# FlexProp Reference

5.9.25-beta

Total Spectrum Software

01/13/2023

# Contents

# General Compiler Features

This document describes compiler features common to all of the Flex languages.

## Languages

The compiler supports Spin (in both Spin 1 and Spin 2 variants), C (and a subset of C++), and BASIC. The choice of which language to use is determined by the file extension.

### Spin OBJ blocks

In Spin OBJ blocks, the default extension for file names is the same as the file name of the source file; that is:

```
OBJ
  a: "somedevice"
```

will look for "somedevice.spin2" first in a .spin2 file, then "somedevice.spin"; in a .spin file the order is reversed.

In order to avoid ambiguity it is suggested that the file name extension always be explicitly given. An explicit file name is always required for non-Spin objects:

```
OBJ
  a: "somedevice.spin"
  b: "otherdevice.bas"
  c: "otherthing.c"
```

### Objects in other languages

In BASIC `class using` and C `struct __using` the full filename of the source object, including extension, must always be used.

BASIC and C also allow inline declarations of classes, using `class`. See the respective language documents for more details.

### Calling cross language

Every language may call functions written in the other languages. Spin and BASIC are case-insensitive, but C is case sensitive. This means that even in a Spin or BASIC program, you must use the proper case in order to access C functions or variables.

## Fast Cache (Fcache)

Fcache is a special feature of the compiler whereby small loops are copied from HUB memory into local (COG) memory before execution. This speeds up repated loops quite a bit. Fcache is available only if optimization is enabled.

Some inline assembly blocks may also be marked to be copied to fcache before execution; see the section on inline assembly for a description of this.

### What loops will be placed in fcache

Loops will be placed in fcache only if (a) they will fit, and (b) they contain no branches to outside the loop (including subroutine calls). The size of the fcache may be set by the `--fcache` flag, but is generally 1024 bytes on P2 and 128 bytes on P1.

## Inline assembly

All of the languages allow inline assembly within functions. There are 3 different forms of inline assembly:

(1) Plain inline assembly. This is generated by `asm`/`endasm` in Spin, `asm`/`end asm` in BASIC, and from an `__asm { }` block in C. These blocks run in hubexec mode (for P2) or LMM (for P1) and are optimized by the optimizer.

(2) HUB non-optimized assembly. This is generated by `orgh` / `end`in Spin, `const asm`/`end asm` in BASIC and `__asm const{}` in C. Like plain assembly this runs from HUB, but is not subject to optimization.

(3) FCACHEd non-optimized assembly. This is generated by `org`/`end` in Spin, `cpu asm`/`end asm` in BASIC, and `__asm volatile{}` in C. This is not subject to optimization, and before execution it is loaded into the FCACHE area, so its timing is based on running from internal memory rather than HUB.

## Restrictions in inline assembly

Inline assembly within a function follows a different path through the compiler than "regular" assembly in a DAT section (Spin) or `shared asm` (BASIC). This has a number of consequences; not all constructions will work properly, and the inline assembly can be limited.

### Only local variables

Only hardware registers and variables local to the function may be used in inline assembly. Global or method variables may not be referenced.

### 64 bit variables

If a local variable needs more than 4 bytes (e.g. 64 bit integers) then the longs of that variable may be referenced by `x+0`, `x+1`, and so on. For a 64 bit variable, `x+0` is the low long, and `x+1` is the high one.

### Local variables not usable in some functions

If a function takes the address of a parameter, or of a local variable, then its local variables are placed on the stack and may not be referred to in inline assembly.

### No branches outside the function

Branching within a function is supported in inline assembly, but trying to branch outside the function or to call another function is not supported. The results are undefined; calls in particular may appear to work in some cases, but then fail when the called function is modified.

It is also not legal to return from inside inline assembly. For Spin2, returns are automatically converted to jumps to the end of the inline assembly (this is for compatibility with PNut), but it's probably better to write this explicitly yourself.

### No register declarations

Do not try to declare registers; the inline assembly probably will not be running from COG memory. If you need some scratch registers in inline assembly, declare them as local variables in the function. The only exception to this is `ORG` in Spin2, or the similar constructs like `cpu asm` or `asm volatile` in C, where the inline assembly is guaranteed to be placed in COG memory. It's still probably

better to declare local variables instead of placing register declarations in the inline assembly, though.

#### General Guidelines

Try to keep inline assembly as simple as possible. Use the high level language for loops and conditional control structures; the high level language is there for a reason!

### Inline assembly in bytecode

Inline assembly is not supported in Spin1 bytecode (`--interp=rom`). It will produce a compile time error. It is supported in Nu bytecode (`-2nu`) but hasn't been thoroughly tested in that mode.

## Functions in COG or LUT memory

Normally functions are placed in HUB memory, because there is a lot more of that. However, it is possible to force some functions to be placed in the chip's internal memory, where they will execute much more quickly. This must be done with care, because internal memory is a very limited resource.

Only small functions should be placed in internal memory. It's probably better if these functions do not call any functions in HUB, at least in the normal case, since any speed advantage of placing a function in COG or LUT will be negated if it calls out to HUB.

Functions in COG or LUT memory are not supported in bytecode output (e.g. `--interp=rom`).

### Spin/Spin2

To put a method into COG memory, place a special comment `{++cog}` after the `PUB` or `PRI` declaration of the method.

```
pub {++cog} add(x, y)
  return x+y
```

Similarly, to put the method into LUT memory (on the P2 only, obviously) then use the comment `{++lut}`.

### BASIC

Place the keyword `for` before the function or subroutine's name in its declaration, followed by a string specifying the memory (`"cog"` or `"lut"`):

```
function for "cog" toupper(c as ubyte) as ubyte
  if c >= asc("a") and c <= asc("z") then
    c = c + (asc("A") - asc("a"))
  end if
  return c
end function
```

### C/C++

Place `__attribute__((cog))` after the function declaration but before its body:

```
int add(int x, int y) __attribute__((cog))
{
  return x+y;
}
```

Similarly use `__attribute__((lut))` to place the function into LUT memory.

## Interrupt Service Routines

The code generated by FlexSpin is *not* interrupt safe, so it is not possible to run interrupt service routines (ISRs) in the same COG as compiled code. ISRs may, of course, run in COGs that contain only PASM code launched by `coginit`.

# ABORT / TRY / CATCH

All of the language supported by FlexSpin support some form of exception throwing or catching.

### Spin and Spin2

Spin and Spin2 support only a subset of the functionality, in the form of the `ABORT` keyword and the `\` operator.

### BASIC and C

BASIC and C support `try` / `catch` which may execute arbitrary code when an exception is thrown. The block inside `try` is executed; if inside that block an error is thrown, the value is passed to the `catch` clause. If a Spin function called from BASIC or C executes `ABORT n`, the effect is as if it did a `throw n` (where `n` is an integer).

In C the try block is marked with `__try` and the catch block with `__catch`:

```
__try {
  x = someFunc();
  y = someOtherFunc();
}
__catch(int e) {
  printf("error number %d was thrown\n", e);
}
```

In BASIC and C++ the normal `try` and `catch` keywords are supported.

For now only integer types are supported in `catch` clauses, and each `try` block should have only one `catch`. This restriction may be lifted at some time in the future to support typed `catch/throw`.

## Register Usage

These comments on register usage apply only to the default assembler output. Bytecode output (such as the Spin1 bytecode output enabled by `--interp=rom`) generally use all of the COG memory for the interpreter.

### P1

Pretty much all of COG RAM is used by the compiler. No specific hardware registers are used.

### P2

Most of COG RAM is used by the compiler, except that $1e0-$1ef are left free for application use. COG RAM from $00 to $ff is used for FCACHE, and so when you are sure no FCACHE is in use you may use this for scratch.

The first 16 registers of LUT memory ($200 to $20f) are always left free. The remainder of the first half of LUT ($210 to $2ff) are used for functions that the

user explicitly asks to put in LUT. If no such functions exist they are free for user use.

The second half of LUT memory (from $300 to $3ff) may be used by compiler internal functions, and so should not be used by assembly code.

`ptra` is used for the stack pointer. Applications should avoid using it.

`pa` is used internally for fcache loading. Applications may use it as a temporary variable, but be aware that any code execution which may trigger an fcache load (e.g. any loop or subroutine call) may trash its value.

# Memory Map

## HUB

Code starts at 0 in HUB (by default, there are command line options to change this). Data starts after the code. The heap is part of the data area. The stack starts after this and grows upwards.

## COG

Most of COG RAM is used by the compiler, except that $1e0-$1ef is left free for application use.

## LUT

The first 16 registers of LUT memory (from $200 to $20f) is left free for use by user PASM code, e.g. for the streamer. The remainder of the first half of LUT memory (from $210 to $300) is used for any functions explicitly placed into LUT. The LUT memory from $300 to $400 (the second half of LUT) is used for internal purposes.

# Warnings

Listed below are warnings which may be enabled on the command line or on a per-function basis. One may also specify `-Wall` to enable all warnings.

### Per-function control of warnings

It is possible to enable or disable individual warnings in a function by using attributes. This doesn't work for all warnings; see the individual warnings for a description. For example, to disable warnings about uninitialized variables in a C function use:

```
int foo(int x) __attribute__(warn(!init-vars)) {
...
}
```

A similar effect is achieved in Spin by adding a comment `{++warn(!init-vars)}` between the `pub` or `pri` and the function name.

In BASIC we use the `for` keyword followed by a string giving the warning options:

```
function for "warn(!init-vars)" myfunc()
```

Multiple warnings may be given, separated by commas. To turn a warning off, prefix it with `!` or with `~`. To enable all warnings, use the word `all`.

Thus, a Spin function with `{++opt(!all,hide-members)}` will always be compiled with no warnings except for uninitialzed variables.

### Warning control on the command line

Multiple `-W` options may be given, or combined separated by commas. So for example to compile with all warnings except uninitialized variables, one would give `-Wall,!init-vars`.

### Assembler usage warnings (-Wasm-usage) (enabled by default)

Warns about some common issues in assembly code, for example forgetting to put `wc` or `wz` on a `cmp` instruction. This option is only available on the command line, it is generally ignored on a per-function basis (even for functions with inline assembly) because assembly parsing is handled specially.

### C constant strings (-Wc-const-strings)

If enabled, all string literals in C are treated as being `const`. This is a useful warning (because it is not legal to modify a string literal) but many older programs do not use `const` consistently enough to prevent this warning.

### Hidden members (-Whide-members)

If enabled, any function local variables which shadow class or object members are warned about.

### Language extensions (-Wlanguage-extensions)

Warns about various FlexProp specific extensions to the Spin, Spin2, and C languages. Note that some of these extensions are detected very early in the parsing process, before functions are recognized, and so it probably isn't useful to enable/disable this warning on a per-function basis.

### Uninitialized variables (-Winit-vars)

Issues a warning about attempts to use uninitialized variables. Note that FlexProp isn't completely able to see all ways a variable could be initialized, so this warning may sometimes be spurious.

## Optimizations

Listed below are optimizations which may be enabled on the command line or on a per-function basis. The general optimization level may be specified by a number: 0 for no optimizations, 1 for basic (reliable) optimizations, and 2 for additional optimizations. `-Os` is generally equivalent to `-O1`, but favors size over speed (and may enable a few additional space optimizations).

### Per-function control of optimizations

It is possible to enable or disable individual optimizations in a function by using attributes. For example, to disable loop reduction for a particular C function, one would add an attribute:

```
int foo(int x) __attribute__(opt(!loop-reduce)) {
...
}
```

A similar effect is achieved in Spin by adding a comment `{++opt(!loop-reduce)}` between the `pub` or `pri` and the function name.

In BASIC we use the `for` keyword followed by a string giving the optimization options:

```
function for "opt(!loop-reduce)" myfunc()
```

Multiple options may be given, separated by commas. To turn an option off, prefix it with `!` or with `~`. To enable all options for a particular optimization level, start the string with `0`, `1`, `2`, etc., or with the word `all` to enable all optimizations (regardless of the compiler optimization level chosen).

Thus, a Spin function with `{++opt(0,peephole)}` will always be compiled with no optimization except peepholes, even when the `-O2` option is given to the compiler.

### Optimization control on the command line

Multiple `-O` options may be given, or combined separated by commas. So for example to compile with no optimizations except basic register and peephole, one would give `-O0,regs,peephole`. To compile with `-O2` but with peepholes turned off, one would give `-O2,!peephole`.

### Optimizing for size

The `-Os` option enables all of the optimizations specified by `-O1`, plus some size related optimizations.

### Multiplication conversion (always)

Multiplies by powers of two, or numbers near a power of two, are converted to shifts. For example

```
a := a*10
```

is converted to

```
a := (a<<3) + (a<<1)
```

A similar optimization is performed for divisions by powers of two.

### Unused method removal (-O1, -Oremove-unused)

This is pretty standard; if a method is not used, no code is emitted for it. This optimization works at high level, so enabling/disabling it per function is not useful.

### Unused feature removal (-O1, -Oremove-features)

Scans the program for certain library features (like file I/O and floating point usage) and disables those features if unused. This can save quite a bit of space. Depends on unused method removal to work properly/

### Dead code elimination (-O1, -Oremove-dead)

Within functions if code can obviously never be reached it is also removed. So for instance in something like:

```
CON
  pin = 1
...
if (pin == 2)
  foo
```

The if statement and call to `foo` are removed since the condition is always false.

### Small Method inlining (-O1, -Oinline-small)

Very small methods are expanded inline. This may be prevented by declaring the method with the "noinline" attribute.

### Register optimization (-O1, -Oregs)

The compiler analyzes assignments to registers and attempts to minimize the number of moves (and temporary registers) required.

### Local register reuse (-O1, -Olocal-reuse)

Reuse registers to reduce the number of temporary registers introduced. This may reduce the readability of the generated code somewhat, although that's probably moot at this stage because many other optimizations also make the code harder to read.

### Branch elimination (-O1, -Obranch-convert)

Short branch sequences are converted to conditional execution where possible.

### Constant propagation (-O1, -Oconst)

If a register is known to contain a constant, arithmetic on that register can often be replaced with move of another constant.

### Peephole optimization (-O1, -Opeephole)

In generated assembly code, various shorter combinations of instructions can sometimes be substituted for longer combinations.

### Tail call optimization (-O1, -Otail-calls)

Convert recursive calls into jumps when possible

### Loop optimization (-O1, -Oloop-basic)

In some circumstances the optimizer can re-arrange counting loops so that the `djnz` instruction may be used instead of a combination of add/sub, compare, and branch. In -O2 a more thorough loop analysis makes this possible in more cases.

### Fcache (-O1, -Ofcache)

Small loops are copied to internal memory (COG) to be executed there. These loops cannot have any non-inlined calls in them.

### Bytecode Macro creation (-O1, -Obcmacros)

Some bytecode backends (e.g. the nucode one) are able to combine bytecodes to create macros. This saves considerable space in the generated code. Note that this option applies globally, and cannot be turned on/off per function.

### Special Function handling (-O1, -Ospecial-functions)

Optimizes some specialized function calls for common cases. For example, in the general case pinwrite() has to handle being able to set ranges of bits, but in a special case where just one bit is being written it can be optimized to a _drvw(). Functions processed with this include:

pinread: Optimized if just one pin is read pinwrite: Optimized if just one pin or one bit are being written

## Reorder instructions for Cordic (-O1, -Ocordic-reorder)

On the P2, reorder instructions to take advantage of cordic pipeline delays.

## Aggressive load/store optimization (-O2, -Oaggressive-mem)

Enables some more aggressive optimizations which attempt to track values and reduce the number of memory accesses.

## Experimental / new optimizations (-O2, -Oexperimental)

Enables some miscellaneous optimizations that are new and hence slightly less well tested. Generally these should be pretty safe, but they're not quite ready for promotion to the default -O1.

## Single Use Method inlining (-O2, -Oinline-single)

If a method is called only once in a whole program, it is expanded inline at the call site.

## Common Subexpression Elimination (-O2, -Ocse)

Code like:

```
c := a*a + a*a
```

is automaticaly converted to something like:

```
tmp := a*a
c := tmp + tmp
```

## Loop Strength Reduction (-O2, -Oloop-reduce)

### Array indexes

Array lookups inside loops are converted to pointers. So:

```
repeat i from 0 to n-1
   a[i] := b[i]
```

is converted to the equivalent of

```
aptr := @a[0]
bptr := @b[0]
repeat n
  long[aptr] := long[bptr]
  aptr += 4
  bptr += 4
```

#### Multiply to addition

An expression like `(i*100)` where `i` is a loop index can be converted to something like `itmp \ itmp + 100`

## Memory Allocation and 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.

## Heap size specification

In SPIN:

```
CON HEAPSIZE=32768
```

In BASIC:

```
const HEAPSIZE=32768
```

In C:

```
enum { HEAPSIZE=32768 };
```

The C version is a little unexpected; one would expect HEAPSIZE to be declared as `const int` or with `#define`. This is a technical limitation that I hope to fix someday.

## Garbage collection

The garbage collector works by scanning memory for tagged pointers into the heap area; these pointers are assumed to be in use, whereas memory in the heap that is allocated but has no pointers to it in RAM are assumed to be garbage and are automatically freed during collection.

Garbage collection happens automatically when the `_gc_alloc()` function finds there is not enough memory to fulfil a request. This can take quite a bit of time, so it is best to avoid memory allocation requests (e.g. BASIC string operations) inside time critical code.

It is also possible to manually trigger a garbage collection request by calling `_gc_collect()`. After this as much memory as possible is freed.

### 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!)

## Terminal Control

The FlexProp system uses the default system terminal, configured when possible to accept VT100 (ANSI) escape sequences. The default serial pins (63 and 62 on P2, 31 and 30 on P1) are used.

### Changing baud rate

All languages have a `_setbaud(N)` function to set the baud rate to `N`.

### Sending and receiving characters

All languages support some functions for doing basic serial I/O. `_tx(ch)` sends a character to the serial port, with appropriate carriage return mapping (see below for how this is changed). `_txraw(ch)` sends the character with no interpretation. `_rx()` reads a character from the serial port, waiting until one is available. `_rxraw(tim)` attempts to read a character, waiting for up to `tim` milliseconds (actually 1/1024ths of a second) for one to be available; if no character is received before the timeout, returns -1.

### Changing echo and CR/LF interpretation

Normally input (e.g. from a C `getchar()`) is echoed back to the screen, and carriage return (ASCII 13) is converted to line feed (ASCII 10). Both of these behaviors may be changed via the `_setrxtxflags(mode)` function. The bits in `mode` control terminal behavior: if `mode & 1` is true, then characters are echoed, and if `mode & 2` is true then carriage return (CR) is converted to line feed (LF) on input, and line feed is converted to CR + LF on output.

The current state of the flags may be retrieved via `_getrxtxflags()`.

# Character Set

The input character set for the compiler is assumed to be UTF-8, or Windows UCS2 (16 bit Unicode).

The character set to use at runtime is set by the compiler switch `--charset=C`, and may be one of `utf8` (the default), `latin1` (for ISO_8859-1), `shiftjis` (for Shift-JIS 2004), or `parallax` (for the Parallax font). For example, if you are using a VGA graphics program with the Parallax font, you would typically use `--charset=parallax`. This would cause any strings in the program to be translated from UTF-8 (the input character set) to the Parallax font encoding.

# File I/O (P2 Only)

C and BASIC have built in support for accessing file systems. The file systems first must be given a name with the `mount` system call, and then may be accessed with the normal language functions.

## Mount

The `mount` call gives a name to a file system. For example, after

```
mount("/host", _vfs_open_host());
mount("/sd", _vfs_open_sdcard());
```

files on the host PC may be accessed via names like "/host/foo.txt", "/host/bar/bar.txt", and so on, and files on the SD card may be accessed by names like "/sd/root.txt", "/sd/subdir/file.txt", and so on.

This only works on P2, because it requires a lot of HUB memory. Also, the host file server requires features built in to `loadp2`.

Available file systems are:

- `_vfs_open_host()` (for the loadp2 Plan 9 file system)
- `_vfs_open_sdcard()` for a FAT file system on the P2 SD card (using default pins 58-61)
- `_vfs_open_sdcardx()` for a FAT file system on SD card using custom pins

It is OK to make multiple mount calls, but they should have different names.

### Options for SD Card

If you define the symbol `FF_USE_LFN` on the command line with an option like `-DFF_USE_LFN` then long file names will be enabled for the SD card.

The pins to use for the SD card may be changed by using `_vfs_open_sdcardx` instead of `_vfs_open_sdcard`. The parameters for `_vfs_open_sdcardx` are the clock pin, select pin, data in, and data out pins, in that order. Thus, `_vfs_open_sdcard` is actually equivalent to `_vfs_open_sdcardx(61, 60, 59, 58)`.

## Command Line Options

### Options for flexspin

There are various command line options for the compiler which may modify the compilation:

```
[ --version ]      print just the compiler version, then exit
[ -1 ]             compile for Prop1 LMM (the default)
[ -1bc ]           compile for Prop1 ROM bytecode
[ -2 ]             compile for Prop2 assembly
[ -2nu ]           compile for Prop2 bytecode
[ -h ]             display this help
[ -L or -I <path> ] add a directory to the include path
[ -o name ]        set output filename
[ -b ]             output binary file format
[ -e ]             output eeprom file format
[ -c ]             output only DAT sections
[ -l ]             output a .lst listing file
[ -f ]             output list of file names
[ -g ]             enable debug statements (default printf method)
[ -gbrk ]          enable BRK based debugging
[ -q ]             quiet mode (suppress banner and non-error text)
[ -p ]             disable the preprocessor
[ -O[#] ]          set optimization level
                     -O0 disable all optimization
                     -O1 apply default optimization (same as no -O flag)
             -O2 apply all optimization (same as -O)
[ -Wall ]          enable all warnings, including warnings about language extensions
[ -Werror ]        turn warnings into errors
[ -Wabs-paths ]    print absolute paths for file names in errors/warnings
[ -Wmax-errors=N ] allow at most N errors in a pass before stopping
[ -D <define> ]    add a define
```

```
[ -2 ]             compile for Prop2
[ -w ]             produce Spin wrappers for PASM code
[ -H nnnn ]        change the base HUB address (see below)
[ -E ]             omit any coginit header
[ --charset=C ]    set the character set to use at runtime
         C = utf8 for UTF-8 encoding (the default)
     C = latin1 for Latin-1 encoding
     C = parallax for Parallax font encoding
[ --code=cog  ]    compile to run in COG memory instead of HUB
[ --fcache=N  ]    set size of FCACHE space in longs (0 to disable)
[ --fixedreal ]    use 16.16 fixed point instead of IEEE floating point
[ --lmm=xxx ]      use alternate LMM implementation for P1
         xxx = orig uses original flexspin LMM
         xxx = slow uses traditional (slow) LMM
[ --nostdlib ]     do not check for include files in the standard place (../include relative to
[ --tabs=N ]       specifify number of spaces between tab stops (default 8)
[ --zip ]          create a zip file containing the source inputs
```

`flexspin.exe` checks the name it was invoked by. If the name starts with the string "bstc" (case matters) then its output messages mimic that of the bstc compiler; otherwise it tries to match openspin's messages. This is for compatibility with Propeller IDE. For example, you can use flexspin with the PropellerIDE by renaming `bstc.exe` to `bstc.orig.exe` and then copying `flexspin.exe` to `bstc.exe`.

## Options for flexcc

`flexcc` is similar to `flexspin`, but has arguments more like the traditional `cc` command line compiler.

```
[ --help ]         display this help
[ -c ]             output only .o file
[ -D <define> ]    add a define
[ -g ]             include debug info in output
[ -L or -I <path> ] add a directory to the include path
[ -o <name> ]      set output filename to <name>
[ -2 ]             compile for Prop2
[ -O# ]            set optimization level:
        -O0 = no optimization
        -O1 = basic optimization
        -O2 = all optimization
[ -Wall ]          enable warnings for language extensions and other features
[ -Werror ]        make warnings into errors
[ -Wabs-paths ]    print absolute paths for file names in errors/warnings
```

```
[ -Wmax-errors=N ] allow at most N errors in a pass before stopping
[ -x ]             capture program exit code (for testing)
[ --code=cog ]     compile for COG mode instead of LMM
[ --fcache=N ]     set FCACHE size to N (0 to disable)
[ --fixedreal ]    use 16.16 fixed point in place of floats
[ --lmm=xxx ]      use alternate LMM implementation for P1
         xxx = orig uses original flexspin LMM
         xxx = slow uses traditional (slow) LMM
[ --nostdlib ]     Do not check for include files in the standard place (../include
[ --version ]      just show compiler version
```

### Changing Hub address

In P2 mode, you may want to change the base hub address for the binary. Normally P2 binaries start at the standard offset of `0x400`, with memory from `0` to `0x400` being used by the initial startup code. But if you want, for example, to load a flexspin compiled program from TAQOZ or some similar program, you may want to start at a different address (TAQOZ uses the first 64K of RAM). To do this, you may use some combination of the `-H` and `-E` flags.

`-H nnnn` changes the base HUB address from `0x400` to `nnnn`, where `nnnn` is either a decimal number like `65536` or a hex number prefixed with `0x`. By default the binary still expects to be loaded at address 0, so it starts with a `coginit #0, ##nnnn` instruction and then zero padding until the hub start. To skip the `coginit` and padding, add the `-E` flag.

#### Example

To compile a program to start at address 65536 (at the 64K boundary), do:

```
flexspin -2 -H 0x10000 -E fibo.bas
```

## Common low level functions

A number of low level functions are available in all languages. The C prototypes are given below, but they may be called from any language and are always available. If a user function with the same name is provided, the built-in function will not be available from user code (but internally the libraries *may* continue to use the built-in version; this isn't defined).

Unless otherwise noted, these functions are available for both P1 and P2.

## Serial port access

### __txraw

```
int _txraw(int c)
```

sends character `c` out the default serial port. Always returns 1.

### __rxraw

```
int _rxraw(int n=0)
```

Receives a character on the default serial port. `n` is a timeout in milliseconds. If the timeout elapses with no character received, `_rxraw` returns -1, otherwise it returns the received character. The default timeout (0) signifies "forever"; that is, if `n` is 0 the `_rxraw` function will wait as long as necessary until a character is received.

### __setbaud

```
void _setbaud(int rate)
```

Sets the baud rate on the default serial port to `rate`. For example, to change the serial port to 115200 baud you would call `_setbaud(115200)`. The default rates set up in C and BASIC initialization code are are 115200 for P1 and 230400 for P2. In Spin you may need to call `_setbaud` explicitly before calling `_rxraw` or `_txraw`.

## Time related functions

### __getsec

Gets elapsed seconds since the system was booted. Uses the system clock, which wraps around after about 50 seconds on the P1.

### __getms

Gets elapsed milliseconds since boot.

### __getus

Gets elapsed microseconds since boot.

**__waitx**

```
void _waitx(unsigned cycles)
```

Pauses for `cycles` cycles. Note that the maximum waiting period is about half of the system clock frequency.

**__waitms**

```
void _waitms(unsigned ms)
```

Wait for `ms` milliseconds. For waits of more than a second, this function will loop internally so as to avoid limits of the 32 bit clock counter.

**__waitus**

```
void _waitus(unsigned us)
```

Wait for `us` microseconds. For waits of more than a second, this function will loop internally so as to avoid limits of the 32 bit clock counter.

## Cog control

**__cogchk**

```
int _cogchk(int id)
```

Checks to see if cog number id is running. Returns -1 if running, 0 if not.

This function may be relatively slow on P1, as it has to manually probe the COGs (on P2 it's a built in instruction).