FlexSpin Language Reference

5.9.14

Total Spectrum Software

07/22/2022

Contents

Flexspin Spin

Introduction

Flexspin was designed to accept the language Spin as documented in the Parallax Propeller Manual. It should be able to compile most Spin programs, as long as there is space. The restriction is an important one; other Spin compilers produce Spin bytecode, a compact form that is interpreted by a program in the Propeller's ROM. Flexspin produces LMM code, which is basically a slightly modified Propeller machine code (slightly modified to run in HUB memory instead of COG). This is much larger than Spin bytecode, but also much, much faster.

Flexspin is able to produce binaries for both the P1 and P2 chips. Any assembly language written in DAT sections (or inside inline ASM blocks) must be for the appropriate chip; it will not be translated.

Flexspin also supports many of the features of the Spin2 language as extensions to Spin 1. It can also accept Spin2 programs as input. Spin1 and Spin2 are not completely compatible. Spin2 features which are not compatible with Spin1 are enabled if the file extension is `.spin2`.

Preprocessor

flexspin 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 in macros. 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. For example:

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

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.

INCLUDE

Includes a file.

WARN

Prints a warning message.

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__`	if compiling for Propeller 2
`__P2__`	if compiling for Propeller 2
`__FLEXSPIN__`	if the `flexspin` front end is used
`__SPINCVT__`	always defined
`__SPIN2PASM__`	if --asm is given (PASM output) (always defined by flexspin)
`__SPIN2CPP__`	if C++ or C is being output (never in flexspin)
`__HAVE_FCACHE__`	if the FCACHE optimization is enabled
`__cplusplus`	if C++ is being output (never in flexspin)
`__DATE__`	a string containing the date when compilation was begun
`__FILE__`	a string giving the current file being compiled
`__LINE__`	the current source line number
`__TIME__`	a string containing the time when compilation was begun
`__VERSION__`	a string containing the full version of flexspin in use

A predefined symbol is also generated for type of output being created:

Symbol	When Defined
`__OUTPUT_ASM__`	if PASM code is being generated
`__OUTPUT_BYTECODE__`	if bytecode is being generated
`__OUTPUT_C__`	if C code is being generated
`__OUTPUT_CPP__`	if C++ code is being generated

Special Preprocessor Comments

If the preprocessor sees the special comment `{$flexspin` at the beginning of a line, it will re-process the rest of the comment as input. This allows flexspin specific code to be placed in such comments. For example, you could do:

```
CON
```

```
  _clkfreq = 160_000_000

PUB main()
  repeat
{$flexspin
#ifdef __FLEXSPIN__
    DEBUG("hello from flexspin", 13, 10)
#else
}
    DEBUG("hello from PNut", 13, 10)
{$flexspin #endif}
    DEBUG("hello from both", 13, 10)
```

Memory Management

There are some built in functions for doing memory allocation. These are intended for C or BASIC, but may be used by Spin programs as well.

Heap allocation

The main function is `_gc_alloc_managed(siz)`, which allocates `siz` bytes of memory managed by the garbage collector. It returns 0 if not enough memory is avilable, otherwise returns a pointer to the start of the memory (like C's `malloc`). As long as there is some reference in COG or HUB memory to the pointer which got returned, the memory will be considered "in use". If there is no more such reference then the garbage collector will feel free to reclaim it. There's also `_gc_alloc(siz)` which is similar but marks the memory so it will never be reclaimed, and `_gc_free(ptr)` which explicitly frees a pointer previously allocated by `_gc_alloc` or `_gc_alloc_managed`.

The size of the heap is determined by a constant `HEAPSIZE` declared in the top level object. If none is given then a (small) default value is used.

Example:

```
' put this CON in the top level object to specify how much memory should be provided fo
' memory allocation (the "heap"). The default is 4K on P2, 256 bytes on P1
CON
   HEAPSIZE = 32768 ' or however much memory you want to provide for the allocator

' here's a function to allocate memory
' "siz" is the size in bytes
PUB allocmem(size) : ptr
  ptr := _gc_alloc_managed(size)
```

The garbage collection functions and heap are only included in programs which explicitly ask for them.

Stack allocation

Temporary memory may be allocated on the stack by means of the call `__builtin_alloca(siz)`, which allocates `siz` bytes of memory on the stack. This is like the C `alloca` function. Note that the pointer returned by `__builtin_alloca` will become invalid as soon as the current function returns, so it should not be placed in any global variable (and definitely should not be returned from the function!)

Interoperation with C and BASIC

C and BASIC files may be included as objects in Spin1 and Spin2 programs. To do this, be sure to include the entire file name (including any extension, like `.c` or `.bas`) in the `OBJ` line.

Note that C is a case sensitive language, so you must use the exact same name that C uses: if `VGA` is a constant defined in a C file `video.c` then it must be used always as all upper case, not as `Vga` or `vga`. For example:

```
OBJ
  screen: "video.c"
...
  x := screen.VGA  ' this is fine
  x := screen.Vga  ' this is an error, the case does not match
```

Calling C standard library functions

A simple way to include the C standard library is to declare an object using `libc.a`. C standard library functions may then be accessed as methods of that object. For example, to call `sprintf` you could do:

```
OBJ
  c: "libc.a"
...
  c.sprintf(@buf, string("the value is %x", 10), val)
```

Extensions to Spin

Flexspin has a number of extensions to the Spin languages (both Spin1 and Spin2). Many Spin2 features may be used in Spin1, and vice-versa. For example in Spin2 functions with no arguments must be called like `foo()`, whereas flexspin still accepts the Spin1 version `foo`.

Absolute address

The `@@@` operator returns the absolute hub address of a variable. This is the same as `@` in Spin code, but in PASM code `@` returns only the address relative to

the start of the DAT section.

Access to member variables

Flexspin allows (read only) access to member variables and constants using the . notation. That is, if S is an object like:

```
CON
  rows = 24
VAR
  long x, y
```

then one may write S.x to access member variable x of S, and S.rows to access the constant rows. The original Spin syntax S#rows is still accepted for accessing constants.

Modifying member variables of another object directly is not permitted, and will produce a syntax error.

Alternate string notation

Flexspin interprets the notation @"some chars" to mean the same thing as STRING("some chars"). This provides a shorter way to write some messages. Newer versions of PNut / PropTool allow this for Spin2.

Bitfield access

Bits m to n of a variable x may be accessed via the notation x.[m..n]. If m is the same as n this may be simplified to x.[n]. This notation may also be applied to hardware registers.

Boolean short-circuit operators

flexspin has operators x __andthen__ y and x __orelse__ y which only evaluate the right hand side (y) if they need to (like C's && and ||). Ordinary boolean AND and OR operators will be optimized to these in the cases where x and y do not have side effects.

CASE_FAST

CASE_FAST is just like CASE, except that each of the case items must be a constant expression. It is guaranteed to compile to a jump table (regular CASE may sometimes compile to a sequence of IF/ELSE IF).

coginit/cognew

The coginit (and cognew) functions in Flexspin can start functions from other objects than the current. (In "regular" Spin only functions from the same object

may be started this way.)

Conditional expressions

IF...THEN...ELSE expressions; you can use IF/THEN/ELSE in an expression, like:

```
r := if a then b else c
```

which is the same as

```
if a then
  r := b
else
  r := c
```

This may also be written in C / Verilog style as:

```
r := (a) ? b : c
```

In the latter form the parentheses around `a` are mandatory in Spin1 to avoid confusion with the random number operator `?`. In Spin2 the question mark is no longer used for random numbers, so the issue does not arise there.

Default function parameters

flexspin permits function parameters to be given default values by adding `= X` after the parameter declaration, where `X` is a constant expression. For instance:

```
VAR
  long a

PUB inc(n=1)
  a += n

PUB main
  inc(2) ' adds 2 to a
  inc(1) ' adds 1 to a
  inc    ' same as inc(1)
```

The default values must, for now, be constant. Perhaps in the future this restriction will be relaxed, but there are some slightly tricky issues involving variable scope that must be resolved first.

Default string parameters

If a default function parameter is declared as a string, and a string literal is passed to it, that string literal is transformed into a string constant. Normally Spin uses just the first character of a string literal when one is seen in an

expression (outside of STRING). Basically flexspin inserts a `string` operator around the literal in this case. So for example in:

```
PUB write(msg = string(""))
  '' do some stuff
...
  write(string("hello, world"))
  write("hello, world")
```

the two calls to `write` will do the same thing. In regular Spin, and in flexspin in the case where the default value is not present on a parameter, the second call will actually be interpreted as:

```
  write($68, $65, ..., 0)  ' $68 = ASCII value of "h"
```

which is probably not what was intended.

Function Aliases

A function may have an "alias" created for it. That is, if you want to be able to call the same function by two different names `add` and `_add`, you can do:

```
PUB _add(x, y)
  return x+y
PUB add = _add
```

The `PUB add = _add` line says that `add` is an alias for `_add`.

Aliases defined this way are "weak"; that is, they may be overridden by later definitions. They are mostly intended for use in libraries where for some reason (e.g. C standard compatibility) we wish to allow the program to use the same name as a library function without a conflict occuring.

Inline assembly with asm/end

flexspin accepts inline assembly in `PUB` and `PRI` sections. Inline assembly starts with `asm` and ends with `endasm`. The inline assembly is somewhat limited; the only operands permitted are immediate values, hardware registers like `OUTA`, local variables (including parameters and result values) of the containing function, or labels of that function. (Spin doesn't support goto and labels, but you can define labels in `asm` blocks and jump to them from other `asm` blocks that are in the same function. Some other languages supported by flexspin do have labels.) Member variables (declared in the `VAR` block) may not be used directly in inline assembly.

Branching inside the function should work, but trying to return from it or to otherwise jump outside the function will almost certainly cause you to come to grief, even if the compiler allows it. Calling subroutines is also not permitted.

All non-branch instructions should work properly in inline assembly, as long as the operands satisfy the constraints above. Conditional execution is allowed, and flag bits may be set and tested.

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

Example:

```
PUB waitcnt2(newcnt, incr)
  asm
    waitcnt newcnt, incr
  endasm
  return newcnt
```

waits until CNT reaches "newcnt", and returns "newcnt + incr".

Note that unlike most Spin blocks, the `asm` block has to end with `endasm`. This is because indentation is not significant inside the assembly code. For example, labels typically start at the leftmost margin.

Spin2 style inline assembly (org/end)

flexspin also accepts Spin2 style inline assembly, marked with `org` and `end` instead of `asm` and `endasm`. So the above example could be written as:

```
PUB waitcnt2(newcnt, incr)
  org
    waitcnt newcnt, incr
  end
  return newcnt
```

There are two important differences between `org`/`end` and `asm`/`endasm`:

(1) Normally inline assembly is treated the same as code generated by the compiler, and subject to optimization. However, code between `org` and `end` is not optimized. So if your inline assembly is timing sensitive, you should use `org` rather than `asm` to start it.

(2) Code between `org` and `end` is executed from the FCACHE area (in COG memory). Ordinary `asm` is hubexec.

Differences from PropTool/PNut

Unlike the "official" Spin2 compiler, flexspin does not accept an address for the inline assembly ("org $xxx"), and the assembly is not called as a subroutine (so no `ret` statement should be included). There are also some important restrictions in FlexSpin's implementation:

(1) `call` instructions may not be used in inline assembly

(2) The `ptra` register must not be modified by the inline assembly.

(3) The area set aside for inline assembly is smaller by default in FlexSpin than in Parallax's Spin2. This may be changed with the --fcache parameter to flexspin.

Method pointers

Pointers to methods may be created with `@` and called using the normal calling syntax. For example:

```
VAR
  LONG funcptr

PUB twice(x) : r
  r := x + x
...
  funcptr := @twice
  y := funcptr(a)
```

will set `y` to `a+a`.

If no parameters are to be passed to the called function, it is still necessary to write `()` after it in order to force it to be interpreted as a call. That is,

```
a = f()
```

causes an indirect call of the function pointed to by `f`, whereas

```
a = f
```

copies the pointer in `f` to `a`.

For functions returning multiple results, a `:N` notation is required after the function call, where `N` is an integer giving the expected number of results:

```
  x,y := fptr(a):2
```

The `:2` indicates that fptr is a pointer to a function which returns 2 results. Putting `:0` or `:1` for functions which return 0 or 1 results is optional in flexspin.

It is the programmer's responsibility to make sure that the number of results and arguments passed to a method called via a pointer are correct. No type checking is done.

Multiple return values and assignments

flexspin allows multiple return values and assignments. For example, to swap two variables `a` and `b` you can write:

```
  a,b := b,a
```

It's also acceptable (and perhaps easier to read) if there are parentheses around some or all of the multiple values:

```
  (a,b) := (b,a)
```

A function can also return multiple values. For instance, a function to calculate both a quotient and remainder could be written as:

```
PUB divrem(x,y)
  return x/y, x//y
```

or

```
PUB divrem(x,y) : q, r
  q := x/y
  r := x//y
```

This could later be used like:

```
  (a,digit) := divrem(a, 10)
```

It is also allowed to pass the multiple values returned from one function to another. So for example:

```
' function to double a 64 bit number
PUB dbl64(ahi, alo): bhi, blo
  bhi := ahi
  blo := alo
  asm
    add blo, blo wc
    addx bhi, bhi
  endasm

' function to quadruple a 64 bit number
PUB quad64(ahi, alo)
  return dbl64(dbl64(ahi, alo))
```

Object pointers

The proposed Spin2 syntax for abstract object definitions and object pointers is accepted. A declaration like:

```
OBJ
  fds = "FullDuplexSerial"
```

declares `fds` as having the methods of a FullDuplexSerial object, but without any actual variable or storage being instantiated. Symbols declared this way may be used to cast parameters to an object type, for example:

```
PUB print(f, c)
  fds[f].dec(c)

PUB doprint22
  print(@aFullDuplexSerialObj, 22)
```

PUB FILE and PRI FILE

A `pub` or `pri` function declaration may include a `file` directive which gives the file which contains the actual definition of the function. This looks like:

```
pub file "utils.spin" myfunc(x, y)
```

This declares a function `myfunc` with two parameters, which will be loaded from the file "utils.spin". The function will be a public function of the object. This provides an easy way to import the same function (e.g. a decimal conversion routine) into many different objects.

`pub file` and `pri file` differ from the `obj` directive in that they do not create a new object; the functions defined in the new file are part of the current object.

Note that there is no need for a body to the function (it is an error to give one). The number of parameters and return values, however, should be specified; they must match the number given in the final definition contained in the file.

The function body need not be in Spin. For example, to use the C `atoi` function in a Spin object on a Propeller1, you could do:

```
obj ser: "spin/FullDuplexSerial.spin"
pub file "libc/stdlib/atoi.c" atoi(str)

pub test
  ser.start(31, 30, 0, 115_200)
  x := string("1234")
  ser.dec(atoi(x))
```

(For Propeller2 you would have to modify this to use "spin/SmartSerial" and to change the output pins appropriately.)

Beware that functions declared with `file` are treated the same as other functions; in particular, note that the first function in the top level object will be used as the starting point for the program, even if that function was declared with `pub file` or `pri file`. So unlike in C, the declaration of external functions should be placed at the end of the file rather than the beginning (unless for some reason you want the main program to come from another file).

Typed parameters and return values

The "expression" in a default parameter may also be a type name, for example `long`, `float`, a flag indicating an unsigned type `+long`, (or one of the pointer types `@long` (pointer to long), `@word` (pointer to word), `@byte`, or `@float`. These do not provide a default value, but do provide a hint to the compiler about what type of value is expected. This isn't terribly useful for Spin, but does make it possible for the compiler to check types and/or convert them if necessary for Spin functions called from C or BASIC.

Variables declared as return values may also be given type hints, which will be used to determine the type of the function when it is accessed from another language.

Example:

```
' negate a floating point value
PUB negfloat(x = float) : r = float
  r := x ^ $80000000
```

Typed local variables

As in Spin2, local variables may be declared prefixed with `byte`, `word`, or `long` to give them a specific size.

Example:

```
PUB dostuff(x) : r | byte tempstring[10]
 ' may use tempstring here as an array
```

Unsigned operators

flexspin has some new operators for treating values as unsigned

```
  a +/ b   is the unsigned quotient of a and b (treating both as unsigned)
  a +// b  is the unsigned remainder of a and b
  a +** b  gives the upper 32 bits of unsigned multiplication
  a +< b   is an unsigned version of <
  a +> b   is an unsigned version of >
  a +=< b  is an unsigned version of =<
  a +=> b  is an unsigned version of =>
```

Most of these are also in Spin2.

Additional Spin2 operators

flexspin accepts some other Spin2 operators:

```
  a \ b   uses the value of a, but then sets a to b
  x <=> y returns -1, 0, or 1 if x < y, x == y, or x > y
```

New operators

flexspin implements some new operators:

```
  a +| b  is the same as a ZEROX b in Spin2
  a -| b  is the same as a SIGNX b in Spin2
```

Array parameters (deprecated)

flexspin allows method parameters to be small arrays; in this case, the caller must supply one argument for each element of the array. For example:

```
pub selector(n, a[4])
  return a[n]

pub tryit
  return selector(n, 1, 2, 3, 4)
```

This particular example could be achieved via `lookup`, but there are other cases where it might be convenient to bundle parameters together in an array.

This feature is still incomplete, and may not work properly for C/C++ output.

New PASM directives

flexspin accepts some Spin2 assembly directives, even in Spin1 mode.

ORGF

Forces the COG PC to reach a certain value by inserting 0 if necessary. For example, `orgf $100` will insert 0 bytes until the COG program counter reaches $100. `orgf X` is basically similar to:

```
long 0[X - $]
```

ORGF is valid only in COG space.

ORGH

Specifies that labels and code after this must be in HUB memory.

```
orgh   ' following code must be in HUB
orgh $400 ' following code must be in HUB at a specific address
```

If an address is given, then the code must not have exceeded that address yet, and 0 bytes will be inserted to force the HUB memory to get up to the given address.

Note that labels normally have two values, their COG memory address (specified by the last ORG) and their hub memory address (specified implicitly by how they are placed in RAM). After `orgh` the COG memory address is no longer valid, and the hub memory address may be explicitly given if the `orgh` had a value.

New intrinsics for both P1 and P2

Flexspin supports some new builtin functions. These typically start with an underscore to avoid confusion with existing variable names. Note that in Spin2 mode many of these are available without the leading underscore, and in fact it's better to use the non-underscore versions since those are also supported in the official Parallax compiler.

__PINW

`_pinw(p, c)` forces `p` to be an output and sets it to 0 if `c` is 0 or 1 if `c` is 1. If c is any other value the result is undefined. Supported for both P1 and P2.

__PINL

`_pinl(p)` drives pin `p` low, i.e. it forces `p` to be an output and sets it to 0. This is supported for both P1 and P2.

__PINH

`_pinh(p)` drives pin `p` high, i.e. it forces `p` to be an output and sets it to 1. This is supported for both P1 and P2.

__PINNOT

`_pinnot(p)` forces `p` to be an output and inverts it. This is supported for both P1 and P2.

__WAITX

`_waitx(n)` waits for `n` cycles, plus the cycle time required for the instruction. This is 2 cycles on P2, and 8 cycles on P1.

In Spin2 mode all of the above are available without the underscore.

Compatibility with other Spin 1 compilers

Limitations

Most Spin language features are supported. There may be some features that do not work; if you find any, please report them so they can be fixed.

The lexer and parser are different from the Parallax ones, so they may well report errors on code the Parallax compiler accepts.

Known Differences

Timing

Timing of produced code is different, of course (in general much faster than the native Spin interpreter). This may affect some objects; sometimes developers left out delay loops in time critical code because the Spin interpreter is so slow they weren't necessary. Watch out for this when porting I2C, SPI and similar functions.

@@ Operator

The Spin1 `@@` operator always truncates its result to 16 bits; flexspin does not do this. This won't matter in typical use (on the P1 addresses always fit in 16 bits anyway) but may be noticeable for some exotic uses.

CHIPVER

The Spin1 CHIPVER command always evaluates as 1, regardless of the actual chip version stored at $FFFF.

Symbol Differences

Local variables may shadow method names

In FlexSpin local variables may have the same name as methods in the object. In the original Spin and Spin2 compilers this will produce an error. The FlexSpin way is more "traditional" (matches how most compilers/languages work).

Special characters in identifiers

Special characters like `$` may be included in Spin function and variable names by preceding them with a backquote, e.g. to define a function `chr$` do:

```
pub chr`$(x)
  return x
```

Opcodes

In regular Spin opcodes like `TEST` are always reserved.

In flexspin, opcodes are only reserved inside `DAT` sections, so it is legal to have a function or variable named `test` or `add`.

Reserved words

flexspin adds some reserved words: `asm`, `endasm`, and `then`. Programs which use these reserved words may not work correctly in flexspin, although there has been some effort made to make them work as regular identifiers in many contexts.

Strings

Literal strings like `"hello"` are treated as lists of characters in regular Spin, unless enclosed in a `STRING` declaraction. So for example:

```
foo("ABC")
```

is parsed as

```
foo("A","B","C")
```

whereas in flexspin it is treated as an array of characters, so it is parsed like:

```
foo("ABC"[0])
```

which will be the same as

```
foo("A")
```

The difference is rarely noticeable, because flexspin does convert string literals to lists in many places.

As an extension, flexspin allows you to write:

```
foo(@"ABC")
```

instead of

```
foo(string("ABC"))
```

Known Bugs

Type narrowing in assignment

A series of assignments like:

```
a := b := 512
```

is evaluated by flexspin as

```
b := 512
a := b
```

This is sometimes different from how the official Spin1 interpreter does it. In particular, if `b` is a byte or word then the result of `b:=512` is truncated to 8 or 16 bits, whereas in the official interpreter the full 32 bit result is assigned to `a`. Thus, if `b` is a byte then flexspin sets `a` to `0` whereas Spin1 sets `a` to `512`.

There are some work-arounds to reduce the impact of this, so it probably will only be noticed in edge cases.

P2 Considerations

Spin1 on P2

Many Spin1 programs may be ported from the Propeller 1 to the Propeller 2, but there are some important exceptions:

- PASM must be translated to P2ASM. The assembly language for the Propeller 1 and Propeller 2 are different; there are some similarities, but in general any assembly language code will have to be translated to work on the different processor.
- WAITPEQ, WAITPNE, and WAITVID are not implemented on P2
- The hardware register set is different; the P2 does not have the CTRx, FRQx, PHSx, VCFG, or VSCL registers.

Compatibility with Spin2

flexspin is mostly, but not completely, compatible with the standard Spin2 compiler. Not all Spin2 builtin functions are available on the P1; only the ones listed in the "New intrinsics for both P1 and P2" are available on all platforms. But when compiling for P2 all of the Spin2 builtin functions should be available.

Expressions involving :

The `:` character is used in multiple conflicting ways in Spin2:

```
a ? b : 2
x := b():1
x := lookup(b : 1, 2)
```

The flexspin parser works differently from the PNut parser, and in complicated nested expressions with multiple `:` it may not give the same results. Some tips for writing code that will work with both compilers:

- In general try not to nest expressions involving `:`

- Do not use `:` in `lookup/lookdown` type expressions except as the separator between the selection expression and the list of results
- Make liberal use of parentheses to resolve ambiguity, e.g. write `a ? (b()) : 1` or `a ? (b():1) : 2` to distinguish between `:1` as an indirect method marker nad `:1` as the "else" part of the `?` operator.

@ Operator

The `@` operator always gives an absolute address in flexspin's Spin2 dialect, even inside assembly code. This is different from the standard Spin2 interpreter, where it produces an address relative to the start of the current object. In most contexts the flexspin behavior is more convenient, but it is something to keep in mind.

If you really need a relative offset, declare a label like `entry` at the start of your assembly and use `@label - @entry` to find the offset of `label` from `entry`. This will work in all compilers.

ORG/END

No address may be given in an ORG/END pair. If no FCACHE is available (e.g. -O0 is given) then the code is run as hubexec, in which case no self-modifying code or local data is permitted.

The space available for use in ORG/END inline assembly is smaller by default in flexspin (128 longs) than in PNut / Proptool (304 longs). You can change the flexspin value with the `--fcache=` flag, e.g. `--fcache=304`, but beware that this may cause the COG memory to overflow if the program uses a lot of local variables or pointers.

Memory map

The location of the clock frequency is at the standard location $14 used by TAQOZ, micropython, and most C compilers, rather than $44 as used by Spin2; similarly, clock mode is at $18 instead of $48.

COG memory is also laid out differently. See the general compiler documentation for details of the memory map.

DEBUG statements

In flexspin, `DEBUG` statements are accepted only in Spin2 methods, *not* in PASM (they are ignored in PASM). This restriction is relaxed if the P2 only `-gbrk` flag is used instead of plain `-g`.

Debug statements are output only when some variant of the `-g` flag (e.g. `-g` or `-gbrk` is given to flexspin.

Only a subset of the Spin2 `DEBUG` directives are accepted in normal `-g` mode:

`ZSTR, UDEC, UDEC_BYTE, UDEC_WORD, UDEC_LONG, SDEC, SDEC_BYTE, SDEC_WORD, SDEC_LONG, UHEX, UHEX_BY`

Other debug directives are ignored, with a warning.

With plain `-g` debugging `DEBUG` in flexspin is implemented differently than in PNut, so timing when debug is enabled may be different.

`DEBUG` statements containing backticks are (partially) translated so as to output the correct strings, but FlexProp only interprets a limited subset of these strings so graphical debug capabiliites are restricted in FlexProp.

Thanks to Ada Gottensträter, flexspin also now supports a `-gbrk` flag to enable `DEBUG` using the standard PNut method (using a BRK) instruction. This method will work inside PASM code, and is generally more compatible with the standard PNut Spin2 code.

ASMCLK instruction

The `ASMCLK` pseudo-instruction is supported as a preprocessor macro in FlexSpin, so only the most common spellings like `ASMCLK`, `AsmClk`, and `asmclk` will work (e.g. `aSmClk` will not work).

REGLOAD/REGEXEC

The `REGLOAD` and `REGEXEC` Spin2 instructions are not supported at this time, mainly because they depend on a particular layout of memory in the Spin2 interpreter.

Detecting FlexSpin versus PNut

FlexSpin understands a special comments that start with `{$flexspin` and treats these as regular code. PNut will ignore these comments, so this provides a way to mix FlexSpin specific code and PNut specific code. See the preprocessor section above ("Special Preprocessor Comments") for further details and an example.