin2 code.
  + $1E0..$1EF, which temporarily contain the method's first 16 long hub RAM variables and are temporarily assigned the same symbolic names.
  + **$1F0..$1FF**, which include **IJMP3, IRET3, IJMP2, IRET2, IJMP1, IRET1, PA, PB, PTRA,  PTRB, DIRA, DIRB, OUTA, OUTB, INA, and INB**.
  + Avoid writing to $130..$1D7 and LUT RAM, since the Spin2 interpreter occupies these areas. You can look in "Spin2\_interpreter.spin2" to see the interpreter code.
* Use the streamer temporarily.
* Use up to 5 levels of the hardware stack for nested CALLs, including CALLs to hub RAM.
* Declare and reference regular and local symbols. These symbols will not be accessible outside of your PASM code.
* Declare BYTE, WORD, and LONG data.
* Use the **RES, ORGF**, and **FIT** directives. The directives **ORG, ORGH, ALIGNW, ALIGNL, and FILE are not allowed within in-line PASM code.**
* Establish an interrupt which executes your code remaining in cog registers $000..$12F. Spin2 accommodates interrupts and only stalls them briefly, when necessary.
* Return to Spin2, at any point, by executing an \_RET\_ or RET instruction.

## 12.5) Calling PASM from Spin2

You can do a **CALL(address)** in Spin2 to execute PASM code in either cog register space or hub RAM.

|  |
| --- |
| PUB go() | x    repeat      call(@random)      pinwrite(56 addpins 7, pr0)      waitms(100)  DAT     orgh    'hub PASM program to rotate a random bit into pr0  random  getrnd  wc   \_ret\_  rcl     pr0,#1 |

Here's the internal Spin2 procedure for executing a CALL:

* Save the current streamer address for restoration after the PASM code executes.
* CALL the PASM code.
* Restore the streamer address and resume Spin2 bytecode execution.

Within code which you CALL, you can do all these things:

* Read and write the following register areas:
  + $000..$12F, which may contain PASM code and/or data which you previously loaded.
  + $1D8..$1DF, which are general-purpose registers, named PR0..PR7, available to both PASM and Spin2 code.
  + $1E0..$1EF, which are available for scratchpad use, but will likely be rewritten when Spin2 resumes.
  + $1F0..$1FF, which include IJMP3, IRET3, IJMP2, IRET2, IJMP1, IRET1, PA, PB, PTRA,  PTRB, DIRA, DIRB, OUTA, OUTB, INA, and INB.
  + Avoid writing to $130..$1D7 and LUT RAM, since the Spin2 interpreter occupies these areas. You can look in "Spin2\_interpreter.spin2" to see the interpreter code.
* Use the streamer temporarily.
* Use up to 5 levels of the hardware stack for nested CALLs, including CALLs to hub RAM.
* Establish an interrupt which executes your code remaining in cog registers $000..$12F. Spin2 accommodates interrupts and only stalls them briefly, when necessary.
* Return to Spin2, at any point, by executing an \_RET\_ or RET instruction.

## 12.6) Launching Cogs with SpinMethod at StackPointer

**Cogspin(CogNum,Spin\_Method(<(parameters)>,@stack)**

returns cogID if started and running or -1 if no cog free

Note: Followin Stack Space was for P1 assignments

Stack Space should be over estimated and then "Stack Space Tool" can be used to determine

actual size of stack required

2 longs for return address

1 long for return result

1 long for each method parameter

1 long for each local variable

1 long for each intermediate concurrent calculation

## 12.7) Launching Cogs with Assembly Program

The COGINIT instruction is used to start cogs from a PASM program:

**COGINIT D/#,S/# {WC}**

**D/# = %0\_x\_xxxx The target cog loads its own registers $000..$1F7 from the hub,**

**starting at address S/#, then begins execution at address $000.**

**%1\_x\_xxxx The target cog begins execution at address S/#.**

**%x\_0\_CCCC The target cog's ID is %CCCC.**

**%x\_1\_xxx0 If a cog is free (stopped), then start it.**

**To know if this succeeded, D must be a register and WC must be**

**used. If successful, C will be cleared and D will be over-**

**written with the target cog's ID. Otherwise, C will be set and D will be overwritten with $F.**

**%x\_1\_xxx1 If an even/odd cog pair is free (stopped), then start them.**

**To know if this succeeded, D must be a register and WC must be**

**used. If successful, C will be cleared and D will be over-**

**written with the even/lower target cog's ID. Otherwise, C will be**

**set and D will be overwritten with $F.**

**S/# = address This value is either the hub address from which the target cog**

**Will load from, or it is the cog/hub address from which the target**

**Cog will begin executing at, depending on D[5]. This 32-bit value**

**Will be written into the target cog's PTRB register.**

If COGINIT is preceded by SETQ, the SETQ value will be written into the target cog's PTRA register. This is intended as a convenient means of pointing the target cog's program to some runtime data structure or passing it a 32-bit parameter. If no SETQ is used, the target cog's PTRA register will be cleared to zero.

**COGINIT #1,#$100 'load and start cog 1 from $100**

**COGINIT #%1\_0\_0101,PTRA 'start cog 5 at PTRA**

**SETQ    ptra\_val 'ptra\_val will go into target cog's PTRA register**

**COGINIT #%0\_1\_0000,addr 'load and start a free cog at addr**

**COGINIT #%1\_1\_0001,addr 'start a pair of free cogs at addr (lookup RAM sharing)**

**COGINIT id,addr WC '(id=$30) start a free cog at addr, C=0 and id=cog if okay**

**COGID   myID 'reload and restart me at PTRB**

**COGINIT myID,PTRB**

The COGSTOP instruction is used to stop cogs. The 4 LSB's of the D/# operand supply the target cog ID.

**COGSTOP #0 'stop cog 0**

**COGID   myID 'stop me**

**COGSTOP myID**

A cog can discover its own ID by doing a COGID instruction, which will return its ID into D[3:0], with upper bits cleared. This is useful, in case the cog wants to restart or stop itself, as shown above.

If COGID is used with WC, it will not overwrite D, but will return the status of cog D/# into C, where C=0 indicates the cog is free (stopped or never started) and C=1 indicates the cog is busy (started).

**COGID   ThatCog  WC 'C=1 if ThatCog is busy**

## 12.1\_Example\_WRD\_COGINIT\_COGEXEC\_NEW

{{12.1\_Example\_WRD\_COGINIT\_COGEXEC\_NEW}}

''Debug must be enabled

{-------------------------------------------------------------------------------------------------------------}

CON {Processor Timing}

**\_clkfreq = 200\_000\_000 'processor clock speed**

VAR

Byte cogStarted\_COGEXEC\_NEW 'cog ID started is returned or -1 if not started

PUB main()

cogStarted\_COGEXEC\_NEW := COGINIT(COGEXEC\_NEW,@\_blink01,$FFF\_FFFF)

'Start next available cog which is 1 and load cog 1 memory with \_blink01 PASM at Cog Memory $000

'PTRA will be loaded with $FFF\_FFFF

repeat 'keep cog 0 running

DAT ORG 0 'COGINIT(COGEXEC\_NEW,@\_blink,PTRAvalue)

\_blink01

MOV DIRA, #$FF 'Set the direction of the first 8 pins to Output

GETCT cogCounterValue 'Get global system counter value

ADDCT1 cogCounterValue,PTRA 'set CT1 event to trigger on CT = countvalue + PTRA

.\_Loop WAITCT1

ADDCT1 cogCounterValue,PTRA

XOR OUTA, #1

NOP

debug(ubin(OUTA)) 'send status to Debug Window

JMP #.\_Loop 'JMP to Loop

cogCounterValue Long 0 'counter value CT storage

'------------------------------------------

## 12.2\_Example\_WRD\_COGINIT\_COGEXEC\_CogID

{{12.1\_Example\_WRD\_COGINIT\_COGEXEC\_CogID}}

''Debug must be enabled

{-------------------------------------------------------------------------------------------------------------}

CON {Processor Timing}

\_clkfreq = 200\_000\_000 'processor clock speed

VAR

Byte cogStarted\_COGEXEC\_CogID 'cog ID started is returned or -1 if not started

Byte Cog\_ID

PUB main()

Cog\_ID := 2 'the cog to be started is 2

cogStarted\_COGEXEC\_CogID := COGINIT(COGEXEC + Cog\_ID,@\_blink01,$FFF\_FFFF)

debug(udec(cogStarted\_COGEXEC\_CogID))

'Start next available cog which is 1 and load cog 1 memory with \_blink01 PASM at Cog Memory $000

'PTRA will be loaded with $FFF\_FFFF

repeat 'keep cog 0 running

DAT ORG 0 'COGINIT(COGEXEC + CogID,@\_blink,PTRAvalue)

\_blink01

MOV DIRA, #$FF 'Set the direction of the first 8 pins to Output

GETCT cogCounterValue 'Get global system counter value

ADDCT1 cogCounterValue,PTRA 'set CT1 event to trigger on CT = countvalue + PTRA

.\_Loop WAITCT1

ADDCT1 cogCounterValue,PTRA

XOR OUTA, #1

NOP

debug(ubin(OUTA)) 'send status to Debug Window

JMP #.\_Loop 'JMP to Loop

cogCounterValue Long 0 'counter value CT storage

'------------------------------------------

## 12.3\_Example\_WRD\_COGINIT\_HUBEXEC

{12.3\_Example\_WRD\_COGINIT\_HUBEXEC}

CON

\_clkfreq = 200\_000\_000 'debug must have a clock greater than 10MHZ

pub go()

coginit(HUBEXEC\_NEW,@blink,0) 'launch hub-exe program in free cog

'%11\_0000 = HUBEXEC\_NEW Starts an available cog in hubexec mode

'coginit(32+16,@blink,0)

DAT orgh 'being hub-exec program

blink setbyte dira,#$FF,#0 'make LEDs outputs

debug(ubin(dira))

mov pb,#0 'clear pb

loop loc pa,#table 'get address of table (relative)

add pa,pb 'add pb

rdbyte pa,pa 'read pa table byte and put in pa

not pa 'NOT for LEDs

setbyte outa,pa,#0 'write to LEDs

debug(ubin(outa))

waitx ##clkfreq\_

debug(udec(#clkfreq\_))

incmod pb,#7 'inc pb 0..7 and repeat

'INCMOD D,{#}S {WC/WZ/WCZ} Increment with modulus. If D = S then D = 0 and C = 1, else D = D + 1 ‘and C = 0. \*

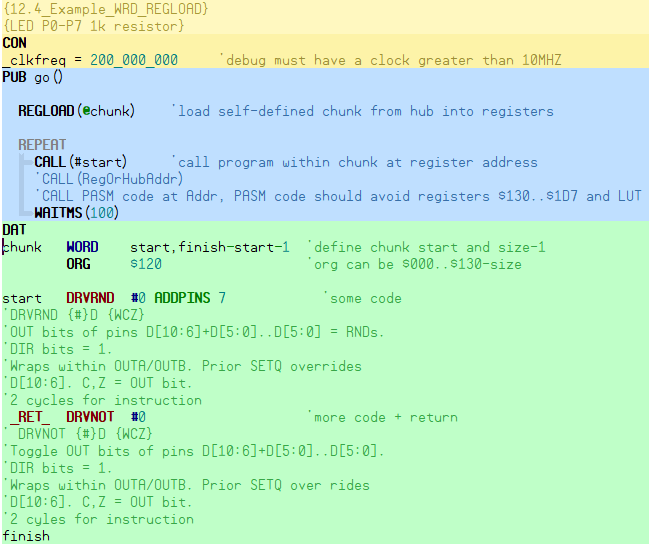
waitx ##clkfreq\_ 'wait 1/10 second

'WAITX {#}D {WC/WZ/WCZ} Wait 2 + D clocks if no WC/WZ/WCZ. If WC/WZ/WCZ, wait 2 + (D & RND) ‘clocks. C/Z = 0.

jmp #loop 'loop

table byte $01,$02,$04,$08,$10,$20,$40,$80

## 12.4\_Example\_WRD\_REGLOAD



# 13.0) Debug for Testing and Troubleshooting

The Spin2 compiler contains a stealthy debugger program that can be automatically downloaded with your application. It uses the last 16KB of RAM plus a few bytes for each Spin2 DEBUG statement and one instruction for each PASM DEBUG statement. You place DEBUG statements in your application which contain output commands that will serially transmit the state of variables and equations as your application runs. Each time a DEBUG statement is encountered during execution, the debugger is invoked and it outputs the message for that statement. Debugging is initiated by adding the **Ctrl key to the usual F10 to 'run' or F11 to 'program'.** This compiles your application with all the DEBUG statements, adds the debugger to the download, and then brings up the DEBUG Output window which begins receiving messages at the start of your application. DEBUG can be used in Spin2 or PASM.

## 13.1) Things to know about the DEBUG system

* To use the debugger, you must configure at least a 10 MHz clock derived from a crystal or external input. You cannot use RCFAST or RCSLOW.
* The debugger occupies the top 16 KB of hub RAM, remapped to $FC000..$FFFFF and write-protected. The hub RAM at $7C000..$7FFFF will no longer be available.
* Data defining each DEBUG statement is stored within the debugger image in the top 16 KB of RAM, minimizing impact on your application code.
* In Spin2, each DEBUG statement adds three bytes, plus any code needed to reference variables and resolve run-time expressions used in the DEBUG statement.
* In PASM, each DEBUG statement adds one instruction (long).
* DEBUG statements are ignored by the compiler when not compiling for DEBUG mode, so you don't need to comment them out when debugging is not in use.
* If no DEBUG statements exist in your application, you will still get notification messages when cogs are started.
* Debugging is invoked by pressing the CTRL key before the usual F9..F11 keys, which compile, download, and program to flash.
* During execution, as DEBUG statements are encountered, text messages are sent out serially on P62 at 2 Mbaud in 8-N-1 format.
* DEBUG messages always start with "CogN  ", where N is the cog number, followed by two spaces, and they always end with CR+LF (new line).
* Up to 255 DEBUG statements can exist within your application, since the BRK instruction is used to interrupt and select the particular DEBUG statement definition.
* You can define several symbols to modify debugger behavior: DEBUG\_COG, DEBUG\_DELAY, DEBUG\_PIN, DEBUG\_TIMESTAMP, etc. See table.
* Each time a debug-enabled cog is started, a debug message is output to indicate the cog number, code address (PTRB), parameter (PTRA), and 'load' or 'jump' mode.
* For Spin2, DEBUG statements can output expression and variable values, hub byte/word/long arrays, and register arrays.
* For PASM, DEBUG statements can output register values/arrays, hub byte/word/long arrays, and constants. PASM syntax is used: implied register or #immediate.
* DEBUG output data can be displayed in decimal, hex, or binary, signed or unsigned, and sized to byte, word, long, or auto. Hub character strings are also supported.
* DEBUG output commands show both the source and value: "DEBUG(UHEX(x))" might output "x = $123".
* DEBUG commands which output data can have multiple sets of parameters, separated by commas: SDEC(x,y,z) and LSTR(ptr1,size1,ptr2,size2)
* Commas are automatically output between data: "DEBUG(UHEX\_BYTE(d,e,f), SDEC(g))" might output "d = $45, e = $67, f = $89, g = -1\_024".
* All DEBUG output commands have alternate versions, ending in "\_" which output only the value: DEBUG(UHEX\_BYTE\_(d,e,f)) might output "$45, $67, $89".
* DEBUG statements can contain comma-separated strings and characters, aside from commands: DEBUG("We got here! Oh, Nooooo...", 13, 13)
* DEBUG statements may contain IF() and IFNOT() commands to gate further output within the statement. An initial IF/IFNOT will gate the entire message.
* DEBUG statements may contain a final DLY(milliseconds) command to slow down a cog's messaging, since messages may stream at the rate of ~10,000 per second.
* DEBUG serial output can be redirected to a different pin, at a different baud rate, for displaying/logging elsewhere.
* LOCK[15] is allocated by the debugger and used among all cogs during their debug interrupts to time-share the DEBUG serial-transmit pin.
* Command-line supports DEBUG-only mode: PNut -debug {CommPort if not 1} {BaudRate if not 2\_000\_000}

|  |  |  |
| --- | --- | --- |
| **DEBUG Statement (v=100, BYTE[a]=1,2,3,4,5)** | **DEBUG Message Output** | **Note** |
| DEBUG("`LOGIC MyDisplay SAMPLES ", SDEC\_(v)) | Cog0  `LOGIC MyDisplay SAMPLES 100 | Regular DEBUG syntax can drive DEBUG displays, but it's not optimal. |
| DEBUG(`LOGIC MyDisplay SAMPLES 100) | `LOGIC MyDisplay SAMPLES 100 | DEBUG-display syntax is simpler and 'CogN' is omitted in the output. |
| DEBUG(`LOGIC MyDisplay SAMPLES `(v)) | `LOGIC MyDisplay SAMPLES 100 | Decimal numbers are output using `(value) notation. Short for SDEC\_. |
| DEBUG(`LOGIC MyDisplay SAMPLES `$(v)) | `LOGIC MyDisplay SAMPLES $64 | Hex numbers are output using `$(value) notation. Short for UHEX\_. |
| DEBUG(`LOGIC MyDisplay SAMPLES `%(v)) | `LOGIC MyDisplay SAMPLES %1100100 | Binary numbers are output using `%(value) notation. Short for UBIN\_. |
| DEBUG(`LOGIC MyDisplay TITLE '`#(v)') | `LOGIC MyDisplay TITLE 'd' | Characters are output using `#(value) notation. |
| DEBUG(`MyDisplay `UDEC\_BYTE\_ARRAY\_(@a,5)) | `MyDisplay 1, 2, 3, 4, 5 | Regular DEBUG commands can follow the backtick, as well. |

## 13.2) Simple DEBUG example in Spin2

|  |
| --- |
| CON \_clkfreq = 10\_000\_000      'set 10 MHz clock (assumes 20 MHz crystal)  PUB go() | i    REPEAT i FROM 0 TO 9         'count from 0 to 9      DEBUG(UDEC(i))             'debug, output i |

When run with Ctrl-F10, the Debug window opens and this is what appears:

|  |
| --- |
| Cog0  INIT $0000\_0000 $0000\_0000 load  Cog0  INIT $0000\_0D6C $0000\_10BC jump  Cog0  i = 0  Cog0  i = 1  Cog0  i = 2  Cog0  i = 3  Cog0  i = 4  Cog0  i = 5  Cog0  i = 6  Cog0  i = 7  Cog0  i = 8  Cog0  i = 9 |

In the first line of the report, you see Cog0 loading the Spin2 set-up code from $00000. In the second line, the Spin2 interpreter is launched from $00D58 with its stack space starting at $0101C. After that, the Spin2 program is running and you see 'i' iterating from 0 to 9.

If you change the "9" to "99" in the REPEAT, data will scroll too fast to read, but by adding a DLY command at the end of the DEBUG statement, you can slow down the output:

    debug(udec(i), dly(250))     'debug, output i with a 250ms delay after each report

Let's say you want to limit the messages being output, so that only odd values of 'i' are shown. You could use an IF at the start of your DEBUG statement to check the least-significant bit of 'i'. When the IF is false, no message will be output, causing only the odd values of i to be shown:

    debug(if(i & 1), udec(i), dly(250))     'debug, output only odd i values with a 250ms delay after each report

## 13.3) Simple DEBUG example in PASM

|  |
| --- |
| CON \_clkfreq = 10\_000\_000       'set 10 MHz clock (assumes 20 MHz crystal)  DAT     ORG          MOV     i,#9            'set i to 9  loop    DEBUG   (UHEX\_LONG(i))  'debug, output i in hex          DJNF    i,#loop         'decrement i and loop if not -1          JMP     #$              'don't go wandering off, stay here  i       RES     1               'reserve one register as 'i' |

When run with Ctrl-F10, the Debug window opens and this is what appears:

|  |
| --- |
| Cog0  INIT $0000\_0000 $0000\_0000 load  Cog0  i = $0000\_0009  Cog0  i = $0000\_0008  Cog0  i = $0000\_0007  Cog0  i = $0000\_0006  Cog0  i = $0000\_0005  Cog0  i = $0000\_0004  Cog0  i = $0000\_0003  Cog0  i = $0000\_0002  Cog0  i = $0000\_0001  Cog0  i = $0000\_0000 |

In the first line of the report, you see Cog0 loading our PASM program from $00000. After that, the program runs and you see 'i' iterating from 9 down to 0.

If you change the "9" to "99" in the MOV instruction and you'd like to slow things down, add a DLY command to the DEBUG statement and be sure to express the milliseconds as #250, since a plain 250 would be understood as register 250:

    debug   (uhex\_long(i), dly(#250))      'debug, output i in hex and delay for 250ms after each report

There are two steps to using graphical DEBUG displays. First, they must be instantiated and, second, they must be fed:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **To Use a Display:** | **1st** | **2nd** | **3rd** | **4th** | **Note** |
| **First, instantiate it.** | ` | display\_type | unknown\_symbol | keyword(s), number(s), string(s) | Unknown\_symbol becomes instance\_name. |
| **Then, feed it.** | ` | instance\_name(s) | keyword(s), number(s), string(s) |  | Multiple displays can be fed the same data. |

To bring this all together, let's show a sawtooth wave on a SCOPE display:

|  |  |  |
| --- | --- | --- |
| |  | | --- | | CON \_clkfreq = 10\_000\_000  PUB go() | i    debug(`SCOPE MyScope SIZE 254 84 SAMPLES 128)    debug(`MyScope 'Sawtooth' 0 63 64 10 %1111)    repeat      debug(`MyScope `(i & 63))      i++      waitms(50) | | https://lh4.googleusercontent.com/QNa3TEwzFJy6XTrHoqnUBjQUJmmz5G4CWYnVLH7gui0K-0MUpWN5nt5JFlJrvIU5H-WSsvBlyrEpQa7dWk5JGvFSorgs7iLi3DKTMckLgKbY0yub-Evp4yHTzOcYYi-5nR3AVTND |

**Step 1 Instanitiate Graphic Display**

debug(`SCOPE Scope\_Name SIZE 254 84 SAMPLES 128) ‘note the backtick “ ` ”

debug(`Scope\_Name ‘Sawtooth’ 0 63 64 10 %1111) ‘note strings have single quote ‘Sawtooth’

**Step 2 Feed Data to Window (Display)**

Debug(`Scope\_Name ( i & 63))

**Debug(`Scope\_Name**

In the example above, a SCOPE is instantiated called MyScope that is 254 x 84 pixels and shows 128 samples. A width of 254 was chosen since samples are numbered 0..127 and I wanted them to be spaced at a constant two-pixel pitch (127 \* 2 = 254). A height of 84 was chosen so that there would be 10 pixels above and below the waveform, which will have a height of 64 pixels.

A channel called "Sawtooth" is defined which, for the purpose of display, has a bottom value of 0 and a top value of 63, is 64 pixels tall within that range, and is elevated 10 pixels off the bottom of the scope window. The %1111 enables top and bottom legend values and top and bottom lines. Within the REPEAT block, the SCOPE is fed a repeating pattern of 0..63 which forms the sawtooth wave. The SCOPE updates its display each time it receives a value. If there were eight channels defined, instead of just one, it would update the display on every eighth value received, drawing all eight channels.

## 13.4) Commands for use in DEBUG statements

|  |  |
| --- | --- |
| **Conditionals** | **Details** |
| IF(condition) | If condition <> 0 then continue at the next command within the DEBUG statement, else skip all remaining commands and output CR+LF. If used as the first command in the DEBUG statement, IF will gate ALL output for the statement, including the "CogN  "+CR+LF. This way, DEBUG messages can be entirely suppressed, so that you can filter what is important. |
| IFNOT(condition) | If condition = 0 then continue at the next command within the DEBUG statement, else skip all remaining commands and output CR+LF. If used as the first command in the DEBUG statement, IFNOT will gate ALL output for the statement, including the "CogN  "+CR+LF. This way, DEBUG messages can be entirely suppressed, so that you can filter what is important. |

|  |  |  |
| --- | --- | --- |
| **String Output \*** | **Details** | **Output** |
| ZSTR(hub\_pointer) | Output zero-terminated string at hub\_pointer | "Hello!" |
| LSTR(hub\_pointer,size) | Output 'size' characters of string at hub\_pointer | "Goodbye." |

|  |  |  |  |
| --- | --- | --- | --- |
| **Decimal Output, unsigned \*** | **Details** | **Min Output** | **Max Output** |
| UDEC(value) | Output unsigned decimal value | 0 | 4\_294\_967\_295 |
| UDEC\_BYTE(value) | Output byte-size unsigned decimal value | 0 | 255 |
| UDEC\_WORD(value) | Output word-size unsigned decimal value | 0 | 65\_535 |
| UDEC\_LONG(value) | Output long-size unsigned decimal value | 0 | 4\_294\_967\_295 |
| UDEC\_REG\_ARRAY(reg\_pointer,size) | Output register array as unsigned decimal values | 0 | 4\_294\_967\_295 |
| UDEC\_BYTE\_ARRAY(hub\_pointer,size) | Output hub byte array as unsigned decimal values | 0 | 255 |
| UDEC\_WORD\_ARRAY(hub\_pointer,size) | Output hub word array as unsigned decimal values | 0 | 65\_535 |
| UDEC\_LONG\_ARRAY(hub\_pointer,size) | Output hub long array as unsigned decimal values | 0 | 4\_294\_967\_295 |
| **Decimal Output, signed \*** | **Details** | **Min Output** | **Max Output** |
| SDEC(value) | Output signed decimal value | -2\_147\_483\_648 | 2\_147\_483\_647 |
| SDEC\_BYTE(value) | Output byte-size signed decimal value | -128 | 127 |
| SDEC\_WORD(value) | Output word-size signed decimal value | -32\_768 | 32\_767 |
| SDEC\_LONG(value) | Output long-size signed decimal value | -2\_147\_483\_648 | 2\_147\_483\_647 |
| SDEC\_REG\_ARRAY(reg\_pointer,size) | Output register array as signed decimal values | -2\_147\_483\_648 | 2\_147\_483\_647 |
| SDEC\_BYTE\_ARRAY(hub\_pointer,size) | Output hub byte array as signed decimal values | -128 | 127 |
| SDEC\_WORD\_ARRAY(hub\_pointer,size) | Output hub word array as signed decimal values | -32\_768 | 32\_767 |
| SDEC\_LONG\_ARRAY(hub\_pointer,size) | Output hub long array as signed decimal values | -2\_147\_483\_648 | 2\_147\_483\_647 |
| **Hexadecimal Output, unsigned \*** | **Details** | **Min Output** | **Max Output** |
| UHEX(value) | Output auto-size unsigned hex value | $0 | $FFFF\_FFFF |
| UHEX\_BYTE(value) | Output byte-size unsigned hex value | $00 | $FF |
| UHEX\_WORD(value) | Output word-size unsigned hex value | $0000 | $FFFF |
| UHEX\_LONG(value) | Output long-size unsigned hex value | $0000\_0000 | $FFFF\_FFFF |
| UHEX\_REG\_ARRAY(reg\_pointer,size) | Output register array as unsigned hex values | $0000\_0000 | $FFFF\_FFFF |
| UHEX\_BYTE\_ARRAY(hub\_pointer,size) | Output hub byte array as unsigned hex values | $00 | $FF |
| UHEX\_WORD\_ARRAY(hub\_pointer,size) | Output hub word array as unsigned hex values | $0000 | $FFFF |
| UHEX\_LONG\_ARRAY(hub\_pointer,size) | Output hub long array as unsigned hex values | $0000\_0000 | $FFFF\_FFFF |
| **Hexadecimal Output, signed \*** | **Details** | **Min Output** | **Max Output** |
| SHEX(value) | Output auto-size signed hex value | -$8000\_0000 | $7FFF\_FFFF |
| SHEX\_BYTE(value) | Output byte-size signed hex value | -$80 | $7F |
| SHEX\_WORD(value) | Output word-size signed hex value | -$8000 | $7FFF |
| SHEX\_LONG(value) | Output long-size signed hex value | -$8000\_0000 | $7FFF\_FFFF |
| SHEX\_REG\_ARRAY(reg\_pointer,size) | Output register array as signed hex values | -$8000\_0000 | $7FFF\_FFFF |
| SHEX\_BYTE\_ARRAY(hub\_pointer,size) | Output hub byte array as signed hex values | -$80 | $7F |
| SHEX\_WORD\_ARRAY(hub\_pointer,size) | Output hub word array as signed hex values | -$8000 | $7FFF |
| SHEX\_LONG\_ARRAY(hub\_pointer,size) | Output hub long array as signed hex values | -$8000\_0000 | $7FFF\_FFFF |
| **Binary Output, unsigned \*** | **Details** | **Min Output** | **Max Output** |
| UBIN(value) | Output auto-size unsigned binary value | %0 | %11111111\_11111111\_11111111\_11111111 |
| UBIN\_BYTE(value) | Output byte-size unsigned binary value | %00000000 | %11111111 |
| UBIN\_WORD(value) | Output word-size unsigned binary value | %00000000\_00000000 | %11111111\_11111111 |
| UBIN\_LONG(value) | Output long-size unsigned binary value | %00000000\_00000000\_00000000\_00000000 | %11111111\_11111111\_11111111\_11111111 |
| UBIN\_REG\_ARRAY(reg\_pointer,size) | Output register array as unsigned binary values | %00000000\_00000000\_00000000\_00000000 | %11111111\_11111111\_11111111\_11111111 |
| UBIN\_BYTE\_ARRAY(hub\_pointer,size) | Output hub byte array as unsigned binary values | %00000000 | %11111111 |
| UBIN\_WORD\_ARRAY(hub\_pointer,size) | Output hub word array as unsigned binary values | %00000000\_00000000 | %11111111\_11111111 |
| UBIN\_LONG\_ARRAY(hub\_pointer,size) | Output hub long array as unsigned binary values | %00000000\_00000000\_00000000\_00000000 | %11111111\_11111111\_11111111\_11111111 |
| **Binary Output, signed \*** | **Details** | **Min Output** | **Max Output** |
| SBIN(value) | Output auto-size signed binary value | -%10000000\_00000000\_00000000\_00000000 | %01111111\_11111111\_11111111\_11111111 |
| SBIN\_BYTE(value) | Output byte-size signed binary value | -%10000000 | %01111111 |
| SBIN\_WORD(value) | Output word-size signed binary value | -%10000000\_00000000 | %01111111\_11111111 |
| SBIN\_LONG(value) | Output long-size signed binary value | -%10000000\_00000000\_00000000\_00000000 | %01111111\_11111111\_11111111\_11111111 |
| SBIN\_REG\_ARRAY(reg\_pointer,size) | Output register array as signed binary values | -%10000000\_00000000\_00000000\_00000000 | %01111111\_11111111\_11111111\_11111111 |
| SBIN\_BYTE\_ARRAY(hub\_pointer,size) | Output hub byte array as signed binary values | -%10000000 | %01111111 |
| SBIN\_WORD\_ARRAY(hub\_pointer,size) | Output hub word array as signed binary values | -%10000000\_00000000 | %01111111\_11111111 |
| SBIN\_LONG\_ARRAY(hub\_pointer,size) | Output hub long array as signed binary values | -%10000000\_00000000\_00000000\_00000000 | %01111111\_11111111\_11111111\_11111111 |

|  |  |
| --- | --- |
| **Delay to Pace Messages** | **Details** |
| DLY(milliseconds) | Delay for some milliseconds to slow down continuous message outputs for this cog. DLY is only allowed as the last command in a DEBUG statement, since it releases LOCK[15] before the delay, permitting other cogs to capture LOCK[15] so that they may take control of the DEBUG serial-transmit pin and output their own DEBUG messages. |

\*  These commands accept multiple parameters, or multiple sets of parameters. Alternate commands with the same names, but ending in "\_", are also available for value-only output (i.e. ZSTR\_, LSTR\_, UDEC\_).

## 13.5) Symbols you can define to modify DEBUG behavior

|  |  |  |
| --- | --- | --- |
| **CON Symbol** | **Default** | **Purpose** |
| DEBUG\_COGS | %11111111 | Selects which cogs have debug interrupts enabled. Bits 7..0 enable debugging interrupts in cogs 7..0. |
| DEBUG\_DELAY | 0 | Sets a delay in milliseconds before your application runs and DEBUG messages start appearing. |
| DEBUG\_PIN | 62 | Sets the DEBUG serial output pin. For DEBUG windows to open, DEBUG\_PIN must be 62. |
| DEBUG\_BAUD | 2\_000\_000 | Sets the DEBUG baud rate. |
| DEBUG\_TIMESTAMP | undefined | By declaring this symbol, each DEBUG message will be time-stamped with the 64-bit CT value. |
| DEBUG\_LOG\_SIZE | 0 | Sets the maximum size of the 'DEBUG.log' file which will collect DEBUG messages. A value of 0 will inhibit log file generation. |
| DEBUG\_LEFT | (dynamic) | Sets the left screen coordinate where the DEBUG message window will appear. |
| DEBUG\_TOP | (dynamic) | Sets the top screen coordinate where the DEBUG message window will appear. |
| DEBUG\_WIDTH | (dynamic) | Sets the width of the DEBUG message window. |
| DEBUG\_HEIGHT | (dynamic) | Sets the height of the DEBUG message window. |
| DEBUG\_DISPLAY\_LEFT | 0 | Sets the overall left screen offset where any DEBUG displays will appear (adds to 'POS' x coordinate in each DEBUG display). |
| DEBUG\_DISPLAY\_TOP | 0 | Sets the overall top screen offset where any DEBUG displays will appear (adds to 'POS' y coordinate in each DEBUG display). |
| DEBUG\_WINDOWS\_OFF | 0 | Disables any DEBUG windows from opening after downloading, if set to a non-zero value. |

## 13.6) Packed-Data Modes

Packed-data modes are used to efficiently convey sub-byte data types, by having the host side unpack them from bytes, words, or longs it receives. As well, bytes can be sent within words and longs, and words can be sent within longs for some efficiency improvement.

To establish packed-data operation, you must specify one of the modes listed below, followed by optional 'ALT' and 'SIGNED' keywords:

**packed\_mode {ALT} {SIGNED}**

The **ALT** keyword will cause bits, double-bits, or nibbles, within each byte sent, to be reordered on the host side, within each byte. This simplifies cases where the raw data you are sending has its bitfields out-of-order with respect to the DEBUG display you are using. This is most-likely to be needed for bitmap data that was composed in standard formats.

The **SIGNED** keyword will cause all unpacked data values to be sign-extended on the host side.

|  |  |  |  |
| --- | --- | --- | --- |
| **Packed-Data**  **Modes** | **Descriptions** | **Final Values** | **Final Values**  **if SIGNED** |
| **LONGS\_1BIT** | Each value received is translated into 32 separate 1-bit values, starting from the LSB of the received value. | 0..1 | -1..0 |
| **LONGS\_2BIT** | Each value received is translated into 16 separate 2-bit values, starting from the LSBs of the received value. | 0..3 | -2..1 |
| **LONGS\_4BIT** | Each value received is translated into 8 separate 4-bit values, starting from the LSBs of the received value. | 0..15 | -8..7 |
| **LONGS\_8BIT** | Each value received is translated into 4 separate 8-bit values, starting from the LSBs of the received value. | 0..255 | -128..127 |
| **LONGS\_16BIT** | Each value received is translated into 2 separate 16-bit values, starting from the LSBs of the received value. | 0..65,535 | -32,768..32,767 |
| **WORDS\_1BIT** | Each value received is translated into 16 separate 1-bit values, starting from the LSB of the received value. | 0..1 | -1..0 |
| **WORDS\_2BIT** | Each value received is translated into 8 separate 2-bit values, starting from the LSBs of the received value. | 0..3 | -2..1 |
| **WORDS\_4BIT** | Each value received is translated into 4 separate 4-bit values, starting from the LSBs of the received value. | 0..15 | -8..7 |
| **WORDS\_8BIT** | Each value received is translated into 2 separate 8-bit values, starting from the LSBs of the received value. | 0..255 | -128..127 |
| **BYTES\_1BIT** | Each value received is translated into 8 separate 1-bit values, starting from the LSB of the received value. | 0..1 | -1..0 |
| **BYTES\_2BIT** | Each value received is translated into 4 separate 2-bit values, starting from the LSBs of the received value. | 0..3 | -2..1 |
| **BYTES\_4BIT** | Each value received is translated into 2 separate 4-bit values, starting from the LSBs of the received value. | 0..15 | -8..7 |

## 13.7) Graphical DEBUG Displays

DEBUG messages can invoke special graphical DEBUG displays which are built into the tool. These graphical displays each take the form of a unique window. Once instantiated, displays can be continuously fed data to generate animated visualizations. These displays are very handy for development and debugging, as various data types can be viewed in their native contexts. Up to 32 graphical displays can be running simultaneously.

Currently, the following graphical DEBUG displays are implemented, but more will be added in the future:

|  |  |
| --- | --- |
| **Display Types** | **Descriptions** |
| **LOGIC** | Logic analyzer with single and multi-bit labels, 1..32 channels, can trigger on pattern |
| **SCOPE** | Oscilloscope with 1..8 channels, can trigger on level with hysteresis |
| **SCOPE\_XY** | XY oscilloscope with 1..8 channels, persistence of 0..512 samples, polar mode, log scale mode |
| **FFT** | Fast Fourier Transform with 1..8 channels, 4..2048 points, windowed results, log scale mode |
| **SPECTRO** | Spectrograph with 4..2048-point FFT, windowed results, phase-coloring, and log scale mode |
| **PLOT** | General-purpose plotter with cartesian and polar modes |
| **TERM** | Text terminal with up to 300 x 200 characters, 6..200 point font size, 4 simultaneous color schemes |
| **BITMAP** | Bitmap, 1..2048 x 1..2048 pixels, 1/2/4/8/16/32-bit pixels with 19 color systems, 15 direction/autoscroll modes, independent X and Y pixel size of 1..256 |
| **MIDI** | Piano keyboard with 1..128 keys, velocity depiction, variable screen scale |

When a DEBUG message contains a backtick (`) character (ASCII $60), a string, containing everything from the backtick to the end of the message, is sent to the graphical DEBUG display parser. The parser looks for several different element types, treating any commas as whitespace:

|  |  |  |
| --- | --- | --- |
| **Element Type** | **Example** | **Description** |
| display\_type | LOGIC, SCOPE, PLOT, BITMAP | This is the formal name of the graphical DEBUG display type you wish to instantiate. |
| unknown\_symbol | MyLogicDisplay | Each graphical DEBUG display Instance must be given a unique symbolic name. |
| instance\_name | MyLogicDisplay | Once instantiated, a graphical DEBUG display instance is referenced by its symbolic name. |
| keyword | TITLE, POS, SIZE, SAMPLES | Keywords are used to configure displays. They might be followed by numbers, strings, and other keywords. |
| number | 1024, $FF, %1010 | Numbers can be expressed in decimal, hex ($), and binary (%). |
| string | 'Here is a string' | Strings are expressed within single-quotes. |

Before getting into how all this fits together, we need to go over some special DEBUG-display syntax that can be used for displays. This syntax is invoked when the first character in the DEBUG statement is the backtick. This causes everything in the DEBUG statement to be viewed as a string, except when subsequent backticks act as 'escape' characters to allow normal or shorthand DEBUG commands.

|  |  |  |
| --- | --- | --- |
| **DEBUG Statement (v=100, BYTE[a]=1,2,3,4,5)** | **DEBUG Message Output** | **Note** |
| DEBUG("`LOGIC MyDisplay SAMPLES ", SDEC\_(v)) | Cog0  `LOGIC MyDisplay SAMPLES 100 | Regular DEBUG syntax can drive DEBUG displays, but it's not optimal. |
| DEBUG(`LOGIC MyDisplay SAMPLES 100) | `LOGIC MyDisplay SAMPLES 100 | DEBUG-display syntax is simpler and 'CogN' is omitted in the output. |
| DEBUG(`LOGIC MyDisplay SAMPLES `(v)) | `LOGIC MyDisplay SAMPLES 100 | Decimal numbers are output using `(value) notation. Short for SDEC\_. |
| DEBUG(`LOGIC MyDisplay SAMPLES `$(v)) | `LOGIC MyDisplay SAMPLES $64 | Hex numbers are output using `$(value) notation. Short for UHEX\_. |
| DEBUG(`LOGIC MyDisplay SAMPLES `%(v)) | `LOGIC MyDisplay SAMPLES %1100100 | Binary numbers are output using `%(value) notation. Short for UBIN\_. |
| DEBUG(`LOGIC MyDisplay TITLE '`#(v)') | `LOGIC MyDisplay TITLE 'd' | Characters are output using `#(value) notation. |
| DEBUG(`MyDisplay `UDEC\_BYTE\_ARRAY\_(@a,5)) | `MyDisplay 1, 2, 3, 4, 5 | Regular DEBUG commands can follow the backtick, as well. |

**There are two steps to using graphical DEBUG displays. First, they must be instantiated and, second, they must be fed:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **To Use a Display:** | **1st** | **2nd** | **3rd** | **4th** | **Note** |
| **First, instantiate it.** | ` | display\_type | unknown\_symbol | keyword(s), number(s), string(s) | Unknown\_symbol becomes instance\_name. |
| **Then, feed it.** | ` | instance\_name(s) | keyword(s), number(s), string(s) |  | Multiple displays can be fed the same data. |

**Note: The backtick is critical and the variables must have single quotes**

debug(`BobTerm 0 '`udec(x)' 9 '`udec(y)') ‘gives decimal value

debug(`BobTerm 0 '`uhex(x)' 9 '`uhex(y)') ‘gives hex value

debug(`BobTerm 0 '`ubin(x)' 9 '`ubin(y)') ‘gives binary value

debug(`BobTerm 0 'Var x = `(x)' 9 'Var y = `(y)') ‘gives decimal value

debug(`BobTerm 0 '`$(x)' 9 '`$(y)') ‘gives hex value

debug(`BobTerm 0 '`%(x)' 9 '`%(y)') ‘gives binary value

CON \_clkfreq = 10\_000\_000

PUB go() | i

  debug(`SCOPE MyScope SIZE 254 84 SAMPLES 128)

  debug(`MyScope 'Sawtooth' 0 63 64 10 %1111)

  repeat

    debug(`MyScope `(i & 63))

    i++

    waitms(50)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | |  | https://lh4.googleusercontent.com/QNa3TEwzFJy6XTrHoqnUBjQUJmmz5G4CWYnVLH7gui0K-0MUpWN5nt5JFlJrvIU5H-WSsvBlyrEpQa7dWk5JGvFSorgs7iLi3DKTMckLgKbY0yub-Evp4yHTzOcYYi-5nR3AVTND |
|  |  |  |

In the example above, a SCOPE is instantiated called MyScope that is 254 x 84 pixels and shows 128 samples. A width of 254 was chosen since samples are numbered 0..127 and I wanted them to be spaced at a constant two-pixel pitch (127 \* 2 = 254). A height of 84 was chosen so that there would be 10 pixels above and below the waveform, which will have a height of 64 pixels. A channel called "Sawtooth" is defined which, for the purpose of display, has a bottom value of 0 and a top value of 63, is 64 pixels tall within that range, and is elevated 10 pixels off the bottom of the scope window. The %1111 enables top and bottom legend values and top and bottom lines. Within the REPEAT block, the SCOPE is fed a repeating pattern of 0..63 which forms the sawtooth wave. The SCOPE updates its display each time it receives a value. If there were eight channels defined, instead of just one, it would update the display on every eighth value received, drawing all eight channels.

## 13.8) Logic Analyzer Display

**Logic analyzer with single and multi-bit labels, 1..32 channels, can trigger on pattern**

|  |  |  |
| --- | --- | --- |
| |  | | --- | | CON \_clkfreq = 10\_000\_000  PUB go() | i    debug(`LOGIC MyLogic SAMPLES 32 'Low' 3 'Mid' 2 'High')    debug(`MyLogic TRIGGER $07 $04 HOLDOFF 2)    repeat      debug(`MyLogic `(i & 63))      i++      waitms(25) | | https://lh6.googleusercontent.com/_XFr5MX3bjMRuHPfMZSD03CeF2WovNwKKkmprG0vYbFjR-43d6PFwpxvqbnuBqAX_VTkkXIuqbMs8m5hrndIb5gDuMWTf42YH1DDj-OLEjSP0LBSGUK7QKRsUeftBHLTnU5q6r1j |

|  |  |  |
| --- | --- | --- |
| **LOGIC Instantiation** | **Description** | **Default** |
| **TITLE 'string'** | Set the window caption to 'string'. | <none> |
| **POS left top** | Set the window position. | 0, 0 |
| **SAMPLES 4\_to\_2048** | Set the number of samples to track and display. | 32 |
| **SPACING 2\_to\_32** | Set the sample spacing. The width of the display will be SAMPLES \* SPACING. | 8 |
| **RATE 1\_to\_2048** | Set the number of samples (or triggers, if enabled) before each display update. | 1 |
| **LINESIZE 1\_to\_7** | Set the line size. | 1 |
| **TEXTSIZE 6\_to\_200** | Set the legend text size. Height of text determines height of logic levels. | editor text size |
| **COLOR back\_color {grid\_color}** | Set the background and grid colors \*. | BLACK, GREY 4 |
| **'name' {1\_to\_32 {color}}** | Set the first/next channel or group name, optional bit count, optional color \*. | 1, default color |
| **packed\_data\_mode** | Enable packed-data mode. See description at end of this section. | <none> |
| **LOGIC Feeding** | **Description** | **Default** |
| **TRIGGER mask match sample\_offset** | Trigger on (data & mask) = match. If mask = 0, trigger is disabled. | 0, 1, SAMPLES / 2 |
| **HOLDOFF 2\_to\_2048** | Set the minimum number of samples required from trigger to trigger. | SAMPLES |
| **data** | Numerical data is applied LSB-first to the channels. |  |
| **CLEAR** | Clear the sample buffer and display, wait for new data. |  |
| **SAVE {WINDOW} 'filename'** | Save a bitmap file (.bmp) of either the entire window or just the display area. |  |
| **CLOSE** | Close the window. |  |

\* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

The LOGIC display can be used to display data that was captured at high speed. In the example below, the P2 is generating 8-N-1 serial at 333 Mbaud using a smart pin. This bit stream can be captured by the streamer. On every clock, the streamer will record the smart pin's IN signal and its output state, as read from an adjacent pin. Every time it gets four two-bit sample sets, it does an RFBYTE to save them to hub RAM, forming contiguous bytes, words, and longs. By invoking the LONGS\_2BIT packed-data mode, we can have the LOGIC display unpack the two-bit sample sets from longs, yielding 16 sets per long.

## 13.9) Scope Display

**SCOPE Display Oscilloscope with 1..8 channels, can trigger on level with hysteresis**

|  |  |  |
| --- | --- | --- |
| |  | | --- | | CON \_clkfreq = 100\_000\_000  PUB go() | a, af, b, bf    debug(`SCOPE MyScope)    debug(`MyScope 'FreqA' -1000 1000 100 136 15 MAGENTA)    debug(`MyScope 'FreqB' -1000 1000 100 20 15 ORANGE)    debug(`MyScope TRIGGER 0 HOLDOFF 2)    repeat      a := qsin(1000, af++, 200)      b := qsin(1000, bf++, 99)      debug(`MyScope `(a,b))      waitus(200) | | https://lh3.googleusercontent.com/uXTurcj1Wj0OjMtfydFvs9mnDjZWcY2n76iU2XtiXqiTlz_Nz1n8HG1UXpjxjgqNdaKoGk2UQ30Uajw6rSuCTS82PK5YrMpusENLgrzLr0trFccvGd1mcXbY5RHfm1grFTxXZAy6 |

|  |  |  |
| --- | --- | --- |
| **SCOPE Instantiation** | **Description** | **Default** |
| **TITLE 'string'** | Set the window caption to 'string'. | <none> |
| **POS left top** | Set the window position. | 0, 0 |
| **SIZE width height** | Set the display size (32..2048 x 32..2048) | 255, 256 |
| **SAMPLES 16\_to\_2048** | Set the number of samples to track and display. | 256 |
| **RATE 1\_to\_2048** | Set the number of samples (or triggers, if enabled) before each display update. | 1 |
| **DOTSIZE 0\_to\_32** | Set the dot size in pixels for showing exact sample points. | 0 |
| **LINESIZE 0\_to\_32** | Set the line size in half-pixels for connecting sample points. | 3 |
| **TEXTSIZE 6\_to\_200** | Set the legend text size. | editor text size |
| **COLOR back\_color {grid\_color}** | Set the background and grid colors \*. | BLACK, GREY 4 |
| **packed\_data\_mode** | Enable packed-data mode. See description at end of this section. | <none> |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **SCOPE Feeding** | **Description** | **Default** |
| **'name' {min {max {y\_size {y\_base {legend {color}}}}}}** | Set first/next channel name, min value, max value, y size, y base, legend, and color \*. Legend is %abcd, where %a to %d enable max legend, min legend, max line, min line. | full, no legend, default color |
| **TRIGGER channel {arm\_level {trigger\_level {offset}}}** | Set the trigger channel, arm level, trigger level, and right offset. If channel=-1, disabled. | -1, -1, 0, width / 2 |
| **HOLDOFF 2\_to\_2048** | Set the minimum number of samples required from trigger to trigger. | SAMPLES |
| **data** | Numerical data is applied to the channels in ascending order. |  |
| **CLEAR** | Clear the sample buffer and display, wait for new data. |  |
| **SAVE {WINDOW} 'filename'** | Save a bitmap file (.bmp) of either the entire window or just the display area. |  |
| **CLOSE** | Close the window. |  |

\* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

## 13.10) Scope\_XY Display

**SCOPE\_XY Display XY oscilloscope with 1..8 channels, persistence of 1..512 samples, polar mode, log scale mode**

|  |  |  |
| --- | --- | --- |
| |  | | --- | | CON \_clkfreq = 100\_000\_000  PUB go() | i    debug(`SCOPE\_XY MyXY RANGE 500 POLAR 360 'G' 'R' 'B')    repeat      repeat i from 0 to 500        debug(`MyXY `(i, i, i, i+120, i, i+240))        waitms(5) | | https://lh6.googleusercontent.com/i3oOr7WWJzUCCRyxgL-Jd1OM4mu-9aYOZaNkJCkpYJoYlIQD09yUMBbqn3PacKKH_Ej9XVKBkj61Rldj2gZ03spfBg0qjXvUxrToYJQ_UQ_wPOplljoVKSK5kQNuy33d0lBzADfL |

|  |  |  |
| --- | --- | --- |
| **SCOPE\_XY  Instantiation** | **Description** | **Default** |
| **TITLE 'string'** | Set the window caption to 'string'. | <none> |
| **POS left top** | Set the window position. | 0, 0 |
| **SIZE radius** | Set the display radius in pixels. | 128 |
| **RANGE 1\_to\_7FFFFFFF** | Set the unit circle radius for incoming data | $7FFFFFFF |
| **SAMPLES 0\_to\_512** | Set the number of samples to track and display with persistence. Use 0 for infinite persistence. | 256 |
| **RATE 1\_to\_512** | Set the number of samples before each display update. | 1 |
| **DOTSIZE 2\_to\_20** | Set the dot size in half-pixels for showing sample points. | 6 |
| **TEXTSIZE 6\_to\_200** | Set the legend text size. | editor text size |
| **COLOR back\_color {grid\_color}** | Set the background and grid colors \*. | BLACK, GREY 4 |
| **POLAR {twopi {offset}}** | Set polar mode, twopi value, and offset. For a twopi value of $100000000 or -$100000000, use 0 or -1. | $100000000, 0 |
| **LOGSCALE** | Set log-scale mode to magnify points within the unit circle. | <off> |
| **'name' {color}** | Set the first/next channel name and optionally assign it a color \*. | default color |
| **packed\_data\_mode** | Enable packed-data mode. See description at end of this section. | <none> |
| **SCOPE\_XY Feeding** | **Description** | **Default** |
| **x y** | X-Y data pairs are applied to the channels in ascending order. In polar mode, x=length and y=angle. |  |
| **CLEAR** | Clear the sample buffer and display, wait for new data. |  |
| **SAVE {WINDOW} 'filename'** | Save a bitmap file (.bmp) of either the entire window or just the display area. |  |
| **CLOSE** | Close the window. |  |

\* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

## 13.11) TERM Display

Terminal for displaying text

|  |  |  |
| --- | --- | --- |
| |  | | --- | | CON \_clkfreq = 10\_000\_000  PUB go() | i    debug(`TERM MyTerm SIZE 9 1 TEXTSIZE 40)    repeat      repeat i from 50 to 60        debug(`MyTerm 1 'Temp = `(i)')        waitms(500) | | https://lh4.googleusercontent.com/miSmPj3o0E_WQ34NWq6j01CefLtJSIEacK4HfB9wILJLOO7oRe3v-C2QSesbGmvWsai9fG9ZDR3R12e6iC2Uc_R4YAN116ctebHg8hAULpi8ppXWTIX824p8uddk_fRaeVU2JWCF |

|  |  |  |
| --- | --- | --- |
| **TERM  Instantiation** | **Description** | **Default** |
| **TITLE 'string'** | Set the window caption to 'string'. | <none> |
| **POS left top** | Set the window position. | 0, 0 |
| **SIZE columns rows** | Set the number of terminal columns (1..256) and terminal rows (1..256). | 40, 20 |
| **TEXTSIZE size** | Set the terminal text size (6..200). | editor text size |
| **COLOR text\_color back\_color ...** | Set text-color and background-color combos #0..#3. \* | default colors |
| **BACKCOLOR color** | Set the display background color. \* | BLACK |
| **UPDATE** | Set UPDATE mode. The display will only be updated when fed an 'UPDATE' command. | automatic update |
| **TERM Feeding** | **Description** | **Default** |
| **character** | 0 = Clear terminal display and home cursor.  1 = Home cursor.  2 = Set column to next character value.  3 = Set row to next character value.  4 = Select color combo #0.  5 = Select color combo #1.  6 = Select color combo #2.  7 = Select color combo #3.  8 = Backspace.  9 = Tab to next 8th column.  13+10 or 13 or 10 = New line.  32..255 = Printable character. |  |
| **'string'** | Print string. |  |
| **CLEAR** | Clear the display to the background color. |  |
| **UPDATE** | Update the window with the current text screen. Used in UPDATE mode. |  |
| **SAVE {WINDOW} 'filename'** | Save a bitmap file (.bmp) of either the entire window or just the display area. |  |
| **CLOSE** | Close the window. |  |

\* Color is a modal value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

# 14.0) Program Structure

## 14.0.1) Propeller Tool (IDE)

The “Propeller Tool” is Parallax IDE (integrated development environment) allows the programs to have code and document comments to organize the documentation:

‘ single line code comment (apostrophe)

‘’ single line document comment (two apostrophe not quotation)

{…} Multi Line Code Comment

{{…}} Multi Line document comment

The “Propeller Tool” Compiles the user program from “Top Object” selected. Objects are compiled from “Libray” file folder and the current directory the “Top Object” is stored in (current working directory)

The default template file under “EDIT→PREFERNCES” can be used to customize User definitions for “New Projects”.

There are several programming tools other than the “Propeller Tool” but this document will only use the “Propeller Tool” with Parallax SPIN and PASM (Propeller Assembly Machine Language).

## 14.0.2) P2 Memory Organization

Cogs use 20-bit addresses for program counters (PC). This affords an execution space of up to 1MB.  Depending on the value of a cog's PC, an instruction will be fetched from either its register RAM, its lookup RAM, or the hub RAM.

|  |  |  |  |
| --- | --- | --- | --- |
| PC Address | Instruction Source | Memory Width | PC Increment |
| **$00000..$001FF** | **cog register RAM** | **32 bits** | **1** |
| **$00200..$003FF** | **cog lookup RAM** | **32 bits** | **1** |
| **$00400..$FFFFF** | **hub RAM** | **8 bits** | **4** |

**REGISTER EXECUTION**

When the PC is in the range of $00000 and $001FF, the cog is fetching instructions from cog register RAM.  This is commonly referred to as "cog execution mode."  There is no special consideration when taking branches to a cog register address.

application

**LOOKUP EXECUTION**

When the PC is in the range of $00200 and $003FF, the cog is fetching instructions from cog lookup RAM.  This is commonly referred to as "lut execution mode."  There is no special consideration when taking branches to a cog lookup address,

**HUB EXECUTION**

When the PC is in the range of $00400 and $FFFFF, the cog is fetching instructions from hub RAM.  This is commonly referred to as "hub execution mode."  When executing from hub RAM, the cog employs the FIFO hardware to spool up instructions so that a stream of instructions will be available for continuous execution.  Branching to a hub address takes a minimum of 13 clock cycles.  If the instruction is not aligned to a slice, one additional clock cycle is required.

While in hub execution mode, the FIFO cannot be used for anything else. So, during hub execution these instructions must be avoided:

RDFAST / WRFAST / FBLOCK

RFBYTE / RFWORD / RFLONG / RFVAR / RFVARS

WFBYTE / WFWORD / WFLONG

XINIT / XZERO / XCONT - when the streamer mode engages the FIFO

It is not possible to execute code from hub addresses $00000 through $003FF, as the cog will instead read instructions from the cog register or lookup RAM as indicated above.

## 14.0.3) COG RAM

Each cog has a primary 512 x 32-bit dual-port RAM, which can be used in multiple ways:

* Direct/Register access
* As a source of program instructions

**GENERAL PURPOSE REGISTERS**

RAM registers $000 through $1EF are general-purpose registers for code and data usage.

**DUAL-PURPOSE REGISTERS**

RAM registers $1F0 through $1F7 may either be used as general-purpose registers, or may be used as special-purpose registers if their associated functions are enabled.

**$1F0 RAM / IJMP3 interrupt call   address for INT3**

**$1F1 RAM / IRET3 interrupt return address for INT3**

**$1F2 RAM / IJMP2 interrupt call   address for INT2**

**$1F3 RAM / IRET2 interrupt return address for INT2**

**$1F4 RAM / IJMP1 interrupt call   address for INT1**

**$1F5 RAM / IRET1 interrupt return address for INT1**

**$1F6 RAM / PA CALLD-imm return, CALLPA parameter, or LOC address**

**$1F7 RAM / PB CALLD-imm return, CALLPB parameter, or LOC address**

**SPECIAL-PURPOSE REGISTERS**

Each cog contains 8 special-purpose registers that are mapped into the RAM register address space from $1F8 to $1FF.  In general, when specifying an address between $1F8 and $1FF, the instruction is accessing a special-purpose register, *not* just the underlying RAM.

**$1F8 PTRA pointer A to hub RAM**

**$1F9 PTRB pointer B to hub RAM**

**$1FA DIRA output enables for P31..P0**

**$1FB DIRB output enables for P63..P32**

**$1FC OUTA output states for P31..P0**

**$1FD OUTB output states for P63..P32**

**$1FE INA \* input states for P31..P0**

**$1FF INB \*\* input states for P63..P32**

**\* also debug interrupt call address**

**\*\* also debug interrupt return address**

**LOOKUP RAM**

Each cog has a secondary 512 x 32-bit dual-port RAM, which can be used in multiple  ways:

* Load/Store access
* As a source or destination for the streamer hardware
* As a lookup table for bytecode execution
* As a data source for smart pins
* As a "RAM sharing" mechanism between paired cogs
* As a source of program instructions (see [COGS > INSTRUCTION MODES > LOOKUP EXECUTION](https://docs.google.com/document/d/1gn6oaT5Ib7CytvlZHacmrSbVBJsD9t_-kmvjd7nUR6o/edit#heading=h.sip3znh0gf5f))

NOTE: The term "lookup" (and "lut", which is short for "look-up table") is due to historical usage in the original Propeller microcontroller.  This RAM can still be used in a "lookup" context, but can also be used for many other purposes, as indicated above.

**PASM Communication Registers**

Each of these cog registers can be referenced by name PR0-PR7

**PR0 $1D8**

**PR1 $1D9**

**PR2 $1DA**

**PR3 $1DB**

**PR4 $1DC**

**PR5 $1DD**

**PR6 $1DE**

**PR7 $1DF**

## 14.0.4) Program Blocks

Spin2 programs are built from one or more objects. Objects are files which contain at least one public method, along with optional constants, child objects, variables, additional methods, and data. Objects are assembled together into a top-level object with an internal hierarchy of sub-objects. Each object instance, at run-time, gets its own set of variables, as defined by the object, to maintain its unique operating state.

Different parts of an object are declared within blocks, which all begin with 3-letter block identifiers.

The compiler can actually generate PASM-only programs, as well as Spin2+PASM programs, depending upon which blocks are present in the .spin2 file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Block Identifier** | **Block Contents** | **Spin2+PASM**  **Programs** | **PASM-only**  **Programs** |
| **CON** | Constant declarations (CON is the initial/default block type) | Permitted | Permitted |
| **OBJ** | Child-object instantiations | Permitted | Not Allowed |
| **VAR** | Variable declarations | Permitted | Not Allowed |
| **PUB** | Public method for use by the parent object and within this object | Required | Not Allowed |
| **PRI** | Private method for use within this object | Permitted | Not Allowed |
| **DAT** | Data declarations, including PASM code | Permitted | Required |

Here are some minimal Spin2 and PASM-only programs. If you copy and paste these into PNut.exe, you can hit F10 to run them.

|  |  |
| --- | --- |
| Minimal  Spin2  Program | **PUB MinimalSpin2Program()         'first PUB method executes**  **REPEAT**  **PINWRITE(63..56, GETRND())    'write a random pattern to P63..P56**  **WAITMS(100)                   'wait 1/10th of a second, loop** |
| Minimal  PASM  Program | **DAT     ORG                       'start PASM at hub $00000 for cog $000**  **loop    DRVRND  #56 ADDPINS 7     'write a random pattern to P63..P56**  **WAITX   ##clkfreq\_/10     'wait 1/10th of a second, loop**  **JMP     #loop** |

Here is a Spin2 program which contains every block type.

|  |  |
| --- | --- |
| All-Block  Spin2  Program | **CON \_clkfreq = 297\_000\_000                      'set clock frequency**  **OBJ vga : "VGA\_640x480\_text\_80x40"              'instantiate vga object**  **VAR time, i                                     'declare object-wide variables**  **PUB go()                                        'this first public method executes, cog stops after**  **vga.start(8)                                  'start vga on base pin 8**  **SEND := @vga.print                            'establish SEND pointer**  **SEND(4, $004040, 5, $00FFFF)                  'set light cyan on dark cyan**  **time := GETCT()                               'capture time**  **i := @text                                    'print file to vga screen**  **REPEAT @textend-i**  **SEND(byte[i++])**  **time := GETCT() - time                        'capture time delta in clock cycles**  **time := MULDIV64(time, 1\_000\_000, clkfreq)    'get time delta in microseconds**  **SEND(12, "Time elapsed during printing was ", dec(time), " microseconds.")  'print time delta**  **PRI dec(value) | flag, place, digit             'private method prints decimals, three local variables**  **flag~                                         'reset digit-printed flag**  **place := 1\_000\_000\_000                        'start at the one-billion's place and work downward**  **REPEAT**  **IF flag ||= (digit := value / place // 10) || place == 1      'print a digit?**  **SEND("0" + digit)                                           'yes**  **IF LOOKDOWN(place : 1\_000\_000\_000, 1\_000\_000, 1\_000)        'also print a comma?**  **SEND(",")                                                 'yes**  **WHILE place /= 10                             'next place, done?**  **DAT**  **text FILE "VGA\_640x480\_text\_80x40.txt"       'include raw file data for printing**  **textend** |

## 14.1) CON Block

Symbolic constants are global to the object. If an object reference is declared in another object constants Of the child object can be referenced using :

**OBJ**

**Num : “Numbers” -🡪 Num.Constant\_Symbol**

**Syntax 1 Symbol = Expression (constants)**

Symbol –desired name of constant

Expression –any valid integer for floating point, or constant algebraic expression

Note: Constant can be used in algebraic expression but must be previously defined.

CON ‘Direct Assignment

**EnableFlow = 8            'single assignments LONG data type**

**x = 5, y = -5, z = 1      'comma-separated assignments**

**HalfPi = 1.5707963268     'single-precision float values**

**xy = x\*y**

Syntax 2 #Symbol (enumerated constants

CON ‘Direct Assignment

#3, a,b,c 'a=3 b=4 c=5precision float values

### 14.1.1) CON  Compiler Enumeration Step option #conVar[step]

**#0,a,b,c,d           'a=0, b=1, c=2, d=3    (start=0, step=1)**

**#1,e,f,g,h           'e=1, f=2, g=3, h=4    (start=1, step=1)**

**#4[2],i,j,k,l        'i=4, j=6, k=8, l=10   (start=4, step=2)**

**#-1[-1],m,n,p    'm=-1, n=-2, p=-3      (start=-1, step=-1)**

**#true,on,off 'true = $FFFF\_FFFF ,on = $FFFF\_FFFF,off = %0**

**#true +true,on,off 'true = $FFFF\_FFFE ,on = $FFFF\_FFFE,off = $FFFF\_FFFF**

### 14.1.2) Vertical Constant Enumeration

**Note: Constant[Step Increment] changes the step value for next constant if not defined the default step is 1 for example conVar[3] would increase next step value by 3**

**#16                'start=16, step=1 set enumeration**

**q                    'q=16**

**r[0]                 'r=17   ([0] is a step multiplier)**

**s                    's=17**

**t                    't=18**

**u[2]                'u=19   ([2] is a step multiplier)**

**v                    'v=21**

**w                    'w=22**

**#16[2] 'start=16 step=2 set enumeration**

**a 'a= 16**

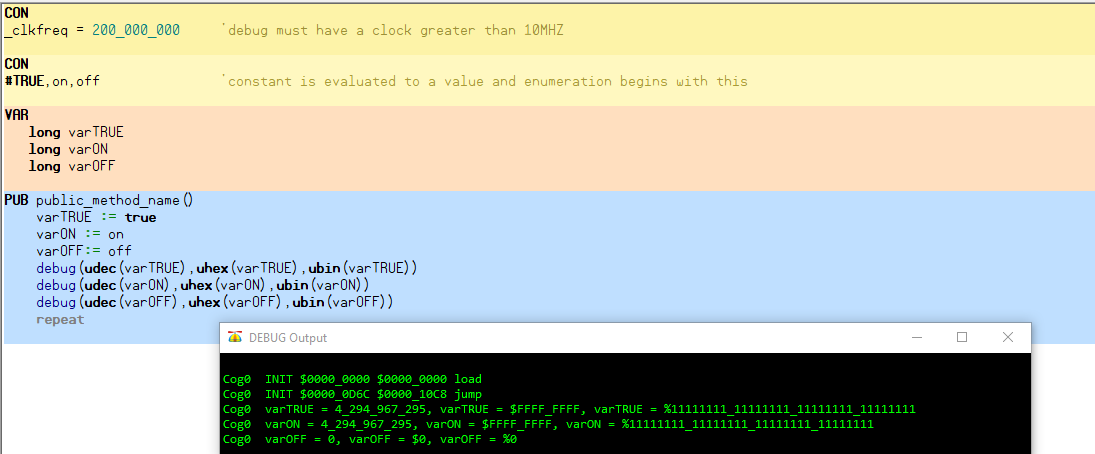
**b 'b= 18**

**c 'c= 20**

**d 'd= 22**

**e 'e= 24**

### 14.1\_Example\_WRD\_Constant\_Enumeration



The #TRUE directive causes the enumeration constant value to be evaluated and then assigned to constants.

The case of “true or True or tRUe” all are legal and will resolve to **TRUE**. Case does not matter in spin2.

## 14.2) OBJ Block

**Symbol<[count]>: “Object\_Name”**

Symbol –name for Object

count –number of objects to be made

Object\_Name –object filename without extension

**OBJ**

NUM: “Numbers”

Term: “Terminal”

Public Methods are accessed alias dot method: NUM.ToStr

|  |  |
| --- | --- |
| OBJ  Child-Object  Instantiations | **OBJ  vga       : "VGA\_Driver"     'instantiate "VGA\_Driver.spin2" as "vga"**  **mouse     : "USB\_Mouse"      'instantiate "USB\_Mouse.spin2" as "mouse"**  **v[16]     : "VocalSynth"     'instantiate an array of 16 objects**  **'..v[0] through v[15]** |

### 14.2.1) Accessing Constants and Pub Method from Objects

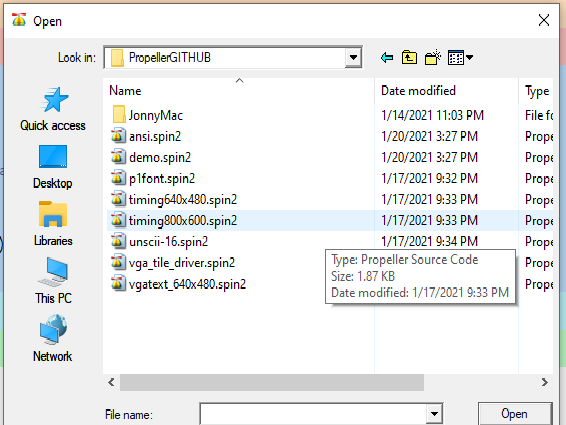
From within a parent-object method, a child-object method can be called by using the syntax:

**object\_name.method\_name({any\_parameters})**

From within a parent-object method, a child-object constant can be referenced by using the syntax:

**object\_name.constant\_name**

### 14.2.2) VGA Object

Using Demo.spin for VGA output (No resistor wire Pins directly to VGA Test Plug)

Set clock to 200\_000\_000 scren size to 600x800 set “CELL\_SIZE = 4” call demo.spin2.

VGA Connector (No dropping resistors uses DAC to generate signal)

Pin- Function- VGA Pin - Wirecolour (VGA Test Plug)

P48 H 13 Torquise-Yel

P49 B 3 RED-yel

P50 G 2 BROWN-yel

P51 R 1 BLACK-yel

P52 V 14 BLUE/White-yel

GND 5,6,7,8,10 purple blue green yellow grey -blk

## 14.3) VAR Block

VAR is an object Block where variables are defined of 3 data types Byte,Word and Long.

VAR

byte a,b,c '8bits

word a,b,c '16bits

long a,b,c '32bits

### 143.1) VAR Compiler Directive

VAR

byte temp ‘temp is a VAR of type byte at address @temp

byte temp[20] ‘reserve 20 bytes at address @temp[0]

Program Usage

Some\_Var ≔ Byte[@MyData] ‘read 1 byte from address @MyData

Some\_Var ≔ Byte[@MyData][Index++] ‘read 1 byte from @MyData+Index offset

VAR

word temp ‘temp is a VAR of type two byte at address @temp

word temp[20] ‘reserve 40 bytes at address @temp[0]

### 14.3.2) Program Usage

Some\_Var ≔ word[@MyData] ‘read 2 byte from address @MyData

Some\_Var ≔ word[@MyData][Index++] ‘read 2 byte from @MyData+Index offset

VAR

Long temp ‘temp is a VAR of type byte at address @temp

Long temp[20] ‘reserve 80 bytes at address @temp[0]

Program Usage

Some\_Var ≔ Long[@MyData] ‘read 4 byte from address @MyData

Some\_Var ≔ Long[@MyData][Index++] ‘read 4 byte from @MyData+Index offset

### 14.3.3) Variables that are pre-defined and Permanent

In Spin2, there are both user-defined and permanent variables. The user-defined variable sources are listed below and the permanent variables are shown in the table.

VAR variables (hub)

PUB/PRI parameters, return values, and local variables (hub)

DAT symbols (hub)

Cog registers

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Variables**  **(all LONG)** | **Variable**  **Name** | **Address**  **or Offset** | **Description** | **Useful in**  **Spin2** | **Useful in**  **Spin2-PASM** | **Useful in**  **PASM-Only** |
| Hub Locations | **CLKMODE**  **CLKFREQ** | **$00040**  **$00044** | Clock mode value  Clock frequency value | Yes  Yes | Yes  Yes | No  No |
| Hub VAR | **VARBASE** | +0 | Object base pointer, @VARBASE is VAR base, used by method-pointer calls | Maybe | No | No |
| Cog Registers | **PR0**  **PR1**  **PR2**  **PR3**  **PR4**  **PR5**  **PR6**  **PR7**  **IJMP3**  **IRET3**  **IJMP2**  **IRET2**  **IJMP1**  **IRET1**  **PA**  **PB**  **PTRA**  **PTRB**  **DIRA**  **DIRB**  **OUTA**  **OUTB**  **INA**  **INB** | **$1D8**  **$1D9**  **$1DA**  **$1DB**  **$1DC**  **$1DD**  **$1DE**  **$1DF**  **$1F0**  **$1F1**  **$1F2**  **$1F3**  **$1F4**  **$1F5**  **$1F6**  **$1F7**  **$1F8**  **$1F9**  **$1FA**  **$1FB**  **$1FC**  **$1FD**  **$1FE**  **$1FF** | Spin2 <-> PASM communication    Interrupt JMP's and RET's    Pointer registers  Data pointer passed from COGINIT  Code pointer passed from COGINIT  Output enables for P31..P0  Output enables for P63..P32  Output states for P31..P0  Output states for P63..P32  Input states from P31..P0  Input states from P63..P32 | Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  No  No  No  No  No  No  No  No  No  No  Yes  Yes  Yes  Yes  Yes  Yes | Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes | No  No  No  No  No  No  No  No  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes |

In Spin2, all variables can be indexed and accessed as bitfields. Additionally, symbolic hub variables can have BYTE/WORD/LONG size overrides:

|  |  |  |
| --- | --- | --- |
| **Variable Usage** | **Example** | **Description** |
| Plain | **AnyVar**  **HubVar.WORD**  **BYTE[address]**  **REG[register]** | Hub or permanent register variable  Hub variable with BYTE/WORD/LONG size override  Hub BYTE/WORD/LONG by address  Register, 'register' may be symbol declared in ORG section |
| With Index | **AnyVar[index]**  **HubVar.BYTE[index]**  **LONG[address][index]**  **REG[register][index]** | Hub or permanent register variable with index  Hub variable with size override and index  Hub BYTE/WORD/LONG by address with index  Register with index |
| With Bitfield | **AnyVar.[bitfield]**  **HubVar.LONG.[bitfield]**  **WORD[address].[bitfield]**  **REG[register].[bitfield]** | Hub or permanent register variable with bitfield  Hub variable with size override and bitfield  Hub BYTE/WORD/LONG by address with bitfield  Register with bitfield |
| With Index and Bitfield | **AnyVar[index].[bitfield]**  **HubVar.BYTE[index].[bitfield]**  **LONG[address][index].[bitfield]**  **REG[register][index].[bitfield]** | Hub or permanent register variable with index and bitfield  Hub variable with size override, index, and bitfield  Hub BYTE/WORD/LONG by address with index and bitfield  Register with index and bitfield |

### 14.3.4) Accessing Bytes of Larger-Sized Variables

Var

Word WordVar

Long LongVar

Pub Main()

WordVar.Byte[0] ≔ 0 ‘ 0000\_0000

WordVar.Byte[1] ≔ 100 ‘0110\_0100 = 64 + 32 + 4 =100

‘0110\_0100\_0000\_0000 = 16384 + 8192 + 1024 = 25600

LongVar.byte[0] ≔ 25

LongVar.byte[1] ≔ 50

LongVar.byte[2] ≔ 75

LongVar.byte[3] ≔ 100

‘01100100\_01001011\_00110010\_00011001 = 1,682,649,625

## 14.4) PUB Block

**PUB methodname({parameter{,...}}) {: result{,...}} {| {ALIGNW/ALIGNL} {BYTE/WORD/LONG} localvar{[count]}{,...}}**

**PUB ObJectMethod(a,b,c) : result01,result02,result03)| var01,var02,var03**

a,b,c are preloaded passed parameters from method call all longs

result01,result02,result03 all longs are cleared to zero before entry and return with a result from method, you must be able to receive the same number of result parameters.

var01,var02,var03 all lonfs are local parameters and could be pre-loaded from a previous method call you must manually clear this in the method unless you are using this retentive feature.

**Note:** In the P1 result was automatically understood as the return method value. In the P2 if no return parameter is declared in the method no return values are generated , ie result is not automatic you must declare if you want a return value.

### 14.4.1\_Example\_WRD\_Multiple\_Result\_Method

{14.4.1\_Example\_WRD\_Multiple\_Result\_Method}

CON

\_clkfreq = 200\_000\_000 'debug must have a clock greater than 10MHZ

PUB main():result01,result02,result03 | a,b,c,x 'method run after boot spin interperter loads

x := 0

a := b := c := 0

repeat until x == 10

a := a + x

b := b + x

c := c + x

result01,result02,result03 := MultipleReturnResult(a,b,c)

debug(udec(result01),udec(result02),udec(result03))

x += 1

PUB MultipleReturnResult(a,b,c):result01,result02,result03|x,y

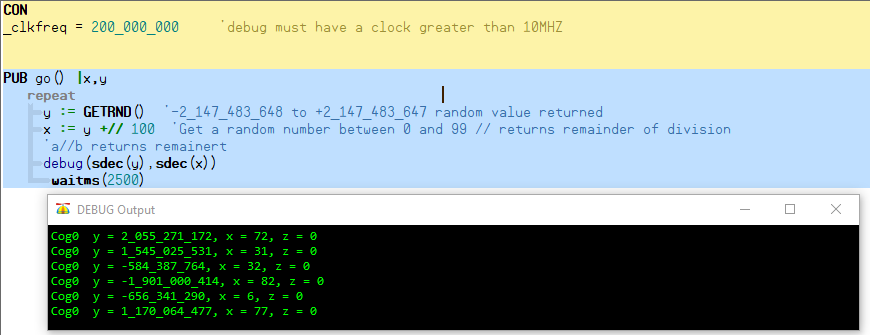
result01 := 123 + a

result02 := 456 + b

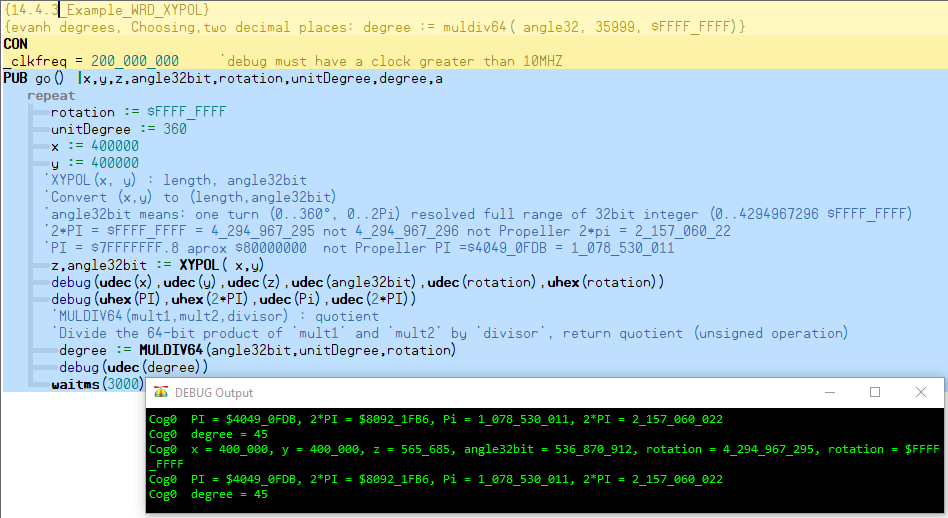
result03 := 789 + c

Note: The common naming convention do not clash because they are generated local to the method, result01,result02,result03 are separate and independent. Compiler keeps track.

### 14.4.2\_Example\_WRD\_GETRND

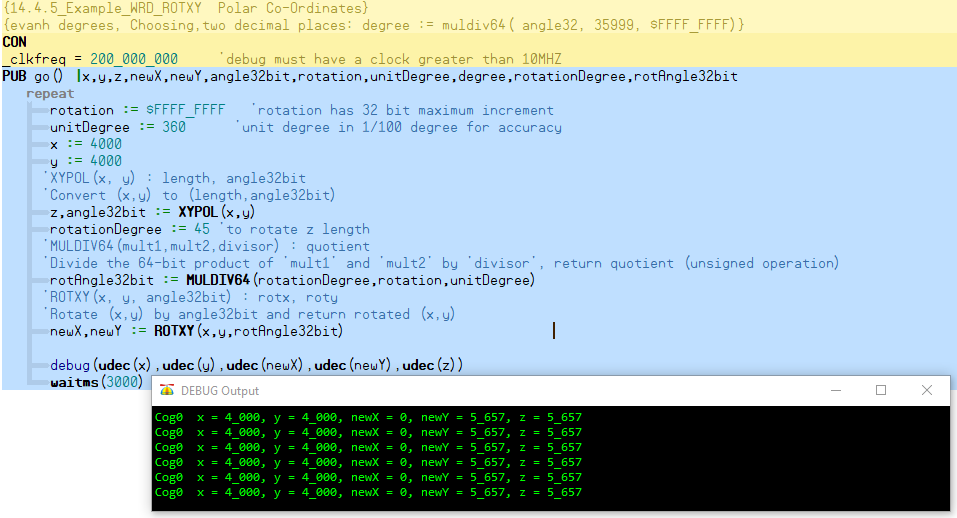


### 14.4.3\_Example\_WRD\_XYPOL (Polar Co-ordinates xy to length,angle32bit)

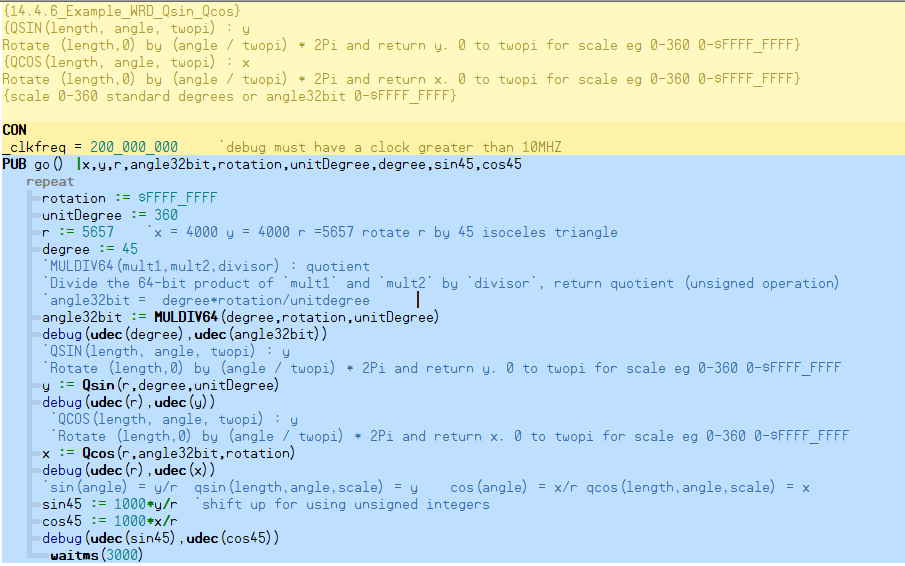


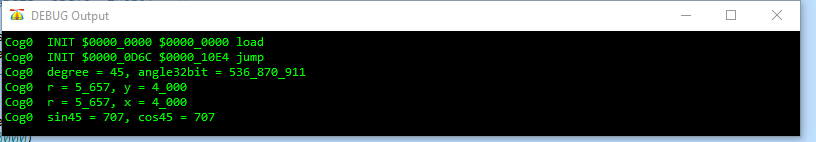
### 14.4.4\_Example\_WRD\_POLXY(Polar Co-Ordinates Length,angle32bit to xy)

### 14.4.5\_Example\_WRD\_ROTXY (Polar Co-Ordinate Rotation)



### 14.4.6\_Example\_WRD\_QSIN\_QCOS





## 14.5) PRI Block

**PRI methodname({parameter{,...}}) {: result{,...}} {| {ALIGNW/ALIGNL} {BYTE/WORD/LONG} localvar{[count]}{,...}}**

The private methods are internal to the object and can not be referenced from outside the object.

result –is the return value

PRI privateMethod()

Code

## 14.6) Dat Block

Reserved Memory DAT block declares data block

**<Symbol> Alignment <Size> <Data> ‘ reserved memory**

Symbol –optional name for the reserved space

Alignment <Size> –the byte alignment for reserved data byte,word,long (1byte,2byte,4byte)

Data –constant expression or comma separated variable or quoted strings treated as same

**<Symbol> <Condition> Instruction <Effects> ‘Propeller Assembly Code**

Symbol –optional name for the command line

Condition –flag condition C Carry or Z Zero IF\_C, IF\_NC, IF\_Z, IF\_NZ

Instruction –assembly language Instruction eg. ADD,MOV,etc

Effects –effects that cause the result to be written when executed WR,WC,WZ

### 14.6.1) Common Dat Declaration

DAT

MyData byte 64,’$AA, 55 ‘creates data table

MyString byte “Hello World”,0 ‘Zero 0 terminated string

Pub GetData |Temp

Temp ≔ MyData[0] ‘get first byte of Data Table

Pub GetData | Temp

Tem ≔ BYTE[@MyData][0]

DAT

MyData word 40\_000, $BB50

MyList word long $FF995544, long 1000 ‘needs clarification book may be wrong pg332

DAT

MyData Long 640\_000, $BB50

MyList byte long $FF995544, long 1000 ‘needs clarification book may be wrong pg237

|  |
| --- |
| DAT  Data Pointers |
| **DAT**  **Str0        BYTE    "Monkeys",0   'strings with symbols**  **Str1        BYTE    "Gorillas",0**  **Str2        BYTE    "Chimpanzees",0**  **Str3        BYTE    "Humanzees",0**  **StrList     WORD    @Str0 'in Spin2, these are offsets of strings relative to start of object**  **WORD    @Str1 'in Spin2, @@StrList[i] will return address of Str0..Str3 for i = 0..3**  **WORD    @Str2 'in PASM-only programs, these are absolute addresses of strings**  **WORD    @Str3 '(use of WORD supposes offsets/addresses are under 64KB)** |

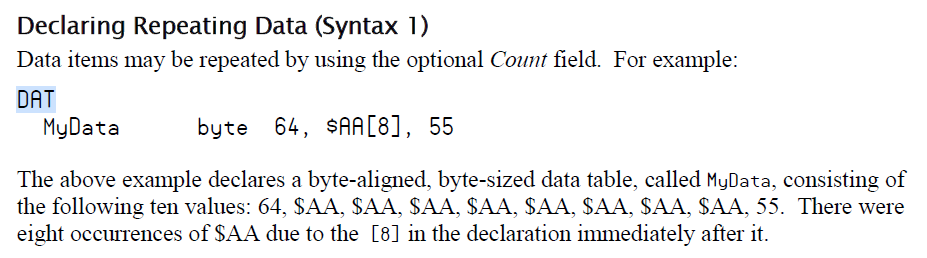
### 14.6.2) Filling Data Tables

symbol data\_type fill\_value[array\_length]

DAT

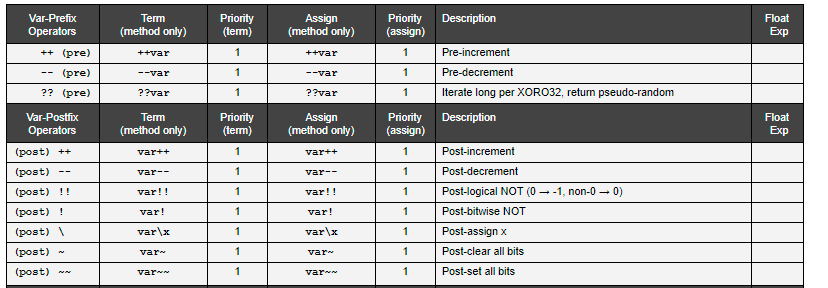
Custom long -1[C\_CHARS]

That is the equivalent of defining C\_CHARS (length) longs filling the value of -1 in the DAT block -- it's creating a DAT array with all values initialized to -1. You can address each long individully using Custom[idx] where idx is 0 to 7. If you need the address of the array you can get it with @Custom. If you later need 10 longs, you only have to change the definition of C\_CHARS.



# 15.0) Operators

## 15.1) Pre and Post Operators



## 15.2) Operator ?? PsedoRandomNumberGenerator

PRNG of XOR032 requires variable to be non zero initially ["Xorshift RNGs" by George Marsaglia](http://www.jstatsoft.org/v08/i14/paper) describes a very efficient system for generating high-quality random numbers using very little compute and storage.

- <https://forums.parallax.com/discussion/168188/xoroshiro-random-number-generator/p1>

Here is the xoroshiro++ pseudo code:

;{s1,s0} = state (input **and** output)

;prn = pseudo-random number (output)

;tmp **and** prn can be the same register

;xoroshiro+

xor s1,s0 ;s1 = s1 ^ s0

mov tmp,s1

rol s0,a ;s0 = s0 rol a

shl tmp,b ;tmp = (s1 ^ s0) shl b

xor s0,tmp

xor s0,s1 ;s0 = s0 rol a ^ (s1 ^ s0) shl b ^ s1 ^ s0

rol s1,c ;s1 = (s1 ^ s0) rol c

mov prn,s0

add prn,s1 ;prn = s0 + s1

;xoroshiro++ enhancement

## 15.3) Pre\_and\_Post\_Operators Examples

{{WRD\_Pre\_and\_Post\_Operators}}

CON

\_clkfreq = 300\_000\_000

VAR

long var01

long var02

PUB main():result01,result02 | x,y

{X++ Post increment after instruction}

x := 0

repeat until x== 10

y := x++ 'Post instruction increment

debug(udec(y),udec(x))

debug("x is incremented after the assignment to y")

waitms(1000)

{++X Pre increment before instruction}

x := 0

repeat until x== 10

y := ++x 'Pre instruction increment

debug(udec(y),udec(x))

debug("x is incremented before the assignment to y")

waitms(1000)

{X-- Post decrement after instruction}

x := 10

repeat until x== 0

y := x-- 'Post instruction increment

debug(udec(y),udec(x))

debug("x is decremented after the assignment to y")

waitms(1000)

{--X Pre decrement before instruction}

x := 10

repeat until x== 0

y := --x 'Pre instruction increment

debug(udec(y),udec(x))

debug("x is decremented before the assignment to y")

waitms(1000)

{??X RandomNumberGenerator RNG ?? of XOR032 requires variable to be non zero initially}

x := 3

y := ??x 'Pre instruction random out

debug(udec(y),udec(x))

debug("X is randomized prior to assignment y")

waitms(1000)

{X! post negate}

x := %10101010\_10101010\_11111111\_00000000

y := x!

debug(ubin(y),ubin(x))

debug("note bits are complemented individually")

waitms(1000)

{X!! post negate}

x := %10101010\_10101010\_11111111\_00000000

y := x!!

debug(ubin(y),ubin(x))

debug("note long word inverted to zero from non zero")

waitms(1000)

{y\x post negate}

x := 1234

y := 4321

debug(udec(var01),udec(y),udec(x))

var01 := y\x

debug(udec(var01),udec(y),udec(x))

debug("var01= y and y= x and x= x")

waitms(1000)

{x~ set x zero}

x := 1234

y := 0

var01 := 0

debug(udec(var01),udec(x))

var01 := x~

debug(udec(var01),udec(x))

debug("var01 = xorig= 1\_234 x final is 0")

waitms(1000)

{x~~ set x one}

x := 1234

y := 0

var01 := 0

debug(udec(var01),udec(x))

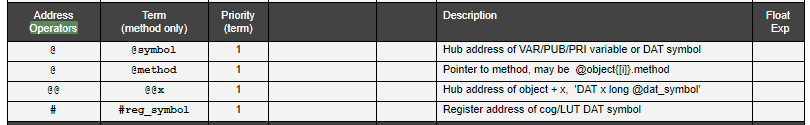
var01 := x~~

debug(udec(var01),udec(x),sdec(x))

debug("var01 = xorig= 1\_234 x final is -1")

waitms(1000)

## 15.2) Address Operators



{{WRD\_Dat\_Symbols\_Address}}

CON

\_clkfreq = 200\_000\_000

VAR

long var01

long var02

long var03[7]

PUB main():result01,result02 | x,y

debug(zstr(@name),zstr(@address),zstr(@country),zstr(@city))

repeat x from 0 to 6

var03[x] := x+2

debug(ubin\_long\_array(@var03,4))

debug(udec\_long\_array(@var03,7))

debug(udec\_long\_array(@Var03 + 0,1))

debug(udec\_long\_array(@Var03 + 4,1))

debug(udec\_long\_array(@Var03 + 8,1))

debug(udec\_long\_array(@Var03 + 12,1))

debug(udec\_long\_array(@Var03 + 16,1))

debug(udec\_long\_array(@Var03 + 20,1))

debug(udec\_long\_array(@Var03 + 24,1))

DAT

name byte "William Robert Drury",0

address byte "86 Hume Drive",0

country byte "Canada",0

city byte "Cambridge",0

## 15.3) Bitwise Decod and ENCOD

Note: |< Decode 0-31 Does not exist In P2 works for P1

DECOD create a binary bit set in accordance with bit number 0-31 BinPattern = **DECOD** PinNum

PinNum =31 BinPattern %10000000\_00000000\_00000000\_00000000

ENCOD create a Decimal number of the highest bit set in a given number PinNum = **ENCOD** BinPattern

BinPattern = %11111111\_11111111\_11111111\_11111111 PinNum = 31

Con

\_clkfreq = 200\_000\_000

Con

PinNum = 31

BitPattern = %00000000\_00000000\_00000000\_00000000

PinNum01 = 31

BitPattern01 = DECOD PinNum01

BitPattern02 = %00000000\_00000000\_00000000\_11111111

PinNum02 = ENCOD BitPattern02

Var

Long VarPinNum01

Long VarBitPattern01

Long VarPinNum02

Long VarBitPattern02

Pub main()

VarPinNum01 := PinNum01

VarBitPattern01 := BitPattern01

debug(UDEC(VarPinNum01),UBIN(VarBitPattern01))

VarPinNum02 := PinNum02

VarBitPattern02 := BitPattern02

debug(UDEC(VarPinNum02),UBIN(VarBitPattern02))

BinPattern := %00000000\_00000000\_00000000\_00000001

PinNum := 31

repeat 32

BinPattern := DECOD PinNum

debug(UDEC(PinNum),UBIN(BinPattern))

PinNum -= 1

waitms(500)

BinPattern := %00000000\_00000000\_00000000\_00000001

PinNum := 0

repeat 32

PinNum := ENCOD BinPattern

debug(UDEC(PinNum),UBIN(BinPattern))

BinPattern <<= 1

waitms(500)

# 16.0) Method Pointer

Method pointers point to a method and are then used to call that method indirectly. Method pointers are contained in a long data type. To establish a method pointer, you can assign a long variable using "@" before the method name. Note that there are no parentheses after the method name:

**LongVar := @SomeMethod                      'a method within the current object**

**LongVar := @SomeObject.SomeMethod           'a method within a child object**

**LongVar := @SomeOject[index].SomeMethod     'a method within an indexed child object**

An @method expression generates a 32-bit value which has two bit fields:

[31..20] = Index of the method, relative to the method's object base. The index of the first method will be twice the number of objects instantiated

[19..0] = Address of the method's VAR base. The method's VAR base, in turn, contains the address of the method's object base.

By putting the method's index and VAR base address together into the 32-bit value, and having the VAR base contain the method's object base address, a complete method pointer is established in a single long, which can be treated as any other variable.

To accommodate method pointers, each object instance reserves the first long of its VAR space for the object base address. When an @method expression executes, that first long is written with the object's base address.

## 16.1) SEND

SEND is a special method pointer which is inherited from the calling method and, in turn, conveyed to all called methods. It's purpose is to provide an efficient output mechanism for data.

SEND can be assigned like a method pointer, but it must point to a method which takes one parameter and has no return values:

**SEND := @OutMethod**

When used as a method, SEND will pass all parameters, including any return values from called methods, to the method SEND points to:

**SEND("Hello! ", GetDigit()+"0", 13)**

Any methods called within the SEND parameters will inherit the SEND pointer, so that they can do SEND methods, too:

**PUB Go()**

**SEND := @SetLED**

**REPEAT**

**SEND(Flash(),$01,$02,$04,$08,$10,$20,$40,$80)**

**PRI Flash() : x**

**REPEAT 2**

**SEND($00,$FF,$00)**

**RETURN $AA**

**PRI SetLED(x)**

**PINWRITE(56 ADDPINS 7, !x)**

**WAITMS(125)**

In the above example, the following values are output in repeating sequence: $00, $FF, $00, $00, $FF, $00, $AA, $01, $02, $04, $08, $10, $20, $40, $80 (but inverted for LEDs)

Though a called method inherits the current SEND pointer, it may change it for its own purposes. Upon return from that method, the SEND pointer will be back to what it was before the method was called. So, the SEND pointer value is propagated in method calls, but not in method returns.

SEND acts as a special type of method pointer, inherited from the calling method and, in turn, conveyed to all called methods. It provides an efficient output mechanism for data.  
  
You may assign SEND as you would any method pointer, but it must point to a method that 1. *takes one parameter* and 2. *has no return values*:

SEND := @OutMethod 'SEND points to OutMethod

SEND will pass all parameters, including any return values from called methods, to the method SEND points to:

**SEND**("Hello! ", GetDigit()+"0", 13)

Any methods called from within the SEND parameters, such as GetDigit() in the example above, will inherit the SEND pointer, so that they also may use the SEND method. The following code provides an example of SEND use. It sends 8-bit patterns of 0s and 1s to LEDS at pins P56 through P63:

PUB go()

  SEND := @SetLED

  REPEAT

    SEND($01, $02, $04, $08, $10, $20, $40, $80)

PRI SetLED(x)

  PINWRITE(56 ADDPINS 7, !x)

  WAITMS(125)

Note: LEDs on the P2 EVAL Board are driven by active-low signals.  
  
Within the go() method, the statement SEND := @SetLED, gives SEND the pointer to the SetLED method. This method satifies the requirement: only one variable and no return value. Note the SetLED method above includes a short delay so the LED patterns remain visible long enough so you can see them.  
  
Next the REPEAT loop executes the SEND($01, $02...) statement that transfers the first parameter $01 to the SetLED method. The LED at pin P56 turns on. When this method finishes, it returns control to the SEND statement, which then sends $02 to the SetLED method, which turns on the LED at P57. Each LED turns on and off in sequence again and again in the REPEAT loop.  
  
A second example shows how other methods can inherit the SEND pointer. An added method, Flash(), will turn all LEDs on and off. This method includes a SEND statement, too.

PUB go()

  SEND := @SetLED

  REPEAT

    SEND(Flash(), $01, $02, $04, $08, $10, $20, $40, $80)

PRI Flash() : x

  REPEAT 2

    SEND($00,$FF,$00)

  RETURN $AA

PRI SetLED(x)

  PINWRITE(56 ADDPINS 7, !x)

  WAITMS(125)

The program will call the Flash() method (in the first SEND() parameter) and will eventually pass the return value from Flash() to the SetLED() method (after Flash() has fully executed).  
  
First, the Flash() method will run and send its own values, $00, $FF, $00, to the LEDs two times. Then, if you watch the LEDs, they next display $AA next. Why?  
  
The Flash() method returns the value $AA to the SEND statement: SEND(Flash(),$01,$02... In effect the $AA value gets inserted in place of the call to Flash() in the list of parameters, making the whole program execution behave as if the SEND(Flash(), $01, $02...) had really been:  
  
SEND($00,$FF,$00)  
SEND($00,$FF,$00)  
SEND($AA, $01, $02...)  
  
The Flash() method inherited the SetLED() address and can use it independent of other uses in this program

## 8.2) RECV

RECV, like SEND, is a special method pointer which is inherited from the calling method and, in turn, conveyed to all called methods. It's purpose is to provide an efficient input mechanism for data.

RECV can be assigned like a method pointer, but it must point to a method which takes no parameters and returns a single value:

**RECV := @InMethod**

An example of using RECV:

**VAR i**

**PUB Go()**

**RECV := @GetPattern**

**REPEAT**

**PINWRITE(56 ADDPINS 7, !RECV())**

**WAITMS(125)**

**PRI GetPattern() : Pattern**

**RETURN DECOD(i++ & 7)**

In the above example, the following values are output in repeating sequence: $01, $02, $04, $08, $10, $20, $40, $80 (but inverted for LEDs)

Though a called method inherits the current RECV pointer, it may change it for its own purposes. Upon return from that method, the RECV pointer will be back to what it was before the method was called. So, the RECV pointer value is propagated in method calls, but not in method returns.

## 8.3) CASE / CASE\_FAST

The CASE construct sequentially compares a target value to a list of possible matches. When a match is found, the related code executes.

Match values/ranges must be indented past the CASE keyword. Multiple match values/ranges can be expressed with comma separators. Any additional lines of code related to the match value/range must be indented past the match value/range:

**CASE target** - CASE with target value

**<match> : <code>** - match value and code

**<indented code>**

**<match..match> : <code>** - match range and code

**<indented code>**

**<match>,<match..match> : <code>** - match value, range, and code

**<indented code>**

**OTHER : <code>** - optional OTHER case, in case no match found

**<indented code>**

CASE\_FAST is like CASE, but rather than sequentially comparing the target to a list of possible matches, it uses an indexed jump table of up to 256 entries to immediately branch to the appropriate code, saving time at a possible cost of larger compiled code. If there are only contiguous match values and no match ranges, the resulting code will actually be smaller than a normal CASE construct with more than several match values.

For CASE\_FAST to compile, the match values/ranges must be unique constants which are all within 255 of each other.

**CASE flag**

**0: CASE\_FAST chr**

**0:     BYTEFILL(@screen, " ", screen\_size)**

**col := row := 0**

**1:     col := row := 0**

**2..7:  flag := chr**

**RETURN**

**8:     IF col**

**col--**

**9:     REPEAT**

**out(" ")**

**WHILE col & 7**

**10:    RETURN**

**11:    color := $00**

**12:    color := $80**

**13:    newline()**

**OTHER: out(chr)**

**2:    col := chr // cols**

**3:    row := chr // rows**

**4..7: background0\_[flag-$04] := chr << 8**

**flag := 0**

# Appendix “A” Programming Style Convention

## A.1) Default Program Template

The default program template is stored in the “Propeller Tool” under Edit→ preferences. One file

Template preference can be used for each of the two propelle types. In particular the propeller ii template used for these examples is store in:

c:\users\wrd\documents\Propeller Tool\templates\WRD\_P2\_Template\_Debug.spin2”

{{WRD\_P2\_Template\_Debug}}

''Debug must be enabled in propeller tool

{===================================================================}

CON {Processor Timing}

\_clkfreq = 200\_000\_000 'processor clock speed

CON

{Processor I/O Hardware}

Rx1 = 63 { I } ' programming / debug

Tx1 = 62 { O }

FsCs = 61 { O } ' flash storage

FsClk = 60 { O }

FsMosi = 59 { O }

FsMiso = 58 { I }

SdClk = 61 { O } ' usd card storage

SdCs = 60 { O }

SdMosi = 59 { O }

SdMiso = 58 { I }

Sda1 = 57 { IO } ' i2c (optional)

Scl1 = 56 { IO }

#0, Cog0,Cog1,Cog2,Cog3,Cog4,Cog5,Cog6,Cog7 'CogNum

#0, P0, P1, P2, P3, P4, P5, P6, P7, P8 ,P9, P10,P11,P12,P13,P14,P15 'PinNum

#16, P16,P17,P18,P19,P20,P21,P22,P23,P24,P25,P26,P27,P28,P29,P30,P31 'PinNum

#32, P32,P33,P34,P35,P36,P37,P38,P39,P40,P41,P42,P43,P44,P45,P46,P47 'PinNum

#48, P48,P49,P50,P51,P52,P53,P54,P55 'PinNum

VAR {Processor Variables}

'CogNumStatus ≔ COGCHK(CogNum)

' Running Check if cog CogNum is running, returns -1 if running or 0 if not

byte Cog0Status,Cog1Status,Cog2Status,Cog3Status,Cog4Status,Cog5Status,Cog6Status,Cog7Status

{Program Test Variables}

long TestVar01,TestVar02,TestVar03

OBJ {Processor Objects}

'mon "VideoDriver"

'term "KeyboardDriver"

'mouse "MouseDriver"

'mem "MemoryStorageDriver"

DAT {Processor Data}

strSalute byte "Hello World",0 'debug notice application running

PUB bootEntry():result01,result02 | x,y 'method run after boot spin interperter loads

debug(zstr(@strSalute)) 'debug used sen string

mainApp() 'turn control over mainApp() method

{==================================================================================}

CON {Application Constants} 'user defined constants

VAR {Application Variables} 'user defined variables

OBJ {Application Objects} 'user defined objects

DAT {Application Data} 'user defined data

PUB mainApp() 'user Spin Program

repeat 'keep cog 0 running

DAT {Application PASM} 'user PASM Program

# Appendix “B” DEBUG INTERRUPT

In addition to the three visible interrupts, there is a fourth "hidden" interrupt that has priority over all the others. It is the debug interrupt, and it is inaccessible to normal cog programs.

Debug interrupts are enabled on a per-cog basis via HUBSET. Each debug-enabled cog will generate a debug interrupt on (re)start from each COGINIT exercised upon it. Within that initial debug ISR and within each subsequent debug ISR, multiple trigger conditions may be set for the next debug interrupt. If no trigger conditions are set before the debug ISR ends, no more debug interrupts will occur until the cog is restarted from another COGINIT.

The last 16KB of hub RAM, which is also mapped to $FC000..$FFFFF, gets partially used as a buffer area for saving and restoring cog registers during debug ISR's. The initial debug ISR routines are also stored in this upper RAM. Once initialized with debug ISR code, this upper hub RAM can be write-protected, in which case it is mapped only to $FC000..$FFFFF and it is only writable from within debug ISR's.

Each cog has an execute-only ROM in cog registers $1F8..$1FF which contains special debug-ISR-entry and -exit routines. These tiny routines perform seamless register-load and register-restore operations for your debugger program, which must be realized entirely within debug ISR's.

|  |
| --- |
| **Execute-only ROM in cog registers $1F8..$1FF**  **(%cccc = !CogNumber)** |
| **Debug ISR Entry - IJMP0 is initialized to $1F8 on cog start** |
| **$1F8 -  SETQ    #$0F    'save registers $000..$00F**  **$1F9 -  WRLONG  0,\*     '\* = %1111\_1111\_1ccc\_c000\_0000**  **$1FA -  SETQ    #$0F    'load program into $000..$00F**  **$1FB -  RDLONG  0,\*     '\* = %1111\_1111\_1ccc\_c100\_0000**  **$1FC -  JMP     #0      'jump to loaded program** |
| **Debug ISR Exit - Jump here to exit your debug ISR** |
| **$1FD -  SETQ    #$0F    'restore registers $000..$00F**  **$1FE -  RDLONG  0,\*     '\* = %1111\_1111\_1ccc\_c000\_0000**  **$1FF -  RETI0           'CALLD IRET0,IRET0 WCZ** |

During a debug ISR, INA and INB, normally read-only input-pin registers, become readable/writable RAM registers named IJMP0 and IRET0, and are used by the debug interrupt as jump and return addresses. On COGINIT, IJMP0 is initialized to $1F8 which is the debug-ISR-entry routine's address.

When a debug interrupt occurs with IJMP0 pointing to $1F8, the following sequence happens:

Cog registers $000 to $00F are saved to hub RAM starting at ($FF800 + !CogNumber << 7), or %1111\_1111\_1ccc\_c000\_0000, where %cccc = !CogNumber.

Cog registers $000 to $00F are loaded from hub RAM starting at ($FF840 + !CogNumber << 7), or %1111\_1111\_1ccc\_c100\_0000, where %cccc = !CogNumber.

A "JMP #$000" executes to run the 16-instruction debugger program that was just loaded into registers $000 to $00F.

Your 16-instruction debugger program will likely want to determine if this debug interrupt was due to a COGINIT, in which case the debugger will probably want to note that a new program is now running in this cog. Depending on what the debugger must do next, it is likely that it will need to save more registers to the upper hub RAM and then load in more code from the upper hub RAM to facilitate more complex operations than the initial 16-instruction ISR can achieve. The ISR may then need to perform some communication between itself and a host system which may be serving as the debugger's user interface. It may be necessary to employ a LOCK to time-share P2-to-host communication channels among cogs, likely on P63 (serial Rx) and P62 (serial Tx). This scenario is somewhat hypothetical, but illustrates the design intent behind the debug interrupt mechanism.

When your debug ISR is complete, you can do a 'JMP #$1FD' to execute the debug-ISR-exit routine which does the following:

Original cog registers $000 to $00F are restored from hub RAM starting at ($FF800 + !CogNumber << 7), or %1111\_1111\_1ccc\_c000\_0000, where %cccc = !CogNumber.

A "RETI0" executes to return to the interrupted cog program.

Here is a table of the hub RAM locations used by each cog for register save/restore and ISR images during the debug interrupt when the register ROM routines are used for ISR entry and exit:

|  |  |  |
| --- | --- | --- |
| **Cog** | **Save/Restore in Hub RAM**  **for Registers $000..$00F** | **ISR image in Hub RAM**  **for Registers $000..$00F** |
| **7** | **$FFC00..$FFC3F** | **$FFC40..$FFC7F** |
| **6** | **$FFC80..$FFCBF** | **$FFCC0..$FFCFF** |
| **5** | **$FFD00..$FFD3F** | **$FFD40..$FFD7F** |
| **4** | **$FFD80..$FFDBF** | **$FFDC0..$FFDFF** |
| **3** | **$FFE00..$FFE3F** | **$FFE40..$FFE7F** |
| **2** | **$FFE80..$FFEBF** | **$FFEC0..$FFEFF** |
| **1** | **$FFF00..$FFF3F** | **$FFF40..$FFF7F** |
| **0** | **$FFF80..$FFFBF** | **$FFFC0..$FFFFF** |

Though the first debug interrupt upon cog (re)start will always use the debug-ISR-entry routine at $1F8, you may redirect IJMP0 during any debug ISR to point elsewhere for use by subsequent debug interrupts. This would mean that you would lose the initial register-saving function provided by the small ROM starting at $1F8, so you would have to use some cog registers for debugger-state storage that don't interfere with the cog program that is being debugged. If no register saving/restoring or host communications are required, your debug ISR may execute very quickly.

What terminates a debug interrupt is not only RETI0 (CALLD INB,INB WCZ), but any D-register variant (CALLD anyreg,INB WCZ). For example RESI0 (CALLD INA,INB WCZ) may be used to resume next time from where this debug ISR left off, but this would imply that you are not using the debug-ISR-entry and -exit routines in the cog-register ROM and have, instead, permanently located debugger code into some cog registers, so that your debugger program is already present at the start of the debug interrupt.

This debug interrupt scheme was designed to operate stealthily, without any cooperation from the cog program being debugged. All control has been placed within the debug ISR. This isolation from normal programming is intended to prevent, or at least discourage, programmers from making any aspect of the debug interrupt system part of their application, thereby rendering the debug interrupt compromised as a standard debugging mechanism. Also, by executing the ISR strictly in cog register space, this scheme does not interfere with the hub FIFO state, which would be impossible to reconstruct if disturbed by hub execution within the debug ISR.

Below are the instructions which are used in the debugging mechanism:

**BRK D/#**

During normal program execution, the BRK instruction is used to generate a debug interrupt with an 8-bit code which can be read within the debug ISR. The BRK instruction interrupt must be enabled from within a prior debug ISR for this to work. Regardless of the execution condition, the BRK instruction will trigger a debug interrupt, if enabled. The execution condition only gates the writing of the 8-bit code:

**D/# = %BBBBBBBB: 8-bit BRK code**

During a debug ISR, the BRK instruction operates differently and is used to establish the next debug interrupt condition(s). It is also used to select INA/INB, instead of the IJMP0/IRET0 registers exposed during the ISR, so that the pins' inputs states may be read:

**D/# = %aaaaaaaaaaaaaaaaeeee\_LKJIHGFEDCBA**

**%aaaaaaaaaaaaaaaaeeee: 20-bit breakpoint address or 4-bit event code (%eeee)**

**%L: 1 = map INA/INB normally, 0 = map IJMP0/IRET0 at INA/INB (default during ISR) \***

**%K: 1 = enable interrupt on breakpoint address match**

**%J: 1 = enable interrupt on event %eeee**

**%I: 1 = enable interrupt on asynchronous breakpoint (via COGBRK on another cog)**

**%H: 1 = enable interrupt on INT3 ISR entry**

**%G: 1 = enable interrupt on INT2 ISR entry**

**%F: 1 = enable interrupt on INT1 ISR entry**

**%E: 1 = enable interrupt on BRK instruction**

**%D: 1 = enable interrupt on INT3 ISR code (single step)**

**%C: 1 = enable interrupt on INT2 ISR code (single step)**

**%B: 1 = enable interrupt on INT1 ISR code (single step)**

**%A: 1 = enable interrupt on non-ISR code  (single step)**

**\* If set to 1 by the debug ISR, %L must be reset to 0 before exiting the debug ISR, so**

**that the RETI0 instruction is able to see IJMP0 and IRET0.**

On debug ISR entry, bits A to L, are cleared to '0'. If a subsequent debug interrupt is desired, a BRK instruction must be executed before exiting the debug ISR, in order to establish the next breakpoint condition(s).

**COGBRK D/#**

The COGBRK instruction can trigger an asynchronous breakpoint in another cog. For this to work, the cog executing the COGBRK instruction must be in its own debug ISR and the other cog must have its asynchronous breakpoint interrupt enabled:

**D/# = %CCCC: the cog in which to trigger an asynchronous breakpoint**

**GETBRK D WCZ**

During normal program execution, GETBRK with WCZ returns various data about the cog's internal status:

**C = 1 if STALLI mode or 0 if ALLOWI mode (established by STALLI/ALLOWI)**

**Z = 1 if cog started in hubexec or 0 if cog started in cogexec**

**D[31:23] = 0**

**D[22] = 1 if colorspace converter is active**

**D[21] = 1 if streamer is active**

**D[20] = 1 if WRFAST mode or 0 if RDFAST mode**

**D[19:16] = INT3 selector, established by SETINT3**

**D[15:12] = INT2 selector, established by SETINT2**

**D[11:08] = INT1 selector, established by SETINT1**

**D[07:06] = INT3 state: %0x = idle, %10 = interrupt pending, %11 = ISR executing**

**D[05:04] = INT2 state: %0x = idle, %10 = interrupt pending, %11 = ISR executing**

**D[03:02] = INT1 state: %0x = idle, %10 = interrupt pending, %11 = ISR executing**

**D[01] = 1 if STALLI mode or 0 if ALLOWI mode (established by STALLI/ALLOWI)**

**D[00] = 1 if cog started in hubexec or 0 if cog started in cogexec**

During a debug ISR, GETBRK with WCZ returns additional data that is useful to a debugger:

**C = 1 if debug interrupt was from a COGINIT, indicating that the cog was (re)started**

**D[31:24] = 8-bit break code from the last 'BRK #/D' during normal execution**

**D[23] = 1 if debug interrupt was from a COGINIT, indicating that the cog was (re)started**

**GETBRK D WC**

**GETBRK with WC always returns the following:**

**C = LSB of SKIP/SKIPF/EXECF/XBYTE pattern**

**D[31:28] = 4-bit CALL depth since SKIP/SKIPF/EXECF/XBYTE (skipping suspended if not %0000)**

**D[27] = 1 if SKIP mode or 0 if SKIPF/EXECF/XBYTE mode**

**D[26] = 1 if LUT sharing enabled (established by SETLUTS)**

**D[25] = 1 if top of stack = $001FF, indicating XBYTE will execute on next \_RET\_/RET**

**D[24:16] = 9-bit XBYTE mode, established by '\_RET\_ SETQ/SETQ2' when top of stack = $001FF**

**D[15:00] = 16 event-trap flags**

**GETBRK D WZ**

**GETBRK with WZ always returns the following:**

**Z = 1 if no SKIP/SKIPF/EXECF/XBYTE pattern queued (D = 0) or 1 if pattern queued (D <> 0)**

**D = 32-bit SKIP/SKIPF/EXECF/XBYTE pattern, used LSB-first to skip instructions in main code**

# Appendix “C” EVENTS

Cogs monitor and track 16 different background events:

* An interrupt occurred
* CT passed CT1 (CT is the 32-bit free-running global counter)
* CT passed CT2
* CT passed CT3
* Selectable event 1 occurred
* Selectable event 2 occurred
* Selectable event 3 occurred
* Selectable event 4 occurred
* A pattern match or mismatch occurred on either INA or INB
* Hub FIFO block-wrap occurred - a new start address and block count were loaded
* Streamer command buffer is empty - it's ready to accept a new command
* Streamer finished - it ran out of commands, now idle
* Streamer NCO rollover occurred
* Streamer read lookup RAM location $1FF
* Attention was requested by another cog or other cogs
* GETQX/GETQY executed without any CORDIC results available

## C.1) Polled Events

Events are tracked and can be polled, waited for, and used as interrupt sources.

Before explaining the details, consider the event-related instructions.

First are the POLLxxx instructions which simultaneously return their event-occurred flag into C and clear their event-occurred flag (unless it's being set again by the event sensor):

Interrupt source (0=off):

POLLINT Poll the interrupt-occurred event flag -

POLLCT1 Poll the CT-passed-CT1 event flag 1

POLLCT2 Poll the CT-passed-CT2 event flag 2

POLLCT3 Poll the CT-passed-CT3 event flag 3

POLLSE1 Poll the selectable-event-1 event flag 4

POLLSE2 Poll the selectable-event-2 event flag 5

POLLSE3 Poll the selectable-event-3 event flag 6

POLLSE4 Poll the selectable-event-4 event flag 7

POLLPAT Poll the pin-pattern-detected event flag 8

POLLFBW Poll the hub-FIFO-interface-block-wrap event flag 9

POLLXMT Poll the streamer-empty event flag 10

POLLXFI Poll the streamer-finished event flag 11

POLLXRO Poll the streamer-NCO-rollover event flag 12

POLLXRL Poll the streamer-lookup-RAM-$1FF-read event flag 13

POLLATN poll the attention-requested event flag 14

POLLQMT Poll the CORDIC-read-but-no-results event flag 15

## C.2) Wait Instructions

Next are the WAITxxx instructions, which will wait for their event-occurred flag to be set (in case it's not, already) and then clear their event-occurred flag (unless it's being set again by the event sensor), before resuming.

By doing a SETQ right before one of these instructions, you can supply a future CT target value which will be used to end the wait prematurely, in case the event-occurred flag never went high before the CT target was reached. When using SETQ with 'WAITxxx WC', C will be set if the timeout occurred before the event; otherwise, C will be cleared.

WAITINT Wait for an interrupt to occur, stalls the cog to save power

WAITCT1 Wait for the CT-passed-CT1 event flag

WAITCT2 Wait for the CT-passed-CT2 event flag

WAITCT3 Wait for the CT-passed-CT3 event flag

WAITSE1 Wait for the selectable-event-1 event flag

WAITSE2 Wait for the selectable-event-2 event flag

WAITSE3 Wait for the selectable-event-3 event flag

WAITSE4 Wait for the selectable-event-4 event flag

WAITPAT Wait for the pin-pattern-detected event flag

WAITFBW Wait for the hub-FIFO-interface-block-wrap event flag

WAITXMT Wait for the streamer-empty event flag

WAITXFI Wait for the streamer-finished event flag

WAITXRO Wait for the streamer-NCO-rollover event flag

WAITXRL Wait for the streamer-lookup-RAM-$1FF-read event flag

WAITATN Wait for the attention-requested event flag

There's no 'WAITQMT' because the event could not happen while waiting.

## C.3) Interupt Jump Instructions

Last are the 'Jxxx/JNxxx S/#' instructions, which each jump to S/# if their event-occurred flag is set (Jxxx) or clear (JNxxx). Whether or not a branch occurs, the event-occurred flag will be cleared, unless it's being set again by the event sensor.

JINT/JNINT Jump to S/# if the interrupt-occurred event flag is set/clear

JCT1/JNCT1 Jump to S/# if the CT-passed-CT1 event flag is set/clear

JCT2/JNCT2 Jump to S/# if the CT-passed-CT2 event flag is set/clear

JCT3/JNCT3 Jump to S/# if the CT-passed-CT3 event flag is set/clear

JSE1/JNSE1 Jump to S/# if the selectable-event-1 event flag is set/clear

JSE2/JNSE2 Jump to S/# if the selectable-event-2 event flag is set/clear

JSE3/JNSE3 Jump to S/# if the selectable-event-3 event flag is set/clear

JSE4/JNSE4 Jump to S/# if the selectable-event-4 event flag is set/clear

JPAT/JNPAT Jump to S/# if the pin-pattern-detected event flag is set/clear

JFBW/JNFBW Jump to S/# if the hub-FIFO-interface-block-wrap event flag is set/clear

JXMT/JNXMT Jump to S/# if the streamer-empty event flag is set/clear

JXFI/JNXFI Jump to S/# if the streamer-finished event flag is set/clear

JXRO/JNXRO Jump to S/# if the streamer-NCO-rollover event flag is set/clear

JXRL/JNXRL Jump to S/# if the streamer-lookup-RAM-$1FF-read event flag is set/clear

JATN/JNATN Jump to S/# if the attention-requested event flag is set/clear

JQMT/JNQMT Jump to S/# if the CORDIC-read-but-no-results event flag is set/clear

Here are detailed descriptions of each event flag. Understand that the 'set' events can also be used as interrupt sources (except in the case of the first flag which is set when an interrupt occurs):

## C.4) Details on Polled/Wait/Interupt Instructions

POLLINT/WAITINT event flag

Cleared on cog start.

Set whenever interrupt 1, 2, or 3 occurs (debug interrupts are ignored).

Also cleared on POLLINT/WAITINT/JINT/JNINT.

POLLCT1/WAITCT1 event flag

Cleared on ADDCT1.

Set whenever CT passes the result of the ADDCT1 (MSB of CT minus CT1 is 0).

Also cleared on POLLCT1/WAITCT1/JCT1/JNCT1.

POLLCT2/WAITCT2 event flag

Cleared on ADDCT2.

Set whenever CT passes the result of the ADDCT2 (MSB of CT minus CT2 is 0).

Also cleared on POLLCT2/WAITCT2/JCT2/JNCT2.

POLLCT3/WAITCT3 event flag

Cleared on ADDCT3.

Set whenever CT passes the result of the ADDCT3 (MSB of CT minus CT3 is 0).

Also cleared on POLLCT3/WAITCT3/JCT3/JNCT3.

POLLPAT/WAITPAT event flag

Cleared on SETPAT

Set whenever (INA & D) != S after 'SETPAT D/#,S/#' with C=0 and Z=0.

Set whenever (INA & D) == S after 'SETPAT D/#,S/#' with C=0 and Z=1.

Set whenever (INB & D) != S after 'SETPAT D/#,S/#' with C=1 and Z=0.

Set whenever (INB & D) == S after 'SETPAT D/#,S/#' with C=1 and Z=1.

Also cleared on POLLPAT/WAITPAT/JPAT/JNPAT.

POLLFBW/WAITFBW event flag

Cleared on RDFAST/WRFAST/FBLOCK.

Set whenever the hub RAM FIFO interface exhausts its block count and reloads its 'block count' and 'start address'.

Also cleared on POLLFBW/WAITFBW/JFBW/JNFBW.

POLLXMT/WAITXMT event flag

Cleared on XINIT/XZERO/XCONT.

Set whenever the streamer is ready for a new command.

Also cleared on POLLXMT/WAITXMT/JXMT/JNXMT.

POLLXFI/WAITXFI event flag

Cleared on XINIT/XZERO/XCONT.

Set whenever the streamer runs out of commands.

Also cleared on POLLXFI/WAITXFI/JXFI/JNXFI.

POLLXRO/WAITXRO event flag

Cleared on XINIT/XZERO/XCONT.

Set whenever the the streamer NCO rolls over.

Also cleared on POLLXRO/WAITXRO/JXRO/JNXRO.

POLLXRL/WAIXTRL event flag

Cleared on cog start.

Set whenever location $1FF of the lookup RAM is read by the streamer.

Also cleared on POLLXRL/WAITXRL/JXRL/JNXRL.

POLLATN/WAITATN event flag

Cleared on cog start.

Set whenever any cogs request attention.

Also cleared on POLLATN/WAITATN/JATN/JNATN.

POLLQMT event flag

Cleared on cog start.

Set whenever GETQX/GETQY executes without any CORDIC results available or in progress.

Also cleared on POLLQMT/WAITQMT/JQMT/JNQMT.

**Example: ADDCT1/WAITCT1**

'ADDCT1 D,S/#' must be used to establish a CT target. This is done by first using 'GETCT D' to get the current CT value into a register, and then using ADDCT1 to add into that register, thereby making a future CT target, which, when passed, will trigger the CT-passed-CT1 event and set the related event flag.

**GETCT   x               'get initial CT**

**ADDCT1  x,#500          'make initial CT1 target**

**.loop  WAITCT1                 'wait for CT to pass CT1 target**

**ADDCT1  x,#500          'update CT1 target**

**DRVNOT  #0              'toggle P0**

**JMP     #.loop          'loop to the WAITCT1**

It doesn't matter what register is used to keep track of the CT1 target. Whenever ADDCT1 executes, S/# is added into D, and the result gets copied into a dedicated CT1 target register that is compared to CT on every clock. When the CT1 target passes CT, the event flag is set. ADDCT1 clears the CT-passed-CT1 event flag to help with initialization and cycling.

**Note:** the .loop operator is cleared after compiling instruction and loop can be re-used for writing jump code. Remember to include #(immediate) directive

# Appendix “D” P2 Edge

## D.1) Edge Specifications

-20MHZ crystal normal clock 180\_000\_000 HZ overclock to 320\_000\_000 HZ (16x)

-VDD Processor Power regulator 1.8V 2.0Amp 16 pin common

-VIO\Vxxxx\Vxxyy\V00-V56 3.3V I/O Power LDO 300mA

-Flash Memory 16MB (128Mb) spi format

# Appendix “E” Byte Word Long Declaration

## A.1) Byte Declaration 8 bits

**Syntax 1** Var (variable declaration)

VAR

byte Temp ‘Temp is a byte variable

byte Str[25] ‘Str is a byte array Str[0]-Str[24]

**Syntax 2** Dat (data declaration)

DAT

MyData byte 41,” A”, $2A

MyString “Hello”, 0 ‘zero terminated string

## A.2) Word Declaration 16 bits

**Syntax 1** Var (variable declaration)

VAR

word var01 ‘var01 Temp is a word variable

word List[25] ‘List is a word array Str[0]-Str[24]

**Syntax 2** Dat (data declaration)

DAT

MyList word $FFFF, 41,” A”, $2A

## A.3) Long Declaration 32 bits

**Syntax 1** Var (variable declaration)

VAR

Long var01 ‘var01 Temp is a Long variable

word List[25] ‘List is a Long array Str[0]-Str[25]

**Syntax 2** Dat (data declaration)

DAT

MyList Long $FFFF, 41,” A”, $2A

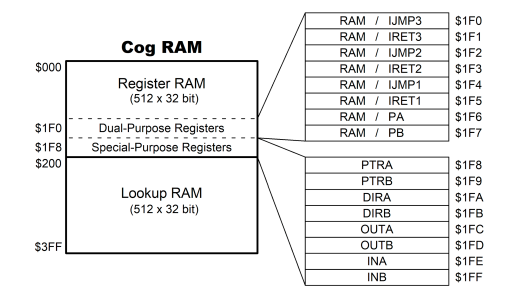
**Syntax 3** PUB/PRI

Pub method| Index, var01 ‘declares local method variable of type Long

# Appendix “F” Hardware and Constants

## F.1) Hardware Register Constants

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Variables**  **(all LONG)** | **Variable**  **Name** | **Address**  **or Offset** | **Description** | **Useful in**  **Spin2** | **Useful in**  **Spin2-PASM** | **Useful in**  **PASM-Only** |
| Hub Locations | **CLKMODE**  **CLKFREQ** | **$00040**  **$00044** | Clock mode value  Clock frequency value | Yes  Yes | Yes  Yes | No  No |
| Hub VAR | **VARBASE** | +0 | Object base pointer, @VARBASE is VAR base, used by method-pointer calls | Maybe | No | No |
| Cog Registers | **PR0**  **PR1**  **PR2**  **PR3**  **PR4**  **PR5**  **PR6**  **PR7**  **IJMP3**  **IRET3**  **IJMP2**  **IRET2**  **IJMP1**  **IRET1**  **PA**  **PB**  **PTRA**  **PTRB**  **DIRA**  **DIRB**  **OUTA**  **OUTB**  **INA**  **INB** | **$1D8**  **$1D9**  **$1DA**  **$1DB**  **$1DC**  **$1DD**  **$1DE**  **$1DF**  **$1F0**  **$1F1**  **$1F2**  **$1F3**  **$1F4**  **$1F5**  **$1F6**  **$1F7**  **$1F8**  **$1F9**  **$1FA**  **$1FB**  **$1FC**  **$1FD**  **$1FE**  **$1FF** | Spin2 <-> PASM communication  Interrupt JMP's and RET's  Pointer registers  Data pointer passed from COGINIT  Code pointer passed from COGINIT  Output enables for P31..P0  Output enables for P63..P32  Output states for P31..P0  Output states for P63..P32  Input states from P31..P0  Input states from P63..P32 | Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  No  No  No  No  No  No  No  No  No  No  Yes  Yes  Yes  Yes  Yes  Yes | Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes | No  No  No  No  No  No  No  No  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes |



## F.2) HUB Memory

Hub Memory is located in (and managed by) the Hub and is accessible to each cog, in a time-shared, round-robin fashion. It consists of Hub RAM and Hub ROM.  
  
Hub RAM is **512 KB**, accessible as bytes, words, and longs. It holds your program, data, global variables, and stack space, which collectively make up your Propeller Application. Hub RAM is also used to share information between cogs or process larger blocks of data than will fit into Cog RAM.  
  
The Hub ROM is 16 KB and holds read-only system resources such as the Boot Loader. It is loaded into the last **16 KB** of Hub RAM upon boot-up.

## F.3) HUB Memory Spin2 Stack.

The Spin Interpreter implements a call stack to facilitate Spin method calling, parameter passing, expression evaluation, and returning method results.   
  
The Propeller Application (if Spin2-based) has an automatically allocated stack located in Hub RAM immediately following the application's global variable memory. It expands and collapses as needed; growing towards higher addresses and shrinking towards lower addresses.   
  
Spin methods that are manually launched into other cogs store their stack starting at the StkAddr address given by the COGSPIN command that launched them (usually inside a long array in variable space). Their stacks expand and contract in the same manner as with the Propeller Application stack. In both cases, the capacity of the stack (method nesting-depth, parameter list length, expression complexity, and return result length) is limited only by the amount of free memory available (for the application) or memory provided (by the developer).

## F.4) DAT Blocks

DAT block symbols exist in Hub RAM, but if they are part of PASM2 code that is launched, they are also in Cog RAM where they are manipulated independently.

The DAT block itself is stored in the application image in Hub RAM. Spin2-based references to DAT symbols access the corresponding location and data in Hub RAM.

When a cog is launched with assembly code, any DAT symbols within 504 longs of the launch point are copied into Register RAM. Unlike with Spin2 code, PASM2 code that references those symbols accesses the corresponding Register RAM\* locations (its local copy) instead of Hub RAM. In addition, those symbolic references are addressed as longs of Register RAM memory, regardless of how the symbol was actually declared.

\* Or Lookup RAM, if the code launched into Register RAM manually loads PASM2+symbol code into, and executes code from, Lookup RAM.

The DAT block's purpose is to hold fixed data and Propeller 2 Assembly code for the application. Symbols may be included to reference this data and code.

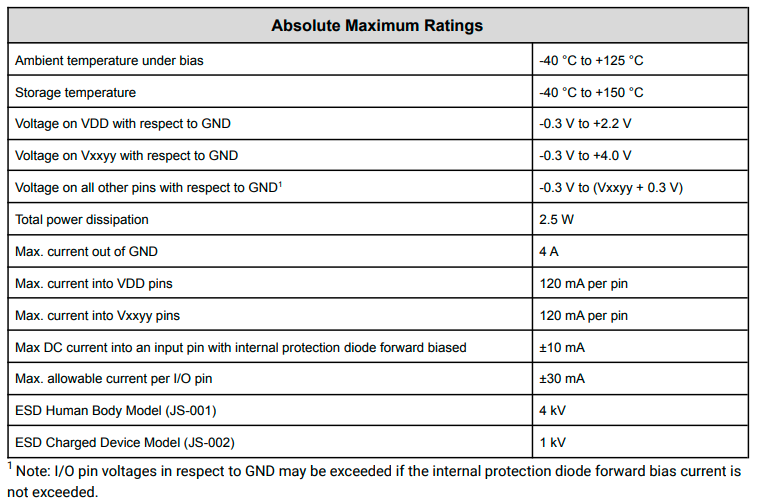
DAT blocks are stored in the application image in Hub RAM. Just like with code in PUB and PRI blocks, there is only one instance of each DAT block in the running application, regardless of how many instances of the containing object there are. This means that Spin-based references to DAT symbols each access the same corresponding location and data in Hub RAM, regardless of which instance of that object is making the reference. This is handy to share memory between multiple instances of a Spin2 object.

When a cog is launched with assembly code, any DAT symbols within 504 longs of the launch point are copied into Register RAM. Unlike with Spin2 code, PASM2 code that references those symbols accesses the corresponding Register RAM\* locations (its local copy) instead of Hub RAM. In addition, those symbolic references are addressed as longs of Register RAM memory, regardless of how the symbol was actually declared. In PASM2, no Hub RAM references can be made by simply using the declared symbolic name; instead, the absolute address of that symbol must be passed from the Spin2 object and used along with instructions like RDLONG and WRLONG.

There's nothing preventing the contents of DAT from being modified at runtime. This naturally leads to a special use–; "special values" may be defined in a DAT block that are easily referenced by every Spin2 object instance (and every new launch of PASM2 code) and can be modified at runtime to instantly change what each Spin2 instance (and future new PASM2 launched cogs) sees.  
  
\* Or Lookup RAM locations, if symbolic data were initially loaded in by the code running in Register RAM.

## F.5) Propeller Electrical Specifications

Absolute Maximum Electrical Ratings Stresses in excess of the absolute maximum ratings can cause permanent damage to the device. These are absolute stress ratings only. Functional operation of the device is not implied at these or any other conditions in excess of those given. Exposure to absolute maximum ratings for extended periods can adversely affect device reliability.



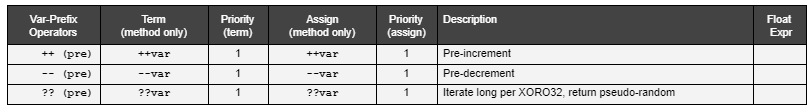
## F.6) Built In Numeric Constants

|  |  |  |
| --- | --- | --- |
| **Symbol Value** | **Symbol Name** | **Details** |
| $0000\_0000 | FALSE | Same as 0 |
| $FFFF\_FFFF | TRUE | Same as -1 |
| $8000\_0000 | NEGX | Negative-extreme integer, -2\_147\_483\_648 ($8000\_0000) |
| $7FFF\_FFFF | POSX | Positive-extreme integer, +2\_147\_483\_647 ($7FFF\_FFFF) |
| $4049\_0FDB | PI | Single-precision floating-point value of Pi, 3.14159265 |

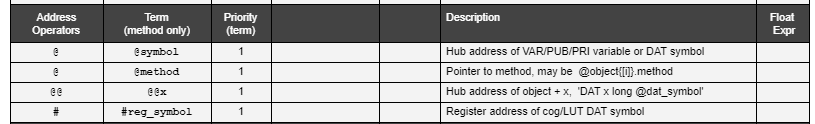
# Appendix “G” Table of Operators

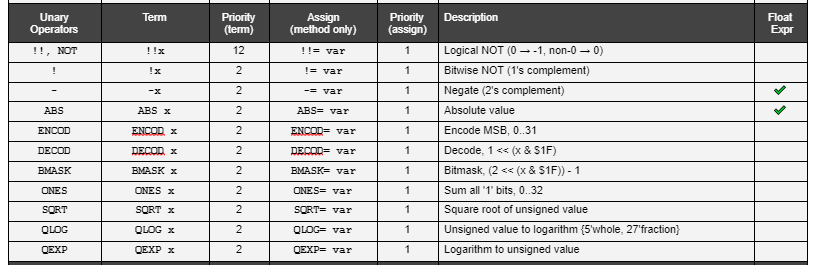
**Operators**

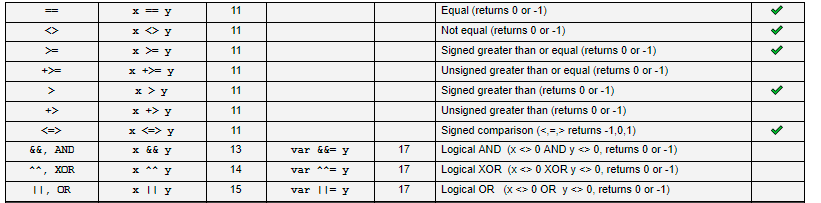
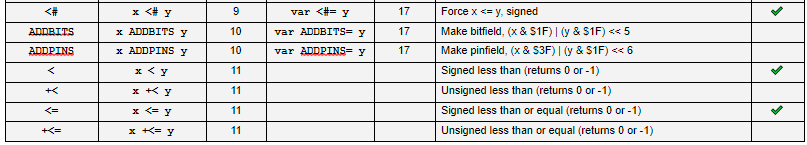
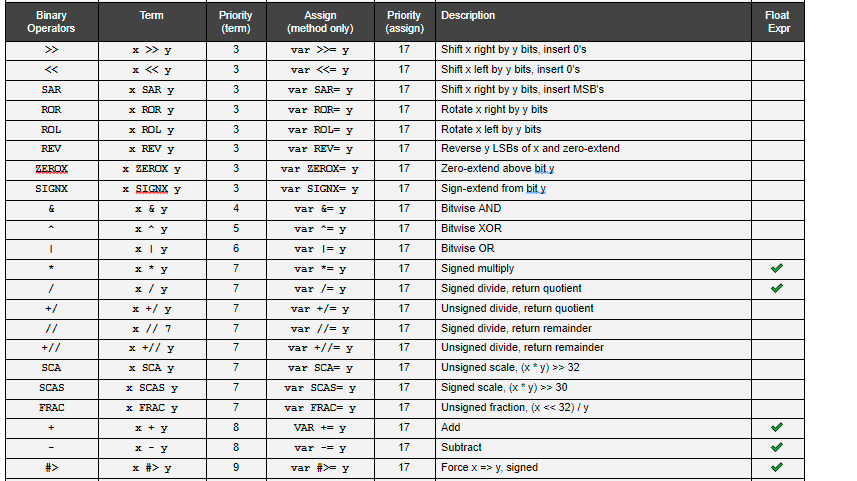
Below is a table of all the operators available for use in Spin2 methods. Compile-time expressions can use the unary, binary, ternary and float operators.

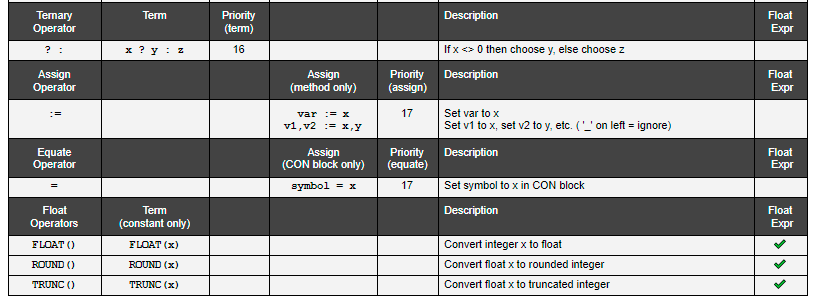












# Appendix “H” Table of Built In Methods

