PropBASIC Syntax Guide

Version 0.15

March 15, 2010

Hitt Consulting

WARRANTY

PropBASIC is provided at no charge and ***without any warranty***, expressed or implied, for merchantability or fitness for any purpose. Use is at your own risk.

COPYRIGHTS AND TRADEMARKS

DISCLAIMER OF LIABILITY

Neither Hitt Consulting nor Parallax, Inc. is responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, or any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Neither Hitt Consulting nor Parallax, Inc. is responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your Propeller microcontroller application, no matter how life-threatening it may be.

CREDITS

PropBASIC Compiler Design/Implementation: *Terry Hitt (Hitt Consulting)*
Documentation Contributors: *Terry Hitt, Parallax, Inc.*

Material from Parallax, Inc. is used with permission.

Contents

About PropBASIC

PropBASIC is a BASIC language compiler for the Propeller (P8X32A) microcontroller from Parallax, Inc. PropBASIC was designed to meet specific goals:

- ✔ Expedite the task of the professional engineer by creating a simple, familiar, yet robust high-level language for the Propeller microcontroller. This allows Propeller-based projects to be prototyped and coded quickly without having to learn to program in Spin or PASM.
- ✔ Assist the student programmer wishing to make the transition from pure high-level programming to low-level programming (Propeller Assembly language [PASM]).

PropBASIC is a non-optimizing, inline compiler. What this means is that each BASIC language statement is converted to a block of assembly code in-line at the program location; no attempt is made to remove redundant instructions that would optimize code space. This allows the advanced programmer to modify code as required for specific projects and, perhaps more importantly, provides an opportunity for the student to learn Propeller Assembly language techniques by viewing a 1-for-1 (from BASIC to Assembly language) output.

Conventions Used in this Document

In syntax descriptions, curly braces { } are used to indicate optional items. For example:

```
PULSIN Pin, State, Variable {, Timeout}
```

In this case, the parameter for Timeout is optional.

In syntax descriptions, brackets [ ] indicate that the parameter must be one of the presented items (separated with the pipe | character). For example:

```
DO {[WHILE | UNTIL] Condition}
  Statement(s)
LOOP
```

In this case, the use of *Condition* with `DO` requires `WHILE` or `UNTIL`

Example code is presented on a tinted background:

```
SUB FLASH_LED
  DO WHILE Alarm = IsActive
    TOGGLE AlarmLed
    DELAY_MS 250
  LOOP
  AlarmLED = IsOff
  ENDSUB
```

Directives

Directives are used to configure the PropBASIC program.

```
DEVICE P8X32A, {OscType {, PLL}}
```

The DEVICE directive specifies the hardware device type (P8X32A), oscillator type, and PLL configuration.

In the (minimal) configuration that follows the oscillator type is assumed to be RCFAST and a PLL setting of PLL1X; the effective frequency is assumed to be 12 MHz:

```
DEVICE          P8X32A
```

In this very typical configuration the oscillator type is a 5 MHz crystal and a PLL setting 16x for an effective frequency of 80 MHz.

```
DEVICE          P8X32A, XTAL1, PLL16X
XIN             5_000_000
```

Note that when a crystal oscillator type is specified the XIN (recommended) or FREQ directive must also be used.

Oscillator Type and PLL Settings			
Setting	XO Resistance	XI / XO Capacitance	Description
RCFAST	Infinite	n/a	Internal fast oscillator (~12 MHz) [1]
RCSLOW	Infinite	n/a	Internal slow oscillator (~20 kHz) [1]
XINPUT	Infinite	6 pF	External oscillator (DC to 80 MHz); XIN pin only
XTAL1	2 kΩ	36 pF	External low-speed crystal (4- to 16 MHz)
XTAL2	1 kΩ	26 pF	External medium-speed crystal (8- to 32 MHz)
XTAL3	500 Ω	16 pF	External high-speed crystal (20- to 80 MHz)
PLL1X	n/a	n/a	Multiply external frequency by 1
PLL2X	n/a	n/a	Multiply external frequency by 2
PLL4X	n/a	n/a	Multiply external frequency by 4
PLL8X	n/a	n/a	Multiply external frequency by 8
PLL16X	n/a	n/a	Multiply external frequency by 16

[1] RC modes are not recommended for programs that require accurate timing or use instructions that rely on accurate timing (e.g., SEROUT, SERIN).

```
XIN Frequency
```

The **XIN** directive specifies the hardware input frequency (pre PLL multiplier) when an external crystal or crystal-oscillator is used. The "standard" Propeller crystal setting is five megahertz (5 MHz).

```
XIN             5_000_000
```

The **XIN** setting will be multiplied by the PLL setting to determine the operating frequency of the PropBASIC program. This value is used by the compiler for calculating delays in time-sensitive instructions (e.g., **PAUSE**, **SERIN**, **SEROUT**).

```
FREQ Frequency
```

The **FREQ** directive specifies the operating frequency (post PLL multiplier) of the PropBASIC program. This value is used by the compiler for calculating delays in time-sensitive instructions (e.g., **PAUSE**, **SERIN**, **SEROUT**) and should, therefore, be the product of the external input frequency and the PLL setting. *An incorrect* **FREQ** *setting may allow the PropBASIC program to compile but not operate as intended hence the use of* **XIN** *instead of* **FREQ** *is recommended.*

> Note that when **FREQ** is used with **RCFAST** and **RCSLOW** modes the assumed frequency (12 MHz for **RCFAST**, 20 kHz for **RCSLOW**) is overridden with the **FREQ** setting. *This should only be used after the actual internal frequency of a particular Propeller chip has been empirically measured.*


```
{Label}         DATA    Const0 {, Const1 {, Const2...}}
{Label}         WDATA   Const0 {, Const1 {, Const2...}}
{Label}         LDATA   Const0 {, Const1 {, Const2...}}
```

The **DATA** directives allow the programmer to create tables of a defined type (byte, word, or long) in the Hub RAM space. Using **DATA**, **WDATA**, or **LDATA** is a convenient way to store output patterns and text messages, and to share information between cogs. A table can be written to, if desired, using **WRxxxx**, and read from using **RDxxxx**.

```
PROGRAM Label
```

The **PROGRAM** directive sets the execution start point (at *Label*) for the PropBASIC program. Note that the **PROGRAM** directive should be placed immediately before the *Label* that defines the beginning of the user program. Auto-generated start-up code will be inserted between the **PROGRAM** directive and *Label*.

```
{Label}          FILE     "filename.ext"
```

The `FILE` directive is used to insert external [byte] data (stored in *filename.ext*) at the current location, usually as named (using *Label*) data

```
LOAD "filename.ext"
```

The `LOAD` directive is used to insert a PropBASIC source code file at the current location.

```
INCLUDE "filename.ext"
```

The `INCLUDE` directive is used to insert a Propeller Assembly code file at the current location.

Conditional Compilation

PropBASIC supports several conditional compilation directives that allow the programmer to adjust the program without major editing/recoding. Conditional compilation directives are only evaluated at compile time.

`'{$DEFINE Symbol}`

Defines a conditional-compilation symbol that could, for example, be evaluated as **True** when using `$IFDEF` (see below).

`'{$UNDEFINE Symbol}`

Removes a conditional-compilation symbol that could, for example, be evaluated as **False** when using `$IFDEF` (see below).

`'{$IFDEF Symbol}`

Evaluates as **True** if *Symbol* has been defined, allowing a specific section to be executed that corresponds to the presence of *Symbol*.

`'{$IFNDEF Symbol}`

Evaluates as **True** if *Symbol* has not been defined, or has been undefined, allowing a specific section to be executed that corresponds to the absence of *Symbol*.

`'{$ELSE}`

Allows for an alternate set of code to run when `$IFxxxx` statement evaluates as **False**.

`'{$ENDIF}`

Terminates a compound `$IFxxxx..$ELSE` structure

`'{$IFFREQ [= | <> | > | < | >= | <=] Value}`

Allows the program to evaluate the `FREQ` setting of the program

`'{$ERROR Message}`

Inserts an error message in the compiled output listing and the termination of the compilation process.

`'{$WARNING Message}`

Inserts a warning message into the compiled output listing; this directive does not stop the compilation process.

IO Pins

PropBASIC IO pins and pin groups are defined using the **PIN** declaration.

```
PinName         PIN     # {[INPUT | OUTPUT | LOW | HIGH]}
GroupName       PIN     MSB#..LSB# {[INPUT | OUTPUT | LOW | HIGH]}
```

The minimal requirement for a pin definition is the pin's symbolic name, the **PIN** declaration, and the pin number, 0 to 31. Special consideration should be given to pins 31 and 30 as these serve as the Propeller's programming port, as well as pins 29 and 28 as these serve as the Propeller's I2C pins for the boot EEPROM. Use caution if any of these pins are required by the program.

```
GreenLed        PIN     0
```

The above definition names pin P0 to 'GreenLED.' When no option is specified the pin is assumed an **INPUT**. The programmer may specify an output mode with **OUTPUT**, **LOW**, or **HIGH**. The **LOW** and **HIGH** options modify the **OUTA** register as well as the **DIRA** register for the pin.

```
LEDS            PIN     23..16 LOW        ' make outputs and low
```


In the above example pins 23..16 are set to output mode and low. Note that the use of a pin group allows the programmer to write a value to, or read a value from, that group of pins without concern for the actual physical connections; this simplifies code changes to accommodate hardware modifications. The order of the pin numbers used is important; the first pin will receive the MSB of the value written to a pin group, the second will receive the LSB. The order applies to reading pin groups as well.

> Note that pin modifiers only apply to the cog in which they are defined. If you define a pin in the main program it may be used in other cogs (tasks), but the task must set the pin to output mode if that's how the task needs to use the pin. Use caution, though, when making a pin an output in two separate cogs as one cog can affect the pin despite what the other is doing (e.g., make the pin high).

PropBASIC allows the programmer to specify how a pin definition is used. For example:

```
TestPin         PIN     3
```

To read the current state of *TestPin* the following syntax is used:

```
  result = TestPin
```

To treat *TestPin* as an absolute value (i.e., 3) use the following syntax (this is used by the compiler when passing a pin name to a subroutine or function):

```
thePin = #TestPin
```

To treat *TestPin* as a mask value use this syntax:

```
testMask = @TestPin
```

After the above line *testMask* will hold %1000.

Constants

PropBASIC constants are defined using the **CON** declaration.

```
Symbol          CON      Value
```

Examples:

```
RoomTemp        CON      72
MaxEEPROM       CON      $7FFF
PinMask         CON      %00000000_00000000_00000000_00001000
qBits           CON      %%0123
```

Values may be specified in decimal (no prefix), hexadecimal ($), binary notation (%), or quaternary (%%) notation with the underscore character used, if desired, as a separator. The legal range for numeric constants is **NEGX** (-2,147,483,648) to **POSX** (2,147,483,647).

Single-character alpha constants may also be defined; for example:

```
First           CON      "A"
Last            CON      "Z"
```

Baudmode constants for **SERIN** and **SEROUT** appear as a string, enclosed in quotes:

```
Baud            CON      "T115200"
```

In the above example *Baud* is defined at True mode at 115.2K baud.

Variables

PropBASIC supports two variable types: **HUB** variables, which are stored in the Propeller's hub RAM and may be shared between cogs, and local variables which are only available within the cog in which they are defined (e.g, the main program or a task).

Hub variables may be bytes, words, or longs and are defined with the **HUB** declaration:

```
Symbol          HUB     VarType{(Elements)} {= Value}
```

Example: a hub-based long variable:

```
bufhead         HUB     Long
```

Example: a hub-based byte array:

```
buffer          HUB     Byte(16)  = 0
```

Note: Hub variables can only be accessed with **RDBYTE**, **WRBYTE**, **RDWORD**, **WRWORD**, **RDLONG**, and **WRLONG**.

Local variables within a cog or task are defined using the **VAR** declaration.

```
Symbol          VAR     Long{(Elements)} {= Value}
```

As PropBASIC is compiled to PASM, the only variable type supported is Long.

```
idx             VAR     Long
```

Note that PropBASIC does not pre-initialize variables to any value unless specifically directed by the programmer. For example:

```
idx             VAR     Long = 0
```

Operators

PropBASIC includes the following unary and binary operators.

Note: Only one operator per line of code is allowed.

Unary Operators		
Operator	Alternate	Description
ABS		Returns the absolute value
SGN		Returns the sign of a value: 1, 0, -1

Binary Operators		
Operator	Alternate	Description
+		Addition
-		Subtraction
/		Division
//		Remainder of a division
*		Multiplication (returns lower 32 bits of 64-bit product)
*/		Multiply middle (returns middle 32 bits of 64-bit product)
**		Multiply high (returns high 32 bits of 64-bit product)
&	AND [1]	Bitwise AND
|	OR [1]	Bitwise OR
^	XOR	Bitwise XOR
&~	ANDN	Bitwise AND-NOT
MIN		Return minimum of two values
MAX		Return maximum of two values
<<	SHL	Shift left
>>	SHR	Shift right

[1] May be used as logical operator in compound `IF..THEN` block.

PropBASIC Aliases

PropBASIC creates and uses the following symbols.

PropBASIC Aliases		
Alias	Alternate	Description
`__InitDirA`		Initial `dira` settings (based on `PIN` definitions/options)
`__InitOutA`		Initial `outa` settings (based on `PIN` definitions/options)
`_FREQ`		System frequency in Hertz
`__temp1`	`__remainder`	Used in Assembly code generated by PropBASIC
`__temp2`		
`__temp3`		
`__temp4`		
`__temp5`		
`__remainder`	`__temp1`	Remainder of a division
`__param1`		Parameter passed to SUB or FUNC, or returned from FUNC
`__param2`		
`__param3`		
`__param4` [1]		
`__paramcnt`		Number of parameters passed to a SUB or FUNC

[1] Parameters *__param1* through *__param4* are always created; *__param5* up to *__param20* are optionally created based on subroutine and function declarations.

Propeller Aliases

The following Propeller symbols may be used in a PropBASIC program.

Propeller Aliases		
Alias	R/W	Description
DIRA	R/W	Direction Register for 32-bit port A
DIRB	R/W	Direction Register for 32-bit port B (future use)
INA	R	Input Register for 32-bit port A
INB	R	Input Register for 32-bit port B (future use)
OUTA	R/W	Output Register for 32-bit port A
OUTB	R/W	Output Register for 32-bit port B (future use)
CNT	R	32-bit System Counter Register
CTRA	R/W	Counter A Control Register
CTRB	R/W	Counter B Control Register
FRQA	R/W	Counter A Frequency Register
FRQB	R/W	Counter B Frequency Register
PHSA	R/W	Counter A Phase-Locked Loop (PLL) Register
PHSB	R/W	Counter B Phase-Locked Loop (PLL) Register
VCFG	R/W	Video Configuration Register
VCL	R/W	Video Scale Register
PAR	R	Cog Boot Parameter Register

Subroutines and Functions

Subroutines and functions allow the programmer to improve program readability and save code space by incorporating frequently-used code blocks that may be called with a custom name. For example, the PAUSE 1 instruction generates the following Assembly code:

```
                mov          __temp1,cnt
                adds         __temp1,_1mSec
                mov          __temp2,#1
__L0001
                waitcnt      __temp1,_1mSec
                djnz         __temp2,#__L0001
```

As PropBASIC is a single-pass, non-optimizing compiler this code (with the appropriate change for the duration) will be generated for each use of PAUSE, potentially consuming valuable code space within the cog. This space can be saved by encapsulating PAUSE in a custom subroutine and calling that. The working code for PAUSE will be compiled just once – within the body of the subroutine – saving precious code space in the cog.

Using Subroutines

```
Name            SUB     {MinParams {, MaxParams}}
```

- ✔ ***Name*** is is the name of the subroutine; this cannot be a reserved word.
- ✔ ***MinParams*** is the minimum number of [long] parameters that must be passed to the subroutine.
- ✔ **MaxParams** is the maximum number of [long] parameters that can be passed to the subroutine.

For a subroutine that handles PAUSE you might use the following declaration:

```
DELAY_MS        SUB     1
```

This declaration tells us that the **DELAY_MS** subroutine requires one parameter which, in this case, will be the delay in milliseconds.

The working code for a subroutine will typically appear at the end of the program listing (see *Anatomy of a PropBASIC Program*) and will be enclosed in a SUB..ENDSUB block. For example:

```
SUB DELAY_MS
  PAUSE __param1
  ENDSUB
```

In the course of the program any PAUSE statements can no be replaced with **DELAY_MS**. The delay value is passed to the subroutine in *__param1* which gets used by the subroutine code.

Using Functions

```
Name            FUNC    {MinParams {, MaxParams}}
```

- ✔ ***Name*** is is the name of the subroutine; this cannot be a reserved word.
- ✔ ***MinParams*** is the minimum number of [long] parameters that must be passed to the subroutine.
- ✔ **MaxParams** is the maximum number of [long] parameters that can be passed to the subroutine.

Functions are very similar to subroutines in that they encapsulate frequently-used code to save program space. The difference is that functions are expected to return one or more parameters, even if no parameters are passed to the function. For example, one might write a function that monitors a temperature sensor and use that function like this:

```
  currentTemp = READ_TEMP
```

Values returned by a function are passed in the *__paramx* variables, typically *__param1*, but if two or more parameters are returned then other variables will be used.

The PropBASIC programmer may consider building a library of commonly used functions. Multiplication and division, for example, generate a significant amount of Assembly code and should be encapsulated in custom functions for most applications. Start with the function declarations:


```
MULT            FUNC    2
DIV             FUNC    2
```

This declaration indicates that two parameters must be passed to the function called **MULT**. And now the working code:

```
FUNC MULT
  __param1 = __param1 * __param2
  RETURN __param1
  ENDFUNC
```

`RETURN` is used to load the *__paramx* variable(s) to pass information back to the calling code. To return multiple values they are separated by a comma within the `RETURN` statement. For example:

```
FUNC DIV
  __param1 = __param1 / __param2
  RETURN __param1, __remainder
```

When two or more parameters are returned by a function the programmer must retrieve *__param2* and higher manually, as shown below.

```
  wholeParts = DIV x, y                  ' wholeParts = x / y
  leftOver = __param2                    ' leftover = __remainder
```

It is important that additional parameters be captured before another subroutine or function is called that could overwrite the returned value.

Using Variable Parameters

Subroutines and functions can make use of a variable parameter count. For example, one might create a subroutine that uses `SEROUT` to transmit a single character, or multiple copies of that character when desired. The subroutine declaration would look something like this:

```
TX_BYTE         SUB     1, 2
```

The declaration indicates that **`TX_BYTE`** requires at least one parameter and will work with up to two. PropBASIC passes the number of parameters used in a subroutine or function call in *__paramcnt*. This can be used in **`TX_BYTE`** as follows:

```
SUB TX_BYTE
  IF __paramcnt = 1 THEN
    __param2 = 1
  ELSE
    __param2 = __param2 MIN 1
  ENDIF
  DO WHILE __param2 > 0
    SEROUT TX, Baud, __param1
    DEC __param2
  LOOP
  ENDSUB
```

To transmit a single character the **`TX_BYTE`** subroutine is called as follows:

```
  TX_BYTE "*"
```

In this case *__paramcnt* will be set to one before the call which will cause the subroutine to load one into *__param2*, which is then used in `DO..LOOP` to control how many times the character (passed in *__param1*) is transmitted.

To create a line of 10 stars call **`TX_BYTE`** like this:

```
  TX_BYTE "*", 10
```

Tasks

The Anatomy of a PropBASIC Program

Like most programming languages, PropBASIC is very flexible and there are infinite *correct* ways to write any given program. That stated, it is in the programmer's interest to use a clean, logical structure when writing PropBASIC applications. The template that follows provides such a structure.

```
' =========================================================================
' File...... template.pbas
' Purpose...
' Author....
' Email.....
' Started...
' Updated...
' =========================================================================


' -------------------------------------------------------------------------
' Device Settings
' -------------------------------------------------------------------------

DEVICE          P8X32A, XTAL1, PLL16X
XIN             5_000_000


' -------------------------------------------------------------------------
' Conditional Compilation Symbols
' -------------------------------------------------------------------------


' -------------------------------------------------------------------------
' Constants
' -------------------------------------------------------------------------


' -------------------------------------------------------------------------
' IO Pins
' -------------------------------------------------------------------------


' -------------------------------------------------------------------------
' Shared (hub) Variables (Byte, Word, Long) - use RDxxxx/WRxxxx
' -------------------------------------------------------------------------


' -------------------------------------------------------------------------
' User Data (DATA, WDATA, LDATA, FILE) - use RDxxxx/WRxxxx
' -------------------------------------------------------------------------
```

```
' ----------------------------------------------------------------------
' TASK Definitions
' ----------------------------------------------------------------------


' ----------------------------------------------------------------------
' Local Variables (Long only)
' ----------------------------------------------------------------------


' ----------------------------------------------------------------------
' SUB and FUNC Definitions
' ----------------------------------------------------------------------


' ----------------------------------------------------------------------
  PROGRAM Start
' ----------------------------------------------------------------------

Start:
  ' setup code


Main:

  ' program code

  GOTO Main
  END


' ----------------------------------------------------------------------
' SUB and FUNC Code
' ----------------------------------------------------------------------


' ----------------------------------------------------------------------
' TASK Code
' ----------------------------------------------------------------------
```

ASM..ENDASM, \

```
ASM
  PASM instructions
ENDASM

\ PASM instruction
```

Function

ASM allows the insertion a block of Propeller Assembly language (PASM) statements into the PropBASIC program. The PASM block is terminated with **ENDASM**. Code in the **ASM..ENDASM** block is inserted into the program verbatim. A single line of Propeller Assembly code may be inserted by prefixing the line with ****.

Explanation

Certain time-critical routines are best coded in straight assembly language, and while the \ symbol allows the programmer to insert a single line of assembly code, it is not convenient for large blocks.

The following program toggles an LED on P16 every 125 milliseconds (1/8 second).

```
DEVICE          P8X32A, XTAL1, PLL16X
XIN             5_000_000

LED             PIN     16  OUTPUT       ' make LED an output

tic             VAR     Long
delay           VAR     Long


PROGRAM Start

Start:
  ASM
    rdlong      tic, #0                  ' read system frequency
    shr         tic, #3                  ' divide by 8
    mov         delay, cnt               ' get system counter
    add         delay, tic               ' add tic timing

Main
    xor         outa, LED                ' toggle LED pin
    waitcnt     delay, tic               ' wait one tic, reload
    jmp         #Main                    ' repeat
  ENDASM
```

Note: Program labels within the **ASM..ENDASM** block do not use the terminating colon as with PropBASIC labels (see the label, *Main*, above).

BRANCH

`BRANCH Offset, Label0 {, Label1, Label2, ...}`

Function

Jump to the program *Label* specified by *Offset*. Note that the value of *Offset* should not be greater than the number of labels minus one, otherwise the BRANCH instruction will be skipped.

- ✔ ***Offset*** is simple variable or array element.
- ✔ ***Label*** is a valid program label that is followed by operational code.

Explanation

The BRANCH instruction is useful when you want to write something like this:

```
Check_Value:
  IF value = 0 THEN Case_0                ' if value is 0, jump to Case_0
  IF value = 1 THEN Case_1                ' if value is 1, jump to Case_1
  IF value = 2 THEN Case_2                ' if value is 2, jump to Case_2

No_Match:
```

The above code is simplified with BRANCH as follows:

```
Check_Value:
  BRANCH value, Case_0, Case_1, Case_2

No_Match:
```


Related instructions: ON..GOTO, IF..THEN

COGID

`COGID Variable`

Function

Moves the ID of the cog, 0 to 7, to *Variable*.

Related instructions: `COGINIT`, `COGSTART`, `COGSTOP`

COGINIT

`COGINIT TaskName, CogNum`

Function

Starts the task defined by *TaskName* in the cog specified by *CogNum*.

✔ ***TaskName*** is the name of the task code to be launched into a new cog

✔ ***CogNum*** is the cog ID, 0 to 7, of the target cog.

Related instructions: `COGID`, `COGSTART`, `COGSTOP`

COGSTART

`COGSTART TaskName {, Variable}`

Function

Starts the task defined by *TaskName* in a new cog (if one is available).

- ✔ ***TaskName*** is the name of the task code to be launched into a new cog
- ✔ ***Variable*** holds the ID, 0 to 7, of the newly-launched cog. If no cog was available then `COGSTART` will return 8 in *Variable.*

Related instructions: **`COGID`**, **`COGINIT`**, **`COGSTOP`**

COGSTOP

`COGSTOP CogNum`

Function

Stops a cog.

✔ ***CogNum*** is a variable or constant value, 0 to 7, which specifies the cog to stop.

Explanation

A cog can be started by a PropBASIC program using `COGINIT` or `COGSTART`. Should the programmer wish to stop a previously-launched cog the `COGSTOP` instruction will do this. The ID of the cog to stop, 0 to 7, must be provided.

Note: The main PropBASIC program runs in cog 0.

Related instructions: `COGID`, `COGINIT`, `COGSTART`

COUNTERA, COUNTERB

```
COUNTERx Mode {, APin {, BPin {, FRQx, {, PHSx}}}}
```

DEC

`DEC Variable {, Delta}`

Function

Decrement (decrease) the value of *Variable.*

✔ ***Variable*** is simple variable or array element.

✔ ***Delta*** is the value to subtract from *Variable*. If not specified, *Delta* is set to one.

Explanation

DEC is a short-form version of:

```
  Variable = Variable - Delta
```

The **DEC** instruction subtracts *Delta* from *Variable*. If *Delta* is not specified it will be set to one (1). Signed operators are used, so subtracting a negative *Delta* has he same effect as adding a positive *Delta*.

```
Main:
  result = 4
  DEC result                              ' result is now 3
  DEC result, -2                          ' result is now 5
  result = 1
  DEC result, 2                           ' result is now -1 ($FFFF_FFFF)
```

Related instructions: **DJNZ**, **INC**

DJNZ

`DJNZ Variable, Label`

Function

Decrement (decrease) value of *Variable* by one and jump to *Label* if *Variable* is not equal to zero.

- ✔ ***Variable*** is simple variable or array element.
- ✔ ***Label*** is a program label that is followed by operational code.

Explanation

The **DJNZ** instruction decrements *Variable* (decreases by one) and if the result of that operation is not zero the program will jump to the location specified by *Label*.

```
Start:
  flashes = 5

Main:
  HIGH AlarmLed
  DELAY_MS 100
  LOW AlarmLed
  PAUSE 400
  DJNZ flashes, Main                    ' loop until flashes = 0
  DELAY_MS 2_000
  GOTO Start
```

Related instruction: **DEC**

DO..LOOP

```
DO {[WHILE | UNTIL] Condition}
  Statement(s)
LOOP

DO
  Statement(s)
LOOP {[UNTIL | WHILE] Condition}

DO
  Statement(s)
LOOP Variable
```

Function

Create a repeating loop that executes the program lines between `DO` and `LOOP`, optionally testing before or after the loop statements.

- ✔ ***Condition*** is a simple statement, such as `idx = 7` that can be evaluated as **True** or **False**. Only one comparison operator is allowed (see `IF..THEN` for valid condition operators).
- ✔ ***Statement*** is any valid PropBASIC statement.
- ✔ ***Variable*** is a simple variable or array element.

Explanation

The `DO..LOOP` structure allows your program execute a series of instructions indefinitely, or until a specified condition terminates the loop. The simplest form is shown here:

```
Alarm_On:
  DO
    HIGH AlarmLED
    DELAY_MS 500
    LOW AlarmLED
    DELAY_MS 500
  LOOP
```

In the above example the alarm LED will flash until the Propeller is reset. `DO..LOOP` allows for condition testing before and after the loop statements as show in the examples below.

```
Alarm_On:
  DO WHILE AlarmStatus = 1
    HIGH AlarmLED
    DELAY_MS 500
    LOW AlarmLED
    DELAY_MS 500
  LOOP
  GOTO Main
```

```
Alarm_On:
  DO
    HIGH AlarmLED
    DELAY_MS 500
    LOW AlarmLED
    DELAY_MS 500
  LOOP UNTIL AlarmStatus = 0
  GOTO Main
```

When the second form is used the loop statements will run at least once before the condition is tested.

DO..LOOP can also be used to emulate a **DJNZ** loop without the need of a specific label; for example:

```
SUB ALARM_BURST
  bCount        VAR     Long

  bCount = __param1                    ' capture burst count
  DO
    TOGGLE AlarmLED                    ' toggle the output
    PAUSE 50                           ' hold briefly
  LOOP bCount                          ' loop while bCount > 0
  LOW AlarmLED                         ' make sure LED is off
  ENDSUB
```

In this form the variable (*bCount*) is decremented at the end of the loop and if not zero, the loop statements will be run again. As above, using this form will cause the loop statements to be run at least one time before *Variable* is tested.

Related instructions: **FOR..NEXT**, **DJNZ**, **EXIT**

END

`END`

Function

Ends program execution.

Explanation

`END` prevents the PropBASIC program from executing any further instructions and places the Propeller in low-power mode until it is reset (via RESn pin).

EXIT

`{IF Condition THEN} EXIT`

Function

Causes the immediate termination of a loop construct (`FOR..NEXT` or `DO..LOOP`) when *Condition* evaluates as **True**.

✔ *Condition* is a simple statement, such as `idx = 7` that can be evaluated as **True** or **False**. Only one comparison operator is allowed (see `IF..THEN` for valid condition operators)..

Explanation

The `EXIT` instruction allows a program to terminate a loop construct before the loop limit test is executed. For example:

```
Main:
  FOR idx = 1 TO 15
    IF idx > 9 THEN EXIT
    SEROUT TX, Baud, "*"
  NEXT
```

In this program, the `FOR..NEXT` loop will not run past nine because the `IF..THEN` test contained within will force the loop to terminate when idx is greater than nine. Note that the `EXIT` command only terminates the loop that contains it. In the above program, only nine asterisks will be transmitted on the TX pin.

Here is the `DO..LOOP` version of the same program:

```
Start:
  idx = 1

Main:
  DO
    IF idx > 9 THEN EXIT
    SEROUT TX, Baud, "*"
    INC idx
  LOOP WHILE idx <= 15
```

EXIT may also be used by itself when part of a larger **IF..THEN..ENDIF** or **DO..LOOP** block:

```
IF idx > 9 THEN
  SEROUT TX, Baud, CR
  idx = 1
  EXIT
ENDIF
```

Related instructions: **IF..THEN**, **DO..LOOP**

FOR..NEXT

```
FOR Variable = StartVal TO EndVal {STEP {-}StepVal}
  Statement(s)
NEXT
```

Function

Create a repeating loop that executes the program lines between `FOR` and `NEXT`, incrementing or decrementing *Variable* according to *StepVal* until the value of *Variable* reaches or passes the *EndVal*.

- ✔ ***Variable*** is simple variable or array element.
- ✔ ***StartVal*** is a constant or variable that sets the starting value of the counter.
- ✔ ***EndVal*** is a constant or a variable that sets the ending value of the counter.
- ✔ ***StepVal*** is an optional constant or a variable by which *Variable* is incremented or decremented (when negative) during each iteration of the loop.
- ✔ ***Statement*** is any valid PropBASIC statement.

Explanation

The `FOR..NEXT` loop allows a program to execute a series of instructions for a specified number of repetitions. By default, each time through the loop *Variable* is incremented by one. It will continue to loop until the value of the *Variable* reaches or surpasses *EndVal*. Also, `FOR..NEXT` loops always execute at least once. The simplest form is shown here::


```
Blink_LED:
  FOR idx = 1 TO 10                     ' blink 10 times
    HIGH LED                            ' light the LED
    PAUSE 200                           ' wait 0.2 secs
    LOW LED                             ' extinguish the LED
    PAUSE 300                           ' wait 0.3 secs
  NEXT
```

In above example the `FOR` instruction initializes *idx* to one. Then the `HIGH`, `PAUSE`, `LOW`, and `PAUSE` instructions are executed. At `NEXT`, *idx* is incremented and then checked to see if it is less than or equal to 10. If it is the loop instructions run again, otherwise the program falls through to the line that follows `NEXT`.

Related instructions: `DO..LOOP`, `EXIT`

GETADDR

`GETADDR HubSymbol, Variable`

Function

Returns the address of a Hub variable or `xDATA` element.

- ✔ ***HubSymbol*** is the variable or named `xDATA` element in the Hub
- ✔ ***Variable*** is the local variable that will hold the address of *HubSymbol*

Explanation

`GETADDR` is used to retrieve the address of a hub-based entity (variable or `xDATA` element) for use with the `RDxxxx` and `WRxxxx` instructions. For example:

```
DEVICE          P8X32A, XTAL1, PLL16X
XIN             5_000_000

LEDs            PIN     23..16 LOW

Pattern         DATA    %00011000, %00100100, %01000010, %10000001

addr            VAR     Long
idx             VAR     Long
bits            VAR     Long


PROGRAM Main

Main:
  GETADDR Pattern, addr                   ' get hub address of Pattern
  FOR idx = 1 TO 4                        ' run 4x
    RDBYTE addr, bits                     ' read pattern bits
    LEDs = bits                           ' output to Demo Board LEDs
    PAUSE 100                             ' hold 0.1s
    INC addr                              ' point to next pattern
  NEXT
  GOTO Main
```

In this example the address of *Pattern*, a Hub-based table, is placed in the local variable *addr*. This variable is used with `RDBYTE` to retrieve LED patterns from the table.

Related instructions: `RDxxxx`, `WRxxxx`

GOSUB (*Obsolete*)

`GOSUB Label`

Function

Jump to the point in the program specified by *Label* with the intention of returning to the line that follows the **GOSUB** instruction.

✔ ***Label*** is a valid program label that is followed by operational code; this code block is terminated with **RETURN**.

Explanation

GOSUB is used to call a block of code (*undeclared* subroutine) that will be terminated with **RETURN**.

Note: **GOSUB** is considered obsolete and existing programs should be updated to use declared subroutines (**SUB**) and functions (**FUNC**).

Related instructions: **RETURN**, **SUB**, **FUNC**

GOTO

`GOTO Label`

Function

Jump to the point in the program specified by *Label*.

✔ ***Label*** is a valid program label that is followed by operational code.

Explanation

The **GOTO** instruction forces the PropBASIC program to jump to a *Label* and execute the code that follows. A common use for **GOTO** is to create endless loops; programs that repeat a group of instructions over and over.

```
Main:
  HIGH RedLed                           ' Red LED on
  LOW GreenLed                          ' Green LED off
  DELAY_MS 250                          ' hold 0.25s

  LOW RedLed                            ' Red LED off
  HIGH GreenLed                         ' Green LED on
  DELAY_MS 750                          ' hold 0.75s

  GOTO Main
```


Related instruction: **ON..GOTO**

HIGH

`HIGH [PinName | PinNum]`

Function

Make the specified *Pin* an output and high (1).

- ✔ ***PinName*** is the symbol of a named (with `PIN`) IO pin.
- ✔ ***PinNum*** is a variable or constant (0 to 31).

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port).

Explanation

The `HIGH` instruction makes the specified *Pin* an output, and then sets its value to 1 (Vdd). For example:

```
HIGH AlarmLed
```

...does the same thing as:

```
OUTPUT AlarmLed
AlarmLed = 1
```

While using the `HIGH` instruction is more convenient, it does arbitrarily make the designated IO pin an output, even if that pin is already in an output state, potentially resulting in unnecessary code space use. If the pin was previously made an output with `LOW`, `HIGH`, or `OUTPUT` (or by using the `OUTPUT` modifier of the `PIN` declaration) you can make the pin "high" by writing a "1" to it as shown in the example above.

Related instructions: `LOW`, `TOGGLE`, `OUTPUT`

I2CREAD

```
I2CREAD SDAPin, SCLPin, Variable {, AckValue}
```

I2CSTART

```
I2CSTART SDAPin, SCLPin
```

I2CSTOP

`I2CSTOP SDAPin, SCLPin`

I2CWRITE

```
I2CWRITE SDAPin, SCLPin, Value {, AckVariable}
```

IF..THEN..ELSE..ENDIF

```
IF Condition THEN
  statement(s)
{ [ELSE | ELSEIF Condition]
  statement(s)}
ENDIF

IF Condition {[OR | AND]
   Condition} THEN
  statement(s)
{ [ELSE | ELSEIF Condition]
  statement(s)}
ENDIF
```

INC

`INC Variable {, Delta}`

Function

Increment (increase) the value of *Variable*.

- ✔ ***Variable*** is simple variable or array element.
- ✔ ***Delta*** is the value to add to *Variable*. If not specified, *Delta* is set to one.

Explanation

INC is a short-form version of:

```
  Variable = Variable + Delta
```

The INC instruction adds *Delta* to *Variable*. If *Delta* is not specified it will be set to one (1). Signed operators are used, so adding a negative *Delta* has he same effect as subtracting a positive *Delta*.

```
Main:
  result = 7
  INC result                            ' result is now 8
  INC result, -1                        ' result is now 7
  result = $FFFF_FFFF                   ' result is -1
  INC result                            ' result is now $0000_0000
```

Related instruction: DEC

INPUT

`INPUT [PinName | PinNum]`

Function

Make the specified *Pin* an input by writing a zero (0) to the corresponding bit of the `DIRA` register.

- ✔ ***PinName*** is the symbol of a named (with `PIN`) IO pin.
- ✔ ***PinNum*** is a variable or constant (0 to 31).

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port).

Explanation

There are several ways to make a pin an input. When a PropBASIC program is reset, all of the IO pins are made inputs. Instructions that rely on input pins (e.g., `PULSIN`, `SERIN`) automatically change the specified pin to input mode. Writing 0s to particular bits of the `DIRA` register makes the corresponding pins inputs. The programmer can manually set any pin to input mode with the `INPUT` instruction.

Related instructions: `OUTPUT`, `REVERSE`

LET

```
{LET} Variable = [Value | PinGroup]
{LET} Variable = {Value} Operator Value
{LET} PinGroup = Value
```

Function

Assign a *Value* or result of an expression to *Variable*, or a *Value* to an output pin group.

- ✔ ***Variable*** is a simple variable or array element
- ✔ ***Value*** is a variable or constant
- ✔ ***PinGroup*** is a [contiguous] group of pins

Explanation

LET is used when assigning a *Value* (or result of an expression) to a *Variable*, or a *Value* to a pin group (defined with PIN). LET is optional and generally not used in modern programs.

This line:

```
propBASIC = 100
```

does exactly the same as:

```
LET propBASIC = 100
```

LOCKCLR

`LOCKCLR ID {, Variable}`

Function
Clear lock to **False** and copies its previous state to *Variable*

- ✔ ***ID*** is a variable or constant, 0 to 7, that specifies the lock to clear.
- ✔ ***Variable*** is a simple variable or array element that will receive the previous lock state.

Explanation
LOCKCLR is one of four lock instructions (LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR) used to manage resources that are user-defined and deemed mutually exclusive. LOCKCLR clears the lock described by *Value* to zero (0) and returns the previous state of that lock in *Variable*.

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

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

Related instructions: LOCKNEW, LOCKRET, LOCKSET

LOCKNEW

`LOCKNEW Variable`

Function

Check out a new lock and store its ID in *Variable*

✔ ***Variable*** is a simple variable or array element that will receive the new lock ID.

Explanation

`LOCKNEW` is one of four lock instructions (`LOCKNEW`, `LOCKRET`, `LOCKSET`, and `LOCKCLR`) used to manage resources that are user-defined and deemed mutually exclusive. `LOCKNEW` checks out a unique lock, from the hub, and retrieves the ID of that lock, storing it in *Variable*.

Locks

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

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

Related instructions: `LOCKCLR`, `LOCKRET`, `LOCKSET`

LOCKRET

`LOCKRET ID`

Function
Release lock back for future "new lock" requests.

✔ ***ID*** is a variable or constant, 0 to 7, that specifies the lock to return to the lock pool.

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

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

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

Related instructions: LOCKCLR, LOCKNEW, LOCKSET

LOCKSET

`LOCKSET ID {, Variable}`

Function

Set lock to true and get its previous state.

- ✔ ***ID*** is a variable or constant, 0 to 7, that specifies the lock to set.
- ✔ ***Variable*** is a simple variable or array element that will receive the previous lock state.

Explanation

`LOCKSET` is one of four lock instructions (`LOCKNEW`, `LOCKRET`, `LOCKSET`, and `LOCKCLR`) used to manage resources that are user-defined and deemed mutually exclusive. `LOCKSET` sets the lock described by the register ID to one (1) and returns the previous state of that lock in *Variable*.

Locks

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

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

Related instructions: `LOCKCLR`, `LOCKNEW`, `LOCKRET`

LOW

`LOW [PinName | PinNum]`

Function
Make the specified *Pin* an output and high (1).

- ✔ ***PinName*** is the symbol of a named (with `PIN`) IO pin.
- ✔ ***PinNum*** is a variable or constant (0 to 31).

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port).

Explanation
The `LOW` instruction makes the specified *Pin* an output, and then sets its value to 0 (Vss). For example:

```
LOW AlarmLed
```

… does the same thing as:

```
OUTPUT AlarmLed
AlarmLed = 0
```

While using the `LOW` instruction is more convenient, it does arbitrarily make the designated IO pin an output, even if that pin is already in an output state, potentially resulting in unnecessary code space use. If the pin was previously made an output with `LOW`, `HIGH`, or `OUTPUT` (or by using the `OUTPUT` modifier of the `PIN` declaration) you can make the pin "low" by writing a "0" to it as shown in the example above.

Related instructions: `HIGH`, `TOGGLE`, `OUTPUT`

NOP

`NOP`

Function

No OPeration – does nothing except consume one PASM instruction (four clock cycles). Useful for allowing IO pins to settle after a change of state.

ON..GOSUB

```
ON Offset GOSUB Label0 {, Label1, ...}
ON Variable = Value0 {, Value1, ...} GOSUB Label0 {, Label1, ...}
```

Function

Jump to the program *Label* specified by *Offset* (if in range) with the intent of returning to the line that follows `ON..GOSUB`. If *Offset*-1 is greater than the number of elements in the address table the `GOSUB` is ignored. Alternate syntax allows *Variable* to be compared to a list of *Values* to create an internal offset.

- ✔ ***Offset*** is simple variable or array element.
- ✔ ***Label*** is a valid program label that is followed by operational code; this code block is terminated with `RETURN`.
- ✔ ***Variable*** is a simple variable or array element.
- ✔ ***Value*** is a numeric or character constant (e.g., "A").

Explanation

The `ON..GOSUB` instruction is useful when you want to write something like this:

```
Process_Cmd:
  IF cmd = 0 THEN
    ROBOT_STOP
  ELSEIF cmd = 1 THEN
    ROBOT_RT
  ELSEIF cmd = 2 THEN
    ROBOT_LF
  ENDIF
```

The above code is simplified with `ON..GOSUB` as follows:

```
Process_Cmd:
  ON cmd GOSUB ROBOT_STOP, ROBOT_RT, ROBOT_LF
```

Alternate syntax allows a non-contiguous list of values to be converted to an internal offset, for example:

```
Process_Cmd:
  ON cmd = "S", "R", "L" GOSUB ROBOT_STOP, ROBOT_RT, ROBOT_LF
```

Note: `ON..GOSUB` should only be used with subroutines that do not expect parameters, as parameter passing with `ON..GOSUB` is not possible.

Related instructions: `GOSUB`, `ON..GOTO`

ON..GOTO

```
ON Offset GOTO Label0 {, Label1, ...}
ON Variable = Value0 {, Value1, ...} GOTO Label0 {, Label1, ...}
```

Function

Jump to the program *Label* specified by *Offset* (if in range). If *Offset*-1 is greater than the number of elements in the address table, the program continues at the line following `ON..GOTO`. Alternate syntax allows *Variable* to be compared to a list of *Values* to create an internal offset.

- ✔ ***Offset*** is simple variable or array element.
- ✔ ***Label*** is a valid program label that is followed by operational code; this code block is terminated with `RETURN`.
- ✔ ***Variable*** is a simple variable or array element.
- ✔ ***Value*** is a variable or constant.

Explanation

The `ON..GOTO` instruction is useful when you want to write something like this:

```
Check_Value:
  IF value = 0 THEN Case_0              ' if value is 0, jump to Case_0
  IF value = 1 THEN Case_1              ' if value is 1, jump to Case_1
  IF value = 2 THEN Case_2              ' if value is 2, jump to Case_2

No_Match:
```

The above code is simplified with `ON..GOTO` as follows:

```
Check_Value:
  ON value GOTO Case_0, Case_1, Case_2

No_Match:
```

`ON..GOTO` is useful for creating command handlers; for example:

```
Get_Cmd:
  SERIN RX, Baud, cmd
  ON cmd = "S", "R", "L" GOTO Cmd_Stop, Cmd_Right, Cmd_Left

Bad_Cmd:
  ' handle bad command here
  GOTO Get_Cmd
```

Related instructions: `BRANCH`, `ON..GOSUB`

OUTPUT

`OUTPUT [PinName | PinNum]`

Function

Make the specified *Pin* an output by writing a one (1) to the corresponding bit of the DIRA register.

- ✔ ***PinName*** is the symbol of a named (with PIN) IO pin.
- ✔ ***PinNum*** is a variable or constant (0 to 31).

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port).

Explanation

There are several ways to make a pin an output. When a PropBASIC program is reset, all of the IO pins are made inputs. Instructions that rely on output pins (e.g., PULSOUT, SEROUT) automatically change the specified pin to output mode. Writing 1s to particular bits of the DIRA register makes the corresponding pins outputs. The programmer can manually set any pin to output mode with the OUTPUT instruction.

Related instructions: INPUT, REVERSE

OWREAD

`OWREAD Pin, Variable{\Bits}`

Function

Receive bits (usually eight) from a 1-Wire device.

- ✔ ***Pin*** is variable or constant (0 to 31) that specifies the Propeller IO pin to use. This pin should be pulled up to Vdd through a 1kΩ ~ 4.7kΩ resistor.
- ✔ ***Variable*** is a variable that will store the received value.
- ✔ ***Bits*** specifies the number of bits to receive from the 1-Wire device. If not specified *Bits* is set to eight.

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port). Pin 31 is useful for receiving via the Propeller programming port to a terminal program.

Explanation

Most 1-Wire transactions require reading data from the device. A bit is read from the 1-Wire device byte generating a brief pulse on *Pin* and then reading the line within 15 µS of the falling edge. This is called a "Read Slot." The `OWREAD` instruction generates *Bits* 1-Wire Read Slot sequences and returns the value in *Variable*.

Related instructions: `OWRESET`, `OWWRITE`

OWRESET

`OWRESET Pin {, Variable}`

Function

Generates a 1-Wire reset sequence on Pin, returning (optional) status information in ByteVar.

✔ ***Pin*** is variable or constant (0 to 31) that specifies the Propeller IO pin to use. This pin should be pulled up to Vdd through a 1kΩ ~ 4.7kΩ resistor.

✔ ***Variable*** is a variable that will store the received value.

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port). Pin 31 is useful for receiving via the Propeller programming port to a terminal program.

Explanation

All transactions on the 1-Wire bus begin with an Initialization sequence that consists of a Reset pulse generated by the master, followed by a Presence pulse generated by the 1-Wire slave. The `OWRESET` instruction generates the 1-Wire Reset pulse on the specified DQ *Pin* and, if *Variable* is specified, will monitor the bus and return status information to the program.

1-Wire Buss Status Returned in *Variable*	
%00	1-Wire buss pin shorted to Vss
%01	Bad connection
%10	Good connection
%11	No device present on 1-Wire buss

Related instructions: `OWREAD`, `OWWRITE`

OWWRITE

`OWWRITE Pin, Value{\Bits}`

Function

Writes Value to the 1-Wire buss.

- ✔ ***Pin*** is variable or constant (0 to 31) that specifies the Propeller IO pin to use. This pin should be pulled up to Vdd through a 1kΩ ~ 4.7kΩ resistor.
- ✔ ***Value*** is a variable or constant.
- ✔ ***Bits*** specifies the number of bits to receive from the 1-Wire device. If not specified *Bits* is set to eight.

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port). Pin 31 is useful for receiving via the Propeller programming port to a terminal program.

Explanation

After reset, 1-Wire transactions require writing values to the buss. A bit is written by generating a timed low pulse on the DQ line; this is called a "Write Slot". The `OWWRITE` instruction generates *Bits* Write Slot sequences to put *Value* on the 1-Wire buss.

Related instructions: `OWRESET`, `OWREAD`

PAUSE

```
PAUSE Duration
```

Function

Pause the program (do nothing) for a number of milliseconds.

✔ *Duration* is a variable or constant value, 0 to POSX (2,147,483,647).

Note: When a constant is used the value may be fractional, e.g., 10.25.

Explanation

PAUSE delays the execution of the next program instruction for a number of milliseconds, specified in *Duration*.

```
Flash:
  FOR flashes = 1 TO 10
    HIGH AlarmLed
    PAUSE 500
    LOW AlarmLed
    PAUSE 500
  NEXT
```

When this code runs the *AlarmLed* pin will go high for 500 milliseconds and then go low for 500 milliseconds. This process will run a total of 10 times controlled by the FOR..NEXT loop.

Note that a PAUSE duration of up to 2,147,483.6 seconds is possible with the Propeller's 32-bit variable/constant values.

As delays are so frequently used in programs, code space can be conserved by encapsulating the PAUSE instruction in a subroutine. Start by defining a shell routine for PAUSE like this:

```
DELAY_MS    SUB   1, 1
```

Then code the subroutine like this:

```
SUB DELAY_MS
  PAUSE __param1
  ENDSUB
```

To use this subroutine you would simply substitute **DELAY_MS** for PAUSE in the body of your program. Note that when using this subroutine only whole values may be specified.

Related instructions: PAUSEUS, WAITCNT

PAUSEUS

```
PAUSEUS Duration
```

Function

Pause the program (do nothing) for a number of microseconds.

✔ ***Duration*** is a variable or constant value, 0 to POSX (2,147,483,647).

Note: When a constant is used the value may be fractional, e.g., 10.25.

Explanation

PAUSEUS delays the execution of the next program instruction for a number of microseconds, specified in *Duration*.

```
Tone:
  OUTPUT Speaker
  FOR timer = 1 TO 2_000
    TOGGLE Speaker
    PAUSEUS 500
  NEXT
```

When this code runs the *Speaker* pin will output a ~1kHz square wave for one second (1,000 milliseconds).

Note that a PAUSEUS duration of up to 2,147.48 seconds is possible with the Propeller's 32-bit variable/constant values.

As delays are so frequently used in programs, code space can be conserved by encapsulating the PAUSEUS instruction in a subroutine. Start by defining a shell routine for PAUSEUS like this:

```
DELAY_US    SUB   1, 1
```

Then code the subroutine like this:

```
SUB DELAY_US
  PAUSEUS __param1
  ENDSUB
```

To use this subroutine you would simply substitute **DELAY_US** for PAUSEUS in the body of your program. Note that when using this subroutine only whole values may be specified.

Related instructions: PAUSE, WAITCNT

PULSIN

`PULSIN Pin, State, Variable`

Function
Measure the width of a pulse (in microseconds) on *Pin* described by *State* and store the result in *Variable*.

- ✔ ***Pin*** is a symbol, variable or constant (0 to 31) that specifies the Propeller IO pin to use. This pin will be set to input mode.
- ✔ ***State*** is a constant (0 or 1) that specifies whether the pulse to be measured is low (0) or high (1). A low pulse begins with a 1-to-0 transition, and a high pulse begins with a 0-to-1 transition..
- ✔ ***Variable*** is simple variable or array element.

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port).

Explanation
`PULSIN` is like a fast stopwatch that is triggered by a change in state (0 or 1) on the specified pin. The entire width of the specified pulse (high or low) is measured, in microseconds and stored in *Variable*.

Many analog properties (voltage, resistance, capacitance, frequency, duty cycle) can be measured in terms of pulse duration. This makes `PULSIN` a valuable form of analog-to-digital conversion.

`PULSIN` makes *Pin* an input and then waits for the desired pulse, for up to the maximum pulse width it can measure `POSX` (2,147,483,647) microseconds. If it sees the desired pulse it measures the time until the end of the pulse and stores the result in *Variable*. If it never sees the start of the pulse, or the pulse is too long (greater than the `POSX` microseconds), `PULSIN` "times out" and store 0 in *Variable*.

Related instruction: `PULSOUT`

PULSOUT

`PULSOUT Pin, Duration`

Function

Generate a pulse on *Pin* with a width of *Duration* microseconds.

- ✔ ***Pin*** is variable or constant (0 to 31) that specifies the Propeller IO pin to use. This pin will be set to output mode.
- ✔ ***Duration*** is a variable or constant that specifies the pulse width in one-microsecond units.

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port).

Explanation

PULSOUT sets *Pin* to output mode, inverts the state of that pin; waits for the specified *Duration* (in microseconds); then inverts the state of the pin again returning the bit to its original state.

Note that a **PULSOUT** duration of up to 2,147.48 seconds is possible with the Propeller's 32-bit variable/constant values.

```
Start:
  LOW Servo

Main:
  FOR position = 1_000 TO 1_999 STEP 10
    PULSOUT Servo, position
    DELAY_MS 20
  NEXT

  FOR position = 2_000 TO 1_001 STEP -10
    PULSOUT Servo, position
    DELAY_MS 20
  NEXT

  GOTO Main
```

Related instruction: **PULSIN**

RANDOM

`RANDOM Seed {, Duplicate}`

Function

Generate a pseudo-random number using *Variable* as the seed.

- ✔ ***Seed*** is a variable or array element that serves as the seed and result for **RANDOM**. Each pass through **RANDOM** stores the next number, in the pseudo-random sequence, in *Seed*.
- ✔ ***Duplicate*** is an optional variable that, if provided, will receive a copy of *Seed* after **RANDOM**. This variable may be modified without affecting the value of *Seed* for the **RANDOM** instruction.

Explanation

RANDOM generates pseudo-random numbers ranging from **$0** to **$FFFF_FFFF**. The value is called "pseudo-random" because it appears random, but is generated by a logic operation that uses the initial value in *Seed* to "tap" into a sequence of essentially random numbers. If the same initial value, called the "seed", is always used, then the same sequence of numbers will be generated.

The code below [pseudo-] randomly selects and lights one of the LEDs on the Propeller Demo board:

```
DEVICE          P8X32A, XTAL1, PLL16X
XIN             5_000_000

LEDs            PIN     23..16  OUTPUT  ' make LEDs outputs

seed            VAR     Long
theLed          VAR     Long


PROGRAM Start

Start:
  RANDOM seed                           ' stir seed
  theLed = seed // 8                    ' randomize, 0 to 7
  theLed = theLed + 16                  ' offset, 16 to 23
  HIGH theLed                           ' LED on
  PAUSE 100                             ' hold 0.1s
  LOW theLed                            ' LED off
  GOTO Start
```

RCTIME

```
RCTIME Pin, State, Variable
```

RDBYTE, RDWORD, RDLONG

`RDxxxx HubAddress{(Offset)}, Variable {, Variable, ...}`

Function

Read one or more values from an address in the Hub.

✔ ***HubAddress*** is the base address, in the Hub, of the value(s) to read. With multiple variables in one instruction this is the address of the first item.

✔ ***Offset*** is a zero-indexed offset which is added to *HubAddress*.

✔ ***Variable*** is a simple variable or array element.

Explanation

RDxxxx reads the value at *HubAddress* and stores it in *Variable*. The following example program uses **RDBYTE** to retrieve LED patterns from a Hub-based **DATA** table.

```
DEVICE          P8X32A, XTAL1, PLL16X
XIN             5_000_000

LEDs            PIN     23..16  OUTPUT  ' make LEDs outputs

Pattern         DATA    %00011000, %00100100, %01000010, %10000001

idx             VAR     Long
bits            VAR     Long


PROGRAM Main

Main:
  FOR idx = 0 TO 3                          ' run 4x
    RDBYTE Pattern(idx), bits               ' read pattern bits
    LEDs = bits                             ' output to Demo Board LEDs
    PAUSE 250                               ' hold 1/4s
  NEXT
  GOTO Main
```

Related instructions: **WRxxxx**, **GETADDR**

RETURN (*from GOSUB* – *Obsolete*)

`RETURN {Value}`

Function

Return from a subroutine (previously called with GOSUB).

✔ ***Value*** is a variable or constant value to be returned to the calling code.

Explanation

RETURN sends the program back to the address (instruction) immediately following the most recent GOSUB. Use of this form is considered obsolete and existing programs should be rewritten to use declared subroutines and functions. If this form is used with the optional return *Value* the programmer should retrieve this value from internal variable __param1 in the line that follows GOSUB.

Related instructions: GOSUB, SUB, FUNC

RETURN (*value from declared Function*)

```
RETURN Value {, Value, {, Value, {, Value}}}
```

Function

Return one or more values from a declared function.

✔ ***Value*** is a variable or constant value to be returned to the calling code.

Explanation

PropBASIC functions allow the programmer to return from one or more values to the calling code. For example, the following function:

```
FUNC TRIPLE_IT
  __param2 = __param1 << 1                ' __param2 = __param1 x 2
  __param1 = __param1 + __param2
  RETURN __param1
  ENDFUNC
```

...would be called like this:

```
  variable = TRIPLE_IT value
```

See the section on defining and using functions (page ??) for additional details.

Related instructions: `FUNC`

REVERSE

`REVERSE [PinName | PinNum]`

Function

Reverse the data direction register (`DIRA`) bit of the specified pin.

✔ ***PinName*** is the symbol of a named (with `PIN`) IO pin.

✔ ***PinNum*** is a variable or constant (0 to 31).

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port).

Explanation

`REVERSE` is convenient way to switch the IO direction of a pin. If the pin is an input, `REVERSE` makes it an output; if it's an output, `REVERSE` makes it an input.

Remember that "input" really has two meanings: (1) Setting a pin to input makes it possible to check the state (1 or 0) of external circuitry connected to that pin. The current state is in the corresponding bit of the `INA` register. (2) Setting a pin to input also disconnects the output driver, possibly affecting circuitry being controlled by the pin.

Related instructions: `INPUT`, `OUTPUT`

SERIN

```
SERIN Pin, BaudMode, Variable
```

Function

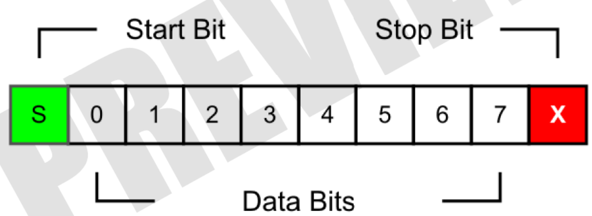
Receive an asynchronous serial byte (e.g., RS-232).

- ✔ ***Pin*** is variable or constant (0 to 31) that specifies the Propeller IO pin to use.
- ✔ ***BaudMode*** is a string constant that specifies serial timing and configuration. PropBASIC will raise an error if the baud rate specified exceeds the ability of the target XIN/FREQ setting.
- ✔ ***Variable*** is a variable that will store the received value.

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port). Pin 31 is useful for receiving via the Propeller programming port to a terminal program.

Explanation

Receive an asynchronous serial byte at the selected baud rate and mode using no parity, eight data bits, and one stop bit (8N1). Serial bits are received LSB-first as shown here:



Using SERIN inline:

```
SERIN 31, T9600, rxResult
```

In the above example the Propeller will receive a byte from an external device at 9600 baud, in True mode on pin 31 (the RX pin of the Propeller's programming port) and store it in the variable *rxResult*. Since SERIN requires a substantial amount of Assembly code a good way to save program space is by placing SERIN in a function. For example:

```
' Use: result = RX_BYTE rxpin

FUNC RX_BYTE
  __param2 = param1
  SERIN __param2, Baud, __param1, Baud
  ENDFUNC
```

This function requires just one parameter: the pin to use for receiving the serial data. The baud rate for **RX_BYTE** is set in a program constant. By using a variable RX pin this routine can be used for multiple devices that use the same baud rate.

Understanding BaudMode

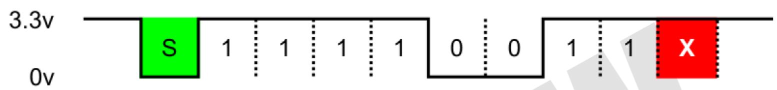
The `SERIN` instruction requires a *BaudMode* parameter which defines the baud rate (in bits per second) and the polarity with which the bits arrive.

There are two modes of serial reception:

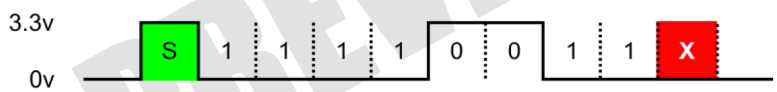
- ✔ ***True*** (“Txxxx”)
- ✔ ***Inverted*** (“Nxxxx”)

…where “xxxx” is the baud rate in bits per second (e.g., 9600).

In *True* mode communications the line idle state is high, the start bit (S) is low, data bits can be read directly from the line, and the stop bit (X) is high. If you looked at the input of a Propeller receiving the value $CF you would see this:



Inverted mode uses the opposite polarity; the line idle state is low, the start bit is high, data bits are inverted (low = 1, high = 0), and the stop bit is low. This is what $CF looks like when receiving using Inverted mode:



Note: As the RX pin used for `SERIN` is set to input mode, `OT` (open-true) and `ON` (open-inverted) are functionally the same as `T` (true) and `N` (inverted).

Related instruction: `SEROUT`

SEROUT

`SEROUT Pin, BaudMode, [Value | String | Label]`

Function

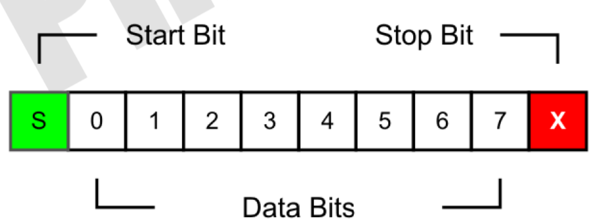
Transmit an asynchronous serial byte or string (e.g., RS-232).

- ✔ ***Pin*** is variable or constant (0 to 31) that specifies the Propeller IO pin to use.
- ✔ ***BaudMode*** is a string constant that specifies serial timing and configuration. PropBASIC will raise an error if the baud rate specified exceeds the ability of the target `XIN/FREQ` setting.
- ✔ ***Value*** is a variable or constant (0 to 255) to be transmitted (only the lower eight bits of the value will be transmitted).
- ✔ ***String*** is an inline string, e.g., “PropBASIC”
- ✔ ***Label*** is `DATA` label that holds a valid z-string

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port). Pin 30 is useful for transmitting via the Propeller programming port to a terminal program.

Explanation

Transmit asynchronous serial byte (or inline/data string) at the selected baud rate and mode using no parity, eight data bits, and one stop bit (8N1). Serial bits are transmitted LSB-first as shown here:



Using `SEROUT` inline:

```
SEROUT 30, T9600, "A"
```

In the above example the Propeller will transmit the letter "A" (decimal 65) to an external device at 9600 baud, in True mode on pin 30 (the TX pin of the Propeller’s programming port). Since `SEROUT` requires a substantial amount of Assembly code a good way to save program space is by placing SEROUT in a subroutine. For example:

```
' Use: TX_BYTE txpin, byteout
' -- shell for SEROUT
' -- allows selection of TX pin for multiple devices (e.g., LCD & terminal)
' -- Baud is set as program constant

SUB TX_BYTE
  SEROUT __param1, Baud, __param2
  ENDSUB
```

This subroutine takes two parameters: the first is the pin to use for transmitting, the second is the value to send. The baud rate for **TX_BYTE** is set in a program constant. By using a variable TX pin this routine can be used for multiple devices that use the same baud rate.

Understanding BaudMode

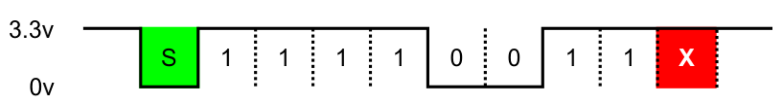
The SEROUT instruction requires a *BaudMode* parameter which defines the baud rate (in bits per second) and the mode in which the transmission pin is controlled. The mode actually defines two aspects of the output: signal polarity and how the transmission pin operates when sending a bit.

There are four modes of transmission:

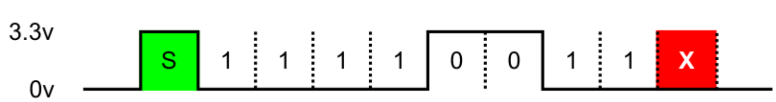
- ✔ ***True*** ("Txxxx")
- ✔ ***Inverted*** ("Nxxxx")
- ✔ ***Open-True*** ("OTxxxx")
- ✔ ***Open-Inverted*** ("ONxxxx")

…where "xxxx" is the baud rate in bits per second (e.g., 9600).

In *True* mode communications the line idle state is high, the start bit (S) is low, data bits can be read directly from the line, and the stop bit (X) is high. If you looked at the output from a Propeller transmitting the value $CF you would see this:

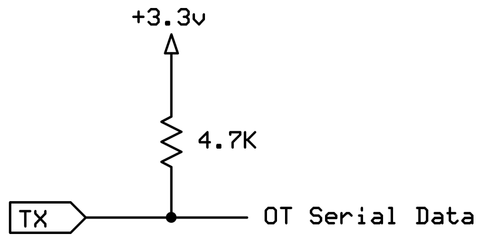


Inverted mode uses the opposite polarity; the line idle state is low, the start bit is high, data bits are inverted (low = 1, high = 0), and the stop bit is low. This is what $CF looks like when transmitting using *Inverted* mode:

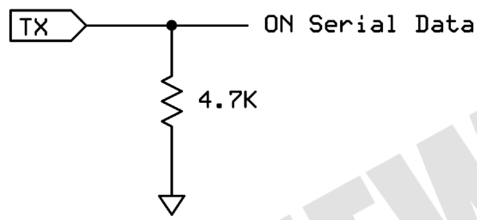


In both *True* and *Inverted* modes the Propeller drives the line high and low. When using a single pin to send and receive serial information an *Open* baud mode must be used. In these modes the Propeller drives the output pin in just one direction and relies on a pull-up (*Open-True*) or pull-down (*Open-Inverted*) resistor to set the other line state.

For *Open-True* mode the Propeller will pull the line low for a start bit or "0" bit, and let it float (high-impedance, input state) for a "1" bit or stop bit. This mode requires a pull-up on the serial pin to set the line for a "1" bit or the stop bit.



For *Open-Inverted* mode the Propeller will drive the line high for a start and zero bit, and let it float for a one bit and stop bit. Since the polarity is inverted we need to add a pull-down resister to the serial pin.



The *Open-True* mode is very popular and used by devices like the Parallax Servo Controller (PSC). By using an *Open* mode several devices may be connected to the serial pin. If a transmission error occurs and two devices attempt to transmit at the same time there will be no electrical problem as the devices drive the serial output in the same direction, and the opposite direction causes the output to float. With two serial devices that used a driven (non-open) mode, there could be a serious electrical conflict if one device attempted to transmit a "1" while another was transmitting a "0"; with both devices driving their pins as outputs this would cause an electrical short circuit, potentially damaging IO pins.

Related instruction: `SERIN`

SEROUT Demo

```
' =========================================================================
'
'   File...... serout_demo.pbas
'   Purpose... SEROUT demo using Propeller Demo Board
'   Author....
'   E-mail....
'   Started...
'   Updated...
'
' =========================================================================


' -------------------------------------------------------------------------
' Device Settings
' -------------------------------------------------------------------------

DEVICE          P8X32A, XTAL1, PLL16X
XIN             5_000_000


' -------------------------------------------------------------------------
' Constants
' -------------------------------------------------------------------------

Baud            CON     "T115200"

' Parallax Serial Terminal (PST) Constants

HOME            CON     1
BKSP            CON     8
TAB             CON     9
LF              CON     10
CLREOL          CON     11
CLRDN           CON     12
CR              CON     13
CLS             CON     16


' -------------------------------------------------------------------------
' I/O Pins
' -------------------------------------------------------------------------

TX              PIN     30  HIGH                    ' output and high (idle)
LED             PIN     16  LOW                     ' output and low


' -------------------------------------------------------------------------
' Variables
' -------------------------------------------------------------------------
```

```
alpha           VAR     Long


' =========================================================================
' Subroutine / Function Declarations
' =========================================================================

TX_BYTE         SUB     2                       ' shell for SEROUT
DELAY_MS        SUB     1                       ' shell for PAUSE


' =========================================================================
  PROGRAM Start
' =========================================================================

Start:
  DELAY_MS 10                                   ' TX idle for 10ms
  TX_BYTE TX, CLS

Main:
  DO
    FOR alpha = "A" TO "Z"
      TOGGLE LED
      TX_BYTE TX, alpha
      DELAY_MS 50
    NEXT
    TX_BYTE TX, CR
  LOOP


' -------------------------------------------------------------------------
' Subroutine / Function Code
' -------------------------------------------------------------------------

' Use: TX_BYTE txpin, byteout
' -- shell for SEROUT
' -- allows selection of TX pin for multiple devices
' -- Baud is set as program constant

SUB TX_BYTE
  SEROUT __param1, Baud, __param2
  ENDSUB

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

' Use: DELAY_MS milliseconds
' -- shell for PAUSE

SUB DELAY_MS
  PAUSE __param1
  ENDSUB
```

SHIFTIN

`SHIFTIN` *`DataPin`*`,` *`ClockPin`*`,` *`Mode`*`,` *`Variable`*`{\`*`Bits`*`}`

SHIFTOUT

```
SHIFTOUT DataPin, ClockPin, Mode, Value{\Bits}
```

STR

`STR ArrayName, Variable, Digits {,Mode}`

TOGGLE

```
TOGGLE [PinName | PinNum]
```

Function

Make the specified *Pin* an output and inverts its state.

- ✔ ***PinName*** is the symbol of a named (with `PIN`) IO pin.
- ✔ ***PinNum*** is a variable or constant (0 to 31).

Note: Exercise care with pins 31 and 30 (Propeller programming port) and 29 and 28 (program EEPROM I2C port).

Explanation

The `TOGGLE` instruction sets a pin to output mode and inverts the output state, changing 0 to 1 and 1 to 0.

```
Flash:
  LOW AlarmLed                           ' start off
  FOR flashes = 1 TO 20                  ' loop 20 times
    TOGGLE AlarmLed                      ' invert state of LED
    DELAY_MS 500                         ' wait 0.5s
  NEXT
```


Related instructions: `HIGH`, `LOW`, `OUTPUT`

WAITCNT

`WAITCNT Target, Delta`

Function

Pause a cog's execution temporarily.

- ✔ ***Target*** is the the target value to compare against the System Counter (**CNT**). When the System Counter has reached *Target's* value, *Delta* is added to *Target* and execution continues at the next instruction.
- ✔ ***Delta*** is the value is added to Target's value in preparation for the next **WAITCNT** instruction. This creates a synchronized delay window.

Explanation

WAITCNT, "Wait for System Counter," is one of four wait instructions (**WAITCNT**, **WAITPEQ**, **WAITPNE**, and **WAITVID**) used to pause execution of a cog until a condition is met. The **WAITCNT** instruction pauses the cog until the global System Counter equals the value in the *Target* register, then it adds *Delta* to *Target* and execution continues at the next instruction.

The following snippet will toggle an LED every 250ms.

```
Main:
  RDLONG 0, delta                         ' read system frequency
  delta = delta >> 2                      ' divide by 4
  target = CNT + delta                    ' sync with system counter
  DO
    TOGGLE 16                             ' toggle led on P16
    WAITCNT target, delta                 ' wait 1/4s (synchronized)
  LOOP
```

Related instructions: **PAUSE**, **PAUSEUS**

WAITPEQ

`WAITPEQ State, Mask`

Function

Pause a cog's execution until selected IO pin(s) match designated *State*.

- ✔ ***State*** is the value to compare against `INA` ANDed with *Mask*.
- ✔ ***Mask*** is the value that is bitwise-ANDed with `INA` before the comparison with *State*.

Explanation

`WAITPEQ`, "Wait for Pin(s) to Equal," is one of four wait instructions (`WAITCNT`, `WAITPEQ`, `WAITPNE`, and `WAITVID`) used to pause execution of a cog until a condition is met. The `WAITPEQ` instruction pauses the cog until the result of `INA` ANDed with *Mask* matches the value of *State*.

```
  WAITPEQ %0011, %1111
```

In the above example the Propeller will wait until the inputs P0..P3 (*Mask* = %1111) until P0 and P1 are high (1), and P2 and P3 are low (0).

Related instructions: `WAITPNE`

WAITPNE

WAITPNE *State*, *Mask*

Function

Pause a cog's execution until selected IO pin(s) do not match designated *State*.

- ✔ ***State*** is the value to compare against INA ANDed with *Mask*.
- ✔ ***Mask*** is the value that is bitwise-ANDed with INA before the comparison with *State*.

Explanation

WAITPNE, "Wait for Pin(s) Not to Equal," is one of four wait instructions (WAITCNT, WAITPEQ, WAITPNE, and WAITVID) used to pause execution of a cog until a condition is met. The WAITPNE instruction pauses the cog until the result of INA ANDed with *Mask* does not match the value of *State*.

```
  WAITPNE %1, %1
```

Assuming an active-low input on P0, the above line would cause the Propeller to wait until P0 goes low.

Related instructions: WAITPEQ

WAITVID

`WAITVID Colors, Pixels`

Function

Pause a cog's execution until its Video Generator is available to take pixel data.

- ✔ ***Colors*** is a value with four byte-sized color values, each describing the four possible colors of the pixel patterns in *Pixels*.
- ✔ ***Pixels*** is the value that is the next 16-pixel by 2-bit (or 32-pixel by 1-bit) pixel pattern to display.

Explanation

WAITVID, "Wait for Video Generator," is one of four wait instructions (**WAITCNT**, **WAITPEQ**, **WAITPNE**, and **WAITVID**) used to pause execution of a cog until a condition is met. The **WAITVID** instruction pauses the cog until its Video Generator hardware is ready for the next pixel data, then the Video Generator accepts that data (*Colors* and *Pixels*) and the cog continues execution with the next instruction.

Make sure to start the cog's Video Generator module and Counter A before executing the **WAITVID** command or it will wait forever.

WRBYTE, WRWORD, WRLONG

`WRxxxx HubAddress{(Offset)}, Value {, Value, ...}`

Function

Write one or more values to an address in the Hub.

- ✔ ***HubAddress*** is the base address, in the Hub, of the value(s) to write. With multiple values in one instruction this is the address of the first item.
- ✔ ***Offset*** is a zero-indexed offset which is added to *HubAddress.*
- ✔ ***Value*** is a variable or constant.

Explanation

`WRxxxx` writes *Value(s)* to the Hub RAM at *HubAddress*, unless a non-zero *Offset* is used. `WRxxxx` is a useful tool for passing values to processes running in other cogs (i.e., TASKs).

Related instructions: `RDxxxx`, `GETADDR`

Programming Examples

The examples that follow are in no way meant to provide an exhaustive demonstration of the features and capabilities of PropBASIC, but should give the inquisitive programmer ample inspiration for developing PropBASIC his/her own projects.

PropBASIC Errors and Warnings

Errors

01 INVALID VARIABLE NAME
You have used a reserved word for a variable name.

02 DUPLICATE VARIABLE NAME
You have declared a variable more than once.

03 CONSTANT EXPECTED
This parameter is required to be a constant.

04 INVALID UNARY OPERATOR
`-` and `~` are the only allowed unary operators.

05 INVALID PARAMETER
Generic invalid parameter error.

06 SYNTAX ERROR
Generic *"I didn't understand what you meant."* error message.

07 INVALID NUMBER OF PARAMETERS
You have too few or too many parameters given.

08 NOT A "FOR" CONTROL VARIABLE
You have specified a parameter after `NEXT` that is not a `FOR` control variable.

09 BAUDRATE IS TOO HIGH
Cannot achieve the desired baud rate.

10 UNKNOWN COMMAND
Command was not recognized.

11 COMMA EXPECTED
A comma is required between parameters.

12 FOR WITHOUT NEXT

13 NEXT WITHOUT FOR

14 TOO MANY SUBS DEFINED
Only 127 subroutines may be defined.

15 ELSE OR ENDIF WITHOUT IF
You are missing an IF statement before ELSE or ENDIF

16 LOOP WITHOUT DO
You are missing a `DO` statement before `LOOP.`

17 EXIT NOT IN FOR-NEXT OR DO-LOOP
The `EXIT` instruction must be inside a `FOR-NEXT` or `DO-LOOP`.

18 NO "PROGRAM" COMMAND USED
You must use the `PROGRAM` directive.

19 TOO MANY DEFINES
Only 512 defines are allowed.

20 NOT IN A SUB OR FUNC
`ENDSUB` and `ENDFUNC` can only be used inside a `SUB` or `FUNC.`

21 SUB OR FUNC CANNOT BE NESTED
SUBs and FUNCs cannot be nested.

22 NOT VALID INSIDE SUB
Command cannot be used inside a `SUB` or `FUNC`.

23 COULD NOT READ SOURCE FILE
`LOAD`, `INCLUDE` or `FILE` could not read the file specified.

24 DIRECTIVE ERROR
The program has used the `$ERROR` directive to cause an error.

25 NO FREQ SPECIFIED
The `FREQ` directive must be used before `PROGRAM`.

26 LONG VAR EXPECTED
A Long (32-bit) `VAR` parameter is expected.

Warnings

01 NOT RECOMMENDED WITH INTERNAL CLOCK
`SERIN` and `SEROUT` are not recommended with the internal clock.

02 ENDFUNC USED WITHOUT RETURN
Function ended without a `RETURN`.

03 DIRECTIVE WARNING:
The program has used the `$WARNING` directive to cause a warning.