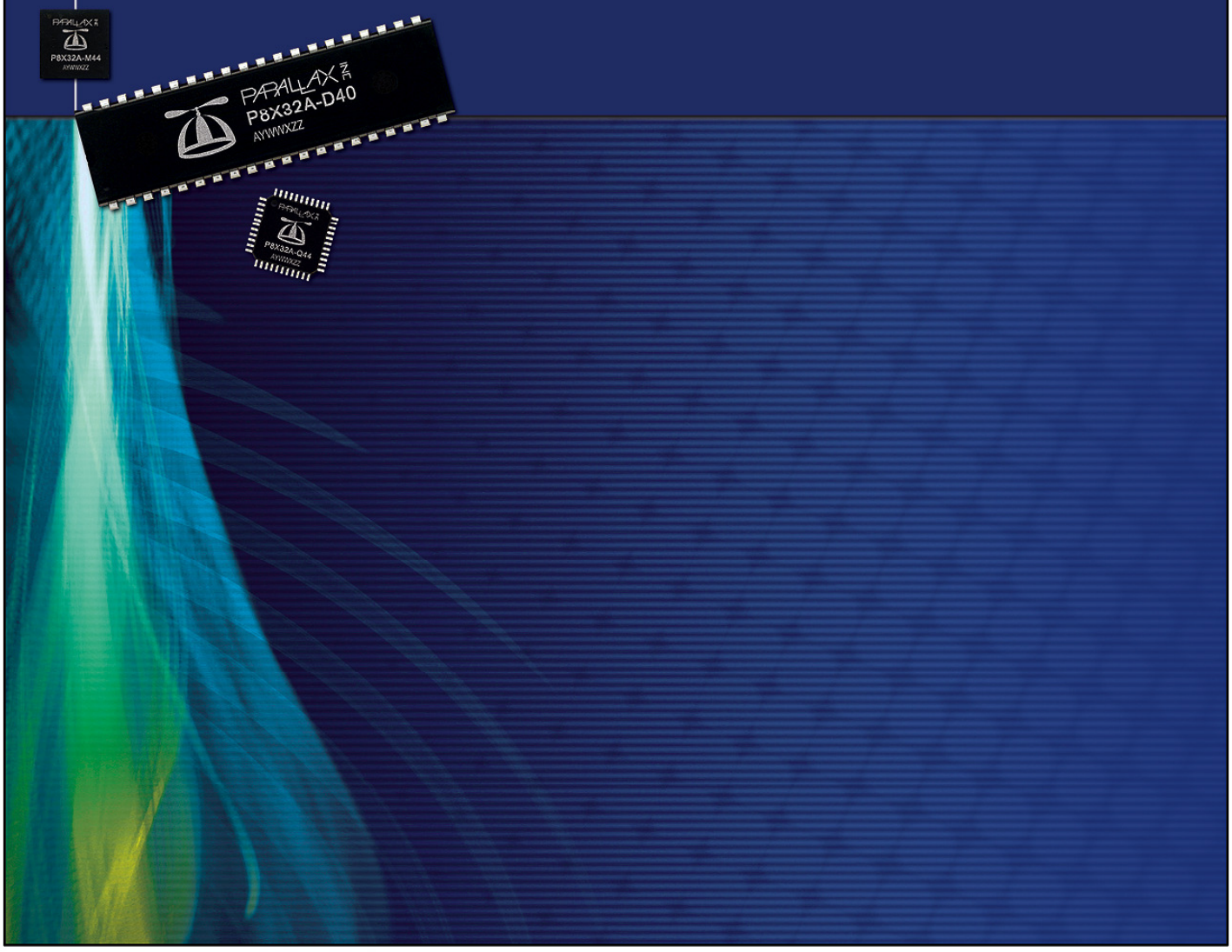
PropBASIC

by Terry Hitt



PropBASIC Syntax Guide

Version 0.11

December 16, 2009

Hitt Consulting

WARRANTY

No warranty, expressed or implied, for any purpose.

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 Design: *Jon McPhalen*
Documentation Contributors: *Terry Hitt, Jon McPhalen*

CONTENTS

About PropBASIC

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

1. 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.

2. 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
  LOW AlarmLED
  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.

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 data (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*}

Allows for the insertion of an error message in the compiled output listing and the termination of the compilation process.

'{$WARNING *Message*}

Allows for the insertion of 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.

```
Symbol          PIN     #{..#}  {[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. 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     16..23 LOW      ' make outputs and low
```

In the above example pins 16-23 (which correspond to the LEDs on the Propeller Demo Board) 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 connnections; this simplifies code changes to accommodate hardware modifications.

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:

```
  thePin = #TestPin
```

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

```
  testMask = @TestPin
```

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

Note: When passing a defined pin as a parameter to a subroutine or function the pin number (#pin) is used unless the @ (mask) modifier is specified in the call.

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
*		Multiplcation (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.

Subroutines and Functions

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)
' -----------------------------------------------------------------------


' -----------------------------------------------------------------------
' User Data (DATA, WDATA, LDATA, FILE)
' -----------------------------------------------------------------------
```

```
' ------------------------------------------------------------------------
' 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-1, otherwise the **BRANCH** instruction will be skipped.

- ✔ ***Offset*** is simple variable or array element.
- ✔ ***Labels*** specify the possible targets for the **BRANCH** instruction.

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 simplied 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*}}}}

DATA, WDATA, LDATA

```
{Label}          DATA    Value1 {, Value2 {, Value3...}}
{Label}          WDATA   Value1 {, Value2 {, Value3...}}
{Label}          LDATA   Value1 {, Value2 {, Value3...}}
```

DEC

DEC *Variable*

Function
Decrement (decrease) the value of *Variable* by one.

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

Explanation
The **DEC** instruction subtracts one to the specified variable. If the *Variable* holds 0 it will roll over to -1 after **DEC**.

```
Main:
  result = 4
  DEC result                            ' result is now 3
  result = 0
  DEC result                            ' 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* (decreses 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}
```

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.

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.

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

END

END

Function
Ends program execution.

Explanation
END prevents the PropBASIC program from executing any further instructions until thePropeller is reset (via RESn pin). **END** does not place the Propeller in low-power mode.

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 surpases *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 DATA element.

- ✔ ***HubSymbol*** is the variable or named DATA element in the hub
- ✔ ***Variable*** is the local variable that will hold the hub address of *HubSymbol*

Explanation

GETADDR is used to retrieve the hub address of a variable or DATA element for use with the RDxxxx and WRxxxx instructions. For example:

```
GETADDR buffer, bufpntr
```

In this example the address of *buffer*, a hub-based array, is placed in the local variable *bufpntr*. Knowing the address of the array and the number of elements in it the program is now able to manipulate data within the array.

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* statement.

✔ ***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*

Function

Increment (increase) value of *Variable* by one.

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

Explanation

The **INC** instruction adds one to the specified variable. If the *Variable* holds -1 (**$FFFF_FFFF**), it will roll over to zero after **INC**.

```
Main:
  result = 7
  INC result                            ' result is now 8
  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 (*Obsolete*)

{LET} *Variable* = [*Value* | *Expression*]

LOCKCLR

LOCKCLR *Value* {, *Variable*}

LOCKNEW

LOCKNEW *Variable*

LOCKRET

LOCKRET *Variable*

LOCKSET

LOCKSET *Value* {, *Variable*}

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 instruction (four clock cycles). Useful for allowing IO pins to settle after a change of state.

ON..GOSUB (*Obsolete*)

```
ON Index GOSUB Label0 {, Label1, Label2, ...}
```

ON..GOTO

ON *Index* GOTO *Label0* {, *Label1*, *Label2*, ...}

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 *DQPin, Variable{\Bits}*

OWRESET

OWRESET *DQPin* {, *StatusVar*}

OWWRITE

OWWRITE *DQPin, Value{\Bits}*

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 instruction: **PAUSEUS**

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 1_000
    Speaker = 1
    PAUSEUS 500
    Speaker = 0
    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 instruction: **PAUSE**

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     16..23  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 *HubVariable*, *LocalVariable*

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 to four values to the calling code. For example, the following function:

```
FUNC DOUBLE_IT
  __param1 = __param1 << 1                ' double by shifting left
  RETURN __param1
  ENDFUNC
```

...would be called like this:

```
  variable = DOUBLE_IT value
```


See the section on defining and using functions (page 16) 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).

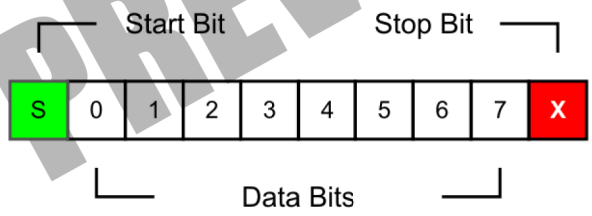
- ✔ ***Pin*** is variable or constant (0 to 31) that specifies the Propeller IO pin to use.
- ✔ ***BaudMode*** is a string constant that specifies serial timing and configuration. PropBASIC will raise an error if the baud rate specified exceeds the ability of the target **XIN/FREQ** setting.
- ✔ ***Variable***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

Note: Open baud modes are not implemented at this time.

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

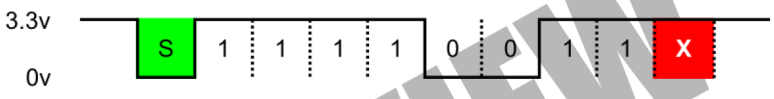
The SERIN instruction requires a *BaudMode* parameter which defines the baud rate (in bits per second) and the polarity with which the bits arrive.

There are two modes of serial reception:

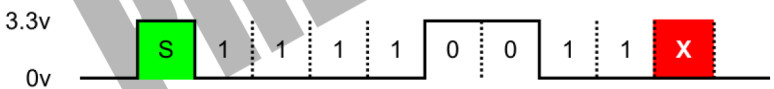
- ✔ ***True*** ("Txxxx")
- ✔ ***Inverted*** ("Nxxxx")

…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

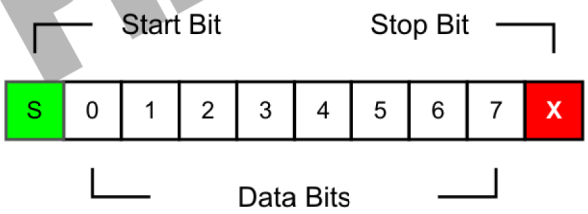
Transmit an asynchronous serial byte or string (e.g., RS-232).

- ✔ ***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

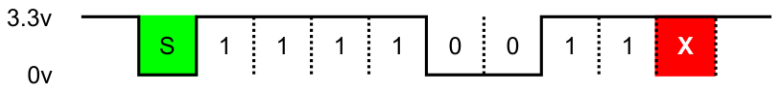
The SEROUT instruction requires a *BaudMode* parameter which defines the baud rate (in bits per second) and the mode in which the transmission pin is controlled. The mode actually defines two aspects of the output: signal polarity and how the transmission pin operates when sending a bit.

There are four modes of transmission:

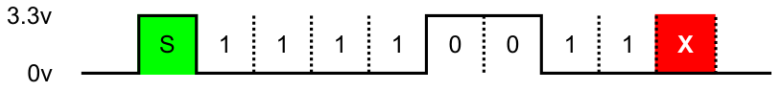
- ✔ ***True*** ("Txxxx")
- ✔ ***Inverted*** ("Nxxxx")
- ✔ ***Open-True*** ("OTxxxx")
- ✔ ***Open-Inverted*** ("ONxxxx")

…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:

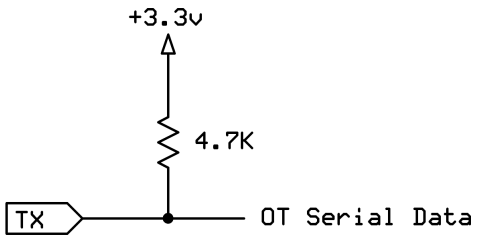


Inverted mode uses the opposite polarity; the line idle state is low, the start bit is high, data bits are inverted (low = 1, high = 0), and the stop bit is low. This is what $CF looks like when transmitting using *Inverted* mode:

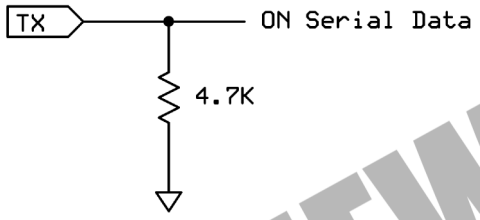


In both *True* and *Inverted* modes the Propeller drives the line high and low. When using a single pin to send and receive serial information an *Open* baud mode must be used. In these modes the Propeller drives the output pin in just one direction and relies on a pull-up (*Open-True*) or pull-down (*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, 2                    ' shell for SEROUT
DELAY_MS        SUB     1, 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*

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*

WAITPEQ

WAITPNE *State*, *Mask*

WAITPNE

WAITPNE *State*, *Mask*

WAITVID

WAITVID *Colors*, *Pixels*

WRBYTE, WRWORD, WRLONG

WRxxxx *HubVairable*, *Value*

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.