FlexBASIC

Introduction

FlexBASIC is the BASIC language support of the fastspin compiler for the Parallax Propeller and Prop2. It is a BASIC dialect similar to FreeBASIC or Microsoft BASIC, but with a few differences. On the Propeller chip it compiles to LMM code (machine language) which runs quite quickly.

fastspin recognizes the language in a file by the extension. If a file has a ".bas" extension it is assumed to be BASIC. Otherwise it is assumed to be Spin.

Preprocessor

fastspin has a pre-processor that understands basic directives like `#include`, `#define`, and`#ifdef / #ifndef / #else / #endif`.

Directives

DEFINE

```
#define FOO hello
```

Defines a new macro `FOO` with the value `hello`. Whenever the symbol `FOO` appears in the text, the preprocessor will substitute `hello`.

Note that unlike the C preprocessor, this one cannot accept arguments. Only simple defines are permitted.

If no value is given, e.g.

```
#define BAR
```

then the symbol is defined as the string `1`.

IFDEF

Introduces a conditional compilation section, which is only compiled if the symbol after the `#ifdef` is in fact defined. For example:

```
#ifdef __P2__
'' propeller 2 code goes here
#else
'' propeller 1 code goes here
#endif
```

IFNDEF

Introduces a conditional compilation section, which is only compiled if the symbol after the `#ifndef` is *not* defined.

ELSE

Switches the meaning of conditional compilation.

ELSEIFDEF

A combination of `#else` and `#ifdef`.

ELSEIFNDEF

A combination of `#else` and `#ifndef`.

ERROR

Prints an error message. Mainly used in conditional compilation to report an unhandled condition. Everything after the `#error` directive is printed. Example:

```
#ifndef __P2__
#error This code only works on Propeller 2
#endif
```

INCLUDE

Includes a file. The contents of the file are placed in the compilation just as if everything in that file was typed into the original file instead. This is often used

```
#include "foo.h"
```

Included files are searched for first in the same directory as the file that contains the `#include`. If they are not found there, then they are searched for in any directories specified by a `-I` or `-L` option on the command line. If the environment variable `FLEXCC_INCLUDE` is defined, that gives a directory to be searched after command line options. Finally the path `../include` relative to the fastspin executable binary is checked.

WARN

`#warn` prints a warning message; otherwise it is similar to `#error`.

UNDEF

Removes the definition of a symbol, e.g. to undefine `FOO` do:

```
#undef FOO
```

Predefined Symbols

There are several predefined symbols:

Symbol	When Defined
`__propeller__`	always defined to 1 (for P1) or 2 (for P2)
`__propeller2__`	only defined if compiling for Propeller 2
`__P2__`	obsolete version of `__propeller2__`
`__FLEXBASIC__`	always defined to the fastspin version number
`__FASTSPIN__`	if the `fastspin` front end is used
`__SPINCVT__`	always defined to the fastspin version number
`__SPIN2PASM__`	if –asm is given (PASM output) (always defined by fastspin)
`__SPIN2CPP__`	if C++ or C is being output (never in fastspin)
`__cplusplus`	if C++ is being output (never in fastspin)

Language Syntax

Comments

Comments start with `rem` or a single quote character, and go to the end of line. Note that you need a space or non-alphabetical character after the `rem`; the word `remark` does not start a comment. The `rem` or single quote character may appear anywhere on the line; it does not have to be the first thing on the line.

There are also multi-line comments, which start with `/'` and end with `'/`.

Examples:

```
rem this is a comment
' so is this
print "hello" ' this part is a comment too
/' here is a multi
   line comment '/
```

Integers

Decimal integers are a sequence of digits, 0-9.

Hexadecimal (base 16) integers start with the sequence “&h”, “0h”, or “0x” followed by digits and/or the letters A-F or a-f.

Binary (base 2) integers start with the sequence “&b” or “0b” followed by the digits 0 and 1.

Numbers may contain underscores anywhere to separate digits; those underscores are ignored.

For example, the following are all ways to represent the decimal number `10`:

```
10
1_0
0xA
&h_a
&B1010
```

Keywords

Keywords are always treated specially by the compiler, and no identifier may be named the same as a keyword.

```
abs
and
any
as
asm
__builtin_alloca
byte
case
catch
class
close
const
continue
cpu
data
declare
def
defint
defsng
delete
dim
direction
do
double
else
```

```
end
endif
enum
exit
for
function
gosub
goto
if
input
integer
let
long
loop
mod
next
new
nil
not
open
option
or
output
pointer
print
program
ptr
put
read
rem
restore
return
select
self
shared
short
single
sqrt
step
sub
then
throw
to
try
type
ubyte
```

```
uinteger
ulong
until
ushort
using
var
wend
while
with
word
xor
```

Variable, Subroutine, and Function names

Names of variables, subroutines, or functions (“identifiers”) consist of a letter or underscore, followed by any sequence of letters, underscores, or digits. Names beginning with an underscore are reserved for system use. Case is ignored; thus the names `avar`, `aVar`, `AVar`, `AVAR`, etc. all refer to the same variable.

Identifiers may have a type specifier appended to them. `$` indicates a string variable or function, `%` an integer variable or function, and `#` a floating point variable or function. The type specifier is part of the name, so `a$` and `a#` are different identifiers (the first is a string variable and the second is a floating point variable). If no type specifier is appended, the identifier is assumed to be an integer. This may be overridden with the `defsng` directive, which specifies that variables starting with certain letters are to be assumed to be single precision floating point.

Variable or function types may also be explicitly given, and in this case the type overrides any implicit type defined by the name. However, we strongly recommend that you not use type specifiers like `$` for variables (or functions) that you give an explicit type to.

Examples:

```
dim a%            ' defines an integer variable
dim a#            ' defines a different floating point variable
dim a as string   ' defines another variable, this time a string
dim a$ as integer ' NOT RECOMMENDED: overrides the $ suffix to make an integer variable

'' this function returns a string and takes a float and string as parameters
function f$(a#, b$)
  ...
end function

'' this function also returns a string from a float and string
function g(a as single, b as string) as string
```

```
  ...
end function
```

Arrays

Arrays must be declared with the `dim` keyword. FlexBASIC supports only one and two dimensional arrays, but they may be of any type. Examples of array declarations:

```
rem an array of 10 integers
rem note that dim gives the last index
dim a(9)
rem same thing but more verbose
dim c(0 to 9) as integer
rem an array of 10 strings
dim a$(9)
rem another array of strings
dim d(9) as string
rem a two dimensional array of strings
dim g$(9, 9)
```

Arrays are by default indexed starting at 0. That is, if `a` is an array, then `a(0)` is the first thing in the array, `a(1)` the second, and so on. This is similar to other languages (such as Spin and C), where array indexes start at 0. The value given in the `dim` is the last array index. This is different from Spin and C, where arrays are declared with their sizes rather than last array index.

Code to initialize an array to 0 could look like:

```
dim a(9) as integer
sub zero_a
  for i = 0 to 9
    a(i) = 0
  next i
end sub
```

It is possible to change the array base by using

```
option base 1  ' make arrays start at 1 by default
```

The array definition may have an explicit lower bound given, for example:

```
dim a(1 to 10)  ' array of 10 items
dim b(0 to 10)  ' array of 11 items
```

This only works for one dimensional arrays. For two dimensional arrays both dimensions must have the same lower bound, and both must start from the default set by the last `option base`.

Note that pointer dereferences (using array notation) always use the last value set for `option base` in the file, since we cannot know at run time what the actual base of the pointed to object was. So it is best to set this just once.

Global, Member, and Local variables.

There are three kinds of variables: global variables, member variables, and local variables.

Global (shared) variables may be accessed from anywhere in the program, and exist for the duration of the program. They are created with the `dim shared` declaration, and may be given an initial value. For example,

```
dim shared x = 2
```

creates a global variable with a value of 2.

A global variable is shared by all instances of the object that creates it. For example, if "foo.bas" contains

```
dim shared ctr as integer

function set_ctr(x)
  ctr = x
end function
function get_ctr()
  return ctr
end function
function inc_ctr()
  ctr = ctr + 1
end function
```

then a program like:

```
dim x as class using "foo.bas"
dim y as class using "foo.bas"

x.set_ctr(0)
y.set_ctr(1)
print y.get_ctr()
y.inc_ctr()
print x.get_ctr()
```

will print 1 and then 2, because `x.ctr` and `y.ctr` are the same (shared) global variable.

Member variables, on the other hand, are unique to each instance of a class. They are created with regular `dim` outside of any function or subroutine. If we modified the sample above to remove the `shared` from the declaration of `ctr`, then the program would print 1 and then 0, because the `y.inc_ctr()` invocation would not affect the value of `x.ctr`.

Member variables are automatically initialized to 0, and may not be initialized to any other value.

Local variables are only available inside the function or subroutine where they are declared, and only exist for as long as that function or subroutine is running. When the routine returns, the variables lose any values they had at the time. They are re-created afresh the next time the function is called. Local variables may be initialized to values, but this initialization is done at run time so it has some overhead.

Extending lines

It is possible to extend a long expression or array initializer over several lines. To do this, add a single `_` immediately before the end of the line. This causes the compiler to treat the end of line like a space rather than an end of line. For example:

```
  x = y + _
      z
```

is parsed like `x = y + z`. This is especially useful for array initializers, which can often be quite long:

```
dim shared as integer a(5) = { _
  1, 2, 3, _
  4, 5 _
  }
```

Note that only shared arrays may be initialized like this.

Multiple statements per line

Generally speaking, you may place multiple statements on one line if you separate them with a colon (`:`). For example, these two bits of code are the same:

```
   x = 1
   y = 2
```

and

```
   x = 1 : y = 2
```

Language features

Types

There are a number of data types built in to the FlexBASIC language.

Unsigned integer types

`ubyte`, `ushort`, and `uinteger` are the names for 8 bit, 16 bit, and 32 bit unsigned integers, respectively. The Propeller load instructions do not sign extend by default, so `ubyte` and `ushort` are the preferred names for 8 and 16 bit integers on the Propeller.

Signed integer types

`byte`, `short`, and `integer` are 8 bit, 16 bit, and 32 bit signed integers.

Floating point types

`single` is, by default, a 32 bit IEEE floating point number. There is an option to use a 16.16 fixed point number instead; this results in much faster calculations, but at the cost of much less precision and range.

`double` is reserved for future use as a double precision (64 bit) floating point number, but this is not implemented yet.

Pointer types

Pointers to any other type may be declared. `T pointer` is a pointer to type `T`. Thus `ushort pointer` is a pointer to an unsigned 16 bit number, and `ubyte pointer pointer` is a pointer to a pointer to an unsigned 8 bit number.

String type

The `string` type is a special pointer. Functionally it is almost the same as a `const ubyte pointer`, but there is one big difference; comparisons involving a string compare the pointed to data, rather than the pointer itself. For example:

```
sub cmpstrings(a as string, b as string)
  if (a = b) then
    print "strings equal"
  else
    print "strings differ"
  end if
end sub
```

```
sub cmpptrs(a as const ubyte pointer, b as const ubyte pointer)
  if (a = b) then
    print "pointers equal"
  else
    print "pointers differ"
  end if
end sub

dim x as string
dim y as string

x = "hello"
y = "he" + "llo"
cmpstrings(x, y)
cmpptrs(x, y)
```

will always print “strings equal” followed by “pointers differ”. That is because the `cmpstrings` function does a comparison with strings (so the contents are tested) but `cmppointers` does a pointer comparison (and while the pointers point at memory containing the same values, they are located in two distinct regions of memory and hence have different addresses.

Classes

FlexBASIC supports classes, which are similar to records or structs in other languages. There are two ways to define classes. A whole BASIC (or Spin, or C) file may be included as a class with the `using` keyword:

```
dim ser as class using "FullDuplexSerial.spin"
```

declares the variable `ser` as a class, using the Spin variables and methods from the given file. This also works for `.bas` or `.c` files. Any functions declared in the file become methods of the new class.

Classes may also be declared directly, with the variables and methods of the class specified between
`class` and `end class`

```
class counter
  dim as integer c
  sub inc()
    c = c + 1
  end sub
  function get() as integer
    return c
  end function
end class
```

```
dim x as counter
...
x.inc
print x.get()
```

Note that `end class` must be spelled out in full (unlike many `end x` pairs which may be abbreviated as just `end`).

Type Aliases

An alias for an existing type may be declared with the `type` keyword. For example:

```
type numptr as integer pointer
type fullduplexserial as class using "FullDuplexSerial.spin"
```

Function declarations

Function names follow the same rules as variable names. Like variable names, function names may have a type specifier appended, and the type specifier gives the type that the function returns.

```
function Add(a, b)
  return a+b
end function
```

This could be written more verbosely as

```
function Add(a as integer, b as integer) as integer
  return a+b
end function
```

It is often useful for documentation to explicitly specify all types like this, even when the default types specified by the variable names would work.

Memory allocation

FlexBASIC supports allocation of memory and garbage collection. Memory allocation is done from a small built-in heap. This heap defaults to 256 bytes in size on Propeller 1, and 4096 bytes on Propeller 2. This may be changed by defining a constant `HEAPSIZE` in the top level file of the program.

Garbage collection works by scanning memory for pointers that were returned from the memory allocation function. As long as references to the original pointers returned by functions like `left$` or `right$` exist, the memory will not be re-used for anything else.

Note that a CPU ("COG" in Spin terms) cannot scan the internal memory of other CPUs, so memory allocated by one CPU will only be garbage collected by that same CPU. This can lead to an out of memory situation even if in fact there is memory available to be claimed. For this reason we suggest that all allocation of temporary memory be done in one CPU only.

new and delete

The `new` operator may be used to allocate memory. `new` returns a pointer to enough memory to hold objects, or `nil` if not enough space is available for the allocation. For example, to allocate 40 bytes one can do:

```
var ptr = new ubyte(40)
if ptr then
  '' do stuff with the allocated memory
  ...
  '' now free it (this is optional)
  delete ptr
else
  print "not enough memory"
endif
```

The memory allocated by `new` is managed by the garbage collector, so it will be reclaimed when all references to it have been removed. One may also explicitly free it with `delete`.

String functions

String functions and operators like `left$`, `right$`, and `+` (string concatenation) also work with allocated memory. If there is not enough memory to allocate for a string, these functions/operators will return `nil`.

Function pointers

Pointers to functions require 8 bytes of memory to be allocated at run time (to hold information about the object to be called). So for example in:

```
'' create a Spin FullDuplexSerial object
dim ser as class using "FullDuplexSerial.spin"
'' get a pointer to its transmit function
var tx = @ser.tx
```

the variable `tx` holds a pointer both to the `ser` object and to the particular method `tx` within it. Since this is dynamically allocated, it is possible for the `@` operator to fail and return `nil`.

__builtin_alloca

Instead of `new`, which allocates persistent memory on the heap, it is possible to allocate temporary memory with the `__builtin_alloca` operator. Memory allocated in this way may only be used during the lifetime of the function which allocated it, and may not be returned from that function or assigned to a global variable. Almost always it is better to use `new` than `__builtin_alloca`, but the latter is more efficient (but dangerous, because the pointer becomes invalid after the function that uses `__builtin_alloca` exits).

Templates

FlexBASIC supports polymorphic programming via templates. These are like parameterized function or class declarations. Only function templates are supported at this time.

Templates are introduced by the keyword `any` followed by a parenthesized list of identifiers which are the types to be subsituted in the declaration. That is, each identifier in the list represents a type, which may vary at compile time.

Function Templates

A function to find the smaller of two items with the same type `t`, which can be string, integer, single, or any other type that supports the `<` operator, may be declared as:

```
any(t) function mymin(x as t, y as t) as t
  if x < y then
    return x
  else
    return y
  end if
end function
```

This declares a family of functions `mymin__T`, where `T` can be any type. Whenever the compiler sees `mymin(some_expr)` it checks the type of `some_expr` and changes the function call to `mymin__T(some_expr)`. So for example:

```
   print mymin(1.7, 2.4), mymin("zzz", "aaa")
```

will print `1.7` and `aaa`.

Propeller Hardware Features

Input, Output, and Direction

For the Propeller we have some special pseudo-variables `direction`, `input`, and `output`. These may be used to directly control pins of the processor. For example, to set pin 1 as output and then set it high do:

```
direction(1) = output
output(1) = 1
```

Similarly, to set pin 2 as input and read it:

```
direction(2) = input
x = input(2)
```

Pin Ranges

Ranges of pins may be specified with `hi,lo` or `lo,hi`. The first form is preferred; if you do

```
output(2, 0) = x
```

then the bottom 3 bits of x are copied directly to the first 3 output pins. If you use the other form

```
output(0, 2) = x     ' note: x is reversed!
output(0, 2) = &b110 ' sets bits 0 and 1 to 1, and bit 2 to 0
```

then the lower 3 bits are reversed; this is useful if you're directly coding a binary constant, but
otherwise is probably not what you want.

Hardware registers

The builtin Propeller hardware registers are available with their usual names, unless they are redeclared. For example, the OUTA register is available as "outa" (or "OUTA", or "Outa"; case does not matter).

The hardware registers are not keywords, so they are not reserved to the system. Thus, it is possible to use `dim` to declare variables with the same name. Of course, if that is done then the original hardware register will not be accessible in the scope of the variable name.

Alphabetical List of Keywords and Built In Functions

ABS

```
y = abs x
```

Returns the absolute value of x. If x is a floating point number then so will be the result; if x is an unsigned number then it will be unchanged; otherwise the result will be an Integer.

AND

```
a = x and y
```

Returns the bit-wise AND of x and y. If x or y is a floating point number then it will be converted to integer before the operation is performed.

Also useful in boolean operations. The comparison operators return 0 for false conditions and all bits set for true conditions, so you can do things like:

```
if (x < y AND x = z) then
  ' code that runs if both conditions are true
end if
```

ANY

```
dim x as any
```

Declares x as a generic 32 bit variable compatible with any other type. Basically this is a way to treat a variable as a raw 32 bit value. Note that no type checking at all is performed on variables declared with type `any`, nor are any conversions applied to them. This means that the compiler will not be able to catch many common errors.

`any` should be used only in exceptional circumstances.

Example: a subroutine to print the raw bit pattern of a floating point number:

```
sub printbits(x as single)
  dim a as any
  dim u as uinteger
  '' just plain u=x would convert x from single to unsigned
  '' instead go through an ANY type, which will do no conversion
  a = x
  u = a
  print u
end sub
```

AS

`as` is a keyword that introduces a type for a function, function parameter, or dimensioned variable.

```
  ' declare a function with an integer parameter that returns a string
  function f(x as integer) as string
  ...
```

ASC

```
  i = ASC(s$)
```

returns the integer (ASCII) value of the first character of a string. If the argument is not a string it is an error.

ASM

Introduces inline assembly. The block between ASM and END ASM is parsed slightly differently than usual; in particular, instruction names are treated as reserved identifiers.

Inside inline assembly any instructions may be used, but the only legal operands are integer constants, registers, and local variables (or parameters) to the function which contains the inline assembly. Labels may be defined, and may be used as the target for `goto` elsewhere in the function. Any attempt to leave the function, either by jumping out of it or returning, will cause undefined behavior. In other words, don't do that!

If you need temporary variables inside some inline assembly, `dim` them as locals in the enclosing function.

Example: to implement a wait (like the built-in `waitcnt`:

```
sub wait_until_cycle(x as uinteger)
  asm
    waitcnt x, #0
  end asm
end sub
```

__BUILTIN_ALLOCA

Allocates memory on the stack. The argument is an integer specifying how much memory to allocate. For example:

```
   dim as integer ptr x = __builtin_alloca(256)
```

creates an array of 64 integers (which needs 256 bytes) and makes `x` into a pointer to it.

The pointer returned from `__builtin_alloca` will become invalid as soon as the current function returns (or throws an exception), so it should never be assigned to a global variable or be returned from the function.

`__builtin_alloca` is awkward to work with, and dangerous. Almost always you should use `new` instead. The only advantages of `__builtin_alloca` is that it is slightly more efficient than `new`, and does not use up heap space (but uses stack space instead).

BYTE

A signed 8 bit integer, occupying one byte of computer memory. The unsigned version of this is `ubyte`. The difference arises with the treatment of the upper bit. Both `byte` and `ubyte` treat 0-127 the same, but for `byte` 128 to 255 are considered equivalent to -128 to -1 respectively (that is, when a `byte` is copied to a larger sized integer the upper bit is repeated into all the other bits; for `ubyte` the new bytes are filled with 0 instead).

CASE

Used in a `select` statement to indicate a possible case to match. Only a subset of FreeBasic's `case` options are available. After the `case` can either be `else` (which always matches), a single expression (which matches if the original expression equals the `case` one), or an inclusive range `a to b` which will match if the original expression is between between `a` and `b` (inclusive).

Example:

```
select case x
case 1
  print "it was 1"
case 2 to 4
  print "it was between 2 and 4"
  print "sorry for being vague!"
case 8
  print "it was 8"
case else
  print "it was something else"
end select
```

All of the statements between the `case` and the next `case` (or `end select`) are executed if the `case` is the first one to match the expression in the `select case`.

CAST

Used to convert between types. `cast(type1, expr)` will calculate `expr` and then convert it to type `type1`. This could involve calculation (if `expr` has an integer type, for example, and `type1` is `single` then the bit pattern of `expr` is changed) or could just mean a different way of interpreting the bits in a value.

For example, to get a pointer to the Propeller 1 LOG table, located in ROM at address `0xC000`, you could do:

```
   dim logptr as ushort ptr
   logptr = cast(ushort ptr, 0xC000)
```

CATCH

Used in a `try` statement to indicate the start of an error handling block.

CHR$

Not actually a reserved word, but a built-in function. Converts an ascii value to a string (so the reverse of ASC). For example:

```
print chr$(65)
```

prints `A` (the character whose ASCII value is 65)

CLASS

A `class` is an abstract collection of variables and functions. If you've used the Spin language, a class is like a Spin object.

Class Using

Spin objects may be directly imported as classes:

```
#ifdef __P2__
   dim ser as class using "spin/SmartSerial"
#else
   dim ser as class using "spin/FullDuplexSerial"
#endif
```

creates an object `ser` based on the Spin file "SmartSerial.spin" (for P2) or "FullDuplexSerial"; this may then be used directly, e.g.:

```
   ser.str("hello, world!")
   ser.tx(13)  ' send a carriage return
   ser.dec(100) ' print 100 as a decimal number
```

BASIC files may also be used as classes. When they are, all the functions and subroutines in the BASIC file are exposed as methods (there are no private methods in BASIC yet). Any BASIC code that is not in a function or subroutine is not accessible.

Abstract classes

Another way to define an object is to first declare an abstract `class` with a name, and then use that name in the `dim` statement:

```
' create abstract class fds representing Spin FullDuplexSerial
' NOTE: use SmartSerial.spin instead if trying on P2
class fds using "FullDuplexSerial.spin"
' create a variable of that type
dim ser as fds
```

This is more convenient if there are many references to the class, or if you want to pass pointers to the class to functions.

Inline Classes

Finally, the functions, subroutines, and variables associated with a class may be defined directly inline, between the `class` and a finishing `end class`. In this case the class name may be used as a type name. For example:

```
class counter
  dim x as integer

  sub reset
    x = 0
  end sub

  sub inc(n = 1)
    x = x + n
  end sub

  function getval()
    return x
  end function
end class

dim cnt as counter

cnt.reset
cnt.inc
cnt.inc
print cnt.getval() ' prints 2
```

```
  cnt.inc
  print cnt.getval() ' prints 3
```

Interoperation with Spin

Using Spin objects with `class using` is straightforward, but there are some things to watch out for:

- Spin does not have any notion of types, so most Spin functions will return type `any` and take parameters of type `any`. This can cause problems if you expect them to return something special like a pointer or float and want to use them in the middle of an expression. You can either use explicit `cast` operations, or assign the results of Spin methods to a typed variable, and then use that variable in the expression instead.
- Spin treats strings differently than BASIC does. For example, in the Spin expression `ser.tx("A")`, `"A"` is an integer (a single element list). That would be written in BASIC as `ser.tx(asc("A"))`. Conversely, in Spin you have to write `ser.str(string("hello"))` where in BASIC you would write just `ser.str("hello")`.

Interoperation with C

C files may be used as classes, but there are some restrictions. BASIC and Spin are both case insensitive languages, which means that the BASIC symbols `AVariable`, `avariable`, and `AVARIABLE` are all the same, and all are translated internally to `avariable`. In C the case of identifiers matters. This makes accessing C symbols from BASIC somewhat tricky. Only C symbols that are all lower case may be accessed from BASIC.

CLKSET

```
  clkset(mode, freq, xsel)
```

Propeller built in function. On the P1, this acts the same as the Spin `clkset` function. On P2, this does two `hubset` instructions, the first to set the oscillator and the second (after a short delay) to actually enable it. The `mode` parameter gives the setup value for the oscillator, and the second hubset to enable the oscillator uses `mode + xsel` as its parameter. If `xsel` is omitted, it defaults to `3`.

For example:

```
  clkset(0x010c3f04, 160_000_000)  ' set P2 Eval board to 160 MHz
```

After a `clkset` it is usually necessary to call `_setbaud` to reset the serial baud rate correctly.

Also note that no sanity check is performed on the parameters; it is up to the programmer to ensure that the frequency actually matches the mode on the board being used.

CLOSE

Closes a file previously opened by `open`. This causes the `closef` function specified in the device driver (if any) to be called, and then invalidates the handle so that it may not be used for further I/O operations. Any attempt to use a closed handle produces no result.

```
close #2  ' close handle #2
```

Note that handles 0 and 1 are reserved by the system; closing them may produce undefined results.

CONST

At the beginning of a line, `const` declares a constant value. For example:

```
const x = 1, y = 2.0
```

declares x to be the integer 1 and y to be the floating point value 2.0. Only numeric values (integers and floats) may be declared with `const`.

Inside a type name, `const` signifies that variables of this type may not be modified. This is mainly useful for indicating that pointers should be treated as read-only.

```
sub trychange(s as const ubyte ptr)
  s(1) = 0  '' illegal, s points to const ubytes
  if (s(1) = 2) then '' OK, s may be read
    print "it was 2"
  end if
end sub
```

CONTINUE

Used to resume loop execution early. The type of loop (FOR, DO, or WHILE) may optionally be given after CONTINUE. However, note that only the innermost containing loop may be continued. This is different from FreeBasic, where for example `continue for` may be placed in a `while` loop that is itself inside a `for` loop. In FlexBasic this will produce an error.

Example:

```
for i = 1 to 5
  if (i = 3) then
    continue for
  end if
  print i
next i
```

will print 1, 2, 4, and 5, but will skip the 3 because the `continue for` will cause the next iteration of the `for` loop to start as soon as it is seen.

The example above could be written more succinctly as:

```
for i = 1 to 5
  if i = 3 continue
  print i
next
```

CPU

Used to start a subroutine running on another CPU. The parameters are the subroutine call to execute, and a stack for the other CPU to use. For example:

```
' blink a pin at a given frequency
sub blink(pin, freq)
  direction(pin) = output
  do
    output(pin) = not output(pin)
    waitcnt(getcnt() + freq)
  loop
end sub
...
dim stack(8) ' small stack, blink does not call many other functions

' start the blinking up on another CPU
var a = cpu(blink(LED, 80_000_000), @stack(1))
```

Note that `cpu` is not a function call, it is a special form which does not evaluate its arguments in the usual way. The first parameter is actually preserved and called in the context of the new CPU.

`cpu` returns the CPU id ("cog id") of the CPU that the new function is running on. If no free CPU is available, `cpu` returns -1.

DATA

Introduces raw data to be read via the `read` keyword. This is usually used for initializing arrays or other data structures. The calculations for converting values from strings to integers or floats are done at run time, so consider using array initializers instead (which are more efficient).

In contrast to some other BASICs, no parsing at all is done of the information following the `data` keyword; it is simply dumped into memory as a raw string. Subsequent `read` commands will read the bytes from memory and convert them to the appropriate type, as if they were `input` by the user.

Unlike most other statements, the `data` statement always extends to the end of the line; any colons (for example) within the data are treated as data.

```
dim x as integer
dim y as string
dim z as single
read x, y, z
print x, y, z
data 1.1, hello
data 2.2
```

will print `1` (`x` is an integer, so the fractional part is ignored), `hello`, and `2.2000`.

The order of `data` statements matters, but they may be intermixed with other statements. `data` statements should only appear at the top level, not within functions or subroutines.

DECLARE

Keyword reserved for future use.

DEF

Define a simple function. This is mostly intended for porting existing BASIC code, but could be convenient for creating very simple functions. The syntax consists of the function name, parameter list, `=`, and then the return value from the expression. All of the types are inferred from the names. So for example to define a function `sum` to return the sum of two integers we would do:

```
DEF sum(x, y) = x+y
```

DEFINT

Dictates the default type for variable names starting with certain letters.

```
defint i-j
```

says that variables starting with the letters `i` through `j` are assumed to be integers.

The default setting is `defint a-z` (i.e. all variables are assumed to be integer unless given an explicit suffix or type in their declaration). A combination of `defsng` and `defint` may be used to modify this.

DEFSNG

Dictates the default type for variable names starting with certain letters.

```
defsng a-f
```

says that variables starting with the letters `a` through `f` are assumed to be floating point.

The default setting is `defint a-z` (i.e. all variables are assumed to be integer unless given an explicit suffix or type in their declaration). A combination of `defsng` and `defint` may be used to modify this.

Putting `defsng a-z` at the start of a file may be useful for porting legacy BASIC code.

DELETE

Free memory allocated by `new` or by one of the string functions (`+`, `left$`, `right$`, etc.).

Use of `delete` is a nice hint and makes sure the memory is free, but it is not strictly necessary since the memory is garbage collected automatically.

DIM

Dimension variables. This defines variables and allocate memory for them. `dim` is the most common way to declare that variables exist. The simplest form just lists the variable names and (optionally) array sizes. The variable types are inferred from the names. For example, you can declare an array `a` of 10 integers, a single integer `b`, and a string `c$` with:

```
dim a(10), b, c$
```

It's also possible to give explicit types with `as`:

```
dim a(10) as integer
dim b as ubyte
dim s as string
```

Only one explicit type may be given per line (this is different from FreeBASIC). If you give an explicit type, it will apply to all the variables on the line:

```
' this makes all the variables singles, despite their names
' (probably NOT a good idea!)
dim a(10), b%, c$, d as single
```

If you want to be compatible with FreeBASIC, put the `as` first:

```
dim as single a(10), b%, c$, d
```

Variables declared inside a function or subroutine are "local" to that function or subroutine, and are not available outside or to other functions or subroutines. Variables dimensioned at the top level may be used by all functions and subroutines in the file.

See also VAR.

DIRECTION

Pseudo-array of bits describing the direction (input or output) of pins. In Propeller 1 this array is 32 bits long, in Propeller 2 it is 64 bits.

```
  direction(2) = input ' set pin 2 as input
  direction(6,4) = output ' set pins 6, 5, 4 as outputs
```

Note that pin ranges may not cross a 32 bit boundary; that is,

```
  direction(33, 30) = input
```

is illegal and produces undefined behavior.

DO

Main loop construct. A `do` loop may have the loop test either at the beginning or end, and it may run the loop while a condition is true or until a condition is true. For example:

```
  do
    x = input(9)
  loop until x = 0
```

will wait until pin 9 is 0.

See also WHILE.

DOUBLE

The type for a double precision floating point number. `double` is not actually implemented in the compiler, and is treated the same as `single`.

ELSE

See IF

END

Used to mark the end of most blocks. For example, `end function` marks the end of a function declaration, and `end if` the end of a multi-line `if` statement. In most cases the name after the `end` is optional.

ENDIF

Marks the end of a multi-line `if` statement. Same as `end if`.

ENUM

Reserved for future use.

EXIT

Exit early from a loop, function, or subroutine.

Just plain `exit` on its own will exit early from the innermost enclosing loop, and will produce an error if given outside a loop.

The `exit` may also have an explicit `do`, `for`, or `while` after it to say what kind of loop it is exiting. In this case the innermost loop must be of the appropriate type. This is different from FreeBasic, where for example `exit while` may be used in a `for` loop that is inside a `while` loop; we do not allow that.

Finally `exit function` and `exit sub` are synonyms for `return`.

EXIT DO

Exit from the innermost enclosing loop if it is a `do` loop. If it is not a `do` loop then the compiler will print an error.

EXIT FOR

Exit from the innermost enclosing loop if it is a `for` loop. If it is not a `for` loop then the compiler will print an error.

EXIT FUNCTION

Returns from the current function (just like a plain `return`). The value of the function will be the last default value established by assigning a value to the function's name, or 0 if no such value has been established. For example:

```
function sumif(a, x, y)
  sumif = x + y
  if (a <> 0)
    exit function
  sumif = 0
end function
```

returns `x+y` if a is nonzero, and 0 otherwise.

EXIT SUB

Returns from the current subroutine. Same as the `return` statement.

EXIT WHILE

Exit from the innermost enclosing loop if it is a `while` loop. If it is not a `while` loop then the compiler will print an error.

FOR

Repeat a loop while incrementing (or decrementing) a variable. The default step value is 1, but if an explicit `step` is given this is used instead:

```
' print 1 to 10
for i = 1 to 10
  print i
next i
' print 1, 3, 5, ..., 9
for i = 1 to 10 step 2
  print i
next i
```

If the variable given in the loop is not already defined, it is created as a local variable (local to the current sub or function, or to the implicit program function for loops outside of any sub or function).

FUNCTION

Defines a new function. The type of the function may be given explicitly with an `as` *type* clause; if no such clause exists the function's type is deduced from its name. For example, a function whose name ends in `$` is assumed to return a string unless an `as` is given.

Functions have a fixed number and type of arguments, but the last arguments may be given default values with an initializer. For example,

```
function inc(n as integer, delta = 1 as integer) as integer
  return n + delta
end function
```

defines a function which adds two integers and returns an integer result. Since the default type of variables is integer, this could also be written as:

```
function inc(n, delta = 1)
  return n+delta
end function
```

In this case because the final argument `delta` is given a default value of 1, callers may omit this argument. That is, a call `inc(x)` is exactly equivalent to `inc(x, 1)`.

Anonymous functions

`function` may also be used in expressions to specify a temporary, unnamed function. There are three forms for this. The long form is very similar to ordinary function declarations. For example, suppose we want to define a function "plusn" which itself returns a function which adds one to its argument. This would look like:

```
' define an alias for the type of a function which takes an integer
' and returns another; this isn't strictly necessary, but saves typing
type intfunc as function(x as integer) as integer

' plusn(n) returns a function which adds n to its argument
function plusn(n as integer) as intfunc
  return function(x as integer) as integer
           return x + n
     end function
end function

dim as intfunc f, g
f = plusn(1) ' function which returns 1 + its argument
g = plusn(7) ' function which returns 7 + its argument

' this will print 1 2 8
print 1, f(1), g(1)
```

The long anonymous form is basically the same as an ordinary function definition, but without the function name. The major difference is that an explicit definition of the return type (e.g. `as integer`) is required, since the compiler cannot use a name to determine a default type for the function.

For simple functions which just return a single expression, an abbreviated anonymous form is available. This omits the return type, which is determined by the expression itself, and puts the expression on the same line. This means we could write the `plusn` function above as:

```
function plusn(n as integer) as intfunc
  return (function(x as integer) x+n)
end function
```

The long and abbreviated forms are compatible with QBasic and some other PC BASICs. FlexBasic also supports a much more convenient short form. This short form starts with `[`, followed by the function parameter list, followed by ':', the statements in the anonymous function, and finally `=>` and a result expression. This sounds more complicated than it is. The above `plusn` function in short notation is:

```
function plusn(n as integer) as intfunc
  return [x:=>x+n]
end function
```

This short form is much easier to write for many inline uses, and is very flexible, but is not compatible with other BASICs.

Closures

You'll note in the examples of anonymous functions that the anonymous function inside `plusn` is accessing the parameter `n` of its parent. This is allowed, and the value of `n` is in fact saved in a special object called a "closure". This closure is persistent, and functions are allowed to modify the variables in a closure. For example, we can implement a simple counter object as follows:

```
type intfunc as function() as integer

' makecounter returns a counter with a given initial value and step
function makecounter(value = 1, stepval = 1) as intfunc
  return (function () as integer
            var r = value
        value = value + stepval
        return r
      end function)
end function

var c = makecounter(7, 3)

' prints 7, 10, 13, 16
```

```
for i = 1 to 4
  print c()
next
```

Using the more compact notation for functions this may be written as:

```
type intfunc as function() as integer

function makecounter(value = 1, stepval = 1) as intfunc
  return [:var r = value : value = value + stepval : => r]
end function

var c = makecounter(7, 3)
for i = 1 to 4
  print c()
next
```

GETCNT

Propeller specific builtin function.

```
  function getcnt() as uinteger
  x = getcnt()
```

Returns the current cycle counter. This is an unsigned 32 bit value that counts the number of system clocks elapsed since the device was turned on. It wraps after approximately 54 seconds on propeller 1 and 27 seconds on propeller 2.

GOSUB

`gosub x` pushes a return value on the stack and jumps to the label `x` (which may be a numeric label). A `return` statement will pop the return value off the stack and resume execution after the original `gosub`.

`gosub` is supported for compatibility with old BASIC code, but should not be used in new code. In new code you should create a subroutine or function instead. See `sub`.

GOTO

`goto x` jumps to a label `x`, which must be defined in the same function. Labels are defined by giving an identifier followed by a `:`. For example:

```
  if x=y goto xyequal
  print "x differs from y"
  goto done
```

```
xyequal:
  print "x and y are equal"
done:
```

Note that in most cases code written with a `goto` could better be written with `if` or `do` (for instance the example above would be easier to read if written with `if` ... `then` ... `else`). `goto` should be used sparingly.

Also note that a label must be the only thing on the line; that is:

```
foo: bar
```

is interpreted as two statements

```
foo
bar
```

whereas

```
foo:
bar
```

is a label `foo` followed by a statement `bar`.

HEAPSIZE

```
  const HEAPSIZE = 256
```

Declares the amount of space to be used for internal memory allocation by things like string functions. The default is 256 bytes for P1 and 4096 bytes for P2. If your program does a lot of string manipulation and/or needs to hold on to the allocations for a long time, you may need to increase this by explicitly declaring `const HEAPSIZE` with a larger value.

IF

An IF statement introduces some code that should be executed only if a condition is true:

```
if x = y then
  print "x and y are the same"
else
  print "x and y are different"
end if
```

There are several forms of `if`.

A “simple if” executes just one statement if the condition is true, and has no `else` clause. Simple ifs do not have a `then`:

```
' simple if example
if x = y print "they are equal"
```

A one line if executes the rest of the statements on the current line if the condition is true. This form of `if` has a `then` that is followed by one or more statements, seperated by `:`. For example:

```
if x = y then print "they are equal" : print "they are still equal"
```

which will print “they are equal” followed by “they are still equal” if `x` equals `y`, but which will print nothing if they are not equal. This form of `if` is provided for compatibility with old code, but is not recommended for use in new code.

Compound if statements have a `then` which ends the line. These statements continue on until the next matching `else` or `end if`. If you want to have an `else` condition then you will have to use this form of if:

```
if x = y then
  print "they are equal"
else
  print "they differ"
end if
```

You may also put an if statement after an else:

```
if x = y then
  print "x and y are the same"
  print "I don't know about z"
else if x = z then
  print "x and z are the same, and different from y"
else
  print "x does not equal either of the others"
end if
```

INPUT

Used for reading data

The `input` keyword when used as a command acts to read data from a handle. It is followed by a list of variables. The data are separated by commas.

```
  print "enter a string and a number: ";
  input s$, n
  print "you entered: ", s, "and", n
```

The input may optionally be preceded by a prompt string, so the above could be re-written as:

```
   input "enter a string and a number: ", s$, n
   print "you entered: ", s, "and", n
```

If the prompt string is separated from the variables by a semicolon `;` rather than a comma, then `"? "` is automatically appended to the prompt.

Used for accessing pins

`input` may also be used to refer to a pseudo-array of bits representing the state of input pins. On the Propeller 1 this is the 32 bit INA register, but on Propeller 2 it is 64 bits.

Bits in the `input` array may be read with an array-like syntax:

```
x = input(0)    ' read pin 0
y = input(4,2)  ' read pins 4,3,2
```

Note that usually you will want to read the pins with the larger pin number first, as the bits are labelled with bit 31 at the high bit and bit 0 as the low bit.

Also note that before using a pin as input its direction should be set as input somewhere in the program:

```
direction(4,0) = input  ' set pins 4-0 as inputs
```

INPUT$

A predefined string function. There are two ways to use this.

The first, and simpler way, is just as `input$(n)`, which reads `n` characters from the default serial port and returns a string made of those characters. `input$(1)` is thus a kind of `getchar` to read a single character.

The second form, `input$(n, h)` reads up to `n` characters from handle `h`, as created by an `open device as #h` statement. If there are not enough characters to fulfil the request then a shorter string is returned; for example, at end of file an empty string “” will be returned.

Example:

```
file$ = ""           ' initialize read data
do
  s$ = input$(80, h) ' read up to 80 characters at a time
  file$ = file$ + s$ ' append to the data
until s$ = ""        ' stop at end of file
' now the whole file is in file$
```

INT

Convert float to int. Any fractional parts are truncated.

```
i = int(3.1415) ' now i will be set to 3
```

INTEGER

A 32 bit signed integer type. The unsigned 32 bit integer type is `uinteger`.

LEFT$

A predefined string function. `left$(s, n)` returns the left-most `n` characters of `s`. If `n` is longer than the length of `s`, returns `s`. If `n` $=<0$, returns an empty string. If a memory allocation error occurs, returns `nil`.

LEN

A predefined function which returns the length of a string.

```
var s$ = "hello"
var n = len(s$) ' now n = 5
```

LET

Variable assignment:

```
let a = b
```

sets `a` to be equal to `b`. This can usually be written as:

```
a = b
```

the only difference is that in the `let` form if `a` does not already exist it is created as a member variable (one accessible in all functions of this file). The `let` keyword is deprecated in some versions of BASIC (such as FreeBASIC) so it's probably better to use `var` or `dim` to explicitly declare your variables.

LONG

A signed 32 bit integer. An alias for `integer`. The unsigned version of this is `ulong`.

LOOP

Marks the end of a loop introduced by `do`. See DO for details.

MOD

`x mod y` finds the integer remainder when `x` is divided by `y`.

Note that if both the quotient and remainder are desired, it is best to put the calculations close together; that way the compiler may be able to combine the two operations into one (since the software division code produces both quotient and remainder). For example:

```
q = x / y
r = x mod y
```

NEW

Allocates memory from the heap for a new object, and returns a pointer to it. May also be used to allocate arrays of objects. The name of the type of the new object appears after the `new`, optionally followed by an array limit. Note that as in `dim` statements, the value given is the last valid index, so for arrays starting at 0 (the default) it is one greater than the number of elements.

```
var x = new ubyte(10)   ' allocate 11 (not 10) bytes and return a pointer to it
x(1) = 1                ' set a variable in it

class FDS using "FullDuplexSerial.spin" ' Use "SmartSerial.spin" on P2
var ser = new FDS       ' allocate space for a new full duplex serial object
ser.start(31, 30, 0, 115_200) ' start up the new object
```

See the discussion of memory allocation for tips on using `new`. Note that the default heap is rather small, so you will probably need to declare a larger `HEAPSIZE` if you use `new` a lot.

Memory allocated by `new` may be explicitly freed with `delete`; or, it may left to be garbage collected automatically.

NEXT

Indicates the end of a `for` loop. The variable used in the loop may be placed after the `next` keyword, but this is not mandatory. If a variable is present though then it must match the loop.

See FOR.

NIL

A special pointer value that indicates an invalid pointer. `nil` may be returned from any string function or other function that allocates memory if there is not enough space to fulfil the request. `nil` is of type `any` and may be assigned to any variable. When assigned to a numeric variable it will cause the variable to become 0.

NOT

```
  a = NOT b
```

Inverts all bits in the destination. This is basically the same as `b xor -1`.

In logical (boolean) conditions, since the TRUE condition is all 1 bits set, this operation has its usual effect of reversing TRUE and FALSE.

ON X GOTO

For compatibility only, FlexBASIC accepts statements like:

```
on x goto 100, 110, 120
```

This is equivalent to

```
select case x
case 1
  goto 100
case 2
  goto 110
case 3
  goto 120
end select
```

OPEN

Open a handle for input and/or output. The general form is:

```
  open device as #n
```

where `device` is a device driver structure returned by a system function such as `SendRecvDevice`, and `n` evaluates to an integer between 2 and 7. (Handles 0 and 1 also exist, but are reserved for system use.)

Example:

```
  open SendRecvDevice(@ser.tx, @ser.rx, @ser.stop) as #2
```

Here the `SendRecvDevice` is given pointers to functions to call to send a single character, to receive a single character, and to be called when the handle is closed. Any of these may be `nil`, in which case the corresponding function (output, input, or close) does nothing.

OPTION

Gives a compiler option. The following options are supported:

OPTION BASE

`option base N`, where `N` is an integer constant, causes the default base of arrays to be set to `N`. After this directive, arrays declared without an explicit base will start at `N`. Typically `N` is either `0` or `1`. The default is `0`.

```
dim a(9) as integer  ' declares an array with indices 0-9
option base 0        ' note: changing option base after declarations is not recommended, but
dim b(5) as integer  ' declares an array with indices 1-5 (5 elements)
```

It is possible to use `option base` more than once in a file, but we do not recommend it. Indeed if you do use `option base` it is probably best to use it at the very beginning of the file, before any array declarations

OPTION EXPLICIT

Requires that all variables be explicitly declared before use. The default is to allow variables in LET and FOR statements to be implicitly declared.

OPTION IMPLICIT

Allows variables to be automatically declared in any assignment statement, `read`, or `input`. The type of the variable will be inferred from its name if it has not already been declared.

OR

```
  a = x or y
```

Returns the bit-wise inclusive OR of x and y. If x or y is a floating point number then it will be converted to integer before the operation is performed.

Also useful in boolean operations. The comparison operators return 0 for false conditions and all bits set for true conditions, so you can do things like:

```
  if (x < y OR x = z) then
    ' code that runs if either condition is true
  end if
```

OUTPUT

A pseudo-array of bits representing the state of output bits. On the Propeller 1 this is the 32 bit OUTA register, but on Propeller 2 it is 64 bits.

Bits in `output` may be read and written an array-like syntax:

```
   output(0) = not output(0)   ' toggle pin 0
   output(4,2) = 1  ' set pins 4 and 3 to 0 and pin 2 to 1
```

Note that usually you will want to access the pins with the larger pin number first, as the bits are labelled with bit 31 at the high bit and bit 0 as the low bit.

Also note that before using a pin as output its direction should be set as output somewhere in the program:

```
   direction(4,0) = output  ' set pins 4-0 as outputs
```

PAUSEMS

A built-in subroutine to pause for a number of milliseconds. For example, to pause for 2 seconds, do

```
  pausems 2000
```

PINLO

Force a pin to be output as 0.

```
  pinlo(p)
```

PINHI

Force a pin to be output as 1.

```
  pinhi(p)
```

PINSET

Force a pin to be an output, and set its value (new value must be either 0 or 1).

```
  pinset(p, v)
```

PINTOGGLE

Force a pin to be an output, and invert its current value.

```
  pintoggle(p)
```

PRINT

`print` is a special subroutine that prints data to a serial port or other stream. The default destination for `print` is the pin 30 (pin 62 on P2) serial port, running at 115_200 baud (230_400 baud on P2).

More than one item may appear in a print statement. If items are separated by commas, a tab character is printed between them. If they are separated by semicolons, nothing is printed between them, not even a space; this differs from some other BASICs.

If the print statement ends in a comma, a tab is printed at the end. If it ends in a semicolon, nothing is printed at the end. Otherwise, a newline (carriage return plus line feed) is printed.

As a special case, if a backslash character \ appears in front of an expression, the value of that expression is printed as a single byte character.

Examples

```
   ' basic one item print
   print "hello, world!"
   ' two items separated by a tab
   print "hello", "world!"
   ' two items with no separator
   print "hello"; "world"
   ' an integer, with no newline
   print 1;
   ' a string and then an integer, nothing between them
   print "then "; 2
```

prints

```
hello, world!
hello  world
helloworld
1then 2
```

`print` may be redirected. For example,

```
print #2, "hello, world"
```

prints its message to the device previously `open`ed as device #2.

PRINT USING

Formats output using a string. The general form of this is:

```
  print using STRING; expr [,expr...] [;]
```

where `STRING` is a string literal and `expr` is one or more expressions.

Within the string literal output fields are specified by special forms, which are replaced by the various expressions.

`&` indicates a variable width field, within which the numbers or strings are printed with the minimum number of characters.

`#` starts a numeric field with space padding; the number of `#` characters indicates the width of the field. The numeric value is printed right-justified within the field. If it cannot fit, the first digit which will fit is replaced with '#' and the rest are printed normally. If the field is preceded by a `-` or `+` the sign is printed there; otherwise, if the value is negative then the `-` sign is included in the digits to print.

`%` starts a numeric field with 0 padding; the number of `%` characters indicates the width of the field. Leading zeros are explicitly printed. If the number cannot fit in the indicated number of digits, the first digit which will fit is replaced with '#' and the rest are printed normally.

`+` indicates that a place should be reserved for a sign character (`+` for non-negative, `-` for negative). `+` must immediately be followed by a numeric field. If the argument is an unsigned integer, instead of `+` a space is always printed.

`-` indicates that a place should be reserved for a sign character (space for non-negative, `-` for negative). `-` must immediately be followed by a numeric field. If the argument is an unsigned integer, a space is always printed.

`!` indicates to print a single character (the first character of the string argument).

`\` indicates a string field, which continues until the next `\`. The width of the field is the total number of characters, including the beginning and ending `\`. The string will be printed left justified within the field. Centering or right justification may be achieved for fields of length 3 or more by using `=` or '>' characters, respectively, as fillers between `\`. If the string is too long to fit within the field, only the first `N` characters of the string are printed.

```
print using "%%%%"; x
```

PROGRAM

This keyword is reserved for future use.

The statements in the top level of the file (not inside any subroutine or function) are placed in a method called `program`. This is only really useful for calling them from another language (for example a Spin program using a BASIC program as an object).

READ

`read` reads data items declared by `data`. All of the strings following `data` keywords are lumped together, and then parsed by `read` in the same way as `input` parses data typed by the user.

REM

Introduces a comment, which continues until the end of the line. A single quote character `'` may also be used for this.

RESTORE

Resets the internal pointer for `read` so that it starts again at the first `data` statement.

RETURN

Return from a subroutine or function. If this statement occurs inside a function, then the `return` keyword may be followed by an expression giving the value to return; this expression should have a type compatible with the function's return value.

A `return` with a value sets the function's result value and exits. If the `return` does not have a value (or indeed if there is no `return`), then the function's result value is the last value assigned to the pseudo-variable that has the same name as the function. That is, two equivalent ways of writing a sum function are:

```
function sum(x, y)
  sum = x+y
end function
```

or

```
function sum(x, y)
  return x+y
end function
```

RIGHT$

A predefined string function. `right$(s, n)` returns the right-most `n` characters of `s`. If `n` is longer than the length of `s`, returns `s`. If `n` =< 0, returns an empty string. If a memory allocation error occurs, returns `nil`.

RND

A predefined function which returns a random floating point number `x` such that `0.0 <= x and x < 1.0`. A single argument `n` is given. If `n` is negative, then it is used as the seed for the random number sequence. If `n` is 0, a new sequence is started with a random seed. If `n` is positive, the next value in the sequence is returned.

```
  f = rnd(0) ' start a new sequence
  i = int(rnd(1)*6) + 1 ' generate random between 1 and 6
```

SELECT CASE

Selects between alternatives. The expression after the initial `select case` is evaluated once, then matched against each of the `case` statements (in order) until one matches or `end select` is reached. `case else` will match anything (and hence should be placed last, since no `case` after it can ever match).

In case of a match, all of the statements between the matching `case` and the next `case` (or `end select`) will be executed.

```
var keepgoing = -1
do
   print "continue? ";
   a$ = input$(1)
   print
   a$ = input$(1)
   select case a$
   case "y"
     keepgoing = 1
     print "great!"
   case "n"
     keepgoing = 0
     print "ok, not continuing "
   case else
     print "I did not understand your answer of "; a$
   end select
loop while keepgoing = -1
```

SELF

Indicates the current object. Not implemented yet.

SENDRECVDEVICE

A built-in function rather than a keyword. `SendRecvDevice(sendf, recvf, closef)` constructs a simple device driver based on three functions: `sendf` to send a single byte, `recvf` to receive a byte (or return -1 if no byte is available), and `closef` to be called when the device is closed. The value(s) returned by `SendRecvDevice` is only useful for passing directly to the `open` statement, and should not be used in any other context (at least not at this time).

_SETBAUD

Set up the serial port baud rate, based on the current clock frequency.

```
  _setbaud(115_200) ' set baud rate to 115_200
```

The default serial rate on P1 is 115_200 baud, and assuming a clock frequency of 80_000_000 (on P2 both defaults are doubled). If these are changed, it is necessary to call `_setbaud` again in order for serial I/O to work.

SHORT

A signed 16 bit integer, occupying two bytes of computer memory. The unsigned version of this is `ushort`. The difference arises with the treatment of the upper bit. Both `short` and `ushort` treat 0-32767 the same, but for `short` 32768 to 65535 are considered equivalent to -32768 to -1 respectively (that is, when a `short` is copied to a larger sized integer the upper bit is repeated into all the other bits; for `ushort` the new bits are filled with 0 instead).

SINGLE

Single precision floating point data type. By default this is an IEEE 32 bit single precision float, but compiler options may change this (for example to a 16.16 fixed point number).

SQR

An alias for `sqrt`, for compatibility with older BASICs.

SQRT

Calculate the square root of a number.

```
x = sqrt(y)
```

This is not a true function, but a pseudo-function whose result type depends on the input type. If the parameter to `sqrt` is an integer then the result will be an integer as well. If the parameter is a single then the result is a single.

STEP

Gives the increment to apply in a FOR loop.

```
for i = 2 to 8 step 2
  print i
next
```

will print 2, 4, 6, and 8 on separate lines.

STR$

Convert a number to a string. The input is a floating point number (integers will automatically be converted to `single`) and the output is a string representing the number. Unlike the format used for regular `print`, `str$` tries to avoid trailing zeros, so the output is somewhat more compact than `print`.

SUB

Defines a new subroutine. This is like a `function` but with no return value. Subroutines have a fixed number and type of arguments, but the last arguments may be given default values with an initializer. For example:

```
  sub say(msg$="hello")
    print msg$
  end sub
```

If you call `say` with an argument, it will print that argument. If you call `say` with no argument it will print the default of `hello`:

```
   say("hi!")   ' prints "hi!"
   say "hi!"    ' the same
   say          ' prints "hello"
```

Subroutines may be invoked with function notation (arguments enclosed in parentheses) or with the arguments separated from the subroutine name by white space, as in the example above.

THEN

Introduces a multi-line series of statements for an `if` statement. See IF for details.

THROW

Throws an error which may be caught by a caller's `try/catch` block. If none of our callers has established a `try / catch` block, the program is ended.

The argument to `throw` may be of any type. Programmers should beware of mixing different types, as the `try / catch` block may need to know what type of value it should expect to receive.

Example:

```
if n < 0 then
  throw "illegal negative value"
endif
```

TO

A syntactical element typically used for giving ranges of items.

TRY

Example:

```
dim errmsg as string
try
  ' run sub1, sub2, then sub3. If any one of them
  ' throws an error, we will immediately stop execution
  ' and jump to the catch block
  sub1
  sub2
  sub3
catch errmsg
  print "a subroutine reports error: " errmsg
end try
```

TYPE

Creates an alias for a type. For example,

```
type uptr as ubyte ptr
```

creates a new type name `uptr` which is a pointer to a `ubyte`. You may use the new type name anywhere a type is required.

UBYTE

An unsigned 8 bit integer, occupying one byte of computer memory. The signed version of this is `byte`. The difference arises with the treatment of the upper bit. Both `byte` and `ubyte` treat 0-127 the same, but for `byte` 128 to 255 are considered equivalent to -128 to -1 respectively (that is, when a `byte` is copied to a larger sized integer the upper bit is repeated into all the other bits; for `ubyte` the new bytes are filled with 0 instead).

UINTEGER

An unsigned 32 bit integer.

ULONG

An unsigned 32 bit integer, occupying four bytes of computer memory. The signed version of this is `long`.

USHORT

An unsigned 16 bit integer, occupying two bytes of computer memory. The signed version of this is `short`. The difference arises with the treatment of the upper bit. Both `short` and `ushort` treat 0-32767 the same, but for `short` 32768 to 65535 are considered equivalent to -32768 to -1 respectively (that is, when a `short` is copied to a larger sized integer the upper bit is repeated into all the other bits; for `ushort` the new bits are filled with 0 instead).

USING

Keyword intended for use in PRINT statements, and also to indicate the file to be used for a CLASS.

VAR

Declare a local variable:

```
VAR i = 2
VAR msg$ = "hello"
```

`var` creates and initializes a new local variable (only available inside the function in which it is declared). The type of the new variable is inferred from the type of the expression used to initialize it; if for some reason that cannot be determined, the type is set according to the variable suffix (if any is present).

`var` is somewhat similar to `dim`, except that the type isn't given explicitly (it is determined by the initializer expression) and the variables created are always local, even if the `var` is in the main program (in the main program `dim` creates member variables that may be used by functions or subroutines in this file).

WAITCNT

Propeller specific builtin function. Waits until the cycle counter is a specific value

```
waitcnt(getcnt() + clkfreq) ' wait one second
```

WAITPEQ

Propeller specific builtin function. Waits for pins to have a specific value (given by a bit mask). Same as the Spin `waitpeq` routine. Note that the arguments are bit masks, not pin numbers, so take care when porting code from PropBasic.

WAITPNE

Propeller specific builtin function. Waits for pins to not have a specific value (given by a bit mask). Same as the Spin `waitpne` routine. Note that the arguments are bit masks, not pin numbers, so take care when porting code from PropBasic.

WHILE

Begins a loop which continues as long as a specified condition is true.

```
' wait for pin to go low
loopcount = 0
while input(1) <> 0
  loopcount = loopcount + 1
wend
print "waited "; loopcount; " times until pin went high"
```

The end of the repeated code may be terminated either with `wend` or with `end while`.

The `while` loop may also be written as `do while`:

```
  do while input(1) <> 0
    loopcount = loopcount + 1
  loop
```

or

```
  do until input(1) = 0
    loopcount = loopcount + 1
  loop
```

WORD

Reserved for use in inline assembler.

XOR

```
  a = x xor y
```

Returns the bit-wise exclusive or of x and y. If x or y is a floating point number then it will be converted to integer before the operation is performed. `xor` is often used for flipping bits.

Propeller Specific Variables

clkfreq

```
  dim clkfreq as uinteger
```

Clkfreq gives the frequency of the system clock in cycles per second.

Sample Programs

Toggle a pin

This program toggles a pin once per second.

```
rem simple program to toggle a pin

const pin = 16

direction(pin) = output

do
  output(pin) = not output(pin)
  pausems 1000
loop
```