**STANDARD I/O PINS**

Notes to writers, editors...

I have moved the I/O PIN TIMING section to here. I think we get a better organization when people can find the pin information in one section/chapter. I used the ExpressPCB schematic-capture program for dra wings in this section.

The basic unit measure of time is the period of the system-clock frequency. The length of this period depends on how you set up the clock in a Propeller-2 microcontroller.

Many modes include wording such as , "Software can wait in a loop that tests the pin's IN flag. When detected, an RDPIN or RQPIN instruction reads a value from the Z register." Should the docs say that, or just mention RDPIN? The RQPIN does not reset the Carry flag, correct? If so, then the "RDPIN or RQPIN" statement could confuse people when they write software. In some cases, their IN flag won't get cleared.

You may download all software examples in this section from the URL here...

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 give you direct access to I/O pins:

# Propeller 2 I/O-pin registers and I/O-pin Instructions

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

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

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

DRVH #8 'Drive P8 high

DRVH #10 ADDPINS 7 'Drive P10..P17 high

## Pin-Direction Instructions

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

## Pin-Output Instructions

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

## Pin-Float Instructions

These instructions change the associated DIR bit(s) to logic-0 (input).

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.

## 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.

## 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 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.

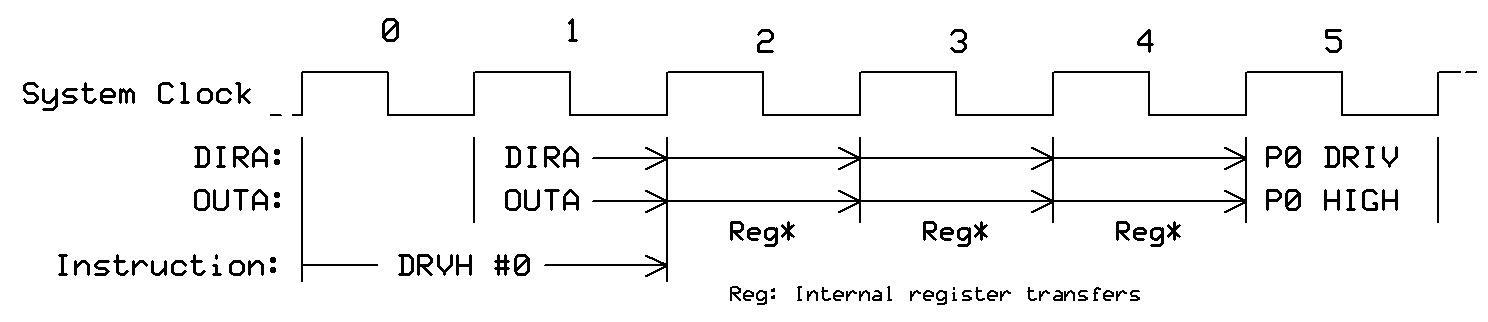
TESTPN {#}D ANDC/ANDZ Get a pin's NOT-state and AND it into the C or Z flag.

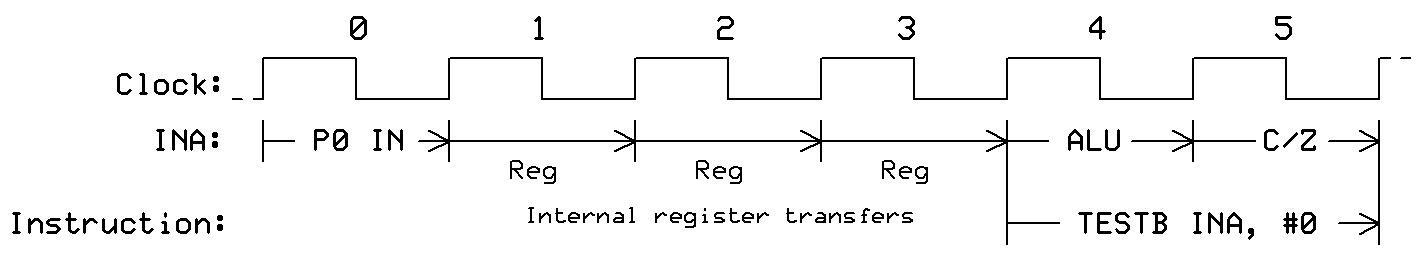
TESTPN {#}D ORC/ORZ Get a pin's NOT-state and OR it into the C or Z flag.

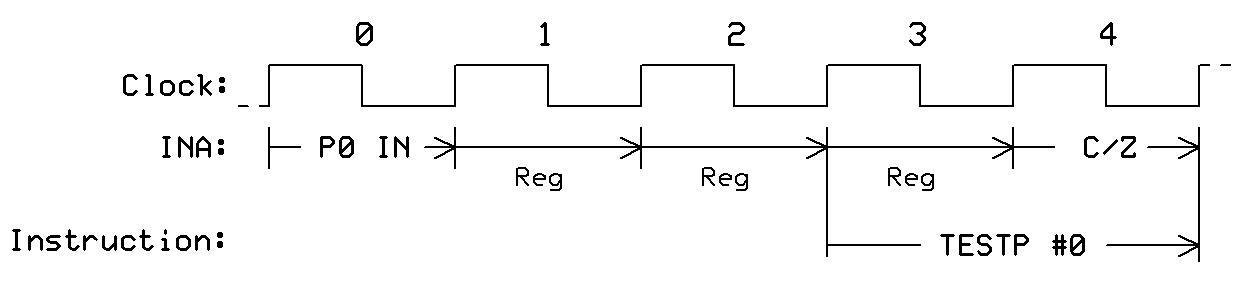
TESTPN {#}D XORC/XORZ Get a pin's NOT-state and XOR it into the C or Z flag.

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

# 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:

# Smart Pins

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. De pending 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.

## ///Block diagram here...

## Smart Pin Registers

Every Smart Pin circuit includes four 32-bit registers, Mode, X, Y, and Z that control operations:

**Register** **Operation**

Mode Controls Smart Pin modes, as well as low-level I/O (write-only)

X Sets parameters for a specific mode (write-only)

Y Sets parameters for a specific mode (write-only)

Z Information to a cog from a configured pin (read-only)

Programs read or write 32-bit values to and from these four registers following completion of the 2-clock instructions shown here:

WRPIN D/#,S/# Write bits D[31:0] to Mode register for Smart Pin S[5:0],

acknowledge Smart Pin.

WXPIN D/#,S/# Write bits D[31:0] to register X for Smart Pin S[5:0],

acknowledge Smart Pin.

WYPIN D/#,S/# Write bits D[31:0] to register Y for Smart Pin S[5:0],

acknowledge Smart Pin.

RDPIN D,S/# {WC} Read Smart Pin S[5:0] register Z[31:0], save result in D[31:0],

acknowledge Smart Pin. (Carry = mode result.)

RQPIN D,S/# {WC} Read Smart Pin S[5:0] register Z[31:0], save result in D[31:0],

flag into C, do not acknowledge.

AKPIN S/# Acknowledge Smart Pin S[5..0].

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. (For more information about flags, see Data Sheet "section name," page XXX.)

Each cog and Smart Pin connects to a common 34-bit bus and bits on this bus are logically OR'ed in the same way DIR and OUT bits are OR'd before going to I/O pins. If you intend to have more than one cog access the *same* Smart Pin with any of these instructions--WRPIN, WXPIN, WYPIN, RDPIN, and AKPIN beware! You must ensure the cogs do so at different times. If two or more cogs try to use these registers simultaneously, they "crash" each other's bus data.

There is one exception, though: Any number of cogs may use an RQPIN (read quiet) instruction simultaneously to read a Smart Pin's data without bus conflict. This instruction does not use the 34-bit cog-to-Smart Pin bus for acknowledgement signals. The RDPIN instruction does.

**Smart Pin Operation Settings and How to Use Them**

### WRPIN

The WRPIN instruction writes 32-bit data, D/#, to the Mode register for I/O pin identified by the S/# value or symbol. Note: The WRPIN instruction sets two logic modes for each Smart Pin. The following tables describe the data fields in the WRPIN instruction. Most likely you will refer often to this table as you study the Smart-Pin modes. Each Smart-Pin mode requires 32 bits that define how pins and internal circuits will function. To make operations easier to understand, we break the 32-bit value into six sections. The LSB always equals 0.

D/# = %AAAA\_BBBB\_FFF\_PPPPPPPPPPPPP\_TT\_MMMMM\_0

**AAAA Logic-input selector (4 bits)**

0xxx = non-inverted logic input (default)

1xxx = inverted logic input

x000 = read this pin's state (default)

x001 = read state of P37 + 1 = P38, pin number plus 1

x010 = read state of P37 + 2 = P39, pin number plus 2

x011 = read state of P37 + 3 = P40, pin number plus 3

x100 = read this pin's OUT bit from cogs

x101 = read state of P37 - 1 = P36, pin number minus 1

x110 = read state of P37 - 2 = P35, pin number minus 2

x111 = read state of P37 - 3 = P34, pin number minus 3

**BBBB Logic-input selector (4 bits)**

0xxx = non-inverted logic input (default)

1xxx = inverted logic input

x000 = read this pin's state (default)

x001 = read state of P37 + 1 = P38, pin number plus 1

x010 = read state of P37 + 2 = P39, pin number plus 2

x011 = read state of P37 + 3 = P39, pin number plus 3

x100 = read this pin's OUT bit from cogs

x101 = read state of P37 - 1 = P36, pin number minus 1

x110 = read state of P37 - 2 = P35, pin number minus 2

x111 = read state of P37 - 3 = P34, pin number minus 3

You might ask, "Why would a Smart Pin need to get information from a nearby pin?" This capability comes in handy when you want to monitor an input stream to calculate a serial-input bit rate, or to test an input with a different cog to obtain debug or diagnostic information. Some mode examples that follow use A and B signals for data and a clock, two encoder inputs, an input and a logic control, and so on.

**FFF A and B logic-and-filtering conditions (3 bits)**

000 = A, B (default)

001 = A AND B, B; logical AND

010 = A OR B, B; logical OR

011 = A XOR B, B; logical XOR

100 = A, B, both filtered using global filt0 settings

101 = A, B, both filtered using global filt1 settings

110 = A, B, both filtered using global filt2 settings

111 = A, B, both filtered using global filt3 settings

The A result will drive the IN signal when you use a pin in a non-Smart Pin mode. Later sections describe digital-filter operations and uses.

**P[12:0] Low-level pin control (13 bits)**

The chart in the original document is difficult to read and it uses M[12..0] rather than P[12..0], which corresponds to the description here. I recommend we change the chart to show P[12..0] so the information does not cause confusion between the P and M fields.

Also, place the "Legends" information below and separate from the main table. I don't understand what the right-most table indicates or how it relates to the table on the left. Overall, the chart is confusing without more explanation and an example or two.

Insert table here...

**%TT Pin DIR/OUT control**

These two bits have a default value of default of %00. Please refer to the table "PAD\_IO Modes" for P[12:0] bit information.

This section needs work. Many questions.

'DAC\_MODE' is enabled when P[12:10] = %101

'BIT\_DAC' outputs {2{P[7:4]}} for 'high' or {2{P[3:0]}} for 'low' in DAC\_MODE What does the number 2 represent? Do not understand this notation.

**1.**  Smart Pin modes disabled (%MMMMM = %00000)

a. DIR instructions enable output.

b. For pins NOT in DAC mode:

0x = OUT instruction drives output

1x = OTHER drives output (See Definitions below)

c. For pin in DAC\_MODE:

00 = OUT enables ADC, bits P[7:0] set DAC voltage

01 = OUT enables ADC, bits P[3:0] select cog DAC channel

10 = OUT drives BIT\_DAC (What does 'drive' mean? Enable?)

11 = OTHER drives BIT\_DAC What is a bit DAC?

**2.** All Smart-Pin modes (%MMMMM > %00000)

x0 = output disabled, regardless of DIR

x1 = output enabled, regardless of DIR

**3.** For Smart Pin DAC modes (%MMMMM = %00001..%00011)

0x = OUT enables ADC in DAC\_MODE, bits P[7:0] overridden

1x = OTHER enables ADC in DAC\_MODE, bits P[7:0] overridden

**4.** For Smart-Pin non-DAC modes (%MMMMM = %00100..%11111)

0x = SMART/OUT drives output or BIT\_DAC if DAC\_MODE

1x = SMART/OTHER drives output or BIT\_DAC if DAC\_MODE

In Section 4. If  *'SMART' = Smart Pin output which overrides OUT/OTHER* means exactly that, why use the notation, SMART/OTHER?

Contradiction? If a Smart Pin is in NON\_DAC mode, how can it be in DAC mode?

Definitions

Other:

For odd-numbered pins (P1, P3...), 'OTHER' indicates the the even-numbered pin's NOT inverted? output state (differential? source). What is 'diff source and what does it mean? OK, so I use P7, what is the "even-numbered pin? P6 or P8?)

For even-numbered pins (P2, P4...), 'OTHER' = unique pseudo-random bit (noise source). Output or input? Same question re pin numbers.

SMART:

For all pins, 'SMART' = Smart Pin output which overrides OUT/OTHER.

**MMMMM Smart Pin Mode (5 bits)**

00000 = Smart Pin operations off (default)

00001 = long repository (P[12:10] != %101)

00010 = long repository (P[12:10] != %101)

00011 = long repository (P[12:10] != %101)

00001 = DAC noise (P[12:10] = %101)

00010 = DAC 16-bit dither, noise (P[12:10] = %101)

00011 = DAC 16-bit dither, PWM (P[12:10] = %101)

00100\* = pulse/cycle output

00101\* = transition output

00110\* = numerically controlled oscillator (NCO) frequency

00111\* = numerically controlled oscillator (NCO) duty cycle

01000\* = PWM triangle waveform

01001\* = PWM sawtooth waveform

01010\* = PWM switch-mode power supply, V and I feedback

01011 = periodic/continuous: A-B quadrature encoder

01100 = periodic/continuous: increment on A-rise & B-high

01101 = periodic/continuous: increment on A-rise & B-high dec on A-rise & B-low

01110 = periodic/continuous: increment on A-rise {/ dec on B-rise}

01111 = periodic/continuous: increment on A-high {/ dec on B-high}

10000 = time A-states

10001 = time A-highs

10010 = time X A-highs/rises/edges -or- timeout on X A-high/rise/edge

10011 = for X periods, count time

10100 = for X periods, count states

10101 = for periods in X+ clocks, count time

10110 = for periods in X+ clocks, count states

10111 = for periods in X+ clocks, count periods

11000 = ADC sample/filter/capture, internally clocked

11001 = ADC sample/filter/capture, externally clocked

11010 = ADC scope with trigger

11011\* = USB host/device (even/odd pin pair = DM/DP)

11100\* = sync serial transmit (A-data, B-clock)

11101 = sync serial receive (A-data, B-clock)

11110\* = async serial transmit (at baud-rate)

11111 = async serial receive (at baud-rate)

\* These modes override the OUT signal.

Timing and Coordination

When a Smart Pin completes a task it raises its IN flag to a logic-1, which software may detect indirectly. The TESTP, or test-pin, instruction lets you transfer the state of the IN flag to either the pin's Carry (C) or Zero (Z) bit, which software can test with an instruction such as:

testp #pin\_number, wc

or wait for it to become a logic-1 in the following loop:

.loop testp #pin\_number, wc 'set carry to IN flag state

if\_nc jmp #.loop 'C = 0, jump

rdpin your\_data, #pin number 'get pin data

Smart Pins use the IN flag to alert a cog, or cogs, to indicate that:

1. The Smart Pin has new data available,

2. A cog may load new information to the Smart Pin, or

3. A Smart Pin process has ended.

When a cog executes a WRPIN, WXPIN, WYPIN, RDPIN or AKPIN instruction, for example, the Smart Pin resets its IN flag to logic 0. Because the RQPIN instruction, "read quiet," does not reset the IN flag, any number of cogs can execute an RQPIN instruction simultaneously without bus conflict.

After a cog executes a WRPIN, WXPIN, WYPIN, RDPIN, or AKPIN instruction the Smart Pin requires two system-clock cycles to reset the IN flag to logic-0 before a program can poll the flag again. Simply insert a no-operation, or NOP, instruction between a Smart Pin instruction and an instruction that reads the IN flag:

wrpin mode, #pin 'acknowledge Smart Pin, clear IN

nop 'delay for 2 (or more) clock cycles

testp pin\_number wc 'Now move IN to Carry

Before you can configure Smart-Pin operations use a DIRL, #pin\_number instruction to put the pin in reset state. Then you may use the WRPIN, WXPIN, and WYPIN instructions to write the mode and related parameters to the pin's registers.

After your program has configured a Smart Pin, change the DIR bit to a logic-1,

DIRH, #pin\_number, to start the Smart Pin mode. Then your program may send the pin other information via the WXPIN and WYPIN instructions, and use the RDPIN and RQPIN to retrieve information from the Smart Pin. In many cases your software must coordinate these register actions with the state of the Smart Pin's IN flag.

At any time a program may use the instruction WRPIN #0, pin\_number to reset a Smart Pin. You do not need to reconfigure the pin by clearing and then setting its DIR bit.

Note that while a Smart Pin is configured, the %TT bits, explained above, will govern the pin's output enable, regardless of the DIR state. Does this mean before or after a Smart Pin configured? I'm not sure whether this means during, or simultaneously, or something else?

--------------------------------------------------------------

**Smart Pin Modes**

**%00001..%00011: Long Depository (when not in DAC mode)**

This mode turns the Smart Pin into a repository, or storage register, for a long value. Instruction WXPIN writes the long value and RDPIN or RQPIN will read the long value. Each WXPIN operation sets the IN flag. A reset (DIR = 0) resets the IN flag to a logic 0. The code snippet below provides an example of repository use:

Code example for Long Depository (non-DAC):

con

\_clkfreq = 160\_000\_000

dat

org 0 ' Save program starting at cog RAM address 0

' Use P12 as a Smart Pin

dirh #12 ' Set direction of P12 to logic-1 (output)

wrpin jat, #12 ' Set repository mode for P12

wxpin Alpha, #12 ' Move Alpha to reg X at P12

nop ' Delay for register clocking

rqpin Beta, #12 ' Read data from reg X at P12

Alpha long $1500\_0000 ' Test data to save in repository

Beta long $2FFF\_FFFF ' Value here will change after rqpin

jat long %0000\_0000\_000\_0000000000000\_00\_00001\_0 'mode bits

**--------------------------------------------------------------**

**%00001: Random noise via DAC**

This mode ignores all mode bits in P[7:0] and sends the pin's 8-bit digital-to-analog converter (DAC) a unique pseudo-random value on every clock. You must set bits bits P[12:10] to %101 on the mode value to configure the pin for DAC output.

You may set 16 bits in the X register (X[15:0]) to a sample period, in clock cycles, if you choose. The Smart Pin's IN flag lets you detect the end of this period. Regardless of the period you set, the DAC continues change its voltage on every clock cycle.

If you don't need a sample period, clear register X bits 15:0 to zero (65,536 clocks). This value maximizes the unused sample period, which reduces switching power.

You may use the RDPIN or RQPIN instruction to retrieve the 16-bit ADC value from the last sample period. Is this the sample period set with X[15:0] or the last random value in the DAC? During reset (DIR=0), IN is in the logic-0 state.

**Example:**

CON

dat

org 0

dirl #20 'Reset Pin P20

wrpin DACconfig, #20 'Config DAC mode

dirh #20 'Start DAC mode

.loop nop 'Loop "forever"

jmp #.loop

'Random dither

DACconfig long %0000\_0000\_000\_10100\_00000000\_01\_00001\_0

The output at pin 20 appears as semi-random noise. (Example courtesy of "Ariba" via Parallax Forum.)

**--------------------------------------------------------------**

**%00010: DAC output with added pseudo-random dither**

The voltage output from a Smart Pin in this mode will include an added 8 bits of dither to the output voltage. These eight bits add a small voltage; less than that contributed by the DAC's least-significant bit (LSB). The dithering occurs randomly to achieve a DAC output with 16-bit resolution *when averaged over time*.

The Smart Pin circuit sends its DAC eight bits of pseudo-randomly-dithered data every system-clock period. The Y register's bits Y[15:0] set the DAC's output voltage for each each sample period. The pin automatically includes the dithering. At the end of a system-clock period the Smart Pin transfers a new 16-bit value to the Y register. That action raises the IN flag, which software could detect, and then update the Y register's value. Your new value takes effect immediately.

In the 32-bit mode value, set bits P[12:10] to %101 to configure the pin for DAC output. This mode overrides bits P[7:0]. You may use bits X[15:0] to set a sample period in units of the system-clock period. The dithered output occurs whether or not you use this register. Why would someone use the register? Just so they could time an update?

Example:

This software produces a sawtooth wave with the added dither.

CON

dat org 0

dirl #20 'Set DAC at pin P20

wrpin DACconfig, #20 'Set DAC configuration

wxpin DACperiod, #20 'Set DAC update time

dirh #20

.myloop wypin DACvolt, #20 'Send voltage value to P20

.waitper nop

testp #20 wc 'Test P20, get Carry flag

if\_nc jmp #.waitper 'Wait for carry flag = 1

add DACvolt, #$100 'Add $100 (256) to DAC voltage

wypin DACvolt, #20 'Load DAC

nop

jmp #.myloop 'Do this loop "forever"

'%00010 mode settings

DACconfig long %0000\_0000\_000\_10100\_00000000\_01\_00010\_0

DACperiod long $100

DACvolt long 0 ' wypin bits Y15:Y0 for voltage

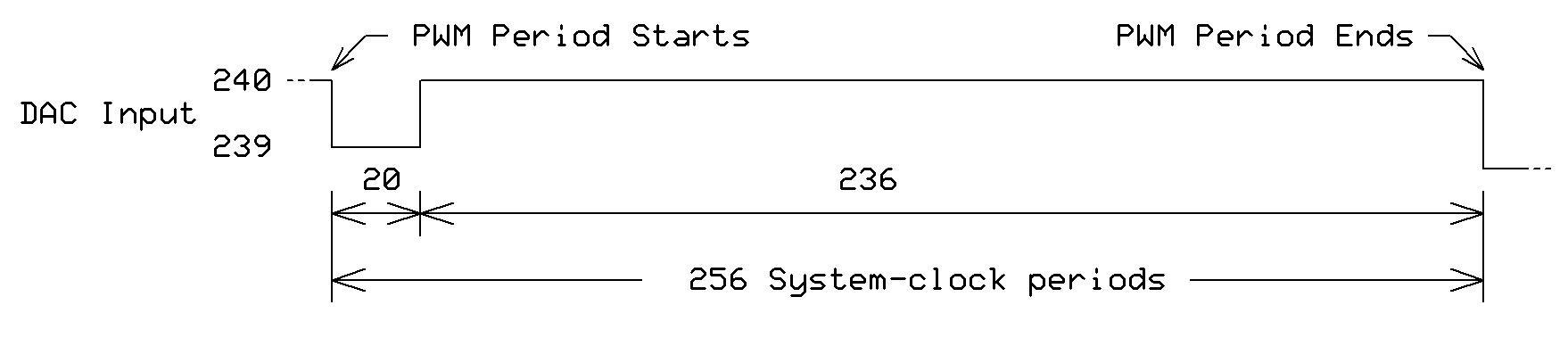
If you need to update the output value at any time and have it take immediate effect, set X[15:0] to one. The IN flag will remain a logic-0. Is this updated value active for only one clock period or does it remain in effect until something else happens?

In this mode, when you set the Smart Pin's OUT bit to logic-1, you enable the ADC and can use the RDPIN or RQPIN instruction to capture the 16-bit ADC value from the *previous* sample period. (You might used this value to calculate current load on the DAC pin.) During reset (DIR=0), IN is low and Y[15:0] is captured. Not sure what this sentence means. Do we need an additional example to show how this action works?

**--------------------------------------------------------------**

**%00011: DAC output with added PWM dither**

The pulse-width-modulation (PWM) dithering technique also extends the *resolution* of an 8-bit digital-to-analog converter (DAC). The 8-bit DAC provides 256 steps of approximately 12.9 mV/step. If you want a DAC output of, say, 3.095 V you can have either 3.083V at DAC input 239, or 3.096V at input 240. There is no step "239.9" for an 8-bit DAC.

But, *proportionally averaging* some of the 3.083V and the 3.096 voltages lets the DAC can get very close to 3.095V, as shown below for a 256-system-clock period (DAC period).

The pulse-width modulator will *add* a logic-1 to the DAC's 8-bit value for a preset portion of the DAC period set in the pin's X register, X[15:0]. The ratio of the logic-0 and logic-1 periods determines the voltage output between the 239 and 240 DAC inputs. You must use a sample period equal to a multiple of 256 (bits X[7:0] = 0), so the PWM has an integral number of 256 steps during which it dithers the DAC output between adjacent 8-bit levels.

To use this mode, set bits P[12:10] to %101 to configure the low-level pin for DAC output. This mode overrides any bits set for P[7:0].

At the start of each DAC period, the Smart Pin receives a 16-bit value from the Y register (Y[15:0]) just as if you have a 16-bit DAC. If you supply a new value for the Y-register bits, it gets transferred to the DAC at the end of the current DAC period and used for the entire following DAC period.

A logic-1 on the IN flag indicates the end of a DAC period, so you may use that flag to coordinate the transfer of another 16-bit DAC value to the Y register. If your value won't change, there's no need to monitor the IN flag.

PWM dithering offers better dynamic range than pseudo-random dithering, because a maximum of only two transitions occur for every 256 clocks. This means, though, that a frequency of the (system-clock frequency) divided by 256 will be present in the output at -48 dB.

When you set the pin's OUT bit to a logic-1, you enable the ADC and may use the RDPIN and RQPIN instructions retrieve the 16-bit ADC accumulation (value?) from the previous sample period. You might use this value to calculate the load impedance at the DAC pin. So the ADC measures the output voltage? What do you do--code please--to get the ADC value? Is it in the Z register?

When code resets the Smart Pin (DIR=0), IN changes to a logic-0 and data in the Y register Y[15:0] gets transferred to the DAC inputs. The example below creates a square wave that switches between $F00F and $01F0 DAC output voltages.

**Example**

CON

dat

org 0

dirl #20 'Setup DAC at Smart-Pin P20

wrpin DACconfig, #20 'Set configuration

wxpin DACperiod, #20 'Set DAC voltage-update time

dirh #20

.myloop wypin DACvolt1, #20 'Send volt value to P20

nop 'Delay for IN to drop

.waitper1 testp #20 WC 'Test P20

if\_nc jmp #.waitper1 'Wait for C = 1

wypin DACvolt2, #20

nop 'Delay for IN to drop

.waitper2 testp #20 WC 'Test P20

if\_nc jmp #.waitper2 'Wait for C = 1

jmp #.myloop 'Do this loop again

'PWM dither

DACconfig long %0000\_0000\_000\_10100\_00000000\_01\_00011\_0

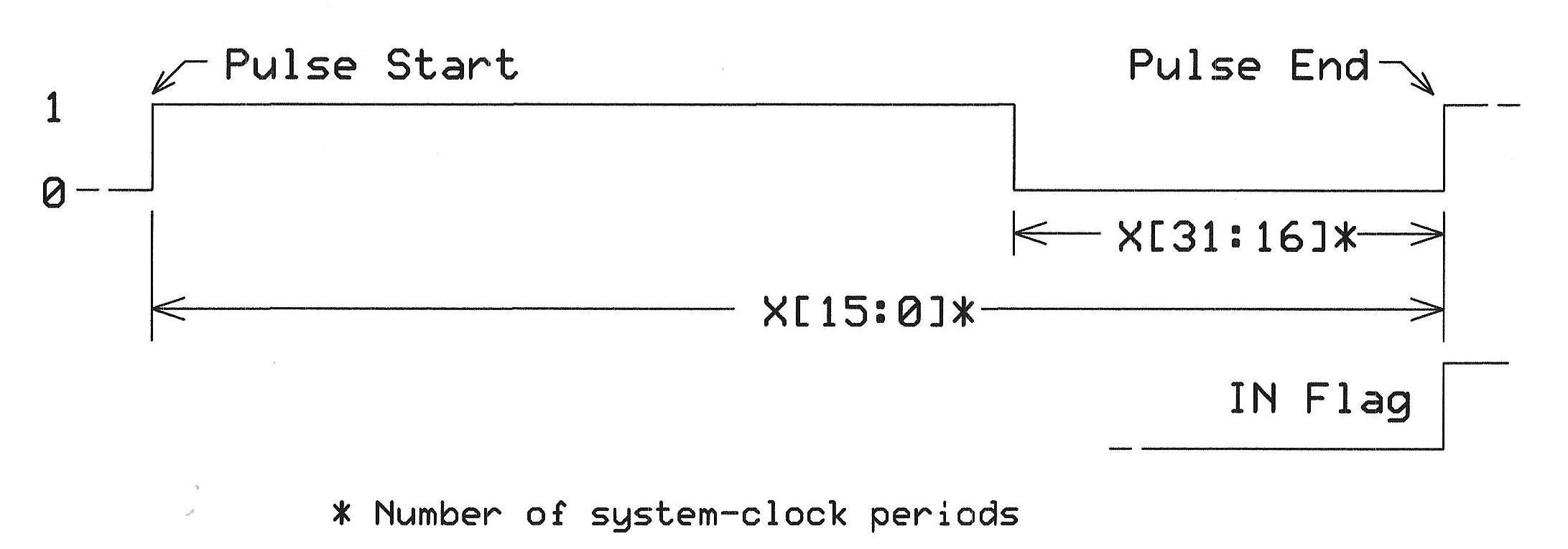
DACperiod long 256

DACvolt1 long $F00F 'wypin bits Y15:Y0 for voltage

DACvolt2 long $01F0

**--------------------------------------------------------------**

**%00100: Pulse/cycle output**

This mode lets a Smart Pin produce a series of logic-1 pulses. (If you need logic-0 pulses you can invert the pin output.) Software sets the pulse period, X[15:0], and the length of the logic-0 state, X[31:16], as shown in the figure below. These values represent the number of *system-clock periods* in each pulse section. Load the Y register Y[31:0] with the number of pulses you need. The Y value decrements by 1 for each pulse. When it decreases to 0, the Smart Pin raises its IN flag. This mode overrides OUT, and controls the pin's output state.

The following code example creates 16 logic-1 pulses:

CON

' 25-MHz system clock frequency

dat

org 0

dirl #20 'Setup Smart Pin at P20

wrpin PulseConfig, #20 'Set config for pulse/cycle

wxpin PulseTiming, #20 'Set cycle time and logic-0

' period

dirh #20 'Finished setup

wypin Cycles, #20 'Pulse count to Y register

nop 'Delay two clocks for IN to drop

.myloop

jmp #.myloop 'Program waits forever

PulseConfig long %0000\_0000\_000\_00000\_00000000\_11\_00100\_0 'Pulse/cycle mode

Cycles long $0010 'Pulse count of 16

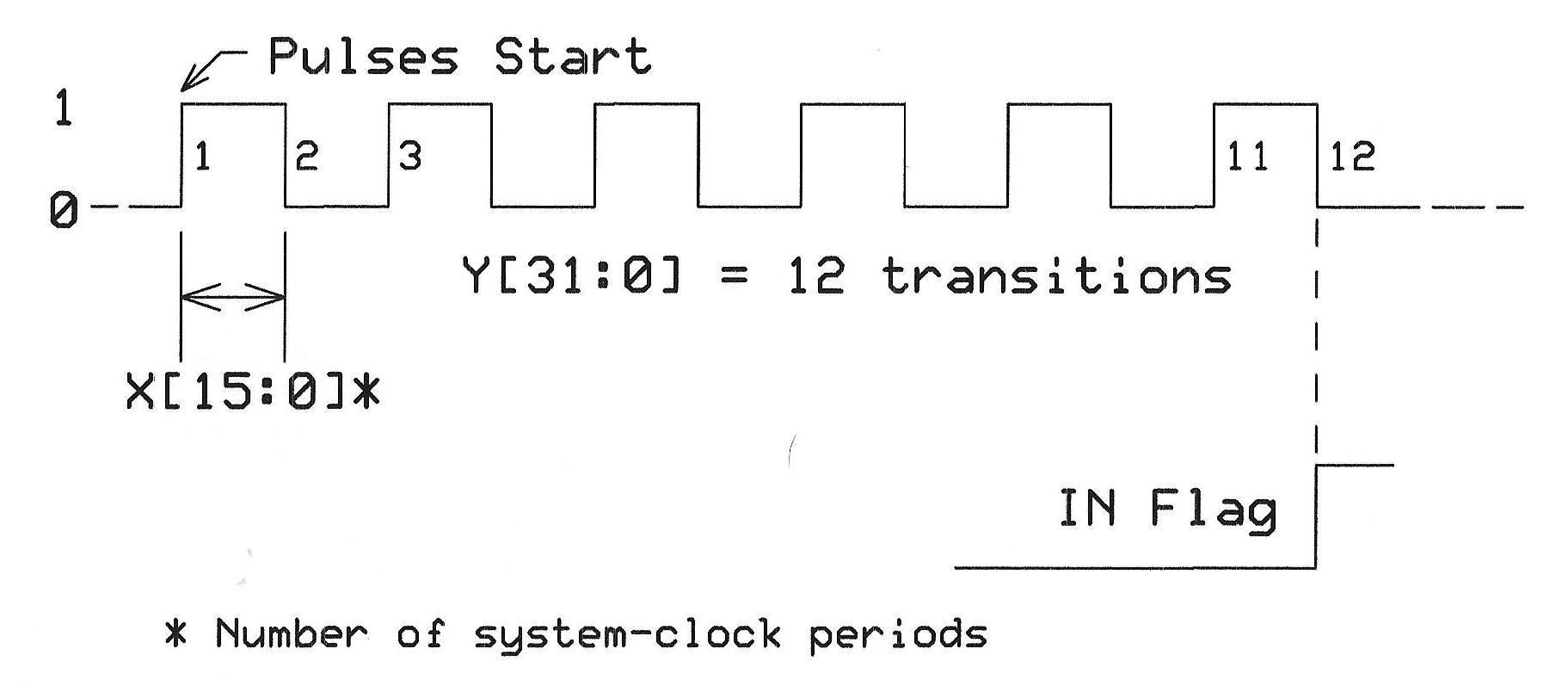
PulseTiming long $01F4\_05DC '60 usec pulse, 20 usec logic-0

To put out logic-0 pulses, replace the dirl instruction with dirh and change the PulseConfig Mode value P5 bit to a 1:

%0000\_0000\_000\_00000\_00***1***00000\_11\_00100\_0

^

**--------------------------------------------------------------%00101: Transition output**

This command creates a series of pulses with equal logic-0 and logic-1 periods. The value X15:0 sets that period as a count of system-clock periods. Use the Y register, Y[31:0], to set the number of *transitions*, or pulse edges, created. The Y value decrements after each edge or transition. When the Y-register value reaches 0, the IN flag gets set to a logic-1. During reset (DIR=0), IN is low, the output is low, and Y is set to zero. The Smart Pin pin has a default output of logic-0 before it starts to put out logic transitions.

The code below creates eight logic transitions. If you choose an odd number of transitions, the output remains a logic-1 when the Y-register value reaches 0.

CON

dat

org 0

dirl #20 'Setup Smart Pin at P20

wrpin TransConfig, #20 'Setup for Transition mode

wxpin TransTiming, #20 'Set time between transitions

dirh #20 'Finished setup

wypin Cycles, #20 'Send pulse count to Y register

.myloop nop 'Delay two clocks for IN to drop

jmp #.myloop 'Program waits forever

TransConfig long %0000\_0000\_000\_0000\_000000000\_11\_00101\_0 'Transition mode

Cycles long $0010 'transition count of 16

TransTiming long $0000\_05DC '1500 sys-clk periods between

' transitions

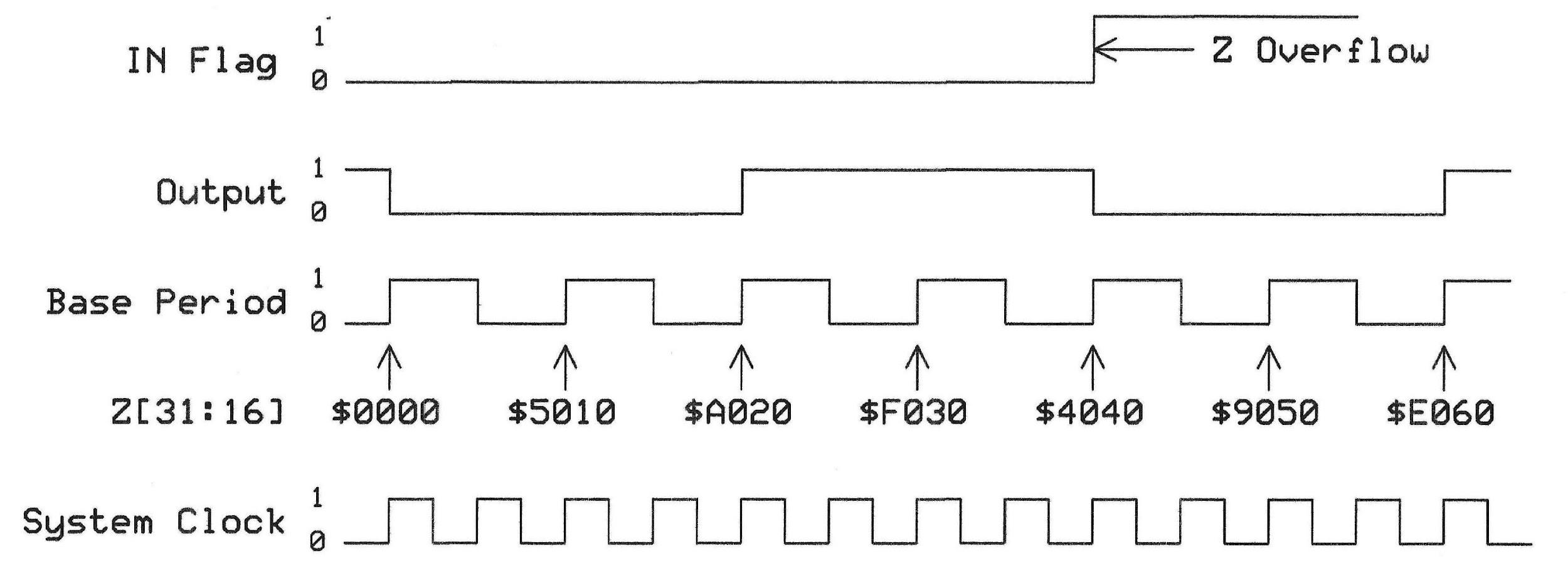
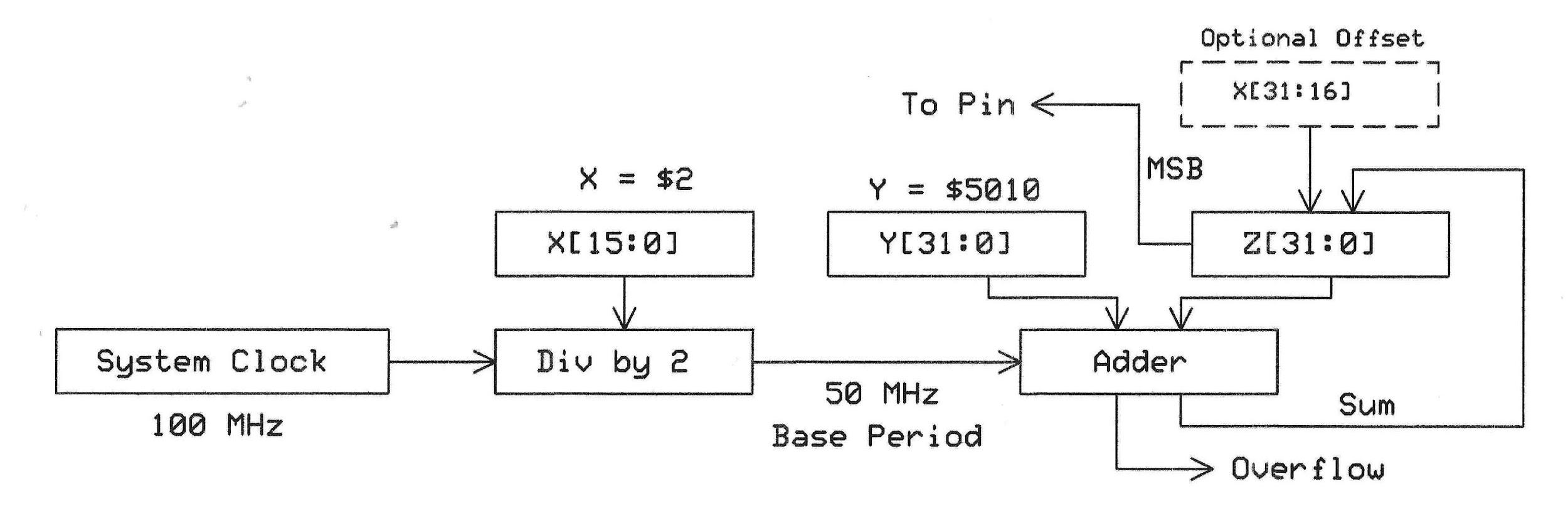
**--------------------------------------------------------------**

**%00110: Numerically Controlled Oscillator (NCO)**

The value X[15:0] (0 < n < 65,536) presets a 16-bit divide-by-n counter that produces an output--a base frequency--referenced to the system-clock frequency. To divide the system-clock frequency by 2, set X = #2.

The Y register (Y[31:0]) holds a value that gets added to a 32-bit value in the Z register (Z[31:0]) at the start of each base period. (This count starts at zero.) If you want a phase difference, use the bits X[31:16] to preload the counter as soon as the WXPIN instruction executes. The diagrams below show timing and a conceptual diagram of the operations.

Conceptual arrangement of a Smart Pin in the NCO Frequency mode:

The Pin output signal reflects the value of the Z31 bit. The circuit raises the IN flag whenever the Z register overflows (Z > 232). During reset (DIR=0), IN equals logic-0, the output is low, and Z is set to zero. This mode overrides OUT to control the pin output state.

Example:

CON

\_clk\_freq = 25\_000\_000

dat

org 0

dirl #20 'Setup Smart Pin at P20

wrpin NCO\_Config, #20 'Set configuration for NCO mode

wxpin #1, #20 'Set divide-by-n to 1, 25-MHz;

' one system-clock period

dirh #20 'Finished setup

qfrac ##123, ##\_clkfreq 'Calc #of 25-MHz cycles for

' 8 msec period

getqx pa 'Save in PA register

wypin pa, #20 'Send pulse count to Y register

.myloop nop 'Delay two clocks for IN to drop

jmp #.myloop 'Program waits forever

NCO\_Config long %0000\_0000\_000\_0000\_000000000\_01\_00110\_0

**--------------------------------------------------------------**

**%00111: Numerically Controlled Oscillator (NCO) Duty Cycle**

This mode operates much like the NCO Mode, %00110. In this mode, though, the Smart Pin create a logic-1 pulse followed by a logic-0. The value X[15:0] (0 < n < 65,536) presets a 16-bit divide-by-n counter that produces an output--an NCO base frequency--based on the system-clock frequency. *This base period represents the duration of the logic-1 pulse.*

The value in the Y register (Y[31:0]) determines the *time between the start of the logic-1 pulses*. This value gets added to a 32-bit value in the Z register (Z[31:0]) at the start of each base period. (This count starts at zero.)The Smart Pin output reflects the logic state of the Z-register overflow. If you want a phase difference, use the bits X[31:16] to preload the counter as soon as the WXPIN instruction executes.

Suppose a Propeller-2 has a 25-MHz system-clock frequency and you must create 1-usec logic-1 pulses with a period of 18 usec. An NCO base frequency of 1 MHz (1-usec period) will work. Set the X-register value to 25 to divide the system-clock frequency by 25. That provides a 1-usec logic-1 state.

Now you need 18 usec between the *start* of each pulse. Because the pin output follows the Z-register *overflow* you must divide the maximum Z-register value (232)\* by 18. That value added 18 times will produce a Z-register overflow:

232 / 18 = 238,609,294 or $0E38\_E38E

Set Y[31:0] to that value so it gets added to Z[31:0] at each base period. Of course, the cog can do the division if you wish.

The Smart Pin circuit raises the IN flag whenever the Z register overflows.

During reset (DIR=0), IN gets reset to logic-0, the pin puts out a logic-0, and Z[31:0] is reset to 0. This mode overrides OUT to control the pin output state.

\* Yes, the real maximum is 232-1, but compared to a value of over four billion, one count doesn't make a difference.

**Example**

CON

\_clkfreq = 25\_000\_000

dat

org 0

dirl #20 'Setup Smart Pin at P20

wrpin NCO\_Config, #20 'Set config for Transition mode

wxpin #25, #20 'Base period at 1 usec (25 MHz/25)

dirh #20 'Finished setup

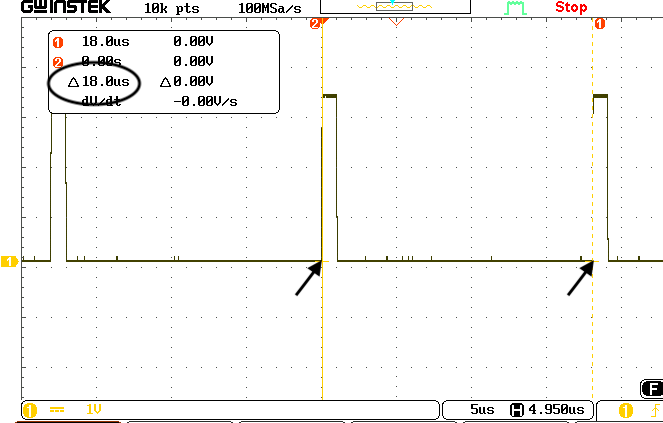
wypin Y\_RegData, #20

.myloop nop

jmp #.myloop 'Program waits forever

NCO\_Config long %0000\_0000\_000\_0000\_000000000\_01\_00111\_0

Y\_RegData long $0E38\_E38E ' 2^32 / 18

This scope image shows the signal present at pin P20. The added arrows indicate time-measurement points

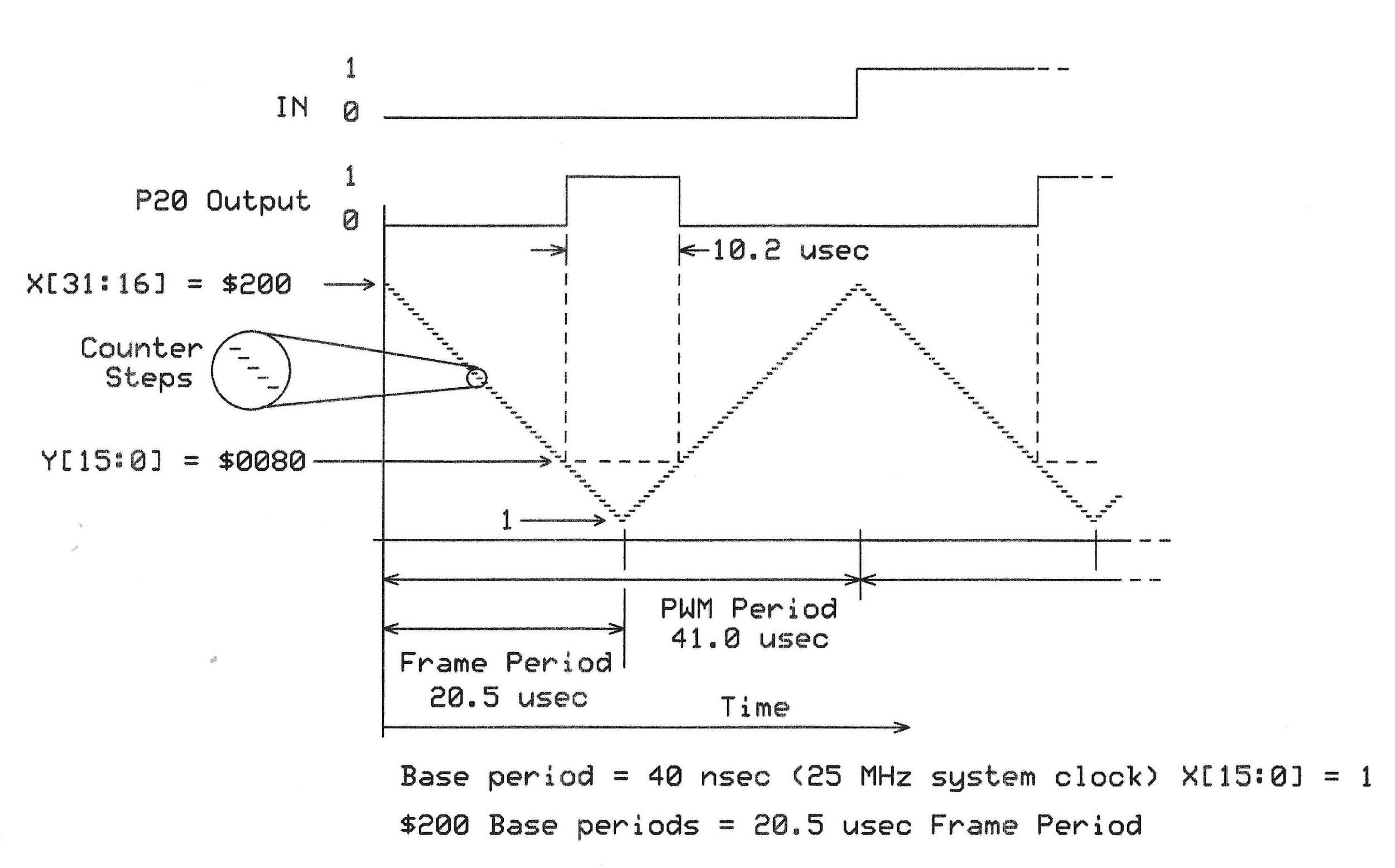
**--------------------------------------------------------------**

**%01000 = Triangle-wave pulse-width modulation (PWM)**

This mode relies on an up-down counter to determine the length of logic-1 pulses. The X-register value X[15:0] acts as a divisor of the Propeller-2 system-clock frequency. The divided frequency provides the PWM *base period*. In the example below X[15:0] equals 1, so no division occurs. The 25-MHz system-clock period--40 nsec--becomes the base period.

The X[31:16] bits sets the number of 40-nsec base periods you want in a *frame period*, as shown in the diagram. In this mode, the PWM period is *twice* the frame period. The example sets X[31:16] to $200 (512), so the frame period equals:

(40 \* 10-9 sec/base period) \* 512 base periods = 20.48 \* 10-6 sec, or a 20.5-usec

and the PWM period equals 41.0 usec.

The Y register value, Y[15:0], determines the logic-1 pulse width. In this example, Y = $80 (128). When the down-count value equals $80, the Smart Pin output becomes logic 1. The count continues to decrease to 1, at which time it starts to count up. For as long as the count is less-then or equal to $80, the output remains a logic-1.

This mode overrides OUT and controls the pin-output state. During reset (DIR=0), IN becomes logic 0, the pin output becomes logic-0, and the counter reloads the Y[15:0] value. The value 0 in register Y produces a constant logic-0 output. When the Y value equals the frame period value, the pin always produces a logic 1.

Example

'25 MHz system-clock frequency

COM

dat

org 0

dirl #20 'Setup Smart Pin at P20

wrpin NCO\_Config, #20 'Set configuration for PWM Triangle

wxpin X\_RegData, #20 'Set base period at 1 usec, or

' (25 MHz/#25)

dirh #20 'Finish setup

wypin Y\_RegData, #20

.myloop nop

jmp #.myloop 'Program waits forever

NCO\_Config long %0000\_0000\_000\_00000\_00000000\_01\_01000\_0

Y\_RegData long $0000\_0080

X\_RegData long $0200\_0001 'Divide system clock by 1

' Frame period = $200 (512)

**--------------------------------------------------------------**

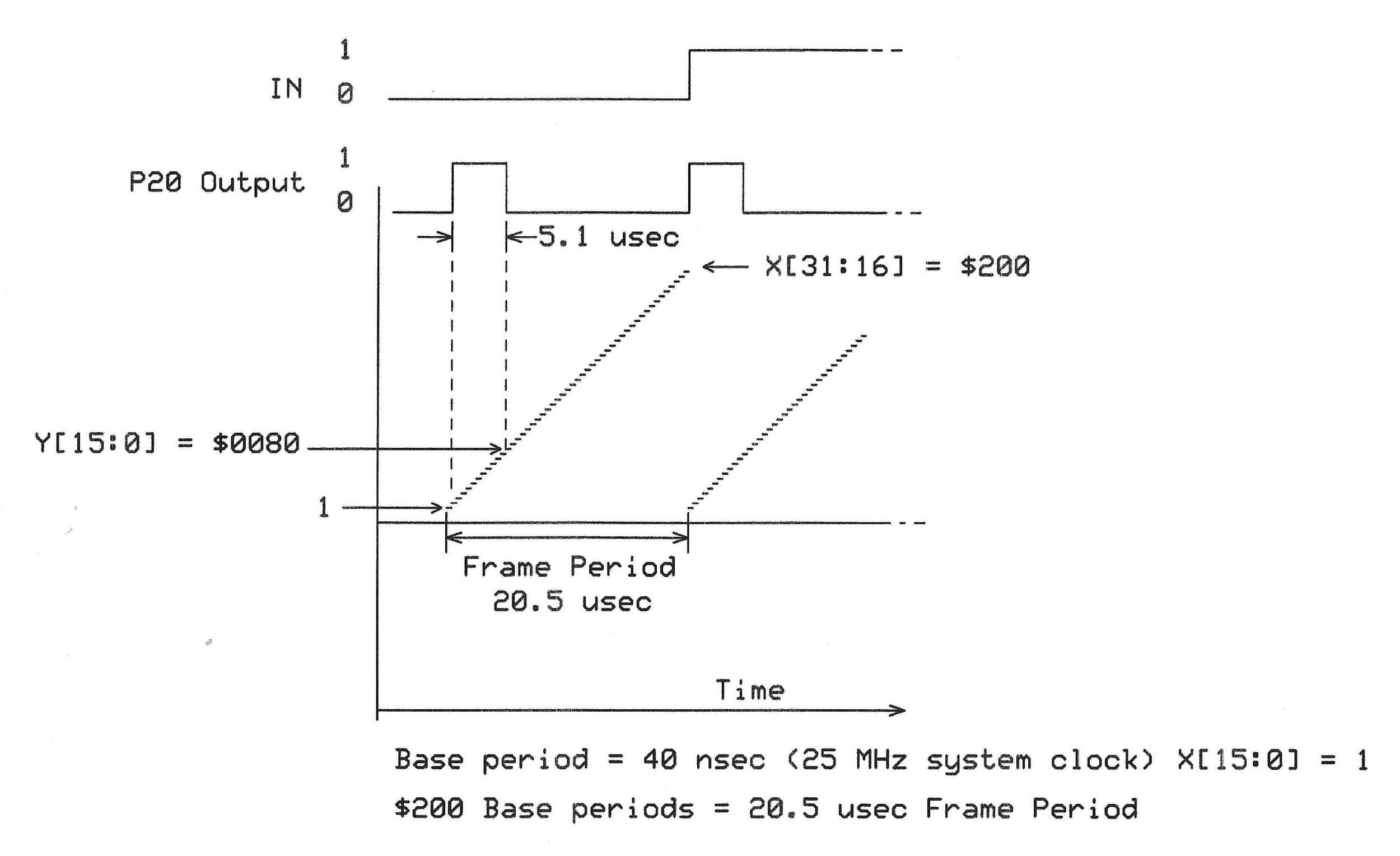
**%01001: PWM sawtooth**

This mode operates in much the same way as the %01000 PWM Triangle mode described above. But it uses only an up counter that always starts with a count of 1.

The X-register value X[15:0] acts as a divisor of the Propeller's system-clock frequency. The divided frequency provides the PWM *base period*. In the example below X[15:0] equals 1, so no division occurs. The 25-MHz system-clock period--40 nsec--becomes the base period.

The X[31:16] bits sets the number of 40-nsec base periods you want in a *frame period*, as shown in the diagram. In this mode, the frame period is the same as the PWM period. The example sets X[31:16] to $200 (512), so the frame period equals:

(40 \* 10-9 sec/base period) \* 512 base periods = 20.48 \* 10-6 sec, or a 20.5-usec

The value you put in the Y register, Y[15:0], sets the logic-1-pulse width. In the example, Y[15:0] equals $80 (128). The PWM logic-1 output starts when the counter gets reset to 1 and counting starts at the 40-nsec per count frequency. When the count equals or exceeds $80, the output changes to a logic-0. The count continues to increase, though.

'25 MHz system-clock frequency

COM

dat

org 0

dirl #20 'Setup Smart Pin at P20

wrpin PWMsaw\_Config, #20 'Set configuration for PWM

' Sawtooth mode

wxpin X\_RegData, #20 'Set base period at 1 usec;

' (25 MHz/#25)

dirh #20 'Finished setup

wypin Y\_RegData, #20

.myloop nop

jmp #.myloop 'Program waits forever

PWMsaw\_Config long %0000\_0000\_000\_00000\_00000000\_01\_01001\_0

Y\_RegData long $0000\_0080

X\_RegData long $0200\_0001 'Divide system clock by 1

'Frame period = 512

When the count reaches $200, the counter resets to 1, the output pin becomes a logic-1, and counting begins again. When the counter loads the Y[15:0] value again, IN becomes logic-1. This mode overrides OUT to control the pin output state.

The value 0 in register Y produces a constant logic-0 output. When the Y value equals the frame-period value, the pin always produces a logic 1. During reset (DIR=0), IN is low, the output is low, and Y[15:0] is captured.

**--------------------------------------------------------------**

**I don't know enough about SMPSs to work on revise this section. --Jon**

**%01010 = PWM switch-mode power supply with voltage and current feedback**

This mode overrides OUT to control the pin output state.

X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.

X[31:16] establishes a PWM frame period in terms of base periods.

Y[15:0] establishes the PWM output value which gets captured at each frame start and used for its duration. It should range from zero to the frame period.

A counter, updating at each base period, counts from one up to the frame period. Then, the 'A' input is sampled at each base period until it reads low. After 'A' reads low, Y[15:0] is captured, IN is raised, and the process repeats.

At each base period, the captured output value is compared to the counter. If it is equal or greater, a high is output. If it is less, a low is output. If, at any time during the cycle, the 'B' input goes high, the output will be low for the rest of that cycle.

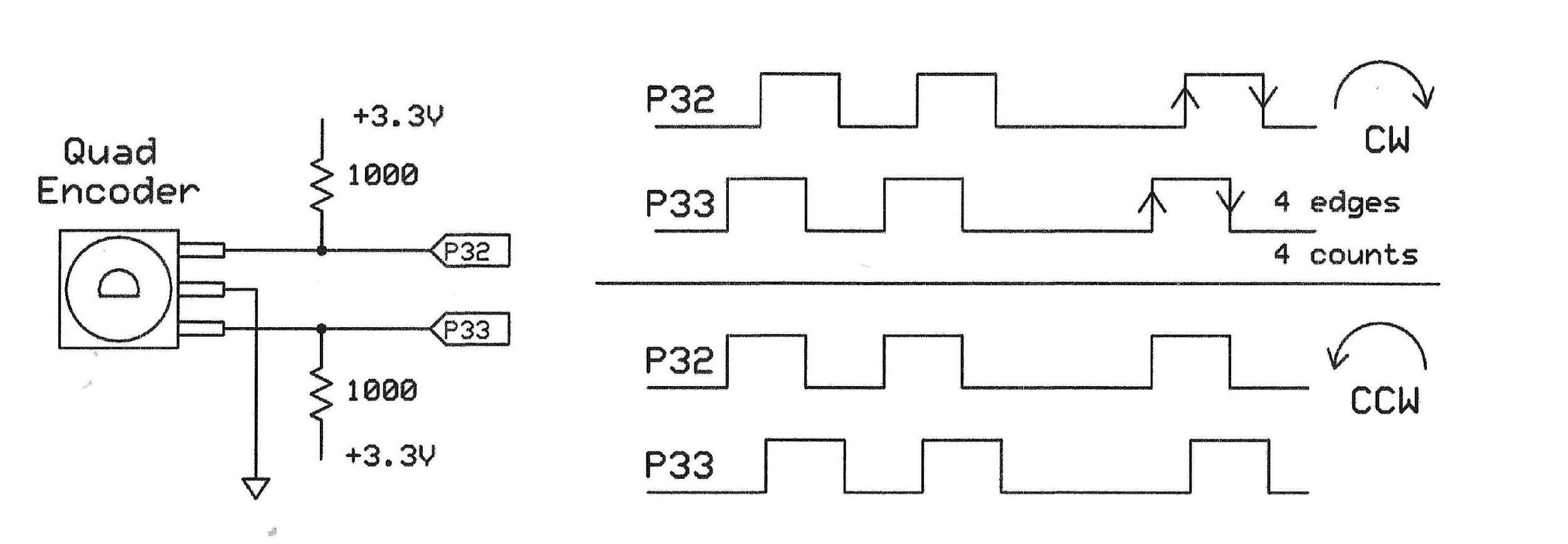
Due to the nature of switch-mode power supplies, it may be appropriate to just set Y[15:0] once and let it repeat indefinitely.

During reset (DIR=0), IN is low, the output is low, and Y[15:0] is captured.

**--------------------------------------------------------------**

**%01011: A/B-input quadrature encoder**

In this mode, you connect a quadrature encoder to a Smart Pin and the pin's Z register holds a 2's complement value that represents the *net total* encoder counts in one direction minus the counts in the opposite direction. Because an encoder provides two signals, you now use the BBBB bits in the 32-bit mode-control value to select a nearby pin for the second signal. Example programs use pins 32 and 33.



*Periodic measurements*: You set the X register pins X[31:0] for a measurement period in terms of system-clock periods. At the end of that period the Z register holds a 32-bit *net value* that represents the final number of counts from the encoder. Four counts clockwise (CW) and three counts counter clockwise (CCW) equals a net of one CW count at the end of the period.

Software can wait in a loop that tests the pin's IN flag. When detected, an RDPIN or RQPIN instruction reads a value from the Z register. If a count occurs coincident with the end of the sample period, it gets added to, or subtracted from, to the next period's count. So the Smart Pin does not miss any counts.

*Continuous measurements*: If you set the X register to $0, the Z-register's value continuously tracks the number of clicks, CW or CCW. In free-run operation you may read the current 32-bit quadrature step count from the Z register at any time with RDPIN or RQPIN.

You may set the quadrature encoder Z-register value to zero by pulsing DIR low at any time. There is no need to execute another WXPIN operation. During reset (DIR = 0), IN gets set to a logic-0, and Z is set to the adder value (0 / 1 / -1).

**Note:** A Smart Pin in quadrature encoder mode counts all four (4) logic transitions the encoder produces; two on the A input and two on the B input per "click" of the shaft. If you need only increments of one per click, use an arithmetic shift right by two bits to divide by four and preserve the sign (+ or -) of the count.

Reference: <https://en.wikipedia.org/wiki/Incremental_encoder>

The examples below use eight LEDs at pins P7:P0 to display the count.

**Example 1: Periodic Measurements**

COM

dat

org 0

mov dira, ##$FF 'Set P15..P8 as outputs for LEDs

dirl #32 'Setup Smart Pin at P32

wrpin QuadEnc\_Config, #32 'Set for Quad-Encoder mode

wxpin X\_RegData, #32 'Set sample period in system-clock

' periods

dirh #32 'Finished setup

.myloop nop

.wait\_here testp #32 WC 'Test carry at Smart Pin P32

nop

if\_nc jmp #.wait\_here 'No carry? Loop

rqpin QuadEnc\_data, #32 'Carry found, get total counts

sar QuadEnc\_data, #2 'Arithmetic shift right 2

' (divide by 4)

mov outa, QuadEnc\_data 'Send count to LEDs

jmp #.myloop 'Program loops forever

'Quad encoder mode, sets B as pin 32+1 (33)

QuadEnc\_Config long %0000\_0001\_000\_00000\_00000000\_00\_01011\_0

'Set 2-sec sample period (25-MHz clock)

X\_RegData long $02FA\_F080

QuadEnc\_data long $0

**Example 2: Free-Run Measurements**

COM

dat

org 0

mov dira, ##$FF 'Set P15..P8 as outputs for LEDs

dirl #32 'Setup Smart Pin at P32

wrpin QuadEnc\_Config, #32 'Set for Quad-Encoder mode

wxpin X\_RegData, #32 'Set continuous count update

dirh #32 'Finished setup

nop 'Short delay

.myloop nop

rqpin QuadEnc\_data, #32 'Get counts

sar QuadEnc\_data, #2 'Arithmetic shift right 2

' (divide by 4)

nop

mov outa, QuadEnc\_data 'send to LEDs

jmp #.myloop 'Program waits forever

'Quad encoder mode, sets B as pin 32+1 (33)

QuadEnc\_Config long %0000\_0001\_000\_00000\_00000000\_00\_01011\_0

X\_RegData long $0 'No period, continuous count-update

QuadEnc\_data long $0

You might find it useful to configure two Smart Pins that connect to same encoder inputs. and configure both for a quadrature encoder. One set configured for continuous measurements (X = 0) for absolute position tracking, and the other set for periodic measurements (X > 0) for velocity measurement.

Would someone set, say, pin 34 as a Smart pin with AAAA = 0110 and BBBB = 0111?

**--------------------------------------------------------------**

**%01100 = Gated Positive-Edge Counter**

Use this mode to count the number of positive edges (logic-0 going to logic-1) at a Smart Pin A input but only while the B pin has a logic-1 input. You may count edges over a given period or continuously. Note: This mode does not "debounce" mechanical-switch signals.

*Periodic measurements*: Set the X register X[31:0] for a measurement period in terms of system-clock periods. At the end of that period the Z register holds a 32-bit value that represents the number of positive edges detected on the A input only when the B input is set at logic-1. The measurement periods continue one after the other.

Software can wait in a loop that tests the pin's IN flag. When detected, an RDPIN or RQPIN instruction reads a value from the Z register. If a positive edge occurs coincident with the end of the sample period, it gets added to the next period's count. So the Smart Pin does not miss any counts.

*Continuous measurements*: If you set the X register to $0, the Z-register's value continuously tracks the number of positive edges. In this mode software may read the current 32-bit quadrature step count from the Z register at any time with RDPIN or RQPIN.

During reset (DIR=0), IN is low and Z is set to the adder value (0/1) to account for an edge that occurred coincident with the end of the periodic measurement time.

**Examples**

Use the examples shown previously for the %01011 = A/B-input Quadrature Encoder.

1. Change the QuadEnc\_Config value to:

%000\_0001\_000\_00000\_00000000\_00\_01100\_0

2. Comment-out the statement: 'sar QuadEnc\_data, #2

**--------------------------------------------------------------**

**%01101 = Positive-Edge Up-Down Counter**

This mode counts positive edges on a Smart Pin A input. The B input determines whether the count increments or decrements on the detected edges. As in other modes, you may count edges over a given period or continuously. When the B pin is at a logic-1, the count increments. A logic-0 at the B pin causes the count to decrement. You may change the logic-input signal going to the B pin at any time. Note: This mode does not "debounce" mechanical-switch signals.

*Periodic measurements*: Set the X register X[31:0] for a measurement period in terms of system-clock periods. At the end of that period the Z register holds a 32-bit value that represents the *net number of positive edges* detected on the A input based on the state or states of the B input during the counting period. The measurement periods continue one after the other. During reset (DIR = 0), IN becomes logic-0 and Z gets automatically set to an adder value (+1 / 0 / -1) that accounts for any edge that occurred coincident with the end of the periodic measurement time. The Smart Pin does not miss any counts.

Software can wait in a loop that tests the pin's IN flag. When detected, an RDPIN or RQPIN instruction reads a value from the Z register.

*Continuous measurements*: If you set the X register to $0, the Z-register's value continuously tracks the number of positive edges, again based on the state or states of the B input during the counting period. In this mode software may read the current 32-bit quadrature step count from the Z register at any time with RDPIN or RQPIN.

Examples

Use the examples previously shown for the %01011 = A/B-input Quadrature Encoder.

1. Change the QuadEnc\_Config value to:

%000\_0001\_000\_00000\_00000000\_00\_01101\_0

2. Comment-out the statement: 'sar QuadEnc\_data, #2

**--------------------------------------------------------------**

**%01110 AND Y = 0: Count A-input positive edges**

**%01110 AND Y > 0: Increment on A-input positive edge and decrement on**

**B-input positive edge**

The value in the X register governs whether the Smart Pin counts for a specific period or continuously.

a. Set X[31:0] = 0 when you want a continuous count, like a totalizer. Software can always read the current 32-bit count via an RDPIN or RQPIN instruction.

b. Set X[31:0] > 0 when you need a count during a specific period. To use this mode first determine how many system-clock periods will occur during that period. Second, put that value in the the X register. At the end of that period, the Smart Pin raises the IN flag. Your software may test the flag's state to detect when to read the count with an RDPIN or RQPIN instruction.

The value in the Y register determines whether to count 1. Only A-input positive edges (Y[31:0] = 0), or 2. To increment on an A-input positive edge and decrement on a B-input positive edge (Y[31:0] = 1).

During reset (DIR = 0), IN is low and Z is set to the adder value (0/1/-1) to account for a count event that occurred coincident with the end of a period. The adder value gets included in the next count period.

**Example**

Count A-input positive edges at pin P53. The B-input signal connects to pin 54, as enabled in the A\_in\_mode BBBB field.

A\_in = 53 'Pin P53, B input at pin P54

dat org

mov dira, ##$FF 'Set P7..P0 as outputs for LEDs

dirl #A\_in

wrpin A\_in\_mode, #A\_in 'Set up mode for pin P53

wxpin ##$773\_5940, #A\_in 'Set for 5-sec, 25 MHz clock

wypin #0, #A\_in 'Enable Smart Pin

dirh #A\_in

.loop rqpin my\_data, #A\_in 'Get count

mov outa, my\_data 'send to LEDs

jmp #.loop 'Loop to get new count

A\_in\_mode long %0000\_0001\_000\_0000000000000\_00\_01110\_0

my\_data long 0

This example increments a count on every A-input positive edges over 5 seconds (##$773\_5940). Change the WYPIN value to #1 and the Smart Pin increments the count on A positive edges and decrements the count on each B-input positive edge, also over 5 seconds. To count in either mode continuously, set the WXPIN value to 0.

--------------------------------------------------------------

**%01111 AND Y = 0: Count A-input highs**

**%01111 AND Y > 0: Increment on A-input high and decrement on B-input high**

The value in the X register governs whether the Smart Pin counts for a specific period or continuously.

a. Set X[31:0] = 0 when you want a continuous count, like a totalizer. Software can always read the current 32-bit count via an RDPIN or RQPIN instruction.

b. Set X[31:0] > 0 when you need a count during a specific period. The value you put in the X register determines the sampling period, in terms of system-clock periods. At the end of the set period, the Smart Pin raises the IN flag. Your software may test the flag's state to detect when to read the count with an RDPIN or RQPIN instruction.

The value in the Y register determines whether to: a. Increment the count only while the A input equals a logic-1 (Y[31:0] = 0), or b. To increment during a logic-1 on the A input and decrement during a logic-1 on the Binput (Y[31:0] = 1).

During reset (DIR = 0), IN is low and Z is set to the adder value (0/1/-1) to account for a count event that occurred coincident with the end of a period. The adder value gets included in the next count period.

The following example shows how to set up a counter for a 1-second period (25-MHz system-clock frequency). The 32-bit count appears as 32 bits shifted out via a synchronous-serial Smart Pin so you can monitor it with a logic analyzer. But you may use the 32 bits in any way you choose.

Example

' Synchronous serial transmission (SPI) of system-clock periods

' Positive trigger on the SPI clock, 25 MHz system clock

CON

clkout = 40 'Pin P40 'Transmitter clock @ P31

txout = 41 'Pin P41 'Transmitter data @ P30

A\_in = 53 'Pin P53 'Counter input

dat

org 0

dirl #txout 'Transmitter setup

wrpin sync\_tx\_mode, #txout 'Set sync tx mode for pin 41

wxpin #%1\_11111, #txout 'Set stop/start mode, 32 bits

dirh #txout 'Enable transmitter output

dirl #clkout 'Clock output setup

wrpin clock\_mode, #clkout 'Set pin as transition mode

wxpin ##$1000, #clkout 'Set base period for transmit

dirh #clkout 'Enable clock-output

dirl #A\_in 'Count A-input highs

wrpin A\_in\_mode, #A\_in 'Set up mode for pin P53

wxpin ##$17D\_7840, #A\_in 'Set continuous count 1-sec,

wypin #0, #A\_in 'Count only A-input highs

dirh #A\_in 'Enable Smart Pin

.test\_loop nop

testp #A\_in wc 'Get state of Carry flag

nop

if\_nc jmp #.test\_loop 'If no C flag, repeat testing

rdpin rcvd\_data, #A\_in 'Found carry bit, get data

wypin rcvd\_data, #txout 'Counter data sent via SPI

wypin #64, #clkout 'Start clock, transmit data

nop

jmp #.test\_loop 'Stay in this loop "forever"

A\_in\_mode long %0000\_0001\_000\_0000000000000\_00\_01111\_0

rcvd\_data long 0

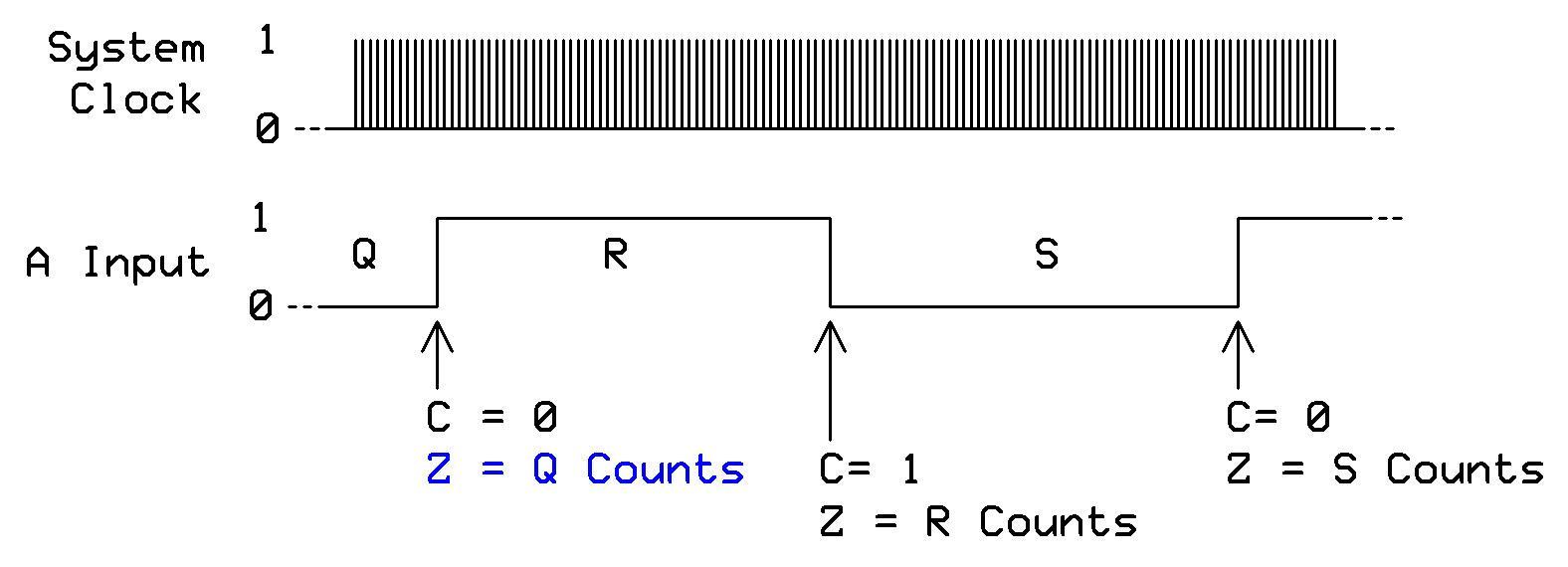
'Positive-edge clock mode

sync\_tx\_mode long %0000\_1111\_000\_0000000000000\_01\_11100\_0

clock\_mode long %0000\_0000\_000\_000000\_0\_000000\_01\_00101\_0

**--------------------------------------------------------------**

**%10000: Time A-input states**

This mode lets a Smart Pin *continuously* measure the time a pin is held in a logic-1 or a logic-0 state. A measurement period ends when the logic state at the input changes. At that time the carry flag indicates whether the measurement represents a logic-1 period, C = 1, or a logic-0 period, C = 0. Software can read the measurement from the Z register via an RQPIN instruction, which does not disturb the the IN, carry, or zero flags. Period measurements use the system-clock period as the unit of time measurement. The Z register has a maximum count of $8000\_0000. During reset (DIR=0), IN is low and Z is set to $00000001. The diagram below shows the results for three measurement periods.

If states change faster than the cog can retrieve measurements, you lose these counts. You can get around this problem when you use two Smart Pins to measure logic-1 states. Connect the signal to both pins but set the mode of one pin to invert the signal. Then, you could capture both states, as long as the sum of the states' periods doesn't exceed the cog's ability to retrieve both results. This technique would help in cases where one of the states was very short in duration, but the other wasn't.

The following example illustrates how to use the carry bit to distinguish between logic-level transitions and how to save the logic-1 and logic-0 measurements. The software continues to update the measurements for as long as you have the Smart Pin running in this mode. You might use this mode to measure PWM timing, in a control loop that must track proportion of logic states, and so on.

**Example**

CON

A\_in = 53 ' Pin P53

dat

org 0

dirl #A\_in 'Count A-input highs mode

wrpin A\_in\_mode, #A\_in 'Set up mode for pin P53

dirh #A\_in 'Enable Smart Pin

.test\_loop\_x nop

rqpin pin\_data, #A\_in 'Get Carry bit

if\_nc waitx #200 'Short delay

if\_nc jmp #.test\_loop\_x 'If C=0 wait for logic-1

mov Logic\_1\_count, pin\_data 'Get logic-1 count

.test\_loop\_y nop 'now wait for C = 0

rqpin pin\_data, #A\_in wc 'Get carry bit

if\_c waitx #200 'Short wait

if\_c jmp #.test\_loop\_y 'If C=1 wait for logic-0

mov Logic\_0\_count, pin\_data 'Get logic-0 count

jmp #.test\_loop\_x 'Go wait for a logic-1

A\_in\_mode long %0000\_0000\_000\_0001000000000\_00\_10000\_0

pin\_data long 0

Logic\_1\_count long 0

Logic\_0\_count long 0

**--------------------------------------------------------------**

**%10001: Time logic-1 states on A-input**

Use this method to measure the duration of a logic-1 state on a Smart Pin. When the pin detects a high-to-low edge, the count accumulated during the preceding logic-1 period goes into the pin's Z register. The IN flag rises, which lets software detect the period's. Use an RDPIN or RQPIN instruction to retrieve the measurement count. The Z register has a limit of $80000000 system-clock periods, the unit of time measurement. During reset (DIR = 0), IN goes to logic-0 and the Z register gets set to $00000001.

**Example**

CON

A\_in = 53 ' Pin P53

dat

org 0

dirl #A\_in 'Count while A-input high

wrpin A\_in\_mode, #A\_in 'Set up mode for pin P53

nop

dirh #A\_in 'Enable Smart Pin

.test\_loop nop

testp #A\_in wc 'Test flag for end of

if\_nc jmp #.test\_loop 'If C=0 wait for logic-1

rdpin pin\_data, #A\_in 'C = 1, so save count

jmp #.test\_loop

A\_in\_mode long %0000\_0000\_000\_0001000000000\_00\_10001\_0

pin\_data long 0

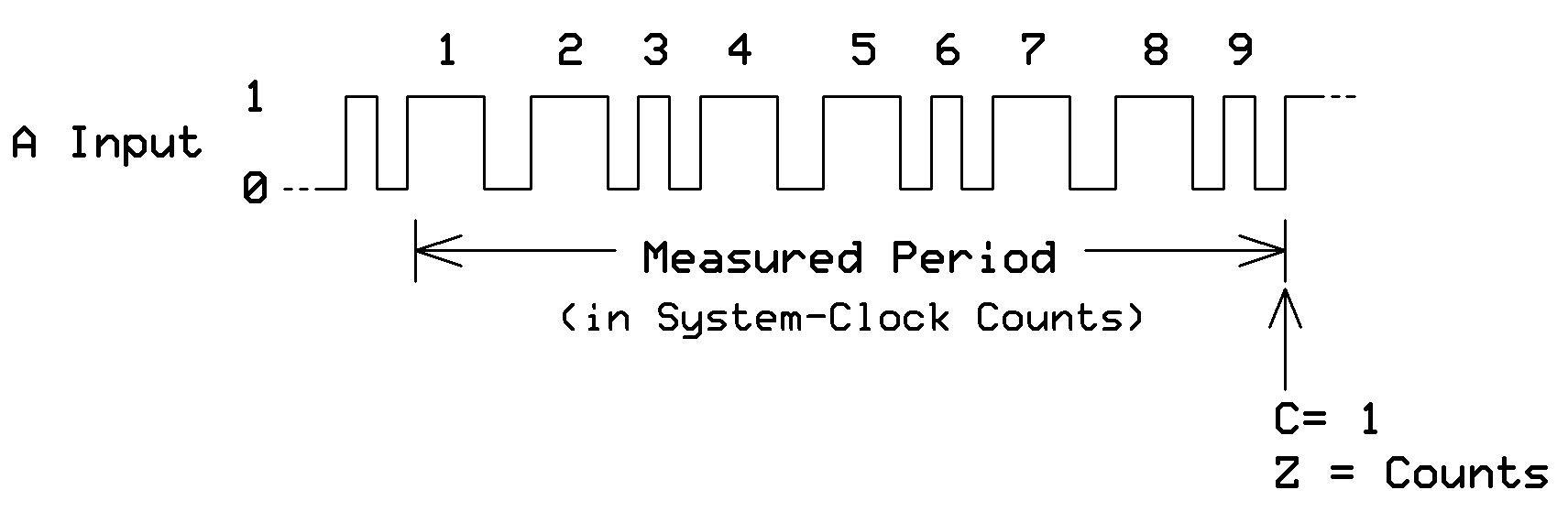
**--------------------------------------------------------------**

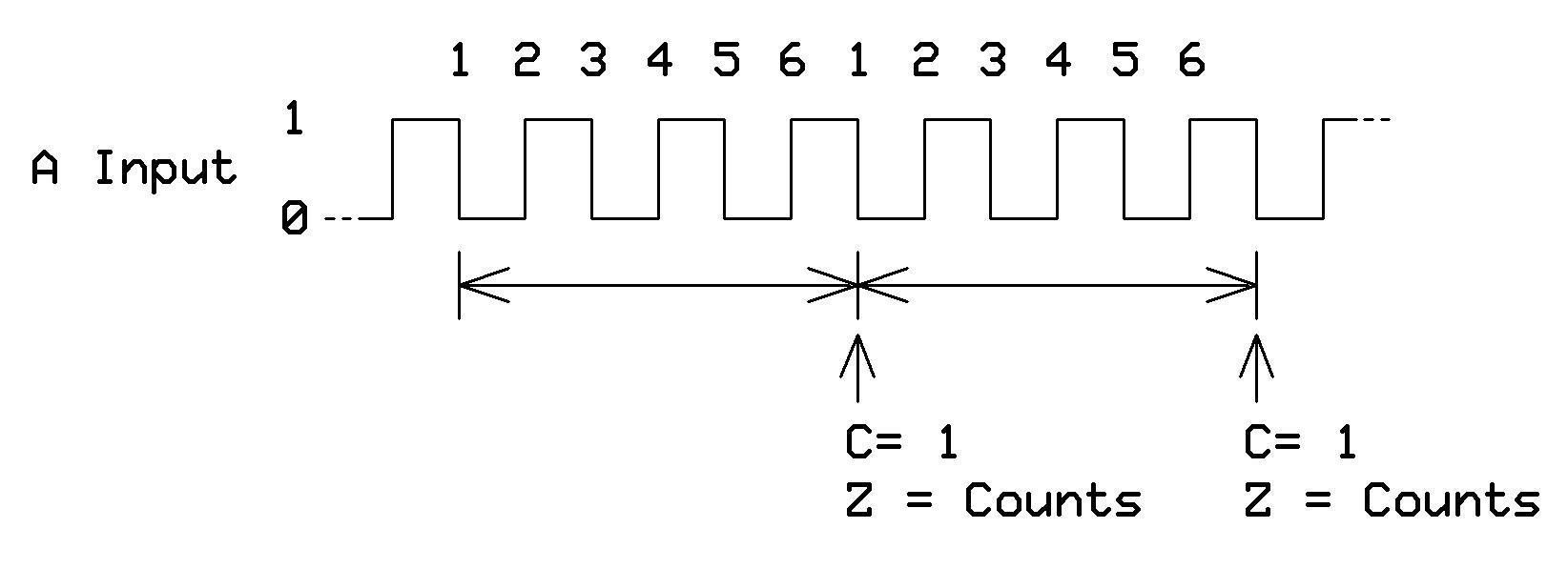
**%10010: Time A-input highs, rises, or edges (Y[2] = 0)**

You can use this mode in three ways, depending on how you set Y-register bits Y[2:0]. In all cases, the Smart Pin uses the system-clock frequency to increment a counter, and timing results go into the pin's Z register.

%000 = *A-input high (logic-1)* I have no idea what this mode measures. Or how to explain it.

%001 = *Measure the period of X number of pulse widths starting on a logic-0 to logic-1 edges on an A input.* The diagram below shows timing for X[31:0] = $9. At the end of the 9th pulse, the C flag gets set and the period becomes available in the pin's Z register. The diagram below illustrates the measurement period in terms of system-clock periods.

In this example, the pattern of three pulses repeats, so we choose 9, a multiple of three for the number of pulses to count to provide an accurate measurement. If you have pulse widths that vary from measurement period to measurement period, the timing values in the Z register will vary, too. See Note below.

%01x = *Measure the period between X number of logic edges on an A input.* The diagram below shows timing for six edges, X[31:0] = 6. At the end of the 6th edge, the C flag gets set and the period measurement becomes available in the pin's Z register. The diagram below illustrates the period timing. See Note below.

**Note:**  When the Smart Pin has detected the set number of your chosen event type it raises the IN flag. Your software can wait for the flag and then use an RDPIN or RQPIN instruction to retrieve the number of time increments. During reset (DIR = 0), IN goes to logic-0 and the Z register gets set to $00000001.The Z register can count as many as $80000000 system clock periods.

**Example**

CON

A\_in = 53 'Pin P53

dat

org 0

dirl #A\_in 'Use A\_in, pin 53

wrpin A\_in\_mode, #A\_in 'Set up mode for pin P53

wxpin ##$7, #A\_in 'Wait for $7 events

wypin ##%001, #A\_in 'Count A-input high states

dirh #A\_in 'Enable Smart Pin

.test\_loop nop

testp #A\_in wc

if\_nc jmp #.test\_loop 'If C=0 wait for logic-1

rdpin pin\_data, #A\_in 'C = 1, so save count

jmp #.test\_loop

A\_in\_mode long %0000\_0000\_000\_0001000000000\_00\_10010\_0

pin\_data long 0

--------------------------------------------------------------

**%10010: Detect missing A-input highs, rises or edge during**

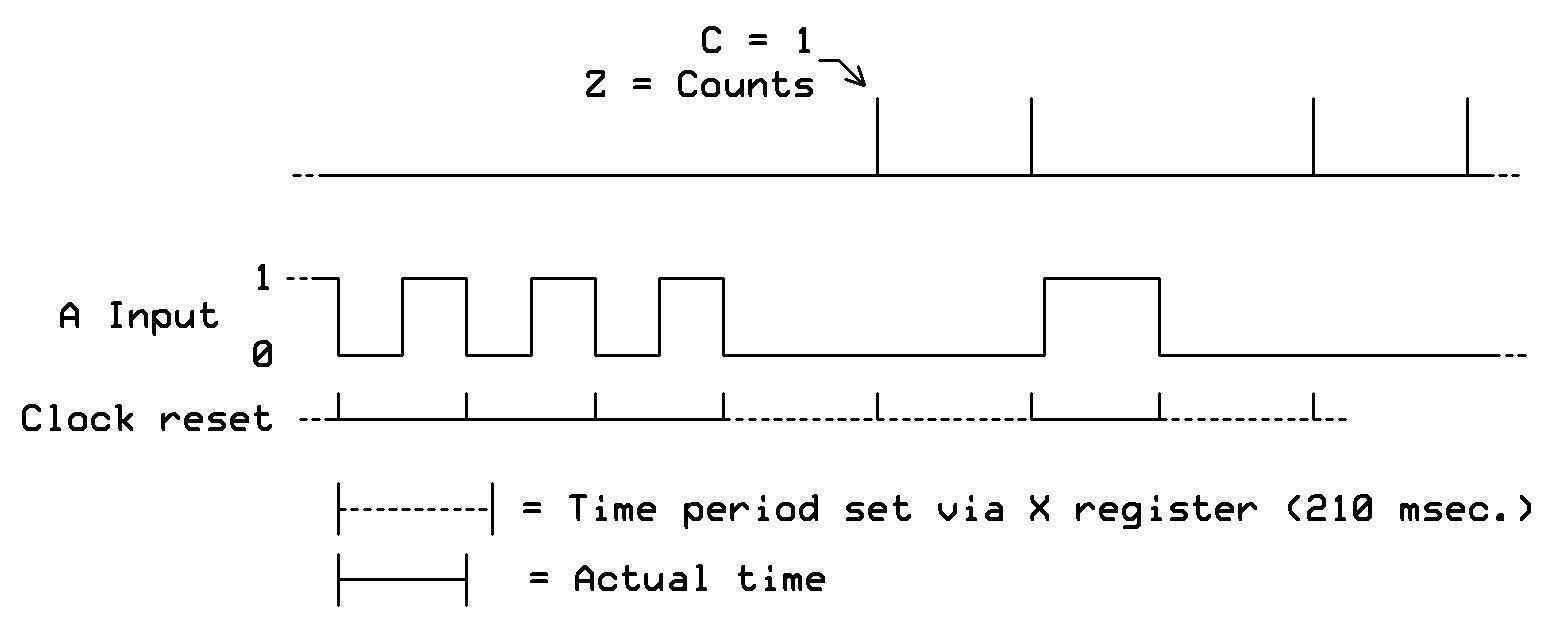
**a timeout period Y[2] = 1**

This mode supplements the mode just described but it lets your software detect missing A-pin events when they fail to occur during a specific period. The Y-register bits Y[2:0] control the event type. The X-register value (X[31:0]) determines the timeout period measured in number of system-clock periods.

%100 = Missing A-input high (logic-1)

%101 = Missing A-input rise (logic-0 to logic-1)

%11x = Missing A-input edge

When you initialize one of these modes, the clock starts and counts accumulate in the Z register at the system-clock frequency. If during this period the Smart Pin detects an event, the timer gets reset to $00000001 and a new count starts. The IN flag remains reset. The diagram below illustrates timing for the Y[2:0] = %100 setting.

If an event fails to occur during the period, the pin raises the IN flag and begins a new timing period. Software may detect the raised IN flag and use an RDPIN or RQPIN to read the Z-register value, which has an upper limit of $80000000 system-clock periods. This register maintains a running count of how many clocks have elapsed since the last A-input event.

During reset (DIR=0), IN is low and Z is set to $00000001.

**Example**

Use the same software example provided in the previous mode's description. To test this mode set the X register to a large value, for example, $50\_0000 or greater, and set the Y register %100, 101, or 110.

**--------------------------------------------------------------**

**%10011: Measure time for X events to occur**

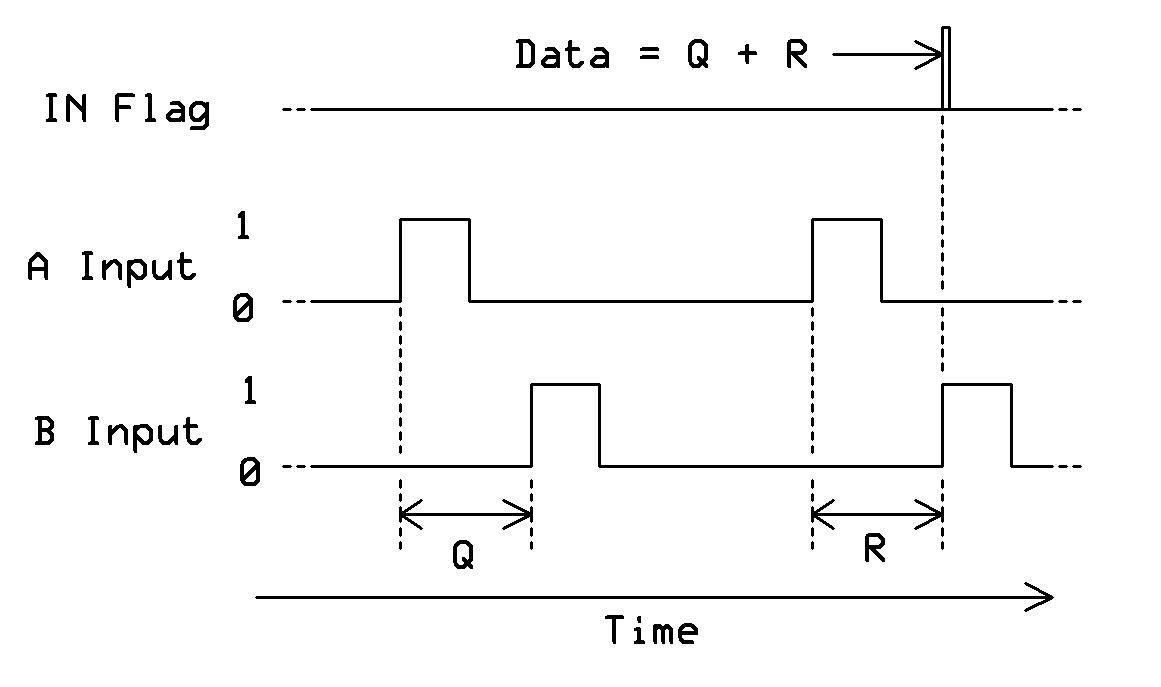
This mode provides a way to measure the time between two events, one on a Smart-Pin A input and the other on the pin's B input. The Y register bits Y[1:0] set the event triggers for the A- and B-pin inputs:

%00 = A-input rise to B-input rise

%01 = A-input rise to B-input edge

%10 = A-input edge to B-input rise

%11 = A-input edge to B-input edge

The value you put in the X register (X[31:0]) sets the number of A-input to B-input events that must occur before you get a result. The diagram below shows the measurements for *two events* (X[1:0] = %10) that yield period counts Q and R.

The *sum* of event these periods becomes available in the pin's Z register when the IN flag becomes a logic-1. You may use an RDPIN or RQPIN instruction to obtain the data. A new set of two measurements starts on the rising edge of the IN flag. The Z register has a limit of $80000000. During reset (DIR=0), IN is low and Z is set to $00000000.

Note: You may set the BBBB bits in the 32-bit mode-setting value to connect the B portion of the Smart Pin to the A pin if you want single-cycle measurements.

**Example**

CON

A\_in = 53 ' Pin P53

dat

org 0

dirl #A\_in 'Use A\_in, pin 53

wrpin A\_in\_mode, #A\_in 'Set up mode for pin P53

wxpin ##$2, #A\_in 'Wait for 2 events

wypin ##%00, #A\_in 'Count A-in rise to B-in rise

dirh #A\_in 'Enable Smart Pin

.test\_loop nop

testp #A\_in wc

if\_nc jmp #.test\_loop 'If C=0 wait for logic-1

rdpin pin\_data, #A\_in 'C = 1, so save count data

jmp #.test\_loop

'Set BBBB for A + 1 = 54

A\_in\_mode long %0000\_0001\_000\_0001000000000\_00\_10011\_0

pin\_data long 0

**--------------------------------------------------------------**

**%10100: Sum pulse duration over X pulses**

This mode requires two Smart-Pin inputs, A and B. The Y register bits Y[1:0] let you set one of the following event triggers for this mode:

%00 = A-input rise to B-input rise

%01 = A-input rise to B-input edge

%10 = A-input edge to B-input rise

%11 = A-input edge to B-input edge

Use the X register bits X[31:0] to set the number of event triggers over which you plan to measure the total duration of A-input logic-1 states. The diagram below shows how the mode works with the Y[1:0] bits set to %00, A-input rise to B-input rise, and the X register set to $2, so the measurement occurs for two events.

diagram

During the sampling period a counter sums the times time of each logic-1 state on the A-input signal. At the end of the X number of events the Smart Pin raises its IN flag and your software may use an RDPIN or RQPIN to retrieve the total time, in number os system-clock periods. The Z register has a limit of $80000000 clock periods.

At the end of the X number of events, the Smart Pin waits for the next event trigger at which it resets the counter and a new count begins. If an A-input rises or edge coincides with a B-input rise or edge at the end of the period, the start of the next period is registered. What does "registered" mean? During reset (DIR=0), IN is low and Z is set to $00000000.

Note: You may set the BBBB bits in the 32-bit mode-setting value to connect the B portion of the Smart Pin to the A pin if you want measurements for only one signal.

Example

--------------------------------------------------------------

**%10101 = For periods in X+ clock cycles, count time**

**%10110 = For periods in X+ clock cycles, count states**

**%10111 = For periods in X+ clock cycles, count periods**

I don't understand how these modes work or how someone would use them. Made no attempt at editing but could if someone explained mode operations. Chip has an example at:

<https://forums.parallax.com/discussion/170882/reciprocal-counter-demo/p1>

Perhaps someone could create a timing diagram that shows what happens. Then refer people to Chip's code, but with more comments for AL neophytes such as me.

X[31:0] establishes the minimum number of clock cycles to track periods for. Periods are A-input rise/edge to B-input rise/edge.

Y[1:0] establishes A-input and B-input rise/edge sensitivity:

%00 = A-input rise to B-input rise

%01 = A-input rise to B-input edge

%10 = A-input edge to B-input rise

%11 = A-input edge to B-input edge

Note: The B-input can be set to the same pin as the A-input for single-pin cycle measurement.

A measurement is taken across some number of A-input rise/edge to B-input rise/edge periods, until X clock cycles elapse and then any period in progress completes. If the A-input rise/edge is ever coincident with the B-input rise/edge at the end of the period, the start of the next period is registered. Upon completion, the measurement is placed in Z, IN is raised, and a new measurement begins. RDPIN/RQPIN can then be used to retrieve the completed measurement. Z will be limited to $80000000.

The first mode accumulates time within each period, for an oversampled period measurement.

The second mode accumulates A-input trigger states within each period, for an oversampled duty measurement.

The third mode counts the periods.

Knowing how many clock cycles some number of complete periods took, and what the duty was, affords a very time-efficient and precise means of determining frequency and duty cycle. At least two of these measurements must be made concurrently to get useful results.

During reset (DIR=0), IN is low and Z is set to $00000000.

**--------------------------------------------------------------**

**%11000: ADC sample, filter, convert, internally clock**

**%11001: ADC sample, filter, convert, externally clock**

Questions:

1. Should this section provide the input-voltage limits for the ADCs?

2. Does the SINC filtering remove the need for an anti-alias filter on an analog input?

3. What is the impedance of an ADC input?

These analog-to-digital converter (ADC) modes let you convert an analog signal into a digital value with from two to 18 bits of resolution, depending on the mode you choose. You may use an external delta-sigma ADC, as well.

*Internal-clock mode, %11000:* The Smart Pin samples the A-input analog signal at a rate that depends on the system-clock frequency. To configure the pin for ADC operation, set P[12:10] = %100 in the 32-bit configuration value. Software examples for this mode use a Smart Pin's internal ADC. Software examples demonstrate only the internal-clock mode.

*Externally-clocked mode, %11001:* This type of operation requires an external delta-sigma ADC that supplies a stream of data bits synchronized with its clock. A Smart Pin will sample the A-input data at every positive-going edge of the clock signal sent to the B input.

The table below shows combinations of the X-register mode-settings, bits X[5:4] for the sampling type and bits X[3:0] that set the ADC resolution. Not all mode-and-period combinations are useful or functional.

Table goes here...

For modes other than SINC2 Sampling, when X[5:4] > %00, you may override the initial period and set a different sample period via the Y-register bits Y[13:0]. If you want SINC3 filtering with a period of 320 clocks, for example, you set the Smart Pin mode with a WXPIN instruction and follow it with the instruction:

WYPIN #320, your\_adc\_smart\_pin\_number

At the end of each sample period, the Smart Pin puts the ADC-measurement value in the Z register. It raises the IN flag and starts a new measurement. Your software can detect the raised IN flag and use RDPIN or RQPIN instructions to retrieve the measurement result.

**About SINC2 and SINC3 filtering**

According to Wikipedia: "In signal processing, a sinc filter is an idealized filter that removes all frequency components above a given cutoff frequency, without affecting lower frequencies, and [it] has [linear phase](https://en.wikipedia.org/wiki/Linear_phase) response." (Ref 1.) Engineers can think of the sinc function as an excellent "brick-wall" low-pass filter.

SINC2 filtering is best for DC measurements, where precision is important. A Smart Pin can make useful measurements with 14-bit resolution every 8,192 clocks by using SINC2 filtering. After starting SINC2 filtering, the filter will become accurate starting on

the second period. This process has the pleasant effect of returning an extra bit of resolution above that obtained with simple bit-summing, and it filter away rectangular-sampling-window effects.

During each clock period a SINC2 filter sums the newest bit from a sigma-delta ADC in an accumulator. In turn, this value gets added to a second accumulator, to create a double integration. At the end of each sampling period, the difference between the new and previous second accumulator's value is the conversion sample, and the 'previous' value is updated. I recall Chip has a block diagram of this operation. Let's include it.

SINC3 filtering is like SINC2, but it employs an additional level of accumulation to increase sensitivity to dynamics (what does this mean, changes?) in the input signal. SINC3 doubles the effective number of bits (ENOB) over simple bit-summing for fast signals, but it is only slightly better than SINC2 filtering at DC measurements over the same sample period.

Because SINC3 takes more Smart Pin circuitry it is has an upper limit of 512 samples per period, which makes it less practical than SINC2 for precision DC measurements, but quite ideal for tracking fast, dynamic signals. After starting SINC3 filtering, the filter will become accurate starting on the third period.

For more information about sinc filters, see chapter 16, "Windowed-Sinc Filters," in *The Scientist and Engineer's Guide to Digital Signal Processing*, by By Steven W. Smith, Ph.D. https://www.dspguide.com/ch16.htm.

**SINC2 Sampling Mode** (X[5:4] = %00)

This mode performs complete SINC2 analog-to-digital conversions and updates the ADC digital output value at the end of each sample period. After you set a Smart Pin in this mode, simply use an RDPIN or RQPIN instruction to read the most-recent ADC result. *Important:* This mode only works when you choose a power-of-2 sample period as shown in the table. This restriction relates to the efficient implementation of the filter function within the Smart Pin, so conversions occur without software intervention.

**Example**

CON

A\_ADC = 46 ' Pin P46

dat

org 0

mov dira, #$FF 'LED output

dirl #A\_ADC 'ADC reset

wrpin A\_ADC\_mode, #A\_ADC 'Set up mode for ADC

wxpin #%00\_0111, #A\_ADC '8-bit resolution

dirh #A\_ADC 'Enable Smart Pin

.test\_loop nop

waitx ##$100\_000 'Delay between output

rdpin pin\_data, #A\_ADC 'Get ADC result

mov outa, pin\_data 'Data to LEDs

jmp #.test\_loop

A\_ADC\_mode long %0000\_0000\_000\_100011\_0000000\_00\_11000\_0pin\_data long 0

*Important:* The variable A\_ADC could enable several pins by having the additional number of pins set via the variable's bits 10:6. If A\_ADC held %00111\_010000, for example, you would have physical pins P16 through P23 simultaneously configured as ADC SINC2 inputs.

**SINC2 Filtering Mode** (X[5:4] = %01)

This mode provides an additional SINC2 filtering mode that lets you choose non-power-of-2 sample periods. But your software must perform the difference computation.

Example

CON

A\_ADC = 46 ' Pin P46

dat

org 0

mov dira, #$FF 'LED output

dirl #A\_ADC 'ADC

wrpin A\_ADC\_mode, #A\_ADC 'Set up mode for ADC

wxpin #%01\_0111, #A\_ADC '8-bit resolution

dirh #A\_ADC 'Enable Smart Pin

setse1 #%001<<6 + A\_ADC 'Special event trigger

' #1 on A\_ADC high

.test\_loop waitse1 'Wait for #1 event

rdpin pin\_data, #A\_ADC 'Event detected, get ADC data

sub pin\_data, diff 'Compute sample

ADD diff, pin\_data 'Update diff value

SHR pin\_data, #6 'Right-justify data

mov outa, pin\_data 'Display result on LEDs

jmp #.test\_loop 'Run loop again

A\_ADC\_mode long %0000\_0000\_000\_100011\_0000000\_00\_11000\_0

pin\_data long 0

diff RES 1

Note: To leave the ENOBs intact, your software must shift the computed ADC value to the right. For SINC2 filtering, shift the value right by [LOG2(samples-per-period)]-1, which in this case is LOG2(128) - 1 = 6, or six bit positions to the right.

**SINC3 Filtering Mode** (X[5:4] = %10)

This mode performs SINC3 filtering, but it requires additional software to obtain accurate ADC values. To begin SINC3 filtering:

WRPIN ##%100011\_00000000\_00\_11000\_0,#adcpin 'configure ADC-and-

' filter pin(s)

WXPIN #%10\_0111,#adcpin 'SINC3 filtering at

' 128 clocks

DIRH #adcpin 'enable Smart Pin(s)

You should add software to detect when the pin's IN flag goes to logic-1:

SETSE1 #%001<<6 + adcpin 'SE1 triggers on pin high

.loop WAITSE1 'wait for sample period done

RDPIN x,#adcpin 'get SINC3 accumulator

SUB x,diff1 'compute sample

ADD diff1,x 'update diff1 value

SUB x,diff2 'compute sample

ADD diff2,x 'update diff2 value

SHR x,#7 'justify 14-bit sample

'now use the "x" data from the ADC as you choose

JMP #.loop 'loop for next period

x RES 1 'sample value

diff1 RES 1 'diff1 value

diff2 RES 1 'diff2 value

To leave the ENOBs intact, your software must shift the computed ADC output value to the right. For SINC3 filtering, you must shift right by LOG2(samples per period), which in this case is LOG2(128) = 7, or seven bit positions to the right.

**Bitstream Capturing Mode** (X[5:4] = %11)

This mode captures the raw bitstream data from a Smart Pin's delta-sigma ADC. It saves the ADC's 32-bit result and your software should read it after every 32 system-clock periods so you get a "snapshot" of contiguous bits in the ADC bitstream. Use the RDPIN or RQPIN instruction to read the snapshots data. In the data, bit 31 will be the most recent ADC bit, while bit 0 will have come from the ADC 31 system-clock periods earlier.

To begin raw-bitstream capturing:

WRPIN ##%100011\_00000000\_00\_11000\_0, adcpin 'configure ADC-sample

' pin(s)

WXPIN #%11\_0101, adcpin 'raw sampling, new result

'every 32 clock periods

DIRH adcpin 'enable Smart Pin(s)

To get a snapshot of the latest 32 bits of ADC bit-stream data, use an RDPIN or RQPIN instruction:

RDPIN your\_bitstream\_data, adcpin 'get snapshot of ADC bitstream

This mode captures the state of the A-input without regard to its pin configuration, so you may use it for purposes other than capturing ADC bitstreams.

Ref. 1. https://en.wikipedia.org/wiki/Sinc\_filter

**--------------------------------------------------------------**

**%11010 = ADC Scope with Trigger**

Need help with this section. Can someone create a very simple example that gets data from the four ADCs and does something with it? There's an example on the forum that uses STREAMERs, but I don't understand how it works. Is it necessary to have an example that uses STREAMERS?--Jon

This mode simultaneously captures the state of as many as four contiguous Smart Pins. Think of this mode as a 4-trace oscilloscope or a 4-channel data-acquisition device on a single chip. Before you use this mode we highly recommend you read the data-sheet section STREAMERS, and understand how streamers work.

Each of the four analog inputs connects to its own 8-bit ADC and one of those inputs also serves as a trigger that starts data acquisition. This type of *hysteretic trigger* uses an arming voltage and a trigger voltage you set. The cog tests for triggering on every system-clock cycle. You must choose an increment of four (0, 4, 8,12...) for the starting pin number. The examples below use pins 52 through 55.

Each *cog* has its own 32-bit "data pipe" that continuously combines the four ADC bytes into a long value. You can use a GETSCP instruction to read these concatenated values simultaneously as a 32-bit value or use an RDPIN instruction to obtain the ADC data from a specific pin. The SCOPE data pipe is generic, so you might find uses other than holding "scope" data.

To use the SCOPE mode:

**1.**  Use a WRPIN instruction to configure all four pins:

wrpin scpmode, #scp\_addr

scpmode = %0000\_0000\_000\_100011\_0000000\_00\_11010\_0

scp\_addr = %00011\_110100

The SCPMODE value uses the standard 32-bit format that includes the PAD-IO Mode bits for an ADC input.

The SCP\_ADDR bits D[5:2] equal the starting pin number, 52. Bits D[1:0] must equal 0 for the scope mode. The SCP\_ADDR value includes bits D[7:6] that indicate "and the following three pins."

**2.** The following figure shows the three windowing functions that filter ADC samples. At the end of a system-clock period the incoming ADC bit shifts into a hardware filter that produces the sample value. This process normalizes ADC results to 8-bits values, but the DC dynamic range amounts to between five and six bits, depending on the filter you choose. The following plots illustrate filter shapes and sizes:

Filter diagram here..

**3.** The scope mode requires a WXPIN instruction that combines the trigger and filter settings.

wxpin scp\_x, #scp\_addr

The trigger and arm values in SCP\_X range from 0 to 252 in multiples of four. The two LSBs are ignored in the value expressed as:

TTTTTT00AAAAAAFF

T = D[15:10] sets the trigger level, 6 MSBs.

A = D[7:2] sets the arm-trigger level, 6 MSBa.

F = D[1:0] selects the filter type:

%00 = 68-tap Tukey filter

%01 = 45-tap Tukey filter

%1x = 28-tap Hann filter

To simplify programming, use the following code to create the SCP\_X value:

scp\_x long (trigger\_level & $FC)<<8 + (arm\_level & $FC) + scope\_filter

Triggering the start of 4-input measurements requires a two-step process; first the Smart Pin (in this example pin P54) must detect a voltage that meets or exceeds the trigger value. Remember, arm and trigger values are proportional to the full-scale voltage range of the ADC input.

After arming occurs, the Smart-Pin input waits to detect a voltage equal to or greater than the trigger setting. Once triggered, the pin raises its IN flag and waits for another arming event. Does the software need to "rearm" the pin at some point?

The table below shows how the relationships between the arm and trigger conditions:

Insert table here. In this table it's not clear what A or B represent. Please clarify.

What does *sample[7:2]* relate to?

What happens if the arm and trigger values are equal, or if both are zero? How would you get the SCOPE mode to continuously sample--like a strip-chart recorder without a trigger? I suppose you could get it to sample every 1 second and have plenty of time to do things between those samples? Or does it always take an ADC reading every system-clock cycle?

4**.** Use the SETSCP instruction to enable the scope inputs:

setscp #1<<6 + scp\_addr

When D[6] = 1 it enables the scope inputs and D[5:2] supplies the pin base number, 52.

5**.** Establish a special-event trigger event, SETSE1. This instruction assigns the trigger event to the first pin, P52. Bits D[6:0] identify the pin, and D[8:7] enable that pin:

setse1 #%001<<6 + scp\_addr

6**.** Set the sample period. The SETXFRG instruction sets the NCO for the cog's STREAMER. Review the STREAMER section of this document for information about how to set this period. In the example program, the value $2000\_0000 creates a sample period of four system-clocks periods. (At 250 MHz, that is every 16 nsec.)

Is that correct or is there another clock and frequency for the streamer?

setxfrq ##$2000\_0000

**6.** Wait for the trigger event, then capture data. The example program includes steps that lead to the WAITSE1 instruction, which causes the cog to wait until special event 1 occurs at the P52 input pin. After that event the program waits for a period that allows the streamer to acquire 640 values, after which the scope action ends with XSTOP:

waitse1 'wait for new trigger event

xinit scp\_d,#0 'trigger detected, start all 4 ADCs

waitx ##640\*4 'allow time to read 640 scope values\*

xstop 'stop streamer, ~640 samples gathered

' in 1k sample buffer

\*Note: 640 multiplied by four because samples are taken every 4 system? clocks.

An RDPIN or RQPIN instruction always returns the 8-bit ADC sample along with the armed-state flag in the C flag. Logic-1 is armed?, logic-0 is unarmed?. Just my guess.

You may use the GETSCP instruction to read the SCOPE data pipe's current four bytes, shown arranged in an earlier diagram in this section:

getscp D 'Put the 8-bit ADC values in long variable D

If the SCOPE data pipe didn't exist, the closest you could come to the GETSCP instruction would be this sequence, which would not have time-aligned samples:

pinblock = 52

RQPIN x,#pinblock | 3 'read pin3 long into x

ROLBYTE y,x 'rotate pin3 byte into y

RQPIN x,#pinblock | 2 'read pin2 long into x

ROLBYTE y,x 'rotate pin2 byte into y

RQPIN x,#pinblock | 1 'read pin1 long into x

ROLBYTE y,x 'rotate pin1 byte into y

RQPIN x,#pinblock | 0 'read pin0 long into x

ROLBYTE y,x 'rotate pin0 byte into y

**Example**

**--------------------------------------------------------------**

I don't know enough about the USB bus and how it operates to edit/revise this section. --Jon

**%11011 = USB host/device**

This mode requires that two adjacent pins be configured together to form a USB pair, whose OUTs (what is this?) will be overridden to control their output states. These pins must be an even/odd pair, having only the LSB of their pin numbers different. For example: pins 0 and 1, pins 2 and 3, and pins 4 and 5 can form USB pairs. They can be configured via WRPIN with identical D data of %1\_11011\_0. Using D data of %0\_11011\_0 will disable output drive and effectively create a USB 'sniffer'. A new

WRPIN can be done to effect such a change without resetting the Smart Pin. **NOTE: In Propeller 2 emulation on an FPGA, there are no built-in 1.5k and 15k resistors, like the ASIC Smart Pins have, so it is up to you to install these yourself on the DP and DM lines.**

WXPIN is used on the lower pin to establish the specific USB mode and set the baud rate. D[15] must be 1 for 'host' or 0 for 'device'. D[14] must be 1 for 'full-speed' or 0 for 'low-speed'. D[13:0] sets the baud rate, which is a 16-bit fraction of the system clock, whose two MSBs must be 0, necessitating that the baud rate be less than 1/4th of the system clock frequency. For example, if the main clock is 80MHz and you want a 12MHz baud rate (full-speed), use 12,000,000 / 80,000,000 \* $10000 = 9830, or $2666. To use this baud rate and select 'host' mode and 'full-speed', you could do 'WXPIN ##$E666,lowerpin'.

The upper (odd) pin is the DP pin. This pin's IN is raised whenever the output buffer empties, signalling that a new output byte can be written via WYPIN to the lower (even) pin. No WXPIN/WYPIN instructions are used for this pin.

The lower (even) pin is the DM pin. This pin's IN is raised whenever a change of status occurs in the receiver, at which point a RDPIN/RQPIN can be used on this pin to read the 16-bit status word. WXPIN is used on this pin to set the NCO baud rate.

These DP/DM electrical designations can actually be switched by swapping low-speed and full-speed modes, due to USB's mirrored line signalling.

To start USB, clear the DIR bits of the intended two pins and configure them each via WRPIN. Use WXPIN on the lower pin to set the mode and baud rate. Then, set the pins' DIR bits. You are now ready to read the receiver status via RDPIN/RQPIN and set output states and send packets via WYPIN, both on the lower pin.

To affect the line states or send a packet, use WYPIN on the lower pin. Here are its D values:

0 = output IDLE - default state, float pins, except possible resistor(s) to 3.3V or GND

1 = output SE0 - drive both DP and DM low

2 = output K - drive K state onto DP and DM (opposite)

3 = output J - drive J state onto DP and DM (opposite), like IDLE, but driven

4 = output EOP - output end-of-packet: SE0, SE0, J, then IDLE

$80 = SOP - output start-of-packet, then bytes, automatic EOP when buffer runs out

To send a packet, first do a 'WYPIN #$80,lowerpin'. Then, after each IN rise on the upper pin, do a 'WYPIN byte,lowerpin' to buffer the next byte. The transmitter will automatically send an EOP when you stop giving it bytes. To keep the output buffer from overflowing, you should always verify that the upper pin's IN was raised after each WYPIN, before issuing another WYPIN, even if you are just setting a state. The reason for this is that all output activity is timed to the baud generator and even state changes must wait for the next bit period before being implemented, at which time the output buffer empties.

There are separate state machines for transmitting and receiving. Only the baud generator is common between them. The transmitter was just described above. Below, the receiver is detailed. Note that the receiver receives not just input from another host/device, but all local output, as well.

At any time, a RDPIN/RQPIN can be executed on the lower pin to read the current 16-bit status of the receiver, with the error flag going into C. The lower pin's IN will be raised whenever a change occurs in the receiver's status. This will necessitate A WRPIN/WXPIN/WYPIN/RDPIN/AKPIN before IN can be raised again, to alert of the next change in status. The receiver's status bits are as follows:

[31:16] <unused> - $0000

[15:8] byte - last byte received

[7] byte toggle - cleared on SOP, toggled on each byte received

[6] error - cleared on SOP, set on bit-unstuff error, EOP SE0 > 3 bits, or SE1

[5] EOP in - cleared on SOP or 7+ bits of J or K, set on EOP

[4] SOP in - cleared on EOP or 7+ bits of J or K, set on SOP

[3] SE1 in (illegal) - cleared on !SE1, set on 1+ bits of SE1

[2] SE0 in (RESET) - cleared on !SE0, set on 1+ bits of SE0

[1] K in (RESUME) - cleared on !K, set on 7+ bits of K

[0] J in (IDLE) - cleared on !J, set on 7+ bits of J

The result of a RDPIN/RQPIN can be bit-tested for events of interest. It can also be shifted right by 8 bits to LSB-justify the last byte received and get the byte toggle bit into C, in order to determine if you have a new byte. Assume that 'flag' is initially zero:

SHR D,#8 WC 'get byte into D, get toggle bit into C

CMPX flag,#1 WZ 'compare toggle bit to flag, new byte if Z

IF\_Z XOR flag,#1 'if new byte, toggle flag

IF\_Z <use byte> 'if new byte, do something with it

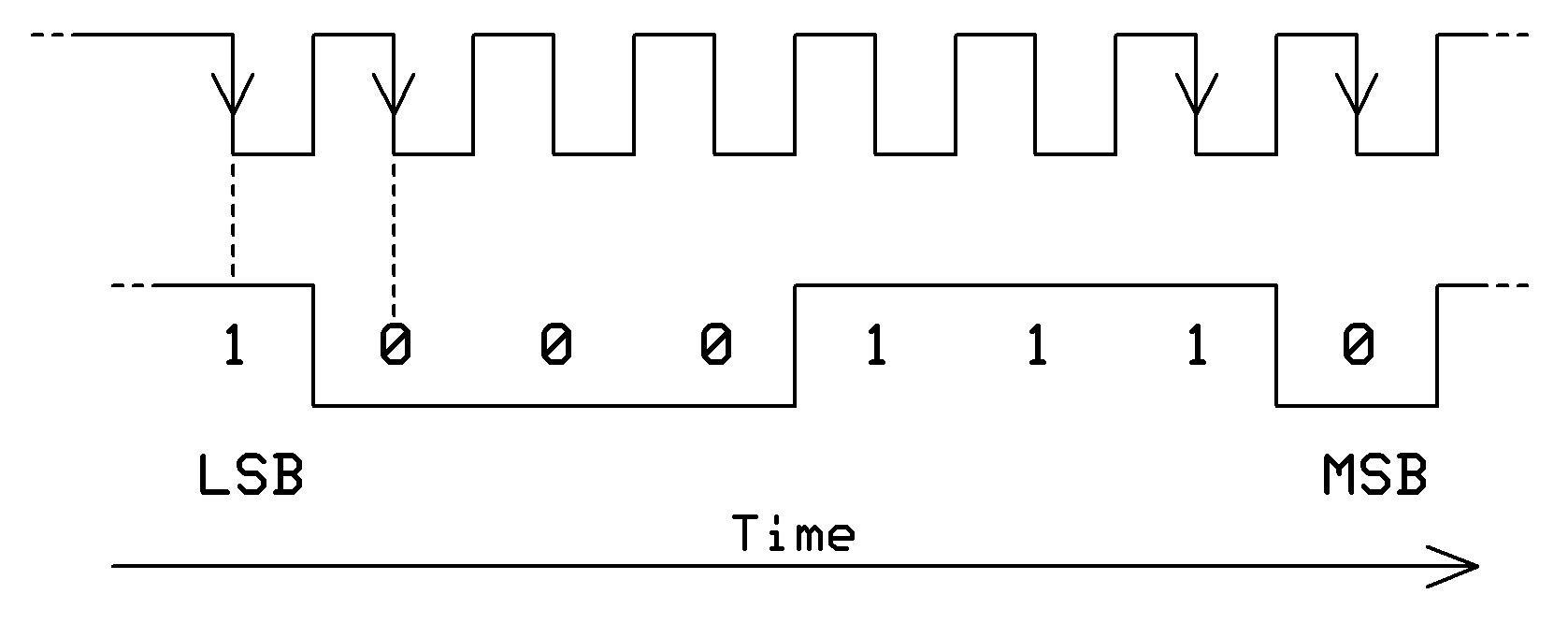
**--------------------------------------------------------------**

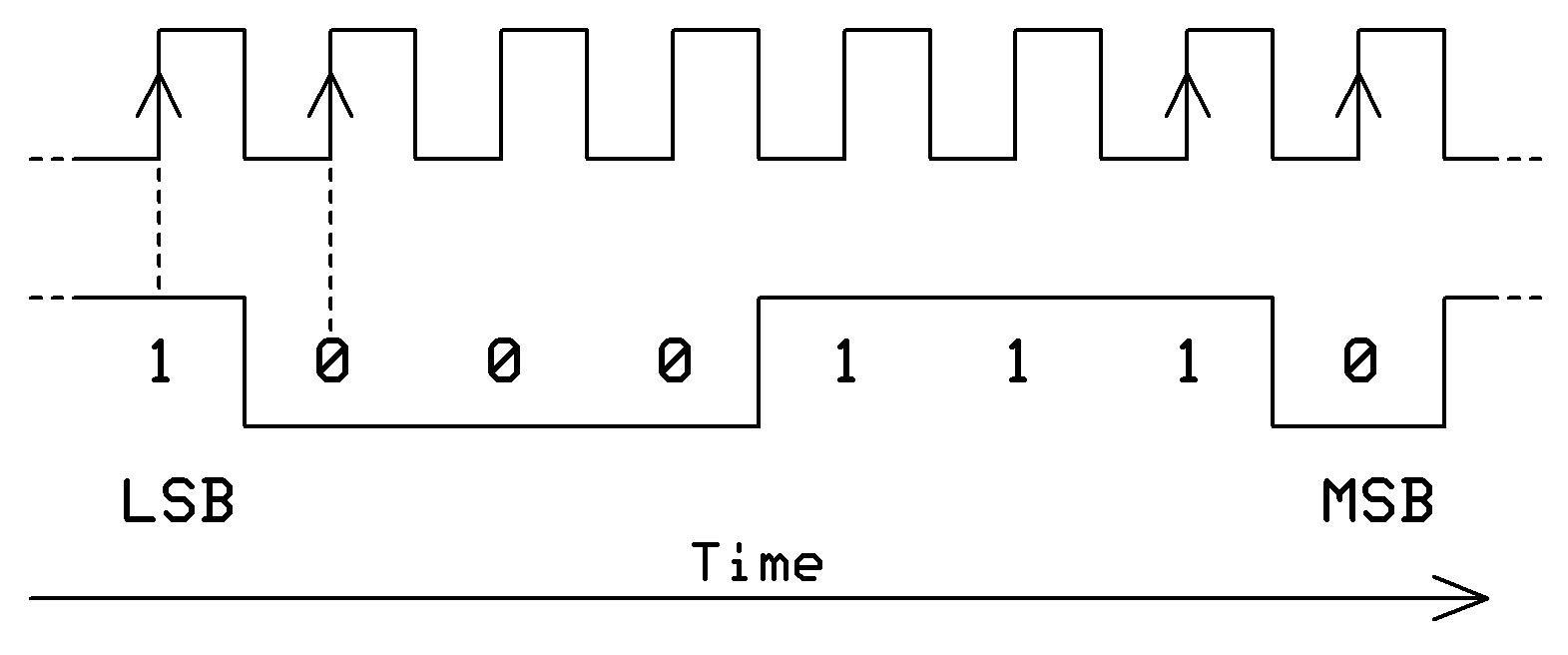
**%11100: Synchronous serial transmit (SST)**

Data from one bit to as many as 32 bits shift out a Smart Pin in this mode, synchronized with a separate clock signal. Setup involves configuring the A and B portions of a Smart Pin circuit and use of two physical pins. Bits shift out one at a time, starting with the LSB first.

The figures below shows timing for a synchronous serial transmission with positive-edge and negative-edge clocking. Software examples demonstrate both.

Serial synchronous transmission with negative-edge clocking:



Serial synchronous transmission with positive-edge clocking.

A WXPIN instruction determines how transmissions proceed and the number of data bits in a transmission. Register bit X[5] selects one of two transmission modes:

1. The *continuous* mode requires bit X[5] = 0. When a Smart Pin operates in this mode, during reset (DIR = 0) a WYPIN instruction puts the first set of data bits (DB1) in the shift register, or shifter, to prime it for transmission. Then, after you reset the Smart Pin (DIR = 1), your program may use another WYPIN instruction to load a second set of data bits (DB2) into the shifter's buffer.

Then the Smart Pin begins to continuously transmit the DB1 bits. After the shifter finishes a transmission, it moves the buffered data (DB2) bits into the shifter and sends it. When the buffer empties, the circuit raises the IN flag. Software can detect this buffer-empty condition and load the next group of bits into the buffer.

This mode allows steady data transmission with a continuously running clock, for as long as your software loads the shifter's buffer immediately after the IN flag becomes a logic 1, and before the current transmission ends.

2. The *start-stop* mode, X[5] = 1, lets software transmit data as needed, rather than continuously. You may use a WYPIN instruction that executes before the first clock signal occurs to change the data in the shifter. (Correct?) Use this mode if you might need to revise the newest data before a stream of clock pulses shift it out.

If a WYPIN instruction executes after a transmission starts, its data gets buffered and loaded into the shifter after the on-going transmission ends. Whenever the buffer empties, the Smart Pin raises the IN flag to indicate you can load the next set of bits into the buffer. During reset, data flows immediately through the buffer into the shifter.

Register bits X[4:0] set the number of bits to transmit, less one. To transmit eight bits, for example, set bits X[4:0] to $07. Synchronous communications do not include a start bit or a stop bit.

If you must send MSB-first data, your software needs to reverse the bit order. Say you have a byte in D[7:0] that you wanted to send MSB-first. First use an SHL D, #32-8 instruction to shift D[7:0] left into D[31:24]. Second, use a REV D instruction to swap the bits so D[31:0] becomes D[0:31]. Now you can shift the D data out MSB first.

During reset (DIR=0) the output is held low. At the end of the reset time, the Smart-Pin output will equal the state of the LSB of the data written by a WYPIN instruction while the pin was in its reset condition. This mode overrides OUT to control the pin output state.

**Example**

For a synchronous serial transmission with a **positive-edge clock:**

con

clkout = 20 ' Pin P20

txout = 21 ' Pin P21

dat org 0

wrpin sync\_tx\_mode, #txout 'Set up sync tx mode

wxpin #%1\_00111, #txout 'Set up stop/start mode

dirh #txout 'Enable Smart Pin

wrpin trans\_mode, #clkout 'Set pin P21 as clock

wxpin ##$1000, #clkout 'Set base period

dirh #clkout 'Enable P21 as clock-output

.loop waitx ##10\_000\_000 'Delay between transmissions

wypin #$85, #txout '8-bit data to transmit:

' 10100001 (LSB first)

wypin #16, #clkout 'Start clock, transmit data

jmp #.loop 'Loop, transmit again

sync\_tx\_mode long %0000\_1111\_000\_0000000000000\_01\_11100\_0 'Pos edge

trans\_mode long %0000\_0000\_000\_0000000000000\_01\_00101\_0 'Pos edge

'Courtesy of ozpropdev via Parallax Forum

For a **negative-edge clock**, substitute the following mode values. Note the highlighted bit changes:

sync\_tx\_mode long %0000\_0111\_000\_0000000000000\_01\_11100\_0 'Neg edge

trans\_mode long %0000\_0000\_000\_0000001000000\_01\_00101\_0 'Neg edge

**--------------------------------------------------------------**

**%11101: Synchronous serial receive (SSR)**

Data from one bit to as many as 32 bits shift into a Smart Pin in this mode, synchronized with an external clock signal sent by the transmitting device. Setup involves configuring the A and B portions of a Smart Pin circuit and use of two physical pins. Bits shift in one at a time, starting with the *LSB first*. The following bits shift in coincident with the positive or negative edge of the transmitter's clock. You will select the type of edge.

A WXPIN instruction determines how receptions proceed and the number of data bits in a in a received group. Register bit X[5] selects one of two reception modes:

X[5] = 0 selects the A input sample just before the B input edge was registered. This requires no hold time on the part of the sender.

X[5] = 1 selects the sample coincident with the B edge being registered. This is useful where transmitted data remains steady after the B edge for a brief time. In the synchronous serial transmit mode, the data is steady for two internal clocks after the B edge was registered, so employing this complementary feature would enable the fastest data transmission when receiving from another Smart Pin in synchronous serial transmit mode.

X[4:0] sets the number of bits, minus 1. For example, a value of seven sets the receiver for an 8-bit value.

When the receiver has shifted in all the required bits, it raises the IN flag. Software can detect the flag and use an RDPIN or RQPIN instruction to obtain the bits, left justified so bit D[31] = data MSB.

If the receiver shifts in LSB-first data, it will require right-shifting, so the LSB moves into bit D[0], unless the received data comprises 32 bits. For an 8-bit value, for example, your software would need the SHR D, #32 - numb\_of\_bits instruction to right-justify the LSB. Here, numb\_of\_bits must equal 8.

If data received from a transmitter arrives with the MSB first you must reverse the data and possibly mask-out unneeded bits, received data always comprises 32 bits, regardless of how many you actually transmitted. For example, if you received 8-bit data you would use an REV D instruction followed by a TRIML D, #8' instruction to LSB-justify the received data. That is, put the LSB into the right-most bit in a long value.

The following example includes code for a positive-edge clocked transmission and a positive-edge reception. It uses two Smart Pins, or four physical pins.

**Example**

' Synchronous serial transmission, Rev. 4, 06-15-2020 at 1450 MDT

'positive edge-trigger (Jon Titus)

CON

clkout = 40 ' Pin P40 'Xmtr clock out, P31

txout = 41 ' Pin P41 'Xmtr data out, to P30

rxin = 30 ' Pin P30 'Receiver data in

rxclkin = 31 ' Pin P31 'Receiver clock in

'Remember to connect these pins as noted!

dat

org 0

coginit id, ##@sync\_receive 'Initialize an available cog

' for sync receive

' Sync-serial transmit of data at set intervals.

dirl #txout 'Transmitter setup

wrpin sync\_tx\_mode, #txout 'Set sync tx mode pin 41

wxpin #%1\_00111, #txout 'Set up stop/start mode,

' 8 bits (7 + 1)

dirh #txout 'Enable xmtr output

dirl #clkout 'Clock output setup

wrpin clock\_mode, #clkout 'Set pin as transition-

'output mode

wxpin ##$1000, #clkout 'Set output base period

dirh #clkout 'Enable clock-output

.loop waitx ##10\_000\_000 'Delay

wypin ##$11, #txout '8-bit data to transmit

wypin #16, #clkout 'Start clock, transmit

jmp #.loop 'Continue to transmit

sync\_tx\_mode long

%0000\_1111\_000\_0000000000000\_01\_11100\_0 'Positive-edge clock mode

clock\_mode long

%0000\_0000\_000\_000000\_0\_000000\_01\_00101\_0 'Clock-mode

id long 16 'Need but, but not used

'====================================

' Sync-serial receive, data to 8 LEDs on P7-P0

' This software runs in a separate cog

org 0

sync\_receive mov dira, ##$00FF 'Pins P7--P0 set as outputs

dirl #rxin 'Reset receiver Smart Pin

wrpin sync\_rx\_mode, #rxin 'Set sync receiver mode

wxpin #%1\_00111, #rxin 'Set receiver to sample on B-

' input edge

dirh #rxin 'Enable Smart-Pin sync receiver

.test\_loop testp #rxin wc 'Wait in loop IN flag = set

nop

if\_nc jmp #.test\_loop 'If no C flag, repeat testing

rqpin rcvd\_data, #rxin 'Put 32-bit data (see text) in

' rcvd\_data

shr rcvd\_data, #24 'shift-right 24 bit places to

' align LSB at bit 0

mov outa, rcvd\_data 'Send 8 bit data to LEDs

jmp #.test\_loop 'Continue to rcv and display

' data

sync\_rx\_mode long %0000\_0001\_000\_0000000000000\_01\_11101\_0

rcvd\_data long $00

**--------------------------------------------------------------**

**%11110: Asynchronous serial transmit (AST)**

This mode lets a Smart Pin transmit as many as 32 bits at a programmable bit rate. Each transmission starts with a logic-0 "start" bit and ends with a logic-1 "stop" bit. The transmission does not include a parity bit, but you could calculate one and insert it in the data at MSB + 1. (Note 1.) If you insert a parity bit, remember to include it when you count the number of bits to transmit, but don't exceed 32 bits.

Use a WRPIN instruction to establish the pin mode:

tx\_pin = 20

txmode = %0000\_0000\_000\_0000000000000\_01\_11110\_0

wrpin ##txmode, #txpin

Next, a WXPIN instruction must set the bit rate (bits/sec) and the number of bits to transmit:

Register X bits X[31:16] set the number of system-clock periods in a bit period. In a case where your calculated bit rate leaves bits X[31:26] are all zero, bits X[15:10] let you set a base-2 fraction of a system-clock period to obtain an accurate bit-per-second transmission rate. First calculate the bit-period value:

system-clocks/bit = system-clock-frequency / bits-per-second

For a 100-MHz system clock and 115,200 bps:

100,000,000 / 115,200 = 868.1 system-clocks/bit

Then use one of the following methods to create the X[31:10] value:

a. ($system-clocks-per-bit \* $1\_0000) & $FFFFFC00

b. (($system-clocks-per-bit << 16 ) & $FFFFFC00)

For the 115,200 bps transmission example above, at this point:

X[31:0] = $3640\_C000

The X-register bits X[4:0] set the number of bits to transmit, *minus 1*. A value of *seven*, for example, set the transmission size to *eight bits*. Note: This number *does not* include the start bit or the stop bit.

bit\_period = $3640\_C00**7**

wxpin ##bit\_period, #txpin

To transmit, load data into the Smart Pin's Y register. Eight-bit data would go into bits Y[7:0]; 10-bit data, for example, would go into bits Y[9:0]. Your data first goes into a single-stage buffer before it advances to a shift register, or "shifter," for output. This buffering arrangement makes it possible to keep the "shifter" constantly busy, so no gaps exist between data transmissions. Whenever data moves from the buffer to the shifter, the Smart Pin raises its IN flag to indicate software may load new data for transmission.

The example below uses a TESTP instruction to check the buffer for an empty condition. A logic-1 in the resulting carry flag (C) signifies "buffer empty," so you may load the next data to send. You could use an RDPIN or RQPIN instruction instead and again monitor C (carry) bit. During reset (DIR=0) the pin output is held at a logic-1. This mode overrides OUT to control the pin's output state.

**Note 1**: You can obtain parity information for a n-bit value by using an OR instruction with a logic-1 for each bit in your data. Assume my\_data holds 10 bits:

parity = 0

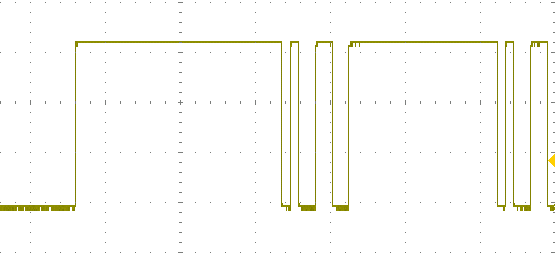
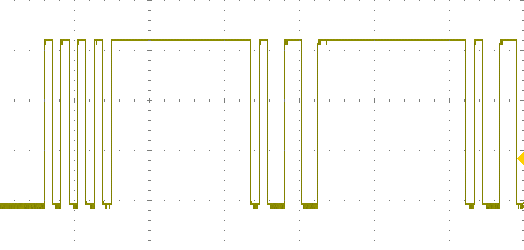
my\_data = $3A2

OR my\_data, #$1111111111 wc

if\_c mov parity, #1

The Carry flag indicates the parity. How does Carry indicate parity? 1 = even?, 0 = odd?

**Note 2**: The Smart Pin produced a logic-0 output prior to its first transmission (below left), which eliminates the logic-0 start bit. To avoid that condition and start with a Smart Pin logic-1 output (below right), send data with all 1's, such as $FF to clear the shifter. The following transmissions will all start from a logic-1 shifter. The following transmissions will all start from a logic-1 transmissions.



Example

'Async Transmit Ver 2, 6-10-2020 at 1440H MDT

CON

' Constants for serial-port control

txpin = 20 ' P20 serial out

bitper = $3640\_0007 ' bit rate for tests

txmode = %0000\_0000\_000\_0000000000000\_01\_11110\_0 'async tx mode

DAT

wrpin ##txmode, #txpin ' Smart Pin async transmit mode

wxpin ##bitper, #txpin ' transmit period for 8 bits

dirh #txpin ' enable Smart Pin tx

nop

wypin #$55, #txpin ' transmit test %01010101

nop

.flag\_test testp #txpin wc ' wait for empty buffer

if\_nc jmp #.flag\_test ' if not empty, test again

waitx ##25\_000\_000 / 70 ' delay inserted for testing

wypin #$99, #txpin ' send testing byte to tx pin

jmp #.flag\_test ' transmit "forever"

'Thanks to Ray Rodrick, *et al*. for this software.

**--------------------------------------------------------------**

**%11111: Asynchronous serial receive (ASR)**

A Smart Pin will receive serial data with a length from 1 to as many as 32 bits at a preset bit/sec rate equal to that rate at a transmitting device.

Use a WRPIN instruction to establish the pin mode:

rx\_pin = 21

bitper = $3640\_0007 ' bit rate for tests

rxmode = %0000\_0000\_000\_0000000000000\_00\_11110\_0 'rcvr mode

wrpin ##rxmode, #rxpin

Next, a WXPIN instruction sets the bit rate and the number of bits to receive:

Bits X[31:16] set the number of system-clock periods in a bit period. In a case where bits X[31:26] are all zero, bits X[15:10] let you set a fraction of a system-clock period to obtain an accurate bit-transmission rate. First calculate the bit-period value:

system-clocks/bit = system-clock-frequency / bits-per-second

For a 100-MHz system clock and 115,200 bps:

100,000,000 / 115,200 = 868.1 system-clocks/bit

Then use one of the following methods to create the X[31:10] value:

a. ($system-clocks-per-bit \* $1\_0000) & $FFFFFC00

b. (($system-clocks-per-bit << 16 ) & $FFFFFC00) + ($numb\_of\_bits - 1)

For the 115,200 bps transmission example above, at this point:

X[31:0] = $3640\_C000

The X-register bits X[4:0] set the number of bits to receive, *minus 1*. A value of seven, for example, set the transmission size to eight bits. Note: This number does not include the start bit or the stop bit.

bit\_period = $3640\_C00**7**

wxpin ##bit\_period, #txpin

Software will test the receiver pin and write the IN flag to the carry flag (C). When detected, the program exits the test loop and moves the received data into a variable.

Received data arrives justified to the left, so an 8-bit value would occupy rcvd \_data bits D[31:24]. You can shift them to bits D[7:0] with the instruction:

shr rcvd\_data, #32 - numb\_bits

**Example**

CON

numb\_bits = 8

rx\_pin = 21 ' P21 serial input

rxmode = %0000\_0000\_000\_0000000000000\_00\_11110\_0 ' async rcvr mode

bitper = $3640\_0007 ' bit rate & bits

DAT

wrpin ##rxmode, #rxpin ' Smart Pin async receive mode

wxpin ##\_bitper, #\_rxpin ' receive period for 8 bits

dirh #\_rxpin ' enable Smart Pin rcvr

nop

.rcvr testp #rxpin wc ' test flag, wait for data

if\_nc jmp #.rcvr ' no flag, test again

rdpin rcvd\_data, #rxpin ' data rcvd, save it

shr rcvd\_data, #32 - numb\_bits 'right-justify data

rcvd\_data long 0 ' data saved here

-----end-----