Using Parallax Propeller™ Floating Point Routines

The Propeller chip can be programmed using either the Spin Language or assembly code. The design objective for the Propeller floating point package was to develop support for floating point functions callable by both Spin and assembler code, and to optimize for speed while using a minimum of memory. Each of the components of this objective plays off against the others. For example, the minimum amount of memory would be used by coding entirely in Spin (e.g. FloatMath), but this comes at the cost of execution speed (371 usec for FAdd in FloatMath vs 4.7 usec in assembler). Alternatively, the fastest execution would be obtained by coding entirely in assembler, but since there are only 496 longs in a cog, you can quickly run out of memory. The most practical solution is a balance between the various design constraints.

The following sections describe how to implement floating point operations using either the Spin Language or assembler code.

Using Floating Point with Spin™

The three options for doing floating point operations in Spin are to use the FloatMath, Float32 or Float32Full objects. The FloatMath object implements all functions in Spin code so no additional cogs are required, but execution speed is much slower than the speed of the Float32 and Float32Full objects. The Float32 and Float32Full objects implement functions in assembler code and provide a Spin interface. Float32 uses one additional cog, while Float32Full uses two additional cogs. Each of the options is summarized below.

FloatMath

The FloatMath object can be used when only basic floating point functions are required and execution speed is not a significant factor. It using the least amount of memory and requires no additional cog.

The FloatMath object contains the following functions:

```
FAdd, FSub, FMul, FDiv
FFloat, FTrunc, FRound
FSqr
FNeg, FAbs
```

Pros:

- no cogs used
- smallest use of memory

Cons:

- slower execution speed
- limited number of functions

FloatMath Execution Speeds (Spin)

FloatMath Functions (Spin)	**Execution Speed (usec)**
`FAdd`	371
`FSub`	404
`FMul`	341
`FDiv`	1413
`FFloat`	192
`FTrunc`	163
`FRound`	163
`FSqr`	1522
`FNeg`	21
`FAbs`	21

Float32

The Float32 object should be used when additional floating point functions are required or when execution speed is a factor. It uses one cog and provides much faster execution speed. There is an easy migration path from FloatMath to Float32 since all common functions have the same name. The only code change required is to add a start call at the beginning of the program to start the Float32 cog.

The Float32 object contains the following functions:

```
FAdd, FSub, FMul, FDiv
FFloat, FTrunc, FRound, Frac
FSqr, Pow
FNeg, FAbs
FCmp, FMin, FMax
Sin, Cos, Tan, Radians, Degrees
Log, Log10, Exp, Exp10
FMod
```

Pros:

- faster execution speed
- uses one cog

Cons:

- doesn't support all functions

Float32 Execution Speeds (Spin)

Float32 Functions (Spin)	**Execution Speed (usec)**
`FAdd`	39
`FSub`	39
`FMul`	46

FDiv	45
FFloat	35
FTrunc	36
FRound	36
FSqr	247
FNeg	21
FAbs	21
FCmp	39
Sin	128
Cos	128
Tan	234
Log	76
Log10	76
Exp	82
Exp10	82
Pow	112
Frac	36
FMod	72
Radians	62
Degrees	62
FMin	47
FMax	47

Float32Full

The Float32Full object should be used when additional floating point functions not supported by Float32 are required (e.g. ASin, ACos, ATan, ATan2, Floor, Ceil), or when the user-defined function processor is used. It uses two cogs. All common functions have the same name, and the extra cog is handled internally, so no code changes are required to switch between Float32 and Float32Full (except the name of the object in the object definition).

The Float32Full object contains the following functions:

```
    FAdd, FSub, FMul, FDiv
    FFloat, FTrunc, FRound, Frac
    FSqr, Pow
    FNeg, FAbs
    FCmp, FMin, FMax
    Sin, Cos, Tan, Radians, Degrees
    ASin, ACos, ATan, ATan2
    Log, Log10, Exp, Exp10
    FMod
    Floor, Ceil
    FFunc  (user-defined functions)
```

Pros:

- faster execution speed
- supports all functions
- user-defined function processor

Cons:

- uses two cogs

Float32Full Execution Speeds (Spin)

Float32Full Functions (Spin)	**Execution Speed (usec)**
`FAdd`	39
`FSub`	39
`FMul`	46
`FDiv`	45
`FFloat`	35
`FTrunc`	36
`FRound`	36
`FSqr`	247
`FNeg`	21
`FAbs`	21
`FCmp`	39
`Sin`	128
`Cos`	128
`Tan`	234
`Log`	76
`Log10`	76
`Exp`	82
`Exp10`	82
`Pow`	112
`Frac`	36
`FMod`	72
`Radians`	62
`Degrees`	62
`FMin`	47
`FMax`	47
`ASin`	359
`ACos`	366
`ATan`	168
`ATan2`	183
`Floor`	49
`Ceil`	49

Using Floating Point in Assembler Code

Assembler code provides significantly faster execution speed. The following shows a quick comparison for a floating point add:

FloatMath FAdd (Spin)	371.0 usec
Float32 FAdd (Spin)	39.0 usec
_FAdd (Assembler)	4.7 usec

The following table shows the execution times for all Float32 and Float32Full function calls in assembler code. In some cases in-line code is used rather than a subroutine call. The in-line code is shown in the table.

Float32 Execution Speeds (Assembler)

Float32 (Assembler)	**Execution Speed (usec)**
`_FAdd`	4.7
`_FSub`	4.8
`_FMul`	10.5
`_FDiv`	13.2
`_FFloat`	5.4
`_FTrunc`	1.7
`_FRound`	1.7
`_FSqr`	217
(FNeg) `xor fnumA, Bit31`	0.05
(FAbs) `andn fnumA, Bit31`	0.05
`_FCmp`	0.8
`_Sin`	93.2
`_Cos`	97.7
`_Tan`	204
`_Log`	44.4
`_Log10`	44.4
`_Exp`	48.5
`_Exp10`	48.8
`_Pow`	79.2
`_Frac`	4.3
`_FMod`	37.9

(Radians)
`call _FMulI`
`long pi / 180.0`
	10.6

(Degrees)
`call _FMulI`
`long 180.0 / pi`
	10.6

(FMin) `call _FCmp`	0.85

`if_nc_and_nz mov fnumA, fnumB`	

(FMax)
`call _FCmp`
`if_c mov fnumA, fnumB`	

_ASin	326.6
_ACos	331.7
_ATan	132.4
_ATan2	146.3
_Floor	15.9
_Ceil	15.9

All floating point operations supported by Float32 and Float32Full are coded as stand-alone subroutines that can be called from assembler code. A command dispatch routine handles command requests from the Spin code, and a Spin function is defined to call each of the assembler routines. The name of the assembler function is the same name as the Spin function, but with an underscore prefix.

To use floating point routines in your own assembler code you can copy and paste the assembler code from the Float32 or Float32A objects. There are various constants, variables and support routines that are required in addition to the individual functions. The following table shows the amount of memory required to support the four basic floating point operations (FAdd, FSub, FMul, FDiv).

Memory usage for basic functions (cog)

	longs
_FAddI/_FAddI	24
_FSub/_FSubI	5
_FMul/_FMulI	18
_FDiv/_FDivI	19
Unpack2	12
Unpack	30
Pack	25
Constants	6
Variables	10
TOTAL	**174**

The assembler functions use either one or two arguments, defined as fnumA and fnumB, and return the result in fnumA. For example:

```
mov  fnumA, val1
mov  fnumB, val2
call _FAdd
```

To facilitate assembler coding, an immediate value call has been added for each of the basic operators (_FAddI, _FSubI, _FMulI, _FDivI, _FCmpI). Using the immediate call, the fnumB value is defined by the long value immediately following the call. This example adds the constant 1.25 to fnumA:

```
call _FAddI
long 1.25
```

The following tables show the cog memory usage for the Float32 and Float32A objects.

Float32 Memory Space (cog)

	longs
command dispatch	56
_FAddI/_FAddI	24
_FSub/_FSubI	5
_FMul/_FMulI	18
_FDiv/_FDivI	19
_FFloat	13
_FTrunc/Fround	17
_FSqr	28
cmdFCmp/_FCmp/_FCmpI	25
_Sin/_Cos	45
_Tan	8
_Log2/_Log/_Log10	27
_Exp2/_Exp/_Exp10	35
_Pow	6
_Frac	10
_Fmod	14
loadTable	19
floatBits	8
_Unpack2	12
_Unpack	30
_Pack	25
Constants	16
Variables	17
TOTAL	**477**

Float32A Memory Space (cog)

	longs
Command dispatch and function processor	94
_Asin/_Acos	36
_Atan	29
_Atan2	25

_Ceil/_Floor	20
linkage	5
sendCmd	15
_FAddI/_FAddI	24
_FSub/_FSubI	5
_FMul/_FMulI	18
_FDiv/_FDivI	19
_FCmp/_FCmpI	22
_FFloat	13
_FTrunc/Fround	17
poly	15
_Unpack2	12
_Unpack	30
_Pack	25
Constants	11
Variables	19
TOTAL	**454**

User-defined Functions

The preferred method of programming for many applications will be Spin code. In floating point intensive applications, the Spin overhead can be a major issue, and the alternative of coding in assembler may not be desirable. Implementing in assembler has limitations due to the space limit of 496 longs for each cog. On the other hand, there may be lots of main memory available. The user-defined function processor capitalizes on this by allowing functions to be defined in main memory, but processed in assembler by Float32A. The built-in function processor makes effective use of the command dispatch routines to accomplish this task. The same command codes are used for both the call linkage and user-defined function operations. A quick example is as follows:

```
DAT
x      long     0
y      long     0
z      long     0
func1  long     JmpCmd+@func1   ' defines runtime offset
       long     LoadCmd+@x      ' fnumA = x
       long     FMulCmd+@y      ' fnumA = fnumA + y
       long     SaveCmd+@z      ' z = fnumA
       long     0               ' end of function
```

This has the advantage of much faster execution, and the variables are all directly accessible by the Spin code.

Floating Point Tips and Traps

Starting Float32 and Float32Full

Tip: Float32 and Float32Full both use additional cogs for their assembler routines. Make sure you add a start call to the beginning of your program.
e.g.

```
OBJ
  f   : "Float32"
…
PUB
  f.start
```

Float32 vs Float32Full

Tip: If you don't need the extra functions provided by the Float32Full object, you should use the Float32 object. Float32 requires less memory and uses only one additional cog instead of two.

Floating Point Operations

Trap: Built-in operators are only used for integer operations. You must call floating point routines to perform floating point operations.
e.g.

Suppose you want to add the two floating point variables x and y.
`x := f.FAdd(x, y)` is correct
`x := x + y` is incorrect, it specifies an integer addition

NaN

Tip: When using Float32 and Float32Full any error that occurs during floating point calculations will result in NaN (Not-a-Number). NaN is equal to the value $7FFF_FFFF, and will propagate through all subsequent floating point functions. If you have a NaN result, you can trace it back to find the error in your calculation. Typical errors are divide by zero or an argument out of range.

Floating Point Constants

Trap: Make sure that you don't forget the decimal point when you specify a floating point constant. If no decimal point is used, Spin will assume it's an integer constant. No error will be displayed at compile time, but the run-time calculation will be incorrect.
e.g.

Suppose you want to add 2.0 to the floating point variable x
`x := f.FAdd(x, 2.0)` is correct
`x := f.FAdd(x, 2)` is incorrect, because 2 is an integer value

Pi

Tip: Spin has a built-in definition for the floating point constant pi (~3.141593).
e.g.

```
angle := f.FDiv(pi, 2.0)
```

Floating Point Expressions

Trap: Although Spin recognizes individual floating point constants, if you try to use floating point constants in an expression, you will not get the desired result because expressions are evaluated as integer expressions.
e.g.

`x := f.FAdd(x, pi)` is correct, because pi is a single floating point constant
`x := f.FAdd(x, pi / 2.0)` will yield an incorrect result

You need to use the constant directive to declare an in-line floating point expression.
e.g.

```
x := f.FAdd(x, constant(pi / 2.0))
```

Floating Point Expressions

Tip: Floating point expressions are valid in the CON section.
e.g.

```
CON
  piBy2 = pi / 2.0
…
PUB or PRI
  x := f.FAdd(x, piBy2)
```

Radians

Trap: Remember to use radians when calling the Sin, Cos, Tan functions, a common error is to use degrees. Two function calls, Degrees and Radians, are provided make it easy to convert.
e.g.

To find the Sin of 60 degrees, first convert to radians, then call Sin.

```
x := f.Sin(Radians(60.0))
```

Square of a number

Tip: To calculate the square of a number, it is faster to multiply the number by itself, then to use the Pow function.
e.g.

`x := f.FMul(x, x)` returns the value x^2