PropBasic 00.01.43

What is PropBasic ?

PropBasic is a BASIC compiler for the Parallax(c) Propeller microcontroller. It translates program code written in the BASIC computer language into Propeller assembly language instructions.

The Propeller microcontroller consists of eight 32-bit processors called COGs. Each cog has it's own 512 longs of memory. This cog ram must hold the PASM code that the cog is executing, and cog variables.

In a PropBasic program, the main code is run in one cog. And any TASKs define will be run in their own cog.

Inside the propeller is also 32K of ram that can be accessed by all cogs via the HUB. The HUB gives each cog access to the hub ram in sequence. Any time one cog needs to exchange information with another cog, it needs to use hub ram.

In PropBasic hub ram variables are accessed using RDxxxx and WRxxx to read and write to hub ram. xxxx may be BYTE, WORD or LONG.

It is important to keep straight the difference between COG memory and HUB memory. Variables declared with VAR exist in the COG memory and are directly addressable from any command. Variables declared with HUB or DATA exist in the HUB memory and are only accessible from specific commands. In other words you cannot perform math on HUB variables unless you first read them into VAR (cog) variables.

To invoke the compiler, you need to run the compiler .exe file with the complete path to your file as the first parameter in quotes.

For example:

```
PropBasic “c:\myfiles\myprog.pbas”
```

There are several option switches that may be used after the filename.

Switches:
- /Q = Quiet (No screen output)
- /P = Pause on warning or error (used to debug compiler)
- /B = Brief output (does not show source code)
- /O = "Output_Directory" Specifies a diffrent directory for output files
- /V = Returns Version number as exit code (exit immediately)
- /NS = No Code (Does NOT include the BASIC code in the output file)
- /VP = Compiling for ViewPort

For Example:

```
PropBasic “c:\myfiles\myprog.pbas” /p
```

Blink an LED

Usually to introduce any microcontroller language it is customary to show how to blink an LED. For this program we will assume you are using the Propeller demo board with LEDs connected to pins P16 through P23.

```
DEVICE P8X32A

LED  PIN 16 OUTPUT

PROGRAM Start

Start:
  TOGGLE LED
  PAUSE 1000
  GOTO Start
```

Let's go over each line. First we have:

```
DEVICE P8X32A
```

The device directive tells the compiler what controller we are using.

```
LED  PIN 16 OUTPUT
```

LED is a pin definition. It is a handy way to reference a pin number without having to remember what pin number you used though out the program. The OUTPUT modifier also tells the compiler that the pin is to be made an output at the start of the program. Normally all pins are inputs at startup.

```
PROGRAM Start
```

The program directive tells the compiler where your program is supposed to start executing.

```
Start:
```

The is a program label (program labels MUST have a colon after them). Labels define locations within a program.

```
TOGGLE LED
```

The toggle command will change the state of a pin. If the pin is high, the toggle command will make it low. If the pin is low, the toggle command will make it high.

```
PAUSE 1000
```

The pause command just waits for the specified number of milliseconds. So here we are waiting for 1000 milliseconds or 1 second.

```
GOTO Start
```

The goto command simple jumps to a new location in the program. Here we go back to the toggle command.

That's it. That is the whole program. If you run this program the LED will light for 1 seconds, then turn off for 1 second, then repeat over and over.

Type of variables:

In the propeller chip there are two types of RAM. There is COG RAM and HUB RAM.

COG RAM:
- 496 LONGs
- Can only be accessed in LONG format (not WORD or BYTE)
- Holds the program code (except for LMM code)
- Cannot be read or written of other COGs.
- Can perform operation on data directly.

HUB RAM:
- 32K Bytes
- Can be read as BYTE, WORD or LONG format
- Holds code until it is loaded into a COG, or executed using LMM.
- Is shared by all COGs.
- Data must be read into COG RAM before any operation can be performed.

Variables are allocated in COG RAM by using the VAR keyword. For example:

```
value VAR LONG
```

The only type of VAR variable is a LONG. An array can be created by specifying the size

```
many VAR LONG (10)
```

VAR arrays are not recommended because they use valuable code space.

Variables are allocated in HUB RAM byte using HUB or DATA. For example:

```
name    HUB STRING(30)
age     HUB BYTE
Message DATA "Hello There.", 0
```

Since "age" is a HUB variable, if we wanted to add 1 to it, we would have to read it into a VAR variable, add 1 to the VAR variable, then write it back to the HUB variable.

```
RDBYTE age, value
value = value + 1
WRBYTE age, value
```

Strings and data labels are passed to subroutines as their HUB address.

Data labels may be used as a string parameter. Data is really just a string that is preset.

Pin variables are names assigned to the propeller I/O pins. For example if you had an LED connected to pin 16 you might define

```
LED PIN 16 OUTPUT
```

The "output" modifier tell the compiler to make the pin an output when your code starts. Value options are "INPUT", "OUTPUT", "HIGH" and "LOW".

Pin variables may encompass multiple pins. If you have LEDs on pins 16 thru 23 (like the Propeller demo board) you might define

```
LEDs PIN 23..16 OUTPUT
```

Notice how we specified the higher pin number first. This is because in binary the more significant digits are on the left. If you define the pin variable with the lower pin number first, any values assigned to the pin variable will have their bit order reversed (this may be exactly what you want).

Strings may have the following embedded control characters:
\r = Carriage Return (13)
\n = Newline (10)
\\ = Backslash (92)
\" = Quote (34)
\123 = Chr(123) [must be 3 digits \000 = Chr(0)]
\x20 = Chr($20) [must be 2 hex digits]

Native versus LMM programs:

PropBasic can generate two different type of code. Native or LMM.

Native code is generated by default. When a native code program is started the code is loaded into a COG's RAM and is executed directly.

LMM code is generated by appending the word LMM to the PROGRAM command or the TASK command. When a LMM program is started a small “execution” program is loaded into the COG RAM with a pointer to the LMM code. The LMM code is read from HUB RAM one instruction at a time. That instruction is executed, then the next instruction is fetched and executed and so on.

Native code has the advantage of being about 5 times faster than LMM code. But it is limited to 496 PASM instruction.

LMM code has the advantage of allowing large programs to be created. Although they run about 5 times slower.

LMM code is also larger for a given set of PropBasic commands. This is because some instructions need extra data. For example a jump and call instruction use 2 LONGs instead of 1.

A single PropBasic program can have some TASKs that are native code and some that are LMM. It is fairly typical for the main program to be LMM, and the TASKs to be native code. Since TASK code tends to be smaller and in some cases needs to run fast (like video drivers).

When a CALL is used in LMM, the return address is stored on a stack that is maintained by the compiler. The stack default to 16 nested calls. But the size can be changed using the STACK directive. The size of the stack may be from 4 to 255. If the STACK directive is used it should be directly after the device directives (DEVICE and FREQ). For example:

```
DEVICE P8X32A, XTAL1, PLL16X
FREQ 80_000_000
STACK 8
```

Math Operators:

Unary Operators:

ABS	Returns the absolute value	`value1 = ABS value2`	8
LEN	Returns the length of a string	`value1 = LEN string1`	9
VAL	Returns the value of a string	`value1 = VAL string1`	10
GETADDR	Returns the address of a hub variable	`value1 = GETADDR string1`	11
SGN	Returns the sign of value 1, 0, -1	`value1 = SGN value2`	12
~	Returns the NOT of value	`value1 = ~value2`	13
-	Returns the negative of value	`value1 = -value2`	14

Binary Operators:

+	Addition	`value1 = value2 + value3`	15
-	Subtraction	`value1 = value2 - value3`	16
*****	Multiplication	`value1 = value2 * value3`	17
***/**	Multiply, shift 16-bits	`value1 = value2 */ value3`	18
******	Multiply, shift 32-bits	`value1 = value2 ** value3`	19
/	Division	`value1 = value2 / value3`	20
//	Remainder	`value1 = value2 // value3`	21
& AND	Bitwise AND	`value1 = value2 & value3` `value1 = value2 AND value3`	22
| OR	Bitwise OR	`value1 = value2 | value3` `value1 = value2 OR value3`	23
^ XOR	Bitwise XOR	`value1 = value2 ^ value3` `value1 = value2 XOR value3`	24
&~ ANDN	Bitwise AND NOT	`value1 = value2 &~ value3` `value1 = value2 ANDN value3`	25
MIN	Minimum of two values	`value1 = value2 MIN value3`	26
MAX	Maximum of two values	`value1 = value2 MAX value3`	27
>> SHR	Shift right	`value1 = value2 >> value3` `value1 = value2 SHR value3`	28
<< SHL	Shift left	`value1 = value2 << value3` `value1 = value2 SHL value3`	29

String Operators:

LEFT	Returns the left section of a string	`string1 = LEFT string2, count`	30
RIGHT	Returns the right section of a string	`string1 = RIGHT string2, count`	31
MID	Returns the middle of a string	`string1 = MID string2, start, count`	32
STR	Converts a value to a string	`string1 = STR value1,digits{,option}`	33
+	Concatenate two strings	`string1 = string2 + string3`	34

* Note that operators are ONLY allowed in assignment operation.
  You may need to use temporary variables to hold calculation needed for other commands.

* To deference a string use the system array __STRING(var). Note there are two underscores.
Strings are passed to subroutines as the location of the string in HUB RAM. Using __STRING(__paramx) allows subroutines to access the strings that were passed.

PropBasic Commands:

ABS

Returns the absolute value.

```
value1 = ABS value2
```

LEN

Returns the length of a string. The length of a string is the number of characters until a zero byte is found. The zero byte is NOT counted as part of the length. The string parameter may be a HUB STRING or a data label.

```
value1 = LEN string1
```

Related commands: LEFT, RIGHT, MID

VAL

Returns the value of a string.
If the string is a negative number, the minus sign MUST be the first character in the string.
The string may contain spaces. Spaces are evaluated as zero.
If the string contains any non-digit characters, the value will not be valid.

```
value1 = VAL string1
```

Related commands: STR

GETADDR

Returns the address of a hub variable.

Var = GetAddr hubVar{(offset)}

```
sharedValues HUB LONG(8)

valueAdr     VAR LONG
index        VAR LONG
temp         VAR LONG

valueAdr = GetAddr sharedValues(index)
RDLONG valueAdr, temp
```

Related commands: HUB, DATA, RDxxxx, WRxxxx

SGN

Returns the sign of value 1, 0, -1.

```
value1 = SGN value2
```

~

Returns the bitwise NOT of value. The ~ operator works on VAR variables as well as PIN variables.

```
value1 = ~value2
```

-

Returns the negative of value.

```
value1 = -value2
```

+

Addition

```
value1 = value2 + value3
```

Related commands: -

-

Subtraction

```
value1 = value2 - value3
```

Related commands: +

*****

Multiplication.

Multiplication is performed with a 64 bit result. The lowest 32-bits of the result are assigned.

```
value1 = value2 * value3
```

Related commands: */, **

***/**

Multiply, shift 16-bits

Multiplication is performed with a 64 bit result. The middle 32-bits of the result are assigned.

The */ operator is useful when you want to multiply by a fractional value greater than 1.
For example if you wanted to multiply a value by 1.5, you would use result = value */ 98304.
98304 is 1.5 * 65536

```
value1 = value2 */ value3
```

Related commands: *, **

**

Multiply, shift 32-bits

Multiplication is performed with a 64 bit result. The highest 32-bits of the result are assigned.

The ** operator is useful when you want to multiply by a fractional value less than 1.
For example if you wanted to multiply a value by 0.125, you would use result = value ** 536870912
536870912 is 0.125 * 65536 * 65536

```
value1 = value2 ** value3
```

Related commands: *, */

/

Division

```
value1 = value2 / value3
```

* Note: immediately after a division operation the remainder is available in the __Remainder variable.

Related commands: //

//

Remainder

```
value1 = value2 // value3
```

* Note: immediately after a division operation the remainder is available in the __Remainder variable.

Related commands: /

& AND

Bitwise AND.

```
value1 = value2 & value3
```

```
value1 = value2 AND value3
```

Related commands: OR, XOR, ANDN

| OR

Bitwise OR.

```
value1 = value2 | value3
```

```
value1 = value2 OR value3
```

Related commands: AND, XOR, ANDN

^ XOR

Bitwise XOR.

```
value1 = value2 ^ value3

value1 = value2 XOR value3
```

Related commands: AND, OR, ANDN

&~ ANDN

Bitwise AND NOT.

```
value1 = value2 &~ value3
```

```
value1 = value2 ANDN value3
```

Related commands: AND, OR, XOR

MIN

Returns the maximum of two values. Yes that's right the MAXIMUM of the two values. It makes more sense grammatically than it does mathmatically. "result = value MIN 5" means that result will always be at least 5.

```
value1 = value2 MIN value3
```

Related commands: MAX

MAX

Returns the minimum of two values. Yes that's right the MINIMUM of the two values. It makes more sense grammatically than it does mathmatically. "result = value MAX 100" means that result will always be less than or equal to 100.

```
value1 = value2 MAX value3
```

Related commands: MIN

>> SHR

Shift right. Each bit shifted right has the effect of dividing by 2.

```
value1 = value2 >> value3
```

```
value1 = value2 SHR value3
```

Related commands: << SHL

<< SHL

Shift left. Each bit shifted left has the effect of multiplying by 2.

```
value1 = value2 << value3
```

```
value1 = value2 SHL value3
```

Related commands: >> SHR

LEFT

Returns the left section of a string.

```
string1 = LEFT string2, count
```

Related commands: RIGHT, MID, LEN

RIGHT

Returns the right section of a string.

```
string1 = RIGHT string2, count
```

Related commands: LEFT, MID, LEN

MID

Returns the middle of a string. "count" characters are returned starting with character "start".

```
string1 = MID string2, start, count
```

Related commands: LEFT, RIGHT, LEN

STR

Converts a value to a string. If a signed option is used, the first character will be a "-" or a space.
If the value is larger than the number of digits specified, the first character will be corrupt.
Options 0 thru 3 will append a zero byte after the digits to form a single string, options 4 thru 7 do not.
For signed options, the sign counts as a digit.
The maximum digits is 11 for signed options and 10 for unsigned options.

```
string1 = STR value1,digits{,option}

Option:
  0 - Unsigned leading zeros, z-string
  1 - (default) Unsigned leading spaces, z-string
  2 - Signed leading zeros, z-string
  3 - Signed leading spaces, z-string
  4 - Unsigned leading zeros, no terminating zero
  5 - Unsigned leading spaces, no terminating zero
  6 - Signed leading zeros, no terminating zero
  7 - Signed leading spaces, no terminating zero
```

Related commands: VAL

+

Concatenate two strings.

```
string1 = string2 + string3
```

* Note: `string1 = string2 + string1` is not allowed.

\ Creates a single line of propeller assembly code.

```
\ pasm command
```

```
\ ROR myVar,#1
```

Related commands: ASM...ENDASM

'

Anything after an apostrophe is a comment and is ignored by the compiler.
Except directives that start with '{$

```
' comment
```

```
' This is a comment
temp = 100 ' This is a comment
```

Related commands: { }

{ }

Creates a multi-line comment

```
{ multi
line
comment }
```

```
{ This is a
multi-line
comment }
```

Related commands: '

_FREQ

Long Constant that holds the initially assigned clock frequency.

```
Rate VAR LONG
Rate = _FREQ / 8000
```

Related commands: FREQ

ASM...ENDASM

Creates a block of propeller assembly code.

```
ASM
  pasm instructions
ENDASM
```

```
ASM
  ROL value,#16
  RAR value,#16
ENDASM
```

Related commands: \

BREAK

Sets a break-point when using a debugger.

```
BREAK
```

Related commands: PROGRAM

BRANCH

Variable determines what label to jump to.

BRANCH var, label0, label1, label2[, label3[,etc]]

```
value VAR LONG

BRANCH value, Forward, Backward, Left, Right

Forward:
' Forward code
GOTO Done

Backward:
' Backward code
GOTO Done

Left:
' Left code
GOTO Done

Right:
' Right code
GOTO Done

Done:
```

Related commands: ON...GOTO

CLKSET

Sets the clock mode.

```
CLKSET mode,freq
```

```
CLKSET %0_0_0_00_001, 20_000    ' Set RCSLOW clock mode
```

Note: See the Propeller Manual for detailed information about CLKSET.

Note: The “freq” parameter is NOT used for PropBasic command timing.

Related commands: DEVICE, FREQ

COGID

Gets the cog ID of the cog running this command.

```
COGID var
```

```
value VAR LONG

COGID value    ' Get this cog's ID
COGSTOP value  ' Stop this cog
```

COGINIT

Initializes a cog with a task. The cog ID must be provided.

```
COGINIT taskname, value
```

```
FlashLED TASK

PROGRAM START

Start:
  COGINIT FlashLED, 1 ' Start task in COG 1
  PAUSE 10_000        ' Let task run for 10 seconds
  COGSTOP 1           ' Stop the task
END

TASK FlashLED
  LED PIN 16 LOW
  DO
    TOGGLE LED
    PAUSE 100
  LOOP
```

COGSTART

Starts a task in a new cog. The next available cog is used.
If a var is given it will be set to the cogID that was used, or 8 if no cog was free.

```
COGSTART taskname{,var}
```

COGSTOP

Stops a cog. If no cogid is provided, the current cog is stopped.

```
COGSTOP {value}
```

* COGINIT differs from COGSTART in that COGSTART uses the next available cog. With COGINIT you must specify what cog to use.

CON

Creates a named constant, with a value or a string.

name CON value

```
MyCon CON 1000
Grade CON "F"
Baud  CON "T115200"
```

COUNTERA / **COUNTERB**

Setup hardware counter parameters.

COUNTERA mode{, apin {, bpin{, frqx{, phsx}}}}

```
COUNTERA 40, 0, 1, 80_000
```

Mode:
0 = Counter Disabled
8 = PLL Internal (Video) *
16 = PLL Single-Ended *
24 = PLL Differential *
32 = NCO/PWM Single Ended – frqx is added to phsx each system clock; apin = phsx[31]
40 = NCO/PWM Differential – frqx is added to phsx each system clock; apin=phsx[31]; bpin=!phsx[31]
48 = DUTY Single-Ended – frqx is added to phsx each system clock; apin=carry
56 = DUTY Differential – frqx is added to phsx each system clock; apin=carry; bpin=!carry
64 = POS detector - frqx is added to phsx each system clock when apin is high
72 = POS detector with feedback - frqx is added to phsx each system clock when apin is high (1)
80 = POSEDGE detector - frqx is added to phsx each system clock when apin goes from low to high
88 = POSEDGE detector with feedback - frqx is added to phsx each system clock when apin goes from low to high
96 = NEG detector - frqx is added to phsx each system clock when apin is low
104 = NEG detector with feedback - frqx is added to phsx each system clock when apin is low (1)
112 = NEGEDGE detector - frqx is added to phsx each system clock when apin goes from high to low
120 = NEGEDGE detector with feedback - frqx is added to phsx each system clock when apin goes from high to low (1)
128 = LOGIC never – Counter off
136 = LOGIC !A & !B - frqx is added to phsx each system clock when apin is low AND bpin is low
144 = LOGIC A & !B - frqx is added to phsx each system clock when apin is high AND bpin is low
152 = LOGIC !B - frqx is added to phsx each system clock when bpin is low
160 = LOGIC !A & B - frqx is added to phsx each system clock when apin is low AND bpin is high
168 = LOGIC !A - frqx is added to phsx each system clock when apin is low
176 = LOGIC A <> B - frqx is added to phsx each system clock when apin is not equal to bpin
184 = LOGIC !A | !B - frqx is added to phsx each system clock when apin is low OR bpin is low
192 = LOGIC A & B - frqx is added to phsx each system clock when apin is high AND bpin is high
200 = LOGIC A = B - frqx is added to phsx each system clock when apin is equal to bpin
208 = LOGIC A - frqx is added to phsx each system clock when apin is high
216 = LOGIC A | !B - frqx is added to phsx each system clock when apin is high OR bpin is low
224 = LOGIC B - frqx is added to phsx each system clock when bpin is high
232 = LOGIC !A | B - frqx is added to phsx each system clock when apin is low OR bpin is high
240 = LOGIC A | B - frqx is added to phsx each system clock when apin is high OR bpin is high
248 = LOGIC always

* For PLL modes add:
0 = VCO / 128 (/8)
1 = VCO / 64 (/4)
2 = VCO / 32 (/2)
3 = VCO / 16 (x1)
4 = VCO / 8 (x2)
5 = VCO / 4 (x4)
6 = VCO / 2 (x8)
7 = VCO / 1 (x16)

* Even if "bpin" is not used it still must be specified. You may use zero.

(1) bpin is set to the state apin was in LAST clock cycle

DATA,WDATA,LDATA Creates data values in HUB ram. DATA = BYTE, WDATA=WORD, LDATA=LONG

```
[label] DATA value1[,value2[,value3[,etc]]]]
```

```
BitMask DATA 1,2,4,8,16
```

```
Message DATA "This is a message.", 0
```

Data labels MUST be on the same line as the DATA command. And there is no colon after a data label.
Data labels may be used in place of a string for command and functions.

Related commands: FILE

DEC

Subtract 1 (or any value) from a variable.

DEC varname{, value}

```
cntr VAR LONG
DEC cntr
DEC cntr, 4
```

Related commands: INC, DJNZ

DEVICE

Sets device type and parameters.

DEVICE deviceID, {settings{,settings}}

```
DEVICE P8X32A, XTAL1, PLL16X
```

deviceID: only P8X32A is supported

settings: RCSLOW, RCFAST, XINPUT, XTAL1..3, PLLX2, PLLX4, PLLX8, PLLX16

Related commands: FREQ, XIN

DJNZ

Decrease variable and jump to label if not zero.

DJNZ var, label

```
LED   PIN 16 LOW
value VAR LONG

value = 100

Again:
  HIGH LED
  PAUSE 100
  LOW LED
  PAUSE 100
DJNZ value, Again
```

Related commands: DEC, DO...LOOP

DO...LOOP

```
DO WHILE var cond value
LOOP

DO
LOOP UNTIL var cond value

DO
LOOP ' always loops

DO
LOOP var ' Loops var times, var = 0 when finished
```

END

```
Ends program execution. Puts cog in low-power mode.

END
```

```
END
```

EXIT

Ends the current DO...LOOP or FOR...NEXT loop.

```
EXIT
IF var cond value THEN EXIT
```

FILE

Loads a binary data file. The contents are read like DATA.

```
{label} FILE "MyFile.bin"
```

```
Message FILE "MyFile.TXT" ' file contains the text HELLO
```

Related commands: DATA

FOR...TO...STEP...NEXT

```
FOR var = startvalue TO endvalue
  ' Code
NEXT

FOR var = startvalue TO endvalue STEP deltavalue
  ' Code
NEXT
```

Related commands: DJNZ

FREQ

```
Sets device frequency after pll multiplier.

FREQ freq

FREQ 80_000_000

Do not use FREQ and XIN together, use one or the other
```

Related commands: _FREQ

FUNC...ENDFUNC

Creates a named function. Returns 1 LONG value.

name FUNC [minParams[,maxParams]]

```
FUNC name
  ...
ENDFUNC
```

Parameters are passed in __paramx variables.

If a variable number of parameters is specified, the parameter count is given in the __paramcnt variable.

If a hub variable/label/string is used as a parameter, it's ADDRESS is passed. The system array __STRING(__paramx) can be used to access a string parameter.

If a pin variable is used as a parameter, the pin NUMBER is passed.

```
Calc FUNC 1

myVar = Calc 1

FUNC Calc
  __param1 = __param1 + 1

  RETURN __param1
ENDFUNC
```

Related commands: SUB...ENDSUB

GOSUB

Jump to a subroutine.

GOSUB subroutine

```
Calc SUB

GOSUB Calc

SUB Calc
  ' Code
  RETURN value
ENDSUB
```

ONLY named subroutines can be used with GOSUB, GOSUB is optional.

Related commands: SUB...ENDSUB

GOTO

Jump to a label.

GOTO label

```
GOTO Start
```

HIGH

Makes a pin an output and high.

HIGH pinname | const

```
LED PIN 0 OUTPUT

HIGH LED
HIGH 3
```

Related commands: LOW, TOGGLE, INPUT, OUTPUT

HUB

Creates HUB variables. Access via GETADDR, RDBYTE, RDWORD, RDLONG, WRBYTE, WRWORD, WRLONG

name HUB type [= value]
name HUB type(elements) [= value]

```
myVar  HUB LONG = 100_000
myVars HUB LONG(8) = 0
```

type: BYTE, WORD, LONG, STRING(length)

Use RDBYTE, RDWORD, RDLONG to read value from HUB variables.
Use WRBYTE, WRWORD, RDLONG to write value to HUB variables.

For an array, all elements are pre-initialized to the same value.
If you need the elements to contain different values, then use DATA instead.

```
myVar  HUB LONG(4) = 0  ' All elements are set to zero
myVars LDATA 0, 1, 2, 3 ' Elements have unique values
```

Related commands: VAR, DATA

I2CREAD

Reads a byte from the I2C bus.

```
I2CREAD SDAPin, SCLPin, var, ackbitvalue
```

I2SPEED

Sets the clock speed for I2C operations.

```
I2CSPEED multipier
```

* “multiplier” may be a floating point value
  A value of 2 would make I2C operations twice as fast as normal.
  A value of 0.5 would make I2C operations half as fast as normal.

I2CSTART

```
Sends an I2C start condition.

I2CSTART SDAPin, SCLPin
```

I2CSTOP

```
Sends an I2C stop condition.

I2CSTOP SDAPin, SCLPin
```

I2CWRITE

```
Writes a byte to the I2C bus.

I2CWRITE SDAPin, SCLPin, value[, ackbitvar]
```

IF...ELSE|ELSEIF...ENDIF

```
IF var cond value THEN label

IF var cond value THEN
  ' code
ENDIF

IF var cond value THEN
  ' code
ELSE
  ' code
ENDIF

IF var cond value THEN
  ' code
ELSEIF var cond value THEN
  ' code
ELSE
  ' code
ENDIF
```

IF...OR|AND

```
IF var cond value OR
 var cond value THEN
  ' Code
ELSE
  ' Code
ENDIF

IF var cond value OR
 var cond value AND
 var cond value THEN
  ' Code
ELSE
  ' Code
ENDIF
```

INC

Adds 1 (or any value) to a variable.

INC varname{,value}

```
cntr VAR LONG

INC cntr
INC cntr, 4
```

Related commands: DEC

INCLUDE

Includes propeller assembly code from a separate file.

```
INCLUDE "MyFile.spin"
```

Related commands: LOAD, FILE

INPUT

Makes a pin an input.

INPUT pinname | const

```
switch PIN 1 INPUT

INPUT switch
INPUT 0
```

Related commands: OUTPUT, LOW, HIGH, TOGGLE

LET

Optional

LOAD

```
      Load PropBasic code from a separate file.

      LOAD "MyFile.pbas"
```

Related commands: INCLUDE

LOCKCLR

Clears a lock ID.
If a second parameter is given, it will hold the previous lock state.

```
LOCKCLR value{,var}
```

LOCKNEW

Retreives a new lock ID.

```
LOCKNEW var
```

LOCKRET

Returns a lock ID.

```
LOCKRET var
```

LOCKSET

Sets a lock ID.
If a second parameter is given, it will hold the previous lock state.

```
LOCKSET value{,var}
```

LOW

Makes a pin an output and low.

LOW pinname | const

```
LED PIN 16 OUTPUT

LOW LED
LOW 4
```

Related commands: HIGH, INPUT, OUTPUT, TOGGLE, REVERSE

NOP

No operation. Does nothing. Uses 1 instruction.

```
NOP
```

ON...GOTO

Jump to label based on value of a variable.

ON *var* GOTO *label1*, *label2* [, *label3*, [, *etc*]]

ON *var* = *value1*, *value2*, *value3* GOTO *label1*, *label2*, *label3*

ON...GOSUB

Same as ON...GOTO except does a subroutine jump.

ON *var* GOSUB *label1*, *label2* [, *label3*, [, *etc*]]

ON *var* = *value1*, *value2*, *value3* GOSUB *label1*, *label2*, *label3*

OUTPUT

Makes a pin an output.

```
OUTPUT pinname | const
```

```
LED PIN 1 OUTPUT

OUTPUT LED
OUTPUT 1
```

Related commands: INPUT, HIGH, LOW, TOGGLE, REVERSE

OWREAD

Reads a byte from the 1-wire buss.

```
OWREAD DQPin, var{\bits}
```

OWRESET

Sends a reset on the 1-wire buss.

```
OWRESET DQPin{,statusVar}
```

OWWRITE

Writes a byte to the 1-wire buss.

```
OWWRITE DQPin, value{\bits}
```

PAUSE

Pauses for milliseconds. Can use fractional values.

```
PAUSE value
```

```
PAUSE 1000
PAUSE 27.6
```

PAUSEUS

Pauses for microseconds. Can use fractional values.

```
PAUSEUS value
```

```
PAUSEUS 1000
PAUSEUS 4.7
```

Related commands: WAITCNT

PIN

Creates a pin variable. #name = pin number, @name = pin mask

name PIN pinnumber [modifier]

```
LED PIN 0 LOW
```

name PIN MSB*pin..LSBpin* [*modifier*]

```
LEDS  PIN 23..16 LOW 'Normal bit order #LEDS gives LSBpin (16)

LEDSR PIN 16..23 LOW 'Reverse bit order #LEDS gives MSBpin (16)
```

modifiers: INPUT, OUTPUT, HIGH, LOW

modifier is only used for the task that defines the pin.

A pin with an output modifier (OUTPUT, HIGH, LOW) will be an input in all other tasks. This is because the all the cog's pin outputs are OR'd together. If you had a pin defined as HIGH, and started another cog, the new cog would hold the pin high and no other cog would be able to change the pin state.

PROGRAM

Sets program start label and main code options.

```
PROGRAM Start {LMM|PASD}
```

The LMM parameter causes the compiler to generate LMM code instead of native PASM code.
LMM code runs slower, but allows much larger programs.
The PASD parameter enables use of the PASD debugger.

PULSIN

Measure incoming pulse width in microseconds.

PULSIN *pin*, *state*, *resultVar*

NOTE: If the clock frequency is less than 20MHz, the result is still in microseconds but the granularity is greater than 1. For example when using RCSLOW (20KHz) the result will always be a multiple of 1000.

```
' This program reads the distance from a PING sensor connected to pin 2.
' Converts the value to tenths of inches and sends the distance to the PC.
'
DEVICE P8X32A, XTAL1, PLL16X
FREQ 80_000_000

Baud      CON "T115200" ' Baud rate to communicate with PC

PingPin   PIN 2 LOW     ' Connected to Sig pin on Ping module
TX        PIN 30 HIGH   ' Send data back to PC

value     VAR LONG

Message  DATA "Distance is "
valueStr DATA "1234.5 inches.", 13, 0

PROGRAM Start

Start:
  DO
    PAUSE 10
    PULSOUT PingPin, 5             ' Trigger PING
    PAUSEUS 5
    PULSIN PingPin, 1, value       ' Measure PING pulse
    value = value ** 291_198_783   ' Convert to tenths of inches (* 0.0678)
    valueStr = STR value, 5, 5     ' Convert value to ASCII
    RDBYTE valueStr(4), value      ' Insert decimal point
    WRBYTE valueStr(4), ".", value
    SEROUT TX, Baud, Message
  LOOP
END
```

Related commands: PULSOUT

PULSOUT

Create a pulse of specified width. Duration is in microseconds. Always pulses pin even if duration is zero.

```
PULSOUT pin, duration
```

NOTE: If the clock frequency is less than 20MHz, the duration is still in microseconds but the granularity is greater than 1. For example when using RCSLOW (20KHz) the duration will be divided by 1024, then that many 1024uSec delays will take place. Here is a table showing the granularity of different clocks:

20MHz and higher = 1uSec
10Mhz to 19.999Mhz = 2uSec
5MHz to 9.999MHz = 4uSec
2.5MHz to 4.999MHz = 8uSec
1.25Mhz to 2.499MHz = 16uSec
20KHz = 1024uSec

Related commands: PULSIN

RANDOM

Creates a random number from a seed variable.

```
RANDOM seedvar[, copyvar]
```

RCTIME

Measures time (in microseconds) for pin to reach "state" level.

```
RCTIME pin, state, resultvar
```

Related commands: PULSIN

RDBYTE

Reads the value of a BYTE hub variable or DATA.

```
RDBYTE bytehubvar{(offset)}, var{,var{,var{,etc}}}
```

RDSBYTE

Reads the value of a signed BYTE hub variable or DATA.

```
RDSBYTE bytehubvar{(offset)}, var{,var{,var{,etc}}}
```

RDLONG

Reads the value of a LONG hub variable or LDATA.

```
RDLONG longhubvar{(offset)}, var{,var{,var{,etc}}}
Note: longhubvar lowest two bits must be zero (long aligned)
```

RDWORD

Reads the value of a WORD hub variable or WDATA.

```
RDWORD wordhubvar{(offset)}, var{,var{,var{,etc}}}
Note: wordhubvar lowest bit must be zero (word aligned)
```

RDSWORD

Reads the value of a signed WORD hub variable or WDATA.

```
RDSWORD wordhubvar{(offset)}, var{,var{,var{,etc}}}
Note: wordhubvar lowest bit must be zero (word aligned)
```

“offset” is in WORDs for RDWORD and RDSWORD
“offset” is in LONGs for RDLONG and RDSLONG

Problems can arise if you use RDWORD to read byte data. Or use RDLONG to read word or byte data. The problem is that the data may not be aligned properly.

In the Propeller chip WORD data is word aligned (lowest bit of the address must be zero), and LONG data is long aligned (lowest two bits of the address must be zero).

```
label1 LDATA 1000
label2 DATA 100
label3 LDATA 2000
```

There will be three bytes not used between label2 and label3 to make sure that "label3 LDATA" is long aligned.

Related commands: WRBYTE, WRWORD, WRLONG

RETURN

Return from a subroutine or function.

```
RETURN value{,value{,value{,value}}}
```

The first value specified (__param1) will be automatically assigned to the destination variable. Additional values will be held in the __param2, __param3, etc variables after the function returns.

```
RETURN 1
```

Related commands: GOSUB, SUB...ENDSUB

REVERSE

Reverse pin direction (input / output)

```
REVERSE pinname | const
```

```
sensor PIN 1

REVERSE sensor
REVERSE 2
```

Related commands: HIGH, LOW, INPUT, OUTPUT, TOGGLE

SERIN

Serial input. Prefix baud value "T" for true mode, "N" for inverted mode.

If SERIN times-out var is not changed. If label is not specified execution continues with the next line of code.

If "var" is a string, characters are stored until a carrage return is received, timeout is only in effect until the first character is received.

```
SERIN pin, baud, var {, timeoutms{, label}}
```

Related commands: SEROUT

SEROUT

Serial output. "T" for true mode, "N" for inverted mode. "O" = Open

```
SEROUT pin, [T | N | OT | ON]baud, char | string
```

Related commands: SERIN

SHIFTIN

SPI input.

SHIFTIN *datapin*, *clockpin*, *mode*, *var*[*bits*][,speed]

If the bits parameter is not specified, 8 bits are received.

mode: LSBPRE, LSBPOST, MSBPRE, MSBPOST

```
' This program will read channel 0 from the MCP3204 chip and
'  send the value to a terminal program running on the PC.
'
' Set terminal program to 115200 baud.
'
DEVICE P8X32A, XTAL1, PLL16X
XIN 5_000_000

ADC_Clk  PIN 2 LOW  ' MCP3204.11
ADC_Dout PIN 3 LOW  ' MCP3204.10
ADC_Din  PIN 4 LOW  ' MCP3204.9
ADC_CS   PIN 5 HIGH ' MCP3204.8

inValue  VAR LONG
ascii    HUB STRING(10)

PROGRAM Start

Start:
  DO
    LOW ADC_CS    ' Enable MCP3204
    PAUSEUS 100
    SHIFTOUT ADC_Din, ADC_Clk, MSBFIRST, %11000\5  ' Select CH0, Single-Ended
    SHIFTIN ADC_Dout, ADC_Clk, MSBPOST, inValue\13 ' Read ADC
    HIGH ADC_CS   ' Disable ADC
    LOW ADC_Clk
    ascii = STR inValue, 4
    ascii = ascii + 13  ' Add a carrage return
    SEROUT 30, T115200, ascii
    PAUSE 1
  LOOP
END
```

Related commands: SHIFTOUT

SHIFTOUT

SPI output.

```
SHIFTOUT datapin, clockpin, mode, value[\bits][,speed]
```

If the bits parameter is not specified, 8 bits are sent.
mode: LSBFIRST, MSBFIRST

Related commands: SHIFTIN

SUB...ENDSUB

Creates a named subroutine with parameters.

name SUB [*minParams*[,*maxParams*]]

Parameters are passed in __paramx variables.

If a variable number of parameters is specified, the parameter count is given in the __paramcnt variable.

If a hub variable/label is used as a parameter, it's ADDRESS is passed.

If a pin variable is used as a parameter, the pin NUMBER is passed.

```
SUB name
  ...
ENDSUB
```

```
SetDAC SUB 1

SetDAC 1

SUB SetDAC
  ' code to set DAC
ENDSUB
```

Related commands: FUNC, ENDFUNC

TASK...ENDTASK

Creates code that runs in a separate cog.

```
name TASK {LMM} {AUTO}

TASK name
  ...
ENDTASK
```

If LMM is specified the compiler will generate LMM code instead of native PASM code. LMM code runs slower, but allows much larger programs.

If AUTO is specified, the TASK is automatically launched at startup

Task code runs in a separate cogs.

VAR variables are not shared between cogs.

SUBs and FUNCs are not shared between cogs.

HUB variables, PINs and DATA are shared between cogs.

Use COGSTART or COGINIT to start tasks.

TOGGLE

Toggles pin state (high / low)

```
TOGGLE pinname | const
```

```
LED PIN 1 OUTPUT

TOGGLE LED
TOGGLE 5
```

Related commands: HIGH, LOW, REVERSE, INPUT, OUTPUT

VAR

Creates a variable. Only LONG types are supported. Arrays are supported.

```
name VAR LONG
name VAR LONG(elements)

myVar  VAR LONG
myVar2 VAR LONG(8)
```

Note: Since VAR arrays are stored in COG ram, they use up valuable code space. Consider using HUB arrays when possible.

Related commands: HUB

WAITCNT

Waits for the system counter to reach the target value. Then adds the delta value to the variable.

```
WAITCNT target, delta
```

Related commands: PAUSE, PAUSEUS

WAITPEQ

Waits for a pin (or set of pins) state to equal a mask value.

`WAITPEQ` *`state`*`,` *`mask`*
INA is anded with “mask” then compared to “state”.

WAITPNE

Waits for a pin (or set of pins) state to NOT equal a mask value.

`WAITPNE` *`state`*`,` *`mask`*
INA is anded with “mask” then compared to “state”.

WAITPEQ and WAITPNE are typically used to pause until a pin has reached a certain state. For example:

```
WAITPEQ myPin, myPin  ' Wait to pin "myPin" to go HIGH
WAITPNE myPin, myPin  ' Wait for pin "myPin" to go LOW
```

WAITVID

Waits for the video serializer to be able to accept new data.

```
WAITVID colors, pixels
```

WATCH

When using a debugger, this updates the variables in the debugger.

```
WATCH
```

WRBYTE

Writes a new value into a BYTE hub variable.

```
WRBYTE bytehubvar{(offset)}, value{, value{,value{, etc}}}
```

WRLONG

Writes a new value into a LONG hub variable.

```
WRLONG longhubvar{(offset)}, value{, value{,value{, etc}}}
```

WRWORD

Writes a new value into a WORD hub variable.

```
WRWORD wordhubvar{(offset)}, value{, value{,value{, etc}}}
```

“offset” is in WORDs for WRWORD
“offset” is in LONGs for WRLONG

Related commands: RDBYTE, RDLONG, RDWORD

XIN

Crystal frequency before pll multiplier

```
XIN freq
```

```
XIN 5_000_000
```

Do not use FREQ and XIN together, use one or the other

Related commands: FREQ

General

Literal values are assumed decimal, but can be prefixed to indicate a different base:

$	Hexidecimal 0..9, A..F
%%	Quaternary 0..3
%	Binary 0..1
"x"	Ascii character
#	Floating Point (only constants supported)

Math operators can only be used when assigning values to a variable.

Math operators cannot be used in commands.

Only 1 math operator can be used per line.

Only LONG vars are supported. LONG arrays are also supported.

Using a variable as an array index generates alot more code. Try to avoid this if possible.

HUB vars can be BYTE, WORD or LONG. Arrays are supported.

HUB vars can ONLY be accessed with RDBYTE, WRBYTE, RDWORD, WRWORD, RDLONG, WRLONG commands.

Be aware that HUB vars must be address aligned by the size. So if you declare a BYTE then a LONG, there will be three wasted address location between them.

PINs, HUB vars and DATA are global to all COGs (tasks).

VARs, SUBs and FUNCs are local only to the TASK they are declared in.

TASK code generates a separate .spin file.

DATA must be declared before the program code. You cannot put the DATA after the program code.

The main code runs in COG 0.

Compiler Directives

Compiler directives are available for conditional compilation.
By default the device name is defined.

```
'{$DEFINE name}
'{$UNDEFINE name}
'{$USES name}

'{$IFDEF name}
'{$IFNDEF name}
'{$IFFREQ condition value}
'{$IFUSED name}
'{$IFNUSED name}

'{$ELSE}
'{$ENDIF}
'{$WARNING message}
'{$ERROR message}

'{$CODE}
'{$TASKS}
```

The IFUSED directive tells the compiler if a subroutine or function has been used.

The USES directive tells the compiler that a pin or long constant is used in a task and that it should not be stripped out. Usually this is used when you have some embedded PASM code that uses a pin or long constant. USES is not needed in normal PropBasic code because the compiler automatically marks the subroutine as used if it is called.

```
'{$USES subName}

'{$IFUSED subName}
SUB subName
' put code for subroutine here
ENDSUB
'{$ENDIF}


'{$WARNING message}
'{$ERROR message}
```

Example:

```
'{$IFNDEF P8X32A}
'{$ERROR This program requires a P8X32A chip}
'{$ENDIF}
```

The CODE and TASKS directives are used in library code. These directives separate the definitions from the code and task sections of the library.

Tips and Tricks

Remember that PropBasic is a single pass compiler. When compiling a line of code the compiler has no idea about what comes after it. So you cannot do things like try to use a variable before it is defined.

Using shifts for multiply and divide

Understanding the */ and ** operators:

When performing multiplication PropBasic performs 32-bit * 32-bit = 64-bit math. Normally only the lower 32-bits of the result are used with the normal multiply operator (*). However, if you want you can access the 32 middle bits (bits 16 to 48) using the */ operator. Or the 32 highest bits using the ** operator. So basically the */ operator does a multiply by the value given, then does a divide by 65536. The ** operator does the multiply by the value given, then does a divide by 4294967296.

```
      value1 = value2 */ 81920 ' 81920 = 1.25 * 65536
      value1 = value2 */ 205887 ' 205887 = Pi * 65536
```

Alignment of different data sizes:

In the propeller data stored in the hub must be aligned according to it's length. WORD data must be word aligned, and LONG data must be long aligned. This can cause problems if you use (for example) RDLONG to read byte data.

__RAM Virtual array:

__STRING virtual array:

When a string (literal or a string variable) is used as a subroutine parameter, what is actually passed to the subroutine is the LOCATION of the string in HUB memory. The location is of course a long variable stored in the __paramx variables. If you want to use any of the commands or functions that have string parameters, you need to use __STRING(__paramx).

Like so:

```
Trim  SUB 2 ' Trim string, length

SUB Trim
  __STRING(__param1) = LEFT __STRING(__param1), __param2
ENDSUB
```